SteadySketch: Finding Steady Flows in Data Streams

Xiaodong Li*, Zhuochen Fan*, Haoyu Li*, Zheng Zhong*, Jiarui Guo*, Sheng Long*, Tong Yang*†, Bin Cui*
*School of Computer Science, and National Engineering Laboratory for Big Data Analysis Technology and Application, Peking University, Beijing, China †Peng Cheng Laboratory, Shenzhen, China

2023 IEEE/ACM 31st International Symposium on Quality of Service (IWQoS) | 979-8-3503-9973-8/23/$31.00 ©2023 IEEE | DOI: 10.1109/IWQOS57198.2023.10188743

***Abstract*—In this paper, we study steady flows in data streams, which refers to those flows whose arrival rate is always non-zero and around a fixed value for several consecutive time windows. To find steady flows in real time, we propose a novel sketch, SteadySketch, aiming to accurately report steady flows with limited memory. To the best of our knowledge, this is the first work to define and find steady flows in data streams. The key novelty of SteadySketch is our proposed reborn technique, which reduces the required memory space by 75%. Experimental results show that SteadySketch improves the Precision Rate (PR) by 81.1% and reduces the Average Relative Error (ARE) by 955.3× compared with the strawman solution. Finally, we provide a concrete case: cache prefetch, and prove that SteadySketch can effectively improve the cache hit ratio. All related codes of SteadySketch are open-source and available at GitHub.**

I. Introduction

A. Background and Motivation

Nowadays, network measurement and monitoring has become a research hotspot in the network field. It provides indispensable information for various network management tasks, such as traffic behavior analysis [1], [2], quality of service/experience [3], [4], performance diagnosis [5] and anomaly detection [6]–[9]. Among the tasks mentioned above, a very important research interest is to define and find new patterns in high-speed data streams, such as burst flows [10], periodic flows [11], and quadratic flows [12], *etc.*

This paper defines a new pattern in network data streams, namely steady flow. In practical data stream scenarios, data stream often arrives at high speed, and each flow may appear many times. We divide the data stream into many time windows. Given a time window and a flow e in this window, suppose e appears x times, we define the arrival rate of e as x. For p continuous time windows, if the arrival rate of a flow is non-zero and steady (the variance of arrival rate is less than a given threshold), we call it a **steady flow**.

Steady flow is an important data stream pattern, and has many applications. Below we show three typical ones. 1) Wireless sensor network. Sink node of WSN is responsible for collecting and processing the data from other sensor nodes. Generally, the sensor node that sends steady flows would have higher data reliability, which is important for data processing of sink nodes. 2) network bandwidth pre-allocate. In the highly dynamic network, one flow with steady connection and speed often means that it has higher importance than the short flows which account for the majority of network. Thus, we can pre-allocate the bandwidth for these flows in advance to improve the quality of network service. 3) Steady Cache Line. In this scenario, a flow refers to a cache line in the cache replacement problem. A steady cache line has a higher probability of recurrence in the next few time windows. So we can reduce the cache thrash by avoiding steady cache line from being evicted [13]. In addition, there are many other applications of steady flow, for example, it also can be used for preventing network attacks such as Advanced Persistent Threat (APT) [14], *etc.*

To the best of our knowledge, this is the first work to define steady flows, and no existing literature has provided the same or similar definition. The related problem is finding persistent flows [15]–[18] and K-persistent spread estimation [19]–[21]. A flow is defined as a persistent flow if the number of time windows where it appears exceeds a given threshold [18]. Moreover, persistent flows only care whether it appears in a window, and do not care about the size or variance of arrival rate. Differently, steady flows focus on the size of arrival rate and the variance of arrival rate across several continuous windows. As a result, none of the existing schemes for mining persistent flows can be directly used for detecting steady flows.

B. Our Proposed Solution

In this paper, in order to find steady flows, we first propose a strawman solution composed of multiple CM sketches [22]. However, we find that there are limitations in terms of speed, accuracy, and memory-efficiency. To address these limitations, we propose the first version to optimize speed. Then, we propose the second version to optimize accuracy and the third to further optimize memory usage. Finally, we integrate the three versions and present the final version, namely SteadySketch. Our key novelty lies in the memory optimization: *reborn technique* (see Section III-D). Below we show the rationale of the reborn technique.

The key idea of reborn technique is rebirth with offset variance calculation. In data stream scenarios, flows with large arrival rate are always more important than flows with small arrival rate, but we cannot know the size of the arrival rate in advance. Therefore, it seems that we have to use large counters (*e.g.*, 32 bits or even 64 bits) for all flow to record their arrival rate. However, using large counters will make our data structure too large to be held in a small cache. So

Xiaodong Li and Zhuochen Fan contribute equally to this paper. Corresponding author: Tong Yang (yangtongemail@gmail.com).

we aim to use small counters to record both small arrival rates and large arrival rates. If a small counter overflows, we regard this as a *rebirth*: a finite cyclic group $\mathbb{Z}_{256}$ in theory [23]. Once the rebirth occurs, it would cause the loss of the most significant bit of the frequency, which can further cause the accuracy loss of variance calculation. For example, given three counters 253, 254, and 255, suppose the incoming flow updates the counter of 255 to 0, this is a rebirth. In this case, the normal variance calculated by the reborn values ($\mathcal{F}_v\{253, 254, 0\}$=21421)[1] in the counters is very large, while actually the real variance ($\mathcal{F}_v\{253, 254, 256\}$=2.33) is very small. In this way, we propose the *offset variance calculation*. The key idea is to offset the values in the counters by a fixed value, and recalculate the variance of these offset values to make the calculated variance close to the real value. In the above example, we can offset the values {253, 254, 0} by 128, and get {125, 126, 128} ($-128 = 128$ in $\mathbb{Z}_{256}$) respectively. The variance calculated is the same as the real value, *i.e.*, $\mathcal{F}_v\{253, 254, 256\}$=$\mathcal{F}_v\{125, 126, 128\}$. Therefore, if a flow is a steady flow, we can always accurately report it as a steady flow.

Further, our experimental results show that we obtain much higher accuracy because we can accommodate much more counters in the same memory size. Compared with the strawman solution, SteadySketch is memory efficient, accurate and fast: it achieves a 95% Precision Rate (PR) with 50KB memory in the CAIDA Dataset for finding steady flows, and the throughput has been improved by 1.73×. Finally, we implement SteadySketch on cache replacement scenario, and results show that SteadySketch can significantly improve the performance of cache hit ratio. More details are provided in Section IV. We have released our source code at GitHub [24].

Key Contributions:

- We propose and define a new problem namely finding steady flows in data streams, which has not been studied before but is important in many applications.
- We propose a novel sketch named SteadySketch to address the above problem with high accuracy and high speed in small memory.
- We conduct extensive experiments, and the results show that our solution significantly outperforms the strawman solution. Particularly, SteadySketch improves the precision rate by 81.1% for finding steady flows and decreases the Average Relative Error (ARE) by $995.3\times$.

II. Problem Statement & Related Work

A. Problem Statement

The symbols frequently used in this paper are shown in Table I.

Steady refers to the situation which continues or develops gradually without any interruptions and is not likely to change quickly. In data streams, it manifests as the arrival rate of a flow which fluctuates slightly around a fixed value without interruption for a period of time. Therefore, we characterize the steady flow from two aspects: continuity and stability.

[1] $\mathcal{F}_v(.)$ is the function of variance calculation and return the calculated variance value

TABLE I: Symbols frequently used in this paper.

Notation	Meaning
e	a distinct flow in data streams
t	current time window
w	the number of bits or counters in each bucket
$g_k(.)$	k^{th} hash function of the SteadyFilter
$f_d(.)$	d^{th} hash function of the RollingSketch
$\mathcal{P}$	the probability of replacing the flow in stage 2
$\langle e, t\rangle$	the steady flow e reported with the time window of t
$\langle e, t_s, t_e\rangle$	the persistent steady flow reported with the start time of t_s and the end time of t_e

Temporary steady flow: Given a data stream, we divide it into fixed-width time windows $w_1, w_2, w_3, \ldots$. Given a flow e and a variance threshold H, the arrival rate of e in the time windows are $r_1, r_2, r_3, \ldots$. The function of $\mathcal{F}_v(.)$ is to calculate the variance and return the variance value. If there exist p consecutive time windows $w_{t-p+1}, \ldots, w_{t-1}, w_t$, where

$$\mathcal{F}_v(r_{t-p+1}, \ldots, r_{t-1}, r_t) \leqslant H$$

and

$$r_i > 0,\ \forall i \in \{t, t-1, \ldots, t-p+1\}$$

then e is one steady flow, and we report it as $< e, t >$. t is the time window of e becoming a standard steady flow.

Persistent steady flow: The data stream is divided into multiple fixed-width time window. In each time window, there could be multiple temporary steady flows. Given a series of temporary steady flows $\{\langle e_1, t_1\rangle, \langle e_2, t_1\rangle, \langle e_1, t_2\rangle ...\}$, we could try to merge steady flows with the same flow ID. Persistent steady flows $\langle e, t_1, t_2\rangle$ will be reported, only when the steady flows $\langle e, t\rangle$ $(t_1 \leqslant t \leqslant t_2)$ are all found. It indicates that the steady process of e lasts from t_1 to t_2.

B. Membership Query

Bloom filter: A Bloom filter [25] is a compact data structure with high spatial efficiency. It uses bit groups to represent a set concisely, and can judge whether a flow belongs to the set. It consists of an array of m bits and is associated with k independent hash functions. Given a flow, it is hashed to k different mapped bits and set to 1. For membership query, the Bloom filter checks whether all k mapped bits are 1. Because it is space-saving and efficient, it is widely used and has produced many variants [26]–[28].

C. Finding Frequent Flows

CM sketch: CM sketch [22] consists of d arrays $A_i (1 \leqslant i \leqslant d)$, each array consists of w counters, and A_i is associated with a hash function $h_i(.)$. Given an incoming flow e, it increments the d mapped counter $A_i[h_i(e)]$ by 1. To query e, CM sketch only reports minimum one among the d mapped counters. CM sketch has been widely used in many scenarios and has derived

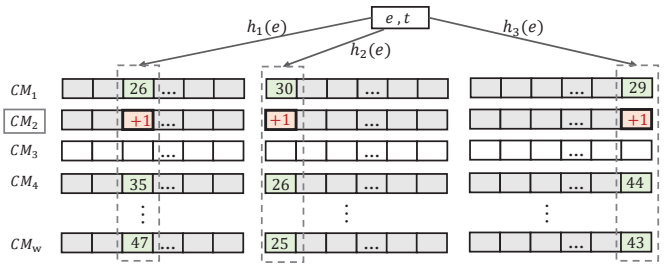


Fig. 1: Examples of insertion in strawman solution with 3 hash functions and w CM sketches. The counters marked green are frequencies of previous p time windows and the sketch CM_2 is selected to represent the current time window.

many variants, such as CU [29], Count [30], and many other typical sketches [10], [31]–[34]

Space-Saving and Unbiased Space-Saving: Both Space-Saving [35] and Unbiased Space-Saving [36] use a data structure called Stream-Summary to keep the top-k frequent flows. When the Stream-Summary is full and a non-recorded flow arrives, Space-Saving directly replaces the least frequent flow with this new flow, while Unbiased Space-Saving uses probability to replace the least frequent flow to achieve unbiased estimation.

III. The SteadySketch

In this section, we introduce different solutions for finding steady flows. First, we propose the strawman solution based on CM sketch [22]. Further, we propose optimization schemes and present the final version in III-E. Finally, we discuss the difference of temporary and persistent steady flows, and add a new data structure for finding persistent steady flows.

A. The Strawman Solution

As shown in **Figure 1**, our strawman solution is based on CM sketch [22]. We construct the strawman solution with w CM sketches, in which p sketches to contain the flow information in past p time windows, one CM sketch is used to contain the flow information of current time, and the other is reserved for the next time window. Thus, the w must be set greater than $p + 2$.

For each incoming flow e with time window t, the CM sketch (Section II-C) representing the current time hash it to a bucket in each array, and increment the counters by 1. Then, the variance of frequency is calculated by the counters of the mapped buckets in previous p sketches. If the variance is less than the steady threshold H, we report the steady flow $\langle e, t\rangle$.

The strawman solution could indeed be used to report steady flows. However, its low accuracy, low throughput, and high memory consumption make it difficult to apply in practice.

B. Speed Optimization

For calculating the variance, the strawman solution needs multiple access to the sketches, which highly reduce the throughput. Inspired by the principle of locality, we set the counters in the same location of different sketches into one bucket. Thus, we merge the multiple sketches to one sketch, and reduce the time complexity to $O(1)$, which greatly improves the throughput.

C. Accuracy Optimization

The key idea of accuracy optimization lies in continuity checking. It refers to checking if the flow appears consecutively in previous p time windows. Due to limited memory and hash collisions, it is hard to determine whether it is interrupted for some small flows. Thus, we propose SteadyFilter to be responsible for continuity checking. Inspired by the Bloom Filter [25], we construct the buckets of array with multiple bits to record the appearance of a flow in multiple time windows, which greatly improve the accuracy.

D. Memory Optimization

The key novelty of this paper lies in this optimization. The main innovation is the proposal of the reborn technique. In SteadySketch, we reduce the counters to 8 bits and utilize the reborn technique to deal with the overflow caused by elephant flows. The key of reborn technique is rebirth and offset variance calculation.

Rebirth: Inspired by the concept of group, the value of an 8-bit counter can be regarded as a finite cyclic group $G = \mathbb{Z}_{256} = \{0, 1, \cdots, 255\}$. Once the frequency is greater than 255, the rebirth is triggered and the value increments from 0 again.

Offset Variance Calculation: Once the rebirth is performed, it indicates that the frequency has not been well recorded. Thus, we propose the *offset variance calculation* to ensure the accuracy of variance calculation. First, we calculate the normal variance using the raw data as usual. Next, we calculate the offset variance. The innovation is that we offset the frequencies by a fixed value: we increment the values by 128.

Example: A steady flow appears 248, 258, 260 times in three time windows, respectively, and it is recorded as 248, 2, 4 in counters. In the normal variance calculation, the variance result is 13339, while the real frequency variance is 27. It is greatly larger than the real value. In the offset variance calculation, the frequency is recalculated as 120, 130, 132, and the variance value is the same as the real variance.

E. Our Final Version

Integrating the above three techniques, SteadySketch includes two parts: a SteadyFilter and a RollingSketch. We would introduce the two parts in details below.

1) ***SteadyFilter****:*

Similar to the typical Bloom filter [25], SteadyFilter consists of k bucket arrays $\mathcal{A}_1$, $\mathcal{A}_2$, $\cdots$, $\mathcal{A}_k$, and k hash functions $g_1(.)$, $g_2(.)$, $\cdots$, $g_k(.)$. Each bucket consists of w bits, which represent the appearance of a flow in w time windows. When querying, if the i^{th} bit in the bucket is set, it indicates that the flow has reached in the i^{th} time window.

Figure 2 shows the data structure of SteadyFilter with the version of one hash function. In this figure, e_1,e_2 and e_3 are the flows inserted into SteadyFilter at time of t_1, t_2 and t_3 respectively. The bits marked red represent the time windows of t_1, t_2 and t_3 in the mapped buckets, while the green ones

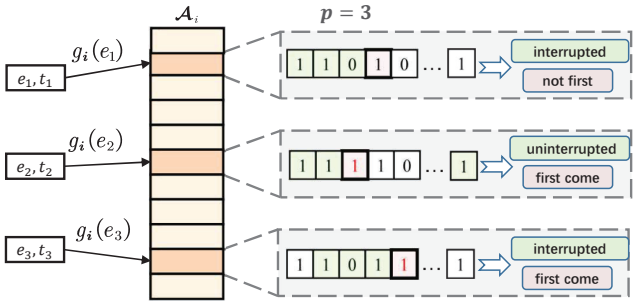


Fig. 2: Examples of insertion in SteadyFilter with one hash function. All the buckets are initialized to (1, 1, 0, 1, 0, $\cdots$, 1). The bits marked red represent the current time window, while the green ones are previous p time windows.

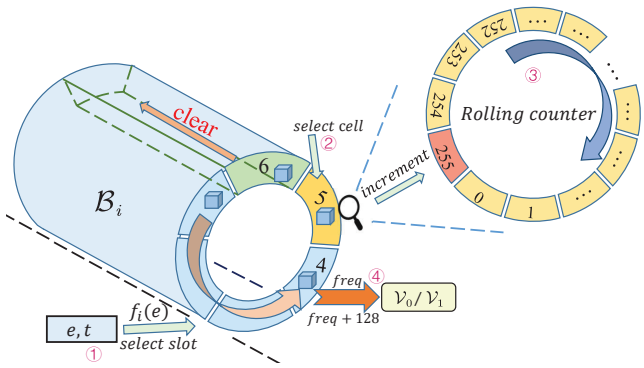


Fig. 3: Examples of insertion in RollingSketch with one hash function. The array consist of m slots and w cells in each slot and the parameter p is set as 4. In addition, ① ② ③ ④ in the figure represent the sequence of insertion.

Algorithm 1: Operations Procedure for RollingSketch

Input: input a coming flow e with the timestamp t
Output: output the steady flow (e, t)

for $i \leftarrow 1$ **to** d **do**
  $B[f_i(e)][t \ \% \ w] \ ++;$
  if $T_{pf} == 1$ **then**
    for $j \leftarrow 1$ **to** p **do**
      $t_l = (t \ - \ j) \ \%w;$
      $G[0][j] \leftarrow min(G[0][j], B[f_i(e)][t_l]);$
      $G[1][j] \leftarrow G[0][j] + 128;$
$\mathcal{V}_0 = \mathcal{F}_v(G[0])$; // $\mathcal{F}_v$ is the function of variance calculation
$\mathcal{V}_1 = \mathcal{F}_v(G[1])$;
if $\mathcal{V}_0 < H \parallel \mathcal{V}_1 < H$ **then**
  report steady flow $< e, t >$;

Algorithm 2: Clear Procedure

Input: current time window t and the time window t_p of last flow

if $t \neq t_p$ **then**
  for *each* $i \in [1, d]$ **do**
    t_n = $(t \ + \ 1) \ \%\ w$;
    for *each* $j \in [0, \ len(A_i)]$ **do**
      $\mathcal{A}_i[j][t_n]$ = 0;
    for *each* $j \in [0, \ len(B_i)]$ **do**
      $\mathcal{B}_i[j][t_n]$ = 0;

are the previous p time windows. If a flow is recognized as uninterrupted and first come in the current time window, it would be marked with T_{pf}, like e_2 in **Figure 2**.

*2) **RollingSketch**:*

RollingSketch consists of d arrays, and each array is associated with a hash function. **Figure 3** shows the data structure of RollingSketch with one array, which looks like a cylinder. In this paper, the cylinder is set to be cut into m slots, and each slot consists of w cells. Each cell records the flow frequency in a time window. Thus, a slot can record the flow frequencies of w time windows.

Insertion (Algorithm 1): We first select the slots $\mathcal{B}_1[f_1(e)]$, $\mathcal{B}_2[f_2(e)]$, $\cdots$, $\mathcal{B}_d[f_d(e)]$ in each array with hash functions $f_1(.)$, $f_2(.)$, $\cdots$, $f_d(.)$. We use the $\mathcal{S}_i (1 \leqslant i \leqslant d)$ to denote the slot selected in each array. Then, we select the cell $\mathcal{S}_i[t\%w]$ and increment it by 1. The insertion process is shown in **Figure 3** and marked as ① ② and ③.

Report (Algorithm 1): First, we select the values in the cells of $\mathcal{S}_i[(t-j)\%w] (1 \leqslant j \leqslant p)$ and perform the *offset variance calculation* (Section III-D) to calculate the variance $\mathcal{V}_0$ and $\mathcal{V}_1$. Once one of the two variances is less than the threshold H, we report the flow e as the steady flow $\langle e, t \rangle$, where t represents the time when the flow e becomes a steady flow.

Clear Strategy (Algorithm 2): As the time window increases, there are not enough bits and cells in each bucket to satisfy all time windows. If we select the cells representing the current time window to clear, we have to put the incoming packets into the buffer queue first, which is not practical in high-speed data streams. Therefore, we select the bits and cells of the next time window in the SteadyFilter and RollingSketch, and set them to 0 when the time window switches. In this way, the clear operation can be implemented in parallel without much impact on throughput.

*3) **Summary** :*

In SteadySketch, we separately design different processing schemes for various types of flows in data streams as follows:

Case 1: The elephant flows. They might lead to overflows in counters. We propose the reborn technique to offset the accuracy loss caused by overflow.

Case 2: The medium-size flows. Their frequencies can be well recorded and calculated by the counters normally.

Case 3: The small flows. It is hard to identify the continuity of a flow in limited memory. In this case, we use SteadyFilter to specifically identify the continuity of flows.

F. Finding Temporary and Persistent Steady flows

As for finding temporary steady flows, the above algorithm is fully capable. However, it cannot effectively report persistent

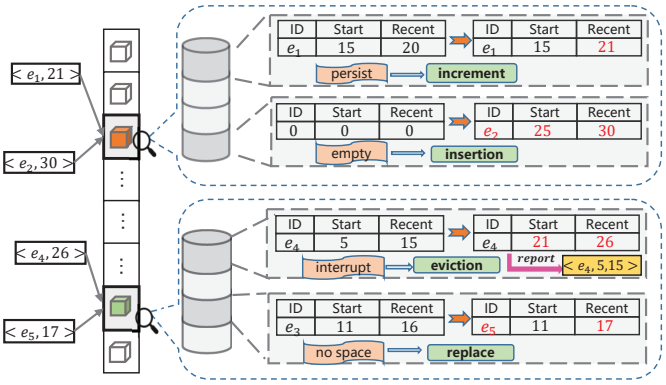


Fig. 4: Insertion examples in Stage 2 and the parameter p is set as 5.

steady flows. Thus, we add **Stage 2** after the RollingSketch for finding persistent steady flows. **Figure 4** shows the data structure and the insertion process of stage 2. Below we show the details.

Data Structure: Stage 2 is constructed with an array of l buckets, and each bucket has four cells. Let G_i be the i^{th} bucket, and $G_i[j]$ be the j^{th} cell in bucket G_i. For each cell, it is designed with the following three fields: 1) A `flow_ID` field $G_i[j].ID$ records the ID of the steady flows, and we call the flow in the cell as the $residing\ flow$. 2) A `start time` field $G_i[j].t_s$ records the start time of the steady flow as a residing flow. 3) A `recent time` field $G_i[j].t_e$ records the recent time of the steady flow as a residing flow. The duration of a steady flow is represented by the interval ΔT, $i.e.$, $\Delta T = G_i[j].t_e - G_i[j].t_s$.

Insertion: When inserting flow e into Stage 2, we first map the flow to a bucket G_v by computing the hash function $h(e)$. Next, we try to insert it. There are four cases as follows:

Case 1: **Insertion**. If e is not in the bucket and there is still at least one empty cell. We select an empty cell, insert $\langle e, t\rangle$ into the bucket, and set the ID of e to $G_v[j].ID$, the timestamp t_s, t_e to $t-p$ and t, respectively.

Case 2: **Eviction**. If the recent time t_e of the residing flow e_p in the cell is not the last time $t\ -\ 1$, it means that e_p is no longer steady. Thus, we report the flow e_p as a persistent steady flow $\langle e_p, G_v[j].t_s, G_v[j].t_e\rangle$. Then, the fields in cell are replaced by $[e,\ t-p,\ t]$, respectively.

Case 3: **Increment.** If e is in the bucket, and $G_v[j].t_e$ equals to $t\ -\ 1$. It shows that e is still steady. We increment $G_v[j].t_e$ by 1.

Case 4: **Replacement**. If the bucket cannot satisfy the above three cases, it means that all flows in the bucket are still steady. In that case, we use ΔT to find the shortest persistent steady flow, and replace it with a certain probability. Then, we kick the residing flow with $\mathcal{P}$, and replace $G_v[j].ID$ and $G_v[j].t_e$ with $\langle e, t\rangle$, respectively, without $G_v[j].t_s$ replacement. The definition of $\mathcal{P}$ is shown in Equation (1):

$$\mathcal{P} = \frac{1}{t_e -\ t_s - p} \tag{1}$$

IV. Experimental Results

A. Experimental Setup

Datasets: We use a total of three real-world datasets and one synthetic dataset as follows.

1) CAIDA Dataset: This dataset is streams of anonymized IP traces collected by CAIDA [37]. For CAIDA Dataset, there are around 30M flows and 900K distinct flows.

2) Campus Dataset: This dataset is comprised of IP packets captured from the network of our campus. For Campus Dataset, there are 10M flows in total, belonging to 1M distinct flows.

3) MAWI Dataset: This real traffic traces data is provided by the MAWI Working Group [38]. For MAWI Dataset, there are around 13M flows.

4) Synthetic Datasets: We generate a synthetic dataset that follows the Zipf [39] distribution using Web Polygraph [40], an opensource performance testing tool. For Synthetic Datasets, there are around 32M flows, and the skewness is 1.5.

For the above datasets, we divide each dataset into multiple time windows, and set the window size around 10K flows.

Implementation: We implement SteadySketch and the strawman solution in C++. The hash functions are implemented using the 32-bit Murmur Hash (obtained from the open-source website [41]) with different initial seeds.

Computation Platform: We conduct all the experiments on a machine with one 8-core processor (8 threads, Intel(R) Core(TM) i7-9700U CPU @ 3.00GHz) and 16 GB DRAM memory. The processor has 512KB L1 cache, 2MB L2 cache for each core, and 12MB L3 cache shared by all cores.

Metrics:

1) Precision Rate (PR): PR is the ratio of the number of correctly steady flows to the number of steady flows reported.

2) Recall Rate (CR): CR is the ratio of the number of correctly reported steady flows to the number of correctly steady flows.

3) Mean Squared Error (MSE): We define the MSE as $\frac{1}{n}\sum_{i=1}^{n}\left(\mathcal{V}_i - \hat{\mathcal{V}}_i\right)^2$, where $\mathcal{V}_i$ is the real variance of steady flow e_i, $\hat{\mathcal{V}}_i$ is the estimated variance of steady flow, and the n is the correct number of steady flows reported.

4) Average Relative Error (ARE): We define the ARE as $\frac{1}{|\Psi|}\sum_{e_i\in\Psi}\frac{|f_i - \hat{f}_i|}{f_i}$, where f_i is the real duration of persistent steady flow e_i, $\hat{f}_i$ is its estimated duration of persistent steady flow, and Ψ is the query set.

5) Throughput: We use Million of operations (insertions) per second (Mops) to measure the throughput. Experiments are repeated 10 times and the average throughput is reported.

B. Experiments on Parameter Settings

Main Parameters: 1) The number of hash functions k in SteadyFilter; 2) The ratio r of the memory size of SteadyFilter to the memory size of the whole SteadySketch; 3) The number of hash functions d in GroupSketch; 4) The the variance threshold H for the steady flows; and 5) The threshold p for time window period.

Parameter Settings: We set $k = 3$, $r = 20\%$, $d = 2$, $H = 5$, and $p = 5$.

C. Finding Temporary Steady Flows

In this section, we compare the performance of SteadySketch in finding temporary steady flows with the strawman solutions in the metrics below.

1) PR (Figure 5(a) - 5(d)): *This experiment shows that the PR of SteadySketch is greatly outperforms the strawman solution.* We find that, on the synthetic dataset, the PR of SteadySketch is around 83.3% higher than the strawman solution on average. On the three real-world datasets, the PR of SteadySketch is around 81.1% higher than the strawman solution on average.

2) CR (Figure 6(a) - 6(d)): *This experiment shows that the CR of SteadySketch is much higher than the strawman solution.* We find that, on the synthetic dataset, the CR of SteadySketch is around 16.6% higher than the strawman solution on average. On the three real-world datasets, the CR of SteadySketch is around 29.9% higher than the strawman solution on average.

3) MSE (Figure 7(a) - 7(d)): *This experiment shows that the MSE of SteadySketch is obviously lower than the strawman solution.* We find that, on the synthetic dataset, the MSE of SteadySketch is around 2.93× lower than the strawman solution on average. On the three real-world datasets, the MSE of SteadySketch is around 3.22× lower than the strawman solution on average.

4) Throughput (Figure 8(a) - 8(d)): *This experiment shows that the throughput of SteadySketch is significantly higher than the strawman solution.* We find that, on the synthetic dataset, the throughput of SteadySketch is around 1.64× higher than the strawman solution on average. On the three real-world datasets, the throughput of SteadySketch is around 1.62× higher than the strawman solution on average.

Analysis: Our experimental results show that SteadySketch achieves higher accuracy and higher throughput.

Compared with the strawman solution, SteadySketch reduce the size of the counter to one quarter, thereby increasing the number of counters by 4×. The reborn technique makes the frequent flows not cause the accuracy loss of variance. Thus, more counters in the same space without loss of accuracy leads to the MSE variance significantly reduced. In addition, compared with the huge superiority of strawman in precision rate, the advantage in recall is not so significant, especially in synthetic dataset. By comparing the results reported by Strawman and SteadySketch, it is found that Strawman reports dozens of times more steady flows than the groundtruth, which contains most steady flows. The reason for this is that there are too many small flows hash collision. As for the synthetic dataset, it follows the Zipf distribution and the skewness is set to 1.5. It would have more small flows compared with the real-world dataset such as CAIDA, which makes the phenomenon more serious.

D. Finding Persistent Steady Flows

In this section, we compare the performance of SteadySketch in finding persistent steady flows with the strawman solutions in the metrics below. Here, m refers to the memory of Stage 2. The total memory varies from 300KB to 600KB, and m has been included and fixed as 150KB.

5) PR (Figure 9(a)-9(d)): *This experiment shows that the PR of SteadySketch is largely outperforms the strawman solution.* we find that the PR of SteadySketch is 53.6%, 70.6%, 43.9%, and 62.2% higher than the strawman solution on average on CAIDA, campus, MAWI, synthetic dataset, respectively.

6) CR (Figure 10(a)-10(d)): *This experiment shows that the CR of SteadySketch is much better than the strawman solution.* We find that the CR of SteadySketch is 34.6%, 49.1%, 27.3%, and 41.8% higher than the strawman solution on average on four datasets, respectively.

7) ARE (Figure 11(a)-11(d)): *This experiment shows that the ARE of SteadySketch is obviously lower than the strawman solution.* We find that the ARE of SteadySketch is 825.3×, 585.3×, 121.8×, and 2288.7× lower than the strawman solution on average, repectively. It is up to 1947× in the CAIDA Dataset, and 5129× in the Synthetic Dataset. On average, the ARE of SteadySketch is 955.3× smaller than the strawman solution.

8) Throughput (Figure 12(a)-12(d)): *This experiment shows that the throughput of SteadySketch is obviously higher than the strawman solution.* We find that the throughput of SteadySketch is 1.73×, 1.78×, 1.63×, and 1.67× higher than the strawman solution, respectively.

Analysis: As the result shows that our solution gets better performance in accuracy, throughput and ARE. Since the superior performance of stage 1, the flows inserted into stage 2 are in high accuracy. Compared with SteadySketch, strawman solution reports a large number of false positive flows in stage 1, which results in lower throughput and accuracy.

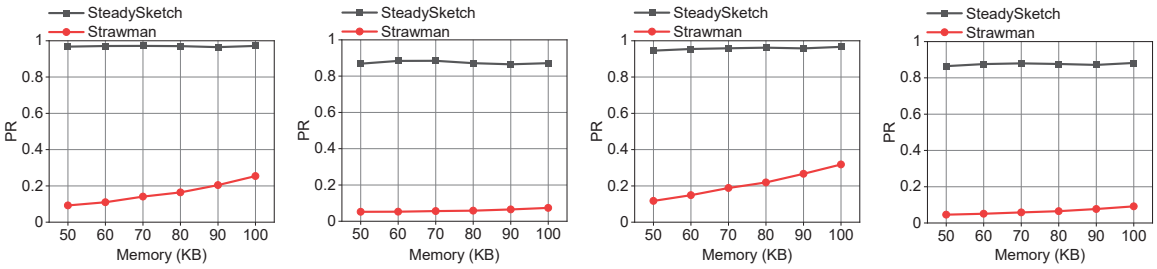
E. Cache Replacement Optimization

To the best of our knowledge, modern caches often adopt Least Recently Used (LRU) eviction strategy. In this section, however, we creatively introduce **SteadySketch** to ***predict*** the coming cache line and thus improve the cache hit rate [2]. Our success depends on a reasonable hypothesis: Once we consider a cache line is steady, it is probably fetched in the near future.

Experimental Setup: We use LRU as the comparison replacement strategy and carry out C++ simulation experiment. In our algorithm, we divide the cache into 3 parts: A SteadySketch (100KB, $k = 3$, $d = 2$, $H = 5$ and $p = 5$), a fully-associative ***steady part*** and a fully-associative ***general part***. Both the steady part and the general part can be seen as a small LRU cache. When we fetch a new cache line, we first insert its address into the SteadySketch to check if it is a steady cache line: A steady cache line will be fetched into the steady part and will not be evicted by unsteady ones; While an unsteady cache line will be fetched into the general part and will not be evicted by steady ones.

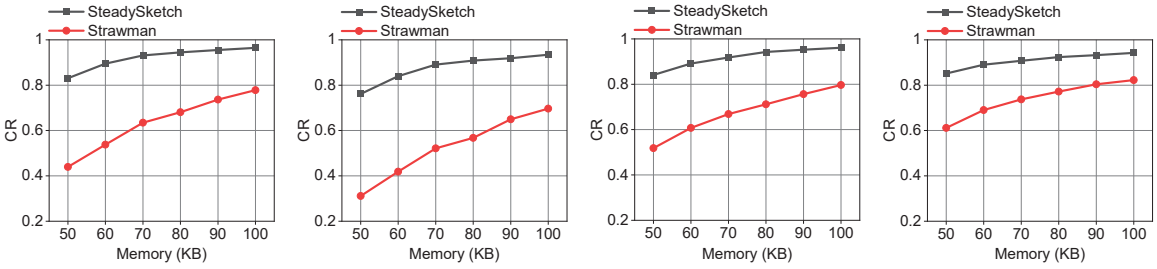
Experimental on Real-World Dataset: We conduct experiments on Campus dataset, and use $P_{M\%}$ to denote the case

[2]The cache hit rate can be calculated as (the number of cache hits)/(the number of memory accesses).



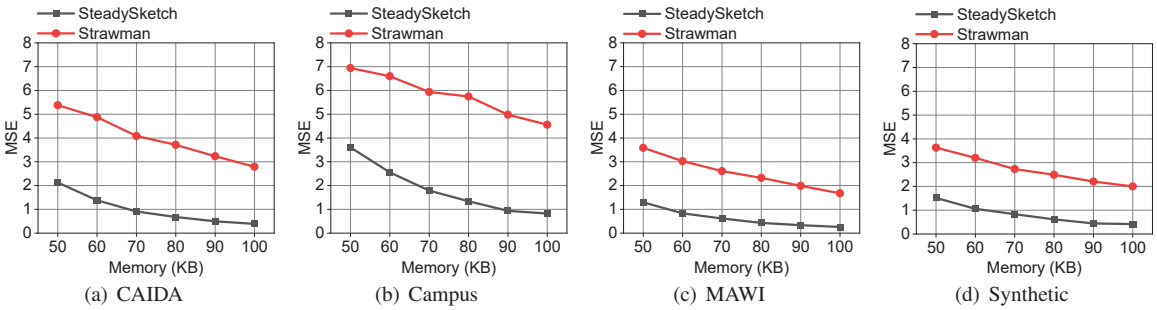
(a) CAIDA (b) Campus (c) MAWI (d) Synthetic

Fig. 5: Temporary Steady Flows: Precision Rate (PR) vs. memory.



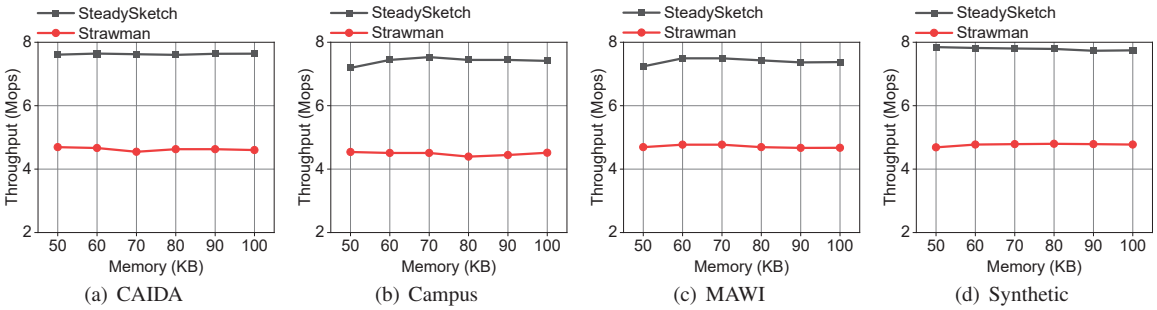
(a) CAIDA (b) Campus (c) MAWI (d) Synthetic

Fig. 6: Temporary Steady Flows: Recall Rate (CR) vs. memory.



(a) CAIDA (b) Campus (c) MAWI (d) Synthetic

Fig. 7: Temporary Steady Flows: Mean Squared Error (MSE) vs. memory.



(a) CAIDA (b) Campus (c) MAWI (d) Synthetic

Fig. 8: Temporary Steady Flows: Throughput vs. memory.

when the general part takes up $M\%$ of the memory of the cache. The experimental results are shown in **Figure 13(a)**: We find that under a limited cache memory, using SteadySketch can significantly improve the cache hit rate by up to 13.02% in a wide range of cache size. So our SteadySketch provides a new way to optimize cache replacement problem.

Experimental On Steady-Synthetic Dataset: Our Steady-Synthetic Dataset a mixture of steady flows and non steady flows and has 10^7 flows totally. The $P_S(\%)$ is the ratio of steady flows in the dataset, while the other flows are random flows, most of them only appear for a short time. In the Steady-Synthetic Dataset experimental, we fix the cache size as 12K and increase the ratio of steady flows in the dataset to observe the cache hit rate of different algorithms. The experimental results are shown in **Figure 13(b)**. As we can see, SteadySketch can significantly improve the cache hit ratio

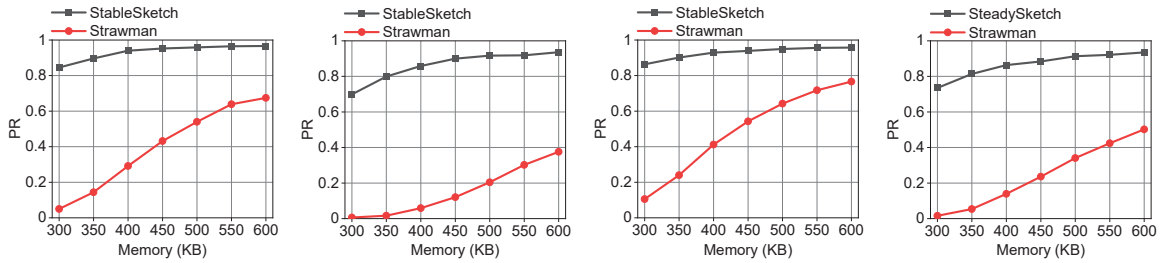


Fig. 9: Persistent Steady Flows: Precision Rate (PR) vs. memory.

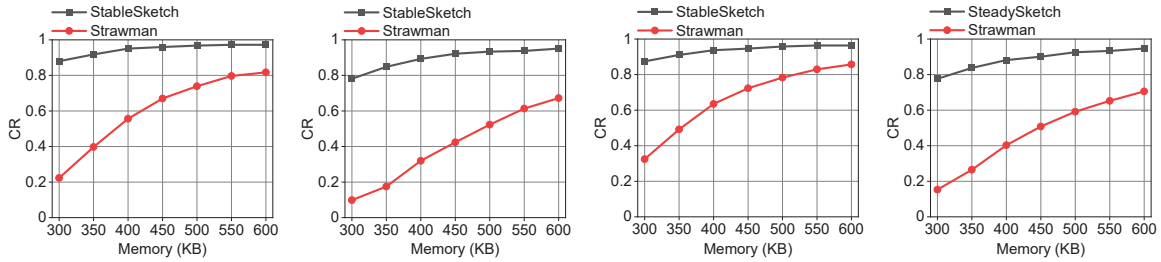


Fig. 10: Persistent Steady Flows: Recall Rate (PR) vs. memory.

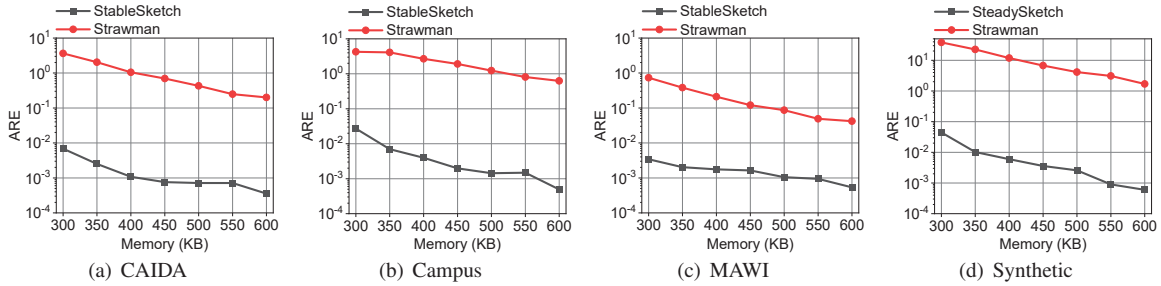


Fig. 11: Persistent Steady Flows: Average Relative Error (ARE) vs. memory.

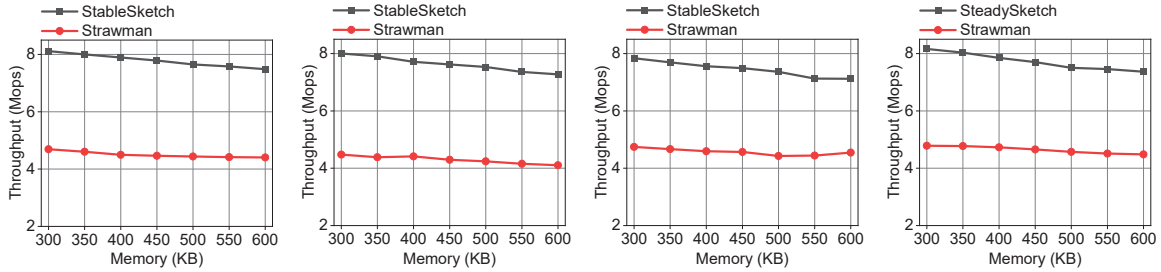
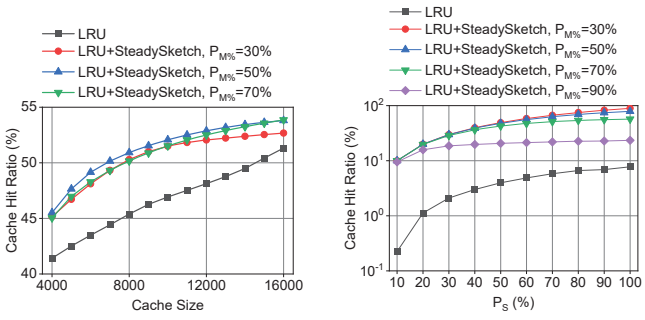


Fig. 12: Persistent Steady Flows: Throughput vs. memory.

compared with typical LRU, especially when the $P_{M\%}$ is relatively small. It means that the larger the proportion of SteadySketch, the more significant the effect will be, when the P_S is set to 30%, the accuracy increased by an average of $44\times$.

V. Conclusion

Finding steady flows in high-speed data streams in real-time is important in many applications. In this paper, we propose a novel algorithm called SteadySketch for real-time steady flows detection, which is fast, memory-efficient, and accurate. Experimental results show that the SteadySketch can achieve 81.1% higher PR, $1.73\times$ higher throughput, and up to $955.3\times$ lower ARE than the strawman solution. Finally, we implement our SteadySketch on a concrete case: cache replacement, and the experiment verifies that SteadySketch can significantly improve the Cache hit ratio.



(a) Campus (b) Steady-Synthetic

Fig. 13: The Cache hit ratio comparison.

ACKNOWLEDGMENT

This work was supported in part by the Key-Area Research and Development Program of Guangdong Province under Grant 2020B0101390001 and in part by the National Natural Science Foundation of China (NSFC) under Grant U20A20179 and Grant 61832001.

REFERENCES

[1] L. Li, K. Xu, T. Li, K. Zheng, C. Peng, D. Wang, X. Wang, M. Shen, and R. Mijumbi, "A measurement study on multi-path tcp with multiple cellular carriers on high speed rails," in *Proc. SIGCOMM*, 2018, pp. 161–175.

[2] L. Li, K. Xu, D. Wang, C. Peng, Q. Xiao, and R. Mijumbi, "A measurement study on tcp behaviors in hspa+ networks on high-speed rails," in *Proc. INFOCOM*, 2015, pp. 2731–2739.

[3] J. Fan, X. Ming-wei, C. Yong, and X. Ke, "Real-time measurement of local qos states," in *Proc. TENCON*, vol. 100, 2004, pp. 48–51.

[4] M. Shen, J. Zhang, K. Xu, L. Zhu, J. Liu, and X. Du, "Deepqoe: Real-time measurement of video qoe from encrypted traffic with deep learning," in *Proc. IWQoS*, 2020, pp. 1–10.

[5] Y. Lei, L. Yu, V. Liu, and M. Xu, "Printqueue: performance diagnosis via queue measurement in the data plane," in *Proc. SIGCOMM*, 2022, pp. 516–529.

[6] M. Wyss, G. Giuliari, M. Legner, and A. Perrig, "Secure and scalable qos for critical applications," in *Proc. IWQoS*, 2021, pp. 1–10.

[7] M. Chen, M. Xu, Q. Li, and Y. Yang, "Measurement of large-scale bgp events: Definition, detection, and analysis," *Computer Networks*, vol. 110, pp. 31–45, 2016.

[8] M. Chen, M. Xu, Y. Yang, and Q. Li, "A measurement study on the distribution disparity of bgp instabilities," in *Proc. LCN*, 2016, pp. 19–27.

[9] Q. Huang and P. P. Lee, "Ld-sketch: A distributed sketching design for accurate and scalable anomaly detection in network data streams," in *Proc. INFOCOM*, 2014, pp. 1420–1428.

[10] Z. Zhong, S. Yan, Z. Li, D. Tan, T. Yang, and B. Cui, "Burstsketch: Finding bursts in data streams," in *Proc. SIGMOD*, 2021, pp. 2375–2383.

[11] Z. Fan, Y. Zhang, T. Yang, M. Yan, G. Wen, Y. Wu, H. Li, and B. Cui, "Periodicsketch: Finding periodic items in data streams," in *Proc. ICDE*, 2022, pp. 96–109.

[12] J. Liu, H. Dai, R. Xia, M. Li, R. B. Basat, R. Li, and G. Chen, "Duet: A generic framework for finding special quadratic elements in data streams," in *Proc. WWW*, 2022, pp. 2989–2997.

[13] R. Huysegems, B. De Vleeschauwer, K. De Schepper, C. Hawinkel, T. Wu, K. Laevens, and W. Van Leekwijck, "Session reconstruction for http adaptive streaming: Laying the foundation for network-based qoe monitoring," in *Proc. IWQoS*, 2012, pp. 1–9.

[14] E. Cole, *Advanced Persistent Threat: Understanding the Danger and How to Protect Your Organization*. Syngress Publishing, 2012.

[15] B. Lahiri, S. Tirthapura, and J. Chandrashekar, "Space-efficient tracking of persistent items in a massive data stream," *The ASA Data Science Journal*, vol. 7, no. 1, pp. 70–92, 2014.

[16] H. Dai, M. Shahzad, A. X. Liu, M. Li, Y. Zhong, and G. Chen, "Identifying and estimating persistent items in data streams," *IEEE/ACM Transactions on Networking*, vol. 26, no. 6, pp. 2429–2442, 2018.

[17] H. Dai, M. Li, A. X. Liu, J. Zheng, and G. Chen, "Finding persistent items in distributed datasets," *IEEE/ACM Transactions on Networking*, pp. 1–14, 2019.

[18] Y. Zhang, J. Li, Y. Lei, T. Yang, Z. Li, G. Zhang, and B. Cui, "On-off sketch: A fast and accurate sketch on persistence," *Proc. VLDB Endow.*, vol. 14, no. 2, pp. 128–140, 2020.

[19] H. Huang, Y.-E. Sun, S. Chen, S. Tang, K. Han, J. Yuan, and W. Yang, "You can drop but you can't hide: k-persistent spread estimation in high-speed networks," in *Proc. INFOCOM*, 2018, pp. 1889–1897.

[20] H. Huang, Y.-E. Sun, C. Ma, S. Chen, Y. Zhou, W. Yang, S. Tang, H. Xu, and Y. Qiao, "An efficient k-persistent spread estimator for traffic measurement in high-speed networks," *IEEE/ACM Transactions on Networking*, vol. 28, no. 4, pp. 1463–1476, 2020.

[21] Y.-E. Sun, H. Huang, S. Chen, Y. Zhou, K. Han, and W. Yang, "Privacy-preserving estimation of k-persistent traffic in vehicular cyber-physical systems," *IEEE Internet of Things Journal*, vol. 6, no. 5, pp. 8296–8309, 2019.

[22] G. Cormode and S. Muthukrishnan, "An improved data stream summary: the count-min sketch and its applications," *Journal of Algorithms*, vol. 55, no. 1, pp. 58–75, 2005.

[23] S. Shahriar, *Algebra in Action: A Course in Groups, Rings, and Fields*. American Mathematical Soc., 2017, vol. 27.

[24] "The source code of SteadySketch." [Online]. Available: https://github.com/steadysketch/SteadySketch

[25] B. H. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Communications of the ACM*, vol. 13, no. 7, pp. 422–426, 1970.

[26] O. Rottenstreich, Y. Kanizo, and I. Keslassy, "The variable-increment counting bloom filter," *IEEE/ACM Transactions on Networking*, vol. 22, no. 4, pp. 1092–1105, 2013.

[27] J. Lu, T. Yang, Y. Wang, H. Dai, X. Chen, L. Jin, H. Song, and B. Liu, "Low computational cost bloom filters," *IEEE/ACM Transactions on Networking*, vol. 26, no. 5, pp. 2254–2267, 2018.

[28] Y. Wu, J. He, S. Yan, J. Wu, T. Yang, O. Ruas, G. Zhang, and B. Cui, "Elastic bloom filter: deletable and expandable filter using elastic fingerprints," *IEEE Transactions on Computers*, vol. 71, no. 4, pp. 984–991, 2021.

[29] C. Estan and G. Varghese, "New directions in traffic measurement and accounting: Focusing on the elephants, ignoring the mice," *ACM Transactions on Computer Systems*, vol. 21, no. 3, pp. 270–313, 2003.

[30] M. Charikar, K. Chen, and M. Farach-Colton, "Finding frequent items in data streams," *Theoretical Computer Science*, vol. 312, no. 1, pp. 3–15, 2004.

[31] T. Yang, J. Jiang, P. Liu, Q. Huang, J. Gong, Y. Zhou, R. Miao, X. Li, and S. Uhlig, "Elastic sketch: Adaptive and fast network-wide measurements," in *Proc. SIGCOMM*, 2018, pp. 561–575.

[32] Y. Zhang, Z. Liu, R. Wang, T. Yang, J. Li, R. Miao, P. Liu, R. Zhang, and J. Jiang, "Cocosketch: High-performance sketch-based measurement over arbitrary partial key query," in *Proc. SIGCOMM*, 2021, pp. 207–222.

[33] Q. Huang, X. Jin, P. P. Lee, R. Li, L. Tang, Y.-C. Chen, and G. Zhang, "Sketchvisor: Robust network measurement for software packet processing," in *Proc. SIGCOMM*, 2017, pp. 113–126.

[34] L. Tang, Q. Huang, and P. P. Lee, "A fast and compact invertible sketch for network-wide heavy flow detection," *IEEE/ACM Transactions on Networking*, vol. 28, no. 5, pp. 2350–2363, 2020.

[35] A. Metwally, D. Agrawal, and A. El Abbadi, "Efficient computation of frequent and top-k elements in data streams," in *Proc. ICDT*, 2005, pp. 398–412.

[36] D. Ting, "Data sketches for disaggregated subset sum and frequent item estimation," in *Proc. SIGMOD*, 2018, pp. 1129–1140.

[37] "The CAIDA Anonymized Internet Traces." [Online]. Available: http://www.caida.org/data/overview/

[38] "MAWI Working Group Traffic Archive." [Online]. Available: http://mawi.wide.ad.jp/mawi/

[39] D. M. Powers, "Applications and explanations of zipf's law," in *Proc. NeMLaP3/CoNLL*, 1998, pp. 151–160.

[40] A. Rousskov and D. Wessels, "High-performance benchmarking with web polygraph," *Software: Practice and Experience*, vol. 34, no. 2, pp. 187–211, 2004.

[41] "The source code of Murmur Hash." [Online]. Available: https://github.com/aappleby/smhasher