HeavyGuardian: Separate and Guard Hot Items in Data Streams

Tong Yang* Peking University

Junzhi Gong* Peking University

Haowei Zhang* Peking University

Lei Zou* Peking University

Lei Shi† Chinese Academy of Sciences

Xiaoming Li* Peking University

ABSTRACT

[1]Data stream processing is a fundamental issue in many fields, such as data mining, databases, network traffic measurement. There are five typical tasks in data stream processing: frequency estimation, heavy hitter detection, heavy change 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, we propose a novel data structure named HeavyGuardian. The key idea is to intelligently separate and guard the information of hot items while approximately record the frequencies of cold items. We deploy HeavyGuardian on 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 codes of HeavyGuardian and other related algorithms are available at GitHub [1].

CCS CONCEPTS

• **Information systems → Data streams**; **Data structures**; *Data mining*;

KEYWORDS

Data stream processing, Data structure, Probabilistic and approximate data

ACM Reference Format:
Tong Yang, Junzhi Gong, Haowei Zhang, Lei Zou, Lei Shi, and Xiaoming Li. 2018. HeavyGuardian: Separate and Guard Hot Items in Data Streams. In *KDD '18: The 24th ACM SIGKDD International Conference on Knowledge Discovery Data Mining, August 19–23, 2018, London, United Kingdom.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3219819.3219978

*Department of Computer Science, Peking University, China
†SKLCS, Institute of Software, Chinese Academy of Sciences, China
[1]Co-primary authors: Tong Yang and Junzhi Gong. Junzhi Gong and Haowei Zhang finished this work under the guidance of their supervisor: Tong Yang. Corresponding author: Lei Zou. This work is supported by National Basic Research Program of China (973 Program, 2014CB340405), Primary Research & Development Plan of China (2016YFB1000304), NSFC (61672061, 61622201 and 61532010), the National Key Research and Development Program of China under grant 2018YFB1003504.

KDD '18, August 19–23, 2018, London, United Kingdom
© 2018 Association for Computing Machinery.
ACM ISBN 978-1-4503-5552-0/18/08…$15.00
https://doi.org/10.1145/3219819.3219978

1 INTRODUCTION

1.1 Background and Motivation

Approximate stream processing has been a hot issue for years. Given a data stream, each item could appear more than once. 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 within fixed-size time windows.[2] The above stream processing tasks and many other stream processing tasks, such as Super-Spreader [2], DDoS victims [3, 4], top-k frequent items [5–7], hierarchical heavy hitters [8–10] *etc*, care more about hot items. In practice, *most items are cold while only a few items are hot* [11–20]. Accurately recording the information of massive cold items wastes much memory, and could incur non-trivial error to the estimation of hot items when 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 the frequencies of cold items.* The key challenge is to judge whether the incoming item is hot or cold in real time. It is difficult to quickly record the history information of items with high accuracy in small memory.

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 [7] and the Augmented sketch [12]. Space-Saving [7] records each incoming item in a data structure named *Stream-Summary.* Stream-Summary is essentially an ordered list, achieving insertion, deletion, and finding the coldest item in $O(1)$ time. For each incoming item e, if e is in Stream-Summary, then it increments the recorded frequency of e by 1. Otherwise, it replaces the coldest item whose frequencies are $\hat{f}_{min}$ with e, and sets the frequency of e to $\hat{f}_{min} + 1$. In this way, many hot items are stored in Stream-Summary and cold items are expected to be evicted. However, there are two limitations: 1) it cannot record frequencies of cold items; 2) many later-incoming cold items are significantly over-estimated, and probably stay in Stream-Summary. Therefore, to reduce such error, Space-Saving usually needs to keep m items in Stream-Summary when requiring to store k (m is much larger than k) hottest items.

[2]Generally, these tasks are measured within fixed-size time windows.

The Augmented sketch [12] is a two-stage data structure: the first stage is a small array which sequentially stores δ hot items, and the second is a classic sketch (*e.g.*, a CM sketch) which stores all item frequencies. Late-incoming hot items are first inserted into the second sketch, and then swapped to the first stage. Because for each incoming item, it 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 many 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 very small cache, requiring frequent communications between the two stages.

In summary, existing algorithms using *record-all-evict-cold* have 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 *separate-and-guard-hot*. It intelligently separates hot items from cold items, and keeps and guards the information of hot items with large counters, while using small counters to record the frequencies of cold items.

We simplify the stream processing problem as follows: given a data stream, how to use a very few cells to accurately record the hottest item with its frequency. We aim to use two cells to store the two hottest items with their frequencies. The king (the hottest item) lives in one cell, while the guardian (the second hottest item) lives in the other cell. If the incoming item is a *supporter*, *i.e.*, it is same as the king or the guardian, then it simply increases the corresponding frequency. Otherwise, the incoming item is a *rebel*, then it weakens the guardian: decreasing the frequency f of the guardian with a biased probability. *The probability decreases exponentially as f increases linearly.* If the guardian is "killed", *i.e.*, its frequency is decreased to 0, it will be deleted, and the incoming item will become a new guardian. If the guardian gains more and more supporters, it could become a king. Further, if we use multiple cells, then each king will have multiple guardians, its frequency is often recorded with no error. Our algorithm is based on the above key idea, and uses several improvements: 1) split the data stream into many small sub-streams, and elect a king and several guardians for each sub-stream; 2) use small counters to store the frequencies of rebels.

Table 1: Main experimental results.

Task	Error rate improvement	Speed improvement
Frequency estimation	$\times 1.72 \sim 53.09$	$\times 1.69 \sim 13.30$
Heavy hitter detection	$\times 6.2 * 10^4 \sim 4.0 * 10^6$	$\times 1.419 \sim 1.928$
Heavy change detection	$\times 137.38 \sim 1448.35$	$\times 1.419 \sim 1.928$
Entropy estimation	$\times 3.69$	$\times 1.91 \sim 3.760$

Main experimental results: We present main experimental results in Table 1. For the first three tasks, we compare HeavyGuardian with the state-of-the-art, while for real-time entropy estimation, we compare HeavyGuardian with the naive algorithm because there is no existing algorithm. Take heavy hitter detection as an example. HeavyGuardian achieves $6.24 * 10^4 \sim 4.02 * 10^6$ times smaller error rate than the state-of-the-art – Space-Saving (See Figure 10(a)), and can use only 0.005 bit for each distinct item (See Figure 10(a)). Further, the speed of HeavyGuardian is up to 1.928 times higher than that of Space-Saving (See Figure 11).

1.3 Main Contributions

(1) We propose a novel data structure, named HeavyGuardian, which can intelligently separate and guard the information of hot items and approximately record the frequencies of cold items in a data stream.
(2) We derive the formula of the error bound of HeavyGuardian, and the experimental results validate its correctness.
(3) We deploy HeavyGuardian on five typical tasks of data stream processing. Extensive experimental results on real and synthetic datasets show that HeavyGuardian achieves both much higher accuracy and higher processing speed than the state-of-the-art at the same time.

2 BACKGROUND

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

2.1 Frequency Estimation

Frequency estimation is to estimate the number of appearances of any item in a data stream. Sketches can achieve memory efficiency and fast speed at the cost of introducing small error, and therefore gain wide acceptance recently [8–10, 12, 13, 21]. Typical sketches include Count sketches [22], Count-min (CM) sketches [23], CU sketches [15], Augmented sketch framework [12], Pyramid sketch framework [13], and more [21, 24]. 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 accommodate the largest frequency since hot items are often more important than cold items. Moreover, as the number of cold items are much larger than that of hot items [11–20], many counters store only a small number, and thus the significant bits of these counters are wasted. Augmented sketch framework uses a filter to accurately store frequencies of only a few hot items (*e.g.*, 32 hot items), thus the improvement is limited. 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 has been proved to achieve much higher accuracy and higher speed than all other algorithms. However, hot items require quite a few memory accesses, and thus insertion speed of the Pyramid sketch 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

Heavy hitter detection is to find items whose frequencies are larger than a predefined threshold. This is a critical task in data mining

[25], information retrieval [26], network measurement and management [27], and more [28]. There are two kinds of algorithms: sketch based algorithms and counter based algorithms. Sketch based algorithms use a sketch (*e.g.*, the CM sketch [23]) to approximately record frequencies of all items, and use a min-heap to maintain the k most frequent items. They 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 [7], Frequent [29], and Lossy counting [30]. These algorithms are similar to each other, and Space-Saving is the most widely used one, and the details are presented in Section 1.1.

2.3 Heavy Change Detection

Given two adjacent time windows, heavy change detection is to find those items whose frequencies differences are larger than a predefined threshold. This is also an important task in many big data scenarios, such as web search engines [31], anomalies detection [32, 33], time series forecasting and outlier analysis [34], *etc.* There are several well-known algorithms for heavy change detection. Based on the classic CM sketch [23], the *k-ary* sketch [32] achieves high accuracy for heavy change detection when memory space is large, but needs to know all item IDs. To address this issue, the *reversible* sketch [33] is based on the *k-ary* sketch, and can decode the item ID at the cost of complexities.

2.4 Real-time Frequency Distribution & Real-time Entropy

Frequency distribution estimation is important in database query optimization [35], structural anomalies detection [36], and network measurement and monitoring [37]. Notable algorithms for frequency distribution estimation include MRAC [37], FlowRadar [14], and more [38]. However, these algorithms cannot support *real-time* frequency distribution. *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 services. If the frequency distribution is estimated in real time, then IP service providers can adjust their strategies immediately, leading to better services for users.

Entropy refers to the uncertainty of a data stream. Formally, entropy is defined as $\sum_e \frac{f_e}{N} log_2 \frac{f_e}{N}$, where f_e is the frequency of e and N is the total number of items. Entropy estimation is important in data mining [39, 40], data quality estimation [41, 42], network measurement and monitoring [43]. For example, changes of entropy indicate anomalous incidents in the stream, which can be used for anomaly detection. 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. FlowRadar [14] can also be used for entropy estimation, but the performance is much poorer than the previous one. Unfortunately, these algorithms cannot support *real-time* entropy estimation. *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.

In summary, existing algorithms cannot estimate frequency distribution or entropy in real time. To support better performance, we manage to quickly update frequency distribution and entropy for each incoming item. To the best of our knowledge, this is *the first effort for real-time estimation of frequency distribution and entropy.*

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 then present an optimization technique.

3.1 HeavyGuardian Basics

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 approximately 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.

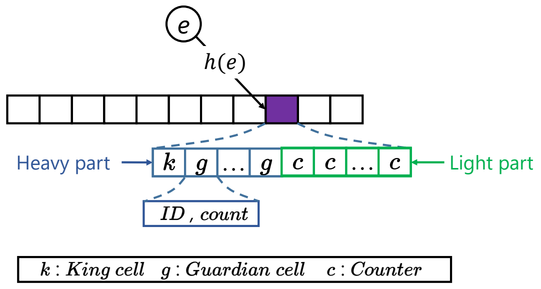


Figure 1: The data structure of HeavyGuardian.

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 predefined 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 $h'(e)$, and then increments counter $A[h(e)][h'(e)]_l$ in the light part of the bucket by 1.

Actually, we can still use $h(.)$ instead of $h'(.)$ in the light part, because each item is expected to be stored in either the heavy part or the light part, and thus $h(.)$ and $h'(.)$ do not need to be independent. Therefore, *only one hash computation is needed to process each incoming item.*

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 $A[h(e)][h'(e)]_l$ in the light part.

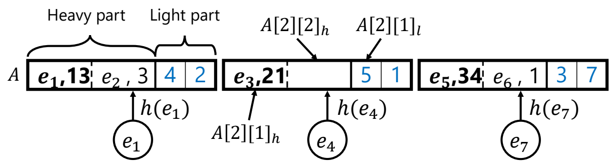


Figure 2: Examples of insertion of HeavyGuardian.

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 $h'(e_7) = 2$, 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) processing one cold item also is fast, as it only needs one additional access of the light part. Therefore, the time complexity of HeavyGuardian is $O(1)$.

3.2 Optimization: Using Fingerprints

When the size of the item ID is large, we propose to use fingerprints[3] to replace IDs in HeavyGuardian, to further improve its memory efficiency. In this way, the memory usage of HeavyGuardian is independent to the size of item ID. Due to hash collisions, some items could share the same fingerprint, and thus some cold items can be treated as hot items, which can lead to lower accuracy. However, as the probability of hash collisions is low, it introduces 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 HeavyGuardian. As shown in Table 2, we set $M = 1,000,000$. When the fingerprint size is larger than 16, the probability of fingerprint collisions is negligible.

Table 2: 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 4.1. *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 on 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.

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

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 possibly 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 summary, the theorem holds at any point of time. □

4.2 The Error Bound of the Heavy Part of HeavyGuardian

THEOREM 4.2. *Given a stream $\mathcal{S}$ with items $e_1, e_2, \cdots, e_M$, it obeys an arbitrary distribution. We assume that the heavy part of each bucket stores the hottest λ_h items that mapped to that bucket. Let e_i be the i^{th} hottest item, 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*

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

where

$$P_{weak} = e^{-\frac{i-1}{w}} \times \frac{[\frac{i-1}{w}]^{\lambda_h - 1}}{(\lambda_h - 1)!} \tag{3}$$

and

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

PROOF. According to the assumption, e_i is a hot item, and it is in the mapped bucket. 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

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

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 $b^{-f_i} \longrightarrow 0$. Therefore, we have

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

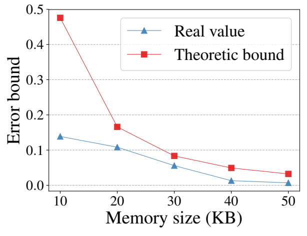
then we solve this equation and get $E(\hat{f_i})$

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

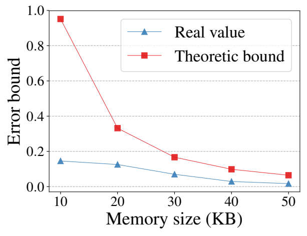
and based on Markov inequality, we have

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

Therefore, the theorem holds. □



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



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

Figure 3: Theoretical error bound and real value.

To validate the error bound derived in Theorem 4.2, we conduct experiments on CAIDA datasets. Here we set $\lambda_h = 2$, $b = 1.08$,

$\epsilon = 2^{-18}$ and 2^{-19}, $N = 3.2 \times 10^7$, and $M = 10^6$. 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 present how to deploy HeavyGuardian on five typical tasks mentioned in Section 2.

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 with large counters, and use small counters to record frequencies of cold items in the light part. Therefore, when the memory space 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 in the heavy part of each bucket, and manage to record the item ID of only hot items *in an auxiliary list.* Because heavy hitter detection and heavy change detection focus on only hot items, we do not need the light parts which store frequencies of cold items. Therefore, we simply set $\lambda_l = 0$.

5.2.1 Detecting Heavy Hitters. **Insertion:** 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 e into the list 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 store the IDs of heavy hitters in the auxiliary list B 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 small probability. The list B only stores the IDs of heavy hitters, and thus the memory usage is small, and in addition, its memory can be dynamically allocated.

Query: To report all heavy hitters, it traverses the list 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 adjacent time windows in a data stream, we deploy one HeavyGuardian to each time window. We set the heavy hitter threshold exactly as the heavy change threshold, and we can then get heavy hitters in each time window. In such a setting of thresholds, heavy changes must be heavy hitters in at least one time window. 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.

5.3 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 auxiliary array of counters to record the distribution of frequencies. 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 frequencies are i. For each incoming item e, we first insert e into HeavyGuardian, and get 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 HeavyGuardian achieves high accuracy, and the recorded frequencies of hot items increase one by one, our proposed algorithm achieves achieve high accuracy for real-time frequency distribution and entropy.

6 EXPERIMENTAL RESULTS

6.1 Experiment Setup

Dataset:

1) CAIDA: This dataset is from *CAIDA Anonymized Internet Trace 2016* [44], consisting of IP packets. Each item is identified by the source IP address and the destination IP address. The dataset contains 10M items, with around 4.2M distinct items.

2) Web page: This dataset is built from a spidered collection of web HTML documents [45]. This dataset contains 10M items, belonging to around 0.4M distinct items.

We also conduct experiments on synthetic datasets with different skewness values. Due to space limitation, we present results of this part of experiments in the technical report [1].

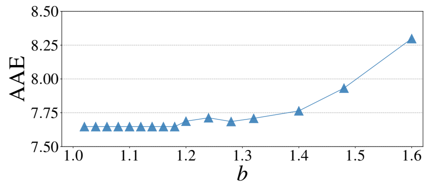
Implementation: HeavyGuardian and other related algorithms are implemented in C++. We tried several different hash functions in our experiments, and found that the performance keeps unchanged. Here we use the Bob hash that literature [46] recommends. All the programs are run on a server with dual 6-core CPUs (24 threads, Intel Xeon CPU E5-2620 @2 GHz) and 62 GB total system memory. The cache size is typically several megabytes. To preserve cache memory to other important tasks, we keep the memory size in our experiment under 1000KB.

6.2 Metrics

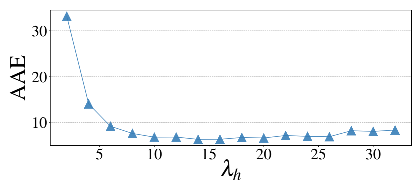
Precision: Precision measures the ratio of number of correctly reported answers to number of reported answers. It is defined as $\frac{|\Omega \cap \Psi|}{|\Omega|}$, where Ω is the set of answers reported by algorithms, and Ψ is the set of correct answers.

Recall: Recall measures the ratio of number of correctly reported answers to number of correct answers. It is defined as $\frac{|\Omega \cap \Psi|}{|\Psi|}$.

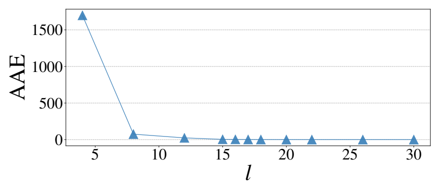
Average Relative Error (ARE): ARE measures the accuracy of the estimated frequency. It is defined as $\frac{1}{M} \sum_{e_j \in \Omega} \frac{|\hat{f}_j - f_j|}{f_j}$, where M



(a) Experiments on b

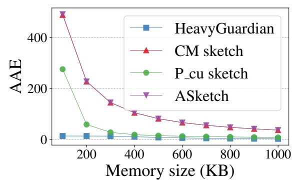


(b) Experiments on λ_h

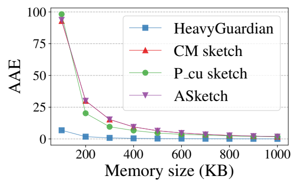


(c) Experiments on l

Figure 4: Experiments on system parameters.

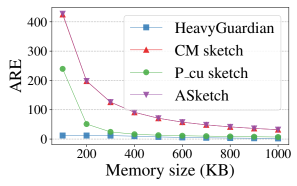


(a) CAIDA

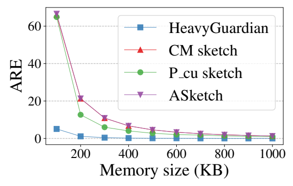


(b) Web page

Figure 5: AAE vs. memory size (frequency estimation).



(a) CAIDA



(b) Web page

Figure 6: ARE vs. memory size (frequency estimation).

is the total number of distinct items, f_j is the real frequency of e_j, and $\hat{f}_j$ is the estimated frequency of e_j.

Average Absolute Error (AAE): AAE measures the accuracy of the estimated frequency (or differences between two adjacent time windows of the data stream). It is defined as $\frac{1}{M}\sum_{e_j\in\Omega}|\hat{f}_j - f_j|$ or $\frac{1}{M}\sum_{e_j\in\Omega}|\hat{d}_j - d_j|$, 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. The throughput is defined as N/T, where N is the number of items, and T is the time used. We use Million of insertions per second (Mips) to measure the throughput.

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. Because these three parameters are only related to the heavy part, we focus on the accuracy of hot items and omit cold items. Further, we set the total memory size to 100KB, use the CAIDA dataset in our experiments, and use AAE to evaluate the accuracy.

Effects of b (Figure 4(a)): In this experiment, we set $\lambda_h = 2$ and do not use fingerprints to replace ID fields. 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.

Effects of λ_h (Figure 4(b)): In this experiment, we do not use fingerprints to replace ID fields. 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. In order to let the size of a bucket fit into a cache line, we set λ_h to 8.

Effects of l (Figure 4(c)): The results show that when l increases, AAE first decreases significantly, and then keeps at a low value when $l > 15$. 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 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. This indicates $\lambda_l = 64$.

6.4 Experiments on Stream Processing Tasks

In this section, we apply HeavyGuardian to the five typical stream processing tasks: 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 these tasks.

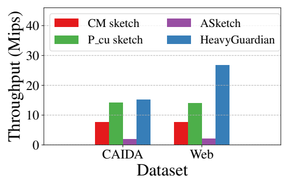
6.4.1 *Experiments on Frequency Estimation.*

In this section, we conduct experiments on frequency estimation. We compare the performance of HeavyGuardian with the CM sketch [23] and the P_{cu} sketch (the CU sketch using the Pyramid sketch framework [13]). We do not conduct experiments for other sketches, because [13] has shown that the performance of the P_{cu} sketch is much better than sketches of CU [15], Count [22], and Augmented [12] in terms of both accuracy and speed. Specifically, we use AAE and ARE as metrics to evaluate the accuracy, and use throughput to evaluate the speed. We vary the memory size to show the change in the performance.

AAE vs. memory size (Figure 5(a), 5(b)): In this experiment, we vary the memory size from 100KB to 1000KB. Compared to the CM sketch, AAE of HeavyGuardian is between 9.56 and 35.86 times smaller in the CAIDA dataset, and between 13.61 to 30.86 times smaller in the Web Page dataset. Compared to the P_{cu} sketch, AAE of HeavyGuardian is between 1.72 to 20.23, and between 11.30 to 32.57 times smaller in the two datasets, respectively. Compared to ASketch, AAE of HeavyGuardian is between 9.58 and 36.10, and between 13.73 and 30.93 times smaller in the two datasets.

ARE vs. memory size (Figure 6(a), 6(b)): Compared to CM sketch, ARE of HeavyGuardian is between 9.30 to 35.24 and between 12.96 to 53.21 smaller in the two datasets, respectively. Compared to P_{cu} sketch, ARE of HeavyGuardian is between 1.68 to 19.85 and between

11.15 to 41.67 smaller in the two datasets, respectively. Compared to ASketch, ARE of HeavyGuardian is between 9.31 and 35.44, and between 13.10 and 53.09 times smaller in the two datasets.


(a) Insertion speed vs. dataset

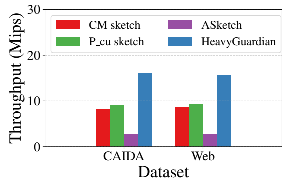
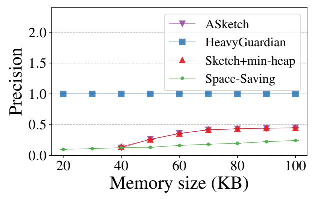

(b) Query speed vs. dataset

Figure 7: Throughput evaluation (frequency estimation).


(a) CAIDA

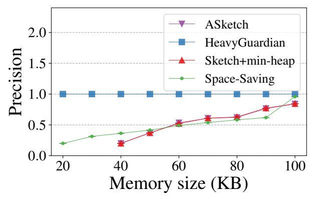

(b) Web page

Figure 8: Precision vs. memory size (heavy hitter detection).

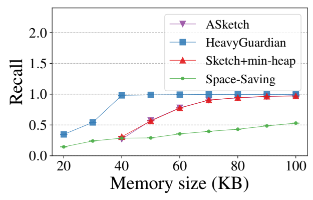
Throughput vs. dataset (Figure 7(a), 7(b)): In this experiment, we set the memory size to 1000KB, and we apply all three algorithms to both two datasets. The insertion throughput and the query throughput of HeavyGuardian is always higher than that of other algorithms for both datasets. Specifically, when applying to the Web page dataset, the insertion throughput of HeavyGuardian is 3.53 times higher than that of CM sketch, is 1.92 times higher than that of P_{cu} sketch, and is 13.30 times higher than that of ASketch. Further, the query throughput of HeavyGuardian is 1.81 times higher than that of CM sketch, is 1.69 times higher than that of P_{cu} sketch, and is 5.37 times higher than that of ASketch.

6.4.2 Experiments on Heavy Hitter Detection & Heavy Change Detection. For heavy hitter detection, we compare the performance of HeavyGuardian with that of one sketch plus a min-heap (*CM sketch+min-heap*) [23], Space-Saving [7], and ASketch [12]. For heavy change detection, we compare the performance of HeavyGuardian with that of the *k-ary* sketch [32] and ASketch. For sketch+min-heap, the *k-ary* sketch, and Asketch, 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, are computed based on the total memory size. We measure precision, recall, and ARE to compare the performances of the three algorithms, and also vary the memory size to show the change in the performance. Further, we also measure the throughput for both datasets to show the speed of these algorithms.

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 and ASketch, because they require 30KB for the min-heap.

Precision vs. memory size (Figure 8(a), 8(b)): For both two datasets, HeavyGuardian always achieves 100% precision for every memory size. However, when the memory size is 40KB, the precision is only 12.7% and 36.4% for Space-Saving, is 26.0% and 40.1% for sketch+min-heap, and is 12.5% and 20.0% for ASketch for the two datasets, respectively.


(a) CAIDA

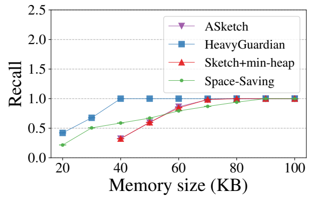
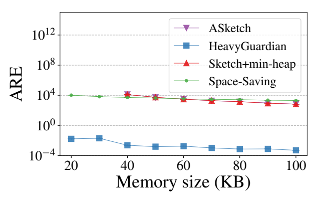

(b) Web page

Figure 9: Recall vs. memory size (heavy hitter detection).


(a) CAIDA

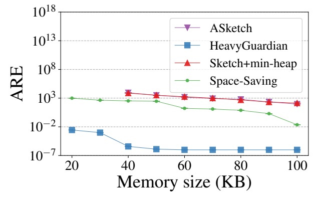

(b) Web page

Figure 10: ARE vs. memory size (heavy hitter detection).

Recall vs. memory size (Figure 9(a), 9(b)): The results show that when memory size is varied from 40KB to 100KB, the recall of HeavyGuardian is almost 100% for both two datasets. However, when the memory size is 40KB, the recall is only 28.6% and 58.8% for Space-Saving, is 15.6% and 32.4% for sketch+min-heap, and is 27.1% and 32.4% for ASketch for the two datasets, respectively.

ARE vs. memory size (Figure 10(a), 10(b)): The results show that for the CAIDA dataset, ARE of HeavyGuardian is between 62,381 and 2,364,154 times smaller than that of sketch+min-heap, is between 330,658 and 4,023,510 times smaller than that of Space-Saving, and is 776,347 and 1,321,717 times smaller than that of ASketch. For the Web Page dataset, ARE of HeavyGuardian is between 2,084,227 and 828,730,985 times smaller than that of sketch+min-heap, is between 23,294 and 225,262,199 times smaller than that of Space-Saving, and is between 2,271,724 and 93,006,636 times smaller than that of ASketch.

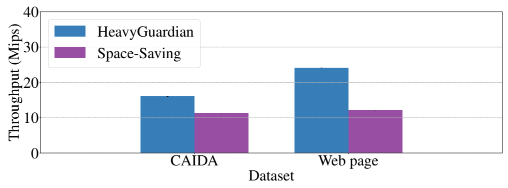
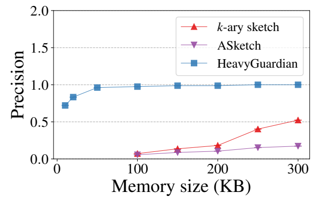
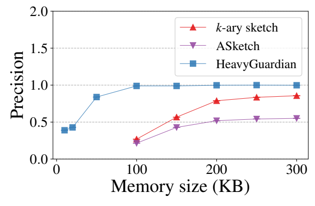


Figure 11: Speed evaluation (heavy hitter detection).

Throughput vs. dataset (Figure 11): In each dataset, the throughput of HeavyGuardian is significantly 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. The throughput of other algorithms is presented in the previous section.

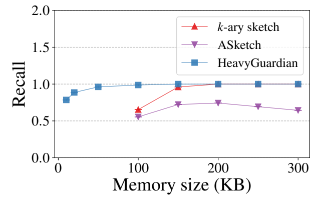


(a) CAIDA

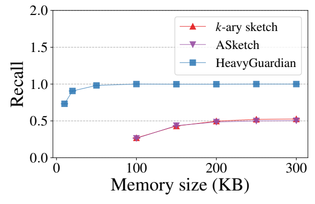


(b) Web page

Figure 12: Precision vs. memory size (heavy change detection).

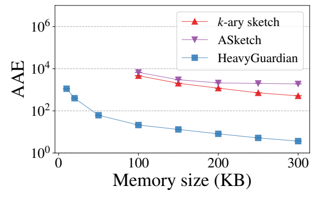


(a) CAIDA

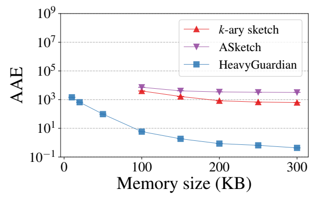


(b) Web page

Figure 13: Recall vs. memory size (heavy change detection).



(a) CAIDA



(b) Web page

Figure 14: AAE vs. memory size (heavy change detection).

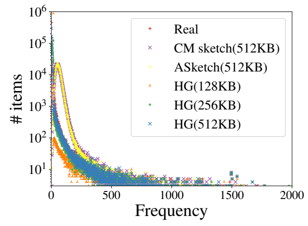
II. Heavy Change Detection

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

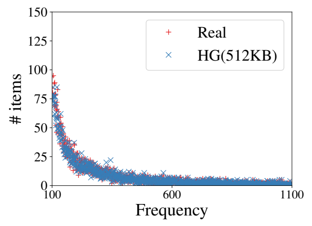
Precision vs. memory size (Figure 12(a), 12(b)): The results show that when memory size is varied from 100KB to 300KB, the precision of HeavyGuardian changes from 97.5% to 100% for the CAIDA dataset, and changes from 98.9% to 100% for the Web Page dataset. While for the *k-ary* sketch, the precision changes from 7.2% to 52.3% for the CAIDA dataset, and changes from 26.9% to 85.7% for the Web Page dataset. For ASketch, the precision is between 5.6% and 17.3% for the CAIDA dataset, and is between 21.5% and 55.1% for the Web Page dataset.

Recall vs. memory size (Figure 13(a), 13(b)): When memory size is larger than 50KB, the recall of HeavyGuardian is almost 100% for both two datasets. However, for the *k-ary* sketch, when memory size is 100KB, the recall is only 65.3% and 26.9% for the two datasets. For ASketch, the recall is 55.4% and 26.4% for the two datasets when the memory size is 100KB.

AAE vs. memory size (Figure 14(a), 14(b)): For the the CAIDA dataset, AAE of HeavyGuardian is between 137.38 and 218.82 times smaller than that of the *k-ary* sketch, and is between 368.66 and 517.95 times smaller than ASketch. For the Web Page dataset, AAE of HeavyGuardian is between 628.45 and 1448.35 times smaller than that of the *k-ary* sketch, and is between 1235.20 and 7363.74 times smaller than ASketch.

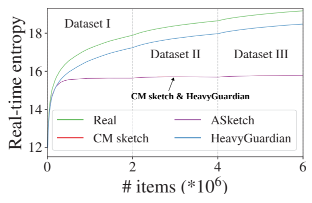


(a) Distribution (1 ~ 2000)

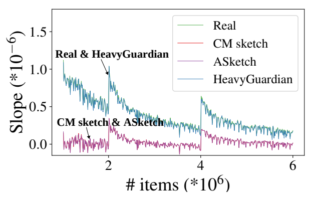


(b)Distribution (100 ~ 1100)

Figure 15: Real-time frequency distribution.



(a) Real-time entropy estimation



(b) The slope of real-time entropy

Figure 16: Real-time entropy distribution.

The insertion 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.

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

Due to space limitation, We only show the experimental results for real-time frequency distribution estimation and real-time entropy estimation on the CAIDA dataset. Because there is no existing algorithm achieving real-time frequency distribution and real-time entropy estimation, we only compare HeavyGuardian with a **naive algorithm:** use a CM sketch or an ASketch to record frequencies of all items, and update the frequency distribution and the entropy based on counters in the CM sketch or the ASketch in real time.

I. Real-time Frequency Distribution Estimation: In the experiments of real-time frequency distribution estimation, we set the memory size to 128KB, 256KB, and 512KB for HeavyGuardian, and 512KB for the CM sketch and ASketch.

Frequency distribution (Figure 15(a) and 15(b)): As shown in Figure 15(a), as the memory size increases, the estimated frequency distribution of HeavyGuardian (HG) is closer to the real frequency distribution, while that of the CM sketch and ASketch suffers from large errors. As shown in Figure 15(b), we limit the frequency from 100 to 1000, and we only compare the real frequency with the estimated frequency of HeavyGuardian 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.

II. Real-time Entropy Estimation: For experiments of real-time entropy estimation, we select 3 segments in one CAIDA dataset (2M items in each segment, and we call them Segment I, II, and III), and combine them together as one big dataset. Note that these 3 segments are randomly picked from three different time points.

Real-time entropy (Figure 16(a) and 16(b)): As shown in Figure 16(a), when changing from one segment to another segment, both the estimated real-time entropy of HeavyGuardian and the real entropy increases more quickly than before, while for the CM sketch and ASketch, this change is not evident (entropies estimated by CM sketch and ASketch are almost the same). In order to make the change more evident, we plot the slope of the real-time entropy in Figure 16(b). The results show that the estimated slope of the real-time entropy of HeavyGuardian is almost the same as the real value, while the slope of the real-time entropy of the CM sketch and ASketch suffers from large errors. The results also show that ARE of the estimated entropy of HeavyGuardian is 0.036, while that of the CM sketch and ASketch is 0.133, which means that HeavyGuardian achieves 3.69 times smaller error than the CM sketch and ASketch.

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 quite a few solutions, but can hardly achieve high accuracy when memory is tight. To address this issue, we propose a novel data structure HeavyGuardian. The key idea is `separate-and-guard-hot`: intelligently separating hot items from cold items using a key technique named "exponential decay", and guarding the information of hot items while approximately recording the frequencies of cold items. We derive theoretical bound for HeavyGuardian, and validate it by experiments. Experimental results also 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.

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] Elisa Bertino. Introduction to data security and privacy. *Data Science and Engineering*, 1(3):125–126, 2016.

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

[5] Ben Chen, Zhijin Lv, Xiaohui Yu, and Yang Liu. Sliding window top-k monitoring over distributed data streams. *Data Science and Engineering*, 2(4):289–300, 2017.

[6] 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.

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

[8] 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, 2003.

[9] 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.

[10] Nan Tang, Qing Chen, and Prasenjit Mitra. Graph stream summarization: From big bang to big crunch. In *Proceedings of the 2016 International Conference on Management of Data*, pages 1481–1496. ACM, 2016.

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

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

[13] 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.

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

[15] 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.

[16] Yin Zhang, Matthew Roughan, Walter Willinger, and Lili Qiu. Spatio-temporal compressive sensing and internet traffic matrices. In *ACM SIGCOMM Computer Communication Review*, volume 39, pages 267–278. ACM, 2009.

[17] Theophilus Benson, Aditya Akella, and David A Maltz. Network traffic characteristics of data centers in the wild. In *Proceedings of the 10th ACM SIGCOMM conference on Internet measurement*, pages 267–280. ACM, 2010.

[18] Graham Cormode, Balachander Krishnamurthy, and Walter Willinger. A manifesto for modeling and measurement in social media. *First Monday*, 15(9), 2010.

[19] Dave Maltz. Unraveling the complexity of network management. 2009.

[20] Ilker Nadi Bozkurt, Yilun Zhou, Theophilus Benson, Bilal Anwer, Dave Levin, Nick Feamster, Aditya Akella, Balakrishnan Chandrasekaran, Cheng Huang, Bruce Maggs, et al. Dynamic prioritization of traffic in home networks. 2015.

[21] Jiecao Chen and Qin Zhang. Bias-aware sketches. *Proceedings of the VLDB Endowment*, 10(9):961–972, 2017.

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

[23] Graham Cormode and Shan Muthukrishnan. An improved data stream summary: the count-min sketch and its applications. *Journal of Algorithms*, 2005.

[24] Yang Zhou, Tong Yang, Jie Jiang, Bin Cui, Minlan Yu, Xiaoming Li, and Steve Uhlig. Cold filter: A meta-framework for faster and more accurate stream processing.

[25] 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.

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

[27] 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.

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

[29] Erik Demaine, Alejandro López-Ortiz, and J Munro. Frequency estimation of internet packet streams with limited space. *AlgorithmsâĂŤESA 2002*, 2002.

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

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

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

[33] 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*, 2004.

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

[35] 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.

[36] Shanshan Ying, Flip Korn, Barna Saha, and Divesh Srivastava. Treescope: finding structural anomalies in semi-structured data. *VLDB*, 2015.

[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] Ge Luo, Lu Wang, Ke Yi, and Graham Cormode. Quantiles over data streams: experimental comparisons, new analyses, and further improvements. *The VLDB Journal*, 25(4):449–472, 2016.

[39] Chun-Hung Cheng, Ada Waichee Fu, and Yi Zhang. Entropy-based subspace clustering for mining numerical data. In *Proceedings of the fifth ACM SIGKDD international conference on Knowledge discovery and data mining*, 1999.

[40] Zhetao Li, Baoming Chang, Shiguo Wang, Anfeng Liu, Fanzi Zeng, and Guangming Luo. Dynamic compressive wide-band spectrum sensing based on channel energy reconstruction in cognitive internet of things. *IEEE Transactions on Industrial Informatics*, 2018.

[41] Xian Li, Xin Luna Dong, Kenneth Lyons, Weiyi Meng, and Divesh Srivastava. Truth finding on the deep web: Is the problem solved? In *Proceedings of the VLDB Endowment*, volume 6, pages 97–108, 2012.

[42] Zhetao Li, Fu Xiao, Shiguo Wang, Tingrui Pei, and Jie Li. Achievable rate maximization for cognitive hybrid satellite-terrestrial networks with af-relays. *IEEE Journal on Selected Areas in Communications*, 36(2):304–313, 2018.

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

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

[46] Christian Henke, Carsten Schmoll, and Tanja Zseby. Empirical evaluation of hash functions for multipoint measurements. *SIGCOMM CCR.*, 2008.