MCSketch: An Accurate Sketch for Heavy Flow Detection and Heavy Flow Frequency Estimation

Abstract. Accurately finding heavy flows in data streams is challenging owing to limited memory availability. Prior algorithms have focused on accuracy in heavy flow detection but cannot provide the frequency of a heavy flow exactly. In this paper, we designed a two-mode counter, called Matthew Counter, for the efficient use of memory and an accurate record flow frequency. The key ideas in Matthew Counter are the use of idle high-bits in the counter and the adoption of a power-weakening method. Matthew Counter allows sufficient competition during the early stages of identifying heavy flows and amplifying the relative advantage when the counter is sufficiently large to ensure the level of accuracy. We also present an invertible sketch, called MCSketch, for supporting heavy-flow detection with small and static memory based on Matthew Counter. The experiment results show that MCSketch achieves a higher accuracy than existing algorithms for heavy flow detection. Moreover, MCSketch reduces the average relative error by approximately 1 to 3 orders of magnitude in comparison to other state-of-art approaches.

Keywords: Data stream mining, Heavy Flow Detection, Power-weaking Increment, Sketch.

1 Introduction

Data stream processing is a significant issue in many applications, including natural language processing [8], financial data trackers [2], [3], and network security [24, 27]. Heavy flow detection is one of the most fundamental tasks in data stream processing. A heavy flow means that its frequency is over a predefined threshold. Finding heavy flows in a data stream with limited memory is a challenging task. It is impossible to accurately track all flows because recording massive data streams wastes a significant amount of memory. Therefore, a sketch as a compact data structure has become popular and has gained wide acceptance in data stream processing [2, 15, 16, 21].

Many researchers have applied a sketch to heavy flow detection and achieved remarkable results with limited memory, including MVSketch [20], HeavyKeeper [7], and WavingSketch [14]. Nevertheless, the detection accuracy of these existing algorithms is insufficient, and the estimation error of the frequency of a heavy flow is large. The error of the flow frequency given under certain scenarios is twice the real frequency of a heavy flow.

To understand the difficulty of estimating the flow frequency, suppose we have one counter that can record the candidate key and count the frequencies of the candidate and other flows. Intuitively, when an incoming item belongs to the candidate key, the count of candidate flows increases by 1; otherwise, the count of the other flows increases by 1. When the count of the other flows is larger than that of the candidate flow, the key of the incoming item is used as a new candidate flow. However, if the frequency of the heavy flow is not significantly greater than that of the other flows, it will be difficult to detect a heavy flow based on the previous strategy. Supposing we have a data stream <A, A, A, B, C, A, B, C>, the record is then (A,1,0), (A,2,0), (A,3,0), (A,3,1), (A,3,2), (A,3,2), (A,3,3), (C,1,0) according to the above strategy. The final recorded heavy flow is C, which is not a real heavy flow.

To solve this problem, we are inspired by the Matthew Effect, which can be summarized by the adage "the rich get richer and the poor get poorer." Specifically, we adopt a probabilistic method called a power-weakening increment for items that do not belong to the current candidate flow. For example, when the incoming item belongs to the candidate key, the count of the candidate flows increases by 1; otherwise, the count of the other flows increases by 1 with the probability of the inverse of the current candidate flow count. For data stream <A, A, A, B, C, A, B, C>, the record is (A,3,0) after three items are inserted. For the fourth item B, the count of other flows increases by 1 with a probability of 1/3 , and thus the probability of candidate flow A being replaced is significantly reduced. Using the power-weakening increment strategy, the heavy flow with relative advantages can be amplified, which is consistent with the Matthew effect.

In addition, the floating-point number inspired us to allow different parts of one counter to take on different functions. We found that when the count is small, the high bit of the counter is in a vacant state. Thus, we can divide the counter into two parts: a high-bit part to count items that do not belong to the candidate flow, and a low-bit part of counting items belonging to the candidate flow.

Based on these two ideas, we present Matthew Counter with two modes. In competitive mode, the nvote-part in Matthew Counter counts the number of items with different fingerprints from the candidate flow, whereas the pvote-part counts the number of items with the same fingerprint as the candidate flow. In exclusive mode, the nvote-part is swallowed by the pvote-part. The nvote-part adopts a power-weakening increment method, which increases the nvote-part with a probability.

To summarize, this study makes the following contributions.

1) We describe the design of Matthew Counter with two modes that can accurately track a heavy flow. Then, based on Matthew Counter, we present MCSketch, which can achieve a high accuracy for a heavy flow detection, and provides an estimated frequency extremely close to the true frequency of a heavy flow.

2) We present a mathematical analysis of the error bounds on the heavy flow frequency of MCSketch and theoretically prove its high precision

3) We implemented MCSketch and other algorithms. Trace-driven evaluations show that MCSketch achieves a high accuracy in heavy flow detection compared to state-of-art methods. The average relative error of a heavy flow frequency estimation is only 0.1 of the current algorithms while maintaining an almost optimal F1 score. We released the source code of MCSketch and the related algorithms on Github [19].

2 Background

The data stream in an observation window is viewed as a set of M flows $F = \{f_1, f_2, ..., f_M\}$, where each flow f_i is composed of items sharing the same flow fingerprint, for example, the source IP in network data stream. The flow frequency refers to the number of items belonging to this flow. Let n_i and $\hat{n}_i$ represent the real and estimated frequencies of flow f_i, respectively. Given a threshold Φ, heavy flow f_i is defined as $n_i \geq \Phi$.

There are two traditional methods used to find a heavy flow: the *record-all* method and the *record-some* method.

The *record-all* method uses a sketch to track all flows (including the Count-Min [5] sketch, CU Sketch [6], or Count Sketch [4]). Because of the hash collisions, the flow frequency estimation is inaccurate, although there are numerous strategies to improve the accuracy, such as a flow separation [26, 28] and noise correction [11, 13, 23]. As a more severe problem, they are non-invertible because we must check every flow in the entire flow key space to recover all heavy flows. Therefore, this method is slow and inaccurate.

Many algorithms use the *record-some* method, including lossy counting [17], space-saving [18], LDSketch [12], HeavyKeeper [7], WavingSketch [14], and ActiveKeeper [25]. The *record-some* method uses a key-value counter to record the partial flow key and flow value. The space-saving algorithm captures each incoming flow and increases its value by 1 if the flow has already been recorded. The smallest flow in a summary with size will be replaced by a new flow with size, which leads to a significant lack of precision. HeavyKeeper uses the exponential decay probability to decay or be replaced. The probability of a decay decreases as the count increases, and thus any flow whose size exceeds a certain threshold will be difficult to replace. WavingSketch uses a hash function to hash incoming items to $+1$ or -1, and then increases or decreases the waving counter by 1. A waving counter is then used to obtain the estimated frequency. If its frequency is larger than the smallest frequency in the list, it is exchanged.

However, these methods only consider the accuracy of heavy flow detection, but cannot precisely determine the frequency of a heavy flow.

3 Design of MCSketch

The key design of MCSketch is to store the frequency of a candidate heavy flow and other flow frequencies separately, and the frequencies of other flows are increased by a certain probability. In this section, we first introduce the main idea of MCSketch, and then describe the operation of our Matthew Counter. Finally, we present the basic data structure and operation of MCSketch. The symbols frequently used in this study are listed in **Table 1**.

Table 1. Frequently Used Notations

Notations	Meaning
S	A data stream
M	The number of flows
N	The number of items
f_i	*i-th* distinct flow
F_i	The fingerprint of flow f_i
n_i	Real frequency of flow
$\hat{n}_i$	Estimated frequency of flow
d	Array number
w	The bucket number in a array
α	Pre-defined parameter
β	Pre-defined parameter

3.1 Main Idea

We aim to use a hash table to find all elephant flows quickly and accurately. The hash table maps items to be processed to a bucket for access, and the average cost for each look is extremely small and independent of the number of items in a well-dimensioned hash table.

Because of the limited buckets in the hash table, each bucket is mapped by many flows. To find the candidate heavy flow in each bucket, the simplest method is using the majority vote algorithm (MJRTY) [3]. Each time a new flow arrives, MJRTY increments the indicator counter by 1 if both fingerprints are the same; otherwise, it decrements the indicator counter by 1. When the indicator counter is 0, MJRTY replaces the current candidate flow with the new flow and resets the counter to 1. MJRTY can consistently and accurately find a heavy flow with more than half of the total num in the bucket. However, there are two main problems. First, MJRTY cannot return the estimated frequency of the candidate flow because the indicator counter only qualitatively represents a certain heavy flow. Second, the size of the heavy flow in the bucket is not always more than half.

1) To address the first problem, the intuitive solution is to split the indicator field in MJRTY into two parts with independent functions. The pvote-part counts the number of items with the same fingerprint as the candidate flow, and the nvote-part computes the number of items with different fingerprints as the candidate flow in a bucket. However, this solution dramatically increases the memory of each bucket. When the total memory of the hash table remains unchanged, the number of buckets decreases and the number of hash conflicts increases.

Fortunately, we found that when the value of the counter is small, a high bit of the counter is not used. Therefore, our idea is to merge the nvote-part and the pvote-part into a single counter. When the value of the candidate flow frequency is small, the nvote-part uses a high bit and the pvote-part uses a low bit. When the value of the pvote-part reaches a predefined threshold, the pvote-part swallows the nvote-part, and thus the count range of the counter is close to the original counter and there is no increase in memory.

2) To address the second problem, we adopt a probabilistic method called a power-weakening increment. Assuming that an nvote-part and a pvote-part are in the bucket, the power-weakening increment method allows sufficient competition during the early stages of identifying the elephant flows and avoids an increase in the nvote-part when the counter is sufficiently large to ensure the level of accuracy. Specifically, when the incoming item belongs to the flow stored in the hashed bucket, the pvote-part is incremented by 1; otherwise, we increase the nvote-part with a certain probability, which becomes smaller in the power order as the value of pvote-part grows. If the nvote-part is incremented to the value of pvote-part, it sets the to 0 to accept the new flow. In this way, the small flow can easily be evicted from the bucket, and the heavy flows can remain stable.

Although the flow frequency will not be overestimated, it may be underestimated, despite the power-weakening increment method reducing the impact of hash collisions. Therefore, there is a small probability of underreporting a real heavy flow. To solve this problem, we use a sketch structure composed of multiple hash tables with different hash functions. A heavy flow can be stored in a bucket in each row. We choose the largest size as the estimated frequency, minimizing the error of the flow sizes and decreasing the probability of an underreporting.

3.2 Matthew Counter

Matthew Counter works in two different modes: competitive and exclusive. Matthew Counter is composed of the following three parts: 1) a flag part indicating the current mode of the counter, where a flag of 0 means competitive mode, and a flag of 1 means exclusive mode. 2) The nvote-part counts the number of items with different fingerprints from the candidate flow in the bucket when in competitive mode. In exclusive mode, the nvote-part is swallowed by the pvote-part. 3) The pvote-part counts the number of items with the same fingerprint as the candidate flow.

A two-mode Matthew counter is shown in **Fig. 1**. Let the number of bits in the counter be r and apply 1 bit for the flag-part. For competitive mode, Matthew Counter has s bits for the nvote-part and $r-s-1$ bits for the pvote-part. For exclusive mode, the pvote-part is $r-1$ bits.

Fig. 1 (a) shows that the incoming item belongs to the candidate flow in competitive mode, and the pvote-part is incremented by 1. If the value of the pvote-part is larger than $2^{r-s-1}-2$, the pvote-part and nvote-part are merged into the pvote-part. At this time, the count range of the pvote-part is $[0, 2^{r-1}-1]$.

The incoming item does not belong to the candidate flow in competitive mode, as shown in **Fig. 1** (b). The incremental probability in the power-weakening method for the nvote-part is defined as follows:

$$P_{nvote-inc} = \begin{cases} 1 & \text{if } C < \beta \\ \left(\dfrac{\beta}{C}\right)^{\alpha} & \text{if } C \geq \beta \end{cases} \quad (1.1)$$

where C is the value of the current pvote-part, and α (a positive number) and β (an integer greater than 1) are predefined parameter. Therefore, the larger the flow, the harder it is for the nvote-part size to be incremented, and this flow is more likely to be a candidate flow. In the following section, we describe an experiment conducted to ensure these two parameters.

Assuming $\beta = 1$ and $\alpha = 1$, we incremented the nvote-part by 1 with a probability of 13^{-1}. If the value of the nvote-part is larger than the value of pvote-part, we set these two parts to 0, which means that the current candidate flow has been evicted and the next incoming flow is accepted as the candidate flow. Otherwise, if the value of the nvote-part is larger than $2^{r} - 2$, the pvote-part and nvote-part are merged into the pvote-part.

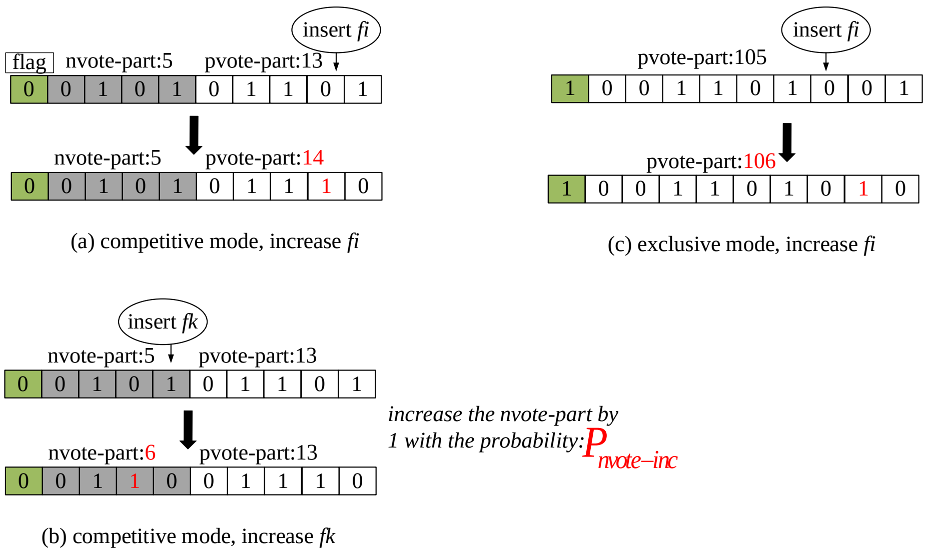

Fig. 1. Matthew Counter

In exclusive mode, only the incoming item belonging to the candidate flow is counted. **Fig. 1** (c) shows that the incoming item belongs to the candidate flow, and the pvote-part is incremented by 1. At this time, the value of pvote-part is sufficiently large, and thus we will no longer replace the candidate flow.

There are two cases for mode shifting from competitive mode to exclusive mode. The first case is when the pvote-part reaches the threshold, and then sets the bits in the nvote-part to 0. Next, the pvote-part and nvote-part are merged into the pvote-part, and increment pvote by 1. In the second case, when the nvote-part reaches the threshold, the bits in the nvote-part are set to 0, and the pvote-part and nvote-part are merged into the pvote-part.

For convenience, we abstract the insertion of data items with different fingerprints in the two modes into two operations. An increase in operation means that the fingerprint of the incoming item is the same as FP in the corresponding bucket, and a decrease in operation means that the fingerprint is different. The pseudocodes of the increase and decrease operations are shown in Algorithms 1 and2, respectively.

Algorithm 1: Increase(MC)

```
input  : Mathew Counter MC
1  if $MC.flag = flase$ then
       // compete mode
2      $MC[pvote - part] + +$;
3      if $MC[pvote - part] > TH_{count}$ then
           // MC switch to Exclusive Mode
4          Set all bit in nvote-part to 0;
5          Incorporate nvote-part into pvote-part;
6  else
7      $MC[pvote - part] + +$;
```

Algorithm 2: Decrease(MC)

```
input  : Mathew Counter MC
// Only in Complete Mode, MC perform decrease
1  if $MC.flag = flase$ then
2      $P_{decay} = \left(\frac{beta}{MC[pvote-part]}\right)^{alpha}$;
3      if $rand(1) < P_{decay}$ then
4          $MC[nvote - part] + +$;
5          if $MC[nvote - part] > MC[pvote - part]$ then
               // clear current MC
6              $MC[nvote - part] \leftarrow 0$;
7              $MC[pvote - part] \leftarrow 0$;
8          else if $MC[nvote - part] > TH_{nvote}$ then
               // switch to Exclusive Mode
9              Set all bit in nvote-part to 0;
10             Incorporate nvote-part into pvote-part;
```

3.3 Data Structure of MCSketch

As shown in **Fig. 2** (a), MCSketch comprises d arrays, and each array has w buckets. Each bucket has a fingerprint field and Matthew Counter. The fingerprint field stores the key of the candidate flow, and Matthew Counter is devoted to choosing and counting this flow. We use $A[u][v]$ to denote the v-th bucket in the u-th array. We then use $A[u][v].MC$ and $A[u][v].FP$ to represent its Matthew Counter and fingerprint, respectively. Each incoming item is inserted into d arrays with d pair-wise independent hash functions.

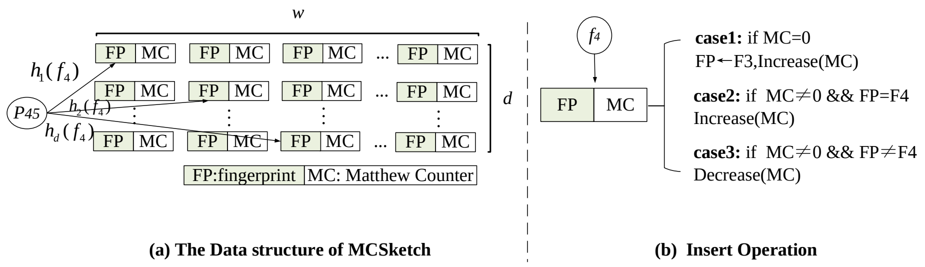
w
FP MC
$h_1(f_4)$
$h_2(f_4)$
$h_d(f_4)$
P45
d
FP:fingerprint
MC: Matthew Counter
(a) The Data structure of MCSketch
f_4
case1: if MC=0 FP←F3,Increase(MC)
case2: if MC≠0 && FP=F4 Increase(MC)
case3: if MC≠0 && FP≠F4 Decrease(MC)
(b) Insert Operation

Fig. 2. MCSketch

Insertion: Initially, all fingerprints were null, and all MC values were 0. The pseudocode for the insertion operation is shown in Algorithm 3.

Assume that the incoming item P_{45} belongs to flow f_4, and the fingerprint of f_4 is F_4. There are three solutions as shown in **Fig. 2** (b). **Case 1:** When $A[u][v].MC = 0$, no flow has been mapped to this bucket, or the value of the nvote-part is larger than pvote-part, which means the candidate streams have been evicted. The bucket tracks the arriving flow as a candidate flow. Then, $A[u][v].FP = F_i$ and $A[u][v].MC$ execute increase operations. **Case 2:** When $A[u][v].MC \neq 0$ and $A[u][v].FP = F_4$, the incoming item belongs to a candidate flow. Then, $A[u][v].MC$ executes an increase operation. **Case 3:** When $A[u][v].MC \neq 0$ and $A[u][v].FP \neq F_4$, the incoming item does not belong to the candidate flow. Then, $A[u][v].MC$ executes a decrease operation.

Algorithm 3: Insertion

```
input  : A packet $P_l$ belonging to flow $f_i$
1 for $u \leftarrow 1$ to $k$ do
2     $v \leftarrow h_u(f_i) \bmod w$;
3     if $A[u][v].MC = 0$ then
4         $A[u][v].FP \leftarrow F_i$;
5         $Increase\,(A[u][v].MC)$;
6     else if $A[u][v].FP = F_i$ then
7         $Increase\,(A[u][v].MC)$;
8     else
9         $Decrease\,(A[u][v].MC)$;
```

Query: The pseudocode for the query operation is shown in Algorithm 4. For convenience, $A[u][v].MC$ represents the pvote-part value of the Matthew Counter. Given a threshold, MCSketch traverses all buckets to find a candidate flow that estimates the frequency over the threshold. Suppose one candidate flow is stored in two or more buckets. In this case, we take the maximum value in the corresponding bucket as the heavy flow frequency because MCSketch has no overestimation error. Because

MCSketch achieves a small error in a heavy flow frequency estimation, it can significantly improve the accuracy of a heavy flow detection.

Algorithm 4: Query

```
input  : A pre-defined Threshold
output: heavy flow set Res
1 for $u \leftarrow 1$ to $k$ do
2     for $v \leftarrow 1$ to $k$ do
3         if $A[u][v].MC > Threshold$ then
4             Traverse Res;
5             if $\exists i, Res[i].FP = A[u][v].FP$ then
6                 $Res[i].Count \leftarrow \max(Res[i].Count, A[u][v].MC)$
7             else
8                 Add A[u][v].FP and A[u][v].MC into Res
9 Return Res;
```

4 Mathematical Analysis

4.1 Proof of No Over-estimation of flow frequency

Theorem 1. Let n_i^t and $\hat{n}_i^t$ be the real frequency and estimated frequency of flow f_i at time t, and $A[u][v]$ be the mapped bucket of flow f_i in the u-th array. For convenience, we use MC to represent $A[u][v].MC$, and $MC[pvote-part]$ is the estimated value of the candidate heavy flow. Here, $\hat{n}_i^t = MC^t[pvote-part]$ when f_i is the candidate flow at time t, or $\hat{n}_i^t = 0$. Thus,

$$\forall_t, \hat{n}_i^t \le n_i^t \tag{2}$$

Proof. We used mathematical induction to prove this. When $t=0$, no item is mapped into the bucket before. Therefore, $MC[pvote-part]=0$, $n_i^0=0$, $\hat{n}_i^0=0$, and the theorem holds.

If the theorem holds when $t=v$, we must have $MC^t[pvote-part] \le n_i^t$. Now, we prove that the theorem also holds when $t=v+1$. For the incoming item, there are four cases:

Case 1: Flow f_i is not the candidate flow in that bucket, $F_i \ne A[u][v].FP$. Then, $\hat{n}_i^{v+1}=0$. Clearly, $\hat{n}_i^{v+1} \le n_i^{v+1}$.

Case 2: $F_i = A[u][v].FP$, and the incoming item is not mapped to $A[u][v]$. Then, $n_i^{v+1}=n_i^v$, and $\hat{n}_i^{v+1}=MC^v[pvote-part]$. Therefore, $\hat{n}_i^{v+1} \le n_i^{v+1}$.

Case 3: $F_i = A[u][v].FP$ and the incoming item mapped to $A[u][v]$ but does not belong to flow f_i. Then, $n_i^{v+1}=n_i^v$. If the value of nvote-part is larger than that of the pvote-part, the pvote-part is set to 0; otherwise, $MC^{v+1}[pvote-part]=MC^{v}[pvote-part]$, and we then have $MC^{v+1}[pvote-part]\le MC^{v}[pvote-part]$. Therefore, we can ensure that $\hat{n}_i^{v+1}=MC^{v+1}[pvote-part]\le n_i^{v+1}$ holds.

Case 4: $F_i = A[u][v].FP$ and the incoming item mapped to $A[u][v]$ belonging to flow f_i. Then, $n_i^{v+1}=n_i^{v}+1$, and $MC^{v+1}[pvote-part]=MC^{v}[pvote-part]+1$. Therefore, we have $\hat{n}_i^{v+1}=MC^{v}[pvote-part]+1\le n_i^{v+1}$.

In summary, the theorem holds at any point in time.

4.2 Error Bound of MCSketch

Theorem 2. Assume that the fingerprint of a heavy flow is held at its mapped bucket at all times. Without loss of generality, we focus on a single array of MCSketch. Given a small positive value ε, and a heavy flow f_i, which is stored in the bucket at all times,

$$\Pr(n_i-\hat{n}_i\ge \varepsilon N)<\frac{1}{\varepsilon w}*\left(\frac{\beta}{\beta+1}\right)^{\alpha} \tag{3}$$

Proof. First we introduce indicator variable $I_{i,j,k}$, defined as

$$I_{i,j,k}=\begin{cases}0 & \text{if}\,(f_i=f_k)\vee(h_j(f_i)\ne h_j(f_k))\\ 1 & \text{if}\,(f_i\ne f_k)\wedge(h_j(f_i)=h_j(f_k))\end{cases} \tag{4}$$

$I_{i,j,k}=1$ means two different flows f_i and f_k are mapped into the same bucket in the j^{th} array. Assuming the hash functions map each flow to one of the w counters uniformly at random, the collision probabilities of two distinct flows are $\frac{1}{w}$, so the expectation of $I_{i,j,k}$ as follows:

$$E(I_{i,j,k})=\Pr[h_j(f_i)=h_j(f_k)]=\frac{1}{\text{range}(h_j)}=\frac{1}{w} \tag{5}$$

Then we define random variable $X_{i,j}=\sum_{k=1}^{M}I_{i,j,k}n_k$, where $X_{i,j}$ means that the sum of the sizes of other distinct flows which are mapped to the same bucket in row j, except for the flow f_i itself. Then the expectation of $X_{i,j}$ is:

$$\mathrm{E}(X_{i,j})=\mathrm{E}\left(\sum_{k=1}^{M}I_{i,j,k}n_k\right)=\sum_{k=1}^{M}\left(E(I_{i,j,k})*n_k\right)=\frac{N-n_i}{w} \tag{6}$$

For each incoming item, if it belongs to flow f_i, the pvote-part is increased by 1; if not, the nvote-part increased by 1 with a certain probability. One extreme situation is that all items that do not belong to candidate flow incremented the nvote-part, then we have $\hat{n}_i = n_i - X_{i,j}$. Another extreme case is that all items that do not belong to candidate flow don't increase the nvote-part, then we have $\hat{n}_i = n_i$. Thus, we have

$$n_i - X_{i,j} \le \hat{n}_i \le n_i \tag{7}$$

Next, we define a random variable $P_{i,j,l}$, representing the probability that the l-th item in $X_{i,j}$ increment the nvote-part. From the definition of power-weaking increment,

$$P_{i,j,l} = \begin{cases} 1 & \text{if } C_l < \beta \\ \left(\dfrac{\beta}{C_l}\right)^{\alpha} & \text{if } C_l \ge \beta \end{cases} \tag{8}$$

Where C_l presents the value of pvote-part when the l-th item in $X_{i,j}$ arrives. We have $\hat{n}_i = n_i - \sum_{l=1}^{X_{i,j}} P_{i,j,l}$. By the Markov inequality, the following formula holds:

$$\Pr(n_i - \hat{n}_i \ge \varepsilon N) = \Pr(\sum_{l=1}^{X_{i,j}} P_{i.j,l} \ge \varepsilon N) \le \frac{E(\sum_{l=1}^{X_{i,j}} P_{i,j,l})}{\varepsilon N} \tag{9}$$

Then we focus on $E(\sum_{l=1}^{X_{i,j}} P_{i,j,l})$. Assume that all items are uniformly distributed. From equation (8), we can get $P_{i,j,l} = 1$ when $1 \le C_l \le \beta$, thus we have $\Pr\{P_{i,j,l} = 1\} = \dfrac{\beta}{\gamma}$. For convenience, let γ presents $n_i - \sum_{l=1}^{X_{i,j}} P_{i,j,l}$. When $\beta < C_l \le n_i - E\left(\sum_{l=1}^{X_{i,j}} P_{i,j,l}\right)$, we have:

$$\Pr\left\{P_{i,j,l} = \left(\frac{\beta}{C_l}\right)^{\alpha}\right\} = \frac{1}{n_i - E(\sum_{l=1}^{X_{i,j}} P_{i,j,l})} = \frac{1}{\gamma} \tag{10}$$

As a result, the expectation of variable $P_{i,j,l}$ can be represented as:

$$E(\sum_{l=1}^{X_{i,j}} P_{i,j,l}) = \sum_{l=1}^{E(X_{i,j})} E(P_{i,j,l}) = E(X_{i,j})\left(\frac{\beta}{\gamma}+\frac{1}{\gamma}\sum_{C_l=\beta+1}^{\gamma}\left(\frac{\beta}{C_l}\right)^{\alpha}\right)$$
$$\because \frac{\beta}{C_l} \le \frac{\beta}{\beta+1}, when\ C_l \in [\beta+1,\gamma] \tag{11}$$
$$\le E(X_{i,j})\left(\frac{\beta}{\gamma}+\frac{(\gamma-\beta)}{\gamma}\left(\frac{\beta}{\beta+1}\right)^{\alpha}\right)$$

Focus on the latter part, we have

$$\frac{\beta}{\gamma}+\frac{(\gamma-\beta)}{\gamma}\left(\frac{\beta}{\beta+1}\right)^{\alpha} = \frac{1}{\gamma}\left\{\beta * \left(1\text{-}\left(\frac{\beta}{\beta+1}\right)^{\alpha}+\left(\frac{\beta}{\beta+1}\right)^{\alpha}\right) + (\gamma\text{-}\beta) * \left(\frac{\beta}{\beta+1}\right)^{\alpha}\right\}$$
$$= \frac{1}{\gamma}\left\{\gamma * \left(\frac{\beta}{\beta+1}\right)^{\alpha} + \beta * \left(1-\left(\frac{\beta}{\beta+1}\right)^{\alpha}\right)\right\}$$
$$\because \frac{\beta}{\beta+1} < 1 \tag{12}$$
$$< \frac{1}{\gamma}\left\{\gamma * \left(\frac{\beta}{\beta+1}\right)^{\alpha}\right\} = \left(\frac{\beta}{\beta+1}\right)^{\alpha}$$

Based on the equation (6), we get:

$$E(\sum_{l=1}^{X_{i,j}} P_{i,j,l}) < \frac{N-n_i}{w} * \left(\frac{\beta}{\beta+1}\right)^{\alpha} \tag{13}$$

Then combine equation (9), so we have

$$\Pr(n_i - \hat{n}_i \ge \varepsilon N) = \Pr(\sum_{l=1}^{X_{i,j}} P_{i.j.l} \ge \varepsilon N) \le \frac{E(\sum_{l=1}^{X_{i,j}} P_{i,j,l})}{\varepsilon N} < \frac{1}{\varepsilon w} * \left(\frac{\beta}{\beta+1}\right)^{\alpha} \tag{14}$$

5 Experiment Results

5.1 Experiment Setup

Platform: Our experiment is run on a server with four 16-core Intel Xeon Gold 5218 @2.3 GHz CPUs and 128 GB of DDR4 memory. The CPU has 64 KB of L1 cache per core, 256 KB of L2 cache per core, and 10 MB of shared L3 cache.

Dataset: The IP trace dataset contains anonymized IP traces collected in 2016 by CAIDA [22]. In our experiment, we used the source IP as the fingerprint of the flow. We use the first 4M items belonging to approximately 50K flows.

Implementation: We compare MCSketch (MC) with state-of-the-art heavy hitter algorithms, including LDSketch (LD), MVSkech (MV), HeavyKeeper (HK), and

WavingSketch (WS). All algorithms were implemented in C++. For all algorithms, the fingerprint field was 32 bits. For MVSketch, the sum field and counter field are both also 32 bits. For LDSketch, because it dynamically expands the associative array of its buckets, we first determine the most available array based on the given memory, and to ensure balance, we limit the most associative arrays of each bucket. According to the original study, the decay probability in HeavyKeeper is 1.08, and the slot number in the heavy part is 8. The hash functions used in the sketches are implemented from a 32-bit Bob hash (obtained from [10]).

Metrics: We used the throughput to measure the insert speed. The precision, recall, and F1 score were used to measure the accuracy of the heavy flow detection. The AAE and ARE are used to measure the error of the flow frequency estimation.

The throughput is defined as N/T, where T is the total processing time. We used millions of inserted items per second (Mps) to measure the throughput. The experiments were repeated 10 times, and the average throughput was reported.

Precision is the fraction of true heavy flows reported over all reported flows. Recall is the fraction of true heavy flows reported over all true heavy flows. F1-score is the harmonic mean of the precision and recall, defined as $\frac{2*Precision*Recall}{Precision+Recall}$.

The average absolute error (AAE) is defined as $\frac{1}{\Psi}\sum_{f_i \in \Psi}|n_i - \hat{n}_i|$, where Ψ is the true heavy flow reported. The average relative error (ARE) is defined as $\frac{1}{\Psi}\sum_{f_i \in \Psi}\frac{|n_i - \hat{n}_i|}{n_i}$.

5.2 Experiments on Parameters

In this section, we describe experiments conducted to evaluate the effects of the MC parameters. We focus on the array number d, and power-weakening parameters α and β. We set the total memory size to 8 KB, and the threshold (which defines a heavy flow) to 1000. We used the F1 score to evaluate the accuracy of the heavy hitter and used the ARE to assess the error of the flow frequency estimation. Because the three parameters may have mutual influence, we first observe the influence of d while predetermining the values of α and β. We then observe the influence of α and β.

Effects of d **.** In this experiment, we set $\alpha = 1$ and $\beta = 1$. **Fig. 3** and **Fig. 4** show that when d increases from 1 to 4, the F1 score increases first and then decreases, and ARE continues to increase. When $d = 2$, the F1 score was the largest (approximately 0.778). This is because when d increases, the number of buckets in each array will be reduced, which will lead to more hash conflicts. For example, in this experiment, the number of heavy flows over 1000 is 631, and the total number of buckets with 8 KB memory is 1024. Therefore, when $d = 3$, the bucket number w in each array is 341, which contains many hash collisions.

Effects of α and β. In this experiment, we set $d = 2$. **Fig. 5** and **Fig. 6** show the F1 score and ARE for different values of α and β, respectively. Given β, the value of the F1 score will increase significantly as α increases, and then maintain a high value for most values of β; however, it will decrease when $\beta = 1$. By contrast, ARE maintains a roughly downward trend as α increases. When $\alpha = 0.6$ and $\beta = 1$, the F1 score is the largest at approximately 0.799.
In the following experiment, we set $\alpha = 0.6$ and $\beta = 1$ for MCSketch. We also set $d = 2$ for all algorithms, if necessary.

5.3 Experiment on Memory

In this section, we conducted experiments with various memory sizes to study the impact of memory size on the accuracy, error, and throughput. We set the threshold of a heavy flow to 100 and varied the memory size from 5 KB to 30 KB on the CAIDA datasets.

Effects of memory on the accuracy. Fig. 7 - **Fig. 9** compare the accuracy of MC with that of other algorithms in terms of heavy hitter detection. As shown in **Fig. 7**, the precision of MC, HK, and LD is consistently 1. This is because the three algorithms do not overestimate the flow frequency. If estimated frequency of one candidate flow exceeds the predefined threshold, then the candidate must be a true heavy flow.

Fig. 8 compares the recall of MC with that of the other algorithms. The recall of all algorithms increases as the memory increases. MC has the largest recall when the memory is less than 10 KB, and the recall of MC always maintains a high value compared to the other algorithms. **Fig. 9** compares the F1 score of the MC with that of the other algorithms. The F1 score of MC is always the largest, and as the memory increases, it continues to increase. When the memory is 10 KB, the F1 scores of MC, MV, LD, HK, and WS are 0.69, 0.34, 0.26, 0.51, and 0.60, respectively.

In conclusion, MC had the best F1 score compared to previous studies in different memories. The recall of MCSketch is close to the current optimal algorithm, and its precision is continuously 1.

Effects of memory on error. Fig. 10 and **Fig. 11** compare the error in the heavy flow frequency estimation of MCSketch with those of the other algorithms. For convenience, we consider the logarithms of ARE and AAE. Regardless of the AAE or ARE, the estimated error of MCSketch is much smaller than that of the other algorithms.

Table 2 shows the ARE and AAE of all algorithms when the memory is 20 KB, and the ARE of MC is 41.5-times lower than that of MV, 1.5-times lower than that of LD, 6-times lower than that of WS, and 2.6-times lower than that of HK. The AAE is 38.1-times lower than that of the MV, 1.8-times lower than that of LD, 4.4-times lower than that of the WS, and 2.6-times lower than that of HK.

The results show that the heavy flow frequency estimation error of MC is much smaller than that of the other algorithms.

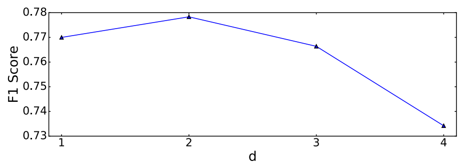

Fig. 3. Effect of *d* on F1 Score

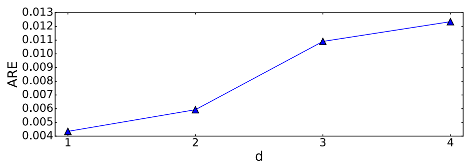

Fig. 4. Effect of *d* on ARE

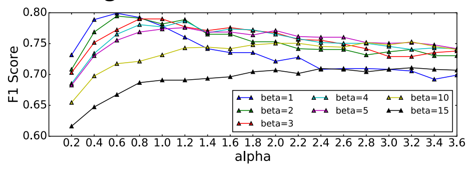

Fig. 5. Effects of α and β on F1 score

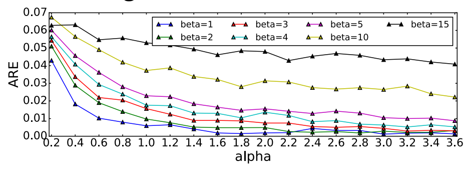

Fig. 6. Effects of α and β on ARE

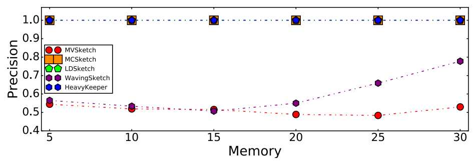

Fig. 7. Precision vs. Memory

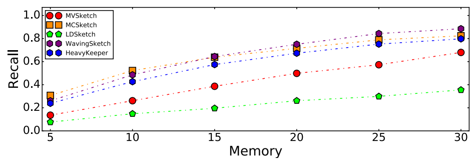

Fig. 8. Recall vs. Memory

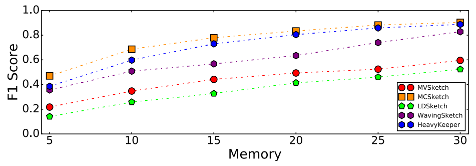

Fig. 9. F1 Score vs. Memory

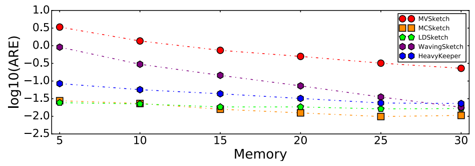

Fig. 10. ARE vs. Memory

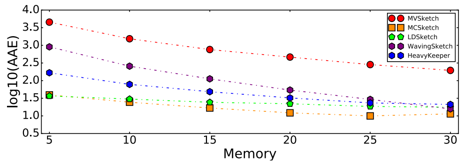

Fig. 11. AAE vs. Memory

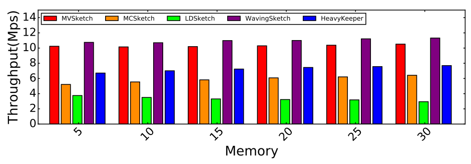

Fig. 12. Throughput vs. Memory

Table 2. Comparison of Heavy Flow Frequency Estimation

Error Metric	Algorithms				
	MC	***MV***	***LD***	***WS***	***HK***
ARE	0.012	0.499	0.018	0.073	0.032
AAE	12.2	465.4	22.1	54.1	32.3

Effects of memory on throughput. Fig. 12 compares the throughput of MC with that of the other algorithms. When the memory size is 30 KB, the throughput of MCSketch is 6.5 Mps while those of MV, LD, WS, and HK are 10.5, 2.9, 11.3, and 7.7 Mps, respectively.

5.4 Experiments on Threshold

In this section, we describe experiments conducted with various thresholds to study the impact of the threshold on the accuracy, error, and throughput. When the threshold of the heavy flow is larger, the number of heavy flows is reduced. We set the memory to 10 KB and varied the threshold from 500 to 1000.

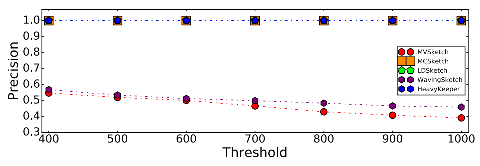

Fig. 13. Precision vs. Threshold

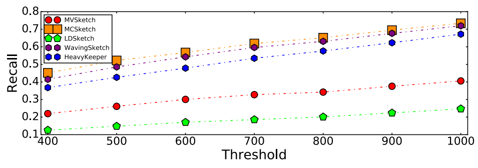

Fig. 14. Recall vs. Threshold

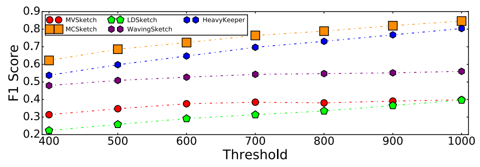

Fig. 15. F1 Score vs. Threshold

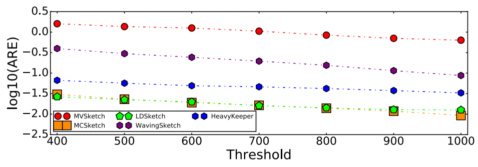

Fig. 16. ARE vs. Threshold

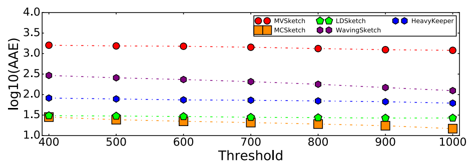

Fig. 17. AAE vs. Threshold

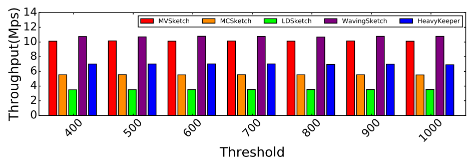

Fig. 18. Throughput vs. Threshold

Effects of threshold on accuracy. In this section, we describe experiments conducted to measure the impact of the threshold on the accuracy of heavy-hitter detection. **Fig. 13** shows that the MC consistently achieves 100% precision for every threshold. **Fig. 14** indicates that the recall of MC increases with an increasing threshold. When the threshold becomes large, the true heavy flow decreases. The high-ranking heavy flow has a larger real frequency, and the impact of the flow frequency estimation error is negligible. **Fig. 15** shows that when the threshold is varied from 400 to 1000, the F1 score of MC is always the highest. When the threshold is 1000, the F1 scores of these algorithms are 0.73, 0.40, 0.25, 0.70, and 0.67, respectively. The F1 score of MC was 1.8-times higher than that of MV, 2.9-times higher than that of LD, 1.04-times higher than that of WS, and 1.09-times higher than that of HK.

In conclusion, MC has the highest F1 score compared to previous studies at different thresholds. Even if the threshold is small and the number of heavy flows that need to be detected is large, MC still has an acceptable accuracy.

Effects of threshold on error. In this section, we describe experiments conducted to measure the impact of the threshold on the heavy flow frequency estimation. **Fig. 16** and **Fig. 17** compare the ARE and AAE of MCSketch with other algorithms. When the threshold is 1000, the ARE of MC, MV, LD, MS, and HK is 0.009, 0.636, 0.013, 0.088, and 0.033, respectively. The AAE of MC, MV, LD, MS, and HK is 14.8, 1201.7, 26.9, 125.0, and 61.9, respectively.

In conclusion, MCSketch has an extremely small error in the heavy flow estimation at different thresholds. Compared to previous studies, the error of MCSketch is only 3% to 50% that of the other algorithms.

Effects of threshold on throughput. Fig. 18 compares the throughput of MCSketch with that of the other algorithms. When the threshold of a heavy flow varies from 400 to 1000, the throughput of MC is basically unchanged at approximately 5.56 Mps. MC is not particularly fast because the update method of Matthew Counter is more complicated.

6 Conclusion

Accurately finding heavy flows in data streams is challenging owing to the limited memory availability. Prior algorithms focus on accuracy in heavy flow detection but cannot provide the frequency of heavy flow exactly. This paper proposes a novel algorithm called MCSketch for heavy flow detection and heavy flow frequency estimation. The key ideas are using idle high-bits and a power-weakening method to allow sufficient competition in the early stages of identifying heavy flows and amplifying the relative advantage when the counter is sufficiently large. The experiment results show that MCSketch achieves a higher accuracy in heavy flow detection and heavy flow frequency estimation. As a shortcoming, the update speed in MCSketch is not sufficiently fast. In general, we believe that MCSketch can be applied to many more applications, such as DDoS detection and traffic engineering. All related source codes have been released on Github [19].

References

1. Ball, B. et al. 2014. A flexible and extensible contract aggregation framework (caf) for financial data stream analytics. Proceedings of the International Workshop on Data Science for Macro-Modeling (2014), 1–6.
2. Basat, R. Ben et al. 2021. SALSA: Self-Adjusting Lean Streaming Analytics. ICDE (2021).
3. Boyer, R.S. and Moore, J.S. 1991. MJRTY—a fast majority vote algorithm. Automated Reasoning. Springer. 105–117.
4. Cormode, G. and Hadjieleftheriou, M. 2008. Finding frequent items in data streams. Proceedings of the VLDB Endowment. 1, 2 (2008), 1530–1541. DOI:https://doi.org/10.14778/1454159.1454225.
5. Cormode, G. and Muthukrishnan, S. 2005. An improved data stream summary: The count-min sketch and its applications. Journal of Algorithms. 55, 1 (2005), 58–75. DOI:https://doi.org/10.1016/j.jalgor.2003.12.001.
6. Estan, C. and Varghese, G. 2002. New directions in traffic measurement and accounting. Computer Communication Review. 32, 1 (2002), 75. DOI:https://doi.org/10.1145/510726.510749.

7. Gong, J. et al. 2018. HeavyKeeper: An accurate algorithm for finding top-k elephant flows. Proceedings of the 2018 USENIX Annual Technical Conference, USENIX ATC 2018. (2018), 909–921. DOI:https://doi.org/10.1109/tnet.2019.2933868.
8. Goyal, A. et al. 2012. Sketch algorithms for estimating point queries in NLP. Proceedings of the 2012 joint conference on empirical methods in natural language processing and computational natural language learning (2012), 1093–1103.
9. Gyurkó, L.G. et al. 2013. Extracting information from the signature of a financial data stream. arXiv preprint arXiv:1307.7244. (2013).
10. Hash Website: http://burtleburtle.net/bob/hash/evahash.html.
11. Huang, Q. et al. 2018. SketChlearn: Relieving user burdens in approximate measurement with automated statistical inference. SIGCOMM 2018 - Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication (Aug. 2018), 576–590.
12. Huang, Q. and Lee, P.P.C. 2015. A hybrid local and distributed sketching design for accurate and scalable heavy key detection in network data streams. Computer Networks. 91, (2015), 298–315. DOI:https://doi.org/10.1016/j.comnet.2015.08.025.
13. Jie, L. et al. 2021. OrderSketch: An Unbiased and Fast Sketch for Frequency Estimation of Data Streams. Computer Networks. (2021), 108563.
14. Li, J. et al. 2020. WavingSketch: An Unbiased and Generic Sketch for Finding Top-k Items in Data Streams. Proceedings of the ACM SIGKDD International Conference on Knowledge Discovery and Data Mining. (2020), 1574–1584. DOI:https://doi.org/10.1145/3394486.3403208.
15. Liu, L. et al. 2020. SF-Sketch: A Two-Stage Sketch for Data Streams. IEEE Transactions on Parallel and Distributed Systems. 31, 10 (2020), 2263–2276. DOI:https://doi.org/10.1109/TPDS.2020.2987609.
16. Liu, Z. et al. 2019. Nitrosketch:Robust and General Sketch-based Monitoring in Software Switches. Proceedings of the ACM Special Interest Group on Data Communication - SIGCOMM '19 (New York, New York, USA, Aug. 2019), 334–350.
17. Manku, G.S. and Motwani, R. 2012. Approximate frequency counts over data streams. Proceedings of the VLDB Endowment. 5, 12 (2012), 1699–1699. DOI:https://doi.org/10.14778/2367502.2367508.
18. Metwally, A. et al. 2005. SpaceSaving: Efficient Computation of Frequent and Top-k Elements in Data Streams. Proceedings of the 10th international conference on Database Theory. (2005), 398–412.
19. Source code related to MCSketch: https://github.com/Paper-commits/MCSketch.
20. Tang, L. et al. 2019. MV-Sketch: A Fast and Compact Invertible Sketch for Heavy Flow Detection in Network Data Streams. Proceedings - IEEE INFOCOM. 2019-April, (2019), 2026–2034. DOI:https://doi.org/10.1109/INFOCOM.2019.8737499.
21. Tang, L. et al. 2020. SpreadSketch : Toward Invertible and Network-Wide Detection of Superspreaders. IEEE International Conference on Computer Communications (2020).
22. The caida anonymized internet traces 2016: http://www.caida.org/data/overview/.
23. Ting, D. 2018. Count-min: Optimal estimation and tight error bounds using empirical error distributions. Proceedings of the ACM SIGKDD International Conference on Knowledge Discovery and Data Mining. (2018), 2319–2328. DOI:https://doi.org/10.1145/3219819.3219975.
24. Wang, J. et al. 2018. Detecting and Mitigating Target Link-Flooding Attacks Using SDN. IEEE Transactions on Dependable and Secure Computing. 16, 6 (2018), 944–956. DOI:https://doi.org/10.1109/TDSC.2018.2822275.

25. Wu, M. et al. 2021. ActiveKeeper : An Accurate and Efficient Algorithm for Finding Top-k Elephant Flows. IEEE Communications Letters. 7798, 1089 (2021), 1–5. DOI:https://doi.org/10.1109/LCOMM.2021.3077902.
26. Yang, T. et al. 2018. Elastic sketch: Adaptive and fast network-wide measurements. SIGCOMM 2018 - Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication (Aug. 2018), 561–575.
27. Zheng, J. et al. 2018. Realtime DDoS Defense Using COTS SDN Switches via Adaptive Correlation Analysis. IEEE Transactions on Information Forensics and Security. 13, 7 (2018), 1838–1853. DOI:https://doi.org/10.1109/TIFS.2018.2805600.
28. Zhou, Y. et al. 2018. Cold filter: A meta-framework for faster and more accurate stream processing. Proceedings of the ACM SIGMOD International Conference on Management of Data (2018), 741–756.