Fast Flow Volume Estimation

Ran Ben Basat, Gil Einziger, Roy Friedman

Abstract

The increasing popularity of jumbo frames means growing variance in the size of packets transmitted in modern networks. Consequently, network monitoring tools must maintain explicit traffic volume statistics rather than settle for packet counting as before. We present constant time algorithms for volume estimations in streams and sliding windows, which are faster than previous work. Our solutions are formally analyzed and are extensively evaluated over multiple real-world packet traces as well as synthetic ones. For streams, we demonstrate a run-time improvement of up to 2.4X compared to the state of the art. On sliding windows, we exhibit a memory reduction of over 100X on all traces and an asymptotic runtime improvement to a constant. Finally, we apply our approach to hierarchical heavy hitters and achieve an empirical 2.4-7X speedup.

1. Introduction

Traffic measurement is vital for many network algorithms such as routing, load balancing, quality of service, caching and anomaly/intrusion detection [37, 47, 10, 29, 23, 25]. Typically, networking devices handle millions of concurrent flows [50, 48, 26]. Often, monitoring applications track the most frequently appearing flows, known as *heavy hitters*, as their impact is most significant.

Most works on heavy hitters identification have focused on packet counting [5, 24, 53, 51]. However, in recent years jumbo frames and large TCP packets are becoming increasingly popular and so the variability in packet sizes grows. Consequently, plain packet counting may no longer serve as a good approximation for bandwidth utilization. For example, in data collected by [32] in 2014, less than 1% of the packets account for over 25% of the total traffic. Here, packet count based heavy hitters algorithms might fail to identify some heavy hitter flows in terms of bandwidth consumption.

Hence, in this paper we explicitly address monitoring of flow *volume* rather than plain packet counting. Further, given the rapid line rates and the high volume of accumulating data, an aging mechanism such as a *sliding window* is essential for ensuring data freshness and the volume estimation's relevance. The window variant is motivated by load balancing applications. In these, one wishes to allocate resources to flows in a way that matches the consumed bandwidth. Volume estimation over sliding windows allows analysis of the most recent bandwidth consumption, which is a good indicator of the future. Hence, we study both streams and sliding windows.

Finally, per flow measurements are not enough for certain functionalities like anomaly detection and Distributed Denial of Service (DDoS) attack detection [55, 49]. In such attacks, each attacking device only generates a small portion of the traffic and is not a heavy hitter. Yet, their combined traffic volume is overwhelming. *Hierarchical heavy hitters* (HHH) aggregates traffic from IP addresses that share some common prefix [8]. In a DDoS, when attacking devices share common IP prefixes, HHH can discover the attack. To that end, we consider volume based HHH detection as well.

Before explaining our contribution, let us first motivate why packet counting solutions are not easily adaptable to volume estimation. Counter algorithms typically maintain a fixed set of counters [5, 6, 22, 44, 45, 38] that is considerably smaller than the number of flows. Ideally, counters are allocated to the heavy hitters. When a packet from an unmonitored flow arrives, the corresponding flow is allocated the minimal counter [45] or a counter whose value has dropped below a dynamic threshold [44].

We refer to a stream in which each packet is associated with a *weight* as a *weighted* stream. Similarly, we refer to streams without weights, or when all packets receive the same weight, as *unweighted*. For unweighted streams, ordered data structures allow constant time updates and queries [45, 5], since when a counter is incremented, its relative order among all counters changes by at most one. Unfortunately, maintaining the counters sorted after a counter increment in a weighted stream either requires to search for its new location, which incurs a logarithmic cost, or resorting to logarithmic time data structures like heaps. The reason is that if the counter is incremented by some value w, its relative position might change by up to w positions. This difficulty motivates our work[1].

1.1. Contributions

We contribute to the following network traffic measurement problems: (i) stream heavy hitters, (ii) sliding window heavy hitters, (iii) stream hierarchical heavy hitters. Specifically, our first contribution is *Frequent items Algorithm with a Semi-structured Table* (FAST), a novel algorithm for monitoring flow volumes and finding heavy hitters. FAST processes elements in worst case $O(1)$ time using asymptotically optimal space. A major part of our contribution lies in the detailed formal analysis we perform, which

[1] The most naive approach treats a packet of size w as w consecutive arrivals of the same packet in the unweighted case, resulting in linear update times, which is even worse.

proves the above properties, as well as in the accompanying performance study. We evaluate FAST on five real Internet packet traces from a data center and backbone networks. We demonstrate a speedup of up to a factor of 2.4X compared to previous works.

Our second contribution is *Windowed Frequent items Algorithm with a Semi-structured Table* (WFAST), a novel algorithm for monitoring flow volumes and finding heavy hitters in sliding windows. We evaluate WFAST on five Internet traces and show that its runtime is reasonably fast, and that it requires as little as 1% of the memory of previous work [36]. We analyze WFAST and show that it operates in constant time and is space optimal, which asymptotically improves both the runtime and the space consumption of previous work. We believe that such a dramatic improvement makes volume estimation over a sliding window practical!

Our third contribution is *Hierarchical Frequent items Algorithm with a Semi-structured Table* (HFAST), which finds hierarchical heavy hitters. HFAST is created by replacing the underlying HH algorithm in [46] (Space Saving) with FAST. We evaluate HFAST and show an asymptotic update time improvement as well as a 2.4-7X speedup on real Internet traces.

2. Related Work

Our work addresses three related problems, which we survey below.

2.1. Streams

Probabilistic short counters, or *estimators*, represent large numbers using small counters by degrading precision [53, 24, 54]. By shrinking counters' size, more flows can be monitored in SRAM. But these methods still require maintaining a flow-to-counter mapping that often requires more space than the counters themselves. Sampling is also an attractive approach when space is scarce [15, 2, 28] despite the resulting sampling error.

Sketches such as *Count Sketch (CS)* [11] and *Count Min Sketch (CMS)* [21] are attractive as they enable counter sharing and need not maintain a flow to counter mapping for all flows. Sketches typically only provide a probabilistic estimation, and often do not store flow identifiers. Thus, they cannot find the heavy hitters, but only focus on the volume estimation problem. Advanced sketches, such as Counter Braids [42], Randomized Counter Sharing [40] and Counter Tree [12], improve accuracy but their queries require complicated decoding procedures that can only be done off-line.

In *counter based* algorithms, a flow table is maintained, but only a small number of flows are monitored. These algorithms differ from each other in the size and maintenance policy of the flow table, e.g., *Lossy Counting* [44] and its extensions [22], *Frequent* [38] and *Space Saving* [45]. Given ideal conditions, counter algorithms are considered superior to sketch based techniques. Particularly, Space Saving was empirically shown to be the most accurate [16, 17, 43]. Many counter based algorithms were developed by the databases community and are mostly suitable for software implementation. The work of [5] suggests a compact static memory implementation of Space Saving that may be more accessible for hardware design. Yet, software implementations are becoming increasingly relevant in networking as emerging technologies such as NFVs become popular.

Alas, most previous works rely on sorted data structures such as *Stream Summary* [45] or SAIL [5] that only operate in constant time for unweighted updates. As mentioned, existing sorted data structures cannot be maintained in constant time in the weighted updates case. Thus, a logarithmic time heap based implementation of Space Saving was suggested [17] for the more general volume counting problem. IM-SUM, DIM-SUM [7] and BUS-SS [27] are very recent algorithms developed for the volume heavy-hitters problem (*only* for streams, with *no* sliding windows support). BUS-SS offers a randomized algorithm that operates in constant time. IM-SUM operates in amortized $O(1)$ time and DIM-SUM in worst case constant time. Empirically, DIM-SUM is slower than FAST. Additionally, DIM-SUM requires $\frac{2+\phi}{\epsilon}$ counters, for some $\phi > 0$, to guarantee $N \cdot M \cdot \epsilon$ error and operates in $O(\phi^{-1})$ time. FAST only needs half as many counters for the same time and error guarantees. The very recent work of [1], introduces a mergeable algorithm that operates in amortized constant time.

2.2. Sliding Windows

Heavy hitters on sliding windows were first studied by [3]. Given an accuracy parameter (ε), a window size (W) and a maximal increment size (M), such algorithms estimate flows' volume on the sliding window with an additive error that is at most $W \cdot M \cdot \varepsilon$.

Their algorithm requires $O\left(\frac{1}{\epsilon}\log^2\frac{1}{\epsilon}\right)$ counters and performs queries and updates in $O\left(\frac{1}{\epsilon}\log\frac{1}{\epsilon}\right)$ time. The work of [39] reduces the space requirements and update time to $O\left(\frac{1}{\epsilon}\right)$. An improved algorithm with a constant *update time* is given in [35]. Further, [5] provided an algorithm that requires $O\left(\frac{1}{\epsilon}\right)$ for queries and supports constant time updates and item frequency queries.

The weighted variant of the problem was only studied by [36], whose algorithm operates in $O\left(\frac{A}{\epsilon}\right)$ time and requires $O\left(\frac{A}{\epsilon}\right)$ space for a $W \cdot M \cdot \varepsilon$ approximation; here, $A \in [1, M]$ is the *average* packet size in the window. In this work, we suggest an algorithm for the weighted problem that (i) uses optimal $O\left(\frac{1}{\epsilon}\right)$ space, (ii) performs heavy hitters queries in optimal $O\left(\frac{1}{\epsilon}\right)$ time, and (iii) performs volume queries and updates in constant time.

2.3. Hierarchical Heavy Hitters

Hierarchical Heavy Hitters (HHH) were first defined by [18], and then extended to multiple dimensions in [19, 20, 30, 55, 46]. HHH algorithms monitor aggregates of flows that share a common prefix. To do so, HHH algorithms treat flows identifiers as a hierarchical domain. We denote by H the size of this domain.

A single dimension algorithm requiring $O\left(\frac{H^2}{\epsilon}\right)$ space was introduced in [41]. Later, [52] showed a two dimensions algorithm requiring $O\left(\frac{H^{3/2}}{\epsilon}\right)$ space and update time. The full and partial ancestry algorithms [20] are trie based algorithms that require $O\left(\frac{H}{\epsilon}\log \epsilon N\right)$ space and operate at $O\left(H\log \epsilon N\right)$ time. The state of the art [46] algorithm requires $O\left(\frac{H}{\epsilon}\right)$ space and its update time for weighted inputs is $O\left(H\log(\frac{1}{\epsilon})\right)$.

The algorithm of [46] solves the approximate HHH problem by dividing it into multiple simpler heavy hitters problems. In our work, we replace the underlying heavy hitters algorithm of [46] with FAST, which yields a space complexity of $O\left(\frac{H}{\epsilon}\right)$ and an update complexity of $O(H)$. That is, we improve the update complexity from $O\left(H\log\left(\frac{1}{\epsilon}\right)\right)$ to $O\left(H\right)$. Alternatively, the recent work of [14] suggests a novel HHH algorithm that takes linear space but optimizes the update time.

3. Preliminaries

Given a set $\mathcal{U}$ and a positive integer $M \in \mathbb{N}^+$, we say that $\mathcal{S}$ is a $(\mathcal{U}, M)$-weighted stream if it contains a sequence of $\langle id, weight\rangle$ pairs. Specifically: $\mathcal{S} = \langle p_1, \dots p_N\rangle$, where $\forall i \in 1,\dots,N : p_i \in \mathcal{U} \times \{1,\dots M\}$. Given a packet $p_i = (d_i, w_i)$, we say that d_i is p_i's id while w_i is its weight; packets that share the same id are part of a flow, and we are interested in estimating the volume of flows. N is the *stream length*, and M is the *maximal packet size.* Notice that the same packet id may possibly appear multiple times in the stream, and each such occurrence may potentially be associated with a different weight. Given a $(\mathcal{U}, M)$-weighted stream $\mathcal{S}$, we denote v_x, the

Symbol	Meaning
$\mathcal{S}$	stream
N	number of elements in the stream
M	maximal value of an element in the stream
W	window size
$\mathcal{U}$	the universe of elements
$[r]$	the set $\{0, 1, ..., r-1\}$
ϕ	FAST performance parameter.
v_x	the volume of an element x in S
$\widehat{v_x}$	an estimation of v_x
v_x^W	the volume of element x in the last W elements of S
$\widehat{v_x^W}$	an estimation of v_x^W
ϵ	estimation accuracy parameter
θ	heavy hitters threshold parameter

Table 1: List of Symbols

volume of flow x, as the total weight of all packets with id x. That is: $v_x \triangleq \sum_{\substack{i\in\{1,...,N\}:\\ d_i=x}} w_i$. For a *window size* $W \in \mathbb{N}^+$, we denote the *window volume* of id x as its total weight of packets with id x within the last W packets, that is: $v_x^W \triangleq \sum_{i\in\{N-W+1,...,N\}:d_i=x} w_i$. We seek algorithms that support the operations:

Add($< x, w >$): append a packet with identifier x and weight w to $\mathcal{S}$.

Query(x): return an estimate $\widehat{v_x}$ of v_x.

WinQuery(x): return an estimate $\widehat{v_x^W}$ of v_x^W.

We now formally define the main problems in this work:

$(\epsilon, \mathbf{M})$-Volume Estimation: Query(x) returns an estimation ($\widehat{v_x}$) that satisfies $v_x \leq \widehat{v_x} \leq v_x + N \cdot M \cdot \epsilon$.

$(\mathbf{W}, \epsilon, \mathbf{M})$-Volume Estimation: WinQuery(x) returns an estimation ($\widehat{v_x^W}$) that satisfies $v_x^W \leq \widehat{v_x^W} \leq v_x^W + W \cdot M \cdot \epsilon$.

$(\theta, \epsilon, \mathbf{M})$-Approximate Weighted Heavy Hitters: returns a set $H \subseteq \mathcal{U}$ such that:
$\forall x \in \mathcal{U} : (v_x > N \cdot M \cdot \theta \implies x \in H) \wedge (v_x < N \cdot M \cdot (\theta - \epsilon) \implies x \notin H)$.

$(\mathbf{W}, \theta, \epsilon, \mathbf{M})$-Approximate Weighted Heavy Hitters: returns a set $H \subseteq \mathcal{U}$ such that:
$\forall x \in \mathcal{U} : (v_x^W > W \cdot M \cdot \theta \implies x \in H) \wedge (v_x^W < W \cdot M \cdot (\theta - \epsilon) \implies x \notin H)$.

Our heavy hitter definitions are asymmetric. That is, they require that flows whose frequency is above the threshold of $N \cdot M \cdot \theta$ (or $W \cdot M \cdot \theta$) are included in the list, but flows whose volume is *slightly* less than the threshold can be either included or excluded from the list. This relaxation is necessary as it enables reducing the required amount of space to sub linear. Let us emphasize that the identities of the heavy hitter flows are not known in advance. Hence, it is impossible to a-priori allocate counters only to these flows. The basic notations used in this work are listed in Table 1.

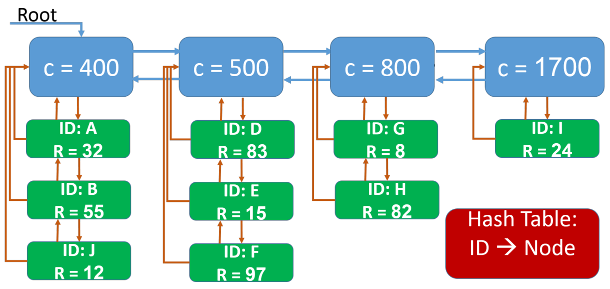


Figure 1: An example of how FAST utilizes the SOS structure. Here, flows are partially ordered according to the third digit (100's), and each flow maintains its own remainder; e.g., the estimated volume of D is $\widehat{v_D} = 583$.

4. Frequent items Algorithm with a Semi-structured Table (FAST)

In this section, we present *Frequent items Algorithm with a Semi-structured Table (FAST)*, a novel algorithm that achieves constant time weighted updates. FAST uses a data structure called *Semi Ordered Summary (SOS)*, which maintains flow entries in a semi ordered manner. That is, similarly to previous works, SOS groups flows according to their volume, each of which is called a *volume group*. The volume groups are maintained in an ordered list. Each volume group is associated with a value C that determines the volume of its nodes. Unlike existing data structures, counters within each volume group are kept unordered. Unlike previous works, the grouping is done at coarse granularity. Each node (inside a group) includes a variable called *Remainder* (denoted R). The volume estimate of a flow is $C + R$ where R is the remainder of its volume node and C is the value of its volume group.

Since the maximal increment, M, is known then a single packet may advance a counter at most M groups. However, as we keep the volume groups at coarse granularity SOS can be configured so that a single packet may advance a flow only $O(1)$ volume groups. For example, in Figure 1, we maintain volume groups at a granularity of 100 and therefore the number of volume groups that a single packet can effect is at most $\frac{M}{100}$. The update complexity of SOS is proportional to how many volume groups a flow entry needs to traverse to maintain the semi ordered state and therefore we can configure SOS so that the update time is at most $O(1)$. Volume queries are satisfied in $O(1)$ time using a separate aggregate hash table which maps each flow identifier to its SOS node. FAST then uses SOS to find a near-minimum flow when needed.

Figure 1 provides an intuitive example for the case $M = 1,000$. Here,

Algorithm 1 FAST (M, ϵ, ϕ)

```
   Initialization:    $C \leftarrow \emptyset, \forall x : c_x \leftarrow 0, r_x \leftarrow 0, s \leftarrow \left\lfloor \frac{M\cdot\phi}{2} + 1 \right\rfloor, \mathfrak{C} \leftarrow \left\lceil \frac{1+\phi}{\epsilon} \right\rceil$.
 1: function ADD(Item $x$, Weight $w$)
 2:     if $x \in C$ or $|C| < \mathfrak{C}$ then
 3:         $c_x \leftarrow c_x + \left\lfloor \frac{r_x+w}{s} \right\rfloor$
 4:         $r_x \leftarrow (r_x + w) \mod s$
 5:         $C \leftarrow C \cup \{x\}$
 6:     else
 7:         Let $m \in \text{argmin}_{y\in C}(c_y)$                ▷ arbitrary minimal item
 8:         $c_x \leftarrow c_m + \lfloor \frac{s-1+w}{s} \rfloor$
 9:         $r_x \leftarrow (s - 1 + w) \mod s$
10:         $C \leftarrow C \setminus \{m\} \cup \{x\}$
11: function QUERY($x$)
12:     if $x \in C$ or $|C| < \mathfrak{C}$ then
13:         return $r_x + s \cdot c_x$
14:     else
15:         return $s - 1 + s \cdot \min_{y\in C} c_y$
```

the volume of an item is calculated by both its group counter (C) and the item's remainder (R), e.g., the volume of A is $400 + 32 = 432$. Flows are partially ordered according to their third digit, i.e., in multiples of 100, or $M/10$. Within a specific group, however, items are unordered, e.g., A, B and J are unordered but all appear before items with volume of at least 500. As the number of lists to skip prior to an addition is $O(1)$, the update complexity is also $O(1)$. Specifically, we need to traverse at most 10 linked lists when updating an item.

Intuitively, flows are only ordered according to volume groups and if we make sure that the maximal weight can only advance a flow a constant number of flow groups then SOS operates in constant time. Alas, keeping the flows only partially ordered increases the error. We compensate for such an increase by requiring a larger number of SOS entries compared to previously suggested fully ordered structures. The main challenge in realizing this idea is to analyze the accuracy impact and provide strong estimation guarantees.

4.1. FAST - Accurate Description

FAST employs $\left\lceil \frac{1+\phi}{\epsilon} \right\rceil$ counters, for some non-negative constant $\phi \geq 0$. ϕ determines how ordered SOS is: for $\phi = 0$, we get full order, while for $\phi > 0$, it is only ordered up to $M \cdot \phi/2$ (all flows that fall into the same volume group are unordered, and each group holds a range of $M \cdot \phi/2$ values). The runtime is, however, $O(1/\phi)$ and is therefore constant for any fixed ϕ. We note that an $\Omega\left(\frac{1}{\epsilon}\right)$ counters lower bound is known [45]. Thus, FAST is asymptotically optimal for constant ϕ. FAST's pseudo code appears in Algorithm 1.

4.2. FAST Analysis

We start by a simple useful observation

Observation 1. *Let $a, b \in \mathbb{N} : a = b \cdot \left\lfloor \frac{a}{b} \right\rfloor + (a \mod b)$.*

For the analysis, we use the following notations: for every item $x \in \mathcal{U}$ and stream length t, we denote by $q_t(x)$ the value of Query(x) after seeing t elements. We slightly abuse the notation and refer to t also as the *time* at which the t^{th} element arrived, where time here is discrete. We denote by C_t the set of elements with an allocated counter at time t, by $r_{x,t}$ the value of r_x and by $c_{x,t}$ the value of c_x. Also, we denote the volume at time t as $v_{x,t} \triangleq \sum_{\substack{i \in \{1,\ldots,t\}: \\ d_i = x}} w_i$.

We now show that FAST has a one-sided error.

Lemma 2. *For any $t \in \mathbb{N}$, after seeing any $(\mathcal{U}, M)$-weighted stream $\mathcal{S}$ of length t, for any $x \in \mathcal{U} : v_x \leq \widehat{v_x}$.*

Proof. We prove $v_{x,t} \leq q_t(x)$ by induction over t.
Basis: $t = 0$. Here, we have $v_{x,t} = 0 = q_t(x)$.
Hypothesis: $v_{x,t-1} \leq q_{t-1}(x)$.
Step: $\langle x_t, w_t \rangle$ arrives at time t. By case analysis:

Consider the case where the queried item x is not the arriving one (i.e., $x \neq x_t$). In this case, we have $v_{x,t} = v_{x,t-1}$. If $x \in C_{t-1}$ but was evicted (Line 10) then $c_x \in \text{argmin}_{y \in C_{t-1}}(c_{y,t-1})$. This means that:

$$\begin{aligned} q_{t-1}(x) = r_{x,t-1} + s \cdot \text{argmin}_{y \in C_{t-1}}(c_{y,t-1}) \\ \leq s - 1 + s \cdot \text{argmin}_{y \in C_t}(c_{y,t}) = q_t(x), \end{aligned}$$

where the last equation follows from the query for $x \notin C_t$ (Line 15). Next, if $x \in C_{t-1}$ and $x \in C_t$, its estimated volume is determined by Line 13 and we get $q_t(x) = q_{t-1}(x) \geq v_{x,t-1} = v_{x,t}$. If $x \notin C_{t-1}$ then $x \notin C_t$, so the values of $q_t(x), q_{t-1}(x)$ are determined by line 15. Since the value of $\min_{y \in C} c_y$ can only increase over time, we have $q_t(x) \geq q_{t-1}(x) \geq v_{x,t}$ and the claim holds.

On the other hand, assume that we are queried about the last item, i.e., $x = x_t$. In this case, we get $v_{x,t} = v_{x,t-1} + w_t$. We consider the following cases: First, if $x \in C_{t-1}$, then $q_t(x) = q_{t-1}(x) + w_t$. Using the hypothesis, we conclude that $v_{x,t} = v_{x,t-1} + w_t \leq q_{t-1}(x) + w_t = q_t(x)$ as required. Next, if $|C_{t-1}| < \mathfrak{C}$, we also have $q_t(x) = q_{t-1}(x) + w_t$ and the above analysis holds. Finally, if $x \notin C_{t-1}$ and $|C_{t-1}| = \mathfrak{C}$, then

$$q_{t-1}(x) = s - 1 + s \cdot \min_{y \in C_{t-1}} c_{y,t-1}. \tag{1}$$

On the other hand, when x arrives, the condition of Line 2 was not satisfied, and thus

$$\begin{aligned} q_t(x) &= r_{x,t} + s \cdot c_{x,t} = (s-1+w) \mod s \\ &\quad + s \cdot \left(\min_{y \in C_{t-1}} c_{y,t-1} + \left\lfloor \tfrac{s-1+w}{s} \right\rfloor\right) \\ \text{(Observation 1)} \quad &= s \cdot \min_{y \in C_{t-1}} c_{y,t-1} + s - 1 + w \\ \text{(1)} \quad &= q_{t-1}(x) + w \\ \binom{\text{induction}}{\text{hypothesis}} \quad &\geq v_{x,t-1} + w = v_{x,t}. \end{aligned}$$ □

We continue by showing that FAST is accurate if there are only a few distinct items.

Lemma 3. *If the stream contains at most $\left\lceil \frac{1+\phi}{\epsilon} \right\rceil$ distinct elements then FAST provides an exact estimation of an items volume upon query.*

Proof. Since $|C| \leq \mathfrak{C}$, we get that the conditions in Line 2 and Line 13 are always satisfied. Before the queried element x first appeared, we have $r_x = c_x = 0$ and thus QUERY(x)= 0. Once x appears once, it gets a counter and upon every arrival with value w, the estimation for x exactly increases by w, since x never gets evicted (which can only happen in Line 10). □

We now analyze the sum of counters in C.

Lemma 4. *For any $t \in \mathbb{N}$, after seeing any $(\mathcal{U}, M)$-weighted stream $\mathcal{S}$ of length t, FAST satisfies: $\sum_{x \in C_t}$ QUERY$(x) \leq t \cdot M \cdot (1 + \phi/2)$.*

Proof. We prove the claim by induction on the stream length t.
Basis: $t = 0$. In this case, all counters have value of 0 and thus $\sum_{x \in C_t} q_t(x) = 0 = t \cdot (M \cdot (1 + \phi/2))$.
Hypothesis: $\sum_{x \in C_{t-1}} q_{t-1}(x) \leq (t-1) \cdot M \cdot (1 + \phi/2)$.
Step: $\langle x_t, w_t \rangle$ arrives at time t. We consider the following cases:

1. $x \in C_{t-1}$ or $|C_{t-1}| < \left\lceil \frac{1+\phi}{\epsilon} \right\rceil$. In this case, the condition in Line 2 is satisfied and thus $c_{x,t} = c_{x,t-1} + \left\lfloor \frac{r_{x,t-1}+w}{s} \right\rfloor$ (Line 3) and $r_{x,t} = (r_{x,t-1} + w) \mod s$ (Line 4). By Observation 1 we get

$$\begin{aligned} q_t(x) &=_{\binom{\text{by line}}{13}} r_{x,t} + s \cdot c_{x,t} \\ &= c_{x,t-1} + \left\lfloor \frac{r_{x,t-1} + w}{s} \right\rfloor + (r_{x,t-1} + w) \mod s \\ &= w + c_{x,t-1} + r_{x,t-1} = q_{t-1}(x) + w. \end{aligned} \tag{2}$$

Since the value of a query for every $y \in C_t \setminus \{x\}$ remains unchanged, we get that

$$\begin{aligned}
\sum_{y \in C_t} q_t(y) &= q_t(x) + \sum_{\substack{y \in C_{t-1} \\ y \neq x}} q_{t-1}(y) \\
\text{(by (3))} \quad &= w + q_{t-1}(x) + \sum_{\substack{y \in C_{t-1} \\ y \neq x}} q_{t-1}(y) \\
&= w + \sum_{y \in C_{t-1}} q_{t-1}(y) \\
\binom{\text{induction}}{\text{hypothesis}} \quad &\leq w + (t-1) \cdot (M \cdot (1 + \phi/2)) \\
&\leq M + (t-1) \cdot (M \cdot (1 + \phi/2)) \\
(\phi \geq 0) \quad &\leq t \cdot (M \cdot (1 + \phi/2)) .
\end{aligned}$$

2. $x \notin C_{t-1}$ and $|C_{t-1}| = \left\lceil \frac{1+\phi}{\epsilon} \right\rceil$. In this case, the condition of Line 2 is false and therefore $c_{x,t} = c_{m,t-1} + \left\lfloor \frac{s-1+w}{s} \right\rfloor$ (Line 8) and $r_{x,t} \leftarrow (s - 1 + w) \mod s$ (Line 9). From Observation 1 we get that

$$\begin{aligned}
q_t(x) &=_{\binom{\text{by Line}}{13}} r_{x,t} + s \cdot c_{x,t} \\
&= c_{m,t-1} + \left\lfloor \frac{s-1+w}{s} \right\rfloor + (s - 1 + w) \mod s \\
&= w + c_{m,t-1} + s - 1 \\
&= q_{t-1}(m) - r_{m,t-1} + \left\lfloor \frac{M\phi}{2} \right\rfloor + w \\
&\leq q_{t-1}(m) + \left\lfloor \frac{M\phi}{2} \right\rfloor + w. \qquad (3)
\end{aligned}$$

As before, the value of a query for every $y \in C_t \setminus \{x\}$ is unchanged, and since $C_{t-1} \setminus C_t = \{m\}$,

$$\begin{aligned}
\sum_{y \in C_t} q_t(y) \quad &= q_t(x) - q_{t-1}(m) + \textstyle\sum_{y \in C_{t-1}} q_{t-1}(y) \\
\text{(by (3))} \quad &\leq \left\lfloor \tfrac{M\phi}{2} \right\rfloor + w + \textstyle\sum_{y \in C_{t-1}} q_{t-1}(y) \\
\binom{\text{induction}}{\text{hypothesis}} \quad &\leq \left\lfloor \tfrac{M\phi}{2} \right\rfloor + w + (t-1) \cdot (M \cdot (1 + \phi/2)) \\
&\leq \left\lfloor \tfrac{M\phi}{2} \right\rfloor + M + (t-1) \cdot (M \cdot (1 + \phi/2)) \\
(\phi \geq 0) \quad &\leq t \cdot (M \cdot (1 + \phi/2)) .
\end{aligned}$$

□

Next, we show a bound on FAST's estimation error.

Lemma 5. *For any $t \in \mathbb{N}$, after seeing any $(\mathcal{U}, M)$-weighted stream $\mathcal{S}$ of length t, for any $x \in \mathcal{U} : \widehat{v_x} \leq v_x + t \cdot M \cdot \epsilon$.*

Proof. First, consider the case where the stream contains at most $\left\lceil \frac{1+\phi}{\epsilon} \right\rceil$ distinct elements. By Lemma 3, $\widehat{v_x} \leq v_x$ and the claim holds. Otherwise, we have seen more than $\left\lceil \frac{1+\phi}{\epsilon} \right\rceil$ distinct elements, and specifically

$$t > \left\lceil \frac{1+\phi}{\epsilon} \right\rceil. \tag{4}$$

From Lemma 4, it follows that

$$\min_{y \in C_t} Query(y) \leq \frac{t \cdot M \cdot (1+\phi/2)}{\left\lceil \frac{1+\phi}{\epsilon} \right\rceil} \leq \frac{t \cdot M \cdot \epsilon \cdot (1+\phi/2)}{1+\phi}. \tag{5}$$

Notice that $\forall x \in C_t$, the value of $\textsc{Query}(x)$ is determined in Line 13; that is, $q_t(x) = r_{x,t} + s \cdot c_{x,t}$. Next, observe that an item's remainder value is bounded by $s-1$ (Line 4 and Line 9). Thus,

$$\forall x, y \in C_t : q_t(x) \geq s + q_t(y) \implies c_{x,t} > c_{y,t}. \tag{6}$$

By choosing $y \in \arg\min_{y \in C_t} q_t(y)$, we get that if $v_{x,t} \geq q_t(y) + s$, then $q_t(x) \geq q_t(y) + s$ and thus $c_{x,t} > c_{y,t}$. Next, we show that if $v_{x,t} \geq t \cdot M \cdot \epsilon$, then $c_x > \min_{y \in C_t} c_y$ and thus x will never be the "victim" in Line 7:

$$\begin{aligned} q_t(x) \geq v_{x,t} \geq t \cdot M \cdot \epsilon = t \cdot M \cdot \epsilon \cdot \frac{1+\phi/2}{1+\phi} + M\phi/2 \cdot \frac{t}{\frac{1+\phi}{\epsilon}} \\ \underset{(5)}{\geq} q_t(y) + M\phi/2 \cdot \frac{t}{\frac{1+\phi}{\epsilon}} \underset{(4)}{>} q_t(y) + M\phi/2. \end{aligned}$$

Next, since $q_t(x)$ and $q_t(y)$ are integers, it follows that $q_t(x) \geq q_t(y) + \left\lfloor \frac{M \cdot \phi}{2} + 1 \right\rfloor = q_t(y) + s$. Finally, we apply (6) to conclude that once x arrives with a cumulative volume of $t \cdot M \cdot \epsilon$, it will never be evicted (Line 10) and from that moment on its volume will be measured exactly. $\square$

Next, we prove a bound on the run time of FAST.

Lemma 6. *let $\phi > 0$, FAST adds in $O\left(\frac{1}{\phi}\right)$ time.*

Proof. As mentioned before, FAST utilizes the SOS data structure that answers queries in $O(1)$. Updates are a bit more complex as we need to handle weights and thus may be required to move the flow more than once, upon a counter increase. Whenever we wish to increase the value of a counter (Line 3 and Line 8), we need to remove the item from its current group and place it in a group that has the increased c value. This means that for increasing a counter by $n \in \mathbb{N}$, we have to traverse at most n groups until

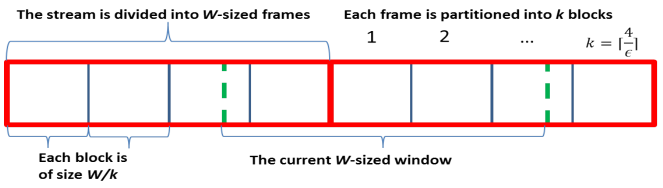

The stream is divided into W-sized frames
Each frame is partitioned into k blocks
1
2
...
$k = \lceil \frac{4}{\epsilon} \rceil$
Each block is of size W/k
The current W-sized window

Figure 2: The stream is divided into intervals of size W called *frames* and each frame is partitioned into k equal-sized *blocks*. The window of interest is also of size W, and overlaps with at most 2 frames and $k+1$ blocks.

we find the correct location. Since the remainder value is at most $s-1$ (Line 4 and Line 9), we get that at any time point, a counter is increased by no more than $\left\lfloor \frac{s-1+w}{s} \right\rfloor$ (Line 3 and Line 8). Finally, since $s = \left\lfloor \frac{M\cdot\phi}{2} + 1 \right\rfloor$, we get that the counter increase is bounded by $\left\lfloor \frac{\left\lfloor \frac{M\cdot\phi}{2}+1 \right\rfloor -1+w}{\left\lfloor \frac{M\cdot\phi}{2}+1 \right\rfloor} \right\rfloor < 1 + \frac{w}{\frac{M\phi}{2}} \leq 1 + \frac{2}{\phi} = O\left(\frac{1}{\phi}\right)$. □

Next, we combine Lemma 2, Lemma 5 and Lemma 6 to conclude the correctness of the FAST algorithm.

Theorem 7. *For any fixed $\phi > 0$, when allocated with $\mathfrak{C} \triangleq \left\lceil \frac{1+\phi}{\epsilon} \right\rceil$ counters, FAST performs updates and queries in constant time, and solves the (ϵ, M)-*Volume Estimation *problem.*

Finally, FAST also solves the heavy hitters problem:

Theorem 8. *For any fixed $\phi > 0$, when allocated with $\mathfrak{C} \triangleq \left\lceil \frac{1+\phi}{\epsilon} \right\rceil$* counters, by returning $\{x \in \mathcal{U} \mid \widehat{v_x} \geq N \cdot M \cdot \theta\}$, FAST solves the (θ, ϵ, M)-Weighted Heavy Hitters problem.

5. Windowed FAST (WFAST)

We now present *Windowed Frequent items Algorithm with a Semi-structured Table (WFAST)*, an efficient algorithm for the (W, ϵ, M)-Volume Estimation and (W, θ, ϵ, M)-Weighted Heavy Hitters problems.

We partition the stream into consecutive sequences of size W called *frames*. Each frame is further divided into $k \triangleq \left\lceil \frac{4}{\epsilon} \right\rceil$ *blocks*, each of size $\frac{W}{k}$, which we assume is an integer for simplicity. Figure 2 illustrates the setting.

WFAST uses a FAST instance y to estimate the volume of each flow within the current frame. Once a frame ends (the stream length is divisible

Symbol	Meaning
k	A constant $k \triangleq \lceil 4/\varepsilon \rceil$
y	A FAST instance using $k(1+\phi)$ counters.
b	A queue of $k+1$ queues. An efficient implementation appears in [5].
B	The histogram of b, implemented using a hash table.
o	The offset within the current frame.

Table 2: Variables used by the WFAST algorithm.

by W), we "flush" the instance, i.e., reset all counters and remainders to 0. Yet, we do not "forget" all information in a flush, as high volume flows are stored in a dedicated data structure. Specifically, we say that an element x *overflowed* at time t if $\left\lfloor \frac{q_{x,t}}{MW/k} \right\rfloor > \left\lfloor \frac{q_{x,t-1}}{MW/k} \right\rfloor$. Intuitively, this means that the rounding of the estimation to the measurement threshold at time t was changed. We use a queue of queues structure b to keep track of which elements have overflowed in each block. That is, each node of the main queue represents a block and contains a queue of all elements that overflowed in its block. Particularly, the secondary queues maintain the ids of overflowing elements. Once a block ends, we remove the oldest block's node (queue) from the main queue, and initialize a new queue for the starting block. Finally, we answer queries about the window volume of an item x by multiplying its overflows count by MW/k, adding the residual count from y (i.e., the part that is not recorded in b), plus $2MW/k$ to ensure an overestimation.

For $O(1)$ time queries, we also maintain a hash table B that tracks the overflow count for each item. That is, for each element x, $B[x]$ contains the number of times x is recorded in b. Since multiple items may overflow in the same block, we cannot update B once a block ends in constant time. We address this issue by *deamortizing* B's update, and on each arrival we remove a *single* item from the queue of the oldest block (if such exists). The pseudo code of WFAST appears in Algorithm 2 and a list containing its variables description appears in Table 2. An efficient implementation of the queue of queues b is described in [5].

5.1. WFAST Analysis

We start by introducing several notations to be used in this section. We mark the queried element by x, the current time by $W+o$, and assume that item W is the first element of the current frame. For convenience, denote $v_x(t_1,t_2) \triangleq \sum_{\substack{i\in\{t_1,\ldots,t_2\}:\\ x_i=x}} w_i$, i.e., the volume of x between t_1 and t_2. The goal is then to approximate the window volume of x, which is defined as

$$v_x^w \triangleq v(o+1, W+o) = v(o+1, W-1) + v(W, W+o), \tag{7}$$

Algorithm 2 WFAST (W, M, ϕ)

```
    Initialization: $y \leftarrow FAST(M, 1/k, \phi), o \leftarrow 0, B \leftarrow$ Empty hash table,
    $b \leftarrow$ Queue of $k+1$ empty queues.
 1: function ADD(Item $x$, Weight $w$)
 2:     $o \leftarrow o + 1 \mod W$
 3:     if $o = 0$ then                                        ▷ new frame starts
 4:         $y$.FLUSH()
 5:     if $o \mod \frac{W}{k} = 0$ then                       ▷ new block
 6:         $b$.POP()
 7:         $b$.APPEND(new empty queue)
 8:     if $b$.tail is not empty then                          ▷ remove oldest item
 9:         $oldID \leftarrow b$.tail.POP()
10:         $B[oldID] \leftarrow B[oldID] - 1$
11:         if $B[oldID] = 0$ then
12:             $B$.REMOVE($oldID$)
13:     $prevOverflowCount \leftarrow \left\lfloor \frac{y.\text{QUERY}(x)}{MW/k} \right\rfloor$
14:     $y$.ADD($x, w$)                                        ▷ add item
15:     if $\left\lfloor \frac{y.\text{QUERY}(x)}{MW/k} \right\rfloor > prevOverflowCount$ then   ▷ overflow
16:         $b$.head.PUSH($x$)
17:         if B.CONTAINS(x) then
18:             $B[x] \leftarrow B[x] + 1$
19:         else
20:             $B[x] \leftarrow 1$                            ▷ adding $x$ to $B$
21: function WINQUERY(Item x)
22:     if $B$.CONTAINS($x$) then
23:         return $MW/k \cdot (B[x] + 2) + (y.\text{QUERY}(x) \mod MW/k)$
24:     else                                                   ▷ $x$ has no overflows
25:         return $2MW/k$+y.QUERY($x$)
```

i.e., the sum of weights in the timestamps within $\langle o+1, o+2, \ldots, W+o \rangle$ in which x arrived. We denote the value returned from $y.\text{QUERY}(x)$ after the t'th item was added by y_t. Similarly, u_t represents whether x arrived at time t $(x = x_t)$ *and* overflowed, i.e., the condition of Line 15 was satisfied. We assume that if x is not allocated a FAST counter at time t, then $B[x] = 0$, which allows us to consider Line 23 for queries. For simplicity, we mark $B[x] = 0$ for $x \notin B$.

We proceed with a useful lemma that bounds WFAST's error on arrivals happening before the flush (Line 4).

Lemma 9. *Let $t_1, t_2 \in \{o+1, \ldots + W - 1\}$ be two timestamps within the previous frame, then* $\left\lfloor \frac{v_x(t_1,t_2)}{MW/k} \right\rfloor \le \sum_{\tau=t_1}^{t_2} u_\tau \le \left\lceil \frac{v_x(t_1,t_2)}{MW/k} \right\rceil$.

Proof. Since y, initialized with $\epsilon = \frac{1}{k}$, is flushed every W elements (Line 4), any element z that satisfies $y.\text{QUERY}(z) > MW/k$ is guaranteed not to lose its counter (see Lemma 5's proof for a similar analysis). This means that once an item overflows, it never loses its counter. Further, being an overestimator, an element with a volume of MW/k (or more) is guaranteed to have a counter since the minimal counter cannot exceed MW/k. Thus, if $v_x(t_1, t_2) < MW/k$ then the claim holds, since after overflowing for the first time, an item has to arrive with a weight of MW/k to overflow again. On the other hand, if $v_x(t_1, t_2) > MW/k$ then x may overflow for the first time before its volume reached MW/k, but from that point on only arrivals of x increase the counter and can cause an overflow. □

We continue with proving the algorithm's correctness.

Theorem 10. *Algorithm 2 solves* (W, ϵ, M)-Volume Estimation.

Proof. We prove the theorem in two steps. We first analyze the volume of x within the previous frame, and then consider the current one. We continue by bounding the error introduced by the deamortization and factor FAST being an approximation algorithm in the first place. Finally, we add up the different error types and show that WFAST provides a decent approximation.

We start with the number of times x has overflowed before the flush. By applying Lemma 9, we get that

$$\left\lfloor \frac{v_x(o+1, W-1)}{MW/k} \right\rfloor \leq \sum_{t=M+1}^{W-1} u_t \leq \left\lceil \frac{v_x(o+1, W-1)}{MW/k} \right\rceil . \quad (8)$$

We continue with analyzing the current frame, which started after y was last flushed (Line 4). As discussed above, an element whose volume is larger than MW/k overflows and will not lose its counter in the flush, thus:

$$y_{W+o} = MW/k \cdot \sum_{t=W}^{W+o} u_t + \left(y_{W+o} \mod MW/k\right). \quad (9)$$

Notice that if x does not have a counter, it did not overflow and the equation still holds.

Next, we consider the number of overflows recorded in $B[x]$ and the number of actual overflows. We have deamortized (Line 9) the process of updating the overflow count (in B). This means that $B[x]$ is not guaranteed to have the exact count of the number of times x overflowed within the blocks overlapping with the current window. Luckily, since x cannot overflow twice in the same block, and specifically in the oldest block ($b.tail$), we get that we underestimate the number of overflows by at most one, and specifically:

$$\sum_{t=o+1}^{W+o} u_t - 1 \leq B[x] \leq \sum_{t=o+1}^{W+o} u_t. \quad (10)$$

Since y is a FAST instance with parameters $(M, \frac{1}{k}, \phi)$, it solves the (ϵ, M)-Volume Estimation problem, thus

$$v(W, W+o) \leq y_{W+o} \leq v(W, W+o) + MW/k. \quad (11)$$

When queried for x, the algorithm returns

$$\begin{aligned}\widehat{v_x^W} &= MW/k \cdot (B[x]+2) + (y_{W+o} \mod MW/k)\\ &=_{(9)} MW/k \cdot \left(B[x]+2-\sum\nolimits_{t=W}^{W+o} u_t\right) + y_{W+o}.\end{aligned}$$

We combine the above inequalities to bound the overestimation:

$$\begin{aligned}\widehat{v_x^W} &= MW/k \cdot \left(B[x]+2-\sum\nolimits_{t=W}^{W+o} u_t\right) + y_{W+o}\\ &\le_{(11)} MW/k \cdot \left(B[x]+3-\sum\nolimits_{t=W}^{W+o} u_t\right) + v(W, W+o)\\ &\le_{(10)} MW/k \cdot \left(\sum\nolimits_{t=o+1}^{W+o} u_t + 3 - \sum\nolimits_{t=W}^{W+o} u_t\right) + v(W, W+o)\\ &= MW/k \cdot \left(\sum\nolimits_{t=o+1}^{W-1} u_t + 3\right) + v(W, W+o)\\ &\le_{(Lemma\ 9)} MW/k \cdot \left(\left\lceil \frac{v_x(o+1, W-1)}{MW/k} \right\rceil + 3\right) + v(W, W+o)\\ &\le_{(7)} v(o+1, W+o) + 4MW/k \le v_x^W + WM\epsilon.\end{aligned}$$

Similarly, we bound the query value from below:

$$\begin{aligned}\widehat{v_x^W} &= MW/k \cdot \left(B[x]+2-\sum\nolimits_{t=W}^{W+o} u_t\right) + y_{W+o}\\ &\ge_{(11)} MW/k \cdot \left(B[x]+2-\sum\nolimits_{t=W}^{W+o} u_t\right) + v(W, W+o)\\ &\ge_{(10)} MW/k \cdot \left(\sum\nolimits_{t=o+1}^{W+o} u_t + 1 - \sum\nolimits_{t=W}^{W+o} u_t\right) + v(W, W+o)\\ &= MW/k \cdot \left(\sum\nolimits_{t=o+1}^{W-1} u_t + 1\right) + v(W, W+o)\\ &\ge_{(Lemma\ 9)} MW/k \cdot \left(\left\lfloor \frac{v_x(o+1, W-1)}{MW/k} \right\rfloor + 1\right) + v(W, W+o)\\ &\ge_{(7)} v(o+1, W+o) = v_x^W.\end{aligned}$$

Showing both bounds, we established that WFAST solves the (W, ϵ, M)-FREQUENCY ESTIMATION problem. □

As a corollary, Algorithm 2 can find heavy hitters.

Theorem 11. *By returning all items $x \in \mathcal{U}$ for which $\widehat{v_x^W} \ge MW\theta$, Algorithm 2 solves* (W, θ, ϵ, M)*-WEIGHTED HEAVY HITTERS.*

WFAST runtime analysis:

As listed in the pseudo code of WFAST (see Algorithm 2) and the description above, processing new elements requires adding them to the FAST instance y, which takes $O(\frac{1}{\phi})$ time, and another $O(1)$ operations. The query processing includes $O(1)$ operations and hash tables accesses. If one is interested in finding the heavy hitters efficiently in addition to per-flow volume queries, we slightly modify the algorithm. Instead of keeping one instance of y, we keep two such that we always have the FAST instance of the last two frames. We note that this at most doubles the space and does not affect the runtime, as only a single instance is updated per packet.

Given a query, we then need to go only over the $O(\frac{1}{\epsilon})$ counters stored in both instances to find the heavy hitters. This is because every flow that has weight at least $MW\epsilon/4$ times during a frame is guaranteed to have a counter from that point and until the end of the frame, and every flow with weight $MW\theta \geq MW\epsilon > 2(MW\epsilon/4)$ is guaranteed to have a weight of at least $MW\epsilon/4$ in at least one of the frames that overlaps with the window. In summary, we get the following theorem:

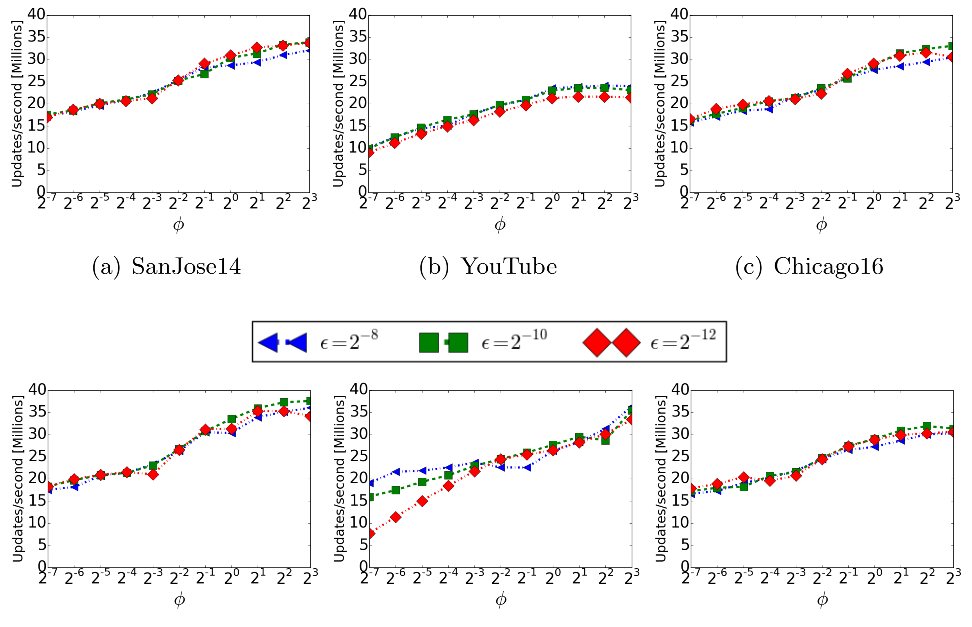
Theorem 12. *For any fixed $\phi > 0$, WFAST processes new elements and answers window-volume queries in constant time, while finding the window's weighted heavy hitters in $O(\frac{1}{\epsilon})$ time.*

6. Hierarchical Heavy Hitters

Hierarchical heavy hitters (HHH) algorithms treat IP addresses as a hierarchical domain. At the bottom are *fully specified* IP addresses such as $p_0 = 101.102.103.104$. Higher layers include shorter and shorter prefixes of the fully specified addresses. For example, $p_1 = 101.102.103.*$ and $p_2 = 101.102.*$ are level 1 and level 2 prefixes of p_0, respectively. Such prefixes *generalize* an IP address. In this example, $p_0 \prec p_1 \prec p_2$, indicating that p_0 satisfies the pattern of p_1, and any IP address that satisfies p_1 also satisfies p_2. The above example refers to a *single dimension* (e.g., the source IP), and can be generalized to *multiple dimensions* (e.g., pairs of source IP and destination IP). HHH algorithms need to find the heavy hitter prefixes at each level of the induced hierarchy. For example, this enables identifying heavy hitters subnets, which may be suspected of generating a DDoS attack. The problem is formally defined in [46, 20].

Hierarchical Fast (HFAST)

Hierarchical FAST (HFAST) is derived from the algorithm of [46]. Specifically, the work of [46] suggests *Hierarchical Space Saving with a Heap(HSSH)*. In their work, the HHH prefixes are distilled from multiple solutions of plain heavy hitter problems. That is, each prefix pattern has its own separate



(a) SanJose14 (b) YouTube (c) Chicago16

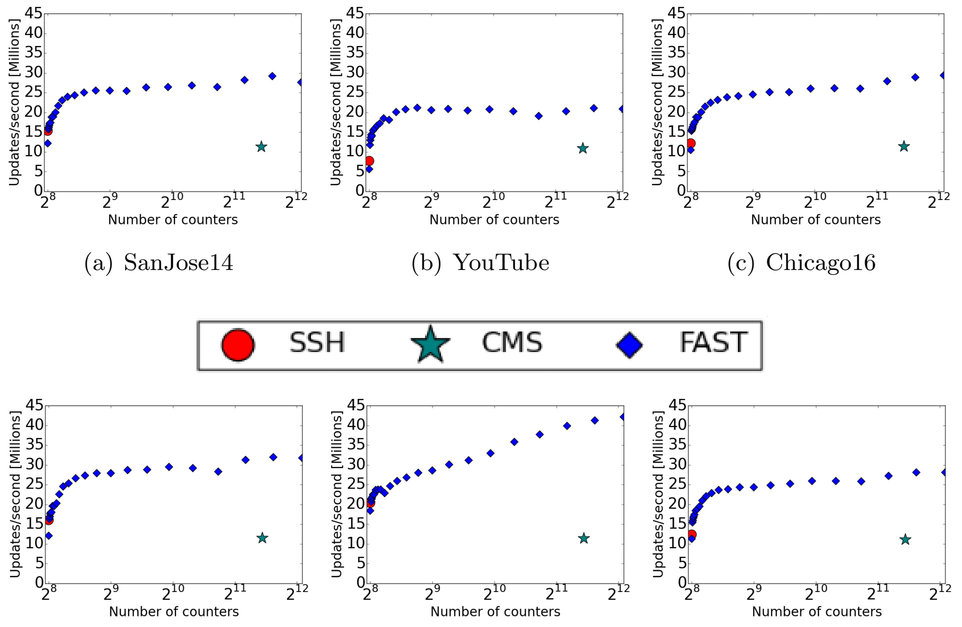
(d) SanJose13 (e) DC1 (f) Chicago15

Figure 3: The effect of parameter ϕ on operation speed for different error guarantees (ϵ). ϕ influences the space requirement as the algorithm is allocated with $\left\lceil \frac{1+\phi}{\epsilon} \right\rceil$ counters.

heavy hitters algorithm that is updated on each packet arrival. For example, consider a packet whose source IP address is 101.102.103.104 where the (one dimensional) HHH measurements are carried according to source addresses. In this case, the packet arrival is translated into the following five heavy hitters update operations: 101.102.103.104, 101.102.103.$*$, 101.102.$*$, 101.$*$, and $*$. Finally, HHHs are identified by calculating the heavy hitters with each separate heavy hitters algorithm.

HFAST is derived by replacing the underlying heavy hitters algorithm in [46] from Space Saving with heap [45] to FAST. This asymptotically improves the update complexity from $O\left(H \log\left(\frac{1}{\epsilon}\right)\right)$ to $O\left(H\right)$, where H is the size of the hierarchy. Since the analysis of [46] is indifferent to the internal implementation of the heavy hitters algorithm, no further analysis is required for HFAST.

Finally, we note that a hierarchical heavy hitters algorithm on sliding windows can be constructed using the work of [46] by replacing each space saving instance with our WFAST. The complexity of the proposed algorithm is $O\left(\frac{H}{\epsilon}\right)$ space and $O\left(H\right)$ update time. To our knowledge, there is no prior



(a) SanJose14 (b) YouTube (c) Chicago16

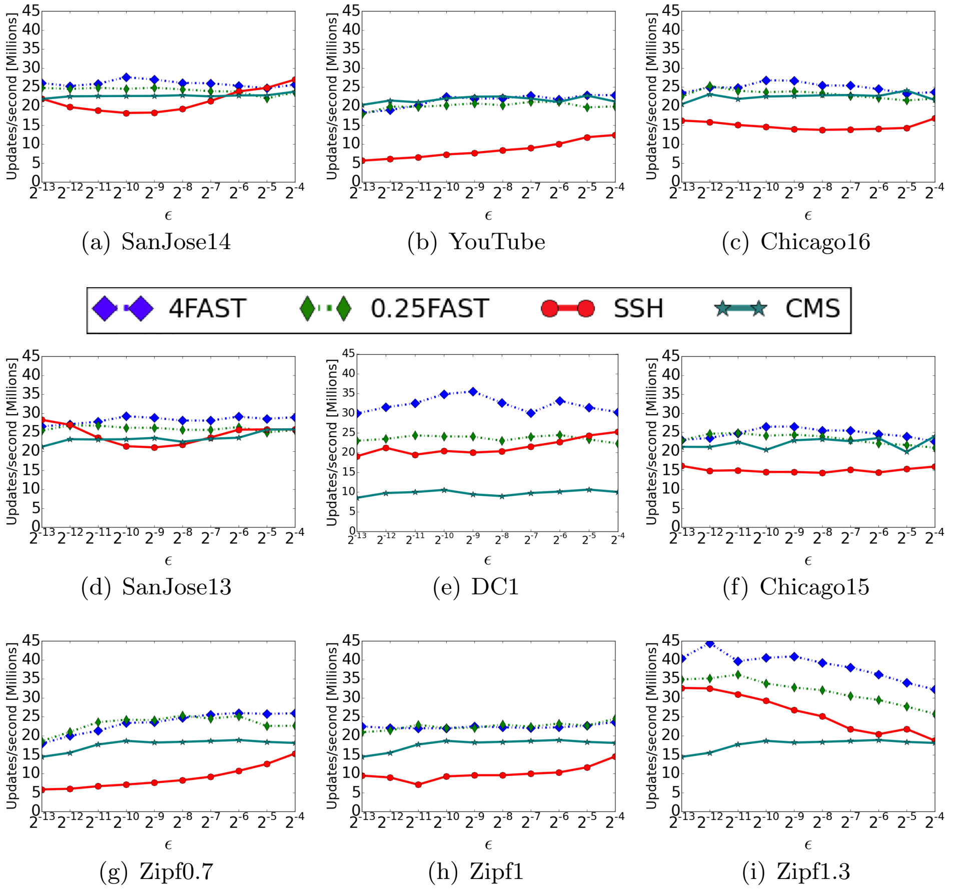
(d) SanJose13 (e) DC1 (f) Chicago15

Figure 4: Runtime comparison for a error guarantee od $\epsilon = 2^{-8}$. All algorithms provide the same guarantee, FAST uses different ϕ values.

work for this problem.

7. Evaluation

Our evaluation is performed on an Intel i7-5500U CPU with a clock speed of 2.4GHz, 16 GB RAM and a Windows 8.1 operating system. We implemented FAST in C++ and released our library as an open source project [4]. Our code is based on the open source library of Cormode, and Hadjieleftheriou [17] which implemented the linked-lists version of the Space Saving in C. We generalized the implementation to our SOS data structure by implementing volume groups and remainders (see Section 4). The code was also converted to C++, which allowed us to further optimize its throughput due to more efficient compilation. We compared to the following algorithms:
Count Min Sketch (CMS) [21] – a sketch based solution that can only solve the volume estimation problem.
Space Saving Heap (SSH) – a heap based implementation [17] of Space Saving [45] that has a logarithmic runtime complexity.



(a) SanJose14 (b) YouTube (c) Chicago16

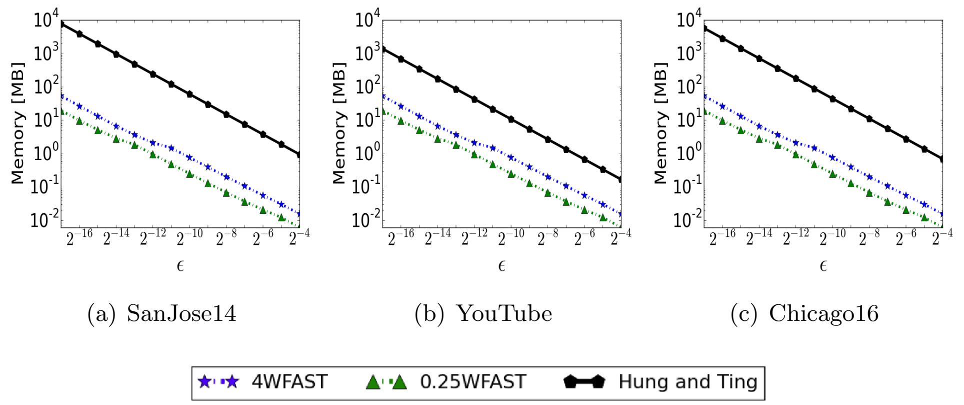
(d) SanJose13 (e) DC1 (f) Chicago15

(g) Zipf0.7 (h) Zipf1 (i) Zipf1.3

Figure 5: Runtime as a function of accuracy guarantee (ϵ) provided by the algorithms.

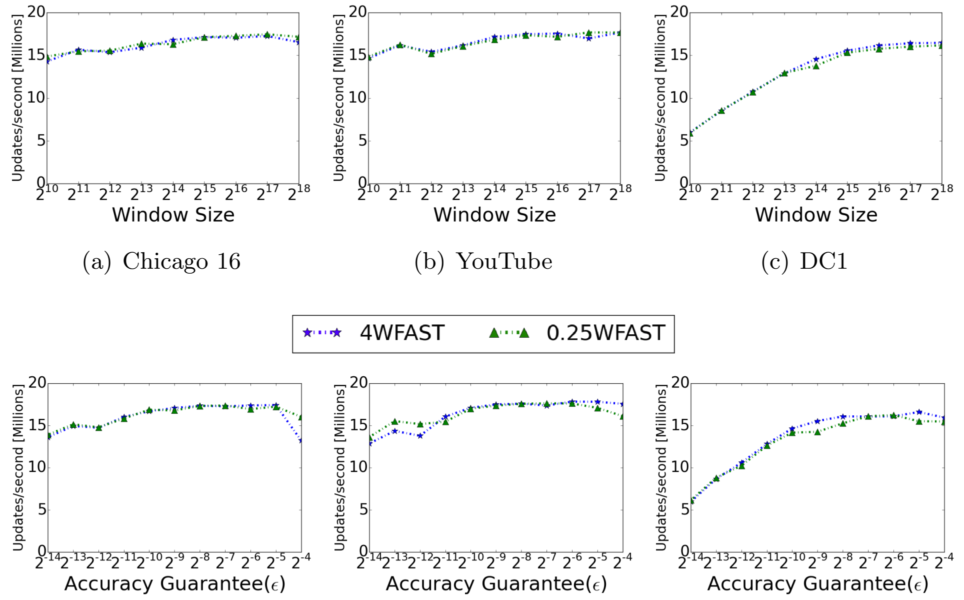
Hierarchical Space Saving Heap (HSSH) – an HHH algorithm [46] that uses SSH as a building block and operates in $O(H \log(\frac{1}{\varepsilon}))$ complexity.
Full Ancestry – a trie based HHH algorithm suggested by [20], which operates in $O(H \log \epsilon N)$ complexity.
Partial Ancestry – a trie based HHH algorithm suggested by [20], which operates in $O(H \log \epsilon N)$ complexity and is faster than Full Ancestry.

Related work implementations were taken from open source libraries released by [16] for streams and by [46] for hierarchical heavy hitters. As we have no access to a concrete implementation of a competing sliding window protocol, we compare WFAST to Hung and Ting's algorithm [36] by conservatively estimating the space needed by their approach.



(a) SanJose14 (b) YouTube (c) Chicago16

Figure 6: Space overheads of WFAST compared to previous works. Note that WFAST operates in constant time while the other algorithm requires linear scanning of all counters.



(a) Chicago 16 (b) YouTube (c) DC1

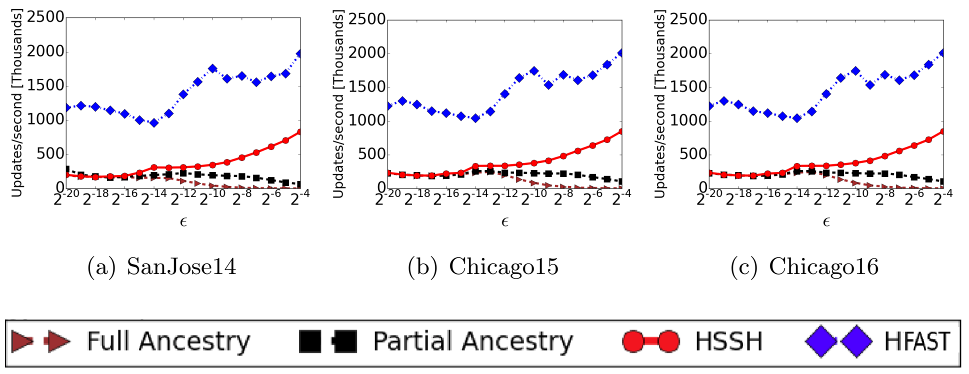
(d) Chicago16 (e) YouTube (f) DC1

Figure 7: WFAST with varying window sizes ($\varepsilon = 2^{-8}$) and varying ε ($W = 2^{16}$).

7.1. Datasets

Our evaluation includes the following publicly available datasets:

1. The CAIDA backbone Internet traces that monitor links in Chicago [33,



(a) SanJose14 (b) Chicago15 (c) Chicago16

Figure 8: Runtime of of HHH algorithms as a function of their accuracy guarantee (ϵ).

Trace	Chicago16	Chicago15	SanJose14	SanJose13	DC1
Date(Y/M/D)	2016/02/18	2015/12/17	2014/06/19	2013/12/19	2010
#Packets	97M	85M	112M	97M	7.3M
Total Volume	94GB	80GB	149GB	110GB	6.1GB
Mean Size	1046B	1013B	1424B	1255B	894B
Max Size	49458B	64134B	65535B	65528B	1476B
Large Packets	0.34%	0.22%	0.78%	0.49%	0%
Large Pkt Traffic	0.5%	0.34%	25.02%	18.81%	0%

Table 3: A summary of key characteristics of the real Internet traces used in this work.

34] and San Jose [31, 32].

2. A datacenter trace from a large university [9].
3. A trace of 436K YouTube video accesses [13]. The weight of a video is its length in seconds.
4. Self generated synthetic traces following a Zipfian distribution with varying skews. A trace of skew X is denoted *ZipfX*. Each trace is unweighed (each element has weight 1) and contains 10M elements.

A summary of key characteristics for CAIDA traces is given in Table 3. As can be seen, the impact of jumbo frames varies between backbone links. Yet, the weight of large packets increases over time in both. In the San Jose link, the number and volume of large packets have increased by 50% within a period of 6 months. For Chicago, large packets are still insignificant, but their number and volume have increased by 50% in two months.

7.2. Experiment Setup

Our experiments are done in the following manner: First the dataset is loaded to the main memory, doing so minimizes the influence of loading the dataset on the throughput evaluation of the algorithms. Then, the tested algorithm is initialized and is sequentially updated for each packet in the trace. We estimate the achieved throughput using the time it takes the algorithm to consume the entire trace. Finally, we minimize noises by averaging each data point with 10 runs.

7.3. Effect of ϕ on Runtime

We begin the evaluation by exploring our trade-off parameter ϕ. Recall that smaller ϕ yields space efficiency while the runtime is proportional to $\frac{1}{\phi}$, i.e, smaller ϕ is expected to result is higher processing time per packet and a slower algorithm. Figure 3 shows runtime performance of FAST as a function of ϕ for three different ε values $(2^{-8}, 2^{-10}, 2^{-12})$. As can be observed, in practice, we indeed get speedup with larger ϕ values. But, we reach a saturation point and increasing ϕ beyond a certain threshold has little impact on performance. It is encouraging that even with small values of ϕ such as 2^{-7}, FAST is still reasonably fast. For the rest of our evaluation, we focus on $\phi = 0.25$ that offers attractive space/time trade off, as well as on $\phi = 4$ that yields higher performance at the expense of more space.

7.4. Speed vs. Space Tradeoff

To explain the tradeoff proposed by FAST, we measured the runtime of the various algorithms for a fixed error guarantee. Here, SSH and CMS are fully determined by the error guarantees (set to $\epsilon = 2^{-8}$) represented by a single measurement point. CMS requires more counters as it uses 10 rows of $\lceil e/\epsilon \rceil$ counters each, while SSH only requires $1/\epsilon$. FAST can provide the same error guarantee for different ϕ values, which affects both runtime and the number of counters. Hence, FAST is represented by a curve. As Figure 4 shows, in all traces, allocating a few additional counters to the $1/\epsilon$ required by SSH allows FAST to achieve higher throughput. Further, on all traces, FAST provides faster throughput than CMS with far fewer counters. While FAST has larger per counter overheads than CMS, its ID to counter mapping allows it to also solve the WEIGHTED HEAVY HITTERS problem.

7.5. Operation Speed Comparison

Figure 5 presents a comparative analysis of the operation speed of previous approaches. Recall that CMS is a probabilistic scheme; we configured it with a failure probability of 0.1%. For FAST, we used two configurations: $\phi = 4$ (4FAST) and $\phi = 0.25$ (0.25FAST).

As can be observed, 4FAST and 0.25FAST are considerably faster than the alternatives in Chicago16 and YouTube. In SanJose14 and SanJose13, SSH is as fast as 4FAST for a large ϵ (small number of counters). Yet, as

ϵ decreases and the number of counters increases, SSH becomes slower due to its logarithmic complexity. In contrast, CMS is almost workload independent. When considering only previous work, in some workloads CMS is faster than SSH, mainly because SSH's performance is workload dependent. The bottom 3 figures (g,h,i) show results for synthetic unweighted Zipf traces with skew parameters of $0.7, 1, 1.3$, respectively. As can be observed, for mildly skewed distributions, CMS is faster than SSH, while for skewed distributions such as when the skew is 1.3, SSH is faster. In all these measurements, 4FAST is faster than the alternatives.

7.6. Sliding Window

We evaluate WFAST compared to Hung and Ting's algorithm [36], which is the only one that supports weighted updates on sliding windows. Figure 6 shows the memory consumption of WFAST with parameters $\phi = 4$ and $\phi = 0.25$ (4WFAST, 0.25FAST) compared to Hung and Ting's algorithm. All algorithms are configured to provide the same worst case error guarantee. As shown, WFAST is up to 100 times more space efficient than Hung and Ting's algorithm. Sadly, we could not obtain an implementation of Hung and Ting's algorithm and thus do not compare its runtime to WFAST. However, WFAST improves their update complexity from $O(\frac{A}{\epsilon})$, where A is the average packet size, to $O(1)$.

Figure 7 shows the operation speed of WFAST for different window sizes and different ε values. As seen, WFAST achieves over 15 million updates per second using a single thread. It is about half as fast as FAST for streams and still within the range of acceptable parameters. There is little dependence in window size and ε with the exception of the DC1 dataset. In this dataset, since the average and maximal packet sizes are similar, the inner working of WFAST causes overflows to be more frequent when ε is close to the window size. Thus, to achieve similar performance as the other traces one needs a sufficiently large window sized.

7.7. Hierarchical Heavy Hitters

In Figure 8, we evaluate the speed of our HFAST compared to the algorithm of [46], which is denoted by HSSH, as well as the Partial Ancestry and Full Ancestry algorithms by [20]. We used the library of [46] for their own HSSH implementation as well as for the Partial Ancestry and Full Ancestry implementations. Since the library was released for Linux, we used a different machine for our HFAST evaluation. Specifically, we used a Dell 730 server running Ubuntu 16.04.01 release. The server has 128GB of RAM and an Intel(R) Xeon(R) CPU E5-2667 v4 @ 3.20GHz processor.

We used two dimensional source/destination hierarchies in byte granularity, where networks IDs are assumed to be 8, 16 or 24 bits long. The weight of each packet is its byte volume, including both the payload size and the header size. As depicted, HFAST is up to 7 times faster than the best alternative and at least 2.4 times faster in every data point.

8. Discussion

In this paper, we presented algorithms for estimating per flow traffic volume in streams, sliding windows and hierarchical domains. We achieved asymptotic and empirical improvements.For streams, FAST processes packets in constant time while being asymptotically space optimal. This is enabled by our novel approach of maintaining only a partial order between counters. An evaluation over real-world traffic traces as well as synthetic ones has yielded a speedup of up to 2.4X compared to previous work.

In the sliding window case, we showed that WFAST works reasonably fast and offers 100x reduction in required space, bringing sliding windows to the realm of possibility. For a given error of $W \cdot M \cdot \epsilon$, WFAST requires $O\left(\frac{1}{\epsilon}\right)$ counters while previous work uses $O\left(\frac{A}{\epsilon}\right)$, where A is the average packet size. Moreover, its update complexity is a constant compared to $O\left(\frac{A}{\epsilon}\right)$in [36].

For hierarchical domains, we presented HFAST that requires $O(\frac{H}{\epsilon})$ space and has $O(H)$ update complexity. This asymptotically improves previous works. Additionally, we demonstrated a speedup of 2.4X-7X on real Internet traces. To our knowledge, there is no prior work on that problem and we plan to examine its possible applications in the future. FAST can be implemented as is in virtual switches such as Open vSwitch and VPP, in these settings the offered speedup directly reduces the measurement overheads. The code of FAST is available as open source [4].

Acknowledgments
We thank the anonymous reviewers for their insightful comments that had significant contribution to this paper. We also thank Yechiel Kimchi for helpful code optimization suggestions. This work was partially funded by the Israeli Science Foundation grant #1505/16 and the Technion-HPI research school.

References

[1] Anderson, D., Bevan, P., Lang, K., Liberty, E., Rhodes, L., and Thaler, J. A high-performance algorithm for identifying frequent items in data streams. In *ACM IMC (2017)*.

[2] Andoni, A., Krauthgamer, R., and Onak, K. Streaming Algorithms via Precision Sampling. In *IEEE FOCS* (2011).

[3] Arasu, A., and Manku, G. S. Approximate counts and quantiles over sliding windows. In *ACM PODS 2004* (2004).

[4] Ben-Basat, R., and Einziger, G. Fast code. available: `https://github.com/ranbenbasat/FAST`.

[5] Ben-Basat, R., Einziger, G., Friedman, R., and Kassner, Y. Heavy Hitters in Streams and Sliding Windows. In *IEEE INFOCOM* (2016).

[6] Ben-Basat, R., Einziger, G., Friedman, R., and Kassner, Y. Randomized admission policy for efficient top-k and frequency estimation. In *IEEE INFOCOM* (2017).

[7] Ben-Basat, R., Einziger, G., Friedman, R., and Kassner, Y. Optimal Elephant Flow Detection. In *IEEE INFOCOM* (2017).

[8] Ben Basat, R., Einziger, G., Friedman, R., Luizelli, M. C., and Waisbard, E. Constant time updates in hierarchical heavy hitters. In *ACM SIGCOMM* (2017).

[9] Benson, T., Akella, A., and Maltz, D. A. Network traffic characteristics of data centers in the wild. In *ACM IMC (2010)*.

[10] Benson, T., Anand, A., Akella, A., and Zhang, M. MicroTE: Fine Grained Traffic Engineering for Data Centers. In *ACM CoNEXT* (2011).

[11] Charikar, M., Chen, K., and Farach-Colton, M. Finding Frequent Items in Data Streams. In *EATCS ICALP* (2002).

[12] Chen, M., and Chen, S. Counter Tree: A Scalable Counter Architecture for Per-Flow Traffic Measurement. In *IEEE ICNP* (2015).

[13] Cheng, X., Dale, C., and Liu, J. Statistics and Social Network of YouTube Videos. In *IWQoS* (2008).

[14] Cho, K. Recursive lattice search: Hierarchical heavy hitters revisited. In *ACM IMC (2017)*.

[15] Choi, B.-Y., Park, J., and Zhang, Z.-L. Adaptive Random Sampling for Load Change Detection. *ACM SIGMETRICS* (2002).

[16] Cormode, G., and Hadjieleftheriou, M. Finding Frequent Items in Data Streams. *VLDB 1*, 2 (2008).

[17] Cormode, G., and Hadjieleftheriou, M. Methods for Finding Frequent Items in Data Streams. *J. VLDB 19*, 1 (2010).

[18] Cormode, G., Korn, F., Muthukrishnan, S., and Srivastava, D. Finding Hierarchical Heavy Hitters in Data Streams. In *VLDB* (2003).

[19] Cormode, G., Korn, F., Muthukrishnan, S., and Srivastava, D. Diamond in the Rough: Finding Hierarchical Heavy Hitters in Multi-dimensional Data. SIGMOD 2004.

[20] Cormode, G., Korn, F., Muthukrishnan, S., and Srivastava, D. Finding Hierarchical Heavy Hitters in Streaming Data. *ACM Trans. Knowl. Discov. Data 1*, 4 (2008).

[21] Cormode, G., and Muthukrishnan, S. An Improved Data Stream Summary: The Count-min Sketch and Its Applications. *J. Algorithms* (2005).

[22] Dimitropoulos, X., Hurley, P., and Kind, A. Probabilistic Lossy Counting: An Efficient Algorithm for Finding Heavy Hitters. *ACM SIGCOMM CCR 38*, 1 (2008).

[23] Dittmann, G., and Herkersdorf, A. Network Processor Load Balancing for High-Speed Links. In *SPECTS* (2002), vol. 735.

[24] Einziger, G., Fellman, B., and Kassner, Y. Independent Counter Estimation Buckets. In *IEEE INFOCOM* (2015).

[25] Einziger, G., and Friedman, R. TinyLFU: A Highly Efficient Cache Admission Policy. In *Euromicro PDP* (2014).

[26] Einziger, G., and Friedman, R. Counting with TinyTable: Every Bit Counts! In *ACM ICDCN* (2016).

[27] Einziger, G., Luizelli, M. C., and Waisbard, E. Constant time weighted frequency estimation for virtual network functionalities. In *IEEE ICCCN 2017*.

[28] Estan, C., Keys, K., Moore, D., and Varghese, G. Building a better NetFlow. *ACM SIGCOMM* (2004).

[29] Garcia-Teodoro, P., Diaz-Verdejo, J. E., Macia-Fernandez, G., and Vazquez, E. Anomaly-Based Network Intrusion Detection: Techniques, Systems and Challenges. *Computers and Security* (2009).

[30] Hershberger, J., Shrivastava, N., Suri, S., and Tóth, C. D. Space Complexity of Hierarchical Heavy Hitters in Multi-dimensional Data Streams. In *ACM PODS* (2005).

[31] Hick, P. CAIDA Anonymized Internet Trace, equinix-sanjose 2013-06-19 13:00-13:05 UTC, Direction B., 2014.

[32] Hick, P. CAIDA Anonymized Internet Trace, equinix-sanjose 2013-12-19 13:00-13:05 UTC, Direction B., 2014.

[33] Hick, P. CAIDA Anonymized Internet Trace, equinix-chicago 2015-12-17 13:00-13:05 UTC, Direction A., 2015.

[34] Hick, P. CAIDA, equinix-chicago 2016-02-18 13:00-13:05 UTC, Direction A., 2016.

[35] Hung, R. Y. S., Lee, L., and Ting, H. Finding frequent items over sliding windows with constant update time. *Inf. Proc. Let.10' 110*, 7.

[36] Hung, R. Y. S., and Ting, H. F. Finding Heavy Hitters over the Sliding Window of a Weighted Data Stream. In *LATIN* (2008).

[37] Kabbani, A., Alizadeh, M., Yasuda, M., Pan, R., and Prabhakar, B. AF-QCN: Approximate Fairness with Quantized Congestion Notification for Multi-tenanted Data Centers. In *IEEE HOTI* (2010).

[38] Karp, R. M., Shenker, S., and Papadimitriou, C. H. A Simple Algorithm for Finding Frequent Elements in Streams and Bags. *ACM Transactions Database Systems* (2003).

[39] Lee, L., and Ting, H. F. A simpler and more efficient deterministic scheme for finding frequent items over sliding windows. In *Proc. of PODS 2006*.

[40] Li, T., Chen, S., and Ling, Y. Per-Flow Traffic Measurement Through Randomized Counter Sharing. *IEEE/ACM Trans. on Networking* (2012).

[41] Lin, Y., and Liu, H. Separator: Sifting Hierarchical Heavy Hitters Accurately from Data Streams. In *ADMA* (2007).

[42] Lu, Y., Montanari, A., Prabhakar, B., Dharmapurikar, S., and Kabbani, A. Counter Braids: a Novel Counter Architecture for Per-Flow Measurement. In *ACM SIGMETRICS* (2008).

[43] Manerikar, N., and Palpanas, T. Frequent Items in Streaming Data: An Experimental Evaluation of the State-of-the-Art. *Data Knowl. Eng.* (2009).

[44] Manku, G. S., and Motwani, R. Approximate Frequency Counts over Data Streams. In *VLDB* (2002).

[45] Metwally, A., Agrawal, D., and Abbadi, A. E. Efficient Computation of Frequent and Top-k Elements in Data Streams. In *IN ICDT* (2005).

[46] Mitzenmacher, M., Steinke, T., and Thaler, J. Hierarchical Heavy Hitters with the Space Saving Algorithm. In *ALENEX* (2012).

[47] Mukherjee, B., Heberlein, L., and Levitt, K. Network Intrusion Detection. *Network, IEEE 8*, 3 (1994).

[48] Ramabhadran, S., and Varghese, G. Efficient Implementation of a Statistics Counter Architecture. *ACM SIGMETRICS* (2003).

[49] Sekar, V., Duffield, N., Spatscheck, O., van der Merwe, J., and Zhang, H. LADS: Large-scale Automated DDOS Detection System. In *USENIX ATEC* (2006).

[50] Shah, D., Iyer, S., Prabhakar, B., and McKeown, N. Maintaining Statistics Counters in Router Line Cards. *IEEE Micro* (2002).

[51] Sivaraman, V., Narayana, S., Rottenstreich, O., Muthukrishnan, S., and Rexford, J. Heavy-hitter detection entirely in the data plane. In *ACM SOSR 2017*.

[52] Truong, P., and Guillemin, F. Identification of heavyweight address prefix pairs in IP traffic. In *ITC* (Sept 2009).

[53] Tsidon, E., Hanniel, I., and Keslassy, I. Estimators Also Need Shared Values to Grow Together. In *IEEE INFOCOM* (2012).

[54] Yang, L., Hao, W., Tian, P., Huichen, D., Jianyuan, L., and Bin, L. CASE: Cache-Assisted Stretchable Estimator for High Speed Per-flow Measurement. In *INFOCOM* (2016).

[55] Zhang, Y., Singh, S., Sen, S., Duffield, N., and Lund, C. Online Identification of Hierarchical Heavy Hitters: Algorithms, Evaluation, and Applications. In *ACM IMC* (2004).