HeavyGuardian: Keep and Guard Hot Items in Data Streams

ABSTRACT

Data stream processing is a fundamental issue in many fields, such as databases, data mining, network traffic analysis. There are five typical tasks in data stream processing: frequency estimation, heavy hitters detection, heavy changes detection, frequency distribution estimation, and entropy estimation. Different algorithms are proposed for different tasks, but they seldom achieve high accuracy and high speed at the same time. To address this issue, In this paper, we propose a novel data structure named HeavyGuardian. The key idea is to intelligently keep and guard the information of hot items while approximately record the frequencies of cold items. We use HeavyGuardian to process the above five typical tasks. Extensive experimental results show that HeavyGuardian achieves both much higher accuracy and higher speed than the state-of-the-art solutions for each of the five typical tasks. The source code of HeavyGuardian and other related algorithms are available at GitHub [1].

1. INTRODUCTION

1.1 Background and Motivation

Approximate stream processing has been a hot issue in recent years. There are five typical stream processing tasks: frequency estimation, heavy hitter detection, heavy change detection, frequency distribution estimation, and entropy estimation. `Frequency estimation` is to estimate the number of appearances of any item in a data stream. `Heavy hitter detection` is to find those items whose frequencies are larger than a predefined threshold. `Heavy change detection` is to find those items whose frequencies change drastically in two adjacent time windows. `Frequency distribution estimation` is to estimate the number of distinct items whose frequencies are equal to any given value. `Entropy estimation` is to estimate the entropy of a data stream in real time or periodically. The above stream processing tasks and many other stream processing tasks, such as Super-Spreader [2], DDoS victims [3], top-k frequent items [4, 5], hierarchical heavy hitters [6, 7] *etc.*, care more about hot items. In practice, most items are cold while only a very few items are hot [8–10]. Accurately recording the information of massive cold items wastes much memory, and could incur non-trivial error to the estimation of hot items when the memory is tight. To achieve efficiency and effectiveness, one elegant solution is *to use a compact data structure to keep and guard the information (item ID and frequency) of hot items and efficiently record only the frequencies of cold items.* The key challenge is to judge whether the incoming item is hot or cold in real time. This is difficult because all hot items are initially cold.

Existing designs of the above data structure use the strategy `record-all-evict-code`. The *key idea* of this strategy is to first record frequencies of all items, and then evict cold items. Two notable algorithms using this strategy are Space-Saving [5] and the Augmented sketch [9]. Space-Saving [5] records each incoming item in a data structure named `Stream-Summary`. For any new incoming item e, the coldest item with frequency $\hat{f}_{min}$ in Stream-Summary will be replaced by e. To make such replacement reasonable, it sets the initial frequency of e to $\hat{f}_{min} + 1$. Consequently, *many later-incoming cold items are significantly overestimated, and seem to be hotter than some hot items.* Assume that there are k heavy hitters. To reduce such error caused by cold items, Space-Saving has to assign much more memory for Stream-Summary, *i.e.*, keeps m (m is much larger than k) items in the Stream-Summary, and reports the k hottest items among those m items, so as to reduce error.

The Augmented sketch [9] is a two-stage data structure: the first stage is a small array which sequentially stores δ hot items, and the second is a sketch which stores all item frequencies. Because each incoming item needs to check the δ hot items one by one, the authors recommended storing only $\delta = 32$ hot items at the first stage to guarantee the processing speed. However, such a strategy provides limited help for streaming tasks, as most tasks often need to report thousands of or more hot items from millions of distinct items. The processing speed of the Augmented sketch is slow because the first stage is similar to a cache, requiring frequent communication between the two layers.

In sum, existing algorithms using `record-all-evict-cold` have the following two shortcomings: 1) They are not memory efficient: Space-Saving needs plenty of additional memory to store those items that are not hot but not evicted, and the second stage of Augmented sketch needs to use plenty of large counters to store frequencies of all items. 2) The information of hot items is not well recorded. The recorded frequencies of hot items in Space-Saving are much larger than the real frequencies; while the Augmented sketch can only accurately store a very few hot items in the first stage.

1.2 Our Proposed Solution

In this paper, we propose a new data structure called HeavyGuardian. The strategy of HeavyGuardian is called *record-and-guard-hot*. It intelligently separates hot items from cold items, and keeps and guards the information of hot items, while using small counters to record only the frequencies of cold items.

Next we briefly present the principle of HeavyGuardian. The basic version uses a very small hash table consisting of buckets. Each bucket is mapped by many distinct items, but stores only at most λ_h (*e.g.*, $\lambda_h = 8$) items. In each bucket, in addition to the λ_h items, we can use λ_l (*e.g.*, $\lambda_l = 64$) small counters to handle cold items. When a new item is mapped to a '`full`' bucket, we use a key technique called *exponential decay*: decreasing the frequency of `the weakest guardian` (the item with the smallest frequency in that bucket) by 1 with a probability b^{-C}, where b is a constant number (*e.g.*, $b = 1.08$) and C is the frequency of the weakest guardian. If it is decreased to 0, the weakest guardian will be replaced by the new item. The hottest item in the bucket will usually be the `king` (the item with the largest frequency), and guarded by other $\lambda_h - 1$ items. Cold items can hardly find room to store, being mapped into a small counter in that bucket. Therefore, HeavyGuardian can achieve high accuracy for hot items with small memory and fast speed. HeavyGuardian can be used for many different tasks, as it accurately records the information of hot items, and also approximately records the frequencies of cold items. To demonstrate it, we show how to process the above mentioned five typical stream processing tasks using HeavyGuardian in Section 5.

Main experimental results: We present main experimental results in Table 1. Typically, the results show that for heavy hitter detection, our HeavyGuardian achieves $887 \sim 5.26 * 10^9$ times smaller error than the state-of-the-art Space-Saving (See Figure 14(c)), and can use only 0.005 bit for each distinct items (See Figure 13(a)). Further, the speed of HeavyGuardian is up to 1.928 times higher than that of Space-Saving (See Figure 15(a)).

1.3 Main Contributions

1. We propose a novel data structure, named HeavyGuardian, which can intelligently keep and guard the information of hot items and approximately record only the frequencies of cold items in a data stream.

2. We derive the error bound of HeavyGuardian, and experimental results match the bound well.

3. We use HeavyGuardian to process five typical tasks for data stream processing. Extensive experimental results on real and synthetic datasets show that HeavyGuardian achieves both much higher accuracy and higher processing speed at the same time than the state-of-the-art.

Table 1: Main experimental results.

Task	Accuracy improvement	Speed improvement
Frequency estimation	$\times 1.68 \sim 100.15$	$\times 1.014 \sim 2.167$
Heavy hitter detection	$\times 887 \sim 5.26 * 10^9$	$\times 1.419 \sim 1.928$
Heavy change detection	$\times 137.38 \sim 1448.35$	$\times 1.419 \sim 1.928$

2. BACKGROUND

Our HeavyGuardian algorithm can support many data stream processing tasks, including frequency estimation, heavy hitter detection, heavy change detection, real-time frequency distribution estimation, and entropy estimation. In this section, we survey existing typical algorithms for each of these tasks.

2.1 Frequency Estimation

Given a data stream and an item, frequency estimation is to estimate the times of appearances of the item in a data stream. A large hash table can be used to store items and their frequencies, but it can hardly catch up with the high speed of data streams. Sketches can achieve memory efficiency and fast speed at the cost of introducing small error, and therefore gain wide acceptance recently [8–10]. Typical sketches include Count sketches [11], Count-min (CM) sketches [12], CU sketches [13], Augmented sketch framework [9], and Pyramid sketch framework [10]. Count sketches, CM sketches, and CU sketches use equal-sized counters to record frequencies, while the size of counters needs to be large enough to accomodate the largest frequency since hot items are often more important than cold items. Moreover, as cold items are much more than hot items, many counters store only a small number, and thus the higher bits of these counters are wasted. As a result, when the memory size is tight, the accuracy of these sketches is poor. Augmented sketch framework uses a filter to accurately store frequencies of only a few hot items (*e.g.*, 32 hot items), thus the overall accuracy is not improved much. Pyramid sketch framework, the state-of-the-art algorithm, can automatically enlarge the size of counters according to the current frequency of the incoming item, and thus achieves much higher accuracy than other algorithms. However, hot items require quite a few memory accesses, and thus the time complexity in the worst case is poor. Our goal is to achieve much higher accuracy and faster speed both in average and in the worst case.

2.2 Heavy Hitter Detection

Given a data stream which has a total of N items, heavy hitter detection is to find items whose frequency is larger than θN, where θ is a pre-defined threshold defined by the user. Heavy hitter detection is a critical task in data mining [14–16], information retrieval [17], network measurement and management [18], security [19], and more [20].

Existing algorithms for heavy hitter detection can be divided into two categories: sketch based algorithms and counter based algorithms. Sketch based algorithms use a sketch (*e.g.*, the CM sketch [12]) to approximately record frequencies of all items, and use a min-heap to maintain the largest k items. For each incoming item e_i, it first inserts e_i into the sketch, and then gets an estimated frequency $\hat{f}_i$ from the sketch. If $\hat{f}_i > \hat{f}_{min}$, where $\hat{f}_{min}$ is the frequency of the item in the root node of the min-heap, then it replaces the item in the root node with e_i, and then adjusts the min-heap to make it ordered. However, sketch-based algorithms require much memory usage to record all item frequencies. Therefore, when the memory space is tight, the accuracy decreases quickly. Counter-based algorithms include Space-Saving [5], Frequent [21, 22], and Lossy counting [23], with Space-Saving the most widely used algorithm among them. These algorithms are similar to Space-Saving, while the Space-Saving algorithm is presented in Section 1.1.

2.3 Heavy Change Detection

Given two consecutive time windows in a data stream, if the frequency difference of an item e_i is larger than θD, then e_i is a heavy change. Here θ is a user-defined threshold, and D is the sum of the frequency difference of all items. Heavy change detection is also an important task in many big data scenarios, such as web search engines [24], anomalies detection [25, 26], time series forecasting and outlier analysis [27–30], *etc.* For example, in web search engines, finding heavy changes enables the server to find trends in user interests. In traffic anomalies detection, it is important to detect the significant changes in traffic pattern.

There are several well-known algorithms for heavy change detection. Based on the CM sketch, the *k-ary* sketch [25] can achieve high accuracy for heavy change detection when memory space is large. It uses two sketches to approximately record the frequency of all items in the two adjacent time windows, and computes the differences of each counter between the two sketches. By querying each item's difference, it reports those items whose difference is larger than the threshold value. To to this, the algorithm requires to record all item IDs, which is time-consuming and space-consuming. To address this problem, the *reversible* sketch [26] is proposed based on the *k-ary* sketch. It can infer the item ID within a time complexity sub-linear to the size of data stream by dividing the item ID into q parts (q is a pre-defined constant number, *e.g.*, 4). Such a high time complexity makes it unable to catch up with the high speed of data streams.

2.4 Real-time Frequency Distribution & Real-time Entropy

Frequency distribution estimation means that given an arbitrary frequency f, it is required to estimate how many items whose frequency is f. Frequency distribution estimation is important in database query optimization [31–33], structural anomalies detection [34], data mining [35], and network measurement and monitoring [36, 37], and has been extensively studied in the literature. Further, *real-time* frequency distribution estimation can support more powerful functions. For example, IP service providers can infer the usage pattern of the network by the estimated frequency distribution, which is important for adjusting the strategies for their service. If the frequency distribution is real-time, then IP service providers can adjust their strategies immediately, leading to better services for users. Notable algorithms for frequency distribution estimation includes MRAC [37], FlowRadar [38], *etc.*. However, these algorithms cannot support *real-time* frequency distribution.

Next, we focus on entropy estimation. Entropy refers to the uncertainty of a data stream. Formally, entropy is defined as

$$entropy = \sum_{e} \frac{f_e}{N} log_2 \frac{f_e}{N}$$

where f_e is the frequency of e and N is the total number of items. Entropy estimation is also important in network measurement and monitoring [39–42]. For example, changes of entropy can indicate anomalous incidents in the traffic, which can be used for anomaly detection. *Real-time* entropy estimation can support better performance in anomaly detection, because in this way, IP service providers can deal with the anomalous incidents in real time, providing better services for users. The most notable algorithm is proposed by Lall *et al.* [43], which uses sampling and simple mathematical derivation to estimate the entropy. It achieves high accuracy, high memory efficiency, and high processing speed at the same time. However, it cannot support real-time entropy estimation. FlowRadar [38] can also be used for entropy estimation, but the performance is much poorer than the previous one.

To the best of our knowledge, existing algorithms estimate frequency distribution or entropy periodically but not in real time. This is the first effort for real time implementation – quickly updating frequency distribution or entropy for each incoming item.

3. THE HEAVYGUARDIAN ALGORITHM

The key design of HeavyGuardian is to store hot items and cold items differently: it keeps and guards frequencies of hot items precisely, and stores frequencies of cold items approximately. In this section, we first present the basic data structure and algorithms of HeavyGuardian, and present some optimizations to further improve the performance of HeavyGuardian. Table 2 summarizes all symbols and abbreviations used in this paper.

Table 2: Symbols and abbreviations in this paper.

Symbol	Description
e_i	The i^{th} item
f_i	The real frequency of e_i
$\hat{f_i}$	The estimated frequency of e_i
$\hat{f}_{min}$	The minimum frequency in the Stream-Summary or the min-heap
e_{min}	The item with the minimum frequency in the Stream-Summary or the min-heap
d	# arrays in the sketch
A	The hash table in HeavyGuardian
$A[i]$	The i^{th} bucket in $A[i]$
$A[i][j]_h$	The j^{th} cell in the heavy part of $A[i]$
$A[i][j]_h.ID$	The ID field in $A[i][j]_h$
$A[i][j]_h.FP$	The fingerprint field in $A[i][j]_h$
$A[i][j]_h.C$	The counter field in $A[j][t]_h$
λ_h	# cells in the heavy part of each bucket
λ_l	# counters in the light part of each bucket
$h(.), g(.)$	Two hash functions associated with the A
$\mathcal{F}_{e_i}$	The fingerprint of e_i
b	The exponential base
N	The number of items
M	The number of distinct items
w	The width of sketches and HeavyGuardian

3.1 HeavyGuardian Basics

3.1.1 Data Structure

The data structure of the basic version of HeavyGuardian is a hash table, with each bucket storing multiple key-value (KV) pairs and several small counters. As shown in Figure 1, a hash table A associated with a hash function $h(.)$ consists of w buckets $A[1 \cdots w]$. Each bucket has two parts: a `heavy part` to precisely store frequencies of hot items, and a `light part` to store frequencies of cold items. For the heavy part of each bucket, there are λ_h ($\lambda_h > 0$) cells, and each cell is used to store one KV pair $< ID, count >$. The key is the item ID, while the value is its estimated frequency (number of appearances) in the data stream. We use $A[i][j]_h$ ($1 \leqslant i \leqslant w, 1 \leqslant j \leqslant \lambda_h$) to denote the j^{th} cell in the heavy part of the i^{th} bucket, and use $A[i][j]_h.ID$ and $A[i][j]_h.C$ to denote the ID field and the count field in the cell $A[i][j]_h$, respectively. Among all KV pairs within the heavy part of one bucket, we call the hottest item (*i.e.*, the item with the largest frequency) the `king`, call other items `guardians`, and call the guardian with the smallest frequency the `weakest guardian`. For the light part of each bucket, there are λ_l (λ_l can be 0) counters, to store frequencies of cold items. We use $A[i][j]_l$ ($1 \leqslant j \leqslant \lambda_l$) to denote the j^{th} counter in the light part of the i^{th} bucket. Since counters in the light part are tailored for cold items, the counter size can be very small (*e.g.*, 4 bits), achieving high memory efficiency.

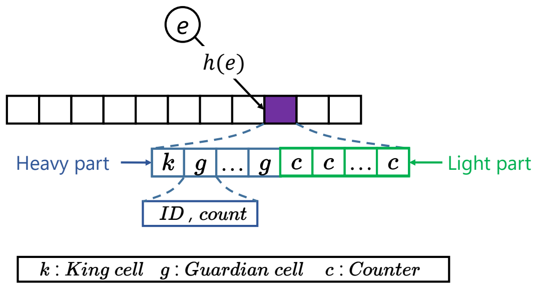


Figure 1: The data structure of HeavyGuardian.

3.1.2 Operations

Initialization: All fields in every bucket is set to 0.

Insertion: For each incoming item e, it first computes the hash function $h(e)$ ($1 \leqslant h(e) \leqslant w$) to map e to bucket $A[h(e)]$. We call $A[h(e)]$ the `mapped bucket`. Given an incoming item e, we first try to insert e into the heavy part. If failed, then we insert it into the light part.

1) Heavy Part insertion: when inserting an item e into the heavy part, there are three cases as follows.

Case 1: e is in one cell in the heavy part of $A[h(e)]$ (being a king or a guardian). HeavyGuardian just increments the corresponding frequency (the count field) in the cell by 1.

Case 2: e is not in the heavy part of $A[h(e)]$, and there are still empty cells. It inserts e into an empty cell, *i.e.*, sets the ID field to e and sets the count field to 1.

Case 3: e is not in any cell in the heavy part of $A[h(e)]$, and there is no empty cell. We propose a novel technique named `Exponential Decay`: it *decays* (decrements) the count field of the weakest guardian by 1 with probability $\mathcal{P} = b^{-C}$, where b is a pre-defined constant number (*e.g.*, $b = 1.08$), and C is the value of the Count field of the weakest guardian. After decay, if the count field becomes 0, it replaces the ID field of the weakest guardian with e, and sets the count field to 1; otherwise, it inserts e into the light part.

2) Light part insertion: To insert an item e to the light part, it first computes another hash function $g(e)$ ($1 \leqslant g(e) \leqslant \lambda_l$), and increments counter $A[h(e)][g(e)]_l$ in the light part of the bucket by 1.

Query: To query an item e, first, it checks the heavy part in bucket $A[h(e)]$, if e matches a cell in the bucket, it reports the corresponding count field; if e matches no cell, it reports counter $A[h(e)][g(e)]_l$ in the light part.

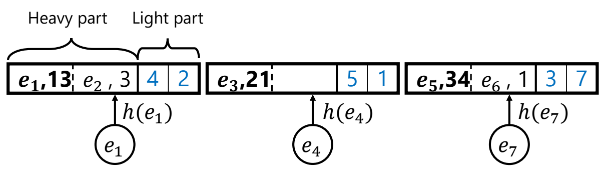


Figure 2: Examples of insertion of HeavyGuardian.

Example: As shown in Figure 2, we set $w = 3$, $\lambda_h = 2$, $\lambda_l = 2$, and $b = 1.08$. It means one bucket has 2 cells in the heavy part, and 2 counters in the light part. When e_1 arrives, it is mapped to bucket $A[1]$, and e_1 is the king in the hot part. Therefore, it increments the corresponding count field ($A[1][1]_h.C$) from 13 to 14. When e_4 arrives, it is mapped to bucket $A[2]$, and e_4 is not in the heavy part of the bucket but there is an empty cell ($A[2][2]_h$). Therefore, it sets the ID field of the empty cell ($A[2][2]_h.ID$) to e_4, and sets the count field ($A[2][2]_h.C$) to 1. When e_7 arrives, it is mapped to bucket $A[3]$. e_7 is not in the heavy part of the bucket, and there is no empty cell. Therefore, it decays the count field of the weakest guardian ($A[3][2]_h.C$) with a probability $1.08^{-1} \approx 0.926$. If the count field is decayed to 0, then it replaces e_6 with e_7, and sets the count field to 1. Otherwise, e_7 is inserted into the light part. Assume that e_7 is mapped to the counter $A[3][2]_l$ by computing $g(e_7) = 2$, and thus $A[3][2]_l$ is incremented from 7 to 8.

Analysis: In the operation of *exponential decay*, when C (the count field of the weakest guardian) is small, $\mathcal{P}$ is close to 1 (*e.g.*, $1.08^{-1} \approx 0.926$, $1.08^{-2} \approx 0.857$), and thus cold items will be evicted from the heavy part soon. When C is large, $\mathcal{P}$ is close to 0 ($1.08^{-1000} \approx 3.77 * 10^{-34}$), and thus not only hot items can hardly be evicted from the heavy part, but also the stored frequencies of hot items can hardly be decreased. As more and more items arrive, the frequencies of the king and guardians vary, and thus the king could become a guardian and a guardian could also become the king. Obviously, hot items seldom become the weakest guardian, and thus its frequency is almost exactly correct. While for each incoming cold item, it will be inserted into the light part in most cases; even if it accidentally becomes the weakest guardian, it will soon be evicted by other new incoming items with a high probability. In this way, the heavy part of each bucket in HeavyGuardian *seldom records cold items, but records and guards the frequencies of hot items*. With regards to the processing speed, 1) processing one hot item only needs to check cells sequentially in the heavy part, and this is fast because these cells can fit into a cache line; 2) one cold item needs one additional access of the counter in the light part, this is also fast. processing cold items is fast as well, since it only needs one additional access of the light part.

3.2 Optimization: using Fingerprints

When the size of item ID is large, we propose to use fingerprints[1] to replace IDs in HeavyGuardian, further improving the memory efficiency of HeavyGuardian. In this way, the memory usage of HeavyGuardian is independent to the size of item ID. Despite that some items could share the same fingerprint due to hash collisions, we argue that it leads to little impact to the performance. Given a bucket, the probability that a certain hot item suffers from fingerprint collisions is

$$Pr\{fingerprint\ collision\} = 1 - (1 - 2^{-l})^{\frac{M}{w}} \tag{1}$$

where l is the fingerprint size (in bit), M is the number of distinct items, and w is the width of the hash table in HeavyGuardian. As shown in Table 3, we set $M = 1,000,000$. When the fingerprint size is larger than 16, the probability of fingerprint collisions is negligible.

Table 3: The probability of fingerprint collisions.

Probability	$w = 10000$	$w = 20000$	$w = 50000$
$l = 8$	0.324	0.178	0.075
$l = 16$	1.52×10^{-3}	7.63×10^{-4}	3.05×10^{-4}
$l = 32$	2.33×10^{-8}	1.16×10^{-8}	4.66×10^{-9}

4. MATHEMATICAL ANALYSIS

In this section, we first prove there is no over-estimation error for items recorded in the heavy part of HeavyGuardian, and then derive the formula of the error bound.

4.1 Proof of no Over-estimation Error

THEOREM 1. *In the basic version of HeavyGuardian, given an arbitrary item e, the estimated frequency of e recorded in the heavy part $\hat{f}_e$ of HeavyGuardian is no larger than its real frequency f_e.*

PROOF. We prove that the theorem holds at any point of time by using mathematical induction. Given any item e, it is mapped to one bucket, and there are some other items mapped to this bucket. For those items that are not mapped to this bucket, they have no impact to the estimated frequency of e, and thus we omit them.

Initially, e is not in the bucket, so the theorem holds. At any point of time, for each incoming item x that mapped to this bucket, there are four cases as follows.

Case 1: $x \neq e$ and e is not in the mapped bucket. Then after insertion, e is still not in the bucket, so the theorem holds.

Case 2: $x \neq e$ but e is in the mapped bucket. Then the estimated frequency of e is possible be decayed, and thus the estimated frequency is still no larger than the real frequency of e. The theorem holds.

Case 3: $x = e$ but e is not in the mapped bucket. After inserting e, if e is not recorded in the bucket, then clearly the theorem holds. If e is recorded, then the estimated frequency is 1, which is no larger than the real frequency of e, so the theorem holds.

Case 4: $x = e$ and e is in the mapped bucket. Then both the estimated frequency and the real frequency of e are incremented by 1, and the theorem still holds.

In sum, the theorem holds at any point of time. □

[1]The fingerprint of an item is a series of bits, and can be computed by a certain hash function.

4.2 The Error Bound of the Heavy Part of HeavyGuardian

According to literature [8–10], the item frequency in a data stream in practice often follows Zipfian distribution [44]. We present its definition as follows.

DEFINITION 4.1. *Assume that there is a data stream $\mathcal{S}$ including M distinct items: $e_1, e_2, \cdots, e_M$. Let f_i be the frequency of e_i, and suppose $f_1 \geqslant f_2 \geqslant \cdots \geqslant f_M$. Let N be the total number of items, and let α be the skewness of the stream (*e.g., $0.3 \sim 3.0$*). If*

$$f_i = \frac{N}{i^{\alpha}\zeta(\alpha)}$$

where

$$\zeta(\alpha) = \sum_{i=1}^{M} \frac{1}{i^{\alpha}}$$

$\mathcal{S}$ obeys the Zipfian distribution [44].

THEOREM 2. *Given a stream $\mathcal{S}$ with items $e_1, e_2, \cdots, e_M$, it obeys the Zipfian distribution. In the basic version of HeavyGuardian, we assume that the λ_h cells in the heavy part of each bucket belong to the hot items with the largest λ_h frequencies among all items mapped to the bucket. Let f_i be the real frequency of e_i, and let $\hat{f}_i$ be the estimated frequency of e_i. Given a small positive number ϵ and a hot item e_i, we have*

$$Pr\{f_i - \hat{f}_i \leqslant \epsilon N\} \geqslant \frac{1}{2\epsilon N}\left[f_i - \sqrt{f_i^2 - \frac{4P_{weak}E(V)}{b-1}}\right] \tag{2}$$

where

$$P_{weak} = \binom{i-1}{\lambda_h - 1}\left(\frac{1}{w}\right)^{\lambda_h - 1}\left(\frac{w-1}{w}\right)^{i-\lambda_h} \tag{3}$$

and

$$E(V) = \frac{1}{w}\sum_{j=i+1}^{M} f_j \quad and \quad f_j = \frac{N}{j^{\alpha}\zeta(\alpha)} \tag{4}$$

PROOF. Because e_i is a hot item, we assume that e_i is in the mapped bucket, and for each incoming e_i, its estimated frequency $\hat{f}_i$ is incremented by 1. After inserting all items, the estimated frequency of e_i is

$$\hat{f}_i = f_i - X_i \tag{5}$$

where X_i is the number of *exponential decays* that are successfully applied to e_i.

Exponential decays are applied to $\hat{f}_i$ *iff* e_i is the weakest guardian and the incoming item is not in this bucket. In other words, there are $\lambda_h - 1$ hotter items (items with a frequency larger than e_i) mapped to this bucket. Therefore, the probability that e_i is the weakest guardian is

$$P_{weak} = \binom{i-1}{\lambda_h - 1}\left(\frac{1}{w}\right)^{\lambda_h - 1}\left(\frac{w-1}{w}\right)^{i-\lambda_h} \tag{6}$$

Note that this probability obeys binomial distribution $B(i-1, \frac{1}{w})$, and it can be approximated by the Poisson distribution $P(\frac{i-1}{w})$. Therefore, we have

$$P_{weak} = e^{-\frac{i-1}{w}} \times \frac{[\frac{i-1}{w}]^{\lambda_h - 1}}{(\lambda_h - 1)!} \tag{7}$$

Let V be the number of items that perform exponential decays on e_i. The expectation of V is

$$E(V) = \frac{1}{w}\sum_{j=i+1}^{M} f_j \tag{8}$$

where $f_j = \frac{N}{j^{\alpha}\zeta(\alpha)}$ as the Zipfian distribution shows. Then we get

$$\begin{aligned} E(X_i) &= P_{weak} \cdot \frac{E(V)}{E(\hat{f}_i)} \sum_{j=1}^{E(\hat{f}_i)} b^{-j} \\ &= P_{weak} \cdot \frac{E(V)}{E(\hat{f}_i)} \frac{b^{-1}[1-b^{-E(\hat{f}_i)}]}{1-b^{-1}} \\ &= P_{weak} \cdot \frac{E(V)}{E(\hat{f}_i)(b-1)} \end{aligned} \tag{9}$$

here we assume that the exponential decays occur randomly as the estimated frequency of e_i grows from 1 to f_i, and $b^{-f_i} \to 0$. Therefore, we have

$$\begin{aligned} E(\hat{f}_i) &= f_i - E(X_i) \\ &= f_i - \frac{P_{weak}E(V)}{E(\hat{f}_i)(b-1)} \end{aligned} \tag{10}$$

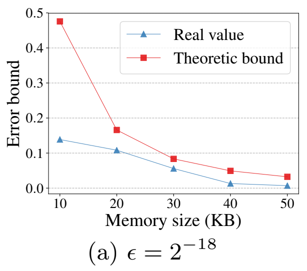
then we solve this equation and get $E(\hat{f}_i)$

$$\begin{aligned} E(\hat{f}_i) &= f_i - \frac{P_{weak}E(V)}{E(\hat{f}_i)(b-1)} \\ E(\hat{f}_i)^2 - f_iE(\hat{f}_i) &+ \frac{P_{weak}E(V)}{b-1} = 0 \\ E(\hat{f}_i) &= \frac{f_i + \sqrt{f_i^2 - \frac{4P_{weak}E(V)}{b-1}}}{2} \end{aligned} \tag{11}$$

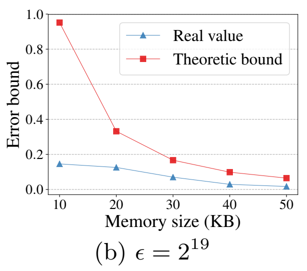
and based on Markov inequality, we have

$$\begin{aligned} Pr\{f_i - \hat{f}_i \leqslant \epsilon N\} &\leqslant \frac{E(f_i - \hat{f}_i)}{\epsilon N} \\ &= \left[f_i - \frac{f_i + \sqrt{f_i^2 - \frac{4P_{weak}E(V)}{b-1}}}{2} \right] \cdot \frac{1}{\epsilon N} \\ &= \frac{1}{2\epsilon N}\left[f_i - \sqrt{f_i^2 - \frac{4P_{weak}E(V)}{b-1}} \right] \end{aligned} \tag{12}$$

Therefore, the theorem holds. $\square$



(a) $\epsilon = 2^{-18}$



(b) $\epsilon = 2^{19}$

Figure 3: Theoretical error bound and real value.

To validate the error bound derived in Theorem 2, we conduct experiments on synthetic Zipfian datasets. Here we set $\lambda_h = 2$, $b = 1.08$, $\epsilon = 2^{-18}$ and 2^{-19}, $N = 3.2 \times 10^7$, $M = 10^6$, and $\alpha = 0.6$. We vary the memory size from 10KB to 50KB, and w is computed based on the memory size. As shown in Figure 3(a) and Figure 3(b), the theoretical error bound is always larger than the experimental values, which confirms the correctness of our derived error bound.

5. HEAVYGUARDIAN DEPLOYMENT

The HeavyGuardian algorithm can support various kinds of data stream processing tasks. In this section, we will present how to deploy HeavyGuardian on the above mentioned five typical tasks.

5.1 Frequency Estimation

In each bucket of HeavyGuardian, the heavy part records frequencies of hot items, and the light part records frequencies of other items. Therefore, HeavyGuardian supports frequency estimation for any item. HeavyGuardian can intelligently record and guard frequencies of hot items in the heavy part, and use small counters to record frequencies of cold items in the light part. Therefore, when the memory size is tight, HeavyGuardian can still achieve high accuracy for frequency estimation.

5.2 Heavy Hitter Detection & Heavy Change Detection

The key design of our algorithm is to use HeavyGuardian to maintain frequencies of only hot items, and manage to store the item ID of only hot items. Because heavy hitter detection and heavy change detection focus on only hot items, we do not need the light parts to store frequencies of cold items. Therefore, we simply set $\lambda_l = 0$.

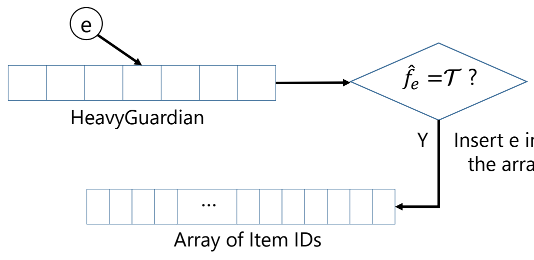


Figure 4: Heavy hitter detection using HeavyGuardian.

5.2.1 *Detecting Heavy Hitters*

Insertion: As shown in Figure 4, for each incoming item e, it first inserts e into HeavyGuardian. It then gets the estimated frequency of e: $\hat{f}_e$, and checks whether $\hat{f}_e$ is equal to the heavy hitter threshold $\mathcal{T}$. If $\hat{f}_e = \mathcal{T}$, it inserts the item ID into the array B.

Note that the condition is $\hat{f}_e = \mathcal{T}$ rather than $\hat{f}_e > \mathcal{T}$, the reason is as follows. HeavyGuardian stores the fingerprints and frequencies of hot items, and the frequency of each heavy hitter increases *one by one* when not considering fingerprint collisions. Therefore, in this way, we can record the IDs of heavy hitters only once. In the worst case, when a cold item and a heavy hitter have the same fingerprint and they are mapped into one bucket, only if the cold item arrives when the recorded frequency is just $\mathcal{T} - 1$, the cold item ID will be recorded. Such situation happens with a very probability.

The array B only stores the IDs of heavy hitters, and therefore the memory usage is small, and in addition, its memory usage can be dynamically allocated.

Query: After insertion, it traverses the array B. For each item ID e, it gets the estimated frequency $\hat{f}_e$ from HeavyGuardian. If $\hat{f}_e$ is larger than or equal to $\mathcal{T}$, it reports e as a heavy hitter.

5.2.2 *Detecting Heavy Changes*

The algorithm for heavy change detection is similar to that for heavy hitter detection. For two consecutive time windows in a data stream, we deploy one HeavyGuardian to each time window, and we can get heavy hitters for each time window based on the algorithm mentioned above. After getting heavy hitters for each time window, we compute the difference of frequencies for each heavy hitter, and if the difference is larger than or equal to the heavy change threshold, we report it as a heavy change.

5.2.3 *Analysis*

We make a brief analysis to show why HeavyGuardian achieves better performance than other existing algorithms. As mentioned in Section 2.2, sketch-based algorithms require large memory usage to store frequencies of all items. They suffer from low accuracy when memory is tight. Counter-based algorithm achieves better memory efficiency, but the estimated frequencies of many cold items are significantly over-estimated, and thus many cold items could be misreported to be heavy hitters or heavy changes. In a word, existing algorithms cannot handle cold items elegantly, and thus cannot achieve high accuracy. On the contrary, HeavyGuardian can intelligently record and guard frequencies of hot items, and can simultaneously reduce the harmful impact of cold items by preventing cold items from being recorded. Therefore, our proposed algorithm achieves much higher accuracy than existing algorithms for heavy hitter detection and heavy change detection. Further, because HeavyGuardianonly records the item if and only if the recorded frequency is equal to the threshold, the processing speed for heavy hitters and heavy changes is fast.

5.3 Real-time Frequency Distribution & Real-time Entropy

The key design of our proposed algorithm is to use HeavyGuardian to maintain frequencies of all items, and use an array of counters to record the distribution of frequencies. As shown in Figure 5, we use an auxiliary array $Dist$ of counters to maintain the frequency distribution. $Dist$ has y counters, and the i^{th} counter records the number of items whose frequency is i. For each incoming item e, we first insert e into HeavyGuardian, and gets the estimated frequency $\hat{f}_e$. Then we increment the $\hat{f}_e{}^{th}$ counter by 1, and decrement the $(\hat{f}_e - 1)^{th}$ counter by 1 if $\hat{f}_e > 1$. In this way, we can estimate the frequency distribution in real time. Further, by computing $\sum_{i=1}^{y} Dist[y] \cdot \frac{y}{N} log_2 \frac{y}{N}$, we can also get the estimated entropy in real time.

As mentioned in Section 2.4, all existing algorithms cannot support real-time frequency distribution estimation and entropy estimation. In contrast, by simply adding an auxiliary array, HeavyGuardian is able to maintain the real-time frequency distribution and entropy. Further, as HeavyGuardianachieve high accuracy, and the recorded frequencies of hot items increase one by one, our proposed algorithms achieve high accuracy for real-time frequency distribution and entropy.

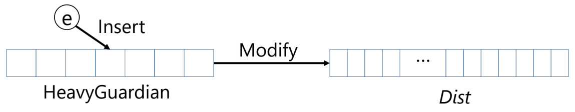


Figure 5: Real-time frequency distribution estimation and entropy estimation using HeavyGuardian.

6. EVALUATION

6.1 Experiment Setup

Dataset:

1) CAIDA: This dataset is from *CAIDA Anonymized Internet Trace 2016* [45], consisting of IP packets. Each item is identified by the source IP address and the destination IP address. The dataset contains 10M IP packets, belonging to around 4.2M distinct items.

2) Campus: This dataset consists of IP packets captured from the network of our campus. Each item is identified by the 5-tuple, *i.e.*, source IP address, destination IP address, source port, destination port, and protocol type. This dataset contains 10M packets, belonging to around 1M distinct items.

3) Web page: This dataset is built from a spidered collection of web HTML documents [46]. This dataset contains 10M items, belonging to around 0.4M distinct items.

4) Synthetic: We generated 10 different datasets according to Zipfian [44] distribution, with different skewness from 0.6 to 3.0. These datasets are generated by Web Polygraph [47]. The default skewness of this dataset is 0.6.

Implementation: HeavyGuardian and other related algorithms are implemented in C++. We tried several different hash functions in our experiments, and found that the performance keeps unchanged. Here we use the Bob hash that literature [48] recommends. All the programs are run on a server with dual 6-core CPUs (24 threads, Intel Xeon CPU E5-2620 @2 GHz) and 62 GB total system memory.

6.2 Metrics

Precision: Precision measures the ratio of the number of correctly reported answers to the number of reported answers. It is defined as

$$Precision = \frac{|\Omega \cup \Psi|}{|\Omega|}$$

where Ω is the set of reported answers, and Ψ is the set of correct answers. Precision is evaluated in all three tasks.

Recall: Recall measures the ratio of the number or correctly reported answers to the number of correct answers. It is defined as

$$Recall = \frac{|\Omega \cup \Psi|}{|\Psi|}$$

Recall is evaluated in all three tasks.

Average Relative Error (ARE): ARE measures the accuracy of the estimated frequency. It is defined as

$$ARE = \sum_{e_j \in \Omega} \frac{|\hat{f}_j - f_j|}{f_j}$$

where f_j is the real frequency of e_j, and $\hat{f}_j$ is the estimated frequency of e_j.

Average Absolute Error (AAE): In the experiments of heavy change detection, AAE measures the accuracy of the estimated frequency differences between two chunks of the data stream. It is defined as

$$AAE = \sum_{e_j \in \Omega} |\hat{f}_j - f_j| \quad or \quad \sum_{e_j \in \Omega} |\hat{d}_j - d_j|$$

where d_j and $\hat{d}_j$ are used only in heavy change detection, and d_j is the real frequency difference of e_j and $\hat{d}_j$ is the estimated frequency difference of e_j.

Throughput: Throughput measures the processing speed of the algorithm. We insert all items in the data stream, and record the time used for processing these items. The throughput is defined as $\frac{N}{T}$, where N is the number of items, and T is the time used. We use Million of insertions per second (Mps) to measure the throughput. Throughput is evaluated in all three tasks.

6.3 Experiments on System Parameters

In this section, we conduct experiments to evaluate the effects of system parameters for HeavyGuardian. We focus on the exponential base b, the number of guardians per bucket λ_h, and the fingerprint size l. Specifically, we only focus on the heavy part of each bucket, and omit the accuracy of items treated as cold items by HeavyGuardian. Further, we set the total memory size to 100KB, use CAIDA dataset in our experiments, and use AAE as the metric to evaluate the accuracy.

1) Effects of b. In this experiment, we set $\lambda_h = 2$ and do not use fingerprints to replace ID fields. Figure 6(a) shows how AAE changes as b increases from 1.02 to 1.6. The results show that when b changes from 1.02 to 1.18, AAE keeps unchanged. While as b is larger than 1.20, AAE increases. The optimal value of b is from 1.02 to 1.18, and we set $b = 1.08$ in our experiments.

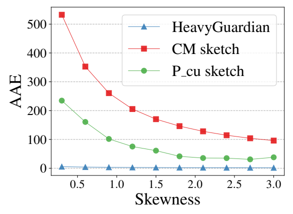
2) Effects of λ_h. In this experiment, we do not use fingerprints to replace ID fields, either. Figure 6(b) shows how AAE changes as λ_h increases from 2 to 32. The results show that as λ_h increases, it first decreases significantly, and when λ_h is between 8 to 20, AAE keeps at a low value. When λ_h becomes larger than 20, AAE increases slightly. Taking the speed into consideration, we set λ_h to 8.

3) Effects of l. Figure 6(c) shows how AAE changes as l changes from 4 to 30. The results show that when l increases, AAE first decreases significantly, and when l becomes larger than 15, AAE keeps at a low value. In order to put more cells in a cache line and reduce fingerprint collisions, we should not use long fingerprints. Therefore, we set $l = 16$.

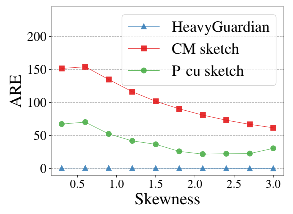
In terms of λ_l, we found that as λ_l increases, the accuracy of cold items increases, while the accuracy of hot items decreases. In order to make a trade-off between hot items and cold items, we set the memory size of the heavy part equal to that of the light part in each bucket. In our implementation, we set the counter size in the heavy part to 16 bits, and set the counter size in the light part to 4 bits. Therefore, we get $\lambda_l = 64$.

6.4 Experiments on Stream Processing Tasks

In this section, we apply HeavyGuardian to specific stream processing tasks, including frequency estimation, heavy hitter detection, heavy change detection, real-time frequency distribution estimation and entropy estimation. We conduct experiments to compare the performance of HeavyGuardian to other existing algorithms for all tasks.



(a) Measuring AAE



(b) Measuring ARE

Figure 9: Experiments on varying skewness.

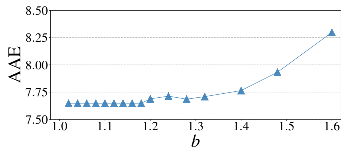
6.4.1 *Experiments on Frequency Estimation*

In this section, we conduct experiments on frequency estimation. We compare the performance of HeavyGuardian with the CM sketch [12] and the P_{cu} sketch (the CU sketch using the Pyramid sketch framework [10]). We do not conduct experiments for other sketches, such as the CU sketch [13] and the Augmented sketch framework [9], because [10] has shown that the performance of the P_{cu} sketch is much better than other sketches. Specifically, we use AAE and ARE as metrics to evaluate the accuracy, and use throughput to evaluate the speed.

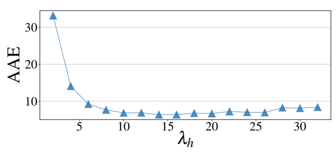
1) Measuring AAE, varying memory size. In this experiment, we vary the memory size from 100KB to 1000KB to show the change of AAE of all three algorithms. Figure 7(a), Figure 7(b), Figure 7(c), and Figure 7(d) show how AAE changes as memory size changes from 100KB to 1000KB for the three algorithms in four different datasets. Comparing to the CM sketch, AAE of HeavyGuardian is between 9.56 and 35.86 times smaller in CAIDA dataset, between 8.08 and 30.40 times smaller in Campus dataset, between 13.61 to 30.86 times smaller in Web page dataset, and between 10.48 to 79.75 in Synthetic dataset. Comparing to the P_{cu} sketch, AAE of HeavyGuardian is between 1.72 to 20.23, between 1.97 to 27.63, between 11.30 to 32.57, and between 3.04 to 48.99 times smaller in the four datasets, respectively. The results show that the accuracy of HeavyGuardian is much higher than existing algorithms.

2) Measuring ARE, varying memory size. Figure 8(a), Figure 8(b), Figure 8(c), and Figure 8(d) show how ARE changes as memory size changes from 100KB to 1000KB for the three algorithms in four different datasets. Comparing to the CM sketch, ARE of HeavyGuardian is between 9.30 to 35.24, between 6.80 to 27.45, between 12.96 to 53.21, and between 20.01 to 171.87 times smaller in the four datasets, respectively. Comparing to the P_{cu} sketch, ARE of HeavyGuardian is between 1.68 to 19.85, between 1.72 to 24.77, between 11.15 to 41.67, and between 1.63 to 34.99 times smaller in the four datasets, respectively.

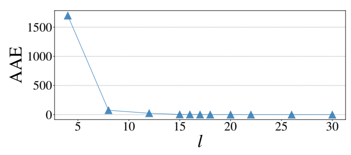
3) Measuring AAE and ARE, varying skewness. In this experiment, we vary the skewness of the synthetic datasets from 0.3 to 3.0, and set the memory size to 100KB. Figure 9(a) shows how AAE changes as the skewness of the dataset changes. The results show that AAE of HeavyGuardian is between 63.77 to 98.99 times smaller than that of the CM sketch, and is between 18.47 to 43.61 times s-



(a) Experiments on b

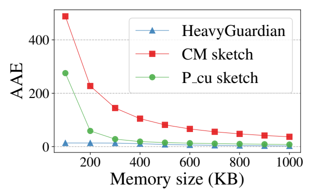


(b) Experiments on λ_h

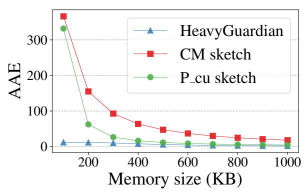


(c) Experiments on l

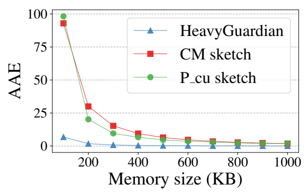
Figure 6: Experiments on system parameters.



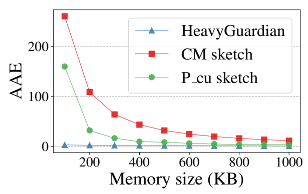
(a) CAIDA



(b) Campus

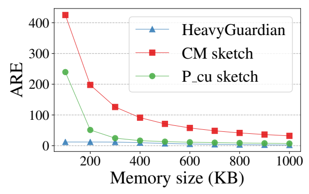


(c) Web page

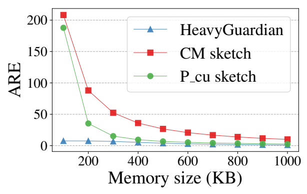


(d) Synthetic

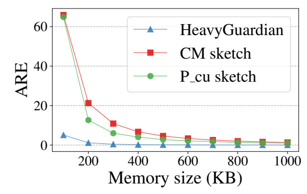
Figure 7: Measuring AAE, varying memory size.



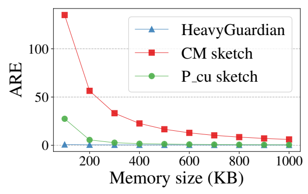
(a) CAIDA



(b) Campus

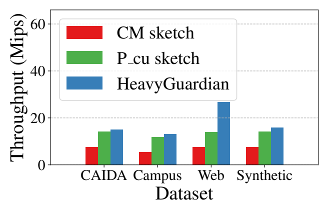


(c) Web page

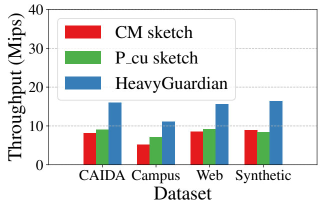


(d) Synthetic

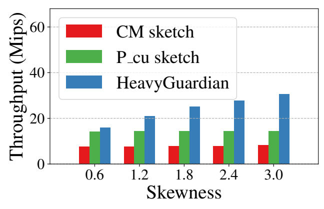
Figure 8: Measuring ARE, varying memory size.



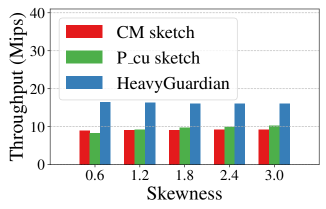
(a) Insertion speed vs. dataset



(b) Query speed vs. dataset



(c) Insertion speed vs. skewness



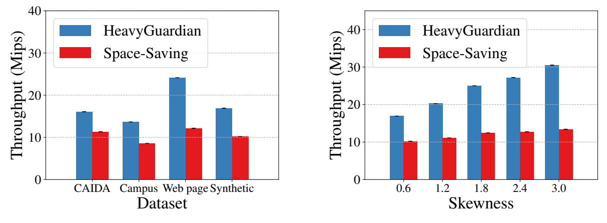
(d) Query speed vs. skewness

Figure 10: Throughput evaluation.

maller than that of the P_{cu} sketch. Figure 9(b) shows how ARE changes as the skewness changes. The results show that ARE of HeavyGuardian is between 157.83 to 225.04 times smaller than that of the CM sketch, and is between 45.66 to 100.15 times smaller than that of the P_{cu} sketch.

4) Measuring Throughput. In this experiment, we set the memory size to 1000KB, and we apply all three algorithms to all four datasets. Further, we vary the skewness of the Synthetic dataset to show the change of the throughput. As shown in Figure 10(a) and Figure 10(b), the insertion throughput and the query throughput of HeavyGuardian is always higher than that of the CM sketch and the P_{cu} sketch for every dataset. Specifically, when applying to the Web page dataset, the insertion throughput of HeavyGuardian is 3.53 times higher than that of the CM sketch, and is 1.92 times higher than that of the P_{cu} sketch. Further, the query throughput of HeavyGuardian is 1.81 times higher than that of the CM sketch, and is 1.69 times higher than that of the P_{cu} sketch.

As shown in Figure 10(c) and Figure 10(d), the insertion throughput and the query throughput of HeavyGuardian is always much higher than that of the CM sketch and the P_{cu} sketch, no matter what skewness the dataset has. Specifically, the insertion throughput of HeavyGuardian is between 1.918 to 3.760 times higher than that of the CM sketch, and is between 1.014 and 2.117 times higher than that of the P_{cu} sketch. Further, the query throughput of HeavyGuardian is between 1.750 and 1.876 times higher than that of the CM sketch, and is between 1.559 and 2.167 times higher than that of the P_{cu} sketch.



(a) Varying dataset (b) Varying skewness

Figure 15: Experiment results of speed for heavy hitter detection.

6.4.2 *Experiments on Heavy Hitter Detection & Heavy Change Detection*

In this section, we conduct experiments for heavy hitter detection and heavy change detection. For heavy hitter detection, we compare the performance of HeavyGuardian with that of one sketch plus a min-heap (*sketch+min-heap*) and Space-Saving. For heavy change detection, we compare the performance of HeavyGuardian with that of the *k-ary* sketch. For sketch+min-heap and the *k-ary* sketch, we set $d = 4$, and set the size of the min-heap to 30KB. Moreover, the width of the sketch, the width of the HeavyGuardian, and the number of buckets in Space-Saving, is computed based on the total memory size. We apply the three algorithms to CAIDA, Campus, Web page, and Synthetic datasets. We measure precision, recall, and ARE to compare the performances of the three algorithms, and also vary the memory size and the skewness of the dataset to show the change of the performance of all these algorithms. Further, we also measure the throughput for all four datasets to show the speed of these algorithms. We do not conduct experiments of heavy change detection on Synthetic dataset, because this dataset cannot generate changes in frequency between two adjacent time windows of the data stream.

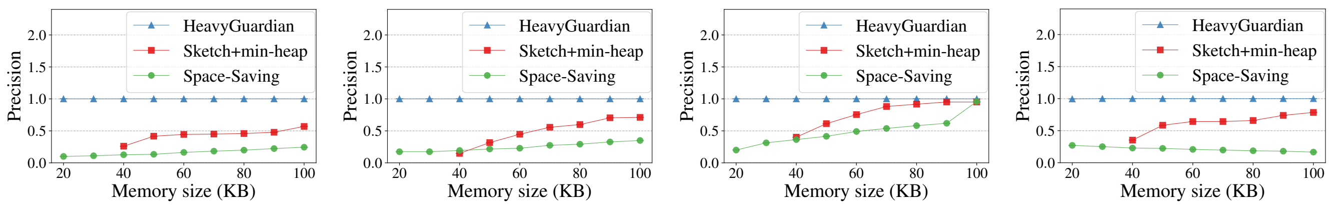
I. Heavy Hitter Detection

In the experiments of varying the memory size, we vary the memory size from 20KB to 100KB for Space-Saving and HeavyGuardian, but vary the memory size from 40KB to 100KB for sketch+min-heap, because sketch+min-heap requires 30KB for te min-heap. In the experiments of varying the skewness of the dataset, we vary the skewness from 0.3 to 3.0 for all algorithms.

1) Measuring precision, varying memory size. Figure 11(a), Figure 11(b), Figure 11(c), and Figure 11(d) show how precision of all three algorithms changes when varying memory size. For all the four datasets, HeavyGuardian always achieves 100% precision for every memory size, no matter how many guardians are used. *It is because the estimated frequency is smaller than the real frequency for almost all items, and thus when the estimated frequency is larger than the heavy hitter threshold, its real frequency should also be larger than the threshold.* However, for Space-Saving, the precision is only 12.7%, 19.3%, 36.4%, and 22.9% when the memory size is 40KB. For sketch+min-heap, when memory size is 40KB, the precision is only 26.0%, 14.8%, 40.1%, and 35.3% for CAIDA, Campus, Web page, and Synthetic dataset, respectively. The results show that the accuracy of HeavyGuardian is much higher than existing algorithms.

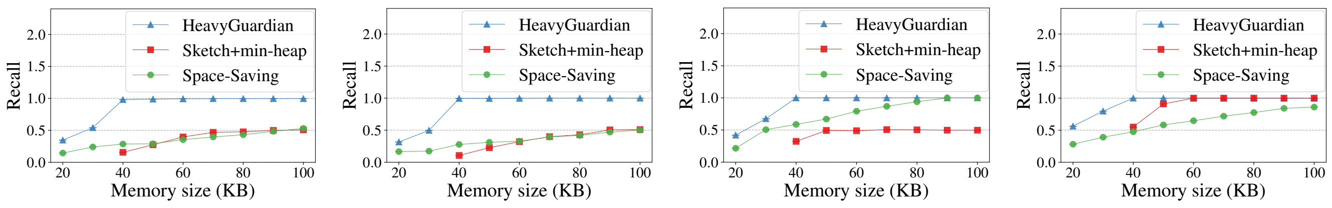
2) Measuring recall, varying memory size. Figure 12(a), Figure 12(b), Figure 12(c), and Figure 12(d) show how recall of all three algorithms changes when varying memory size. The results show that when memory size is varied from 40KB to 100KB, the recall of HeavyGuardian is almost 100% for all four datasets. Only when the memory size is smaller than 40KB, the recall of HeavyGuardian decreases, but also much higher than Space-Saving. While for Space-Saving, when the memory size is 40KB, the recall is only 28.6% for CAIDA dataset, 27.8% for Campus dataset, 58.8% for Web page dataset, and 47.5% for Synthetic dataset. For sketch+min-heap, the recall is only 15.6% for CAIDA dataset, 10.6% for Campus dataset, 32.4% for Web page dataset, and 54.9% for Synthetic dataset.

3) Measuring ARE, varying memory size. Figure 13(a), Figure 13(b), Figure 13(c), and Figure 13(d) show the results. The results show that for CAIDA dataset, ARE of HeavyGuardian is between 62,381 and 2,364,154 times smaller than that of sketch+min-heap, and is between 330,658 and 4,023,510 times smaller than that of Space-Saving. For Campus dataset, ARE of HeavyGuardian is between 444,521 and 5,009,227 times smaller than that of sketch+min-heap, and is between 173,786 and 1,370,122 times smaller than that of Space-Saving. For Web page dataset, ARE of HeavyGuardian is between 2,084,227 and 828,730,985 times smaller than that of sketch+min-heap, and is between 23,294 and 225,262,199 times smaller than that of Space-Saving. For Synthetic dataset, ARE of HeavyGuardian is between 7,164,726 and 60,424,840 times smaller than that of sketch+min-heap, and is between 41,360 and 282,281,271 times smaller than that of Space-Saving. The huge differences of ARE show that the accuracy for frequen-



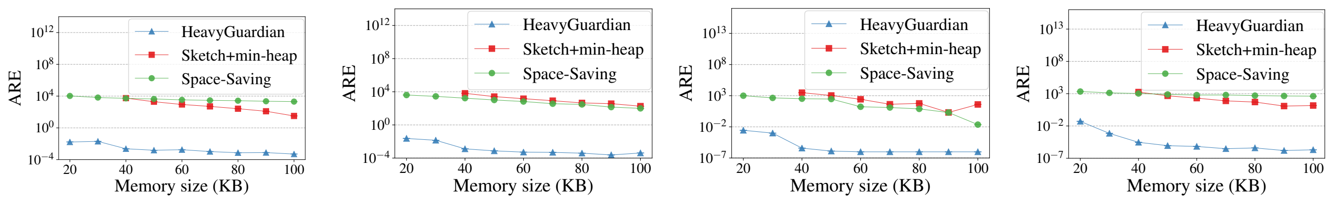
(a) CAIDA (b) Campus (c) Web page (d) Synthetic

Figure 11: Measuring Precision, varying memory size.



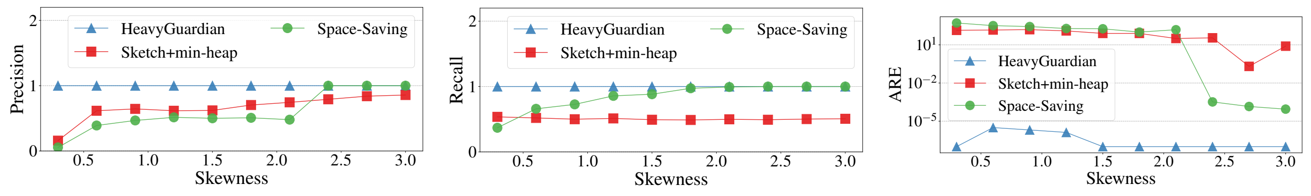
(a) CAIDA (b) Campus (c) Web page (d) Synthetic

Figure 12: Measuring Recall, varying memory size.



(a) CAIDA (b) Campus (c) Web page (d) Synthetic

Figure 13: Measuring ARE, varying memory size.



(a) Measuring Precision (b) Measuring Recall (c) Measuring ARE

Figure 14: Experiments on varying skewness.

cy estimation of HeavyGuardian is significantly higher than existing algorithms. *Note that for CAIDA dataset, HeavyGuardian works well when the memory is 20KB, and thus each distinct item only requires around 0.005 bit on average.*

4) Measuring precision, varying skewness. Figure 14(a) shows how precision changes when varying the skewness of the Synthetic dataset. The results show that HeavyGuardian achieves 100% precision no matter what skewness the dataset has. While for sketch+min-heap, its precision is only 16% when skewness is 0.3, and for Space-Saving, the precision is 5.4%. The results show that HeavyGuardian works well for various kinds of datasets, while other algorithms are sensitive to the skewness of the dataset.

5) Measuring recall, varying skewness. Figure 14(b) shows how recall changes when varying the skewness of the dataset. For HeavyGuardian, the recall is 100% no matter what skewness the dataset has. While for sketch+min-heap, the recall is 53.6% when the skewness is 0.3. For Space-Saving, the recall is 36.9% when the skewness is 0.3.

6) Measuring ARE, varying skewness. As shown in Figure 14(c), for all values of skewness, ARE of HeavyGuardian is between 2,053,082 and 1,396,800,962 times smaller than that of sketch+min-heap, and is between 887 and 5,256,620,846 times smaller than that of Space-Saving.

7) Measuring Throughput. As shown in Figure 15(a), in every four dataset, the throughput of HeavyGuardian is significant higher than that of Space-Saving. Specifically, the throughput of HeavyGuardian is between 1.419 and 1.982 times higher than that of Space-Saving. As shown in Figure 15(b), the throughput of HeavyGuardian is also significantly higher than that of Space-Saving for each skewness. The throughput of HeavyGuardian is between 1.436 and 1.828 times higher than that of Space-Saving. We do not conduct experiments on sketch+min-heap, because the time complexity of sketch+min-heap is not $O(1)$.

II. Heavy Change Detection

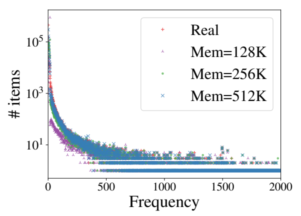
In experiments for heavy change detection, we vary the memory size from 10KB to 300KB for HeavyGuardian to show the change of the performance. For the *k-ary* sketch, we vary the memory size from 100KB to 300KB, because it requires 60KB for the two min-heaps to store the item IDs.

1) Measuring precision, varying memory size. Figure 16(a), Figure 16(b), and Figure 16(c), show how precision of both two algorithms changes when varying memory size. The results show that when memory size is varied from 100KB to 300KB, the precision of HeavyGuardian changes from 97.5% to 100% for CAIDA dataset, changes from 95.5% to 100% for Campus dataset, and changes from 98.9% to 100% for CAIDA dataset. While for the *k-ary* sketch, the precision changes from 7.2% to 52.3% for CAIDA dataset, changes from 6.7% to 31.2% for Campus dataset, and changes from 26.9% to 85.7% for Web page dataset. When the memory size is only 20KB, the precision of HeavyGuardian still reaches 83.3%, 70.4%, and 42.8% for CAIDA, Campus, and Web page dataset.

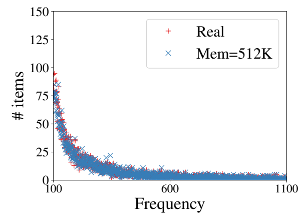
2) Measuring recall, varying memory size. Figure 17(a), Figure 17(b), and Figure 17(c), show how recall of both two algorithms changes when varying memory size. When memory size is larger than 50KB, the recall of HeavyGuardian is almost 100% for all three datasets. However, for the *k-ary* sketch, when memory size is 100KB, the recall is only 65.3%, 48.6%, 26.9% for CAIDA, Campus, and Web page dataset, respectively. When memory size is only 10KB, the recall of HeavyGuardian is still 78.4%, 81.0%, and 73.3% for CAIDA, Campus, and Web page dataset, respectively.

3) Measuring AAE, varying memory size. Figure 18(a), Figure 18(b) and Figure 18(c), show the results. For CAIDA dataset, AAE of HeavyGuardian is between 137.38 and 218.82 times smaller than that of the *k-ary* sketch. For Campus dataset, AAE of HeavyGuardian is between 250.01 and 417.31 times smaller than that of the *k-ary* sketch. For Web page dataset, AAE of HeavyGuardian is between 628.45 and 1448.35 times smaller than that of the *k-ary* sketch.

The speed of HeavyGuardian for heavy change detection is same as that for heavy hitter detection. Due to space limitation, we omit speed evaluation for heavy change detection.



(a) Distribution from 1 to 2000



(b) Distribution from 100 to 1100

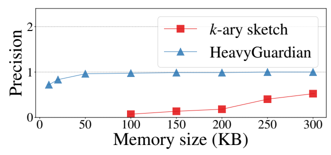
Figure 19: Real-time frequency distribution & entropy.

6.4.3 *Experiments on Real-time Frequency Distribution Estimation & Entropy Estimation*

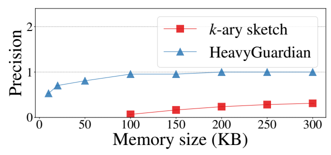
Due to space limitation, We conduct experiments for real-time frequency distribution estimation and entropy estimation only on CAIDA dataset. Because there is no existing algorithm achieves real-time frequency distribution and entropy estimation, we only compare the estimated frequency distribution and entropy with the real value.

In the experiments of real-time frequency distribution estimation, we set the memory size to 128KB, 256KB, and 512KB for HeavyGuardian. As shown in Figure 19(a), as the memory size increases, the estimated frequency distribution is closer to the real frequency distribution. Further, the results show that as the given frequency (the x-axis in Figure 19(a)) increases, the estimated number of items whose frequency is equal to the given value (the y-axis in Figure 19(a)) increases. As shown in Figure 19(b), we limit the frequency from 100 to 1000, and we only compare the real frequency with the estimated frequency when the memory size is 512KB. The results show that the two distributions are almost the same, which shows the accuracy of HeavyGuardian for frequency distribution estimation.

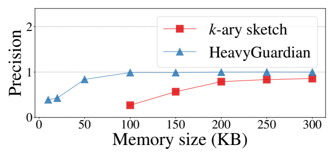
In the experiments of entropy estimation, we first conduct experiments on fixed sized time windows, and then conduct experiments for real-time entropy estimation. Specifically, we set the memory size to 512KB. Further, for experiments of real-time entropy estimation, we select 3 different CAIDA datasets (2M items in each dataset), and combine them together as one big dataset. As shown in Figure 20(a), we compute one entropy for every 1M items, and the trace of



(a) CAIDA

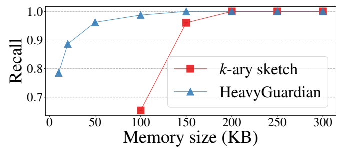


(b) Campus

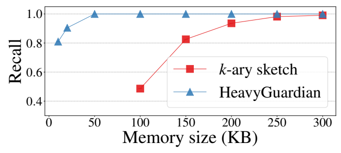


(c) Web page

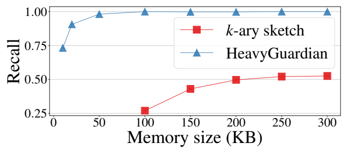
Figure 16: Measuring Precision, varying memory size.



(a) CAIDA

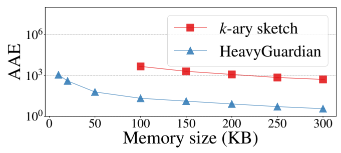


(b) Campus

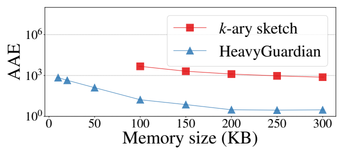


(c) Web page

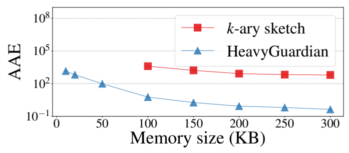
Figure 17: Measuring Recall, varying memory size.



(a) CAIDA

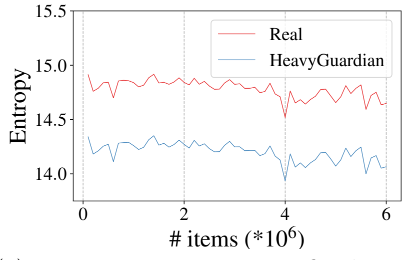


(b) Campus

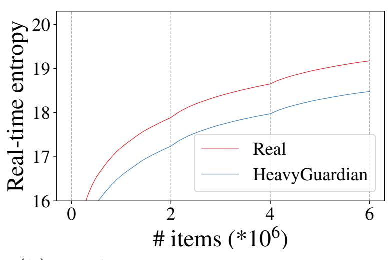


(c) Web page

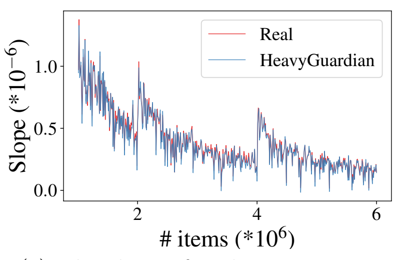
Figure 18: Measuring AAE, varying memory size.



(a) Entropy estimation in fixed sized time windows



(b) Real-time entropy estimation



(c) The slope of real-time entropy

Figure 20: Real-time entropy distribution.

the estimated entropy is almost the same as that of the real entropy, and thus we can easily detect anomalies when the entropy changes significantly. As shown in Figure 20(b), when changing to another dataset, traces of both the estimated real-time entropy and the real entropy changes significantly. We also plot the slope of the real-time entropy in Figure 20(c). The results show that the estimated slope of the real-time entropy is almost the same as the real value, and the slope becomes significantly high when changing to another dataset. The experimental results show that HeavyGuardian works well in real-time entropy estimation, which supports it to detect anomalies in real time.

7. CONCLUSION

There are five typical stream processing tasks: frequency estimation, heavy hitter detection, heavy change detection, frequency distribution estimation, and entropy estimation. Each of these tasks has quiet a few solutions, but can hardly achieve high accuracy on real datasets when memory is tight. To address this issue, in this paper, we propose a novel data structure HeavyGuardian. The key idea is to intelligently keep and guard the information of hot items and approximately record the frequencies of cold items using a key technique named "exponential decay". To the best of our knowledge, this is the first effort to use one data structure to process five streaming processing tasks. We derive theoretical bound for HeavyGuardian, and validate it by experiments. Extensive experimental results show that HeavyGuardian achieves both much smaller error and faster speed than the state-of-the-art algorithms for each of the five typical tasks.

8. REFERENCES

[1] The source codes of heavyguardian and other related algorithms. `https://github.com/Gavindeed/HeavyGuardian`.

[2] Shoba Venkataraman, Dawn Song, Phillip B Gibbons, and Avrim Blum. New streaming algorithms for fast detection of superspreaders. *Department of Electrical and Computing Engineering*, page 6, 2005.

[3] Minlan Yu, Lavanya Jose, and Rui Miao. Software defined traffic measurement with opensketch. In *NSDI*, volume 13, pages 29–42, 2013.

[4] Ahmed Metwally, Divyakant Agrawal, and Amr El Abbadi. Efficient computation of frequent and top-k elements in data streams. In *International Conference on Database Theory*, pages 398–412. Springer, 2005.

[5] Ahmed Metwally, Divyakant Agrawal, and Amr El Abbadi. Efficient computation of frequent and top-k elements in data streams. In *Proc. Springer ICDT*, 2005.

[6] Graham Cormode, Flip Korn, S Muthukrishnan, and Divesh Srivastava. Finding hierarchical heavy hitters in data streams. In *Proceedings of the 29th international conference on Very large data bases-Volume 29*, pages 464–475. VLDB Endowment, 2003.

[7] Ran Ben Basat, Gil Einziger, Roy Friedman, Marcelo Caggiani Luizelli, and Erez Waisbard. Constant time updates in hierarchical heavy hitters. *arXiv preprint arXiv:1707.06778*, 2017.

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

[9] Pratanu Roy, Arijit Khan, and Gustavo Alonso. Augmented sketch: Faster and more accurate stream processing. In *Proc. SIGMOD*, 2016.

[10] Tong Yang, Yang Zhou, Hao Jin, Shigang Chen, and Xiaoming Li. Pyramid sketch: a sketch framework for frequency estimation of data streams. *Proceedings of the VLDB Endowment*, 10(11):1442–1453, 2017.

[11] Moses Charikar, Kevin Chen, and Martin Farach-Colton. Finding frequent items in data streams. *Automata, languages and programming*, pages 784–784, 2002.

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

[13] Cristian Estan and George 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.

[14] Katsiaryna Mirylenka, Graham Cormode, Themis Palpanas, and Divesh Srivastava. Conditional heavy hitters: detecting interesting correlations in data streams. *The VLDB Journal*, 24(3):395–414, 2015.

[15] Joong Hyuk Chang and Won Suk Lee. Finding recent frequent itemsets adaptively over online data streams. In *Proceedings of the ninth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 487–492. ACM, 2003.

[16] Yin-Ling Cheung and Ada Wai-Chee Fu. Mining frequent itemsets without support threshold: with and without item constraints. *IEEE Transactions on Knowledge and Data Engineering*, 16(9):1052–1069, 2004.

[17] Gobinda G Chowdhury. *Introduction to modern information retrieval*. Facet publishing, 2010.

[18] Vibhaalakshmi Sivaraman, Srinivas Narayana, Ori Rottenstreich, S Muthukrishnan, and Jennifer Rexford. Heavy-hitter detection entirely in the data plane. In *Proceedings of the Symposium on SDN Research*, pages 164–176. ACM, 2017.

[19] Yu Zhang, BinXing Fang, and YongZheng Zhang. Identifying heavy hitters in high-speed network monitoring. *Science China Information Sciences*, 53(3):659–676, 2010.

[20] Mohamed A Soliman, Ihab F Ilyas, and Kevin Chen-Chuan Chang. Top-k query processing in uncertain databases. In *Data Engineering, 2007. ICDE 2007. IEEE 23rd International Conference on*, pages 896–905. IEEE, 2007.

[21] Erik Demaine, Alejandro López-Ortiz, and J Munro. Frequency estimation of internet packet streams with limited space. *AlgorithmsESA 2002*, pages 11–20, 2002.

[22] Richard M Karp, Scott Shenker, and Christos H Papadimitriou. A simple algorithm for finding frequent elements in streams and bags. *ACM TODS*, 28(1):51–55, 2003.

[23] Gurmeet Singh Manku and Rajeev Motwani. Approximate frequency counts over data streams. In *Proc. VLDB*, pages 346–357, 2002.

[24] Monika Rauch Henzinger. Algorithmic challenges in web search engines. *Internet Mathematics*, 1(1):115–123, 2004.

[25] Er Krishnamurthy, Subhabrata Sen, and Yin Zhang. Sketchbased change detection: Methods, evaluation, and applications. In *In ACM SIGCOMM Internet Measurement Conference*. Citeseer, 2003.

[26] Robert Schweller, Ashish Gupta, Elliot Parsons, and Yan 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. ACM, 2004.

[27] Chung Chen and Lon-Mu Liu. Forecasting time series with outliers. *Journal of Forecasting*, 12(1):13–35, 1993.

[28] Chung Chen and Lon-Mu Liu. Joint estimation of model parameters and outlier effects in time series. *Journal of the American Statistical Association*, 88(421):284–297, 1993.

[29] Ruey S Tsay. Time series model specification in the presence of outliers. *Journal of the American Statistical Association*, 81(393):132–141, 1986.

[30] Ruey S Tsay. Outliers, level shifts, and variance changes in time series. *Journal of forecasting*, 7(1):1–20, 1988.

[31] Viswanath Poosala and Yannis E Ioannidis. Estimation of query-result distribution and its application in parallel-join load balancing. In *VLDB*, pages 448–459, 1996.

[32] Yannis E Ioannidis and Viswanath Poosala. Histogram-based approximation of set-valued query-answers. In *VLDB*, volume 99, pages 174–185, 1999.

[33] Viswanath Poosala and Yannis E Ioannidis. Selectivity estimation without the attribute value independence assumption. In *VLDB*, volume 97, pages 486–495, 1997.

[34] Shanshan Ying, Flip Korn, Barna Saha, and Divesh Srivastava. Treescope: finding structural anomalies in semi-structured data. *Proceedings of the VLDB Endowment*, 8(12):1904–1907, 2015.

[35] Daniel Barbará, Julia Couto, Sushil Jajodia, and Ningning Wu. Adam: a testbed for exploring the use of data mining in intrusion detection. *ACM Sigmod Record*, 30(4):15–24, 2001.

[36] Nick Duffield, Carsten Lund, and Mikkel 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.

[37] Abhishek Kumar, Minho Sung, Jun Jim Xu, and Jia Wang. Data streaming algorithms for efficient and accurate estimation of flow size distribution. In *Proc. ACM SIGMETRICS*, pages 177–188, 2004.

[38] Yuliang Li, Rui Miao, Changhoon Kim, and Minlan Yu. Flowradar: A better netflow for data centers. In *NSDI*, pages 311–324, 2016.

[39] Laura Feinstein, Dan Schnackenberg, Ravindra Balupari, and Darrell Kindred. Statistical approaches to ddos attack detection and response. In *DARPA Information Survivability Conference and Exposition, 2003. Proceedings*, volume 1, pages 303–314. IEEE, 2003.

[40] Anukool Lakhina, Mark Crovella, and Christophe Diot. Mining anomalies using traffic feature distributions. In *Proc. ACM SIGCOMM*, pages 217–228, 2005.

[41] Arno Wagner and Bernhard Plattner. Entropy based worm and anomaly detection in fast ip networks. In *Enabling Technologies: Infrastructure for Collaborative Enterprise, 2005. 14th IEEE International Workshops on*, pages 172–177. IEEE, 2005.

[42] Kuai Xu, Zhi-Li Zhang, and Supratik Bhattacharyya. Profiling internet backbone traffic: behavior models and applications. In *Proc. ACM SIGCOMM*, pages 169–180, 2005.

[43] Ashwin Lall, Vyas Sekar, Mitsunori Ogihara, Jun Xu, and Hui Zhang. Data streaming algorithms for estimating entropy of network traffic. In *Proc. ACM SIGMETRICS*, pages 145–156, 2006.

[44] David MW Powers. Applications and explanations of zipf's law. In *Proceedings of the joint conferences on new methods in language processing and computational natural language learning*, pages 151–160. Association for Computational Linguistics, 1998.

[45] The caida anonymized internet traces 2016. `http://www.caida.org/data/overview/`.

[46] Frequent itemset mining dataset repository. `http://fimi.ua.ac.be/data/`.

[47] Alex Rousskov and Duane Wessels. High-performance benchmarking with web polygraph. *Software: Practice and Experience*, 2004.

[48] Christian Henke, Carsten Schmoll, and Tanja Zseby. Empirical evaluation of hash functions for multipoint measurements. *SIGCOMM Comput. Commun. Rev.*, 38(3):39–50, July 2008.