On the Practical Detection of the Top-k Flows

Jalil Moraney, Danny Raz
Computer Science Department
Technion - Israel Institute of Technology
{jalilm, danny}@cs.technion.ac.il

***Abstract*—Monitoring network traffic is an important building block for various management and security systems. In typical settings, the number of active flows in a network node is much larger than the number of available monitoring resources and there is no practical way to maintain per-flow state at the node. This situation gave rise to the recent interest in streaming algorithms where complex data structures are used to perform monitoring tasks like identifying the top-k flows using a constant amount of memory. However, these solutions require complicated per-packet operations, which are not feasible in current hardware or software network nodes.**

In this paper, we take a different approach to this problem and study the ability to perform monitoring tasks using efficient built-in counters available in current network devices. We show that by applying non-trivial control algorithms that change the filter assignments of these built-in counters at a fixed time interval, regardless of packet arrival rate, we can get accurate monitoring information. We provide an analytical study of the top-k flows problem and show, using extensive emulation over recent real traffic, that our algorithm can perform at least as well as the best-known streaming algorithms without using complex data structure or performing expensive per-packet operations.

***Index Terms*—top-k Flows; Efficient Monitoring; Software-Defined Networks.**

I. Introduction

The rapid growth of Cloud Computing and the expansion of Infrastructure as a Service (IaaS) [1] as the preferred solution for providing low cost IT introduce many new challenges in the field of infrastructure management. In this context, it is important to address problems related to efficient monitoring of network resources, since the monitored information is a crucial building block in any IaaS management solution. The networked cloud environment is distributed and the system behavior depends on many parameters that belong to different elements of the network and the cloud. Thus, the monitoring process itself, i.e., the process of collecting the relevant information from the different network locations, requires a considerable amount of resources and should be optimized with respect to the overall gained value.

A typical monitoring task, which is straightforward for memory-intensive solutions, is the detection of the top-k flows, i.e., identifying the k flows with the highest rate in a given period of time. When sufficient memory is available, one can track the frequency of each flow and easily detect the k most frequent flows. However, with memory constraints in mind, it is challenging to design monitoring algorithm for the top-k problem using constant (or even sublinear) space with respect to the number of active flows or the overall traffic size.

Detecting the top-k flows that go through a given network node has been addressed before by works in the streaming domain [3], [4] and [2]. All of these approaches require an operation per an arriving packet. In some cases, the operation can be as complicated as querying all counters (usually to calculate a minimal value) and assigning a new value to a different counter. Therefore, these algorithms are required to perform "per packet" complicated operation on complex data structures (such as doubly linked lists) in line rate. This makes using the results of [3] and [4] unsuitable for deployment on commodity network nodes.

Another drawback of the approaches in [3] and [4] is that they perform poorly on heavy-tailed traffics. In [3] on the arrival of a packet from a non-monitored flow, all counters are decremented and possibly a single counter is reassigned, while in [4] on arrival of every non-monitored packet, the minimal counter is reassigned causing poor performance on heavy-tailed traffic. The approach in [2] dealt with heavy-tailed traffic by probabilistically deciding if to reassign a counter on the arrival of a non-monitored packet, which indeed yielded better performance than [3] and [4]. The authors of [2] also deal with the need to use complex data structure by assuming the existence of d-way associative cache that supports metadata updates in the nodes hardware. The usage of such cache required $O(d)$ operation per arriving packet regardless if any counter will be reassigned or even updated by this arrival and to achieve good detection d should be at least 16. This usage of 16-way cache comes at a cost of low precision of about 0.5, which means that the algorithm outputs $2k$ candidate flows that contain $0.9k$ flows of the actual top-k.

It is important to note that these streaming algorithms aim at identifying the top-k flows in terms of *packet rate*, regardless of the packets' sizes. While this might be relevant in some settings, the more important practical problem is identifying the top-k flows in terms of bandwidth, i.e., in terms of bits per second rather than packets per second. This requires, in streaming terminology, to address the more complex *weighted* version of the top-k problem.

Another undesirable character of these approaches is that they require an additive slack of $N\varepsilon$, where N is the number of packets and $frac1\varepsilon$ is correlated to the number of counters. When the number of counter is not big enough for a given N, this additive slack may become too big to provide any meanigfull information about the a top-k flows.

In this paper, we a take a different approach that takes

#Counters	32	64	128	512	1024
"Basic Splitting"	0.976, 75.65%	0.976, 75.65%	0.932, 72.24%	0.911, 70.62%	0.892, 69.14%
"Hash then Split"	0.964, 74.72%	0.957, 74.18%	0.874, 67.75%	0.888, 68.83%	0.851, 65.96%
"RAP" [2]	0.841, 65.19%	0.840, 65.19%	0.833, 64.57%	0.831, 64.41%	0.827, 64.10%
"16W-RAP" [2]	0.835, 64.72%	0.832, 64.49%	0.829, 64.26%	0.824, 63.87%	0.819, 63.48%

TABLE I: Throughput of the monitoring algorithms in million packets per second and performance rate in comparison to an OpenVSwitch performing at 1.29 MPPS.

advantage of flow table entries, already exist in network nodes. Since clearly, we do not have enough counters to explicitly count the packets in each active flow, we have to reconfigure the counters filter over time. However, this is done on a fixed time interval (say every second) and thus all of our operations can be handled in line rate with no need for additional data structures. Moreover, the use of the built-in counters allows us to solve both unweighted and weighted versions of the problem at the same complexity since these built-in counters can count packets and bytes with no additional cost.

In order to evaluate the expected effect of the complex "per pack" operation on the switch's throughput we implemented the state of the art streaming algorithms from [2] on top of an an OpenvSwitch and measured the expected throughput (see Table I). With no addition monitoring algorithm the switch could handle 1.29 MPPS (million packets per second), however when the streaming algorithms are added the throughput went down to 0.83 MPPS. The monitoring technique presented in this paper are implemented on hardware devices using build in fast counters and will have a very small impact on the throughput. However, just to be in the safe side we also implemented these algorithms in software and measured the impact of this implementation on the measured throughput. The results indicate that even in software implementation the most complex version of our techniques (called "Hash then Split") preforms better than the best available streaming techniques.

In general we adopt the approach of [5], and assign to each counter an aggregated subset of the flows rather than a single flow. Then, at the end of each period, the values of the counters are evaluated and new subsets are assigned to the counters. Furthermore, we provide a mechanism to estimate the flow's frequency throughout several periods based on the values at the end of each period. The two main motivations behind this aggregated periodic approach are: minimizing the monitoring related traffic and avoiding interferences from low frequency flows in heavy-tailed traffic. Moreover, in this approach, the algorithms can guarantee full precision in the sense of returning exactly k flows. Such favorable property allows users not to worry about choosing which k flows of the output to treat as the top-k flows.

We present three new techniques that dramatically improve the overall performance. First, we introduce the basic algorithm which in contrast to [5] focuses on several groups of the flow universe simultaneously. Then, we suggest splitting the monitoring process into rounds and leverages the data collected from past rounds in the current round. Finally, we suggest hashing the packets before processing them to break several unwanted properties of the traffic.

The result is a family of efficient monitoring algorithms to the top-k problem which are deployable out of the box on any OpenFlow enabled network node. The algorithms use a configurable constant number of counters and guarantee that no deviation from the allocated counters will ever occur. We evaluate the expected performance of our algorithms on real network traffic through an extensive simulation study using CAIDAs traces [6], [7], [8]. The results indicate that using only a small constant number of counters the algorithms can identify the top-k flows (in terms of the amount of traffic) with very good accuracy. For example, one can detect the top-8 flows out of thousands active flow, using only 32 counters with an average detection rate of 87%.

We also compare the performance of our algorithms to RAP, the best streaming algorithm reported to date [2]. In doing so, one should be very careful in defining the appropriate way to measure counters' usage since the authors of [2] use 16-way cache and assume metadata availability. When using our algorithms to solve the top-k packet rate problem, they perform almost as well as RAP (on CADIA 2015 data), and in some setting a bit better. However, when we implemented RAP and modified it to detect the weighted top-k flows (in terms of bit rate), our algorithms outperform RAP by 10-15%.

The rest of this paper is organized as follows. First, we define the top-k problem and introduce possible approximations for it. Then, we introduce our basic algorithm which solves the top-k problem locally on a network node and suggest two improvements for it. Afterward, we evaluate the suggested algorithms and compare them to the state of art algorithm. Finally, we provide a short description of related work and conclude.

II. The top-k Flows Problem

Given a network node and a set of flows, we are interested in detecting the top-k flows, usually with respect to the total number of bits, in a given time interval. The solution is straightforward when there are sufficient counters, by allocating a specific counter to each flow. However, the number of available counters at the network node is much smaller than the number active of flows [9]. Thus, the main challenge is detecting the top-k flows while using a limited (constant) number of counters.

In the classical frequent-element problem [10], a stream of elements $S = q_1, q_2, \ldots, q_t$ and a set of objects $O = \{o_1, o_2, \ldots, o_n\}$ are given, where any element of the stream belongs to exactly one object, i.e. $1 \leq \forall j \leq t, 1 \leq \exists i \leq n, q_j \in o_i$ and for all $z \neq i, q_j \notin o_z$.

The frequency of each object o_i in the stream,denoted by n_i, is defined as the number of elements that belong to o_i in S. Without loss of generality we can assume that the o_i's are sorted such that $n_1 \geq n_2 \geq \ldots \geq n_n$. The basic notion of the frequent-element problem is the $ExactTop$, its input consists of a stream S, a set of objects O and an integer k, and it returns k objects from O that contain the most frequent objects in S.

When there are more than k objects in the stream that have very close frequencies to the top-k frequencies, it is not important to detect precisely the top-k objects. It is enough to detect any k objects within a slack of the k^{th} frequency. Thus, the $ApproxTop$ approximation problem was suggested in [10]. It has an additional input parameter ε which defines the slack's percentage. A solution to the $ApproxTop$ problem is a set of any k flows that their frequency is S is at least $(1-\varepsilon)n_k$.

Finding the top-k flows from a network node traffic could be formulated as a frequent-element problem where the traffic is a stream of packets, $S = p_1, p_2, \ldots, p_t$. Each packet p_i is part of a specific flow, $flow(p_i) \in \{f_1, f_2, ...f_n\}$. On contrary to a stream of simple elements, packets have different sizes and thus the weighted version of the frequent elements problem should be used. Note that this makes the algorithms more complex and the performance reported in [2] do not apply directly to this case.

We say that an algorithm is a local top-k algorithm if it runs on a network node with traffic S and solves the $ApproxTop(S, O, k, \varepsilon)$ problem.

III. The "Basic Splitting" Algorithm

The main concept of our top-k algorithms is to use *prefix* trie to decide which aggregate set of flows (denoted by flowset in [11]) to monitor in each time step. This follows the steps of [5], [11], [12].

In the "Basic Splitting" algorithm, we identify each flow by a unique *string* over some alphabet and each flowset by a regular expression over the same alphabet, such that all flows contained in the flowset are the flows represented by the strings matching the flowset's regular expression. The motivation behind this approach is to identify each flow by an IP address and each flowset by a CIDR mask, such that a flowset is the group of all flows that their corresponding IP address is included in the flowset's CIDR mask.

We partition the time into constant length discrete segments called epochs. The algorithm allocates counters at the beginning of each epoch to measure the aggregated size of a given flowset. At the end of the epoch and after receiving the measurements, the algorithm decides on a new allocation of the counters to (possibly new) flowsets.

The algorithm works as follows. Given m counters, at each epoch the algorithm partitions the universe of flows into a disjoint set of flowsets F. For each flowset $f \in F$, it assigns a counter to measure the aggregated size of all flows contained in f. At the end of each epoch, it examines the values of the counters and sorts the flowsets according to their size.

According to the sorted results, the algorithm chooses the biggest $m/2$ flowsets and partition each of them into two disjoint flowsets. The motivation is to prepare a new set of m flowsets to monitor at the next epoch. We denote the partitioning operation by $refine(f)$, which generates a set of disjoint flowsets that covers the biggest $m/2$ flowsets of the last epoch.

After generating the refined flowsets, the set F is modified to include the new flowsets and to exclude the old flowset. After a constant number of epochs, we get to a point where each flowset contains a single flow. At this point, the algorithm is monitoring a set of single flows for a whole epoch, it sorts them according to their size and outputs the highest k flows as the top-k flows.

The main drawback of the "Basic Splitting" algorithm is that it assumes the stability of top-k flows through any subinterval. This assumption motivates the periodic refinement the algorithm performs. While in many cases the top-k flows are active throughout the whole monitoring interval, it is not always the case. Furthermore, the algorithm suffers from lack of "recovery mechanism", which allows reconsidering previously discarded flowsets. This can make it miss some of the top-k flows, even if they are very significant but were not active at the beginning of the monitoring interval.

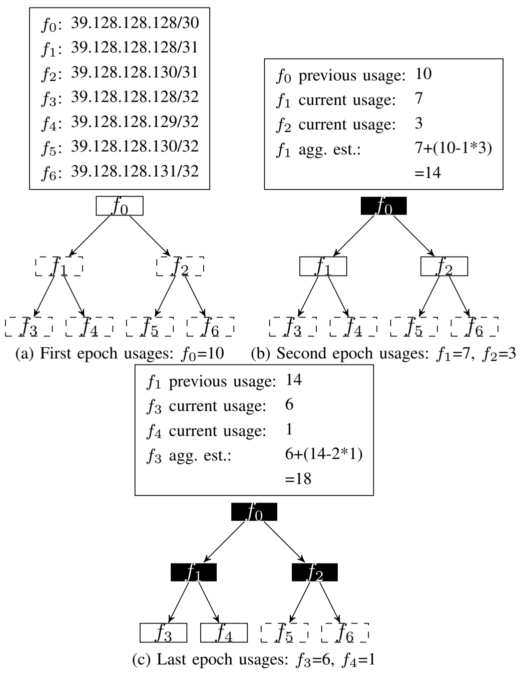
Another drawback of the algorithm is that it does not estimate the number of active flows in each flowset it monitors and does not consider it in its refinement decisions. This is crucial since the number of non-significant flows can be as high as $O(n)$ while the number of significant flows is usually in the hundreds. This drawback might lead to missing top-k flows since an aggregated set of non-significant flows might mask the significant ones, in terms of aggregated size.

IV. The "Multi Round" Algorithm

To overcome the lack of "recovery mechanism" and the fact that the top-k flows stability assumption does not always hold, we suggest the "Multi Round" monitoring algorithm. This algorithm uses the same concept of the "Basic Splitting" algorithm but in an iterative fashion. It splits the entire monitoring period into several monitoring rounds and uses a shorter epoch that allows performing a full "Basic Splitting" in each round.

After the first round, the algorithm generates a list of suspect top-m flows (of this round) and their corresponding values. Since we suspect stability of top-k flows, we allocate for each of the top-$\frac{m}{2}$ flows an exclusive counter in the next round. Thus, in the next round these flows are measured explicitly by $\frac{m}{2}$ exact counters, while the rest of the counters are used to detect the larger candidate $\frac{m}{2}$ flows among the rest.

From these m flows, exact and candidate, the algorithm should decide on the top-k flows, by sorting the flows according to their values. However, there is an inherent problem with this approach, the counters of the exact $\frac{m}{2}$ flows are updated throughout the whole current round, while the rest of



(a) First epoch usages: f_0=10 (b) Second epoch usages: f_1=7, f_2=3

(c) Last epoch usages: f_3=6, f_4=1

Fig. 1: The algorithm's estimation process at non-first round - closeup look on a subtree

the counters (those used to detect the round's candidates) are updated only in the final epoch of the current round.

Thus, there is a need to estimate the values of the candidate $\frac{m}{2}$ flows through the whole round. This estimation is based on the proportion of the sizes of sibling flows. The motivation behind this form of estimation is to counteract the sibling's size throughout the whole round and accumulate the flow's size throughout all epochs.

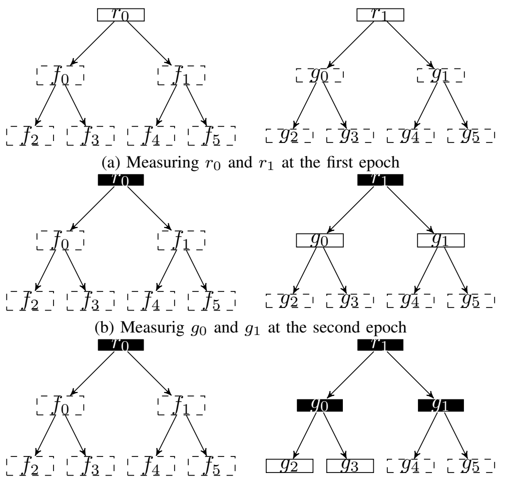
This process is best described using auxiliary variables. We denote by $estimate_{f_j,i}$, the auxiliary variable of the counter measuring f_j at the end of epoch i. Also, $counter_{f_j,i}$ is the value of the counter measuring f_j at end of epoch i, and $parent(f_j)$ is the flowset that its refinement yielded f_j. $sibling(f_j)$ is the other flowset resulted from $refine(parent(f_j))$.

The update of these auxiliary variables for epoch i is done according to $estimate_{f_j,i} = counter_{f_j,i} + (estimate_{parent(f_j),i-1} - i * counter_{sibling(f_j),i})$. The algorithm achieves that by updating the counter's value after the end of the epoch, without maintaining additional hardware counters.

Figure 1 describes an example for this estimation process for the subtree of the flowset 39.128.128.128/30. The aggregated size per epoch of all flows in this subtree is 10, and the sizes of the single flows f_3, f_4, f_5, f_6 are $6, 1, 2, 1$ accordingly. Subfigure 1b shows the computation of the variable $estimate_{f_1,1}$ and Subfigure 1c shows the same process for $estimate_{f_3,2}$, which is the size of f_3 throughout these three epochs.

V. The "Hash then Split" Algorithm

The "Basic Splitting" algorithm takes decisions at the end of each epoch. A problematic aspect of this approach that



r_0 r_1 f_0 f_1 g_0 g_1
f_2 f_3 f_4 f_5 g_2 g_3 g_4 g_5

(a) Measuring r_0 and r_1 at the first epoch

(b) Measurig g_0 and g_1 at the second epoch

(c) Measurig g_2 and g_3 at the last epoch

Fig. 2: The true heaviest flow (top-1) f_2 is masked by several mid-weight flows g_2, g_3, g_4, g_5.

several mid-weighted flows aggregated into a single flowset, might mask a true top-k flow.

Figure 2 illustrates such masking situation. In this example, two counters are used to detect the top-1 flows (heaviest flow). The arrows between two flowsets indicate a relation of a $refine$ operation. We also assume a constant sizes of $(11, 1, 1, 1, 5, 4, 4, 4)$ for $(f_2, f_3, f_4, f_5, g_2, g_3, g_4, g_5)$ respectively. The correct output for this setup should be f_2, but as one can see in this example, the algorithm outputs g_2 instead.

Subfigure 2a depicts the first epoch of the monitoring, where the two counters measure the flowsets r_0 and r_1. At the end of this epoch, the usage values of r_0 and r_1 are 14 and 17 respectively. Thus, at the end of this epoch, the algorithm will assign the counters to measure g_0 and g_1 in the next epoch.

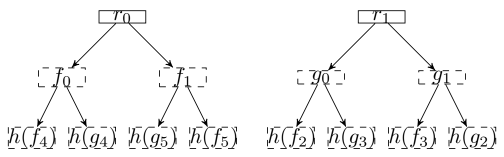
After this decision had been taken, it is impossible for the algorithm to detect the actual top-1 flow. This is true since the algorithm does not have any recovery mechanism. I.e., once it decides not to explore down a branch containing a true top-k flow, it stops assigning counters to that branch and the flow will never be considered again.

Subfigures 2b and 2c depict the behavior of the algorithm after the crucial wrong decision. They show how it measures g_0 and g_1 in the second epoch and g_2 and g_3 in the last epoch. This leads to outputting g_2 with usage value of 5 as the heaviest flow, even though f_2 having usage value of 11.

The probability of such a masking to occur is highest during the very first decisions the algorithm takes. This is true, since the aggregated flowsets gets smaller by half with each new epoch, making it less likely for heavy-weighted flows to be masked by fewer mid-weighted flows.

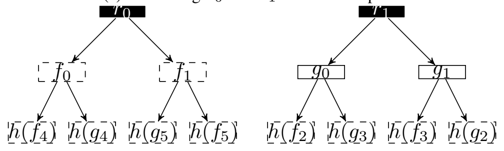
To overcome the lack of recovery mechanism, which is most evident when such masking occurs, we consider hashing the flows passing through the network node. The hashing process takes place by applying a bijection function $h : 2^{\alpha} \rightarrow 2^{\alpha}$, where the flows are defined as strings from α.

The idea behind this is to distribute flows of all weights through all of the branches. This prevents the case of heavy



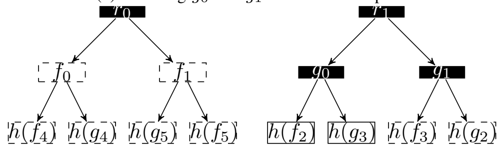
r_0
r_1
f_0
f_1
g_0
g_1
$h(f_4)$ $h(g_4)$ $h(g_5)$ $h(f_5)$ $h(f_2)$ $h(g_3)$ $h(f_3)$ $h(g_2)$

(a) Measuring r_0 and r_1 at the first epoch



r_0
r_1
f_0
f_1
g_0
g_1
$h(f_4)$ $h(g_4)$ $h(g_5)$ $h(f_5)$ $h(f_2)$ $h(g_3)$ $h(f_3)$ $h(g_2)$

(b) Measurig g_0 and g_1 at the second epoch



r_0
r_1
f_0
f_1
g_0
g_1
$h(f_4)$ $h(g_4)$ $h(g_5)$ $h(f_5)$ $h(f_2)$ $h(g_3)$ $h(f_3)$ $h(g_2)$

(c) Measurig f_2 as $h^{-1}(g_2)$ and g_3 as $h^{-1}(g_3)$ at the last epoch

Fig. 3: The true heaviest flow (top-1) $f2$ is masked by several mid-weight flows g_2, g_3, g_4, g_5.

aggregated flowsets that do not contain any heavy single flow. Since the output of the algorithm should be the id of the flow as decoded in the packets and the hashed id, the hash must be invertible and thus a bijection.

An important aspect that must be addressed is the ability to implement such a hash function using current switches in line rate. In order to do so, we limit our implementation to bit-swap hash functions, where several bits in the prefix and suffix of the IP address of the flow are swapped. It is possible to implement such bit-swap functions using an Experimenter Action in OpenFlow, and the packet handling requires one additional table entry (see [13] for more details).

To view the strengths of this approach we revisit the example from Figure 2 and consider the behavior of the algorithm while applying the hash function $h = \{f_2 \rightarrow g_2, f_3 \rightarrow g_4, f_4 \rightarrow f_2, f_5 \rightarrow f_5, g_2 \rightarrow g_5, g_3 \rightarrow g_3, g_4 \rightarrow f_3, g_5 \rightarrow f_4\}$.

Subfigure 3a depicts the first epoch of the monitoring, where the two counters measure the flowsets r_0 and r_1. At the end of this epoch, the sizes of r_0 and r_1 are 10 and 21 respectively. Thus, at the end of this epoch, the algorithm will assign the counters to measure g_0 and g_1 in the next epoch.

In contrary to the previous scenario, now the heaviest flow f_2 is measured in the second epoch, as the flow $h^{-1}(g_2)$. In the second epoch, the sizes of g_0 and g_1 are 15 and 6 respectively. Thus, in the last epoch the algorithm monitors f_2 as $h^{-1}(g_2)$ and g_3 as $h^{-1}(g_3)$. In the last epoch, the algorithm (instead of outputting g_2 as the heaviest flows) outputs $f_2 = h^{-1}(g_2)$ which is indeed the heaviest flow.

A two-round algorithm that hashes the flow based on a given hash function is described in Algorithm “Hash then Split”. The algorithm differs from the “Multi Round” Algorithm by the fact that in the second around it uses a given hash function to distribute the flows and prevent masking.

It is worth to note that the function $get\_round\_hash\_function$ actually determines how the flows are hashed. In the first round, this function assigns both h and h^{-1} the identity function and no hashing occurs at the first round. This means, that the first round detects the heaviest $\frac{m}{2}$ flows and assigns them exact counters.

The exclusion of the heaviest flows and the use of hash functions allow the less heavy flows of the true top-k to stand out. Therefore, after the second round, the algorithm holds candidate heaviest $\frac{m}{2}$ flows from the first round and candidate next heaviest $\frac{m}{2}$ flows from the second round.

Algorithm “Hash then Split”: solving $ExactTop(S, O, k)$ using m counters.

```
Input : A stream of packets $S$, A set of flows $O$ and positive integers $k, m$.
Output: top-$k$ flows from $O$ in $S$
$F = generate_flowsets(O)$;
$number\_rounds = 2$;
$exact\_flows = \phi$;
foreach $round$ in $\{1..number\_rounds\}$ do
    $exact\_counters = assign\_exact\_counters(F, \frac{m}{2}, exact\_flows)$;
    $needed\_epochs = calculate\_needed\_epochs(m, round)$;
    $packets = get\_round\_packets(round)$;
    foreach $counter$ in $exact\_counters$ do
        counter_packets=$\{p \in packets : flow(p) \in counter.flow\}$;
        counter.value= $\sum_{p \in counter\_packets} size(p)$;
    end
    $h, h^{-1} = get\_round\_hash\_function(round)$;
    foreach $epoch$ in $\{1..needed\_epochs\}$ do
        $aggregate\_counters = assign\_aggregate\_counters(F, \frac{m}{2}, epoch, round)$;
        $epoch\_start, epoch\_end = calculate\_epoch\_times(epoch, round)$;
        $packets = get\_epoch\_packets(epoch\_start, epoch\_end, round, exact\_flows)$;
        foreach $counter$ in $aggregate\_counters$ do
            counter_packets=$\{p \in packets : h(flow(p)) \in counter.flow\}$;
            counter.value= $\sum_{p \in counter\_packets} size(p)$;
        end
        foreach $counter$ in $aggregate\_counters$ do
            $counter.value += parent\_counter.value - (epoch - 1) * sibiling\_counter.value$
        end
        $\{hf_i\}_{i=1}^{\frac{m}{2}} = sort(F, aggregate\_counters)$;
        $F = \bigcup_{i=1}^{i=\frac{m}{4}} refine(hf_i)$;
    end
    $\{f_i\}_{i=1}^{\frac{m}{2}} = \{h^{-1}(hf_i)\}_{i=1}^{\frac{m}{2}}$;
    $exact\_flows = sort(exact\_flows, exact\_counters, \{f_i\}_{i=1}^{\frac{m}{2}}, aggregate\_counters)[1 : \frac{m}{2}]$;
end
return $exact\_flows[1 : k]$
```

VI. Results

A. Setup

We used *Mininet* [14], [15] to emulate a software-defined network environment and *Open vSwitch* [16], [17] as the OpenFlow enabled switch. *Ryu* SDN framework [18] was used to build the monitoring application in Python. The setup of the experiments used to evaluate the performance of the algorithms on local node problems was a single switch connected to two hosts via different ports, and the routing entries were set to simply forward traffic from one port to the other.

Tcpreplay[1] was used to replay CAIDA traffic trace [6], [8] into the network. Considering the prefixing nature of the algorithm, in order to keep the original characteristics of the real traffic and the privacy of the users, one should be careful to use real traffic traces that went through "prefix-preserving anonymization" such as the CAIDA traces.

For the evaluation we used two CAIDA traces, from 2014 and 2016[2], considering 1-minute interval. The CAIDA'14 trace has the property that the top-15 flows are the same 15 flow throughout every 12-second interval in this minute. On the other hand, the CAIDA'16 trace has a very significant heavy flow over the whole minute, while the next heaviest flows vary depending on which subinterval is considered. In this sense, the CAIDA'16 trace is an adversarial input due to lack of consistent heavy flows throughout the monitoring intervals.

B. Evaluation of local top-k algorithms

First, we evaluated the performance of the algorithm "Basic Splitting" on both traces. For that, we conducted a series of experiments to solve $ApproxTop(S, k, \varepsilon)$, with different values of k and m using traffic from both datasets and a constant $\varepsilon = 0.05$.

To quantify the performance of the algorithms we define the approximate top-k set (ATk), to be the set of flows with frequency higher than $(1 - \varepsilon)n_k$, where n_k is the frequency of the k^{th} heaviest flow. Then the *Detection Rate (DR)* is the percentage of approximate top-k flows the algorithm managed to detect, i.e., $DR = \frac{|f \in output \cap ATk|}{k}$. We note that since all algorithms output exactly k flows, the detection rate is at most 1.

Figure 4 depicts the effect of k and m on the detection rate of the algorithm "Basic Splitting" using the CAIDA'14 traces. It is evident that for small values of k the algorithm performs well, achieving a detection rate of at least 0.9. As expected, for a given number of counters, increasing k decreases the detection rate. Furthermore, one can see that as the value of m increases the detection rate for a given k also increases, which is expected since the algorithm is using more counters to detect the same heaviest top-k flows.

As expected, when considering experiments with the adversarial traces of CAIDA'16, the performance degrades as seen in Figure 5. For example, the perfect detection rate for $k = 8$ decreases to just above 86%. Furthermore, the algorithm now needs 128 counters instead of 32 to achieve a detection rate about 85% for $k = 12$.

Next, we evaluated the effects of several short monitoring rounds instead of a single longer monitoring round, and the effects of hashing the flows. For that, we conducted the same series of experiments each with the appropriate algorithm compared the achieved detection rates. Note that since the Open vSwitch [17] we used in the evaluation does not support (yet) Experimenter Actions we perform the hash directly on the traces.

[1]Pcap editing and replaying utilities. available at: http://tcpreplay.appneta.com

[2]Data from 2015 was used in the comparison to RAP.

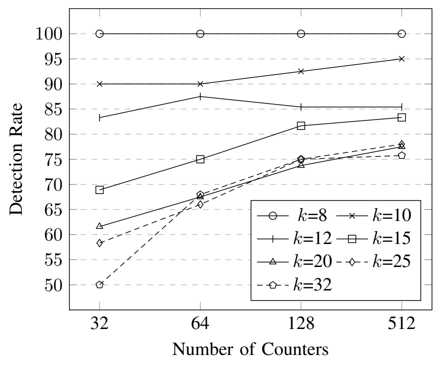


Fig. 4: Effects of k and m on the detection rate of Algorithm "Basic Splitting" - CAIDA'14 traces.

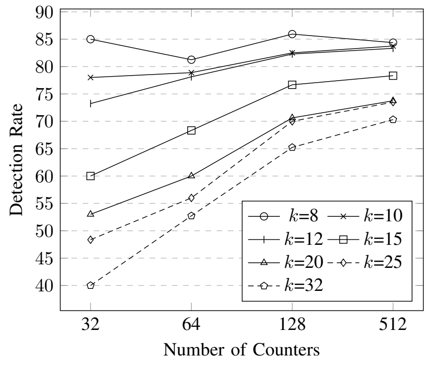


Fig. 5: Effects of k and m on the detection rate of Algorithm "Basic Splitting" - CAIDA'14 and CAIDA'16 traces

Figure 6 depicts the gains in the detection rate for these two approaches compared to the original approach. The horizontal axis marks a set of experiments where the value is the detection rate of the "Basic Splitting" algorithm, while the vertical axis marks the gains the other approaches yielded for this set of experiments. These results show that both approaches provide gains over the "Basic Splitting" algorithm in different cases. Thus, the combined approach of several rounds and hashing the flows starting from the second round was presented at algorithm "Hash then Split".

Figures 7 and 8 show the effect of k and m on the detection rate of the algorithm "Hash then Split" for CAIDA'14 and both traces, respectively. One can see that the same trends we noted regarding the performance of algorithm "Basic Splitting" also hold for this case. Furthermore, these figures also show the superiority of the algorithm "Hash then Split" over the algorithm "Basic Splitting" in all settings.

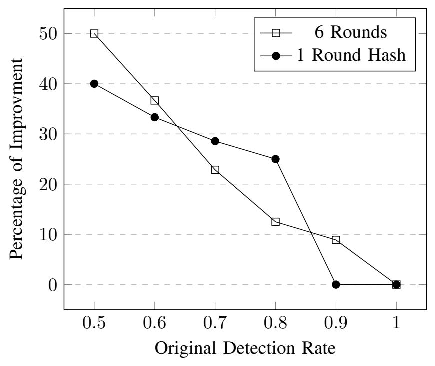


Fig. 6: Gains in the Detection Rate of the MultiRound and hashing approach against the BasicSplitting $k = 10, m = 32$.

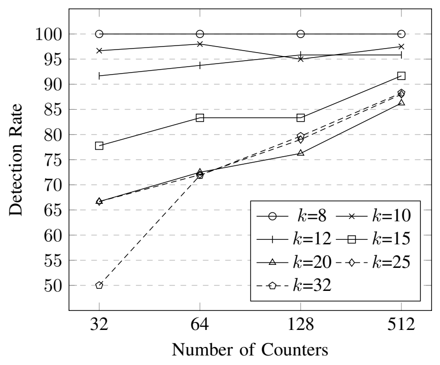


Fig. 7: Effects of k and m on the detection rate of Algorithm “Hash then Split” - CAIDA’14 traces.

C. Comparison to state of art streaming algorithms

The RAP family of algorithms introduced recently in [2] are the best performing top-k streaming algorithms. As mentioned in the Introduction, these algorithms require per packet operations and complex data structures, and even when adapted to perform (i.e., dW-RAP) in line-rate they exhibit low precision rate (of about 50%). Moreover, these algorithms are designed to address the packet rate version of the problem.

Still, it is important to compare the expected performance of our algorithm that use only available counters and works

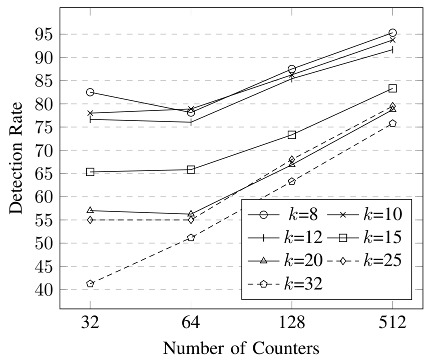


Fig. 8: Effects of k and m on the detection rate of Algorithm “Hash then Split” - CAIDA’14 and CAIDA’16 traces

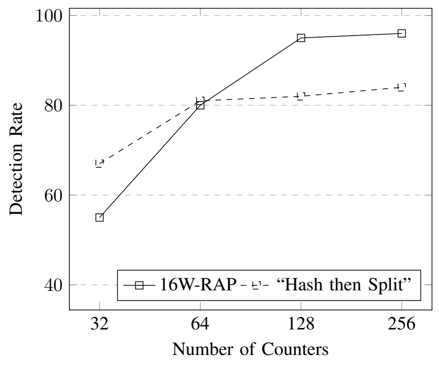


Fig. 9: Comparison of Detection Rate for finding top-32 flows vs. number of counters - CAIDA’15 traces

well also on the traffic rate version. In [2] the evaluation was done on several traces, where CAIDA15 [7] being the dominant real-world trace. The reported experiments included a convergence period and the results depend on the number of packets in the experiment. Since our algorithms are not confined by the number of packets but by time considerations (since this is the way monitoring is done in the industry) we had to slightly modify our algorithms to allow a comparison. We changed the algorithms epochs and rounds to be based on the number of packets passed rather than on time units. Figure 9 compares the detection rate of RAP and the “Hash then Split” algorithms in this modified setting for the same CAIDA15 [7] traces when the goal is the packet based top-k flows. Note that dW-RAP uses a metadata field (flow ID) and we use very basic counters thus in order to compare we assume that pointers to metadata have the same size as counters. One can see that for this objective and the same amount of memory there is no big difference, “Hash then Split” performs a bit better for a small number of counters and a bit worse for large numbers.

However, when we implemented RAP for the more relevant traffic metric the picture is a bit different. As already mentioned in the paper, it is not straightforward to support packet size and total bit count of flows while keeping the deterministic and probabilistic bounds of RAP, so we had to modify the criterion for flow eviction to be a probabilistic function of the amount of bytes in the traffic instead of the number of packets. As depicted in Figure 10, “Hash then Split” performs significantly better than the weighted version of RAP (denoted BW-RAP) in all settings when the number of counters is less than 512. If more than 512 counters are available then both algorithms achieve almost the same high detection rate.

VII. Related Work

Formally, the top-k problem aims at finding the top-k elements that have the highest frequency from a stream of elements. The space requirements for an exact solution to this problem is impractical [10] and thus many relaxations of the original problem were proposed.

The $FindCandidateTop(S, k, l)$ problem was proposed in [10] defining the l elements among which the top-k ele-

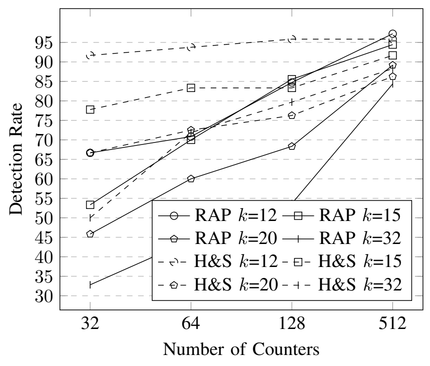


Fig. 10: Comparison of "Hash then Split" and BW-RAP algorithms - CAIDA'14 and CAIDA'16 traces

ments are concealed, with no guarantees on the rank of the remaining $l-k$ elements. While the more practical approximation $FindApproxTop(S,k,\varepsilon)$, also proposed in [10] requires a list of k elements, where every element in the list has a frequency within $(1-\varepsilon)$ of the k^{th} element frequency.

To evaluate the performance of algorithms solving these approximations, two parameters were proposed [19] (in addition to the space requirement): *recall,* the number of correct elements found as a percentage of the number of actual correct elements; and the *precision*, the number of correct elements found as a percentage of the entire output.

Metwally et al. proposed the *Space-Saving* algorithm in [4]. It builds on the ideas of *Frequent* [3] while improving the time complexity of looking up a value of a counter from $O(m)$ to $O(1)$. The main difference from *Frequent* is that when a non-monitored item arrives, the algorithms assign it the minimal counter while preserving its value. They proved that regardless of the data distribution, *Space-Saving* needs $min(|A|, \frac{N}{\varepsilon F_k})$ counters to correctly solve $FindApproxTop(S,k,\varepsilon)$. Where A is the elements universe, N is the size of the stream and F_k is the frequency of the k^{th} element. When considering high volume network traffic, the value of $|A|$ is at least 2^{32} while $\frac{N}{F_k}$ can be very high depending on the traffic's characteristics, thus the number of counters can be unrealistically high.

Ben-Basat et al. [2] introduced the *Randomized Admission Policy (RAP)* algorithm which randomly decides if to reassign a counter to measure non-monitored items. The probability of reassigning a counter is in inverse relation to the counter's value. The motivation behind this reassignment policy is that infrequent non-monitored items will need to arrive several consecutive times to replace a monitored item. The authors acknowledge the difficulty of implementing RAP in hardware and presented a hardware-friendly variant, *d-Way Randomized Admission Policy (dW-RAP)*, which performs poorer than RAP. This variant assumes the existence of d-way associative cache in the node's hardware, where its entries can be partitioned to store metadata (ID) and value.

Several other recent works have also focused on efficient resource constrained flow monitoring, realizing that measurement is crucial for network management and control and even more so in the SDN domain [5], [20], [12]. Moraney and Raz introduced in [5] an efficient scheme to detect flow anomalies, based on a variation of the MRT algorithm presented in [12]. The scheme used a constant number of counters, in periodic assessment and reassignment fashion, to detect anomalous flows regardless of the number of active flows.

In [5] an efficient anomaly detection mechanism was introduced. The mechanism is based on detecting "over usage" in regards to flows, i.e., flows that surpass a given threshold. While such mechanism is efficient in detecting flows that have a certain individual property, it is not usable in detecting flows that uphold a global property such as top-k flow. Furthermore, the authors assumed a given application-specific threshold that defines when the local property holds. However, setting such thresholds is a non-trivial hard task since they need to capture the various aspects of the anomaly and the traffic.

The authors of [20] concentrated on the tradeoff between the amount of available resource and the accuracy of the measurement. Note, that like many of the works in this area, they do not present an analytical framework to study this tradeoff and rather concentrated on important system related issues. The same authors proposed more recently in [12] a network wise dynamic monitoring system where the SDN controller dynamically configures monitoring rules in the different network elements. Such a centralized management entity can thus make use of global information in order to utilize the distributed monitoring resources (typically TCAM rules) in an efficient way, that is, getting as much precision as possible for the given monitoring resources.

VIII. Conclusions

In this paper, we develop a family of practical, efficient memory-constrained algorithms for detecting the top-k flows in terms of total traffic rate. These algorithms use built-in counters available in any switching node and are deployable out of the box on any OpenFlow enabled node. We evaluated the expected performance of these algorithms using real-life packet traces, and the evaluation shows that our new algorithms achieve high detection rates while maintaining full precision regardless of the packet rate. We also show that for the top-k packet rate problem these algorithms perform as well as the best streaming algorithms that use complex data structures and much more elaborated computations. Moreover, for the more relevant weighted top-k problem our algorithms outperform state-of-the-art streaming algorithm when evaluated over recent real traffic.

One future direction is to use this infrastructure as a building-block for detecting network-wise top-k flows. We believe that combining the local performance described in this paper with a smart global policy about the number of counters to be used in each node, will lead to a deployable memory-efficient monitoring system for efficient detection of network-wise top-k flows.

References

[1] S. Goyal, “Software as a Service, Platform as a Service, Infrastructure as a Service A Review,” *International journal of Computer Science & Network Solutions*, vol. 1, no. 3, pp. 53–67, 2013.

[2] R. Ben-Basat, G. Einziger, R. Friedman, and Y. Kassner, “Randomized Admission Policy for Efficient Top-k and Frequency Estimation,” in *IEEE INFOCOM 2017 - The 36th Annual IEEE International Conference on Computer Communications*, 2017.

[3] E. D. Demaine, A. Lopez-Ortiz, and J. I. Munro, “Frequency estimation of internet packet streams with limited space,” in *Proceedings of the 10th Annual European Symposium on Algorithms*. Springer-Verlag, 2002, pp. 348–360. [Online]. Available: http://dl.acm.org/citation.cfm?id=647912.740658

[4] A. Metwally, D. Agrawal, and A. E. Abbadi, “Efficient computation of frequent and top-k elements in data streams,” in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, ser. Lecture Notes in Computer Science, T. Eiter and L. Libkin, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, vol. 3363 LNCS, pp. 398–412. [Online]. Available: http://link.springer.com/10.1007/b104421http://link.springer.com/10.1007/978-3-540-30570-5_27

[5] J. Moraney and D. Raz, “Efficient detection of flow anomalies with limited monitoring resources,” in *2016 12th International Conference on Network and Service Management (CNSM)*. IEEE, oct 2016, pp. 55–63. [Online]. Available: http://ieeexplore.ieee.org/document/7818400/

[6] “The CAIDA UCSD Anonymized Chigaco Internet Traces, 2014.” http://www.caida.org/data/passive/passive_2014_dataset.xml.

[7] “The CAIDA UCSD Anonymized Chigaco Internet Traces, 2015.” http://www.caida.org/data/passive/passive_2015_dataset.xml.

[8] “The CAIDA UCSD Anonymized Chigaco Internet Traces, 2016.” http://www.caida.org/data/passive/passive_2016_dataset.xml.

[9] C. Rotsos, N. Sarrar, S. Uhlig, R. Sherwood, and A. W. Moore, “Oflops: An open framework for openflow switch evaluation,” in *Proceedings of the 13th International Conference on Passive and Active Measurement*, ser. PAM’12. Berlin, Heidelberg: Springer-Verlag, 2012, pp. 85–95. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-28537-0_9

[10] M. Charikar, K. Chen, and M. Farach-Colton, “Finding frequent items in data streams,” *Theoretical Computer Science*, vol. 312, no. 1, pp. 3–15, jan 2004. [Online]. Available: http://linkinghub.elsevier.com/retrieve/pii/S0304397503004006

[11] L. Yuan, C.-N. Chuah, and P. Mohapatra, “ProgME: towards programmable network measurement,” in *Proceedings of the 2007 conference on Applications, technologies, architectures, and protocols for computer communications - SIGCOMM ’07*, J. Murai and K. Cho, Eds. New York, New York, USA: ACM Press, 2007, pp. 97–108. [Online]. Available: http://dl.acm.org/citation.cfm?doid=1282380.1282392

[12] M. Moshref, M. Yu, R. Govindan, and A. Vahdat, “DREAM: Dynamic Resource Allocation for Software-defined Measurement,” in *Proceedings of the 2014 conference on Applications, technologies, architectures, and protocols for computer communications - SIGCOMM ’14*, aug 2014, pp. 419–430. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2740070.2626291

[13] “OpenFlow Switch Specification 1.5.1,” Tech. Rep., 2015.

[14] B. Lantz, B. Heller, and N. McKeown, “A network in a laptop,” in *Proceedings of the Ninth ACM SIGCOMM Workshop on Hot Topics in Networks - Hotnets ’10*, G. G. Xie, R. Beverly, R. Morris, and B. Davie, Eds. New York, New York, USA: ACM Press, 2010, pp. 1–6. [Online]. Available: http://dl.acm.org/citation.cfm?id=1868466http://portal.acm.org/citation.cfm?doid=1868447.1868466

[15] Mininet, “Mininet: An Instant Virtual Network on your Laptop (or other PC) - Mininet,” http://mininet.org/, p. www.mininet.org, 2014. [Online]. Available: http://mininet.org/

[16] B. Pfaff, J. Pettit, T. Koponen, K. Amidon, M. Casado, and S. Shenker, “Extending Networking into the Virtualization Layer,” in *8th ACM Workshop on Hot Topics in Networks*, New York City, NY, USA, 2009. [Online]. Available: http://www.icsi.berkeley.edu/pubs/networking/extendingnetworking09.pdf

[17] OpenvSwitch, “Production Quality, Multilayer Open Virtual Switch,” http://openvswitch.org/, 2016. [Online]. Available: http://openvswitch.org/

[18] RYU SDN Project Team, “RYU SDN Framework,” http://www.osrg.github.io/ryu, p. 455, 2016. [Online]. Available: https://osrg.github.io/ryu-book/en/Ryubook.pdf

[19] G. Cormode and S. Muthukrishnan, “What’s hot and what’s not: tracking most frequent items dynamically,” *ACM Transactions on Database Systems*, vol. 30, no. 1, pp. 249–278, 2005.

[20] M. Moshref, M. Yu, and R. Govindan, “Resource/accuracy tradeoffs in software-defined measurement,” *ACM SIGCOMM workshop on Hot topics in software defined networking (HotSDN)*, p. 73, 2013. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2491185.2491196