HeavyCache: A Generic Sketch for Summarizing Data Streams

Abstract— **Nowadays, massive data appears in the form of high speed data streams. It is an important and challenging problem to perform various tasks on data streams, such as finding heavy hitters, estimating frequencies, and *etc.* Current applications often need to handle several tasks at the same time. Traditional sketch-based solutions rely on different data structures and algorithms for different tasks. As a result, multiple data structures are needed in practice. In this paper, we propose a generic sketch algorithm, namely HeavyCache, which can quickly record each item and perform a broad spectrum of data mining tasks. The key idea is to leverage cache mechanism to separate heavy items from light items. Specifically, HeavyCache accurately records the detailed information of items, while simply approximately records the frequencies of light items. We show how HeavyCache is used to process the six typical tasks. Experimental results show that our HeavyCache significantly outperforms state-of-the-art solutions in terms of both accuracy and speed.**

I. Introduction

With the development of network, there is a growing need of new techniques for dealing with various data streams, such as IP traffic [36], sensor data [27], [37], click streams [20], and *etc.* Recently, data mining on data streams is widely studied, which offers scalable data analysis techniques for large volumes of data stream. Many industries and research communities have been benefited from data analysis on vast amounts of historical data streams. For example, in data centers, the operators can figure out problems (often known as trouble-shooting [21]) by analyzing the results of characteristics of traffic streams. Advertising companies can advertise more precisely by analyzing the click streams of different users [20]. Thus, research on data mining in data streams helps many systems work better in various areas, which is meaningful.

The useful data analysis tasks often rely on some fundamental mining tasks, such as finding heavy hitters [10], estimating entropy [8] ,and *etc.* However, the large volume and the high speed of data streams bring challenges to these fundamental tasks. The first challenge is how to keep mining results accurate with limited memory usage for such high speed data streams, because many devices have limited memory resources, such as IoT (Internet of Things) devices, network switches and routers, FPGAs, and *etc.* A classic and widely used approach is sampling [4], [38]. Only a small portion of items in the stream is recorded (sampled), which incurs limited processing overhead, but this approach shows poor accuracy on some tasks [15], [33]. Another approach is the *sketch* [7], [11], [16], [28], [34]. A sketch is a probabilistic method for recording data streams with small memory usage, so that it can be fitted into small fast memory (*e.g.* CPU caches, SRAM in FPGA). Several sketch algorithms are proposed for data mining tasks, such as well known Space-Saving [32] for detecting heavy hitters, k-ary [23] for detecting heavy changes, and MRAC [24] for estimating frequency distribution. These algorithms show good performance on their focused tasks in terms of speed and accuracy.

Another challenge is that we often need to perform multiple tasks on the same stream in practice. For example, in IDS (Intrusion Detecting System), DDoS attack always possesses three characteristics: high frequency of incoming packets, asymmetry in interaction patterns, and high diversity of source IP addresses [29]. This means after recording a data stream, the results of frequency estimation [18], heavy hitters, frequency distribution estimation, and *etc.*, can be required. In such scenarios, a straightforward solution is to deploy multiple sketches to record the data stream. And during recording, for each incoming item, multiple sketches need to be updated. While sometimes we even do not know what tasks we need before we start analyzing, we have to deploy all the sketches we may need. Obviously, such a solution is both time-consuming and space-consuming.

For this challenge, we prefer a *generic* sketch, which means using only one sketch for recording data streams, and this data structure can be queried for multiple tasks. Among all existing work, UnivMon [30] is the first and the only generic sketch algorithm. UnivMon is based on the theory of universal streaming [6], and it can perform a spectrum of tasks by one data structure. Unfortunately, to achieve generic, UnivMon sacrifices the processing speed: the time complexity of recording one item using Univmon is $O(logn)$, where n is the number of distinct items, which is not fast enough for high speed streams in the software environment.

In summary, there are three goals in designing a sketch algorithm: generic, fast and accurate. Though it seems that these three goals are conflicting with each other, the goal of this paper is achieving these three goals at the same time.

Our idea is based on two observations of real world data streams and existing data mining tasks. First, according to the analysis of our campus traffic streams, CAIDA Traffic streams [3], and literature [9], [12], [31], [40], the frequency distribution in data streams in the real world is highly skewed. "Skewed" means that the number of `heavy items` (items whose frequency is large) is small, while the number of `light items` (items whose frequency is small) is large. Second but more important, according to our analysis of six widely used tasks (frequency estimation, cardinality estimation, entropy estimation, frequency distribution estimation, heavy hitter detection, and heavy change detection), these tasks are either *Frequency-Focus* or *Heavy-Focus*. Frequency-Focus tasks only need to record the frequencies of all items, while Heavy-Focus tasks need to record the frequencies and keys of only heavy items. According to these observations, the keys of light items are not necessary for these data mining tasks.

Based on these two observations, we propose a novel algorithm, namely HeavyCache, which *leverages cache mechanism*[1] *to separate heavy items from light items*. Specifically, we design a *cache part* to keep track of heavy items as well as a *count part* to record the frequencies of light items.

Note that the cache part accommodates heavy items, which are often more important than light items. On the one hand, the cache part should be efficient and accurate to record the heavy items, and we simply divide the cache part into many small caches, each of which records both keys and frequencies of heavy items. On the other hand, the count part can be extremely simple because it is acceptable to discard keys of light items in it based on our classification of tasks. To this end, we just use a number of counters, and each light item is hashed into a counter to record its frequency. To achieve accurate separation, for each incoming item α, a small cache is chosen and cooperates with the count part to offer α the best accommodation. The cooperation is performed according to our proposed CACHE (Check And Compare, Hold or Evict) algorithm. To minimize the errors, we propose a novel strategy, namely ***consecutive increment***, which has no overhead and no side effect. This strategy does not permit the recorded size of a heavy item to increase by more than one for each update, because the frequency increases at most one when an item arrives.

Our HeavyCache achieves the three design goals: generic, fast, and accurate. We show how to perform six widely used tasks based on HeavyCache, and we believe that our HeavyCache can perform any task which can be categorized as Heavy-Focus or Frequency-Focus. Our experimental results show that our HeavyCache outperforms the state-of-the-art algorithms up to 113 times in terms of accuracy and up to 37 times in terms of speed. The source code of HeavyCache is available at [2].

We make the following key contributions as follows.

- Based on our two key observations about the characteristics of data streams and tasks, we propose a generic sketch algorithm, HeavyCache, which achieve generic, accurate, and fast at the same time.
- We show how to perform six widely used tasks on HeavyCache, and proof that our HeavyCache can give it an untrival upper bound of its frequency, which can be reported when querying.
- We conduct extensive experiments based on network traffic streams, and experimental results show that HeavyCache achieves much higher accuracy and higher speed than the state-of-the-art.

II. Background and Related Work

In this paper, we adopt a simplified version of the widely used cash register model of data streams [9], [11]. A stream S of length m can be denoted as $S = (a_1, a_2, ..., a_m)$, where the i-th item $a_i = e_j$ is one of the items in the item space $U = \{e_1, e_2, ..., e_n\}$. f_e denotes the frequency (the number of occurrences) of e in the stream S. Notice that the item space U is usually large, and only a small portion of the items occur in S. In other words, for most $e \in U$, $f_e = 0$. Each item has an identifier, namely *key*, which distinguishes it from other items. In data streams in the real world, the frequency distributions are often skewed. For example, in a network stream, a large portion of packets may belong to just a small number of flows [2]. Similarly, in click streams, most clicks are on a few hot items. We call the item which has a large frequency a heavy item, and the item which has a small frequency a light item.

We list some fundamental mining tasks in Table I. These tasks are the fundamental components for some more complex analysis. An example is detecting DDoS attack in network stream. One can use the frequencies of incoming flows and the cardinality of incoming flows to distinguish that whether the server is attacked or not [29]. We observe that these six tasks can be categorized into two classes: Heavy-Focus and Frequency-Focus. Heavy-Focus tasks mainly care about the frequencies and keys of heavy items. Heavy hitter detection and heavy change detection [3] belong to this category. Frequency-Focus tasks need to track the frequencies of all items, but can ignore keys. The remaining four tasks belong to this category. Frequency estimation is to estimate the frequency of given item keys. Cardinality estimation, frequency distribution estimation, and entropy estimation can be seen as some forms of combination of frequencies of all items.

Many streaming algorithms are proposed for these tasks, as shown in Table I, and most of them target at one specific task. In practice, we often need to perform more than one task. In such scenarios, we should deploy multiple aforementioned algorithms to record the stream, and each item in the stream need to be inserted into each algorithm, which causes the degradation of both the memory efficiency and processing speed. The ideal solution is to use one data structure to record the stream, and one can perform multiple tasks based on this data structure. We call such solution a "generic" solution. Sampling offers such a generic method, which randomly record the items in a data stream and form a smaller data stream, while this approach suffers from low accuracy in some tasks.

The state-of-the-art generic algorithm is UnivMon [30], which is based on the idea of universal streaming [6]. In universal streaming, one can use one single data structure to estimate a number of tasks, which can be expressed as certain forms of the sum of the item frequencies. For example, the cardinality can be expressed as $\sum x^0$, and the entropy can be expressed as $\sum x * log(x)$, so these two tasks can be handled by universal streaming. Specifically, in the recording phase, UnivMon samples the original stream by item recursively and gets a dozen of sub-streams. For each sub-stream, UnivMon uses a sketch to maintain the set of L2-Heavy Hitters (L2HH) of the sub-stream. In the reporting phase, UnivMon uses these sets of L2HH to calculate the sum function for the specific task. The theory of universal streaming guarantees the bounds of the accuracy of this estimation. The entropy and the cardinality can be estimated by this way. The sets of L2HH maintained by UnivMon can be used to detect

[1]Different with caches in other systems, HeavyCache caches "heavy items" but not hot items.

[2]The packets with the same key belongs to one flow.

[3]If an item is a heavy change, it must be a heavy hitter in one of the streams.

TABLE I
DEFINITION OF TASKS

Tasks	Definition	Category	Algorithms
Frequency estimation	Given an item e, reporting the estimation of its frequency f_e.	Frequency-Focus	CM sketch [11] CU sketch [16] Count [7] CSM [28]
Frequency distribution estimation	Estimate the number of items for each possible frequency.	Frequency-Focus	MRAC [24]
Entropy estimation	Reporting the entropy of frequencies of items in S. The entropy of S can be calculated as $log(m) - \sum f_{e_i} * log(f_{e_i})/m$, where m is the length of S.	Frequency-Focus	AMS [26] UnivMon [30]
Cardinality estimation	Reporting the number of distinct items in S.	Frequency-Focus	FM sketch [17] UnivMon [30]
Heavy hitter detection	Reporting the set of items whose frequencies are larger than the predefined threshold $\mathcal{T}$, together with their frequencies, *i.e.* reporting $\{(e, f_e) \mid f_e \geqslant \mathcal{T} \wedge f_e \in U\}$.	Heavy-Focus	SpaceSaving [32] Frequent [14] UnivMon [30]
Heavy change detection	Given two streams S_1 and S_2, reporting the set of items whose frequency differences in S_1 and S_2 are larger than the predefined threshold $\mathcal{T}$, together with their frequency differences, *i.e.* reporting $\{(e, f_e^1 - f_e^2) \mid \lvert f_e^1 - f_e^2 \rvert \geqslant \mathcal{T} \wedge f_e \in U\}$, where f_e^1 and f_e^2 denote the frequency of e in S_1 and S_2, respectively.	Heavy-Focus	k-ary [23] Rev sketch [35] UnivMon [30]

heavy hitters and heavy changes, because these two tasks need L1 heavy hitters (L1HH), and L2HH is a stronger notion than L1HH. The key contribution of UnivMon is that it is generic, and can handle various tasks. However, its key idea of universal-streaming produces a dozen of sub-streams, and thus processing one item often needs to access multiple sub-streams, which is slow.

While sketches are powerful tools for summarizing streams, many stream processing systems are based on sketches or use sketches as a component, and our proposed HeavyCache is orthogonal to these works. OpenSketch [41] proposed a sketch framework for network measurement, which targets at sketching for multiple tasks efficiently, and this goal is similar to ours. OpenSketch divides the recording phase into three stages, and implement different sketches by these stages. It allows different sketches share the same components. In other words, OpenSketch explores the commonality in different sketch algorithms, while our proposed HeavyCache focus on the commonality in different tasks. Some approximate query processing systems [5], [13] use sketches as basic operators to summarize the streaming data, and transform queries of users to queries to the sketches. To improve the recording speed of sketches, SketchVisor [22] build a fast path for a sketch to record items those cannot handled by the sketch, and use compressive sensing to recover information in the fast path during reporting.

III. HEAVYCACHE ALGORITHM

In this section, we give a comprehensive description of HeavyCache. We first give a high-level overview in Section III-A. Then we describe how HeavyCache processes an item during recording in Section III-B. Finally, we discuss the benefits of HeavyCache and some implementations issues in Section III-C.

A. Motivation and Overview of HeavyCache

HeavyCache is motivated by two key observations. First, both Heavy-Focus and Frequency-Focus tasks do not need to maintain keys of light items (see Section II). Thus, it is sufficient for all the aforementioned tasks to (i) record keys and frequencies of heavy items and (ii) record frequencies of light items. Second, the workloads incurred by heavy items and light items are different due to the typically skewed data streams. Particularly, a small number of heavy items may account for a large number of occurrences in a stream, making it consumes more resources to process heavy items.

Based on the two observations, HeavyCache separates the processing of heavy items and light items into two parts, namely the *cache part* and *count part*. The cache part tracks both keys and frequencies for every item in it to achieve higher accuracy, while the count part employs a count array to approximately count the remaining items. We also design a mechanism to automatically switch an item between the two parts. Since heavy items are more important for many data mining tasks, the mechanism guarantees that most heavy items are stored in the cache part at the end of a data stream. To this end, the cache part is designed to leverage the locality of heavy items in our context. Particularly, given that heavy items dominate skewed streams in the real world, most items only need to access the cache part, incurring limited processing overheads. We will elaborate how the separation design boosts both the speed in the recording phase and the accuracy in the reporting phase in the rest of this section.

B. Recording Approach

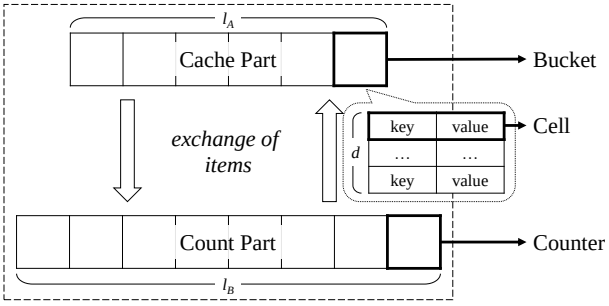


Fig. 1. Data structure of HeavyCache.

Data structure: As shown in Figure 1, a HeavyCache is composed of two parts: a *cache part* A and a *count part* B.

The cache part is composed of l_A buckets, and each bucket is composed of d cells. Each cell stores the information of an item in the form of key-value pairs, where the key is identifier of the item and the value is a counter represents the number of items. Let $A[i]$ be the i^{th} bucket of the cache part. The count part is a counter array composed of l_B counters. Each counter is composed of w bits, representing a number in range $[0, 2^w - 1]$. Let $B[i]$ be the i^{th} counter of the count part. A HeavyCache is associated with two hash functions, $h(.)$ and $g(.)$, and their outputs are uniformly distributed in range $[0, l_A - 1]$ and $[0, l_B - 1]$, respectively. We use $h(\alpha)$ or $g(\alpha)$ to denote the hash value of the key of item α. Next, we describe the algorithm, namely the CACHE algorithm, which makes the cache part store mostly heavy items, and the count part store mostly light items.

Algorithm 1: CACHE algorithm for HeavyCache

```
   Input: Item α to insert
1  lookup α in bucket A[h(α)];
2  if a cell (C) in A[h(α)] with key α found then
3  |  C.val ← C.val + 1;
4  else if an empty cell (C) found then
5  |  C.key ← α;
6  |  C.value ← 1;
7  else
8  |  B[g(α)] ← B[g(α)] + 1;
9  |  C_min ← cell with the smallest value in A[h(α)];
10 |  if C_min.val < B[g(α)] then
11 |  |  B[g(C_min.key)] ←
   |  |   max(B[g(C_min.key)], C_min.value);
12 |  |  C.key ← α;
13 |  |  C.value ← B[g(α)];
```

CACHE algorithm: HeavyCache uses an algorithm called *Check And Compare, Hold or Evict (CACHE)* (Algorithm 1), to separate heavy items from light items. The *initialization* of HeavyCache is to clear all the cells and counters. The insertion of an item α proceeds in the following three steps.

1) Check (line 1∼6). HeavyCache computes the hash function and locates a bucket in the cache part, $A[h(\alpha)]$, and checks whether α matches any of the keys in the d cells. If a cell is matched, HeavyCache increments the value of that cell by 1 and the insertion ends. Otherwise, HeavyCache checks whether $A[h(\alpha)]$ has an empty cell. If it is the case, HeavyCache sets the key of the empty cell to α and the value to 1, and the insertion ends, and we say that the item incurs a *cache hit* in this case. Otherwise, HeavyCache continues to the next step.

2) Compare (line 8∼10). HeavyCache computes $g(\alpha)$ and locates a counter in the count part, $B[g(\alpha)]$. HeavyCache then increments the counter by 1 and gets its new value v. Next, HeavyCache gets the cell $\mathcal{C}_{min}$ that holds the smallest value among all the d cells in $A[h(\alpha)]$, and compares the value of $\mathcal{C}_{min}.val$ with v. Then, HeavyCache continues to the next step.

3) Hold or Evict (line 10∼13). If the value of $\mathcal{C}_{min}$ is larger, HeavyCache holds the key-value pair in $\mathcal{C}_{min}$, and the insertion ends, and we say that the item incurs a *cache miss* in this case. Otherwise, HeavyCache evicts the item in $\mathcal{C}_{min}$ to the count part, and places α into $\mathcal{C}_{min}$. Specifically, HeavyCache computes $g(\mathcal{C}_{min}.key)$ and locates a counter in the count part, $B[g(\mathcal{C}_{min}.key)]$. HeavyCache sets this counter to $max(\mathcal{C}_{min}.val, B[g(\mathcal{C}_{min}.key)])$. Then, HeavyCache sets the key-value pair in $\mathcal{C}_{min}$ to $< \alpha, \mathcal{C}_{min}.val + 1 >$. The insertion ends, and we say that the item incurs an *eviction* in this case.

Notice that when the eviction happens, as is discussed in the above paragraph, the value of $\mathcal{C}_{min}$ is set to $\mathcal{C}_{min}.val + 1$ instead of v. This strategy, called *consecutive increment*, reduces the over-estimation error of frequency estimation of HeavyCache and guarantees that HeavyCache still has only one-sided error at the same time. This is because when switching α to the cache part, we actually need to set the value of $\mathcal{C}_{min}$ to the estimated frequency of item α, and this size is no larger than $min(\mathcal{C}_{min}.val+1, v)$. The detailed proof of one-sided error is shown in Section V.

Examples of CACHE algorithm: When inserting an item δ, one of the following four cases will happen, and we use one example per case in Figure 2 to show how HeavyCache works.

Case I: HeavyCache locates the bucket $A[h(\delta)]$ and finds a cell c with key δ, so HeavyCache increments the value of c from 3 to 4. The insertion ends.

Case II: HeavyCache locates the bucket $A[h(\delta)]$ and cannot find a cell with key δ, but finds an empty cell c. So HeavyCache sets the key of c to δ and sets the value of c to 1. The insertion ends.

Case III: HeavyCache locates the bucket $A[h(\delta)]$ and can neither find a cell with key δ nor a cell with key *None*. In this case, HeavyCache first checks the count part, locates the counter $B[g(\delta)]$, and increments the counter from 1 to 2. Then HeavyCache finds the cell c with the smallest value among all the cells in bucket $A[h(\delta)]$. The value of c is 3 and larger than δ's value 2. Thus, the insertion ends.

Case IV: HeavyCache locates the bucket $A[h(\delta)]$ and can neither find a cell with key δ nor a cell with key *None*. First, HeavyCache checks the count part, locates the counter $B[g(\delta)]$, and increments the counter from 8 to 9. Second, HeavyCache finds the cell c with the smallest value among all the cells in bucket $A[h(\delta)]$, and finds the value of c is 3, which is smaller than δ's value, 9. Therefore, HeavyCache inserts the current key-value pair of c, $(\varepsilon, 3)$, into the count part. It locates the counter $B[g(\varepsilon)]$ and sets its value to the larger number between the current value of the counter and the recorded value 3. So $B[g(\varepsilon)]$ is set to 3. Third, HeavyCache sets the key-value pair of c to $(\delta, 4)$. Notice that the value of c is not set to the value returned by the count part, which is 9. According to our proposed strategy of consecutive increment, HeavyCache sets the current value of c to 4 $(3 + 1)$. The insertion ends.

C. Discussion

The benefits of CACHE algorithm come from three aspects: accuracy, memory efficiency, and processing speed.

1) *Accuracy*: The CACHE algorithm caches and separates heavy items from light items, making both of them more accurate. Heavy items are mostly stored in the cache part,

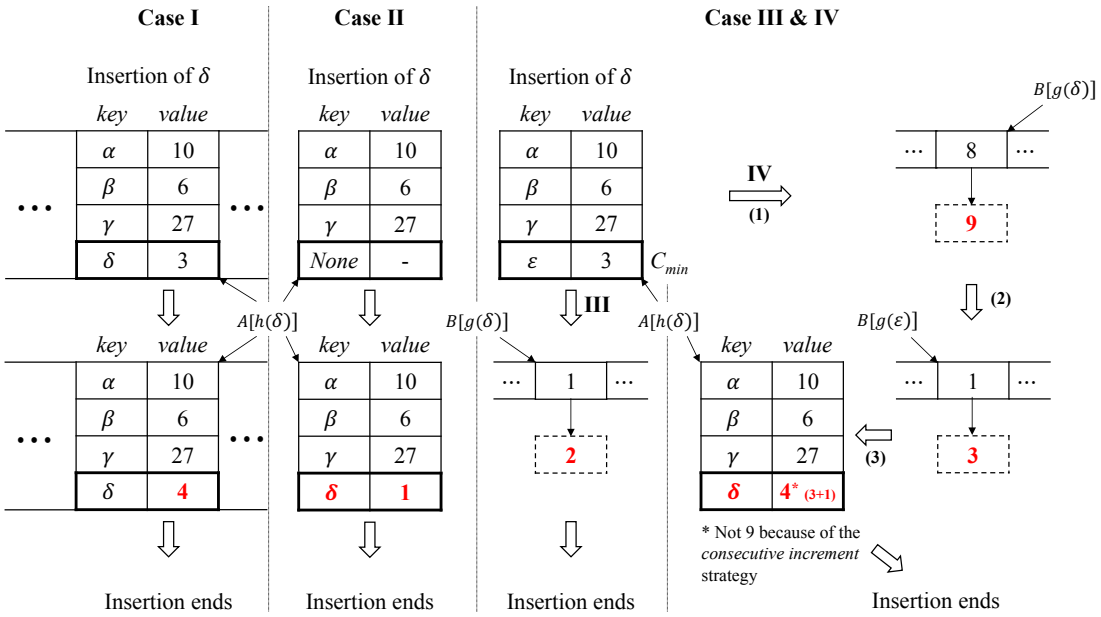


Fig. 2. Four cases encountered during insertion.

where each item has its own cell, so collision will never happen and the value in each cell can be very accurate. heavy items are rarely stored in the count part so only a few light items will collide with heavy items in a counter. As a result, it rarely happens that the light items overestimated greatly by their collisions with heavy items.

2) *Memory efficiency*: In a data structure of sketch, the counters should be large enough to accommodate the largest frequency of items. It is a waste of memory to store light items in such large counters. Unfortunately, the skewness of realistic data streams amplifies such waste, because the number of light items is much larger than the number of heavy items. In HeavyCache, due to the efficient separation of heavy items and light items, we can use large counters in the cache part and small counters in the count part. This improves the memory efficiency.

3) *Processing speed*: The recording procedure of HeavyCache is very simple. A cache hit and a cache miss need one and two memory accesses, respectively. When an eviction happens, HeavyCache needs to access the bucket of the inserted item in the cache part, and the counters of both the inserted item and the evicted item in the count part, so three memory accesses are needed. Furthermore, due to the skewness of realistic data stream, a large proportion of items belong to heavy items, and heavy items are mostly stored in the cache part. As a result, when an item is being inserted, it is likely that this insertion results in a cache hit. Therefore, a large part of insertions of items only needs one hash computation and one memory access in the cache part, which is very fast. For example, in our evaluation (Section VI), about 80% of items incur cache hits when dealing with network traffic streams, while less than 1% of items incur evictions.

In the software implementation of HeavyCache, we can use SIMD instructions to accelerate the key comparisons in the cache part. We can compare multiple keys in one single SIMD instruction. For example, the AVX2 instruction set supports that 8 32-bit integers can be compared with other 8 32-byte integers in one instruction. To leverage the power of SIMD instructions, we have to make the keys sequentially stored in a bucket. So if one bucket contains 4 cells, we put the 4 keys together, and then store the 4 values. To support an item with a longer key, we calculate a 32-bit fingerprint as its key, and we store its key in the other place. During comparison, we should match the keys if the fingerprints are matched, and this will incur an additional memory access. If we care more about speed, we can only match the fingerprints of the keys. This will cause false positives, because different keys may have the same fingerprint. The false positive rate is determined by the fingerprint length and the number of distinct keys.

IV. Applications

Since HeavyCache records the keys of heavy items and frequencies of all items, it records all the information needed for various mining tasks. Generally speaking, the cache part of HeavyCache stores the keys and frequencies of almost all heavy items, and this information can be used to perform Heavy-Focus tasks. The count part provides information of remained items. This counter array can be approximately seen as the linear projection of remained items by hash function $g(.)$ [4]. Combing the information provided by these two parts, HeavyCache can perform Frequency-Focus tasks.

In this section, we show the specific reporting methods for six typical mining tasks: frequency estimation, heavy hitter detection, heavy change detection, frequency distribution, entropy estimation, and cardinality estimation.

Frequency estimation: Suppose HeavyCache needs to return the estimated frequency $\hat{f}_\alpha$ of item α. First, HeavyCache computes $h(\alpha)$ and locates a bucket in the cache part, $A[h(\alpha)]$, and checks whether α matches any of the keys in the d cells.

[4]We use the word "approximately" because when an eviction occurs, our CACHE algorithm does not keep the linear property of the count part. But because only a small proportion of items incur evictions, the count part can be seen as a good approximation.

If is matched, the HeavyCache returns the value of that cell as $\widehat{f_\alpha}$. Otherwise, the HeavyCache checks if any of the keys in the d cells are *None*. In this case, HeavyCache returns 0 as $\widehat{f_\alpha}$. Otherwise, HeavyCache computes $g(\alpha)$ and locates a counter in the count part, $B[g(\alpha)]$. Let v denotes the value of $B[g(\alpha)]$. Then HeavyCache finds the smallest value v_{min} in bucket $A[f(\alpha)]$ In this case, as mentioned in Section III-B, HeavyCache returns $min(v_{min}, v)$ as $\widehat{f_\alpha}$.

Similar to CM sketches [11] and CU sketches [16], an estimated frequency given by HeavyCache only has over-estimation error. Furthermore, different with traditional sketches, HeavyCache can give a non-trivial upper bound (not the value it self) of the over-estimation error if this item is stored in the cache part. In other words, HeavyCache can give the range of the real frequency of item α in the cache part: the real frequency of item α is between $\widehat{f_\alpha} - min(v_{min}, v)$ and $\widehat{f_\alpha}$. This means that the over-estimation error is no larger than the corresponding counter value in the count part and the minimum value of the corresponding bucket in the cache part. Since almost none of heavy items is in the count part, the counter value is usually small, so this means the range given by HeavyCache is usually small, and the over-estimation error cannot be too large. The detailed proof of one-sided error is shown in Section V.

Heavy hitter detection: Suppose HeavyCache needs to get all the heavy hitters whose frequency is over the threshold $\mathcal{T}$ after the insertion of items. HeavyCache supports this task natively because HeavyCache records the heavy items in its cache part. To get the heavy hitters, HeavyCache first collects all the valid key-value pairs (the key field is not *None*) in the $l_A \cdot d$ cells in the cache part, and then returns the item whose size is over $\mathcal{T}$ among them.

Heavy change detection: Suppose HeavyCache needs to detect the items whose frequency differences are larger than a threshold $\mathcal{T}$ between data stream t_1 and t_2. If an item is a heavy change, this item must be a heavy hitter whose size is larger than $\mathcal{T}$ in at least one of the two data streams. Therefore, HeavyCache first gets the heavy hitters from two data streams. Suppose the heavy hitter sets of two data streams are P_1 and P_2, respectively. HeavyCache considers their union $P_1 \cup P_2$ as the candidates of heavy changers. Then, for each item $\alpha \in P_1 \cup P_2$, HeavyCache performs the frequency estimation in two data streams, and get two estimated frequencies f_α^1 and f_α^2. If $|f_\alpha^1 - f_\alpha^2| \geqslant \mathcal{T}$, HeavyCache consider this item as a heavy changer and the change size is $|f_\alpha^1 - f_\alpha^2|$.

Frequency distribution and entropy estimation: To estimate the frequency distribution of a set of items in a data stream, HeavyCache calculates the distribution for the cache part and the count part separately, and then combine the two distribution and get the final result. For the cache part, HeavyCache uses the distribution of values in each bucket as the frequency distribution, because we expect that these values count the frequency exactly. For the count part, HeavyCache leverages the EM algorithm in MRAC [24] to calculate the frequency distribution from the counter array. Since the count part is only an approximation of a linear projection of items, the EM algorithm cannot converge to the exact result. We will show that this approximation incurs only small error in our evaluation (Section VI). Finally, HeavyCache combines the frequency distribution calculated from the cache part and the count part and returns the result. Once the frequency distribution is calculated, the entropy can be derived by the formulation $\sum n_j * j/m$, where j is the frequency and n_j is the number of items whose frequencies are j.

Cardinality estimation: Suppose HeavyCache need to get the cardinality of a set of items after recording the items of a data stream. For each item k in the cache part, HeavyCache computes $g(k)$ and locates a counter $B[g(k)]$ in the count part, and sets the value of $[B[g(k)]]$ to $max(B[g(k)], v)$. Then, HeavyCache utilizes the linear counting method [39] to compute the cardinality n. Specifically, let m be the number of counters, m_0 be the number of counters with value zero, and then the cardinality is computed as $n = m \cdot \ln\left(\frac{m}{m_0}\right)$.

V. Mathematical Analysis

In this section, we derive the error bound of HeavyCache when performing frequency estimation, which is presented in Theorem V.1:

Theorem V.1. *Let $f(\alpha)$ be the true frequency of an item α, $\widehat{f(\alpha)}$ be the estimated frequency of the item α, $\mathcal{A}$ be the bucket α is mapped to, $\mathcal{B}(\alpha)$ be the counter α is mapped to, and then at any time during insertion, the true and the estimated frequency of α satisfies the following inequality:*

$$\forall \alpha, f(\alpha) \leqslant \widehat{f(\alpha)} \leqslant f(\alpha) + min(min\_val, \mathcal{B}(\alpha)) \quad (1)$$

where min_val is the minimum value among all the cells of $\mathcal{A}$.

To prove Theorem V.1, we first prove Lemma V.1:

Lemma V.1. *Let $\mathcal{A}$ be any of the bucket in the cache part, S be the set of all the items that are mapped to $\mathcal{A}$, S_1 be the set of all the items that are currently residing in $\mathcal{A}$[5], S_2 be the set of all the items that are mapped to but not currently residing in $\mathcal{A}$, which means $S_2 = S - S_1$, $\mathcal{A}(\alpha)$ be the value of the cell that holds $\alpha(\alpha \in S_1)$, $f(\alpha)$ be the true frequency of item α. During insertions, the following inequalities are always satisfied:*

$$\begin{aligned} 0 \leqslant \mathcal{A}(\alpha) - f(\alpha) \leqslant min(min\_val, \mathcal{B}(\alpha)), \forall \alpha \in S_1 \\ f(\alpha) \leqslant min(min\_val, \mathcal{B}(\alpha)), \forall \alpha \in S_2 \end{aligned} \quad (2)$$

Proof. We prove it using mathematical induction. When the HeavyCache is initialized, all the cells and counters, and the true frequencies of all the items are 0. Therefore inequality 2 is satisfied.

Suppose inequality 2 is satisfied before the insertion of α. When HeavyCache is inserting an item α, one of four possible cases would happen:

Case I: $\alpha \notin S$. All values in $\mathcal{A}$ are not changed, and the values of counters can only be bigger, so the above inequality still holds true.

Case II: $\alpha \in S_1$.

[5] When the bucket $\mathcal{A}$ is not full (empty cell still exists in $\mathcal{A}$), the next items to be inserted into this bucket is always regarded as residing in $\mathcal{A}$.

- For $\alpha, \mathcal{A}(\alpha)_{new} - f(\alpha)_{new} = (\mathcal{A}(\alpha)+1) - (f(\alpha)+1) = \mathcal{A}(\alpha) - f(\alpha)$, $min\_val_{new} \geqslant min\_val$, so the inequality holds true.
- For $\forall \beta \in (S-\alpha)$, $min\_val_{new} \geqslant min\_val$ and $\mathcal{A}(\beta)$, $f(\beta)$, $\mathcal{B}(\beta)$ are not changed, so the inequality holds true.

Case III: *$\alpha \in S_2$ and eviction does not happen.*

- For $\beta \in S_1$, $\mathcal{A}(\beta), f(\beta), \mathcal{B}(\beta), min\_val$ are not changed, so the inequality holds true.
- For α, $min\_val(\alpha) \geqslant \mathcal{B}(\alpha)_{new} = \mathcal{B}(\alpha)+1$, so $min(min\_val, \mathcal{B}(\alpha)_{new}) = \mathcal{B}(\alpha)+1 \geqslant f(\alpha)+1 = f(\alpha)_{new}$.
- For $\beta \in S_2 - \alpha$, $\mathcal{B}(\beta)_{new} \geqslant \mathcal{B}(\beta)$ and $f(\beta), min\_val$ are not changed, so the inequality holds true.

Case IV: *$\alpha \in S_2$ and eviction happens. α moves from S_2 to S_1 and ε moves from S_1 to S_2.*

- For α, $\mathcal{A}(\alpha)_{new} - f(\alpha)_{new} = min\_val + 1 - (f(\alpha)+1) = min\_val - f(\alpha) \geqslant min(min\_val, \mathcal{B}(\alpha)) - f(\alpha) \geqslant 0$. On the other hand, $min\_val - f(\alpha) \leqslant min\_val \leqslant \mathcal{B}(\alpha)+1 = \mathcal{B}(\alpha)_{new}$, therefore $\mathcal{A}(\alpha)_{new} - f(\alpha)_{new} \leqslant min(min\_val, \mathcal{B}(\alpha)_{new})$.
- For $\beta \in S_1 - \alpha$, $\mathcal{B}(\beta)_{new} \geqslant \mathcal{B}(\beta), min\_val_{new} > min\_val$, and $\mathcal{A}(\beta), f(\beta)$ are not changed, so the inequality holds true.
- For ε, $\mathcal{B}(\varepsilon)_{new} = max(\mathcal{B}(\varepsilon), min\_val)$, therefore $f(\varepsilon)_{new} = f(\varepsilon) \leqslant \mathcal{A}(\varepsilon) = min\_val = min(min\_val_{new}, \mathcal{B}(\varepsilon)_{new})$.
- For $\beta \in S_2 - \varepsilon, \mathcal{B}(\beta)_{new} \geqslant \mathcal{B}(\beta), min\_val_{new} > min\_val$, and $f(\beta)$ is not changed, so the inequality holds true.

Therefore, the above inequality is always satisfied, and Lemma V.1 holds true. □

Finally, we prove that Theorem V.1 can be derived from Lemma V.1.

Proof. Since HeavyCache returns $\mathcal{A}(\alpha)$ as the estimated frequency when $\alpha \in S_1$, and $min(min\_val, \mathcal{B}(\alpha))$ as the estimated frequency when $\alpha \in S_2$, based on Lemma V.1, $0 \leqslant \widehat{f(\alpha)} - f(\alpha) \leqslant min(min\_val, \mathcal{B}(\alpha))$ can always be satisfied, so the correctness of Theorem V.1 is proved. □

VI. Experimental results

In this section, we first describe the setup of our experiments, and then show the performance of HeavyCache under different settings. Next, we compare HeavyCache with prior work for each task in terms of speed and accuracy.

A. Experiment Setup

1) Tasks and Implementation

We conduct experiemnts on the aforementioned six tasks, and for each task, we compare our HeavyCache with following algorithms:

- **Frequency estimation:** Count Sketch [7], CM Sketch [11], CM-CU [16], CSM [28]
- **Heavy hitter detection:** Count+Heap [7], CM+Heap [11], Space-Saving [32], UnivMon [30]
- **Heavy change detection:** UnivMon [30]
- **Cardinality estimation:** MRAC [24]
- **Frequency distribution estimation :** UnivMon [30]
- **Entropy estimation:** UnivMon [30]

We implement all these algorithms in C++, and all the experiments are performed on a server with two 8-core CPUs (Intel Xeon E5-2630V3@2.40GHz). The source codes of all algorithms are available at [2].

In terms of the parameter settings, for HeavyCache, we allocate 0.3 MB memory for the cache part, and vary the memory for the count part. We allocate 4 bytes for each counter in the cache part and 2 bytes for each counter in the count part for HeavyCache. And as for other algorithms, we allocate 4 bytes for each counter to accommodate the maximum size of the keys of items. All the hash functions are Bob hash as recommended in [1]. For sketches (Count [7], CM [11], CU [16], *etc.*), we set the number of hash functions to 3 as recommended in [19]. As for a sketch plus a heap (CM + Heap and Count + Heap), we set the capacity of heap to 4096, and ignore the memory cost of the heap. As for UnivMon, we set the number of levels to 14, and each level records 1000 heavy hitters. As for MRAC, since we conduct experiments on different memory sizes, we do not implement the multi-resolution version as discussed in its original paper.

2) Test Streams

We use three different networking IP traces in our experiments. In IP traces, each item in the stream is called a packet, and the set of packets whose flow IDs (keys) are same is called a flow. The details of those IP traces are shown in Table II. The first two traces are obtained from CAIDA [3], and the last one is captured from the backbone network in our campus. Each trace contains 10-minute traffic data, and is split into ten 1-minute traces. For each trace, we consider two commonly used flow IDs as item keys: source IP (SrcIP) and 5-tuple, whose lengths are 4 bytes and 13 bytes, respectively. We plot the mean value of the results in a trace in each figure, and the error bars show the min and max values.

3) Metrics

We consider the following metrics in our experiments:

Throughput: Throughput is used to measure the recording speed of the tested algorithms, and is defined as N/T, where N is the number of items in a stream and T is the amount of time used to process this stream. We use Million insertions per second (Mips) to measure the throughput.

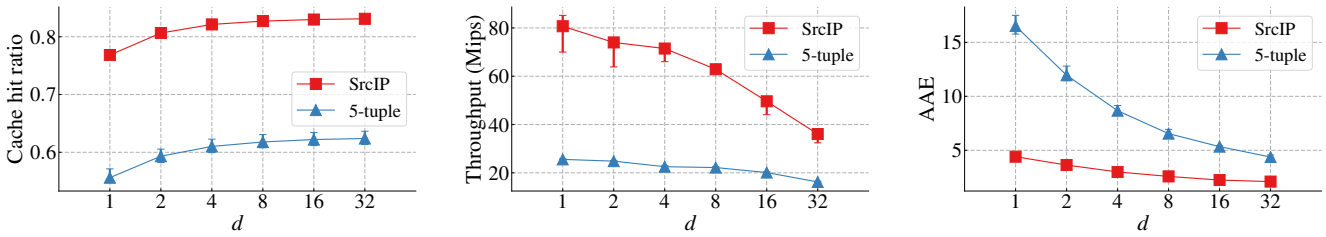
Average absolute error (AAE): AAE is used to measure the accuracy in three experiments including frequency distribution estimation, heavy hitter detection, and heavy change detection. Let m be the number of queries, $\hat{f}_i$ be the i-th reported value, and f_i denotes the i-th real value. AAE is defined as $(\sum |\hat{f}_i - f_i|)/m$.

Precision and Recall: Precision and recall are used to measure the accuracy of the reported results in the experiment of heavy hitter detection and heavy change detection. Let S_{est} be the reported set, and S_{real} be the real set. Then, precision is defined as $|S_{est} \cap S_{real}|/|S_{est}|$, and recall is defined as $|S_{est} \cap S_{real}|/|S_{real}|$.

Relative error (RE): RE is the relative error of a value, and is used to measure the accuracy in the experiments of Cardinality and Entropy. Let $\hat{v}$ be the reported value, and v be the true value. RE is defined as $|\hat{v} - v|/v$.

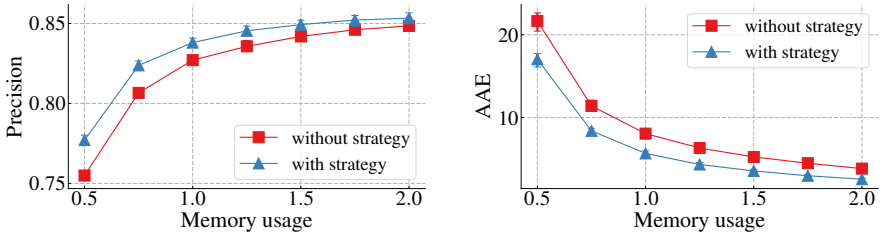
TABLE II
TEST STREAMS IN EXPERIMENTS

Trace	Location	Date	# packets	# flows (SrcIP)	# flows (5-tuple)
CAIDA1	Equinix-Chicago	2015/12/17	25.4M∼28.4M	0.59M∼0.62M	1.23M∼1.31M
CAIDA2	Equinix-Chicago	2016/04/06	29.3M∼32.1M	0.44M∼0.47M	0.97M∼1.02M
Campus	–	2013/10/24	11.8M∼12.2M	0.38M∼0.47M	1.22M∼1.36M



(a) Cache hit ratio (b) Throughput (c) AAE of items in the Cache part

Fig. 3. The performance of HeavyCache with different d (number of cell per bucket). (a) shows the percentage of items that incur cache hits. (b) shows the recording speed. (c) shows the accuracy of the cache part, in terms of the average absolute error of items.



(a) Cache precision (b) Cache AAE

Fig. 4. Impact of the consecutive increment strategy. Cache precision means the percentage of correctly cached items. It is defined as n/N, where N is the capacity of the cache part, and n is number of items in the cache part whose sizes are no smaller N-th largest item in the traffic stream.

Weighted mean relative error (WMRD): WMRD can measure the distance between two vectors, and is used to measure the accuracy in the frequency distribution estimation experiment [24], [25]. Suppose the reported frequency distribution is $< \hat{n}_1, \hat{n}_2, \ldots, \hat{n}_m >$ and the real frequency distribution is $< n_1, n_2, \ldots, n_m >$, where $\hat{n}_i$ and n_i is the reported number and the real number of items whose frequencies are i, respectively. Then WMRD is defined as $\sum |n_i - \hat{n}_i| / \sum (\frac{n_i + \hat{n}_i}{2})$.

B. Experiments on HeavyCache

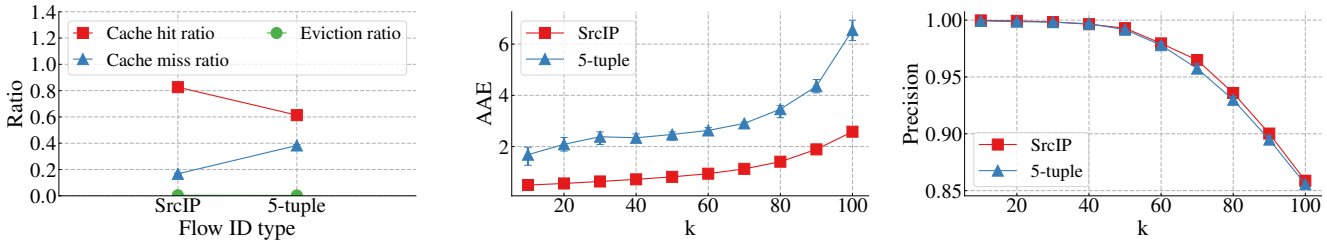
In this section, we show the performance of HeavyCache under two design decisions: the number of cells per bucket and the consecutive increment strategy. Then we show some properties of HeavyCache. We have conducted experiments on all the three traces and show only the results on CAIDA1 due to the space limitation.

Choice of d (Figure 3): In this experiment, the memory allocated to HeavyCache is 2MB, 0.3MB for the cache part and 1.7MB for the count part. According to Figure 3(a), when d is increasing, the number of items which hit in the cache part is increasing. However, according to Figure 3(b), the results show that the throughput decreases when d is increasing, and the throughput drops significantly when d exceeds 8. Though more items hit in the cache part, the throughput is still decreasing because a larger d causes more key comparisons in the cache part, which cost more time. According to Figure 3(c), the AAE of the cache part decreases when d is increasing, which means that the larger d is, the more accurate HeavyCache is.

To sum up, there is a tradeoff between the recording throughput and the accuracy of HeavyCache with regarding to d, and we fix d to 8 in the following experiments.

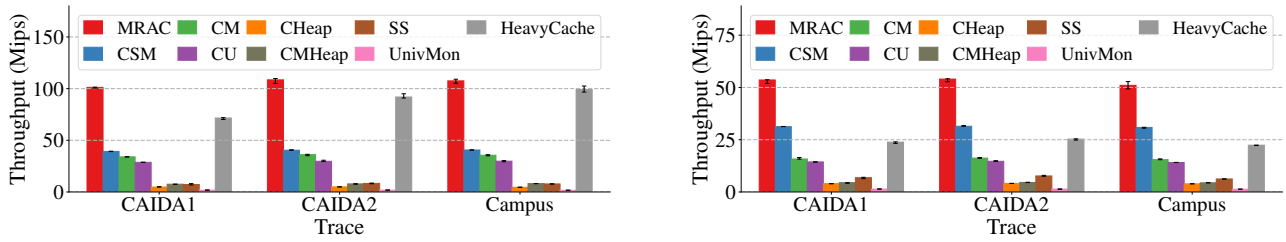
Impact of consecutive increment strategy (Figure 4): In this experiment, the memory for the cache part is fixed to 0.3MB and the memory for the count part is varied from 0.25MB to 2.0MB. The results show that the consecutive increment strategy does improve the accuracy of the cache part in terms of both precision and AAE. However, with memory usage increasing, the improvement becomes smaller, because less collision occurs in the count part.

Properties of HeavyCache (Figure 5): In this part, we show some properties of HeavyCache. First, we test the cache hit ratio, cache miss ratio and eviction ratio of HeavyCache when recording SrcIP traces and 5-tuple traces. According to Figure 5(a), the results show that, when items in a stream are recorded, most of them hit in the cache part, and few of them are stored in the count part and cause cache miss, hardly any of them causes eviction. Second, we sort the items in the cache part in descending order by their frequencies and show the AAE and precision of the first $k\%$ items, where k ranges from 10 to 100. Figure 5(b) shows that with the increment of k, the AAE is increasing when recording SrcIP traces and 5-tuple traces. According to Figure 5(c), the results show that, with the increment of k, the precision is decreasing when recording SrcIP traces and 5-tuple traces. This means the larger the item is, the more accurately HeavyCache reports.



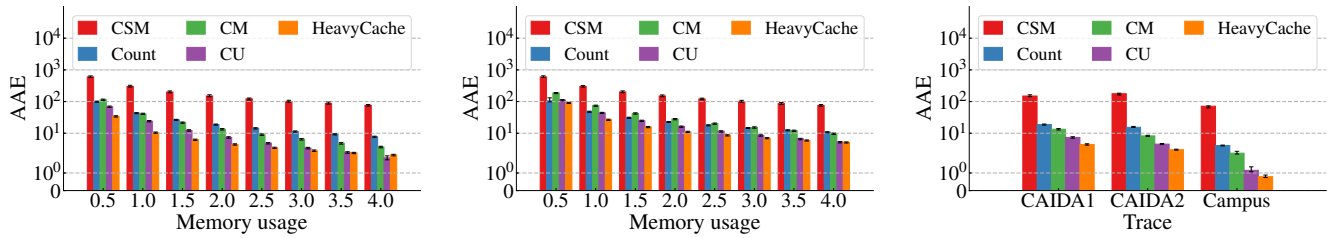
(a) Proportion (b) AAE (c) Precision

Fig. 5. Properties of HeavyCache. (a) shows the proportions of cache hit, cache miss and eviction when recording the traffic streams. (b) and (c) show the accuracy of the first $k\%$ items in the cache part.



(a) Throughput vs. Trace (SrcIP) (b) Throughput vs. Trace (5-tuple)

Fig. 6. Recording speed of HeavyCache and other algorithms using different traces. CHeap and SS are the abbreviations of Count+Heap and Space-Saving, respectively.



(a) AAE (SrcIP) (b) AAE (5-tuple) (c) AAE vs. Trace (SrcIP)

Fig. 7. Accuracy of frequency estimation. (a) and (b) show AAE performance of the five algorithms under different memory usages on CAIDA1. (c) shows the AAE performance on different traces.

C. Experiments on Recording Speed

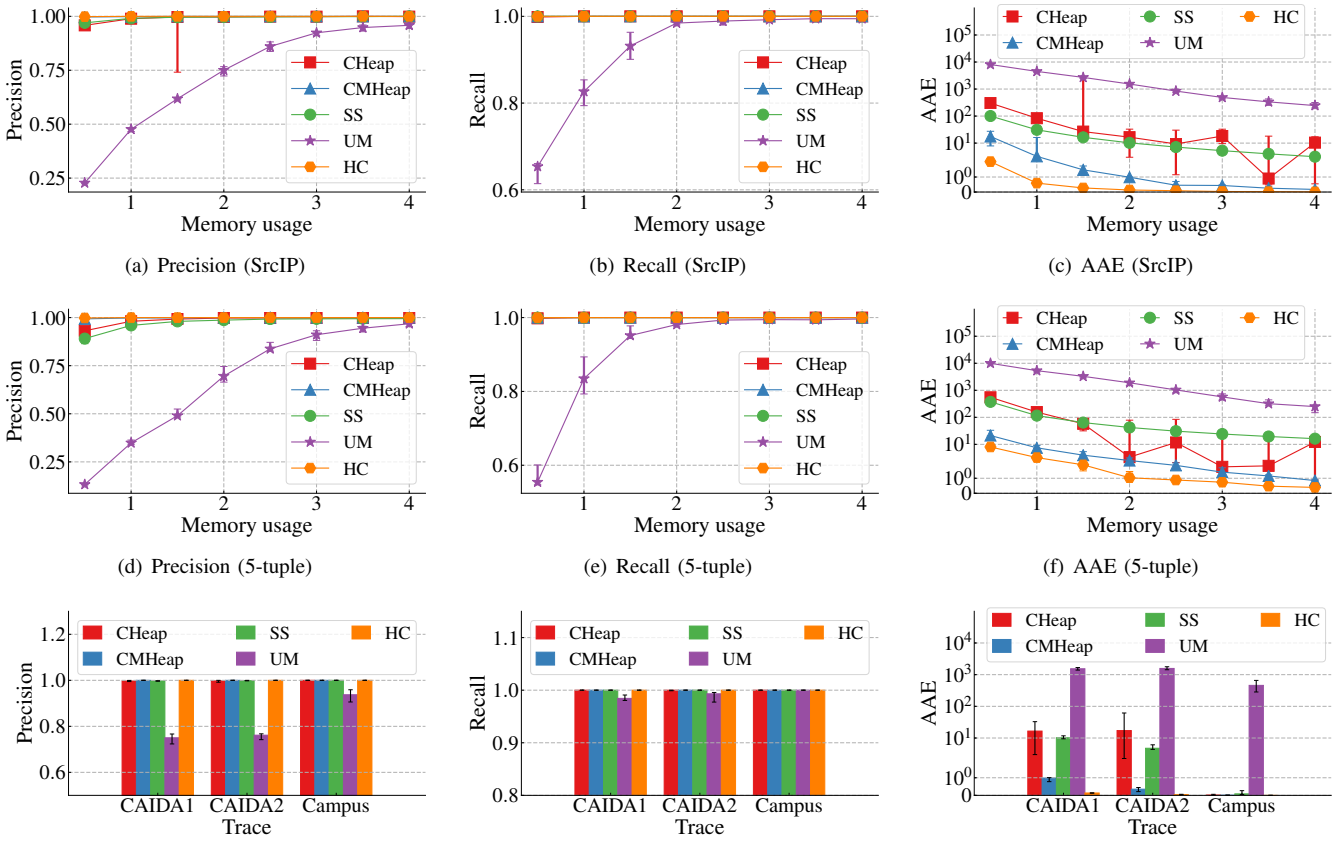
The results show that the throughput of HeavyCache is about 2.06, 2.45, 1.79, 8.44, 36.84, 13.59 and 9.31 times higher than CM, CU, CSM, Space-Saving, UnivMon, Count +Heap and CM+Heap when recording SrcIP traces, respectively. When recording 5-tuple traces, the throughput of HeavyCache is 1.48, 1.62, 3.66, 16.93, 5.80 and 5.23 times higher that CM, CU, Space-Saving, UnivMon, Count+Heap and CM+Heap respectively. The recording throughput of HeavyCache is lower than MRAC when recording SrcIP traces and is lower than MRAC and CSM when recording 5-tuple traces. The reason is that the recording procedures of MRAC and CSM are simple. Specifically, MRAC needs only one hash computation and one memory access to record an item, and CSM needs only two hash computations and one memory access to record an item.

From the comparison of Figure 6(a) and Figure 6(b), we find that the longer keys of items (5-tuple traces) result in the decrease of recording throughput for all these algorithms. Unfortunately, the throughput of HeavyCache decreases more than other algorithms. The reason is that, on the one hand, the number of 5-tuple items is larger than the number of SrcIP items in one time window, so the cache effect is better when recording SrcIP traces using the same number of memory. On the other hand, when dealing with 5-tuple traces, there are fewer cells in the cache part of HeavyCache, because the memory usage for a cell is larger to record a longer key of item. These two factors make HeavyCache have a bigger performance loss on recording throughput than other algorithms.

D. Experiments on Tasks

In this section, for each of the six tasks, we compare the performance of HeavyCache and other algorithms. Specifically, we conduct experiments to show the performance of HeavyCache and other tested algorithms using different memory sizes and traces. For brevity, when varying memory size, we show only the results on CAIDA1. For different traces, we show the results with the memory size fixed to 2MB. Note that the memory allocated to cache part of HeavyCache is fixed to 0.3MB, and d is fixed to 8.

Frequency Estimation (Figure 7): The results show that when dealing with SrcIP traces, the AAE of HeavyCache is 33.63, 3.00, 1.66 and 4.28 times lower than CSM, CM, CU and Count. When dealing with 5-tuple traces, the AAE of HeavyCache is 13.40, 2.52, 1.46 and 2.04 times lower than CSM, CM, CU and Count. HeavyCache gains the benefit from two aspects. On the one hand, HeavyCache uses smaller counters in the count part so that the number of counters is larger than other algorithms. On the other hand, HeavyCache records heavy items mostly in the cache part. Therefore, most of the collisions in the count part occur between light items,



(a) Precision (SrcIP) (b) Recall (SrcIP) (c) AAE (SrcIP)

(d) Precision (5-tuple) (e) Recall (5-tuple) (f) AAE (5-tuple)

(g) Precision vs. Trace (SrcIP) (h) Recall vs. Trace (SrcIP) (i) AAE vs. Trace (SrcIP)

Fig. 8. Accuracy of heavy hitter detection. (a) - (c) show the results of experiments on SrcIP traces. (d) - (f) show the results of experiments on 5-tuple traces. (g) - (i) show the results of experiments on different traces. CHeap, SS, UM and HC are the abbreviations of Count+Heap, Space-Saving, UnivMon and HeavyCache, respectively.

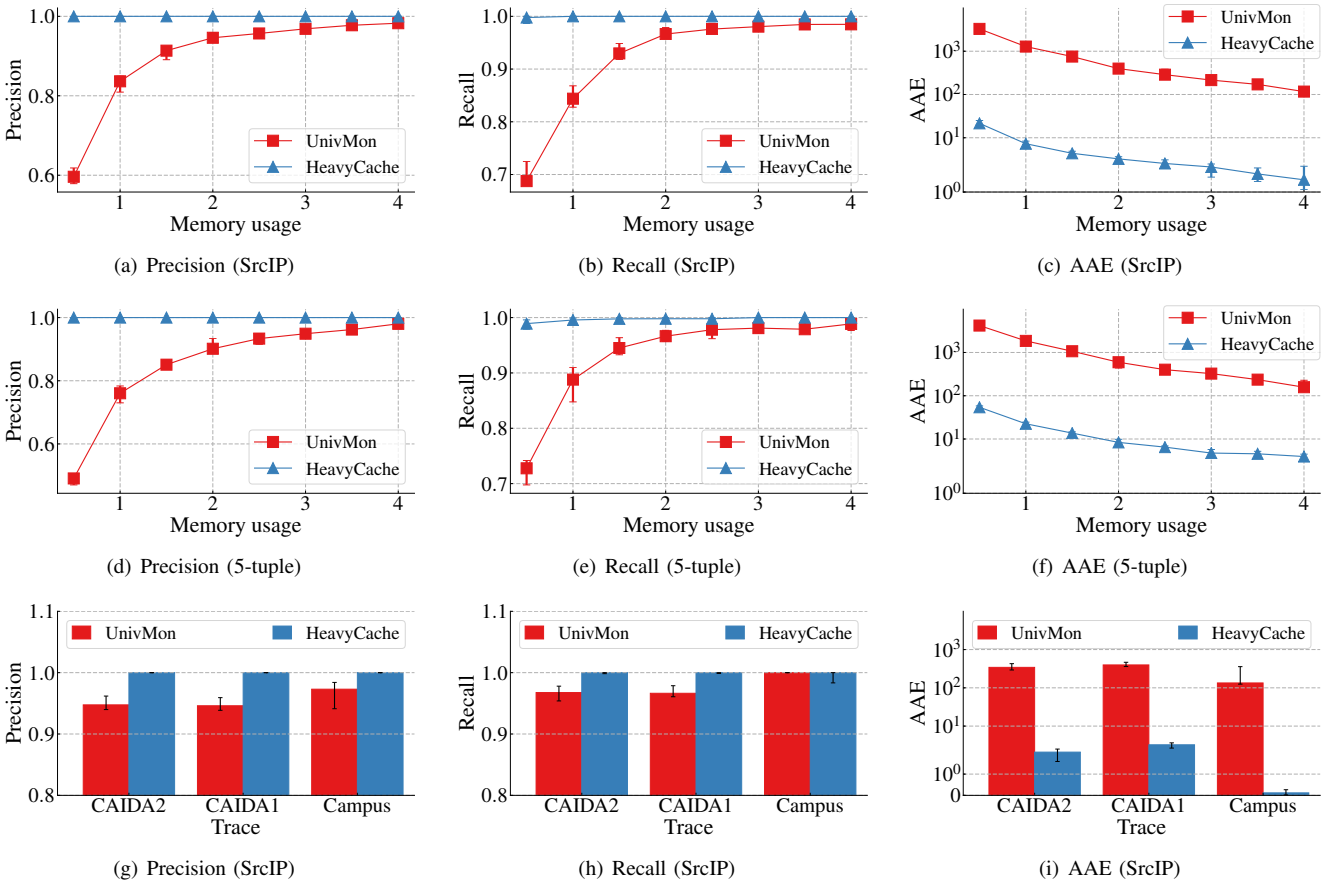
which reduces the over-estimation resulted by collisions.

Heavy Hitter Detection (Figure 8): According to Figure 8(a), 8(b), 8(d), and 8(e), the results show that the precision and recall of HeavyCache can achieve nearly 100% when the memory size varies from 0.5 to 4.0MB. According to Figure 8(c) and 8(f), when dealing with SrcIP traces, the AAE of HeavyCache is 0.14 when the memory size is 2.0MB, which is 58.51, 7.02, 73.16 and 9710.34 times lower than Count+Heap, CM+Heap, Space-Saving and UnivMon, respectively. When dealing with 5-tuple traces, the AAE of HeavyCache is 0.99 when the memory size is 0.5MB, which is 35.85, 2.65, 39.49 and 1746.51 times lower than Count+Heap, CM+Heap, Space-Saving and UnivMon, respectively. According to Figure 8(g), 8(h) and 8(i) on different traces, the precision and recall of HeavyCache achieve nearly 100%, and the AAEs of HeavyCache are under 3.3. HeavyCache gets a better accuracy because the heavy hitters are kept in the cache part, and the items do not suffer from collisions when they are stored in the cache part.

Heavy Change Detection (Figure 9): In this experiments, we compare traffic streams of two time windows and detect the heavy changes. According to Figure 9(a), 9(b), 9(d) and 9(e), the results show that the precision and recall of HeavyCache are nearly 100% when the memory size varies from 0.5 to 4.0MB, which means that HeavyCache can correctly detect almost all the heavy changes. As for AAE, when dealing with SrcIP traces, the AAE of HeavyCache is 19.9 when the memory size is 0.5MB and 1.03 when the memory size is 4.0MB, and the AAE of UnivMon is 3389.4 when the memory size is 0.5MB and 110.6 when the memory size is 4.0MB. When dealing with 5-tuple traces, the AAE of HeavyCache is 53.3 when the memory size is 0.5MB and 4.5 when the memory size is 4.0MB, and the AAE of UnivMon is 4117.5 when the memory size is 0.5MB and 151.3 when the memory size is 4.0MB. The performance on different traces also show that HeavyCache achieves better accuracy and recall and lower AAE than UnivMon.

Cardinality Estimation (Figure 10): According to Figure 10(a), when dealing with SrcIP traces, the RE of HeavyCache and the RE of UnivMon are similar, which are about 0.01 when the memory size is 0.5MB, and the RE of HeavyCache achieves about 33.5 times lower than UnivMon when the memory size is no less than 1.0MB. According to Figure 10(b), when dealing with 5-tuple traces, the RE of HeavyCache is 0.0115 and the RE of UnivMon is 0.0103 when the memory size is 0.5MB, and the RE of HeavyCache is about 476.5 times lower than UnivMon when the memory size is no less than 1.0MB. The results on different traces also show that HeavyCache achieves lower AAE than UnivMon.

Frequency Distribution Estimation (Figure 11): Figure 11 shows the WMRD of HeavyCache and MRAC after five iterations. According to Figure 11(a) 11(b) and 11(c), the

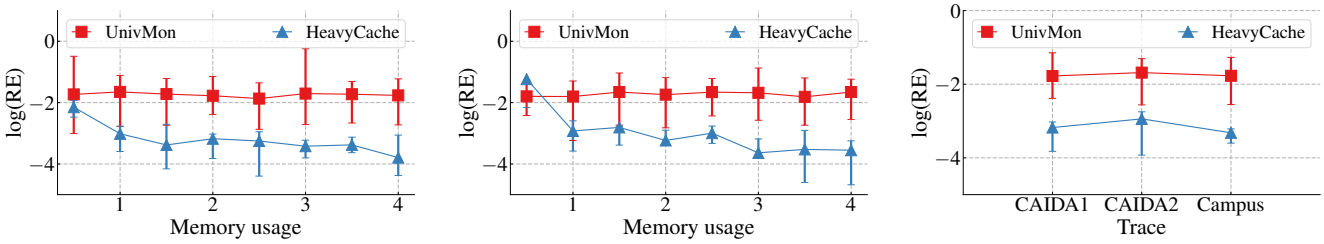


(a) Precision (SrcIP) (b) Recall (SrcIP) (c) AAE (SrcIP)

(d) Precision (5-tuple) (e) Recall (5-tuple) (f) AAE (5-tuple)

(g) Precision (SrcIP) (h) Recall (SrcIP) (i) AAE (SrcIP)

Fig. 9. Accuracy of heavy change detection. (a) - (c) show the results of experiments on SrcIP traces. (d) - (f) show the results of experiments on 5-tuple traces. (g) - (i) show the results of experiments on different traces.

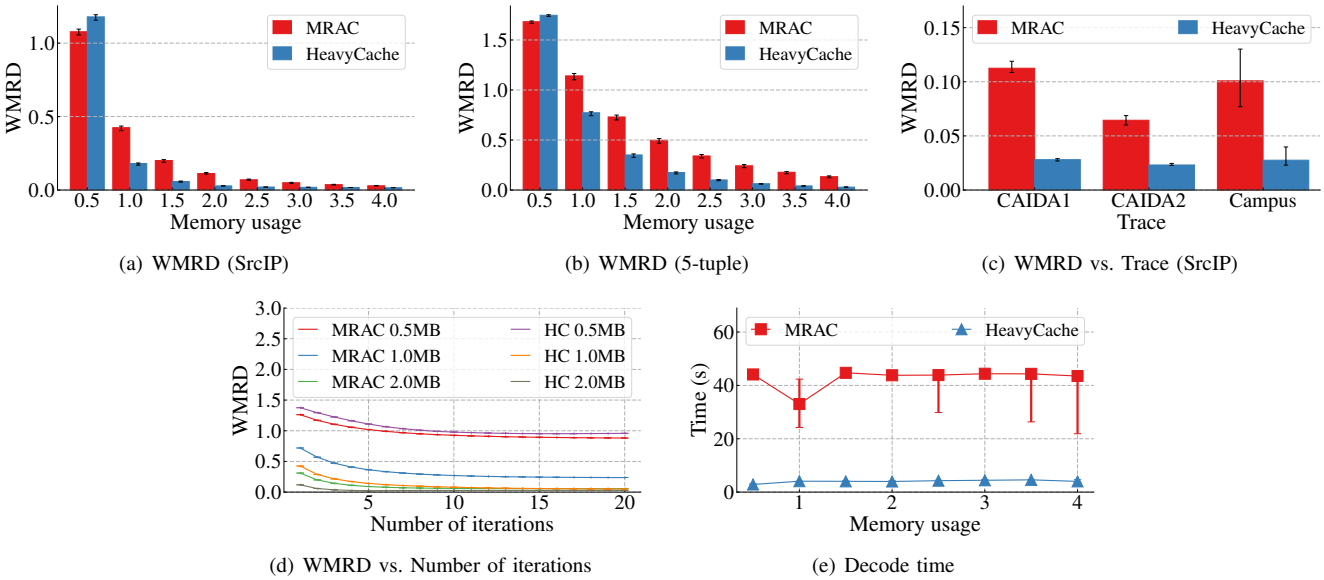


(a) RE (SrcIP) (b) RE (5-tuple) (c) RE vs. Trace (SrcIP)

Fig. 10. Accuracy of cardinality estimation. (a) and (b) show the relative error of the results produced by UnivMon and HeavyCache under different memory usage. (c) shows the relative error of results produced by UnivMon and HeavyCache on different traces. Note that the Y-axises represent $log_{10}(RE)$.

results show that, when the memory size is 0.5MB, MRAC shows a little advantage over HeavyCache. When the memory size is over 1.0MB, HeavyCache achieves a lower WMRD than MRAC. When the memory is over 1.0MB, HeavyCache has more counters than MRAC because it uses smaller counters. The reason is that when the memory size is 0.5MB, HeavyCache uses 0.3MB memory for the cache part and only 0.2MB for the count part, which may lead to a small number of counters. The results on different traces show that the RE of HeavyCache is about 3.4 times lower than MRAC. According to Figure 11(d), when the memory size is 0.5MB, the WMRD of MRAC is lower than HeavyCache after each iteration. When the memory size is 1.0MB and 2.0MB, the WMRD of MRAC is higher than HeavyCache after each iteration. According to Figure 11(e), under different memory size, the average running time per iteration of HeavyCache is about 11 times shorter than MRAC. This is because the maximum counter size is much smaller in HeavyCache, due to the efficient separation of heavy items and light items. And this means the EM algorithm can converge much more quickly in HeavyCache.

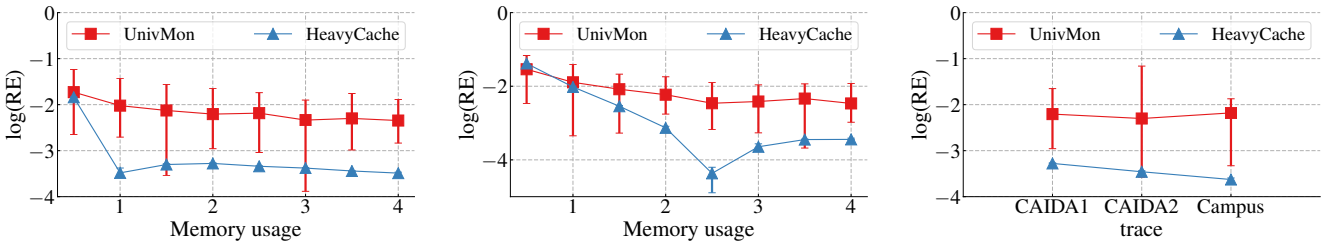
Entropy Estimation (Figure 12): According to 12(a), when dealing with SrcIP traces, the relative error of HeavyCache is about 0.015 when the memory is 0.5MB and is under 0.001 when the memory size is over 1.0MB. The relative error of UnivMon is about 0.024 when the memory is 0.5MB and is about 0.0074 when the memory size is 4.0MB. Figure 12(c) shows that HeavyCache can get relative error below 0.001 on different traces when the memory size is 2.0MB. The reason for HeavyCache's good performance on entropy estimation is that HeavyCache calculates the entropy by estimating the



(a) WMRD (SrcIP) (b) WMRD (5-tuple) (c) WMRD vs. Trace (SrcIP)

(d) WMRD vs. Number of iterations (e) Decode time

Fig. 11. Accuracy of frequency distribution estimation. (a) and (b) show the accuracy of frequency distribution estimated by MRAC and HeavyCache under different memory sizes. (c) shows the accuracy on different traces. (d) shows the accuracy after different number of iterations. (e) shows the average decoding time per iteration under different memory usage.



(a) RE (SrcIP) (b) RE (5-tuple) (c) RE vs. Trace (SrcIP)

Fig. 12. Accuracy of entropy estimation. (a) and (b) show the relative error of entropies estimated by MRAC and HeavyCache under different memory sizes. (c) shows the relative error of experiments results on different traces. Note that the Y-axises represent $log_{10}(RE)$.

frequency distribution, which is very accurate.

VII. Conclusion

Nowadays, it is an important and challenging issue to do data mining on high speed data streams. In this paper, we propose a novel data structure, namely HeavyCache, targeting at accurately answering a spectrum of queries. We show how to use HeavyCache to process six typical tasks: frequency estimation, heavy hitter detection, heavy change detection, frequency distribution estimation, entropy estimation, and cardinality estimation. We conduct extensive experiments to compare the performance of HeavyCache with the state-of-the-art solutions side by side. Our results show that our HeavyCache outperforms the state-of-the-art algorithms up to 113 times in terms of accuracy and up to 37 times in terms of speed. We believe that HeavyCache can be used to process many more data mining tasks.

References

[1] Bob Hash. http://burtleburtle.net/bob/hash/evahash.html.

[2] HeavyCache. https://github.com/HeavyCache/HeavyCache.

[3] The CAIDA Anonymized Internet Traces 2016. http://www.caida.org/data/overview/.

[4] S. Agarwal, B. Mozafari, A. Panda, H. Milner, S. Madden, and I. Stoica. Blinkdb: queries with bounded errors and bounded response times on very large data. In *Proceedings of the 8th ACM European Conference on Computer Systems*, pages 29–42. ACM, 2013.

[5] N. Agrawal and A. Vulimiri. Low-latency analytics on colossal data streams with summarystore. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 647–664. ACM, 2017.

[6] V. Braverman and R. Ostrovsky. Generalizing the layering method of indyk and woodruff: Recursive sketches for frequency-based vectors on streams. In *Approximation, Randomization, and Combinatorial Optimization. Algorithms and Techniques*, pages 58–70. Springer, 2013.

[7] M. Charikar, K. Chen, and M. Farach-Colton. Finding frequent items in data streams. *Automata, languages and programming*, pages 784–784, 2002.

[8] C.-H. Cheng, A. W. Fu, and Y. Zhang. Entropy-based subspace clustering for mining numerical data. In *Proceedings of the fifth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 84–93. ACM, 1999.

[9] G. Cormode. Sketch techniques for approximate query processing. *Foundations and Trends in Databases. NOW publishers*, 2011.

[10] G. Cormode, F. Korn, S. Muthukrishnan, and D. Srivastava. -finding hierarchical heavy hitters in data streams. In *Proceedings 2003 VLDB Conference*, pages 464–475. Elsevier, 2003.

[11] G. Cormode and S. Muthukrishnan. An improved data stream summary: the count-min sketch and its applications. *Journal of Algorithms*, 55(1):58–75, 2005.

[12] G. Cormode and S. Muthukrishnan. Summarizing and mining skewed data streams. In *Proceedings of the 2005 SIAM International Conference on Data Mining*, pages 44–55. SIAM, 2005.

[13] H. Cui, K. Keeton, I. Roy, K. Viswanathan, and G. R. Ganger. Using data transformations for low-latency time series analysis. In *Proceedings of the Sixth ACM Symposium on Cloud Computing*, pages 395–407. ACM, 2015.

[14] E. D. Demaine, A. López-Ortiz, and J. I. Munro. Frequency estimation of internet packet streams with limited space. In *European Symposium on Algorithms*, pages 348–360. Springer, 2002.

[15] N. Duffield, C. Lund, and M. Thorup. Estimating flow distributions from sampled flow statistics. In *Proceedings of the 2003 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 325–336. ACM, 2003.

[16] C. Estan and G. Varghese. New directions in traffic measurement and accounting: Focusing on the elephants, ignoring the mice. *ACM Transactions on Computer Systems (TOCS)*, 21(3):270–313, 2003.

[17] P. Flajolet and G. N. Martin. Probabilistic counting algorithms for data base applications. *Journal of computer and system sciences*, 31(2):182–209, 1985.

[18] A. Goyal, H. Daumé III, and G. Cormode. Sketch algorithms for estimating point queries in nlp. In *Proceedings of the 2012 Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning*, pages 1093–1103. Association for Computational Linguistics, 2012.

[19] A. Goyal, H. Daumé III, and G. Cormode. Sketch algorithms for estimating point queries in nlp. In *Proceedings of the 2012 Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning*, pages 1093–1103. Association for Computational Linguistics, 2012.

[20] Ş. Gündüz and M. T. Özsu. A web page prediction model based on click-stream tree representation of user behavior. In *Proceedings of the ninth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 535–540. ACM, 2003.

[21] R. Heider. Troubleshooting cfm 56-3 engines for the boeing 737 using cbr and data-mining. In *European Workshop on Advances in Case-Based Reasoning*, pages 512–518. Springer, 1996.

[22] 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 *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, pages 113–126. ACM, 2017.

[23] E. Krishnamurthy, S. Sen, and Y. Zhang. Sketchbased change detection: Methods, evaluation, and applications. In *In ACM SIGCOMM Internet Measurement Conference*. Citeseer, 2003.

[24] A. Kumar, M. Sung, J. J. Xu, and J. Wang. Data streaming algorithms for efficient and accurate estimation of flow size distribution. In *Proc. ACM SIGMETRICS*, pages 177–188, 2004.

[25] A. Kumar, M. Sung, J. J. Xu, and E. W. Zegura. A data streaming algorithm for estimating subpopulation flow size distribution. *ACM SIGMETRICS Performance Evaluation Review*, 33(1):61–72, 2005.

[26] A. Lall, V. Sekar, M. Ogihara, J. Xu, and H. Zhang. Data streaming

[35] R. Schweller, A. Gupta, E. Parsons, and Y. Chen. Reversible sketches for efficient and accurate change detection over network data streams. In *Proceedings of the 4th ACM SIGCOMM conference on Internet measurement*, pages 207–212, 2004.

algorithms for estimating entropy of network traffic. In *Proc. ACM SIGMETRICS*, pages 145–156, 2006.

[27] S. Li, L. Da Xu, and X. Wang. Compressed sensing signal and data acquisition in wireless sensor networks and internet of things. *IEEE Transactions on Industrial Informatics*, 9(4):2177–2186, 2013.

[28] T. Li, S. Chen, and Y. Ling. Per-flow traffic measurement through randomized counter sharing. *IEEE/ACM Transactions on Networking (TON)*, 20(5):1622–1634, 2012.

[29] H. Liu, Y. Sun, and M. S. Kim. Fine-grained ddos detection scheme based on bidirectional count sketch. In *Computer Communications and Networks (ICCCN), 2011 Proceedings of 20th International Conference on*, pages 1–6. IEEE, 2011.

[30] Z. Liu, A. Manousis, G. Vorsanger, V. Sekar, and V. Braverman. One sketch to rule them all: Rethinking network flow monitoring with univmon. In *Proceedings of the 2016 conference on ACM SIGCOMM 2016 Conference*, pages 101–114. ACM, 2016.

[31] N. Manerikar and T. Palpanas. Frequent items in streaming data: An experimental evaluation of the state-of-the-art. *Data & Knowledge Engineering*, 68(4):415–430, 2009.

[32] A. Metwally, D. Agrawal, and A. El Abbadi. Efficient computation of frequent and top-k elements in data streams. In *Proc. Springer ICDT*, 2005.

[33] A. Ramachandran, S. Seetharaman, N. Feamster, and V. Vazirani. Fast monitoring of traffic subpopulations. In *Proceedings of the 8th ACM SIGCOMM conference on Internet measurement*, pages 257–270. ACM, 2008.

[34] P. Roy, A. Khan, and G. Alonso. Augmented sketch: Faster and more accurate stream processing. In *Proc. SIGMOD*, 2016.

[36] T. Skeie, S. Johannessen, and O. Holmeide. Timeliness of real-time ip communication in switched industrial ethernet networks. *IEEE Transactions on Industrial Informatics*, 2(1):25–39, 2006.

[37] H. Sun, X. Wang, R. Buyya, and J. Su. Cloudeyes: Cloud-based malware detection with reversible sketch for resource-constrained internet of things (iot) devices. *Software: Practice and Experience*, 47(3):421–441, 2017.

[38] M. Wang, B. Li, and Z. Li. sflow: Towards resource-efficient and agile service federation in service overlay networks. In *Distributed Computing Systems, 2004. Proceedings. 24th International Conference on*, pages 628–635. IEEE, 2004.

[39] K.-Y. Whang, B. T. Vander-Zanden, and H. M. Taylor. A linear-time probabilistic counting algorithm for database applications. *ACM Transactions on Database Systems (TODS)*, 15(2):208–229, 1990.

[40] T. Yang, Y. Zhou, H. Jin, S. Chen, and X. Li. Pyramid sketch: a sketch framework for frequency estimation of data streams. *Proceedings of the VLDB Endowment*, 10(11):1442–1453, 2017.

[41] M. Yu, L. Jose, and R. Miao. Software defined traffic measurement with opensketch. In *NSDI*, volume 13, pages 29–42, 2013.