HeavyKeeper: An Accurate Algorithm for Finding Top-*k* Elephant Flows

Junzhi Gong, Tong Yang, Haowei Zhang, and Hao Li, *Peking University;*
Steve Uhlig, *Queen Mary, University of London;* Shigang Chen, *University of Florida;*
Lorna Uden, *Staffordshire University;* Xiaoming Li, *Peking University*

https://www.usenix.org/conference/atc18/presentation/gong

This paper is included in the Proceedings of the
2018 USENIX Annual Technical Conference (USENIX ATC '18).
July 11–13, 2018 • Boston, MA, USA

ISBN 978-1-939133-02-1

HeavyKeeper: An Accurate Algorithm for Finding Top-*k* Elephant Flows

Junzhi Gong
Peking University

Tong Yang*
Peking University

Haowei Zhang
Peking University

Hao Li
Peking University

Steve Uhlig
UK & Queen Mary, University of London

Shigang Chen
University of Florida

Lorna Uden
Staffordshire University

Xiaoming Li
Peking University

Abstract

[1] Finding top-*k* elephant flows is a critical task in network traffic measurement, with many applications in congestion control, anomaly detection and traffic engineering. As the line rates keep increasing in today's networks, designing accurate and fast algorithms for online identification of elephant flows becomes more and more challenging. The prior algorithms are seriously limited in achieving accuracy under the constraints of heavy traffic and small on-chip memory in use. We observe that the basic strategies adopted by these algorithms either require significant space overhead to measure the sizes of all flows or incur significant inaccuracy when deciding which flows to keep track of. In this paper, we adopt a new strategy, called *count-with-exponential-decay*, to achieve space-accuracy balance by actively removing small flows through decaying, while minimizing the impact on large flows, so as to achieve high precision in finding top-*k* elephant flows. Moreover, the proposed algorithm called HeavyKeeper incurs small, constant processing overhead per packet and thus supports high line rates. Experimental results show that HeavyKeeper algorithm achieves 99.99% precision with a small memory size, and reduces the error by around 3 orders of magnitude on average compared to the state-of-the-art.

1 Introduction

1.1 Background and Motivation

Finding the largest *k* flows, also referred to as the top-*k* elephant flows, is a fundamental network management function, where a flow's ID is usually defined as a combination of certain packet header fields, such as source IP address, destination IP address, source port, destination port, and protocol type, and the size of a flow is defined as the number of packets of the flow. Elephant flows contribute a large portion of network traffic. Many management applications can benefit from a function that can find them efficiently, such as congestion control by dynamically scheduling elephant flows [1], network capacity planning [2], anomaly detection [3], and caching of forwarding table entries [4]. Such a function also has applications beyond networking in areas such as data mining [5–7], information retrieval [8], databases [9], and security [10, 11].

In real network traffic, it is well known that the distribution of flow sizes (the number of packets in a flow), is highly skewed [12–21], *i.e.*, the majority are mouse flows, while the minority are elephant flows. most flows are small while a few flows are very large. The small flows are usually called *mouse flows*, while the large ones are called *elephant flows*.

Finding the top-*k* elephant flows (or top-*k* flows for short) in high-speed networks is a challenging task. [22] Extremely high line rates of modern networks make it practically impossible to accurately track the information of all flows. Consequently, approximate methods have been proposed in the literature and gained wide acceptance [14, 23–27]. In order to keep up with the line rates, these algorithms are expected to use on-chip memory such as SRAM whose latency is around 1ns [28, 29], in contrast to a latency of around 50ns when off-chip DRAM is used [29]. However, on-chip memory is small. Adding to the challenge, it is highly desirable to keep per-packet processing overhead small and constant, which helps pipelining.

Traditional solutions to finding the top-*k* flows follow two basic strategies: *count-all* and *admit-all-count-some*. The count-all strategy relies on a sketch (*e.g.*, CM sketch [14]) to measure the sizes of all flows, while us-

[1] Co-primary authors: Junzhi Gong and Tong Yang. Corresponding author: Tong Yang (yangtongemail@gmail.com). Junzhi Gong, Haowei Zhang, Hao Li finished this work under the guidance of their supervisor: Tong Yang. This work is supported by Primary Research & Development Plan of China (2016YFB1000304), National Basic Research Program of China (973 Program, 2014CB340405), NSFC (61672061), the OpenProject Funding of CAS Key Lab of Network Data Science and Technology, Institute of Computing Technology, Chinese Academy of Sciences.

ing a min-heap to keep track of the top-k flows. For each incoming packet, it records the packet in the sketch and retrieves from the sketch an estimate $\hat{n}_i$ for the size of the flow f_i that the packet belongs to. If $\hat{n}_i$ is larger than the smallest flow size in the min-heap, it replaces the smallest flow in the heap by flow f_i. As a large sketch is needed to count all flows, these solutions are not memory efficient.

The *admit-all-count-some* strategy is adopted by Frequent [30], Lossy Counting [26], Space-Saving [24] and CSS [23]. These algorithms are similar to each other. To save memory, Space-Saving only maintains a data structure called Stream-Summary to counts only some flows (m flows). Each new flow will be inserted into the summary, replacing the smallest existing flow. The initial size of the new flow is set as $\hat{n}_{min}+1$, where $\hat{n}_{min}$ is the size of the smallest flow in the summary. By keeping m flows in the summary, the algorithm will report the largest k flows among them, where $m > k$. It assumes every new incoming flow is an elephant, and expels the smallest one in the summary to make room for the new one. But most flows are mouse flows. Such an assumption causes significant error, especially under tight memory (for a limited value of m).

1.2 Our Proposed Solution

In this paper, we propose a new algorithm, HeavyKeeper, based on a different strategy, called *count-with-exponential-decay*, which keeps all elephant flows while drastically reducing space wasted on mouse flows. Unlike *count-all*, our strategy only keeps track of a small number of flows. Unlike *admit-all-count-some*, we do not automatically admit new flows into our data structure and the vast majority of mouse flows will be by-passed. For a small number of mouse flows that do enter our data structure, they will decay away to make room for true elephants. The decay is not uniform for the flows in our data structure. The design of exponential decay is biased against small flows, and it has a smaller impact on larger flows. This design works extremely well with real traffic traces under small memory where the previous strategies will fail.

Main experimental results: As shown in Table 1, when compared with *Space-Saving*, *Lossy counting*, *CSS*, and *CM sketch*, HeavyKeeper achieves 99.99% percent precision, and much smaller error than all of them.

Table 1: Main experimental results.
Precision is defined as the ratio between the number of correctly reported elephant flows and the total number of reported flows.

Algorithm	Top-k precision	Avg. relative error of flow sizes
Space-Saving [24]	0.27	172.7222
Lossy counting [26]	0.39	54.8440
CSS [23]	0.49	18.9356
CM sketch [14]	0.93	0.2951
HeavyKeeper	0.9999	0.0011

Contributions: This paper makes the following contributions.

1. We propose a new data structure, named HeavyKeeper, which achieves high precision for finding top-k flows, and achieves constant and fast speed as well as high memory efficiency.

2. We develop a mathematical analysis for HeavyKeeper, to theoretically prove its high precision.

3. We conduct extensive experiments on real network streams and synthetic datasets, and results show that HeavyKeeper reduces the error by around 3 orders of magnitude on average compared to the state-of-the-art.

4. We integrate HeavyKeeper and other related algorithms with Open vSwitch (OVS) platform. We also conduct experiments on throughput on OVS platform to show the impact of the algorithms. The results show that HeavyKeeper has little impact on the throughput, while other algorithms decrease the throughput significantly. We release the source code of HeavyKeeper and related algorithms at GitHub [31].

2 Preliminaries

2.1 Problem Statement

Simply speaking, finding top-k flows refers to finding the largest k flows. Let $\mathscr{P} = \mathbb{P}_1, \mathbb{P}_2, \cdots, \mathbb{P}_N$ be a network stream with N packets. Each packet $\mathbb{P}_l$ $(1 \leqslant l \leqslant N)$ belongs to a flow f_i, where $f_i \in \mathscr{F} = \{f_1, f_2, \cdots, f_M\}$ and $\mathscr{F}$ is the set of flows. Let n_i be the real flow size of flow f_i in $\mathscr{P}$. We order all flows $(f_1, f_2, \cdots, f_M)$ so that $n_1 \geqslant n_2 \geqslant \cdots \geqslant n_M$.

Given an integer k and a network stream $\mathscr{P}$, the output of top-k is a list of k flows from $\mathscr{F}$ with the largest flow sizes, *i.e.*, $f_1, f_2, \cdots, f_k$.

2.2 Prior Art and Limitations

The count-all strategy: As mentioned above, the *count-all* strategy uses sketches (such as the CM sketch [14] or the Count sketch [25]) to record the sizes of all flows, and uses a min-heap to keep track of the top-k flows, including the flow IDs and their flow sizes. Take the CM sketch as an example. It records packets in a CM sketch, consisting of a pool of counters. For each arrival packet, it hashes the packet's flow ID f to d counters and increases these d counters by one. The smallest value of the d counters is used as the estimated size of the flow.

If this estimated flow size is larger than the smallest flow size in the min-heap, we replace the smallest flow in the heap by flow f.

The problem is that all flows are pseudo-randomly mapped to the same pool of counters through hashing. Each counter may be shared by multiple flows, and thus record the sum of sizes of all these flows. Consequently, the CM sketch has an over-estimation problem, which will become severe in a tight memory space where the number of counters is far smaller than the number of flows, resulting in aggressive sharing. In such a case, a small flow may be treated as an elephant flow if all its d counters are shared with real elephant flows.

The admit-all-count-some strategy: As mentioned above, quite a few algorithms use the *admit-all-count-some* strategy, including Frequent [30], Lossy counting [26], and Space-Saving [24], with Space-Saving being the most widely used among them. Take Space-Saving as an example. Recognizing that it is infeasible to count the sizes of all flows, Space-Saving counts only the sizes of some flows in a data structure called Stream-Summary, which incurs $O(1)$ overhead to search or update a flow, or find the smallest flow. The selection of which flows to store in the summary is rather simple: For each arrival packet, if its flow ID is not in the summary, the flow will be admitted into the summary, replacing the smallest existing flow. The new flow's initial size is set to $\hat{n}_{min}+1$, where $\hat{n}_{min}$ is the smallest flow size in the summary before replacement. Therefore, later incoming mouse flows will be largely over-estimated, which is drastically inaccurate. In the end, the largest k flows in the summary will be reported. A recent work CSS [23] is proposed based on Space-Saving. It inherits the above strategy of Space-Saving, and redesigns the data structure of Stream-Summary by using TinyTable [32] to reduce memory usage.

The strategy of *admit-all-count-some* is to admit all new flows while expelling the smallest existing ones from the summary. To give new flows a chance to stay in the summary, their initial flow sizes are set as $\hat{n}_{min}+1$. Such a strategy drastically over-estimates sizes of flows, and we show an example here. Assume $\hat{n}_{min} = 10,000$. Given an new incoming flow, its size will be over-estimated as $10,001$. Early arrived elephant flows with flow sizes less than $10,008$ will be expelled. Therefore, massive mouse flows will cause significant over-estimation errors.

3 The Design of HeavyKeeper

In this section, we present the data structure and algorithm of our HeavyKeeper, and show how to find the top-k flows accurately and efficiently.

3.1 Rationale

We aim to use a small hash table to store all elephant flows. As there are a great number of flows, each bucket of the hash table will be mapped by many flows, and we aim to store only the largest flow with its size, which cannot be achieved with no error when using small memory. To address this problem, we propose a probabilistic method called *exponential-weakening decay*. Specifically, when the incoming flow is different from the flow in the hashed bucket, we decay the flow size with a decay probability, which is exponentially smaller as the flow size grows larger. If the flow size is decayed to 0, it replaces the original flow with the new flow. In this way, mouse flows can easily be decayed to 0, while elephant flows can easily keep stable in the bucket. There are two shortcomings: 1) With a small probability we elect the wrong flow as the largest flow; 2) The stored flow size is a little smaller than the true frequency because of the decay operations. To address these problems, we use multiple hash tables with different hash functions. An elephant flow could be stored in multiple hash tables, we choose the recorded largest size, minimizing the error of flow sizes.

3.2 The HeavyKeeper Structure

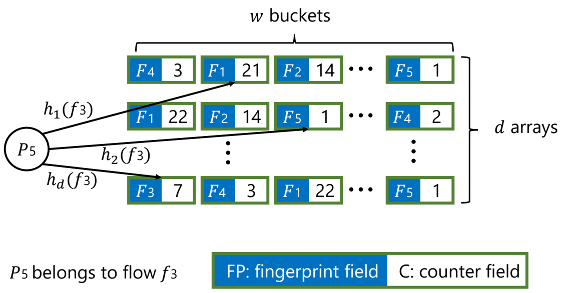

Figure 1: The data structure of HeavyKeeper.

As shown in Figure 1, HeavyKeeper is comprised of d arrays, and each array is comprised of w buckets. Each bucket consists of two fields: a fingerprint field and a counter field.[2] For convenience, we use $A_j[t]$ to represent the t^{th} bucket in the j^{th} array, and use $A_j[t].FP$ and $A_j[t].C$ to represent its fingerprint field and counter field, respectively. Arrays $A_1...A_d$ are associated with hash functions $h_1(.)...h_d(.)$, respectively. These d hash functions $h_1(.)...h_d(.)$ need to be pairwise independent.

Insertion: Initially, all fingerprint fields are *null*, and all counter fields are 0. For each incoming packet $\mathbb{P}_l$ belong-

[2] The fingerprint of a flow is a hash value generated by a certain function (for example, if we use $h_f(.)$ as the fingerprint hash function, the fingerprint of flow f_j is $h_f(f_j)$). Although there can be hash collisions among flows, the probability is quite small. For example, if we set the fingerprint size to 16 bits, and there are 10000 buckets in the array, the probability of fingerprint collisions is $1.52*10^{-3}$.

ing to flow f_i, HeavyKeeper computes the d hash functions, and maps f_i to d buckets $A_j[h_j(f_i)]$ $(1 \leqslant j \leqslant d)$ (one bucket in each array), which we call ***d* mapped buckets** for convenience. As shown in Figure 2, for each mapped bucket, HeavyKeeper applies different strategies for the following three cases:

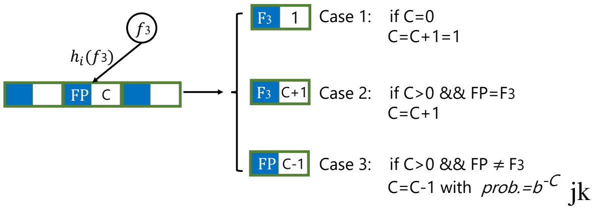

Figure 2: The main insertion cases of HeavyKeeper. Note: 1) $\mathbb{F}_3$ is the fingerprint of flow f_3. 2) $b > 1$ and $b \approx 1$ (*e.g.*, $b = 1.08$). 3) In Case 3, when C is decayed to 0, the fingerprint field will be replaced by $\mathbb{F}_3$, and then counter C is set to 1.

Case 1: When $A_j[h_j(f_i)].C = 0$. It means that no flow has been mapped to this bucket, then HeavyKeeper sets $A_j[h_j(f_i)].FP = \mathbb{F}_i$ and $A_j[h_j(f_i)].C = 1$, where $\mathbb{F}_i$ represents the fingerprint of f_i.

Case 2: When $A_j[h_j(f_i)].C > 0$ and $A_j[h_j(f_i)].FP = \mathbb{F}_i$. It means $A_j[h_j(f_i)].C$ is probably the estimated size of f_i. In this case, HeavyKeeper increments $A_j[h_j(f_i)].C$ by 1.

Case 3: When $A_j[h_j(f_i)].C > 0$ and $A_j[h_j(f_i)].FP \neq \mathbb{F}_i$. It means that $A_j[h_j(f_i)].C$ is not the estimated size of f_i. In here, HeavyKeeper applies the *exponential-weakening decay* strategy to this bucket: it decays $A_j[h_j(f_i)].C$ by 1 with a probability P_{decay}. After decay, if $A_j[h_j(f_i)].C = 0$, HeavyKeeper replaces $A_j[h_j(f_i)].FP$ with $\mathbb{F}_i$, and sets $A_j[h_j(f_i)].C$ to 1. Therefore, as long as flows are mapped to a bucket, its counter field will never be 0.

Query: To query the size of a flow f_i, HeavyKeeper first computes the d hash functions to get d buckets $A_j[h_j(f_i)]$ $(1 \leqslant j \leqslant d)$. Among the d mapped buckets, it chooses those buckets whose fingerprint fields are equal to $\mathbb{F}_i$. It then reports the maximum counter field of those buckets, *i.e.*, $max_{1 \leqslant j \leqslant d}\{A_j[h_j(f_i)].C\}$ where $A_j[h_j(f_i)].FP = \mathbb{F}_i$.

For convenience, for those d mapped buckets of f_i, if $A_j[h_j(f_i)].FP = \mathbb{F}_i$, we say that f_i is ***held*** at bucket $A_j[h_j(f_i)]$. Ignoring the limited impact of fingerprint collisions, we prove that *the reported size for each flow is equal to or smaller than the real flow size* in Section 4.1. If a flow is *held* at no mapped bucket, it reports that it is a mouse flow. If a flow is *held* at multiple buckets, HeavyKeeper reports the maximum counter field.

Decay probability: The decay probability P_{decay} in the exponential-weakening decay strategy is an important parameter. Here, we use the following exponential function to calculate the probability:

$$P_{decay} = b^{-C} \quad (b > 1)$$

where C is the value in the current counter field, and where b ($b > 1$ and $b \approx 1$, *e.g.*, $b = 1.08$) is a pre-defined exponential base number. Therefore, the larger size a flow has, the harder its size is decayed. For elephant flows, it is held at several buckets, and the corresponding counter fields are incremented regularly, while decayed with a very small probability. Therefore, the error rate for estimated sizes of elephant flows is very small.

Note: Our data structure of d arrays may show some similarity with that of CM [14]. But similarity stops there. CM records the sizes of all flows; we record the sizes of a small number of flows. CM does not store flow IDs; we do. CM stores information of each flow in d counters; we keep each flow mostly in one bucket, while d-hashing helps find an empty bucket. CM does not have to worry about the issue of kicking out existing flows to make room for new ones, which is what our exponential delay does.

Example: As shown in Figure 1, given an incoming packet $\mathbb{P}_5$ belonging to flow f_3, we compute the d hash functions to obtain one bucket in each array. In the mapped bucket of the first array, the fingerprint field is not equal to $\mathbb{F}_3$ and the counter field is 21, thus we decay the counter field from 21 to 20 with a probability of 1.08^{-21} (assume $b = 1.08$). In the second mapped bucket, the fingerprint field is not $\mathbb{F}_3$ yet, and with a probability of 1.08^{-1}, we decay the counter field from 1 to 0. If the counter field is decayed to 0, we set the fingerprint field to $\mathbb{F}_3$, and set the counter field to 1. In the last mapped bucket, the fingerprint field is $\mathbb{F}_3$, we increment the counter field from 7 to 8.

Analysis: HeavyKeeper uses fingerprint to identify and keep elephant flows. If a mouse flow with a small flow size is held at a bucket, it will be replaced by other flows mapped to this bucket soon, because each flow mapped to this bucket with a different fingerprint will decay the counter field with a high probability ($b^{-C} \to 1$ when C is small). If an elephant flow is held at a bucket, the corresponding counter field can easily be incremented to a large value since elephant flows have many incoming packets. Moreover, the decay probability becomes very small ($b^{-C} \to 0$ when C is large) as the counter field increases to a large value. Therefore, mouse flows can hardly be held in HeavyKeeper for a long time, and thus have a large probability to be *passers-by* of HeavyKeeper. However, elephant flows can keep stable in HeavyKeeper, and the estimated sizes of elephant flows are accurate.

3.3 Basic Version for Finding the Top-*k* Elephant Flows

To find top-*k* elephant flows, our basic version just uses a HeavyKeeper and a min-heap. The min-heap is used to store the IDs and sizes of top-*k* flows. For each incoming packet $\mathbb{P}_l$ belonging to flow f_i, we first insert it into HeavyKeeper. Suppose that HeavyKeeper reports the size of f_i as $\hat{n}_i$. If f_i is already in the min-heap, we update its estimated flow size with $max(\hat{n}_i, min\_heap[f_i])$, where $min\_heap[f_i]$ is the recorded size of f_i in min-heap. Otherwise, if $\hat{n}_i$ is larger than the smallest flow size which is in the root node of the min-heap, we expel the root node from the min-heap, and insert f_i with $\hat{n}_i$ into the min-heap. To query top-k flows, we simply report the k flows in the min-heap with their estimated flow sizes.

3.4 Optimizations

In this section, we propose further optimization methods to avoid accidental errors and improve speed.

Optimization I: Fingerprint Collisions Detection.
Problems: Assume that there is a bucket in HeavyKeeper where flow f_i is held, and a mouse flow f_j mapped to the same bucket has the same fingerprint as f_i, *i.e.*, $\mathbb{F}_i = \mathbb{F}_j$ due to hash collisions. Then, the mouse flow f_j is also held at this bucket, and its estimated size is drastically over-estimated. In the worst case, if flow f_j has a fingerprint collision in all d arrays, the mouse flow f_j will probably be inserted into the min-heap. It can hardly be expelled due to its drastically over-estimated size. To address this problem, we propose a solution based on the following Theorem.

Theorem 1. *When there is no fingerprint collision, after a flow f_i is inserted into HeavyKeeper, if its estimated size $\hat{n}_i$ is larger than n_{min}, then we must have*

$$\hat{n}_i = n_{min} + 1$$

The proof of this Theorem is not hard to derive and we skip it due to space limitations.

Solution: Based on Theorem 1, if f_i is not in the min-heap but $\hat{n}_i > n_{min} + 1$, then f_i is a mouse flow whose size is drastically over-estimated due to fingerprint collision. Therefore, we should not insert f_i into the min-heap in this case.

Optimization II: Selective Increment.
Problem: If a flow f_i is not in the min-heap, then the estimated flow size should be no larger than n_{min}. However, due to fingerprint collisions, there could be some mapped buckets of flow f_i where the fingerprint field is $\mathbb{F}_i$ and the counter field is larger than n_{min}. In this case, flow f_i is not the flow that is held at this bucket, and thus increasing the corresponding counter field can only incur extra error.

Solution: In this case, instead of incrementing or decaying the corresponding counter field, we make no change.

Optimization III: Speed Acceleration.
Problem: Our basic version of using the min-heap is the most memory efficient solution. However, the processing speed is limited, because the time complexity for updating and searching a flow in the min-heap is $O(log(k))$ and $O(k)$ respectively, which are time-consuming.

Solution: The min-heap is actually used to record the flow IDs of elephant flows and their estimated flow sizes. In this optimization version, instead of using the min-heap, we use a single array to record the flow IDs. Specifically, we define a flow size threshold η (*e.g.*, $\eta = 1000$). For each incoming flow, if its estimated size *is equal to* η, we record the flow ID in the array. As we record the fingerprints of elephant flows, the flow size will increases at most by 1 for each incoming packet when assuming there is no fingerprint collision. Therefore, any flow whose estimated size is larger than or equal to η is recorded in this array once in most cases. Further, this optimization of using an array is only suitable for sketches that record flow IDs or fingerprints.

Algorithm 1: Insertion process for finding top-k flows.

```
Input: A packet $\mathbb{P}_l$ belonging to flow $f_i$
$flag \leftarrow false$;
if $f_i \in min\_heap$ then
    $flag \leftarrow true$;
$maxv \leftarrow 0$;
for $j \leftarrow 1$ to $d$ do
    $C \leftarrow A_j[h_j(f_i)].C$;
    if $A_j[h_j(f_i)].FP = \mathbb{F}_i$ then
        if $flag = true$ or $C < min\_heap.n_{min}$ then
            $A_j[h_j(f_i)].C++$;
        $maxv \leftarrow max(maxv, A_j[h_j(f_i)].C)$;
    else
        if $rand(1) < b^{-C}$ then
            $A_j[h_j(f_i)].C--$;
            if $A_j[h_j(f_i)].C = 0$ then
                $A_j[h_j(f_i)].FP \leftarrow \mathbb{F}_i$;
                $A_j[h_j(f_i)].C \leftarrow 1$;
                $maxv \leftarrow max(maxv, 1)$;
if $flag = true$ then
    $min\_heap[f_i] \leftarrow max(maxv, min\_heap[f_i])$;
else
    if $min\_heap$ has empty buckets or
      $maxv - n_{min} = 1$ then
        $min\_heap.insert(f_i)$;
```

3.5 Final Version

Based on the basic version, we propose the common final version using the first two optimization methods, and propose the accelerated final version using the third optimization methods. The insertion and query processes of the common final version of our algorithm are as follows (see pseudo-code in Algorithm 1).

Insertion: All counters and fingerprints in HeavyKeeper and the min-heap are initialized to 0. For each incoming packet $\mathbb{P}_l$ belonging to flow f_i, these are the following three steps for each insertion:

Step 1: Check whether flow f_i is already monitored by the min-heap. For convenience, we use a boolean variable $flag$ to represent the result.

Step 2: Insert f_i into HeavyKeeper. According to Optimization II, for each mapped bucket, if the fingerprint field is equal to $\mathbb{F}_i$, increment the counter field only when $flag = true$ or $C < n_{min}$, where C is the original value in the counter field.

Step 3: Get an estimated size $\hat{n}_i$ of flow f_i from HeavyKeeper. According to Optimization III, if $flag = true$, we update the estimated size of flow f_i in the min-heap with $\hat{n}_i$. If $flag = false$, insert flow f_i into the min-heap with $\hat{n}_i$ in only two cases: 1) the number of flows that are in the min-heap is less than k; 2) $\hat{n}_i = n_{min} + 1$.

Query top-k flows: It reports the k flows recorded in the min-heap and their estimated flow sizes.

Analysis: Since HeavyKeeper achieves very small error rate on the flow size estimation of elephant flows, it can significantly reduce the error in finding top-*k* elephant flows. Furthermore, the first two optimizations reduce the impact of fingerprint collisions, and enhance the precision of finding top-*k* elephant flows and their flow size estimation. The third optimization method has a constant processing time for insertions: 1) For most incoming packets, they are only inserted into HeavyKeeper, which requires d (*e.g.*, $d = 2$) memory accesses. 2) For some packets belonging to elephant flows, they are inserted into both HeavyKeeper and the array. It requires $d+1$ memory accesses in the worst case. Therefore, the time complexity of insertion process is $O(d)$. Therefore, the processing speed of the accelerated final version is fast on average and constant in the worst case.

3.6 Other uses of HeavyKeeper

Besides finding top-*k* flows in a network stream, HeavyKeeper can also perform other tasks in network traffic measurement, such as heavy hitter detection and change detection. Due to space limitations, here we only briefly introduce how to perform these tasks using HeavyKeeper.

Heavy hitter detection: Given a threshold θ, a heavy hitter [13] is a flow whose size $n_i > \theta N$, where N is the number of packets in total. The heavy hitter detection algorithm is very similar to that of finding top-*k* flows. The only difference is that when querying heavy hitters, it reports those flows whose estimated size is larger than θN in min-heap.

Change detection: The network stream is divided into fixed-size time bins. Given a flow, if the difference of its flow sizes in two adjacent time bins is larger than a predefined threshold, then the flow is called a heavy change [13, 33]. We use the very flow ID as the fingerprint of each flow. For two adjacent time bins, we insert their packets into two different HeavyKeepers. By comparing buckets in the corresponding location in the two HeavyKeepers, we obtain the estimated difference of sizes of the flows, and report the heavy changes by checking if the difference is larger than the threshold.

4 Mathematical Analysis

In this section, we first prove that there is no over-estimation in HeavyKeeper, and then derive the formula of its error bounds.

4.1 Proof of No Over-estimation Error of HeavyKeeper

Theorem 2. *Let $n_i(t)$ be the real size of flow f_i at time t, and let $A_j[h_j(f_i)](t).C$ be the counter field of the mapped bucket of flow f_i in the j^{th} array at time t. If there is no fingerprint collision, then*

$$\forall j,t,\ A_j[h_j(f_i)](t).C \leqslant n_i(t) \tag{1}$$

Proof. When $t = 0$, no packet maps into this bucket, so $n_i(0) = 0$ and $A_j[h_j(f_i)](t).C = 0$. Therefore, the theorem holds at time 0. Let's now prove by induction that the theorem holds at any time.

When $t = 0$, the theorem holds.

If the theorem holds when $t = v$, let's prove that the theorem also holds when $t = v+1$. There are three cases when $t = v+1$:

Case 1: The new incoming packet is not mapped to bucket $A_j[h_j(f_i)]$. Then $n_i(v+1) = n_i(v)$ and $A_j[h_j(f_i)](v+1).C = A_j[h_j(f_i)](v).C$. Therefore, $A_j[h_j(f_i)](v+1).C \leqslant n_i(v+1)$.

Case 2: The new incoming packet belongs to flow f_i. Then $n_i(v+1) = n_i(v)+1$ and $A_j[h_j(f_i)](v+1).C = A_j[h_j(f_i)](v).C+1$. Therefore, $A_j[h_j(f_i)](v+1).C \leqslant n_i(v+1)$.

Case 3: The new incoming packet is mapped to bucket $A_j[h_j(f_i)]$ but does not belong to flow f_i. Then $A_j[h_j(f_i)](v+1).C = A_j[h_j(f_i)](v).C$ or $A_j[h_j(f_i)](v+1).C = A_j[h_j(f_i)](v).C - 1$, and $n_i(v+1) = n_i(v)$. Therefore, $A_j[h_j(f_i)](v+1).C \leqslant n_i(v+1)$.

Therefore, for any time t,

$$A_j[h_j(f_i)](t).C \leqslant n_i(t)$$

□

4.2 Error Bound of HeavyKeeper

Definition 4.1. *Given a small positive number ε, $Pr\{n_i - \hat{n}_i \geqslant \lceil \varepsilon N \rceil\}$ ($n_i \geqslant \hat{n}_i$) represents the probability that the error of the estimated flow size $n_i - \hat{n}_i$ is larger than εN. If $Pr\{n_i - \hat{n}_i \geqslant \lceil \varepsilon N \rceil\} \leqslant \delta$, the algorithm is said to achieve (ε,δ)-counting.*

(ε,δ)-counting is a metric to evaluate the error rate of the algorithm. Here HeavyKeeper is proved to achieve (ε,δ)-counting, showing that HeavyKeeper achieves a low error rate in estimating the sizes of top-*k* flows.

Theorem 3. *Let's assume that there is no fingerprint collision and the fingerprint of an elephant flow is held at its mapped bucket all the time. Let's focus on one single array of HeavyKeeper. Given a small positive number ε, and an elephant flow f_i whose size is n_i is held at that bucket,*

$$Pr\{n_i - \hat{n}_i \geqslant \lceil \varepsilon N \rceil\} \leqslant \frac{1}{\varepsilon w n_i (b-1)} \tag{2}$$

where w is the width of each array, N the total number of packets, and b the exponential base.

Proof. Let's focus on the j^{th} array. Flow f_i is correctly reported, so at the end, the fingerprint of flow f_i is held in the $h_j(f_i)^{th}$ bucket of the j^{th} array. Let $I_{i,j,i'}$ be a binary random variable, defined as

$$I_{i,j,i'} = \begin{cases} 0 & (f_i = f_{i'}) \vee (h_j(f_i) \neq h_j(f_{i'})) \\ 1 & (f_i \neq f_{i'}) \wedge (h_j(f_i) = h_j(f_{i'})) \end{cases} \tag{3}$$

$I_{i,j,i'} = 1$ iff different flows f_i and $f_{i'}$ are held at the same bucket in the j^{th} array. We define random variable $X_{i,j}$ as:

$$X_{i,j} = \sum_{v=1}^{M} I_{i,j,i'} n_{i'} \tag{4}$$

$X_{i,j}$ represents the sum of the sizes of the flows held at the same bucket as flow f_i, except for the size of flow f_i itself. Assume that for each incoming packet, if it belongs to flow f_i, the counter field is incremented by 1; if not, the counter field is decayed with a certain probability. We have

$$n_i - X_{i,j} \leqslant A_j[h_j(f_i)].C \leqslant n_i \tag{5}$$

Specifically, if all packets that do not belong to flow f_i decay the counter field, then $A_j[h_j(f_i)].C = n_i - X_{i,j}$. If those packets do not decay the counter field, then $A_j[h_j(f_i)].C = n_i$. Let's define another random variable $P_{i,j,l}$. Among the $X_{i,j}$ packets defined above, $P_{i,j,l}$ is defined as the probability that the l^{th} packet decays the counter field. Therefore,

$$A_j[h_j(f_i)].C = n_i - \sum_{l=1}^{X_{i,j}} P_{i,j,l} \tag{6}$$

Given a small positive number ε, the following formula based on the Markov inequality holds

$$\begin{aligned} &Pr\{A_j[h_j(f_i)].C \leqslant n_i - \varepsilon N\} \\ &= Pr\{n_i - \sum_{l=1}^{X_{i,j}} P_{i,j,l} \leqslant n_i - \varepsilon N\} \\ &= Pr\{\sum_{l=1}^{X_{i,j}} P_{i,j,l} \geqslant \varepsilon N\} \leqslant \frac{E(\sum_{l=1}^{X_{i,j}} P_{i,j,l})}{\varepsilon N} \end{aligned} \tag{7}$$

Now let's focus on $E(\sum_{l=1}^{X_{i,j}} P_{i,j,l})$. Assume that all packets are uniformly distributed, we have the following formula:

$$Pr\{P_{i,j,l} = \frac{1}{b^{\mathrm{C}}}\} = \frac{1}{A_j[h_j(f_i)].C} = \frac{1}{n_i - E(\sum_{l=1}^{X_{i,j}} P_{i,j,l})} \tag{8}$$

where $1 \leqslant \mathrm{C} \leqslant n_i - E(\sum_{l=1}^{X_{i,j}} P_{i,j,l})$. Let β be $n_i - E(\sum_{l=1}^{X_{i,j}} P_{i,j,l})$ for convenience. As a result,

$$\begin{aligned} E(\sum_{l=1}^{X_{i,j}} P_{i,j,l}) &= \sum_{l=1}^{E(X_{i,j})} E(P_{i,j,l}) \\ &= E(X_{i,j}) \sum_{\mathrm{C}=1}^{\beta} \frac{1}{b^{\mathrm{C}}} \frac{1}{\beta} = \frac{E(X_{i,j})}{\beta} \cdot \sum_{\mathrm{C}=1}^{\beta} \frac{1}{b^{\mathrm{C}}} \\ &= \frac{E(X_{i,j})}{\beta} \cdot \frac{\frac{1}{b}(1 - (\frac{1}{b})^{\beta})}{1 - \frac{1}{b}} \\ &\leqslant \frac{E(X_{i,j})}{n_i b} \cdot \frac{1 - (\frac{1}{b})^{n_i}}{1 - \frac{1}{b}} = \frac{E(X_{i,j})(1 - (\frac{1}{b})^{n_i})}{n_i(b-1)} \end{aligned} \tag{9}$$

Furthermore, for $E(X_{i,j})$, based on Equation 4,

$$E(X_{i,j}) = E(\sum_{v=1}^{M} I_{i,j,i'} n_{i'}) \quad \leqslant \sum_{i'=1}^{M} n_{i'} E(I_{i,j,v}) = \frac{N}{w} \tag{10}$$

Therefore, based on Equation 9,

$$E(\sum_{l=1}^{X_{i,j}} P_{i,j,l}) \leqslant \frac{N(1 - (\frac{1}{b})^{n_i})}{w n_i (b-1)} \leqslant \frac{N}{w n_i (b-1)} \tag{11}$$

then

$$\begin{aligned} Pr\{A_j[h_j(f_i)].C \leqslant n_i - \varepsilon N\} &\leqslant \frac{E(\sum_{l=1}^{X_{i,j}} P_{i,j,l})}{\varepsilon N} \\ &\leqslant \frac{N}{\varepsilon N w n_i (b-1)} = \frac{1}{\varepsilon w n_i (b-1)} \end{aligned}$$

Note that for an elephant flow f_i, n_i is very large, and $(\frac{1}{b})^{n_i} \approx 0$. The estimated size of f_i is the maximum value of $A_j[h_j(f_i)].C$, so we have

$$Pr\{n_i - \hat{n}_i \geqslant \lceil \varepsilon N \rceil\} \leqslant Pr\{\hat{n}_i \leqslant n_i - \varepsilon N\} \leqslant \frac{1}{\varepsilon w n_i (b-1)}$$

□

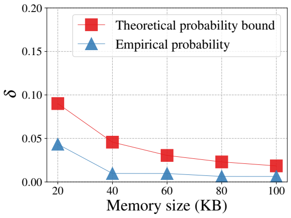

Figure 3: Theoretical bound and empirical probability of HeavyKeeper ($\varepsilon = 2^{-16}$).

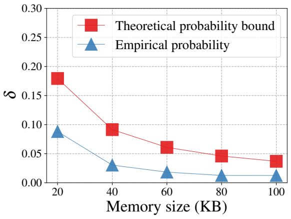

Figure 4: Theoretical bound and empirical probability of HeavyKeeper ($\varepsilon = 2^{-17}$).

To validate the correctness of this error bound, we conduct experiments on the dataset mentioned in Section 5.1. Here, we let $N = 10^7$, $\varepsilon = 2^{-16}$ and 2^{-17}, and vary memory size from 20KB to 100KB. As shown in Figure 3 and Figure 4, the empirical probability of HeavyKeeper is always lower than the theoretical probability bound, confirming the correctness of Theorem 3. Moreover, for the CSS algorithm, achieving such a (ε,δ)-counting requires at least $O(\varepsilon^{-1})$ buckets (*i.e.*, $m = O(\varepsilon^{-1})$), which requires a memory size much larger than 100KB. Therefore, HeavyKeeper is much more memory efficient than CSS.

5 Experimental Results

5.1 Experiment Setup

Platform: Our experiments are run on a server with dual 6-core CPUs (24 threads, Intel Xeon CPU E5-2620 @2 GHz) and 62 GB total system memory. Each core has two L1 caches with 32KB memory (one instruction cache and one data cache) and one 256KB L2 cache. All cores share one 15MB L3 cache.

Dataset:

1) Campus dataset: The first dataset is comprised of IP packets captured from the network of our campus. We rely on the usual definition of a flow, through its 5-tuple, *i.e.*, source IP address, destination IP address, source port, destination port, and protocol type. There are 10M packets in total, belonging to 1M flows. For convenience, we use *campus dataset* to denote this dataset.

2) CAIDA dataset: The second dataset is a CAIDA Anonymized Internet Trace from 2016 [34], made of anonymized IP packets. Each flow in this dataset is identified by the source and destination IP address. We use the first 10M packets, belonging to about 4.2M flows.

3) Synthetic datasets: We generate 10 different synthetic datasets according to a Zipf [35] distribution with different skewness (from 0.3 to 3.0). Each dataset is comprised of 32M packets, belonging to $1 \sim 10$M flows depending on the skewness. Each packet is 4 bytes long. The code of the dataset generator is the one from Web Polygraph [36].

Implementation: The implementation of HeavyKeeper is done in C++. We also implemented in C++ the other related algorithms including Space-Saving (SS), Lossy counting (LC), and CM sketch. The source code of CSS was provided by its author [23], and is written in Java. It is much slower than Space-Saving written in C++. Therefore, we do not include CSS in our speed experiments. For Space-Saving, Lossy counting, and CSS, the **number of buckets** m is determined by the memory size, which is usually much larger than k. When querying top-k flows, they report the largest k flows of them. For CM sketch, the size of the heap is k, the number of arrays is 3, and the width of each array is determined by the memory size. In our algorithm, the number of buckets m in Stream-Summary is equal to k, and HeavyKeeper occupies the rest memory size. Here we set $d = 2$, and w depends on the memory size. Both the fingerprint field and the counter field are 16-bit long. For experiments on throughput, we ignore operations on the min-heap for the CM sketch, because we can only record flows whose estimated size is larger than a pre-defined threshold.

5.2 Metrics

Precision: Precision is defined as $\frac{\mathscr{C}}{k}$. Only $\mathscr{C}$ flows belong to the real top-k flows.

Average Relative Error (ARE): ARE is defined as $\frac{1}{|\Psi|}\sum_{f_i \in \Psi} \frac{|\hat{n}_i - n_i|}{n_i}$, where Ψ is estimated set of top-k flows, $\hat{n}_i$ is the estimated size of flow f_i, and n_i is the real size of flow f_i. ARE evaluates the error rate of the estimated flow size reported by the algorithm.

Average Absolute Error (AAE): AAE is defined as $\frac{1}{|\Psi|}\sum_{f_i \in \Psi} |\hat{n}_i - n_i|$, similarly to ARE.

Throughput: We perform insertions of all packets, record the total time used, and calculate the throughput. The throughput is defined as $\frac{N}{T}$, where N is the total number of packets, and T is the total measured time. We use Million of insertions per second (Mps) to measure the throughput.

5.3 Experiments on Precision

To achieve a head-to-head comparison, we use the same memory size for each algorithm. We perform the experiments for varying memory size and k on the campus and CAIDA datasets, and varying skewness on the synthetic datasets. For experiments of varying memory size, we set $k = 100$. For experiments of varying k, we set the memory size to 100KB. For experiments of varying skewness, we set the memory size to 100KB and set $k = 1000$.

Precision vs. memory size: As shown in Figure 5, for the campus dataset, when memory size is 10KB, the precision of Space-Saving, Lossy counting, CSS, and CM

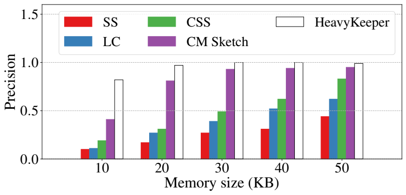

Figure 5: Precision vs. memory size (Campus).

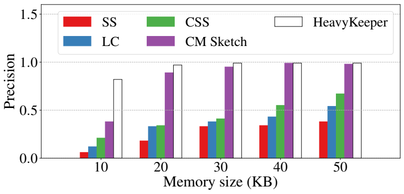

Figure 6: Precision vs. memory size (CAIDA).

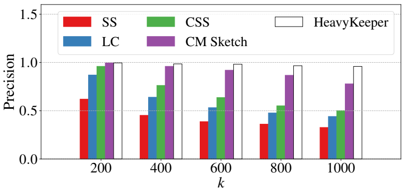

Figure 7: Precision vs. k (Campus).

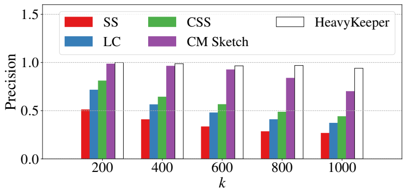

Figure 8: Precision vs. k (CAIDA).

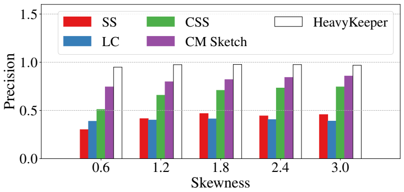

Figure 9: Precision vs. skewness (Synthetic).

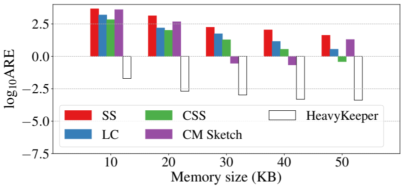

Figure 10: ARE vs. memory size (Campus).

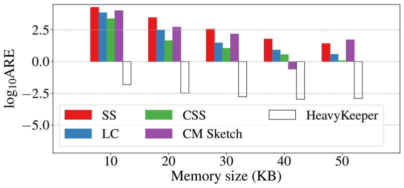

Figure 11: ARE vs. memory size (CAIDA).

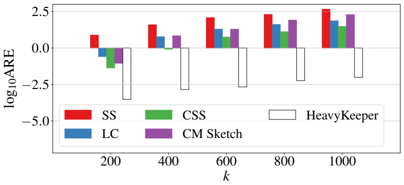

Figure 12: ARE vs. k (Campus).

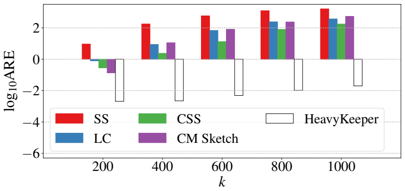

Figure 13: ARE vs. k (CAIDA).

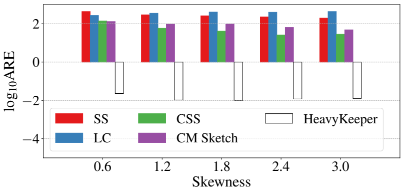

Figure 14: ARE vs. skewness (Synthetic).

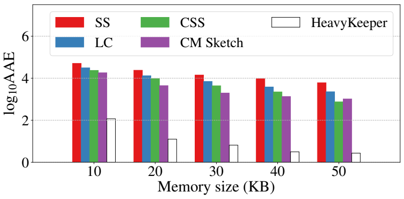

Figure 15: AAE vs. memory size (Campus).

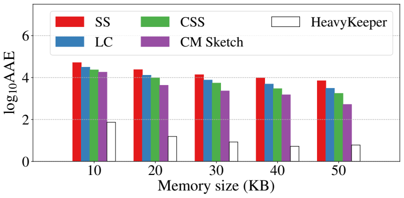

Figure 16: AAE vs. memory size (CAIDA).

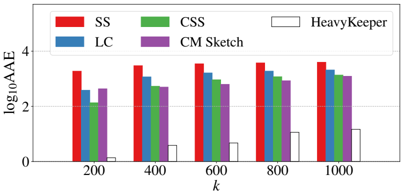

Figure 17: AAE vs. k (Campus).

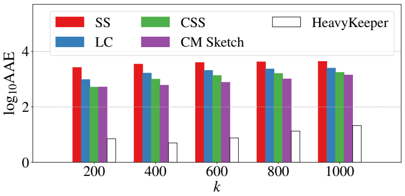

Figure 18: AAE vs. k (CAIDA).

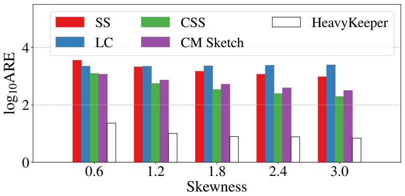

Figure 19: AAE vs. skewness (Synthetic).

sketch is respectively 10%, 11%, 19%, and 41%, while the one of HeavyKeeper is 82%. Furthermore, we find that the precision of HeavyKeeper reaches 100% for a memory size of 30KB, while the corresponding precision of Space-Saving, Lossy counting, CSS, and CM sketch is 27%, 39%, 49%, and 93%. This implies that HeavyKeeper has indeed much better precision than the other three algorithms. We find that Lossy counting is more accurate than Space-Saving. However, as will be mentioned later, Lossy counting is much slower than the other algorithms. For the CAIDA dataset (see Figure 6), we find that the precision of HeavyKeeper reaches 99.99% when memory size is larger than 20KB, while for Space-Saving, Lossy counting, CSS, and CM sketch, precision is respectively 18%, 33%, 34%, and 89% when memory size is 50KB.

Precision vs. k: As shown in Figure 7, for the campus dataset, as k becomes larger, the precision of HeavyKeeper stays high, while it degrades for other algorithms. For the campus dataset, as k becomes larger, the precision of HeavyKeeper is always higher than 95.9%, while that of Space-Saving, Lossy counting, CSS, and CM sketch reaches 32.7%, 44.1%, 50.1%, and 77.9% respectively when $k = 1000$. This happens for two main reasons: 1) larger k requires larger memory usage to store information about more flows; 2) as k increases, the difference of flow sizes among flows becomes smaller, so it is easy to mistake other flows for top-k flows. For the CAIDA dataset (Figure 8), we find that the precision of HeavyKeeper is always above 94%, while for Space-Saving, Lossy counting, CSS, and CM sketch, it is 26.6%, 37.1%, 44%, and 70% respectively when $k = 1000$.

Precision vs. skewness: As shown in Figure 9, the precision of HeavyKeeper reaches 99.99%. For all considered values of skewness, the precision of HeavyKeeper does not go below 94.9%, while the highest precision for Space-Saving, Lossy counting, CSS, and CM sketch is 46.8%, 41.3%, 74.5%, and 85.7%, respectively.

5.4 Experiments on AAE and ARE

In this section, we focus on the ARE and the AAE of the estimated frequency of reported top-k flows. We also conduct experiments with varying memory size, k, and skewness. The parameter settings are the same as in Section 5.3.

ARE vs. memory size: As shown in Figure 10, for the campus dataset, we find that the ARE of HeavyKeeper is smaller than 0.01 when memory size is larger than 20KB, while for Space-Saving, Lossy counting, CSS, and CM sketch, it is larger than 100. Furthermore, we find that the ARE of HeavyKeeper is between 100158 and 648291 times smaller than the one of Space-Saving, between 8450 and 78209 times smaller than the one of Lossy counting, between 945 and 49561 times smaller than the one of CSS, and between 279 and 226986 times smaller than the one of CM sketch. For the CAIDA dataset (see Figure 11), we find that the ARE of HeavyKeeper is between 21119 and 1190365 times smaller than the one of Space-Saving, between 2955 and 456275 times smaller than the one of Lossy counting, between 950 and 154047 times smaller than the one of CSS, and between 238 and 656145 times smaller than the one of CM sketch.

ARE vs. k: As shown in Figure 12, for the campus dataset, we find that the ARE of HeavyKeeper is between 25579 and 56791 times smaller than the one of Space-Saving, between 852 and 9312 times smaller than the one of Lossy counting, between 142 and 3132 times smaller than the one of CSS, and between 293 and 20370 times smaller than the of of CM sketch. For the CAIDA dataset (see Figure 13), we find that the ARE of HeavyKeeper is between 4506 and 121912 times smaller than the one of Space-Saving, between 383 and 23666 times smaller than the one of Lossy counting, between 137 and 8816 times smaller than the one of CSS, and between 66 and 27290 times smaller than the one of CM sketch.

ARE vs. skewness: As shown in Figure 14, for all considered values of skewness, we find that the ARE of HeavyKeeper is between 15566 and 27829 times smaller than that of Space-Saving, between 11915 and 41575 times smaller than that of Lossy counting, between 2174 and 6099 times smaller than that of CSS, and between 3819 and 10080 times smaller than that of CM sketch.

AAE vs. memory size: As shown in Figure 15, for the campus dataset, we find that the AAE of HeavyKeeper is between 433 and 3013 times smaller than that of Space-Saving, between 267 and 1221 times smaller than that of Lossy counting, between 200 and 758 times smaller than that of CSS, and between 155 and 428 times smaller than that of CM sketch. When memory size is 50KB, the AAE of HeavyKeeper is only 2.73, confirming that the estimated flow sizes of almost all reported flows are accurate. For the CAIDA dataset (see Figure 16), we find that the AAE of HeavyKeeper is between 697 and 1810 times smaller than that of Space-Saving, between 421 and 928 times smaller than that Lossy counting, between 289 and 647 times smaller than the one of CSS, and between 86 and 284 times smaller than that of CM sketch.

AAE vs. k: As shown in Figure 17, for the campus dataset, we find that the AAE of HeavyKeeper is between 271 and 1382 times smaller than that of Space-Saving, between 142 and 346 times smaller than that of Lossy counting, between 93 and 196 times smaller than that of CSS, and between 74 and 318 times smaller than that of CM sketch. For CAIDA dataset (see Figure 18), we find that the AAE of HeavyKeeper is between 206 and 694 times smaller than that of Space-Saving, between 118 and 329 times smaller than that of Lossy counting, between 73 and 199 times smaller than that of CSS, and between 67 and 121 times smaller than that of CM sketch.

AAE vs. skewness: From Figure 19, we find that the AAE of HeavyKeeper is between 137 and 209 times smaller than that of Space-Saving, between 96 and 355 times smaller than that of Lossy counting, between 28 and 55 times smaller than that of CSS, and between 45 and 73 times smaller than that of CM sketch.

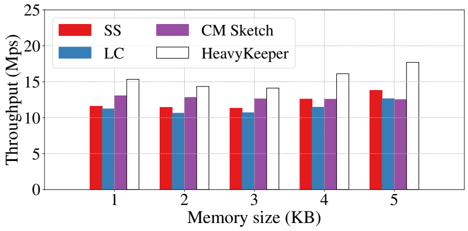

Figure 20: Throughput vs. memory size.

5.5 Experiments on Throughput

We now turn to the throughput of the algorithms. We only report results for the campus dataset due to space limitations. We set $k = 100$, and vary memory size from 10KB to 50KB. Here we use CAIDA datasets.

Throughput vs. memory size: As shown in Figure 20, we find that the throughput of HeavyKeeper is always higher than other algorithms. Indeed, the average throughput of HeavyKeeper is 15.52Mps, while it is 12.15Mps, 11.34Mps, and 12.72Mps for Space-Saving, Lossy counting, and CM sketch. These results show that HeavyKeeper not only is more accurate than previous work, but also achieves higher throughput as well.

6 Open vSwitch Deployment

In this section, we implement our HeavyKeeper algorithm on a software switch platform: Open vSwitch (OVS). We first present details of our implementation, and then present experimental results to show the performance of our algorithm running on Open vSwitch.

6.1 OVS Implementation

The OVS implementation of our HeavyKeeper algorithm consists of three components: 1) the modified OVS datapath, 2) the shared memory buffering flow IDs, and 3) the user-space program of HeavyKeeper processing flow IDs. For each incoming packet, it will be first inserted into the OVS datapath for forwarding. Besides, we modify the source codes of OVS datapath to parse the flow ID of the incoming packet, and then insert its flow ID into the shared memory (the shared memory is created initially). Finally, the user-space program will read the flow IDs from the shared memory, and process them as incoming packets.

In order to improve the performance of OVS, we integrate OVS with DPDK (Data Plane Development Kit). DPDK implements the datapath entirely in the user-space, and thus it eliminates the overhead of a context switch and memory copies between user-space and kernel-space.

6.2 OVS Evaluation

To evaluate the performance of HeavyKeeper implemented in OVS, we conduct experiments to evaluate the throughput of HeavyKeeper and other algorithms. Besides, we also show the throughput of OVS without using any algorithm to show the impact of algorithms. We set the memory size to 50KB.

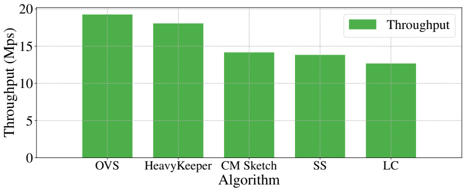

Figure 21: Throughput on OVS platform.

As shown in Figure 21, the throughput of HeavyKeeper is near the original throughput of OVS. Specifically, the throughput of the original OVS is 19.22Mps, and that of HeavyKeeper is 18.03Mps. However, the throughput of CM sketch, Space-Saving, and Lossy Counting is 14.14Mps, 13.80Mps, and 12.64Mps, respectively. The results show that our HeavyKeeper algorithm has little impact to the performance of OVS, while other algorithms decrease the throughput significantly.

7 Conclusion

Finding the top-k elephant flows is a critical task for network traffic measurement. As the line rate increases, it is more and more challenging to design an accurate algorithm that achieves fast and constant speed. Existing algorithms for finding top-k flows cannot achieve high precision when traffic speed is high and memory usage is small, because they do not handle massive mouse flows effectively. In this paper, we propose a novel data structure, called HeavyKeeper, which achieves a much higher precision on top-k queries and a much lower error rate on flow size estimation, compared to previous algorithms. The key idea of HeavyKeeper is that it intelligently omits mouse flows, and focuses on recording the information of elephant flows by using the exponential-weakening decay strategy. Our evaluation confirms that HeavyKeeper achieves 99.99% precision for finding the top-k elephant flows, while also achieving a reduction in the error rate of the estimated flow size by about 3 orders of magnitude compared to the state-of-the-art algorithms.

References

[1] Anirudh Sivaraman, Suvinay Subramanian, and et al. Programmable packet scheduling at line rate. In *Proc. ACM SIGCOMM*, 2016.

[2] Anja Feldmann, Albert Greenberg, and et al. Deriving traffic demands for operational ip networks: Methodology and experience. In *Proc. ACM SIGCOMM*, 2000.

[3] Anukool Lakhina, Mark Crovella, and Christiphe Diot. Characterization of network-wide anomalies in traffic flows. In *Proc. ACM IMC*, 2004.

[4] Ori Rottenstreich and János Tapolcai. Optimal rule caching and lossy compression for longest prefix matching. *IEEE/ACM Transactions on Networking*, 25(2):864–878, 2017.

[5] Katsiaryna Mirylenka, Graham Cormode, Themis Palpanas, and Divesh Srivastava. Conditional heavy hitters: detecting interesting correlations in data streams. *The VLDB Journal*, 24(3):395–414, 2015.

[6] Joong Hyuk Chang and Won Suk Lee. Finding recent frequent itemsets adaptively over online data streams. In *Proc. ACM SIGKDD*, pages 487–492. ACM, 2003.

[7] Yin-Ling Cheung and Ada Wai-Chee Fu. Mining frequent itemsets without support threshold: with and without item constraints. *IEEE TKDE*, 16(9):1052–1069, 2004.

[8] Gerard Salton and Michael J McGill. Introduction to modern information retrieval. 1986.

[9] Mohamed A Soliman, Ihab F Ilyas, and Kevin Chen-Chuan Chang. Top-k query processing in uncertain databases. In *Proc. IEEE ICDE*, pages 896–905, 2007.

[10] Yu Zhang, BinXing Fang, and YongZheng Zhang. Identifying heavy hitters in high-speed network monitoring. *Science China Information Sciences*, 53(3):659–676, 2010.

[11] Elisa Bertino. Introduction to data security and privacy. *Data Science and Engineering*, 1(3):125–126, 2016.

[12] Pratanu Roy, Arijit Khan, and Gustavo Alonso. Augmented sketch: Faster and more accurate stream processing. In *Proc. SIGMOD 2016*.

[13] Graham Cormode. Sketch techniques for approximate query processing. *Foundations and Trends in Databases. NOW publishers*, 2011.

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

[15] Cristian Estan and George Varghese. *New directions in traffic measurement and accounting*, volume 32. ACM, 2002.

[16] Yin Zhang, Matthew Roughan, Walter Willinger, and Lili Qiu. Spatio-temporal compressive sensing and internet traffic matrices. In *ACM SIGCOMM Computer Communication Review*, volume 39, pages 267–278. ACM, 2009.

[17] Theophilus Benson, Aditya Akella, and David A Maltz. Network traffic characteristics of data centers in the wild. In *Proceedings of the 10th ACM SIGCOMM conference on Internet measurement*, pages 267–280. ACM, 2010.

[18] Graham Cormode, Balachander Krishnamurthy, and Walter Willinger. A manifesto for modeling and measurement in social media. *First Monday*, 15(9), 2010.

[19] Dave Maltz. Unraveling the complexity of network management. 2009.

[20] Ilker Nadi Bozkurt, Yilun Zhou, Theophilus Benson, Bilal Anwer, Dave Levin, Nick Feamster, Aditya Akella, Balakrishnan Chandrasekaran, Cheng Huang, Bruce Maggs, et al. Dynamic prioritization of traffic in home networks. 2015.

[21] Zhetao Li, Fu Xiao, Shiguo Wang, Tingrui Pei, and Jie Li. Achievable rate maximization for cognitive hybrid satellite-terrestrial networks with af-relays. *IEEE Journal on Selected Areas in Communications*, 36(2):304–313, 2018.

[22] Muhammad Habib ur Rehman, Chee Sun Liew, Assad Abbas, Prem Prakash Jayaraman, Teh Ying Wah, and Samee U Khan. Big data reduction methods: a survey. *Data Science and Engineering*, 1(4):265–284, 2016.

[23] Ran Ben-Basat, Gil Einziger, Roy Friedman, and Yaron Kassner. Heavy hitters in streams and sliding windows. In *Proc. IEEE INFOCOM*, 2016.

[24] Ahmed Metwally, Divyakant Agrawal, and Amr El Abbadi. Efficient computation of frequent and top-k elements in data streams. In *Proc. Springer ICDT 2005*.

[25] Moses Charikar, Kevin Chen, and Martin Farach-Colton. Finding frequent items in data streams. *Automata, languages and programming*, pages 784–784, 2002.

[26] Gurmeet Singh Manku and Rajeev Motwani. Approximate frequency counts over data streams. In *Proc. VLDB*, pages 346–357, 2002.

[27] Zhetao Li, Baoming Chang, Shiguo Wang, Anfeng Liu, Fanzi Zeng, and Guangming Luo. Dynamic compressive wide-band spectrum sensing based on channel energy reconstruction in cognitive internet of things. *IEEE Transactions on Industrial Informatics*, 2018.

[28] Yuliang Li, Rui Miao, Changhoon Kim, and Minlan Yu. Flowradar: A better netflow for data centers. In *NSDI*, pages 311–324, 2016.

[29] Wang Feng and Hamdi Mounir. Matching the speed gap between sram and dram. In *Proc. IEEE HSPR*, pages 104–109, 2008.

[30] Erik Demaine, Alejandro López-Ortiz, and J Munro. Frequency estimation of internet packet streams with limited space. *AlgorithmsESA 2002*, pages 11–20, 2002.

[31] The source codes of heavykeeper and other related algorithms. `https://github.com/papergitkeeper/heavy-keeper-project`.

[32] Gil Einziger and Roy Friedman. Counting with tinytable: Every bit counts! In *Proc. ICDCN 2016*.

[33] Robert Schweller, Ashish Gupta, Elliot Parsons, and Yan Chen. Reversible sketches for efficient and accurate change detection over network data streams. In *Proc. ACM IMC 2004*.

[34] The caida anonymized internet traces 2016. `http://www.caida.org/data/overview/`.

[35] David MW Powers. Applications and explanations of zipf's law. In *Proceedings of the joint conferences on new methods in language processing and computational natural language learning*, pages 151–160. Association for Computational Linguistics, 1998.

[36] Alex Rousskov and Duane Wessels. High-performance benchmarking with web polygraph. *Software: Practice and Experience*, 2004.