Per-flow Counting for Big Network Data Stream over Sliding Windows

You Zhou† Yian Zhou†‡ Shigang Chen† Youlin Zhang†
†Department of Computer & Information Science & Engineering, University of Florida, Gainesville, FL, USA
‡Google Inc., 1600 Amphitheatre Parkway, Mountain View, CA, USA
Email: youzhou@cise.ufl.edu yianzhou@google.com sgchen@cise.ufl.edu youlin@cise.ufl.edu

***Abstract*—Per-flow counting for big network data streams is a fundamental problem in various network applications such as traffic monitoring, load balancing, capacity planning, etc. Traditional research focused on designing compact data structures to estimate flow sizes from the beginning of the data stream (i.e., landmark window model). However, for many applications, the most recent elements of a stream are more significant than those arrived long time ago, which gives rise to the sliding window model. In this paper, we consider per-flow counting over the sliding window model, and propose two novel solutions, ACE and S-ACE. Instead of allocating a separate data structure for each flow, both solutions utilize the counter sharing idea to reduce memory footprint, so they can be implemented in on-chip SRAMs in modern routers to keep up with the line speed. ACE has to reset the sliding window periodically to give precise estimates, while S-ACE based on a novel segment design can achieve persistently accurate estimates. Our extensive simulations as well as experimental evaluations based on real network traffic trace demonstrate that S-ACE can achieve fast processing speed and high measurement accuracy even with a very tight memory.**

I. Introduction

Network data streams arise in many applications such as high-speed network traffic measurement, Internet data analysis, finance, etc [1], [2], [3], [4], [5], [6]. Per-flow counting over big network data stream consisting of numerous flows is a fundamental problem. In a general definition, per-flow counting is to count the number of elements for each flow, or flow size in short. It has many important applications in various domains such as load balancing, capacity planning, resource fairness, and intrusion detection. To keep up with the line speed of modern network devices (e.g., routers), the per-flow counting module needs to be implemented in SRAM. The limited SRAM cannot accommodate numerous flows in big network data stream, which poses the major challenge for per-flow counting over big network data streams.

Many approaches [7], [8], [9], [10], [11], [12], [13], [14], [15] have been proposed to estimate flow sizes. Giving one counter for each flow requires more memory than the available size on SRAM. One important thread of research in this area is based on sketch. The representative work includes count-min sketch [7], which are typically optimized and have been implemented in hardware. These approaches can mainly answer point queries. That is, given a flow label, they can provide an estimation for the flow size. Although the memory needed to encode each flow has been greatly reduced, when the number of flows are extremely large, the memory requirement is still very high. To further reduce memory overhead, better alternatives are counter sharing methods [9], [10], [11], [13]. In particular, [11] leverages a counter sharing mechanism, where all flows share a common memory space. Therefore, it can do per-flow counting for big network data streams.

Traditional research focused on estimating flow sizes from the beginning of the data stream (i.e., landmark window model). In the landmark model, given a "landmark" time point, the data analysis are only on the data stream which falls between the landmark and the current time point. When more and more elements pass through the router, the landmark window runs out of capacity, and has to reset to zero periodically [16]. This is the major disadvantage of this model. For many real-time applications, the most recent elements of a stream are more significant than those arrived long time ago [17], [18], which gives rise to the sliding window model. For example, an ISP may monitor the data streams to identify the user who sends most packets in the last hour. In the counter based sliding window model, it removes an expired element as a new element arrives, thereby it always maintains the most recent W elements in the data stream. This paper mainly focuses on per-flow counting under this sliding window model.

Datar et al. [19] first introduce the sliding window model in data streams, and propose an exponential histogram to provide approximation for basic counting. Zhu et al. [20] subdivide the sliding windows equally into basic windows to facilitate the efficient elimination of old data. However, it only provides accurate statistics (e.g., Discrete Fourier Transform) when a basic window is expired, and cannot give accurate estimate when some elements in the oldest basic window are active. Arasu et al. [21] study the problem of maintaining counts and quantiles over a stream sliding window, and there are some work [22], [23] to improve its performance. However, they don't support constant time point query and need to allocate memory dynamically. Typically, they need more memory s-pace than the landmark model, which makes them hard to implement in hardware.

In this paper, we tackle the per-flow counting problem for big network data stream over sliding windows. To achieve optimal memory efficiency, we adopt the counter sharing idea to the sliding window model, and propose two novel per-flow counting schemes, ACE and S-ACE. The memory overhead of ACE and S-ACE is the same as randomized counter sharing in [11], which is very compact for hardware implementation in routers. For ACE, we propose an aging algorithm to eliminate one element as a new element comes. It is simple and efficient, but requires resetting the sliding window periodically to give

978-1-5386-2704-4/17/$31.00 ©2017 IEEE

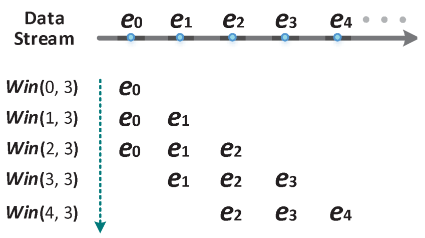

Data Stream
e_0 e_1 e_2 e_3 e_4
$Win(0, 3)$ e_0
$Win(1, 3)$ e_0 e_1
$Win(2, 3)$ e_0 e_1 e_2
$Win(3, 3)$ e_1 e_2 e_3
$Win(4, 3)$ e_2 e_3 e_4

Fig. 1: An example of sliding windows with $W = 3$.

accurate flow size estimates. To achieve persistently accurate per-flow counting without periodical sliding window resetting, we propose a novel segment window design in the advanced S-ACE scheme. S-ACE achieves the optimal processing speed, two memory accesses to encode one element. Our extensive simulations as well as experimental evaluations based on real network traffic trace demonstrate that S-ACE can work in very tight memory space with high accuracy.

II. Preliminaries

A. Network Data Stream and Sliding Windows

We consider a network data stream S as a time ordered series of elements $\langle e_0, e_1, e_2, \ldots e_i, \ldots \rangle$, where the subscript is the arriving sequence order index, called time point. For example, the element e_i is passing by the router at time point i. Each element is associated with a flow label f. A flow f consists of the elements with the same flow label f. The flow label can be flexibly defined depending on application context. For example, the flow label can be source address, destination address or other user-defined flow identifiers.

A sliding window [20] over a network data stream S is a multi-set of last W elements of the stream passed by so far, where the nonnegative integer W is called its window size. Therefore, given the length of the sliding window W and the current time point t (i.e., when the element e_t arrived), the sliding window maintains the most recent W elements in S, $\langle e_{\max\{0,t-W+1\}}, \ldots, e_t \rangle$, which can be denoted by $Win(t, W)$. An example of sliding windows with window size $W = 3$ for a network data stream is illustrated in Fig. 1.

B. Problem Statement

The problem we tackle in this paper is the per-flow traffic measurement over sliding windows. At any time point t, we need to maintain a data structure for the last W elements $Win(t, W)$ over a network data stream S. Given flow f, the data structure can be used to return the estimated flow size of f (i.e., the number of elements with flow label f) in the sliding window $Win(t, W)$. The goal is to minimize the memory requirement in such continuous computation, as well as to keep up with the high-speed network data stream processing in real time.

Clearly, we are only interested in the recent past. It is desirable if we can remove the elements outside a sliding window. On the one hand, as the network data in a sliding window continuously change as new elements arrive, it is infeasible to remove the exact outdated elements without using a space of $O(W)$. Therefore, one major challenge is to develop a space-efficient technique to continuously summarize a data stream in the sliding window model. On the other hand, when the data stream over the sliding window is summarized, we lose the temporal information related to the expired elements, which causes the accuracy problem. Hence, the other challenge is to measure per-flow size over sliding windows with high approximate accuracy.

C. Randomized Counter Sharing

For network data streams, Li et al. [11] proposed an efficient per-flow traffic measurement scheme called randomized counter sharing. This scheme leverages a counter sharing mechanism, which is illustrated in Fig. 2. Each flow randomly picks a number of counters from the physical counter array to form its virtual counter. When recoding an element of a particular flow, it randomly maps to a counter of the flow's virtual counter, and increases the counter by one. To estimate the size of a flow, it first adds up the values of the counters that the flow is mapped to, and then removes the noise introduced by other flows. Since the virtual counters of all flows share the same counter pool (physical counter array), large flows can 'borrow' memory from small flows to utilize the available counter bits. This scheme [11] can achieve reasonably accurate results for per-flow size estimation even under very tight memory space.

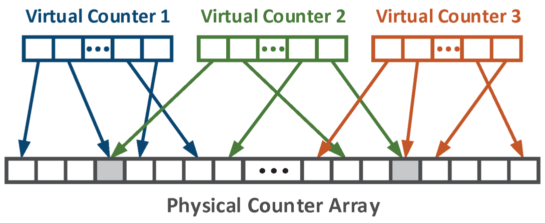


Fig. 2: An illustration of counter sharing.

This randomized counter sharing scheme works perfectly over the landmark window model, which maintains all elements in the network data stream seen so far. However, in the sliding window model that we consider in this paper, only the last W elements are of interest. The actual contents of most recent W elements change because the oldest element is removed when a new element arrives. This makes the existing per-flow traffic measurement based on a whole data stream not trivially applicable.

III. Aging Counter Estimation over Sliding Windows

In this section, we adopt randomized counter sharing to the sliding window model, and propose an **A**ging **C**ounter **E**stimation (ACE) scheme to measure per-flow traffic over sliding windows. In our ACE scheme, when a new element arrives, to maintain the most recent W elements in the sliding window, we first process an aging algorithm in the previous window, which tries to remove the oldest element, and then encode the new element as described in randomized counter sharing. Below we describe the ACE scheme in detail.

ACE includes an online operation module and a real time estimation module. The online operation module records the data stream over sliding window to the physical counter array, while the real time estimation module answers queries of flow sizes based on data recorded from the online operation module. We first give our data structure design.

A. Virtual Aging Counter

The flow size information over sliding window is stored in an aging counter array C of m counters. The ith counter in C is denoted by $C[i], 0 \leq i < m$. Suppose the total memory size of C is M bits, and the size of each counter is b bits. Then the number of counters in C is $m = \lfloor \frac{M}{b} \rfloor$.

Each flow is allocated a virtual aging counter consisting of s counters to record the elements of the flow. Those virtual aging counters share a common aging counter array C. Specifically, consider an arbitrary flow f, we pseudo-randomly select s counters from C to form a virtual aging counter, denoted as C_f, where the ith counter in C_f is denoted by $C_f[i], 0 \leq i < s$. According to the randomized counter sharing [11], one possible approach to form C_f from C using one master hash function is as follows,

$$C_f[i] = C[H(f \oplus R[i]) \mod m],\ 0 \leq i < s, \tag{1}$$

where $\oplus$ is the XOR operator and R is an array of s random constants. Through this, we can construct a virtual aging counter for each flow from the same counter pool C. Note that the aging counter array C and virtual aging counter C_f are denoted by C^t and C_f^t at time point t, respectively.

B. Online Operation

When an element e_t arrives at time point t, the router takes two steps to process the new element, an aging step and an encoding step. The aging step tries to remove the oldest element in the previous window at time point $t-1$, and the encoding step records the new element e_t in C to form the current window. The window after online operation of e_t is denoted as $\widehat{Win}(t, W)$. We point out that $\widehat{Win}(t, W)$ can be slightly different from the real window $Win(t, W)$ due to the approximate deletion in the aging step.

1) Aging Step: The elements in the network data stream arrive continuously and expire after exactly W steps. Therefore, when the total number of elements in the window is less than W (i.e., $t < W$), no element is expired in the window such that the aging step is not needed. When $t \geq W$, a new coming element causes one expired element, which should be removed from the current window. However, as we cannot store the information about the arrival order of elements due to memory limitation, exactly removing the outdated element is not applicable. Hence, probabilistic deletion is the only viable choice. A straightforward aging algorithm is to let the router randomly delete one element in $\widehat{Win}(t-1, W)$ such that every element in this window has the same probability to be removed. However, since all elements are recorded in the compact aging counter array C, equal probability deletion will cause more memory accesses, which is not applicable. For example, one way is to generate a random index i in $[0, W)$, and delete the i-th element in the counter array C. However, in order to find this element in the counter array C, we need to traverse C, which causes $O(m)$ memory accesses. Although some other algorithms may improve this performance, the memory access overhead will still be high. To achieve more efficient probabilistic deletion, we propose a very simple aging algorithm with $O(1)$ memory access as illustrated in Algorithm 1. It randomly picks a counter in C and decreases it by one if applicable.

Algorithm 1 Aging Algorithm

Input: aging counter array C^{t-1} which records $\widehat{Win}(t-1, W)$
Result: delete one element in C^{t-1}

```
1: $isDeleted$ = false;
2: while $isDeleted$ = false do
3:     select a counter $C^{t-1}[r]$ uniformly at random;
4:     if $C^{t-1}[r] > 0$ then
5:         $C^{t-1}[r]$ = $C^{t-1}[r] - 1$;
6:         $isDeleted$ = true;
7:     end if
8: end while
```

2) Encoding Step: After removing one element from the previous window $\widehat{Win}(t-1, W)$, the router then encodes the new coming element e_t as follows. It first extracts its flow label f, generates a random integer q to select a counter $C_f[q \mod s]$ from flow f's virtual aging counter C_f, and increases it by one. Hence,

$$C^t[p] = C^{t-1}[p] + 1, \tag{2}$$

where $p = H(f \oplus R[q \mod s]) \mod m$. Therefore, with one deletion and one insertion, the counter array C^t maintains the new sliding window $\widehat{Win}(t, W)$ with fixed size W.

C. Real Time Flow Size Estimation

To answer the size of flow f in the sliding window under query at time point t, similar to the randomized counter sharing scheme in [11], we first add up the values of its virtual aging counter C_f, and then remove the noise introduced by other flows from the sum $\sum_{i=0}^{s-1} C_f^t[i]$. Let n^t be the actual size of flow f at time point t, and $\hat{n}^t$ be the estimated flow size. Due to counter sharing, each element of other flows has a probability of $\frac{s}{m}$ to be encoded in one of the s counters in flow f's virtual aging counter, thereby the expected noise in C_f is $\frac{s(W-n^t)}{m} \approx \frac{sW}{m}$ $(n^t \ll W)$. Hence, we have $n^t \approx \sum_{i=0}^{s-1} C_f^t[i] - \frac{sW}{m}$, and the estimated size $\hat{n}^t$ of flow f at time point t is

$$\hat{n}^t = \sum_{i=0}^{s-1} C_f^t[i] - \frac{sW}{m}. \tag{3}$$

D. ACE Performance Analysis

When $t < W$, the sliding window is equivalent to the landmark window, thereby ACE is the same as the randomized counter sharing scheme, which can provide accurate estimations even with tight memory space. Therefore, we only analyze the ACE performance for $t \geq W$ when the aging step is involved.

Consider an arbitrary flow f. Let I_X^t be the indicator of whether the arrival element e_t is recorded in the virtual aging counter C_f, and I_Y^t be the indicator of whether the deleted element in the aging step belongs to C_f. Recall that, in the online operation module of ACE, upon the arrival of e_t, we delete one element in $\widehat{Win}(t-1, W)$ in the aging step, and record e_t in C^t in the encoding step. Therefore, we have

$$\sum_{i=0}^{s-1} C_f^t[i] = \sum_{i=0}^{s-1} C_f^{t-1}[i] + I_X^t - I_Y^t. \tag{4}$$

At the time point $t-1$, by (3), the estimated flow size of f is

$$\hat{n}^{t-1} = \sum_{i=0}^{s-1} C_f^{t-1}[i] - \frac{sW}{m}. \tag{5}$$

Combining (3) (4) (5), we have

$$\hat{n}^t - \hat{n}^{t-1} = I_X^t - I_Y^t. \tag{6}$$

Let I_α^t be the indicator of whether the new coming element e_t belongs to flow f, and I_β^t be the indicator of whether the expired element at time point t comes from flow f. Hence, we have

$$n^t - n^{t-1} = I_\alpha^t - I_\beta^t. \tag{7}$$

According to Algorithm 1, every counter has almost the same probability $\frac{1}{m}$ to be selected in the aging step (the probability for a counter with zero value is very small). Hence, the probability for $I_Y^t = 1$ is

$$\mathbb{P}(I_Y^t = 1) = \frac{s}{m}. \tag{8}$$

With regard to the indicator I_X^t, if $e_t \in f$ (i.e., $I_\alpha^t = 1$), the element will be encoded in C_f. Otherwise, if $e_t \notin f$ (i.e., $I_\alpha^t = 0$), due to counter sharing, the element can be mapped to C_f with a probability $\frac{s}{m}$. So the probability of $I_X^t = 1$ is

$$\begin{aligned} \mathbb{P}(I_X^t = 1) &= \mathbb{P}(I_\alpha^t = 1) + \frac{s}{m}\mathbb{P}(I_\alpha^t = 0) \\ &= (1 - \frac{s}{m})\mathbb{P}(I_\alpha^t = 1) + \frac{s}{m}. \end{aligned} \tag{9}$$

Combining (6) (8) (9), the expectation value of $\hat{n}^t - \hat{n}^{t-1}$ is

$$\begin{aligned} E(\hat{n}^t - \hat{n}^{t-1}) =& E(I_X^t) - E(I_Y^t) \\ =& \mathbb{P}(I_X^t = 1) - \mathbb{P}(I_Y^t = 1) \\ =& E(I_\alpha^t) - \frac{s}{m}E(I_\alpha^t). \end{aligned} \tag{10}$$

And the expectation value of $n^t - n^{t-1}$ is

$$E(n^t - n^{t-1}) = E(I_\alpha^t) - E(I_\beta^t). \tag{11}$$

Therefore, we have

$$E(n^t - n^{t-1}) - E(\hat{n}^t - \hat{n}^{t-1}) = \frac{s}{m}E(I_\alpha^t) - E(I_\beta^t). \tag{12}$$

We find the estimated flow sizes form a Markov chain. From the equation above, when deriving $\hat{n}^t$ from $\hat{n}^{t-1}$, the ACE scheme will introduce a small error. For example, when $E(I_\alpha^t) = E(I_\beta^t)$ such that $E(\hat{n}^t - \hat{n}^{t-1}) > E(n^t - n^{t-1})$, the estimates will have positive bias. Therefore, the estimation accuracy of ACE decreases when t increases. It may work fine when t is slightly larger than W. However, as t becomes large (e.g., $t > 2W$), the sliding window accumulates many expired elements, which introduces more noise in the size estimation, and makes ACE no longer accurate. This trend can also be observed in the simulation results in Section V.

One way to solve this problem is to reset the measurement window after a period of time T ($T > W$). However, when T is set too small (slightly larger than W), the window is reset too often such that the sliding window model is broken. If we choose a large T, then ACE cannot provide accurate estimations in the end. Hence, the challenge is how to maintain the sliding window model and provide persistently accurate flow size estimates for real time queries.

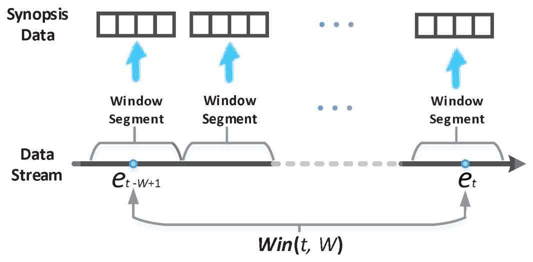


Fig. 3: An illustration of segment window design.

IV. Segment Aging Counter Estimation over Sliding Windows

A. Segment Design

The previous design using one data synopsis (one aging counter array C) drops the order information of all elements, thereby it cannot provide persistently accurate flow size estimates in the sliding window model. In this section, we propose a novel segment design as illustrated in Fig. 3, where we use multiple data synopses to store the elements arriving in different window segments. This design maintains the relative order between the window segments with their data synopses. For example, the elements in left window segments arrived earlier than those in the right window segments in Fig. 3.

Even though we still cannot distinguish the order within each window segment, this design significantly improves the probability to delete the correct outdated elements. The idea is to always insert new element in the newest window segment called *"head segment"*, and delete old element in the oldest window segment called *"tail segment"*. For example, suppose the number of segments is 100. Since the expired element must be in the tail segment, we can filter out at least 99% elements in the current sliding window. Moreover, when all elements in the tail segment are expired, a new segment window is fully constructed as the new head segment and no aging error exists. We reach a time point that all elements in the current sliding window are correct and the reset happens automatically without breaking the sliding window model. Therefore, this design can provide persistently accurate flow size estimates for real time queries.

However, there are still some technical challenges in making our segment design usable. For example, when a new element arrives, how do we perform the aging step in the tail segment and the encoding step in the head segment? How to answer per-flow counting queries in real time? In fact, since these segment windows and data synopses are separate in data structure, it is still unclear whether or not we can combine them to do the per-flow counting. Another challenge is how to design an efficient data structure to recycle the memory without allocating new memory for each new segment. When all elements in a segment are expired, we don't want to simply

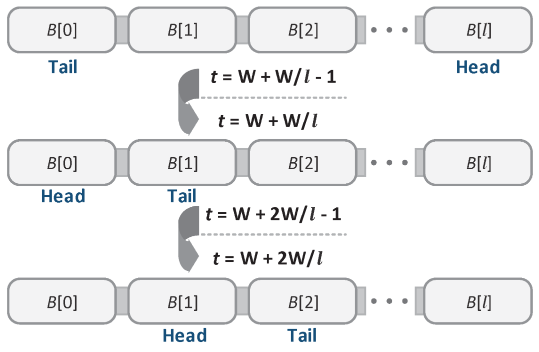


Fig. 4: An example of memory reuse.

delete it as we might be able to reuse it to store new coming elements. To answer these questions, we propose our advanced **S**egment **A**ging **C**ounter **E**stimation (S-ACE) scheme. Below we first give our data structure with segment design.

B. Segment Aging Counter

The physical aging counter B is divided into $(l+1)$ segments, each of which is called a segment aging counter. The ith segment is denoted as $B[i], 0 \leq i \leq l$. Suppose the size of B is M and each counter is allocated with b bits, then each segment has $m' = \lfloor \frac{M}{b(l+1)} \rfloor$ counters. The jth counter in $B[i]$ is denoted as $B[i][j], 0 \leq j < m'$. Note that every segment is one data synopsis of the data stream in a window segment, which contains exact $\frac{W}{l}$ elements except for the head segment and the tail segment when $t \geq W$. This is because the tail segment is eliminating the oldest elements and the head segment is under construction with newest elements. But the sum of the elements in these two segments is also $\frac{W}{l}$, thereby the total number of elements in the sliding window is W.

Each segment aging counter encodes the data of a window segment, and its data structure is presented in Section III-A. Consider an arbitrary flow f constructing its virtual aging counter $B_f[i]$ in the ith segment $B[i]$, $0 \leq i \leq l$, which contains s counters. We pseudo-randomly select s counters from $B[i]$ to form it. The j-th counter of $B_f[i]$, denoted as $B_f[i][j]$, is chosen from segment $B[i]$ as follows,

$$B_f[i][j] = B[i][H(f \oplus R[i][j]) \mod m'],\ 0 \leq j < s, \quad (13)$$

where R is a two dimensional array of $(l+1) \times s$ random constants. Note that the segment aging counter $B[i]$ and virtual segment aging counter $B_f[i]$ for flow f are denoted by $B^t[i]$ and $B^t_f[i]$ at time point t, respectively.

Now, we discuss how to reuse these $(l+1)$ segments to continuously record elements without allocating extra memory for new segment. Generally, when the current head segment is full (recorded $\frac{W}{l}$ elements), for the new coming elements, we need a new segment, which becomes the new head segment. If we allocate new memory space for the new segment, the memory overhead will increase over time, which is not acceptable. Recall that the elements in the tail segment are all expired when the head segment is full. Hence, we can reset the tail segment, and reuse it as the new head segment. In this case, the previous second oldest segment becomes the new tail segment. Mathematically, we can calculate that, at time point t, the segment aging counter index H of the head segment is

$$H = \frac{t}{W/l} \mod (l+1), \quad (14)$$

and the index T of the tail segment is

$$T = \begin{cases} 0 & t < W, \\ (\frac{t}{W/l} + 1) \mod (l+1) & t \geq W. \end{cases} \quad (15)$$

An example of memory reuse is illustrated in Fig. 4. When $t < W + \frac{W}{l} - 1$, the tail segment is always $B[0]$, and the head segment keeps moving forward to fill up window segments as new elements come. When $t = W + \frac{W}{l} - 1$, the element $e_{W+\frac{W}{l}-1}$ is recorded in the head segment $B[l]$, which means that $B[l]$ is full, and the tail segment $B[0]$ is expired. We can reset $B[0]$, and reuse it as the new head segment. Then $B[1]$ becomes the new tail segment. When the element $e_{W+\frac{W}{l}}$ comes, it will be recorded in the new head segment $B[0]$. Following this memory recycle design, we can continuously record elements using only $(l+1)$ segment aging counters.

C. Online Operation

When an element e_t arrives, the router takes a "virtual" aging step and an encoding step to process the new element. The "virtual" aging step is actually an no-op for our S-ACE scheme, which virtually deletes an element from the tail segment (we will explain shortly), while the online step inserts the new element into the head segment.

1) "Virtual" Aging Step: Clearly, when $t < W$, no aging step is needed. Below we only consider $t \geq W$. After recording W elements, the first l segments are full, and each of them stores $\frac{W}{l}$ elements. Note that S-ACE always operates the aging step in the tail segment and the encoding step in the head segment. Since the two segments are stored in separate data structures, the aging step and the encoding step won't interfere with each other. This motivates us to design a new aging algorithm called *"proportional aging"*.

The basic idea is that, when an element is to be deleted from the tail segment $B[T]$, instead of decreasing a certain counter in $B[T]$ by one, we proportionally decrease all counters in $B[T]$ by a fraction of one. More specifically, $B^t[T][j] = B^{t-1}[T][j] - \frac{B^{t-1}[T][j]}{W/l}$, $0 \leq j < m'$. Intuitively, the tail segment contains the distribution information of the elements inserted to this window segment, so when we delete elements from it, it is natural to preserve this distribution by proportionally decreasing the values of all the counters in it.

We stress that we don't actually perform this aging operation on the tail segment. Instead, we keep the tail segment the way it is, like a "snapshot". This snapshot carries the element insert distribution, and can be used to perform a logical batch aging in the flow size estimation module in Section IV-D. The reason is that, the segment design allows us to calculate the number of elements in the head segment at time point t, $n_H = (t+1) \mod (\frac{W}{l})$, which is exactly the number of elements to be deleted in the tail segment. In the estimation module, we just need to deduct each counter value in the tail segment snapshot (i.e., $B^t[T][j]$) by its proportion value $\frac{B^t[T][j]}{W/l}$ times the total delete number n_H to get the actual counter value.

2) Encoding Step: With aging step being no-op, encoding becomes the only online operation. When a new element e_t arrives, the router inserts it to the head segment. The router first extracts its flow label f, generates a random integer $Q \in [0, s)$ to select a counter of flow f's virtual aging counter $B_f[H]$ in the head segment $B[H]$, and increases it by one. Hence,

$$B_f^t[H][Q] = B_f^{t-1}[H][Q] + 1, \tag{16}$$

where H is given by (14).

When $t \geq W$ and $(t+1) \mod (\frac{W}{l}) = 0$, the router resets the tail segment $B[T]$, where T is given by (15), since all elements in this segment window are expired.

D. Real Time Flow Size Estimation

To answer the size of a flow f in the sliding window under query at time point t, similar to ACE, we first add up the values of its virtual segment aging counters in all segments, and then subtract the noise introduced by other flows and the proportional expired elements n_e in B_f^t from the sum $\sum_{i=0}^{l}\sum_{j=0}^{s-1} B_f^t[i][j]$. Clearly, the expected noise in B_f^t is $\frac{s(W-n^t)}{m'}$, since in each window segment, each element of other flows has a probability of $\frac{s}{m'}$ to be encoded in one of the s counters in B_f^t. In addition, according to our proportional aging algorithm, n_e is $\sum_{j=0}^{s-1} \frac{((t+1) \mod (\frac{W}{l}))\times B_f^t[T][j]}{W/l}$. Therefore, we have

$$\begin{aligned} n^t \approx & \sum_{i=0}^{l}\sum_{j=0}^{s-1} B_f^t[i][j] - \frac{s(W-n^t)}{m'} \\ & - \sum_{j=0}^{s-1} \frac{\big((t+1) \mod (\frac{W}{l})\big)\times B_f^t[T][j]}{W/l}. \end{aligned} \tag{17}$$

Hence, the estimated size $\hat{n}^t$ of flow f at time point t is

$$\begin{aligned} \hat{n}^t = & \frac{m'}{m'-s}\sum_{i=0}^{l}\sum_{j=0}^{s-1} B_f^t[i][j] - \frac{sW}{m'-s} \\ & - \frac{lm'\big((t+1) \mod (\frac{W}{l})\big)}{(m'-s)W}\sum_{j=0}^{s-1} B_f^t[T][j]. \end{aligned} \tag{18}$$

E. S-ACE Performance Analysis

When $t < W$ or $(t+1) \mod (\frac{W}{l}) = 0$, our S-ACE scheme is similar with randomized counter sharing scheme. Without loss of generality, we only analyze the S-ACE performance when $W < t+1 < (W + \frac{W}{l})$. Consider the flow f. Let $I_{X_H}^t$ be the indicator of whether the arrival element e_t is recorded in the virtual aging counter vector $B_f[H]$ of flow f. At the time point $t-1$, the estimated flow size of f is

$$\begin{aligned} \hat{n}^{t-1} = & \frac{m'}{m'-s}\sum_{i=0}^{l}\sum_{j=0}^{s-1} B_f^{t-1}[i][j] - \frac{sW}{m'-s} \\ & - \frac{lm'\big(t \mod (\frac{W}{l})\big)}{(m'-s)W}\sum_{j=0}^{s-1} B_f^{t-1}[T][j]. \end{aligned} \tag{19}$$

Note that the new element e_t will be encoded in the head segment $B[H]$, the counters in other segment remain the same value. That is $B_f^{t-1}[i] = B_f^t[i], i \in [0, l], i \neq H$. Combining (18) with $W < t < (W + \frac{W}{l})$, we have

$$\hat{n}^t - \hat{n}^{t-1} = \frac{m'}{m'-s} I_{X_H}^t - \frac{lm'}{(m'-s)W}\sum_{j=0}^{s-1} B_f^t[T][j]. \tag{20}$$

Let n_T^t be the total number of elements in flow f encoded in the tail segment at time point t. Due to counter sharing, we have

$$E(\sum_{j=0}^{s-1} B_f^t[T][j]) = n_T^t + \frac{s}{m'}(\frac{W}{l} - n_T^t), \tag{21}$$

where $\frac{s}{m'}\big(\frac{W}{l} - n_T^t\big)$ is expected noise in $B_f^t[T]$. Recall that I_β^t is the indicator of whether the new coming element e_t belongs to flow f. By (9), the probability of $I_{X_H}^t = 1$ is

$$\begin{aligned} \mathbb{P}(I_{X_H}^t = 1) &= \mathbb{P}(I_\alpha^t = 1) + \frac{s}{m'}\mathbb{P}(I_\alpha^t = 0) \\ &= \big(1 - \frac{s}{m'}\big)\mathbb{P}(I_\alpha^t = 1) + \frac{s}{m'}. \end{aligned} \tag{22}$$

Therefore, the expected value of $\hat{n}^t - \hat{n}^{t-1}$ is

$$\begin{aligned} & E(\hat{n}^t - \hat{n}^{t-1}) \\ & = \frac{m'}{m'-s}E(I_{X_H}^t) - \frac{lm'}{(m'-s)W}E(\sum_{j=0}^{s-1} B_f^t[T][j]) \\ & = \mathbb{P}(I_\alpha^t = 1) - \frac{l}{W}n_T^t = E(I_\alpha^t) - \frac{l}{W}n_T^t. \end{aligned} \tag{23}$$

Recall that I_β^t is the indicator of whether the expired element at time point t comes from flow f. The expectation value of $n^t - n^{t-1}$ is

$$E(n^t - n^{t-1}) = E(I_\alpha^t) - E(I_\beta^t). \tag{24}$$

when deriving $\hat{n}^t$ from $\hat{n}^{t-1}$, S-ACE will introduce some error. But when the arriving order of elements from each flow is evenly distributed, the probability for an element of flow f to be expired is $\frac{ln_T^t}{W}$. So the error will be very small, which will be demonstrated in the simulation results in Section V. Moreover, When $t \geq W$ and $(t+1) \mod (\frac{W}{l}) = 0$, we reset the tail segment since all elements in this segment window are expired. The current window $\widehat{Win}(t, W)$ is the same as real window $Win(t, W)$ such that S-ACE can provide accurate estimates as [11] in the landmark window model. In summary, this design can provide persistent accurate flow size estimates for real time queries.

V. Simulation Studies

In this section, we present simulation studies that justify the performance analysis of ACE and S-ACE. We compare both schemes in terms of processing time, memory efficiency and estimation accuracy over the simulated data.

A. Performance Metrics

We employ the following three metrics same as [13] to evaluate the performance of a per-flow counting scheme for network data streams.

Processing time: The average time required for encoding an element. It is measured by the average number of memory accesses and the number of hash value computations as in [13].

Memory requirement: The memory overhead to achieve reasonable estimation results for per-flow traffic measurement.

Estimation accuracy: We evaluate the estimation accuracy by the relative bias $Bias(\frac{\hat{n}}{n})$ and relative standard error $StdErr(\frac{\hat{n}}{n})$ as [13]:

$$
\begin{aligned}
Bias(\frac{\hat{n}}{n}) &= E(\frac{\hat{n}}{n}) - 1, \\
StdErr(\frac{\hat{n}}{n}) &= \sqrt{Var(\frac{\hat{n}}{n})} = \frac{\sqrt{Var(\hat{n})}}{n}.
\end{aligned} \tag{25}
$$

B. Simulation Setup

The dataset we use to evaluate our schemes is 20 simulated traces of 10^7 elements, generated with a Zipf distribution [24] of skew 1, over a domain of 10^6 possible flows. We refer to this dataset as *Zipf-1*. Hence, the simulation has 20 repeated runs, and we provide the mean results over these runs. The window size W in all simulations is 10^6. We measure the network data stream among 2 windows. We skip the first window where all algorithms based on landmark window model can work fine, and focus on the performance when $t \geq W$, where the sliding window model is working.

We run two simulation sets to evaluate our schemes. The first set of our simulations is used to compare ACE and S-ACE. We allocate the same size of memory for both schemes, and evaluate the impact of memory size on their performance. We vary the available memory space M from 0.5MB, 1MB to 2MB. For ACE, according to randomized counter sharing [11], we set the size s of the virtual aging counter of each flow to 100. For S-ACE, we set the number of segment windows to $l + 1 = 101$, and the size s of virtual aging counter in each segment to 16. The second set of our simulations evaluates the impact of the aging step in S-ACE with regard to the measurement accuracy.

C. S-ACE v.s. ACE

1) Processing Time: We first compare ACE and S-ACE in terms of processing time. In each simulation run, we record the average number of memory accesses and hash computations for maintaining the sliding window when a new element comes. The average results of 20 runs are presented in Table I. Both schemes only need 1 hash computation for each packet to locate its corresponding counter. In addition, due to the aging step, ACE needs more than 3 memory accesses in online operation. By contrast, S-ACE only requires 2 memory accesses. Moreover, the average number of memory accesses of ACE decreases when the memory space is reduced. This is because ACE has less probability to hit counter with value zero in the aging step when less memory are available. Clearly, S-ACE is more efficient than ACE in terms of processing time.

2) Memory Overhead and Estimation Accuracy: We compare the estimation accuracy of ACE and S-ACE under different available memory space with regard to the relative bias and relative standard error. Because there are too few flows for some flow sizes, we compute the relative bias and relative standard error by dividing the flow size axis into measurement bins.

TABLE I: Comparison of processing time by ACE and S-ACE.

memory overhead (MB)	number of memory accesses		number of hash computaions	
	ACE	S-ACE	ACE	S-ACE
0.25	3.55	2	1	1
0.5	4.18	2	1	1
1	5.45	2	1	1
2	7.98	2	1	1

We first study the estimation accuracy of ACE. The relative bias and relative standard error of ACE are presented in Figure 5 and Figure 6, respectively. Each figure contains 3 plots, whose available memory ranges from 0.5MB, 1MB to 2MB, from left to right. In each plot, the x-coordinate is the true flow size n, y-coordinate is the time point t, and the z-coordinate is the relative bias in Figure 5 or relative standard error in Figure 6. When $M = 0.5$MB, the simulation results are shown in Figure 5a and Figure 6a. We can see that the relative bias and relative standard error increase as the time point t grows. Hence, the ACE scheme can only work for a limited time, as it yields non-reasonable estimates when t is large (e.g., $t = 2W$). The same trends are observed in other plots. Although the ACE scheme becomes more accurate when the available memory space increases, it still cannot work well as t increases.

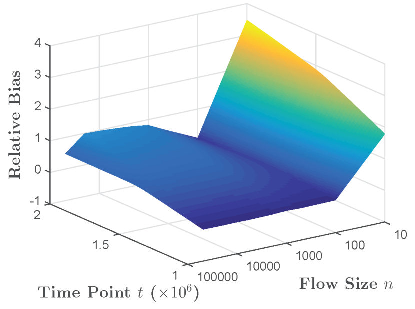
The simulation results of S-ACE are presented in Figure 7 and Figure 8. When $t = W$ (no element is expired), S-ACE can provide accurate estimates for all flows even under a tight memory space, e.g., $M =$0.5MB. Moreover, as illustrated in Figure 7a and Figure 8a, as time passes by, the relative bias and relative standard error are stabilized, and S-ACE yields accurate measurement. Hence, S-ACE can work persistently well in sliding window model. Also, when M increases, S-ACE gives more accurate estimates.

In summary, the ACE scheme can only work for short term traffic measurement over sliding window model. As mentioned before, it needs to reset the time point to 0 when it can no longer provide accurate results, such that the sliding window model is broken. By contrast, the S-ACE scheme performs persistently well, thereby it suites for long term traffic measurement over the sliding window model.

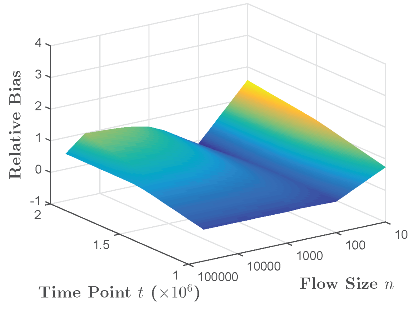
D. Aging Step in S-ACE

Our segment design maintains the relative order between the window segments. When a new window segment is fully constructed by $\frac{W}{l}$ elements, we simply remove the expired window segment without error since all elements in the segment are expired. However, when the new window segment is under construct (less than $\frac{W}{l}$ is encoded), the aging step is proportionally approximated.

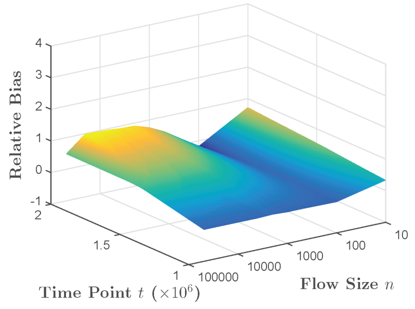
We use simulations to justify this approximation within the window segment. We set the memory space to 1MB, the number of segment windows to 101, and the size s of virtual counter vector in each segment to 16. Each segment window contains up to 10^4 elements. We sample the S-ACE estimations each time when 2000 elements are recorded, and compute their estimation bias and error. Therefore, the samples are divided to 5 categories with 0%, 20%, 40%, 60% and 80% expired elements in the segment, which are denoted by 0% aging,



(a) Memory space $M = 0.5MB$

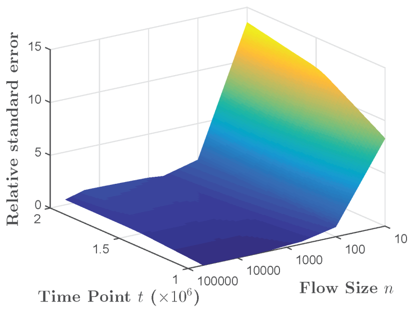


(b) Memory space $M = 1MB$

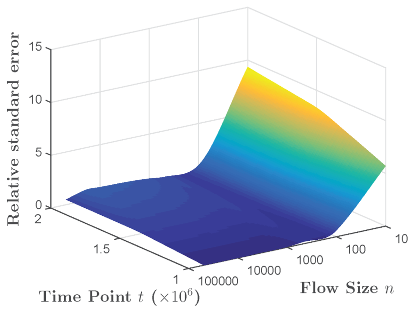


(c) Memory space $M = 2MB$

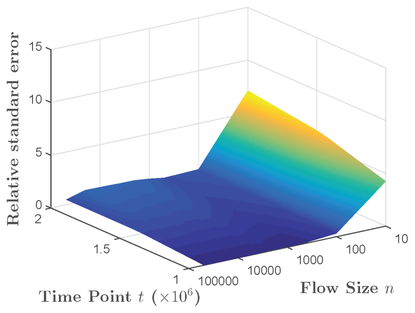
Fig. 5: Relative bias of ACE when $M = 0.5MB$, $1MB$, and $2MB$



(a) Memory space $M = 0.5MB$

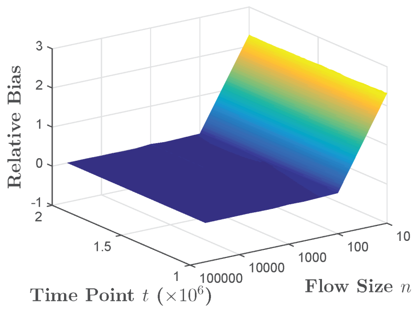


(b) Memory space $M = 1MB$

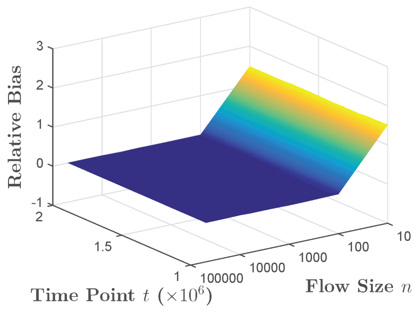


(c) Memory space $M = 2MB$

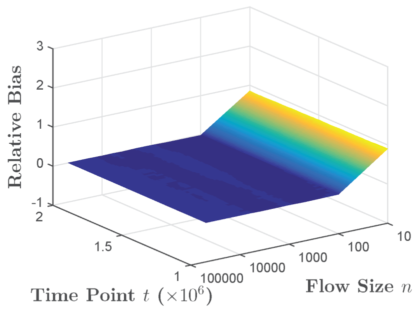
Fig. 6: Relative standard error of ACE when $M = 0.5MB$, $1MB$, and $2MB$



(a) Memory space $M = 0.5MB$

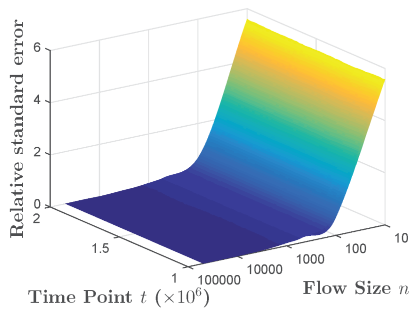


(b) Memory space $M = 1MB$

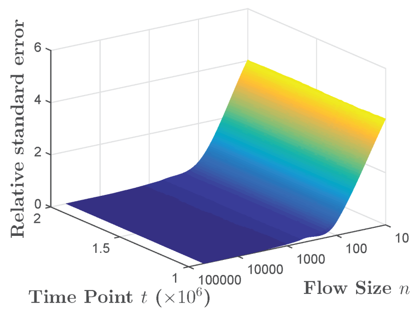


(c) Memory space $M = 2MB$

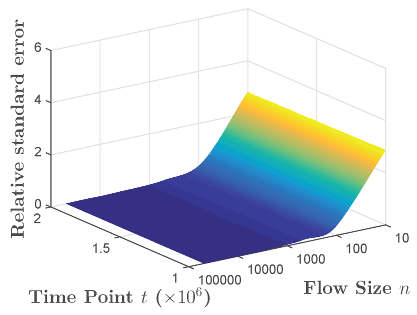
Fig. 7: Relative bias of S-ACE when $M = 0.5MB$, $1MB$, and $2MB$



(a) Memory space $M = 0.5MB$



(b) Memory space $M = 1MB$



(c) Memory space $M = 2MB$

Fig. 8: Relative standard error of S-ACE when $M = 0.5MB$, $1MB$, and $2MB$

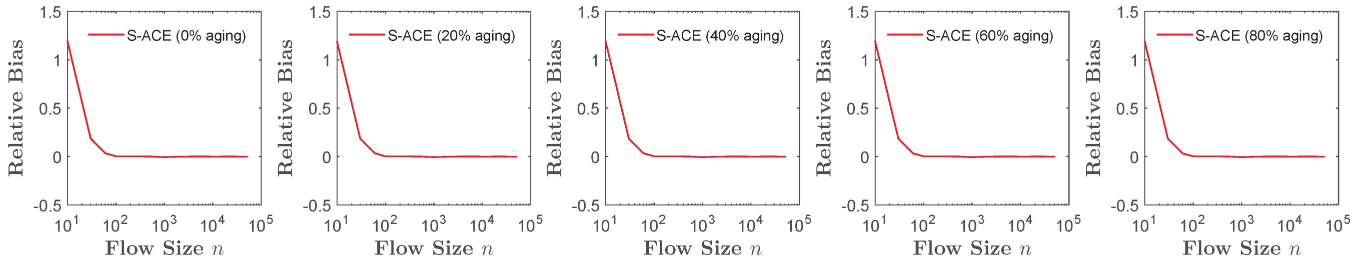


Fig. 9: Relative bias $Bias(\frac{\hat{n}}{n})$ with different aging percentage.

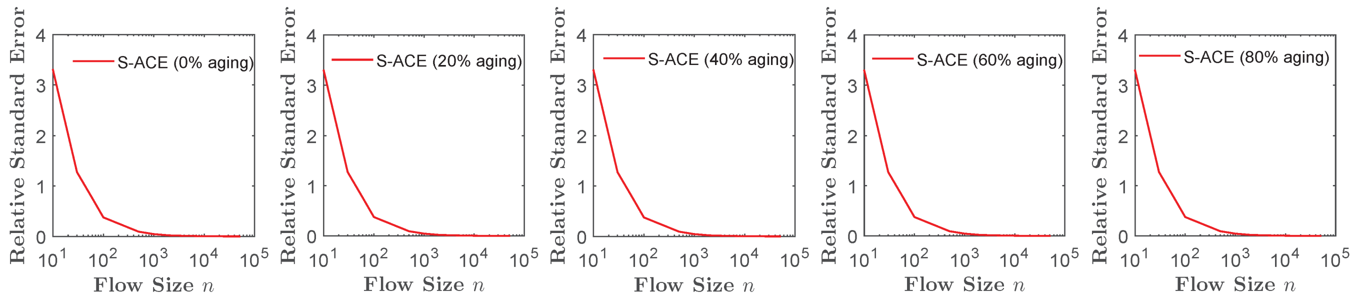


Fig. 10: Relative standard error $StdErr(\frac{\hat{n}}{n})$ with different aging percentage.

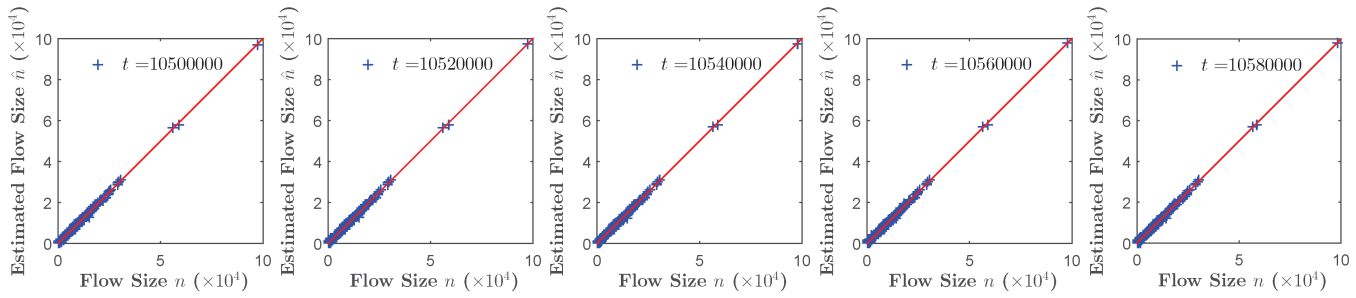


Fig. 11: Per-flow counting using S-ACE in short time interval.

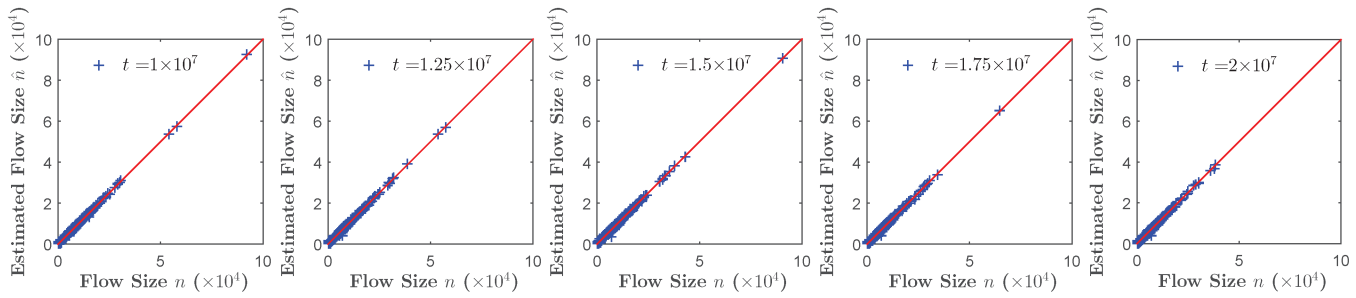


Fig. 12: Per-flow counting using S-ACE in long time interval.

20% aging, 40% aging, 60% aging and 80% aging. Note that 100% aging percentage is equivalent to the case of 0% aging percentage.

The relative bias and relative standard error of these categories with different aging percentages are presented in Figure 9 and Figure 10. From the figures, one can see that the estimation accuracy stays roughly the same no matter how much the aging percentage is, and the aging process in S-ACE does not introduce much error when the number of segment windows is large (e.g., 101).

VI. Experimental Evaluation

We now evaluate the S-ACE scheme based on real network data stream. The data we use is the CAIDA anonymized Internet Trace 2015 [25], which contains $30 \cdot 10^6$ packets. The parameters are set as follows: $W = 10^6$, M = 1MB, l = 100, and $s = 16$.

The estimation results in short time interval and long time interval are given in Figure 11 and Figure 12, respectively. Each figure includes 5 plots, each representing the per-flow counting results in a different time point t. Each point in each plot represents a flow. Its x-coordinate is the actual flow size n, and y-coordinate is the estimated flow size $\hat{n}$. The equality line ($y = x$) is given for reference. The closer a point is to the equality line, the more accurate the estimation is. Clearly, S-ACE provides very accurate estimates in both situations.

We then study the real time query on the real data stream. We query 3 flows in the time point interval from 10^7 to

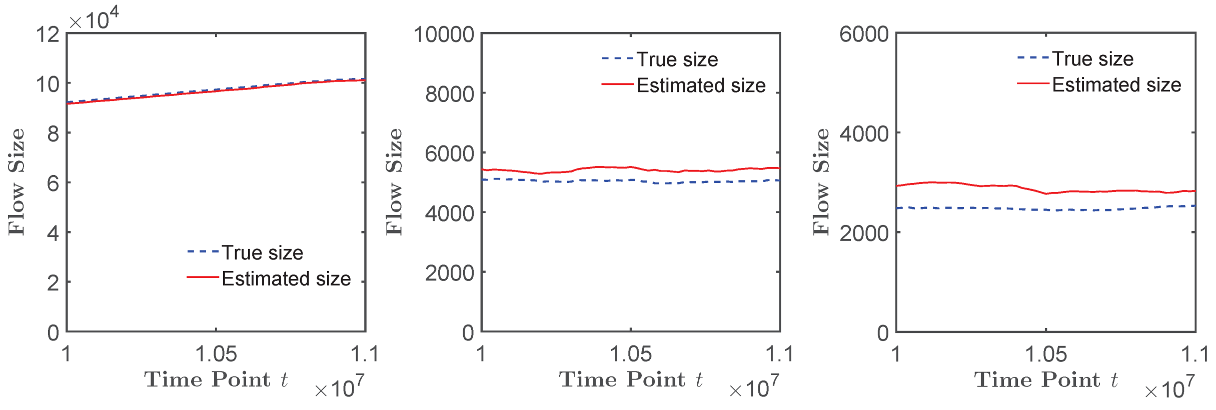


Fig. 13: S-ACE estimates of four flows for time point t between 10^7 and 1.1×10^7.

1.1×10^7. The corresponding results are illustrated in Figure 13. Take a flow with source address 192.205.38.168 and destination address 133.32.39.30 as an example. The results of this flow are shown in the first plot. The true size increases as t grows. The estimates of S-ACE has the same trend with small estimation errors. Similarly, the estimated flow sizes closely follow their actual sizes for the other two flows, as shown in the second plot (source address 100.120.47.9, destination address 215.158.65.254) and third plot (source address 30.196.59.77, destination address 92.168.216.18). We find that the relative standard errors for small flows are relatively higher, but S-ACE is still useful since the absolute errors for small flows are much smaller than those of large ones.

VII. Conclusion

In this paper, we propose two schemes, ACE and S-ACE, for per-flow counting in big network data stream over the sliding window model. Both schemes leverage the counter sharing idea, and greatly reduce the memory overhead. ACE has to reset the window periodically to give precise estimates, while S-ACE can achieve persistently accurate estimates via a novel segment window design. Extensive simulations as well as experimental evaluations based on real network trace data demonstrate the superior performance of S-ACE.

VIII. Acknowledgment

This work is supported in part by the National Science Foundation under grant STC-1562485.

References

[1] C. Estan and G. Varghese, "New Directions in Traffic Measurement and Accounting," *Proc. of ACM SIGCOMM*, August 2002.

[2] A. Kumar, J. Xu, and J. Wang, "Space-Code Bloom Filter for Efficient Per-flow Traffic Measurement," *IEEE Journal on Selected Areas in Communications*, vol. 24, no. 12, pp. 2327–2339, 2006.

[3] X. Dimitropoulos, P. Hurley, and A. Kind, "Probabilistic Lossy Counting: An Efficient Algorithm for Finding Heavy Hitters," *Proc. of ACM SIGCOMM*, 2008.

[4] M. Yu, L. Jose, and R. Miao, "Software Defined Traffic Measurement with OpenSketch," *Proc. of NSDI*, pp. 29–42, 2013.

[5] Y. Zhou, Y. Zhou, M. Chen, Q. Xiao, and S. Chen, "Highly Compact Virtual Counters for Per-Flow Traffic Measurement through Register Sharing," *Proc. of IEEE Globecom*, 2016.

[6] Y. Zhou, Y. Zhou, M. Chen, and S. Chen, "Persistent Spread Measurement for Big Network Data Based on Register Intersection," *Proc. of ACM SIGMETRICS*, 2017.

[7] G. Cormode and S. Muthukrishnan, "An Improved Data Stream Summary: the Count-Min Sketch and Its Applications," *Proc. of LATIN*, 2004.

[8] Q. Zhao, J. Xu, and Z. Liu, "Design of a Novel Statistics Counter Architecture with Optimal Space and Time Efficiency," *ACM SIGMETRICS Performance Evaluation Review*, vol. 34, no. 1, pp. 323–334, 2006.

[9] Y. Lu, A. Montanari, B. Prabhakar, S. Dharmapurikar, and A. Kabbani, "Counter Braids: A Novel Counter Architecture for Per-Flow Measurement," *Proc. of ACM SIGMETRICS*, June 2008.

[10] Y. Lu and B. Prabhakar, "Robust Counting Via Counter Braids: An Error-Resilient Network Measurement Architecture," *Proc. of IEEE INFOCOM*, April 2009.

[11] T. Li, S. Chen, and Y. Ling, "Fast and Compact Per-Flow Traffic Measurement through Randomized Counter Sharing," *Proc. of IEEE INFOCOM*, pp. 1799–1807, April 2011.

[12] Y. Zhou, S. Chen, Y. Zhou, M. Chen, and Q. Xiao, "Privacy-Preserving Multi-Point Traffic Volume Measurement Through Vehicle-to-Infrastructure Communications," *IEEE Transactions on Vehicular Technology*, vol. 64, no. 12, pp. 5619–5630, 2015.

[13] M. Chen and S. Chen, "Counter Tree: A Scalable Counter Architecture for Per-Flow Traffic Measurement," *Proc. of IEEE ICNP*, November 2015.

[14] Y. Zhou, S. Chen, Z. Mo, and Q. Xiao, "Point-to-Point Traffic Volume Measurement through Variable-Length Bit Array Masking in Vehicular Cyber-Physical Systems," *Proc. of IEEE ICDCS*, pp. 51–60, 2015.

[15] Y. Zhou, Z. Mo, Q. Xiao, S. Chen, and Y. Yin, "Privacy-Preserving Transportation Traffic Measurement in Intelligent Cyber-physical Road Systems," *IEEE Transactions on Vehicular Technology*, vol. 65, no. 5, pp. 3749–3759, 2016.

[16] L. Golab, D. DeHaan, E. D. Demaine, A. Lopez-Ortiz, and J. I. Munro, "Identifying Frequent Items in Sliding Windows over On-Line Packet Streams," *Proc. of ACM IMC*, pp. 173–178, 2003.

[17] N. Rivetti, Y. Busnel, and A. Mostéfaoui, "Efficiently Summarizing Data Streams over Sliding Windows," *Proc. of IEEE International Symposium on Network Computing and Applications*, pp. 151–158, 2015.

[18] R. Ben-Basat, G. Einziger, R. Friedman, and Y. Kassner, "Heavy Hitters in Streams and Sliding Windows," *Proc. IEEE INFOCOM*, 2016.

[19] M. Datar, A. Gionis, P. Indyk, and R. Motwani, "Maintaining Stream Statistics over Sliding Windows," *SIAM journal on computing*, vol. 31, no. 6, pp. 1794–1813, 2002.

[20] Y. Zhu and D. Shasha, "StatStream: Statistical Monitoring of Thousands of Data Streams in Real Time," *Proc. of VLDB*, pp. 358–369, 2002.

[21] A. Arasu and G. S. Manku, "Approximate Counts and Quantiles over Sliding Windows," pp. 286–296, 2004.

[22] L.-K. Lee and H. Ting, "A Simpler and More Efficient Deterministic Scheme for Finding Frequent Items over Sliding Windows," pp. 290–297, 2006.

[23] R. Y. Hung, L.-K. Lee, and H.-F. Ting, "Finding Frequent Items over Sliding Windows with Constant Update Time," *Information Processing Letters*, vol. 110, no. 7, pp. 257–260, 2010.

[24] "Zipf's Law." [Online]. Available: https://en.wikipedia.org/wiki/Zipf%27s_law

[25] "CAIDA," 2015. [Online]. Available: http://www.caida.org/home/