WavingSketch: An Unbiased and Generic Sketch for Finding Top-k Items in Data Streams

Jizhou Li[*]
Peking University

Zikun Li[†]
Peking University

Yifei Xu[†]
Peking University

Shiqi Jiang[‡]
Peking University

Tong Yang[†][§]
Peking University

Bin Cui[†][¶]
Peking University

Yafei Dai[†]
Peking University

Gong Zhang[‖]
Theory Research Lab, Huawei, China

ABSTRACT

[1] Finding top-k items in data streams is a fundamental problem in data mining. Existing algorithms that can achieve unbiased estimation suffer from poor accuracy. In this paper, we propose a new sketch, WavingSketch, which is much more accurate than existing unbiased algorithms. WavingSketch is generic, and we show how it can be applied to four applications: finding top-k frequent items, finding top-k heavy changes, finding top-k persistent items, and finding top-k Super-Spreaders. We theoretically prove that WavingSketch can provide unbiased estimation, and then give an error bound of our algorithm. Our experimental results show that, compared with the state-of-the-art, WavingSketch has 4.50 times higher insertion speed and up to 9×10^6 times (2×10^4 times in average) lower error rate in finding frequent items when memory size is tight. For other applications, WavingSketch can also achieve up to 286 times lower error rate. All related codes are open-sourced and available at Github anonymously.

CCS CONCEPTS

• **Information systems → Data stream mining**; *Data structures.*

KEYWORDS

data stream mining, top-k item, unbiased estimation, waving counter

ACM Reference Format:
Jizhou Li, Zikun Li, Yifei Xu, Shiqi Jiang, Tong Yang, Bin Cui, Yafei Dai, and Gong Zhang. 2020. WavingSketch: An Unbiased and Generic Sketch for Finding Top-k Items in Data Streams. In *Proceedings of the 26th ACM SIGKDD Conference on Knowledge Discovery and Data Mining (KDD '20), August 23–27, 2020, Virtual Event, CA, USA.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3394486.3403208

1 INTRODUCTION

1.1 Background and Motivation

One of the most fundamental problems in approximate data stream mining is finding top-k items. Top-k is defined in terms of various metrics. Four kinds of top-k items under different metrics have attracted wide attention by researchers: 1) top-k *frequent items* [1–5], 2) top-k *heavy changes* [6–8], 3) top-k *persistent items* [5, 9], and 4) top-k *Super-Spreaders* [10]. *Frequent items* refer to items whose numbers of appearances exceed a predefined threshold. *Heavy changes* refer to items whose frequencies change drastically over two adjacent time windows. *Persistent items* are items which appear in more time windows than others. *Super-Spreaders* refer to sources that connect to a large number of distinct destinations. Although these four kinds of tasks have different metrics, we find that if an algorithm does well in finding frequent items, it can also well handle the other three tasks. The reason behind is discussed below. To find heavy changes in two adjacent time windows, we can first find all frequent items in both time windows respectively since heavy changes must be frequent items in at least one time window. Then heavy changes can be detected through calculate the absolute change in frequencies of those frequent items in the two time windows. Persistent items and frequent items are equivalent if the frequency of an item is defined as the number of time windows in which it appears. Similarly, for finding Super-Spreaders, we only need to view the number of destinations an item connects to as its frequency. In summary, if a data structure can accurately find frequent items, it can also well handle other three tasks. Therefore, we only consider finding frequent items in this section below. In practice, sketches, a kind of probabilistic data structure, have been widely used in finding top-k items, due to their memory efficiency and small error for estimating item frequencies.

[*]Shenzhen Graduate School, Peking University, China
[†]Department of Computer Science and technology, Peking University, China
[‡]School of Mathematical Sciences, Peking University, China
[§]PCL Research Center of Networks and Communications, Pengcheng Laboratory, Shenzhen, China (e-mail: yangtongemail@gmail.com)
[¶]National Engineering Laboratory for Big Data Analysis Technology and Application (PKU), China
[‖]Theory Research Lab, Huawei, China
[1]Jizhou Li, Zikun Li and Yifei Xu contribute equally to this paper, and they together with Shiqi Jiang complete this work under the guidance of the corresponding author: Tong Yang (yangtongemail@gmail.com).

Unbiased estimation is well acknowledged as a theoretically elegant and pragmatic property. Unbiased estimation of item frequencies is beneficial for several global estimation problems, such as global heavy hitters, global distribution, global entropy, *etc.* For example, to measure distributed data streams, one data structure is deployed for each data stream. If their estimations are biased, the error will accumulate when the data structures are aggregated. Further, unbiased approaches could stimulate theoretical progress in sketches. Although numerous sketches have been proposed, only a very few of them (Count-Min Sketch [11], Count Sketch [4]) have explicit and concise theory bounds and proofs, and most of the others show error bounds and proofs that are fairly complicated. One of the primary reasons for this is that their estimations are biased.

Among a large number of algorithms for finding frequent items [1, 4, 11–15], one recent work, Unbiased Space Saving (USS) [15], achieves unbiased estimation. While USS makes a great contribution in terms of unbiased estimation, its accuracy is relatively low, *i.e.*, its variance of estimation is large. As a result, when it is applied to other tasks (*e.g.*, finding heavy changes, persistent items, or Super-Spreaders), the inaccuracy of frequency estimation will incur large error for other tasks, which may be the reason why USS was only used for frequent items in the paper [15]. The design goal of our study is to devise an algorithm which is accurate, generic, and can provide unbiased estimation.

1.2 Our Proposed Approach

Towards the above design goal, we propose the WavingSketch in this paper. We use a simple example to show the key idea of WavingSketch. We use a counter and a list. The counter, namely the Waving Counter, provides an unbiased estimation of item frequencies and the list is used to store k' ($k' > k$) frequent items. For each incoming item e, we use a hash function $s(e)$ to hash e to $+1$ or -1, and then increase or decrease the Waving Counter by 1. Afterwards, we estimate e's frequency using the Waving Counter. If its frequency is larger than the smallest frequency in the list, we exchange them. Based on this idea, we use a group of Waving Counters and lists, and add additional fields in the list to achieve higher accuracy (see details in Section 3.1).

Below we explain the rationale of WavingSketch. The value of the Waving Counter fluctuates over time, and it is quite analogous to the waves in the sea. Where there is a heavy swell, there is probably a strong flow driving it. Therefore, we expect to catch a frequent item when the absolute value of the counter is fairly high. Specifically, given an incoming item, we use the Waving Counter to unbiasedly estimate its frequency. If the estimated frequency is large enough, it is with high probability that the incoming item is frequent enough to replace the least frequent item in the list.

WavingSketch has four advantages. First, WavingSketch can provide unbiased estimation, and the theoretical proofs are provided in Section 5.1. Second, our theoretical and experimental results show that the error of WavingSketch is much smaller than Space Saving and Unbiased Space Saving. Third, WavingSketch is generic. To verify this, we apply WavingSketch to four applications: finding frequent items, finding heavy changes, finding persistent items, and finding Super-Spreaders. Fourth, WavingSketch is fast. For each insertion or query, we only need to access one bucket and it often requires only one memory access.

Main Experimental Results: In finding frequent items, compared with USS, WavingSketch achieves 4.50 times higher insertion speed and up to 9×10^6 times (2×10^4 times in average) smaller error rate. In finding heavy changes, WavingSketch improves the F1 Score [16] by 8 times in average when using only 1/10 of the memory size of other algorithms and improves the insertion speed by up to 1.9 times. In finding persistent items, WavingSketch achieves up to 7.5 times higher insertion speed and up to 286 times smaller error rate. In finding Super-Spreaders, WavingSketch achieves up to 14 times lower error rate. All related codes are open-sourced and available at Github anonymously [17].

1.3 Contributions

- We propose the WavingSketch, which is accurate, generic, and can provide unbiased estimation.
- We theoretically prove that WavingSketch can provide unbiased estimation, and then give an error bound of our algorithm.
- We apply WavingSketch to four applications: finding frequent items, finding heavy changes, finding persistent items, and finding Super-Spreaders.
- We conduct extensive experiments, and the results show that WavingSketch achieves up to two orders of magnitude smaller error than the state-of-the-art.

2 RELATED WORK

In this section, we only show the related algorithms for the four typical top-k tasks, for other related work and applications please refer to [18–28].

2.1 Finding Frequent Items

To find frequent items, two types of solutions exist. The first, *sketch-based algorithms*, record the frequencies of all items by hashing, but do not solve the hash collision. The second, *KV-based algorithms*, record $< ID, frequency >$ pairs of a subset of items that have a large frequency.

Sketch-based Algorithms: Typical sketches include the CM [11], CU [12], Count [4], and ASketch [29]. These sketches often consist of multiple arrays, each containing many counters. Each array is associated with a hash function that maps items to the counters. Hash collision may lower their accuracy, so they use some methods to reduce the error, however these methods are usually memory inefficient. This is because they also record relatively unimportant small items. Furthermore, sometimes multiple memory accesses per insertion decrease their throughput.

KV-based Algorithms: Typical KV-based algorithms include Space Saving [13, 30], Unbiased Space Saving [15], Lossy Counting [14], HeavyGuardian [31], and Cold filter [24]. Space Saving and Unbiased Space Saving use a data structure called Stream-Summary to record frequent items. When the data structure is full, and an item that is not recorded in the data structure arrives, the least frequent item will be replaced by the new item. The state-of-the-art, Unbiased Space Saving, achieves unbiased estimation by replacing the least frequent item with a certain probability. Although the average estimated frequency is unbiased, it considers the frequencies of all non-recorded items as 0. This means that, the estimation of all non-recorded items are heavily biased downward, and the estimation of all recorded items are heavily biased upward. Also, for each insertion, many pointer operations reduce the throughput of Space Saving and Unbiased Space Saving.

2.2 Finding Heavy Changes

To find heavy changes, there are two kinds of algorithms. One is *record all*. This kind of algorithms build a data structure to record all items in each period, and then decode and report heavy changes. Typical algorithms include k-ary [7], the reversible sketch [6], and FlowRadar [8]. The other kind of solutions only records frequent items. A typical algorithm is Cold filter [24]. Both solutions are not memory efficient because their data structures are not compact.

2.3 Finding Persistent Items

Again, two types of solutions exist. The first, *record all*, records all items. A typical algorithm is PIE [9] with Raptor codes [32] to generate different fingerprints in different periods. However, due to the recording of infrequent items and collisions, the accuracy of PIE is low. The second kind, called *record samples*, samples and removes duplicates before recording items. A typical algorithm is Small-Space [33]. Indeed, sampling can save memory, but the incurred error is hard to reduce.

2.4 Finding Super-Spreaders

There are two kinds of solutions. The first is *record all*. A typical algorithm is OpenSketch [34], which combines the techniques of CM sketches (presented above) and bitmaps. It is accurate when large amount of memory is used. The second kind is *record samples*. Sampling can automatically filter many infrequent items to save memory. The typical algorithms are called one-level filtering and two-level filtering [10]. Similarly, sampling makes error hard to reduce.

3 THE WAVINGSKETCH ALGORITHM

In this section, we first present the data structure of WavingSketch, then we show its operations. We list the symbols frequently used in this paper and their meanings in Table 1.

Table 1: Symbols frequently used in this paper.

Notation	Meaning
$\mathcal{S}$	a data stream
e_i	i_{th} distinct item in $\mathcal{S}$
f_i	frequency of item e_i
$\hat{f}_i$	estimated frequency of item e_i
$B[i]$	i_{th} bucket of WavingSketch
$B[i].count$	Waving Counter of $B[i]$
$B[i].heavy$	Heavy Part of $B[i]$
$B[i].v[e_i]$	recorded value of e_i in $B[i].heavy$
l	number of buckets in WavingSketch
d	number of cells in $B[i].heavy$
$h(.)$	hash function from items to $\{1,\ldots,l\}$
$s(.)$	hash function from items to $\{+1,-1\}$

3.1 Data Structure

Data Stream: a data stream $\mathcal{S}$ is a series of items, each of which could appear more than once. The number of appearances of an item e is called e's *frequency*.

Data Structure (Figure 1): The data structure of WavingSketch is an array which consists of l buckets. Let $B[i]$ be the i^{th} bucket. Each item e_i in the data stream is mapped into one bucket $B[h(e_i)]$ through a hash function $h(.)$. Each bucket consists of two parts: a Waving Counter $B[i].count$, and a Heavy Part $B[i].heavy$. We use another hash function $s(.)$ to map each incoming item to $\{+1,-1\}$. For each item e mapped into bucket $B[i]$, it will be recorded in one or both of the two parts. The Heavy Part consists of d cells. Each cell is used to store a key-value (KV) pair and a flag $<ID,\ frequency,\ flag>$. The key is the item ID, the value is its estimated frequency, and the flag indicates whether the frequency has error. The Waving Counter provides an unbiased estimation for frequencies of items that are inserted into it.

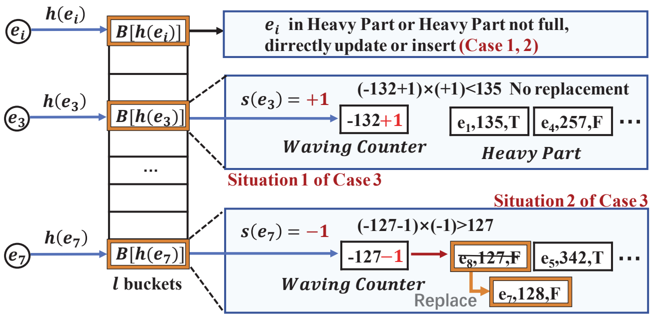


Figure 1: Data structure and insertion examples of WavingSketch.

3.2 Operations of WavingSketch

Initialization: All fields in the data structure are initially 0 or *null*.

Insertion (Figure 1): The pseudocode of the insertion operation is shown in Algorithm 1 in appendix A. Given an incoming item e_i, we first compute the hash function $h(e_i)$ to map e_i to bucket $B[h(e_i)]$. Below we show how to insert e_i into $B[h(e_i)]$. There are three cases as follows:

Case 1: (See lines 2 to 5 in Algorithm 1). If e_i is already recorded in $B[h(e_i)].heavy$, there are two situations. 1) e_i is recorded with flag of $true$, we just increment the corresponding frequency in the Heavy Part; 2) e_i is recorded with flag of $false$, we not only increment the corresponding frequency, but also add $B[h(e_i)].count$ by $s(e_i)$ ($s(e_i) \in \{-1,+1\}$).

Case 2: If e_i is not recorded in $B[h(e_i)].heavy$ and $B[h(e_i)].heavy$ is not full, we just insert $<e_i,\ 1,\ true>$ into $B[h(e_i)].heavy$ (See lines 6-7 in Algorithm 1).

Case 3: (See lines 8 to line 15 in Algorithm 1). If $B[h(e_i)].heavy$ is full and the e_i is not recorded in $B[h(e_i)].heavy$, there are two situations. Let $\hat{f}_i$ be $B[h(e_i)].count * s(e_i)$. 1) If $\hat{f}_i$ is smaller than the smallest counter in $B[h(e_i)].heavy$, we insert e_i into $B[h(e_i)].count$, *i.e.*, add $B[h(e_i)].count$ by $s(e_i)$; 2) If $\hat{f}_i$ is not smaller than the smallest counter in $B[h(e_i)].heavy$, after inserting e_i into $B[h(e_i)].count$, we replace the item with the smallest counter. Specifically, we replace the ID field of that cell with e_i, set the frequency field to

$\hat{f}_i + 1$ and set the flag field to $false$. If the flag of the replaced item e_r is $true$, then e_r is inserted into $B[h(e_i)].count$, *i.e.*, adding $B[h(e_i)].count$ by $v_r * s(e_r)$, where v_r is the value of the frequency field before replacement.

Below we use two examples to show how WavingSketch deals with Case 3 (See Figure 1).

Example 1: When e_3 arrives, it is mapped to bucket $B[h(e_3)]$, and $s(e_3) = +1$. e_3 is not in $B[h(e_3)].heavy$ and $B[h(e_3)].heavy$ is full, thus $\hat{f}_3 = B[h(e_3)].count \cdot s(e_3) = -132$. Because $\hat{f}_3$ is smaller than the smallest frequency 135 in $B[h(e_3)].heavy$, we insert e_3 to $B[h(e_3)].count$ and set it to $-132 + s(e_3) = -131$.

Example 2: When e_7 arrives, it is mapped to bucket $B[h(e_7)]$, and $s(e_7) = -1$. e_7 is not in $B[h(e_7)].heavy$ and $B[h(e_7)].heavy$ is full, thus $\hat{f}_7 = B[h(e_7)].count \cdot s(e_7) = 127$. Because $\hat{f}_7$ is not smaller than the smallest frequency 127 in $B[h(e_7)].heavy$, we insert e_7 to $B[h(e_3)].count$, replace the ID field of that cell with e_7, set the frequency field to $\hat{f}_7 + 1 = 128$, and set the flag field to $false$.

Unbiased Estimation: To give an unbiased estimation of an item e_i, we traverse the Heavy Part of $B[h(e_i)]$. If e_i is in the Heavy Part with flag of $true$, we report the frequency field as its frequency. Otherwise, we report the value of the Waving Counter as its frequency.

Top-k Query: For top-k queries, we only focus on the items stored in the Heavy Part, and report the frequency field as its frequency. Essentially, WavingSketch is similar to Count Sketch+Minheap (Count+Heap for short): they both identify frequent items based on unbiased estimation of item frequencies. Although they both cannot prove the unbiasedness when reporting frequent items, the bias should be very small. However, WavingSketch is faster and much more accurate than Count+Heap (see Figure 2-5).

4 APPLICATIONS

In this section, we apply WavingSketch to four applications: finding frequent items (Section 4.1), finding heavy changes (Section 4.2), finding persistent items (Section 4.3), and finding Super-Spreaders (Section 4.4). The settings of each application are shown in Table 2.

4.1 Finding Frequent Items

Problem Statement: Given a data stream $\mathcal{S}$ with N distinct items $(e_1, e_2..., e_N)$, find all items that have top-k largest frequencies.

Data Structure and Insertion: Because WavingSketch can be directly used to find frequent items, the data structure and insertion process are the same as presented in Section 3.2.

Report: We simply traverse the bucket array and return the IDs of items that have top-k largest frequencies.

Analysis: WavingSketch has the following advantages. First, this data structure has high memory efficiency, since it uses no pointer and almost has no empty cells. Second, each insertion only need to access one bucket. And this insertion can be accelerated through SIMD instructions [35]. After using SIMD, the access time of a bucket is similar to the time of one memory access. Third, WavingSketch can achieve significantly smaller error than the unilateral accumulation of SS [13] and USS [15], which is proved in experiments (see Section 6).

4.2 Finding Heavy Changes

Problem Statement: The data stream $\mathcal{S}$ is divided into two equal-sized periods: S_1 and S_2. Suppose that the frequency of e_i in S_1 is $f_i^{'}$ and the frequency of e_i in S_2 is $f_i^{''}$. We define $\left|f_i^{'} - f_i^{''}\right|$ as Δf_i. The problem consists in finding all items that have top-k largest Δf_i.

Data Structure: For each period, we build a WavingSketch to record frequent items, and compare the frequent items in adjacent periods to find heavy changes.

Insertion: For each input e_i, we insert e_i to a WavingSketch according to its period. The insertion process is the same as in Section 3.2.

Report: For two adjacent periods, we traverse all items in the data structures. For each item e_i, we query its frequency in the two WavingSketch, and get two frequencies, *i.e.*$f_i^{'}$ and $f_i^{''}$. We calculate Δf_i for each item, and report the items with top k largest Δf_i. Note that if an item does not appear in the Heavy Part of WavingSketch, the queried frequency is 0.

4.3 Finding Persistent Items

Problem Statement: The data stream $\mathcal{S}$ is divided into T equal-sized time windows. We define the persistency of an item as the number of time windows it appears in. The problem consists in finding all items that have top-k largest persistencies.

Data Structure: Our data structure consists of two parts. The first part is a Bloom filter[36] used to remove duplicates, because we need to check whether an item has probably appeared in the current window in finding persistent items.

Bloom filter: Bloom filter [36] is a compact data structure consisting of a number of bits and is often used to judge whether an item exists in a set. It has z hash functions. There are two operations for this data structure. One is insertion. The z hash functions are computed to pick z bits in the Bloom filter, and all the z bits are set to 1. The other is to judge whether an item is in the set. The same z hash functions are computed to get the z bits, and if all the z bits are 1, the Bloom filter reports $true$. If the item is indeed in the set, $true$ is always reported, *i.e.*, it has no false negative error. Though there might be false positives error, (*i.e.* items not in the set reported mistakenly to be in), the probability is often small enough to be acceptable in practice.

Insertion: Given an incoming item e_i, we first check the Bloom filter to judge whether e_i has appeared in this time window: if the Bloom filter reports true, which means e_i is a duplicate, then e_i is discarded. Otherwise, e_i is inserted into the Bloom filter, and then inserted into WavingSketch.

Periodical Emptying: Because we only remove duplicates inside a time window, we should empty the Bloom filter by setting all bits to 0 at the end of each time window.

Report: The process of reporting persistent items is the same as Section 4.1.

4.4 Finding Super-Spreaders

Problem Statement: Given a data stream with $<e_i, e_j>$ pairs, we define the connection of e_i as the number of distinct e_j it pairs with. The problem consists in finding all items that have top-k largest connections.

Table 2: Settings of WavingSketch for different applications.

Applications	Input	Remove Duplicates	Input of WavingSketch	Report
Frequent Items	Item e_i	×	e_i	top-k largest f_i
Heavy Changes	Item e_i	×	e_i	top-k largest Δf_i
Persistent Items	Item e_i	✓	e_i	top-k largest f_i
Super-Spreaders	Pair $<e_i, e_j>$	✓	e_i	top-k largest f_i

Data Structure: We need to check whether a pair has appeared for finding Super-Spreaders. Therefore, we need a Bloom filter[36] to remove duplicates in this application. As a result, the data structure is the same as finding persistent items.
Insertion: Given an incoming pair $<e_i, e_j>$, we first check the Bloom filter to judge whether the pair $<e_i, e_j>$ has appeared before: if the Bloom filter reports *true*, which means $<e_i, e_j>$ is a duplicate, then $<e_i, e_j>$ is discarded; Otherwise, $<e_i, e_j>$ is inserted into the Bloom filter, and then inserted into WavingSketch.
Report: The process of reporting Super-Spreaders is the same as Section 4.1.

5 MATHEMATICAL ANALYSIS

In this section, we provide a theoretical analysis for WavingSketch. First, we prove that our algorithm can provide an unbiased estimated frequency in Section 5.1. Then, we show the variance and the error bound of WavingSketch in Section 5.2. Due to space limitation, we show how the parameter of WavingSketch influences its performance in Appendix D.2.

5.1 Proof of Unbiasedness

In this section, we prove that for each item e_i, WavingSketch can provide an unbiased estimated frequency $\hat{f_i}$. If e_i is in the Heavy Part and is error-free (flag is $true$), $\hat{f_i}$ is the corresponding count in the Heavy Part. Otherwise, $\hat{f_i} = B[h(e_i)].count \cdot s(e_i)$.

Theorem 5.1. *The estimation of f_i is unbiased, i.e., $E(\hat{f_i}) = f_i$.*

Proof. For an item e_i, we prove that the expected increment to $\hat{f_i}$ is 1 if e_i is the next item and 0 otherwise. Let $\hat{f_i}'$ be the estimated frequency after the next item comes. We separately consider the four cases to analyze whether e_i is error-free and whether it's the next item.
Case 1: e_i is error-free and e_i is the next item.
Then the corresponding count in the Heavy Part is increased by 1, i.e., $B[h(e_i)].v[e_i]' = B[h(e_i)].v[e_i] + 1$. e_i is still in the Heavy Part and is error-free. Thus, we have $\hat{f_i}' = B[h(e_i)].v[e_i]' = \hat{f_i} + 1$.
Case 2: e_i is error-free and e_i is not the next item.
The corresponding count in the Heavy Part stays the same. If e_i is still in the Heavy Part, we have $\hat{f_i}' = B[h(e_i)].v[e_i]' = \hat{f_i}$. Otherwise, e_i is eliminated from the heavy part, then e_i is inserted into the Waving Counter and is no longer error-free. Then $B[h(e_i)].count' = B[h(e_i)].count + \hat{f_i} \cdot s(e_i)$. Thus, $\hat{f_i}' = B[h(e_i)].count' \cdot s(e_i) = B[h(e_i)].count \cdot s(e_i) + \hat{f_i} \cdot s(e_i)^2$. There is same chance for $s(e_i)$ to be 1 and -1, so $E(s(e_i)) = 0$. Since $s(e_i)$ and $B[h(e_i)].count$ are independent, we have $E(B[h(e_i)].count \cdot s(e_i)) = B[h(e_i)].count \cdot E(s(e_i)) = 0$. Thus, $E(\hat{f_i}') = E(B[h(e_i)].count \cdot s(e_i)) + \hat{f_i} = \hat{f_i}$.
Case 3: e_i is not error-free and e_i is the next item.
We have $\hat{f_i} = B[h(e_i)].count \cdot s(e_i)$ and $B[h(e_i)].count$ is added by $s(e_i)$. If no error-free item is removed from the Heavy Part, we have $\hat{f_i}' = (B[h(e_i)].count + s(e_i)) \cdot s(e_i) = \hat{f_i} + 1$. Otherwise, e_i replaces the item with the lowest count in the Heavy Part and that item is error-free. Let e_m be that item, and we have $B[h(e_i)].count' = B[h(e_i)].count + s(e_i) + \hat{f_m} \cdot s(e_m)$. Thus, our estimation satisfies $\hat{f_i}' = B[h(e_i)].count' \cdot s(e_i) = \hat{f_i} + 1 + \hat{f_m} \cdot s(e_m) \cdot s(e_i)$. Since, $\hat{f_m} \cdot s(e_m)$ and $s(e_i)$ are independent, we have $E(\hat{f_i}') = \hat{f_i} + 1 + E(\hat{f_m} \cdot s(e_m)) \cdot E(s(e_i)) = \hat{f_i} + 1$.
Case 4: e_i is not error-free and e_i is not the next item.
Let e_j be the next item. We have $\hat{f_i} = B[h(e_i)].count \cdot s(e_i)$. If e_j is error-free, then e_j does not influence the Waving Counter. Thus, $\hat{f_i}' = \hat{f_i}$. Otherwise, $B[h(e_i)].count$ is added by $s(e_j)$.
If no error-free item is removed from the Heavy Part, we have $\hat{f_i}' = (B[h(e_i)].count + s(e_j)) \cdot s(e_i) = \hat{f_i} + s(e_i) \cdot s(e_j)$. Since $s(e_i)$ and $s(e_j)$ are independent, $E(s(e_i) \cdot s(e_j)) = E(s(e_i)) \cdot E(s(e_j)) = 0$. Thus, $E(\hat{f_i}') = \hat{f_i} + E(s(e_i) \cdot s(e_j)) = \hat{f_i}$. Otherwise, e_j replaces the item with the lowest count in the Heavy Part and that item is error-free. Let e_m be that item, and we have $B[h(e_i)].count' = B[h(e_i)].count + s(e_j) + \hat{f_m} \cdot s(e_m)$. Thus, our estimation satisfies $\hat{f_i}' = B[h(e_i)].count' \cdot s(e_i) = \hat{f_i} + s(e_j) \cdot s(e_i) + \hat{f_m} \cdot s(e_m) \cdot s(e_i)$. As proved before, we have $E(s(e_j) \cdot s(e_i)) = 0$ and $E(\hat{f_m} \cdot s(e_m) \cdot s(e_i)) = 0$. Thus, $E(\hat{f_i}') = \hat{f_i} + E(s(e_j) \cdot s(e_i)) + E(\hat{f_m} \cdot s(e_m) \cdot s(e_i)) = \hat{f_i}$.

Therefore, we've proved that the expected increment to $\hat{f_i}$ is 1 if e_i is the next item and 0 otherwise, which indicates that we always have $E(\hat{f_i}) = f_i$. In other words, our estimation is unbiased. □

5.2 Variance and Error Bound

We show the variance and the error bound of our estimation for each item e_i in Theorem 5.2 and 5.3. Due to space limitation, the details of proofs are provided in Appendix D.1.

Theorem 5.2. *Let $e_1, e_2, \cdots, e_n$ be the items inserted to $B[h(e_i)]$. We can get the bound of the variance of our estimation*

$$Var(\hat{f_i}) \leqslant \sum_{e_j \neq e_i} (f_j)^2 \tag{1}$$

Theorem 5.3. *Let $l = \frac{e}{\epsilon^2}$, then $P\left(\left|\hat{f_i} - f_i\right| \geqslant \epsilon \left\|f\right\|_2\right) \leqslant \frac{1}{e}$*

6 EXPERIMENTAL RESULTS

In this section, we provide experimental results of WavingSketch. We describe the experiment setup in Section 6.1. Since finding frequent items is the basis of the other three applications, we show how WavingSketch performs in this application compared with prior algorithms in Section 6.2, and show how parameter settings and data distribution can affect its performance in Section 6.3 and Section 6.4. Finally, we show how WavingSketch performs in other applications compared with prior algorithms in Section 6.5. All abbreviations used in the evaluation and their full name are shown in Table 3.

Table 3: Abbreviations in experiments.

Abbreviation	Full name
Count+Heap	Count Sketch[4] with a heap
SS	Space Saving[13]
USS	Unbiased Space Saving[15]
FR	FlowRadar[8]
CF	Cold Filter[24]
OLF	One-level Filtering[10]
TLF	Two-level Filtering[10]
WavingSketch	The WavingSketch in Section 3
WavingSketch_C	WavingSketch for heavy changes
WavingSketch_P	WavingSketch for persistent items
WavingSketch_S	WavingSketch for Super-Spreaders

6.1 Experimental Setup

Implementation: We have implemented WavingSketch and all other algorithms in C++. The hash functions are implemented using the 32-bit Bob Hash (obtained from the open-source website [37]) with different initial seeds.

Datasets: We use four kinds of datasets: 1) Synthetic Datasets; 2) IP Trace Dataset; 3) Web Page Dataset; 4) Network Dataset. The details are shown in appendix B.

Computation Platform: We conduct all the experiments on a machine with two 6-core processors (12 threads, Intel Xeon CPU E5-2620 @2 GHz) and 64 GB DRAM memory. Each processor has three levels of cache memory: one 32KB L1 data caches and one 32KB L1 instruction cache for each core, one 256KB L2 cache for each core, and one 15MB L3 cache shared by all cores.

Metrics:

1) Average Relative Error (ARE): $\frac{1}{|\Psi|}\sum_{e_i\in\Psi}|f_i-\hat{f_i}|/f_i$, where f_i is the real frequency of item e_i, $\widehat{f_i}$ is its estimated frequency, and Ψ is the query set. In the experiments, we query the dataset by querying each actually frequent item once in the sketch.

2) Recall Rate (CR): The ratio of the number of correctly reported items to the number of correct items.

3) Precision Rate (PR): The ratio of the number of correctly reported items to the number of reported items.

4) F1 Score: $\frac{2*CR*PR}{CR+PR}$

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

6.2 Experiments on Finding Frequent Items

Parameter settings: See details in appendix C.1.

Comparison with prior algorithms: We compare WavingSketch with 3 algorithms: Count+Heap[4], USS[15], and SS[13]. We choose Count+Heap because it is a typical unbiased sketch-based algorithm. We choose USS because it is a typical unbiased KV-based algorithm. And we choose SS because it is classic and has been shown in [30] to be better than many algorithms, like Lossy Counting[14] and Frequent[1].

ARE (Figure 2(a)-2(d)): We find that, on the synthetic dataset, the ARE of WavingSketch is around 7×10^{-5} under a memory of 200KB. On the three real-world datasets, the ARE of WavingSketch is around 1565, 21737, and 19802 times lower than Count+Heap, USS, and SS, respectively.

CR (Figure 3(a)-3(d)): We find that, on the synthetic dataset, the CR of WavingSketch is around 1.31, 1.39, and 1.38 times higher than Count+Heap, USS, and SS, respectively. On the three real-world datasets, the CR of WavingSketch is around 1.25, 1.35, and 1.41 times higher than Count+Heap, USS, and SS, respectively.

PR (Figure 4(a)-4(d)): On the synthetic dataset, the PR of WavingSketch is around 1.61, 3.68, and 3.67 times higher than Count+Heap, USS, and SS, respectively. On the three real-world datasets, the PR of WavingSketch is around 1.54, 2.33, and 2.39 times higher than Count+Heap, USS, and SS, respectively.

Throughput (Figure 5(a)-5(d)): On the synthetic dataset and the three real-world datasets, the insertion throughput of WavingSketch is around 4.05, 4.50, and 2.50 times higher than Count+Heap, USS, and SS, respectively.

Analysis. 1) The ARE of SS and USS is much higher than WavingSketch, because the recorded items' frequency tends to be greatly overestimated in SS and USS. The ARE of Count+Heap is higher than WavingSketch because Count+Heap needs to keep a large Count Sketch to ensure the accuracy. So when the memory is very limited, its accuracy will be much worse than WavingSketch. 2) The throughput of WavingSketch is much higher than that of prior algorithms, because WavingSketch only needs one memory access for each insertion. In SS and USS, the pointer operations will lead to cache misses and make the throughput much lower. In Count+Heap, multiple accesses to memory and the $O(\log k)$ time complexity of the heap operations slow down the insertion throughput. 3) We find that the PR of SS and USS sometimes decreases as memory consumption increases. This is common for algorithms that overestimate the frequency. For example, if the only 200 items are recorded, there are at most 200 items whose estimated frequency exceeds the predefined threshold. However, if 2000 items are recorded, there may be 1000 items whose estimated frequency exceeds the predefined threshold due to overestimation, which leads to a decrease in PR.

6.3 Experiments on Parameter Settings

Parameter settings: See details in appendix C.3.

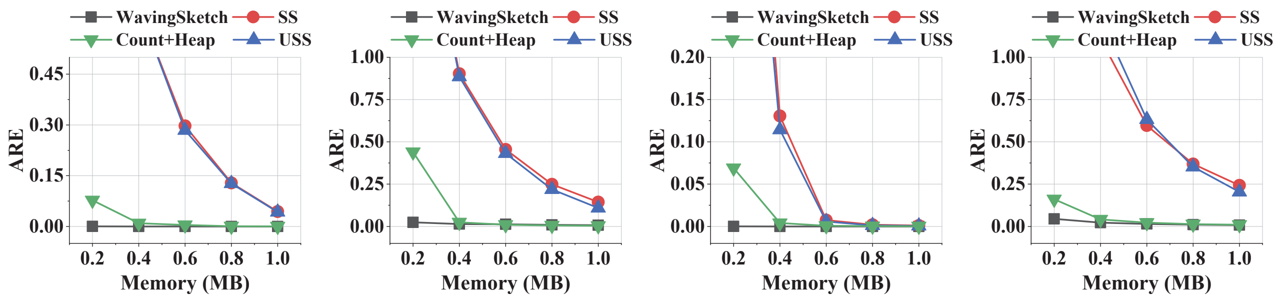
ARE (Figure 6(a)): We find that the ARE of $d = 8$ is around 1.16 times higher than that of $d = 16$ and is almost equal to that of $d = 32$, and $d = 64$.

CR (Figure 6(c)): We find that the CR of $d = 8$ is almost equal to that of $d = 16$ and is around 1.01 and 1.03 times higher than that of $d = 32$ and $d = 64$, respectively.

PR (Figure 6(b)): We find that the PR of $d = 8$ is around 1.04, 1.06, and 1.06 times lower than that of $d = 16$, $d = 32$, and $d = 64$, respectively.

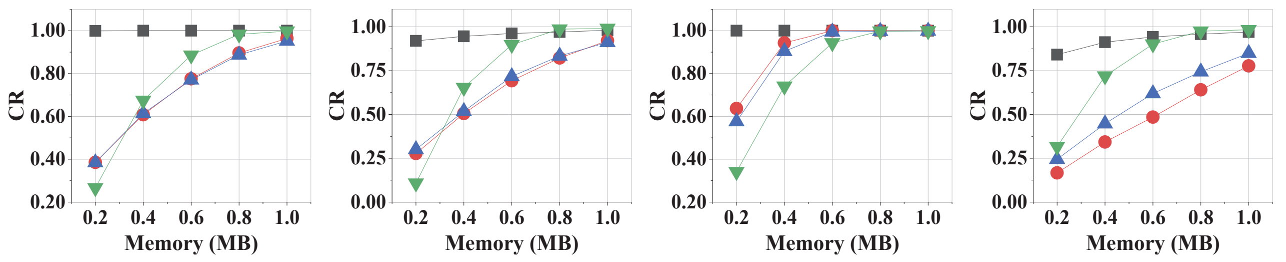
Throughput (Figure 6(d)): We find that the throughput of $d = 8$ is around 1.07, 1.17, and 1.39 times higher than that of $d = 16$, $d = 32$, and $d = 64$, respectively.

Analysis: According to the results, given an amount of memory, a higher value of d typically goes with a higher precision of WavingSketch and a lower throughput, other aspect of performance is not influenced explicitly. In other words, the value of d is selected based on a trade-off between precision and throughput. If the application requires a higher speed, we can use a smaller value of d. If the application requires a higher precision, we can use a larger value of d.



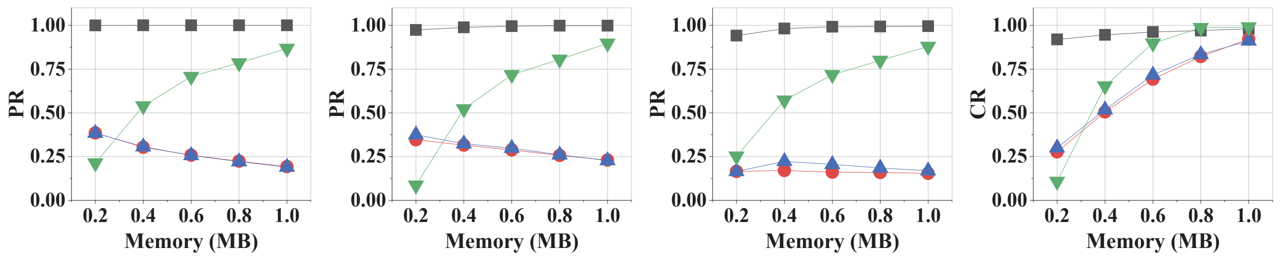
(a) Synthetic (b) IP trace (c) Web page (d) Network

Figure 2: ARE of finding frequent items.



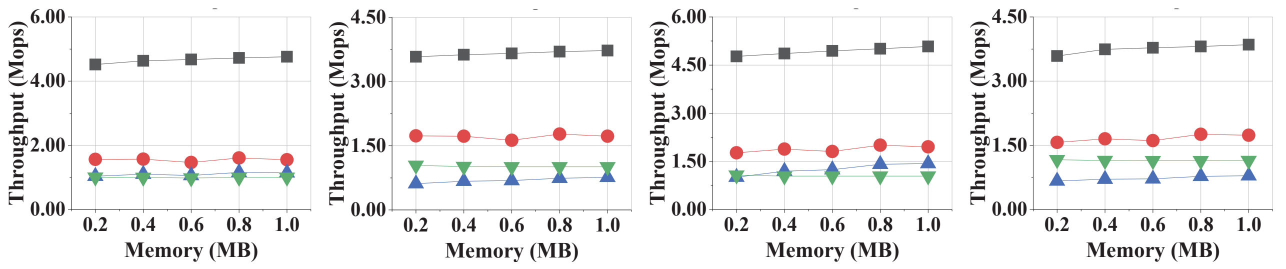
(a) Synthetic (b) IP trace (c) Web page (d) Network

Figure 3: CR of finding frequent items. The legend is the same as that of Figure 2.



(a) Synthetic (b) IP trace (c) Web page (d) Network

Figure 4: PR of finding frequent items. The legend is the same as that of Figure 2.



(a) Synthetic (b) IP trace (c) Web page (d) Network

Figure 5: Throughput of finding frequent items. The legend is the same as that of Figure 2.

6.4 Experiments on Distributions

Parameter Setting: See details in appendix C.4.

ARE (Figure 7(a)): We find that the ARE of a skewness of 0.3 is around 2.5 and 5.1 times higher than that of a skewness of 0.6 and 3 respectively.

F1 Score (Figure 7(b)): We find that the F1 Score of a skewness of 0.3 is around 1.53 and 2.18 times lower than that of a skewness of 0.6 and 3 respectively.

Analysis: According to the results, WavingSketch can achieve higher accuracy under higher skewness of the dataset.

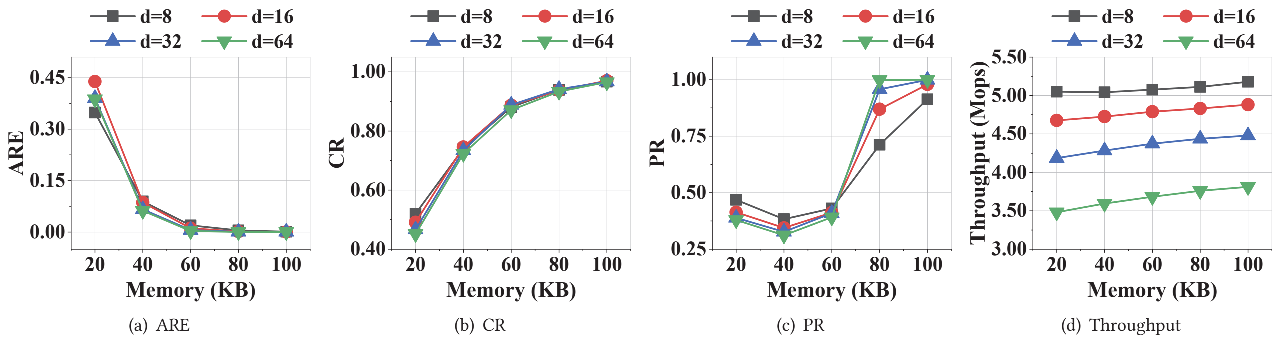


Figure 6: Evaluation on Parameter Setting.

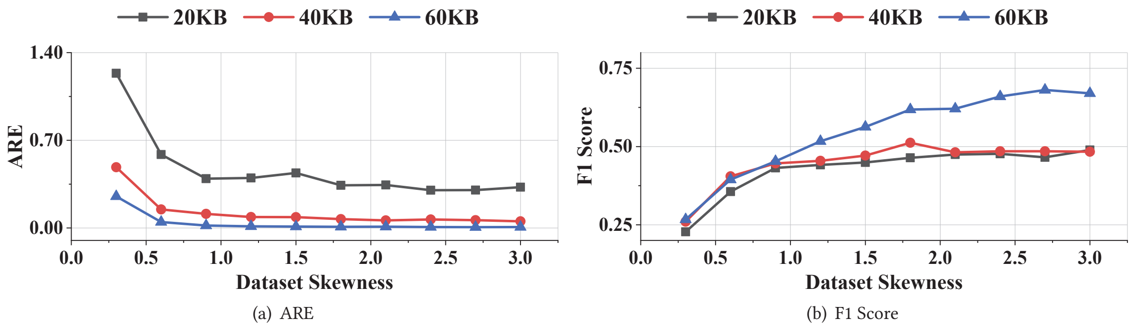


Figure 7: Evaluation on Distributions.

6.5 Experiments on other applications

Comparison with prior algorithms: For finding heavy changes, we compare WavingSketch_C with FR[8] and FR with CF[24]. For finding persistent items, we compare WavingSketch_P with PIE[9] and Small-Space[33]. For finding Super-Spreaders, we compare WavingSketch_S with OLF[10], TLF[10], and OpenSketch[34].

Parameter settings: See details in appendix C.2.

Finding Heavy Changes (Figure 8(a)-8(b)): We find that the F1 Score of WavingSketch_C is around 10 times and 6 times higher than FR and FR with CF, respectively. The throughput of WavingSketch_C is around 1.35 and 1.92 times higher than FR and FR with CF, respectively.

Finding Persistent Items (Figure 9(a)-9(b)): We find that the F1 Score of WavingSketch_P is around 3.71 and 4.35 times higher than PIE and Small-Space, respectively. The throughput of WavingSketch_P is close to Small-Space, and is around 7.55 times higher than PIE.

Finding Super-Spreaders (Figure 10(a)-10(b)): We find that the F1 Score of WavingSketch_S is around 22.18, 17.73, and 1.21 times higher than OLF, TLF, and OpenSketch, respectively. The throughput of WavingSketch_S is lower than OLF and TLF, but higher than OpenSketch.

Analysis. 1) The F1 Score shows that WavingSketch_C greatly outperforms FR and FR+CF while using merely $\frac{1}{10}$ memory they use. This is because finding heavy changes requires frequency of items. Since WavingSketch provides more accurate estimation, it also performs better in this task. 2) On finding persistent items, the F1 Score of WavingSketch_P is much better than prior algorithms. For Small-Space, sampling can enhance the throughput, but the low sample rate under small memory magnifies the error. For PIE, though it uses 200 times memory as WavingSketch_P, hash collisions still

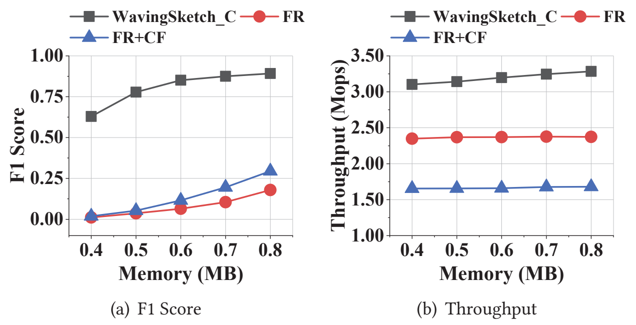


Figure 8: Evaluation on finding Heavy Changes.

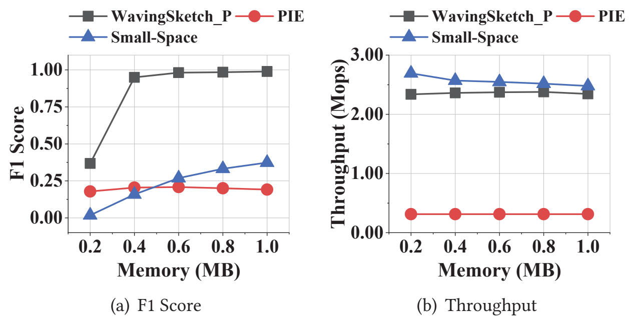
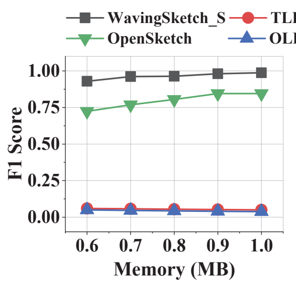
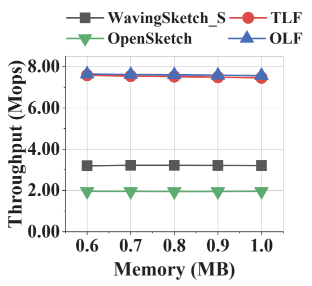


Figure 9: Evaluation on finding Persistent Items.



(a) F1 Score



(b) Throughput

Figure 10: Evaluation on finding Super-Spreaders.

lower its accuracy seriously. 3) On finding Super-Spreaders, though the Bloom filter requires a lot of memory, WavingSketch_C still greatly outperforms prior algorithms. The reason for this is the significant amount of memory required by prior algorithms to remove duplicates. For example, OpenSketch uses bitmaps, and OLF uses a hash-table. In contrast, Bloom filter is more effective.

7 CONCLUSION

In this paper, we propose an algorithm called WavingSketch for finding top-k items. It can provide unbiased estimation and outperform the state-of-the-art, Unbiased Space Saving in terms of accuracy and speed. We prove mathematically the unbiasedness and show that the error is much lower than that of the state-of-the-art, Unbiased Space Saving. Besides, WavingSketch is generic. We show how it can be applied to four applications: finding frequent items, finding heavy changes, finding persistent items, and finding Super-Spreaders. We conduct extensive experiments on three real-world and one synthetic datasets. Experimental results show that, compared with Unbiased Space Saving, WavingSketch achieves 4.50 times higher insertion speed in average and up to 9×10^6 times (2×10^4 times in average) lower error rate in finding frequent items.

ACKNOWLEDGMENT

This work is supported by National Natural Science Foundation of China (NSFC) (No. 61832001, 61702016, 61672061), National Key R&D Program of China (No. 2018YFB1004403), Huawei Technologies, and the project of "FANet: PCL Future Greater-Bay Area Network Facilities for Large-scale Experiments and Applications (No. LZC0019).

REFERENCES

[1] G. Lukasz, D. David, D. Erik D, L. Alejandro, and M. J Ian. Identifying frequent items in sliding windows over on-line packet streams. In *IMC*. ACM, 2003.
[2] Richard M Karp, Scott Shenker, and Christos H Papadimitriou. A simple algorithm for finding frequent elements in streams and bags. *ACM Transactions on Database Systems (TODS)*, 28(1):51–55, 2003.
[3] M. Nishad and P. Themis. Frequent items in streaming data: An experimental evaluation of the state-of-the-art. *Data & Knowledge Engineering*, 2009.
[4] Moses Charikar, Kevin Chen, and Martin Farach-Colton. Finding frequent items in data streams. In *Automata, Languages and Programming*. Springer, 2002.
[5] Zhewei Wei, Ge Luo, Ke Yi, Xiaoyong Du, and Ji-Rong Wen. Persistent data sketching. In *Proc. ACM SIGMOD*, pages 795–810. ACM, 2015.
[6] Robert Schweller, Zhichun Li, Yan Chen, et al. Reversible sketches: enabling monitoring and analysis over high-speed data streams. *IEEE/ACM Transactions on Networking (ToN)*, 15(5):1059–1072, 2007.
[7] K. Balachander, S. Subhabrata, Z. Yin, and C. Yan. Sketch-based change detection: methods, evaluation, and applications. In *Proceedings of the 3rd ACM SIGCOMM conference on Internet measurement*, pages 234–247. ACM, 2003.
[8] Yuliang Li, Rui Miao, Changhoon Kim, and Minlan Yu. Flowradar: a better netflow for data centers. In *USENIX NSDI*, pages 311–324. USENIX Association, 2016.
[9] D. Haipeng, S. Muhammad, L. Alex X, and Z. Yuankun. Finding persistent items in data streams. *Proc. VLDB*, 2016.
[10] S. Venkataraman, D. Xiaodong Song, P. B. Gibbons, and A. Blum. New streaming algorithms for fast detection of superspreaders. In *NDSS*, 2005.
[11] Graham Cormode and S Muthukrishnan. An improved data stream summary: the count-min sketch and its applications. *Journal of Algorithms*, 55(1), 2005.
[12] Cristian Estan and George Varghese. New directions in traffic measurement and accounting. *ACM SIGMCOMM CCR*, 32(4), 2002.
[13] 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*. Springer, 2005.
[14] M. Gurmeet Singh and M. Rajeev. Approximate frequency counts over data streams. In *Proc. VLDB*, pages 346–357, 2002.
[15] Daniel Ting. Data sketches for disaggregated subset sum and frequent item estimation. In *SIGMOD Conference*, 2018.
[16] Ming Ji, Jun Yan, Siyu Gu, Jiawei Han, Xiaofei He, Wei Vivian Zhang, and Zheng Chen. Learning search tasks in queries and web pages via graph regularization. In *Proc. ACM SIGIR*, pages 55–64. ACM, 2011.
[17] Source code related to WavingSketch. https://github.com/WavingSketch/Waving-Sketch.
[18] Graham Cormode. Sketch techniques for approximate query processing. *Foundations and Trends in Databases. NOW publishers*, 2011.
[19] Pinghui Wang, Yiyan Qi, Yuanming Zhang, Qiaozhu Zhai, Chenxu Wang, John CS Lui, and Xiaohong Guan. A memory-efficient sketch method for estimating high similarities in streaming sets. In *SIGKDD*, pages 25–33, 2019.
[20] Jiawei Jiang, Fangcheng Fu, Tong Yang, and Bin Cui. Sketchml: Accelerating distributed machine learning with data sketches. In *SIGMOD*. ACM, 2018.
[21] Tong Yang, Jie Jiang, Peng Liu, Qun Huang, Junzhi Gong, Yang Zhou, Rui Miao, Xiaoming Li, and Steve Uhlig. Elastic sketch: Adaptive and fast network-wide measurements. In *ACM SIGCOMM 2018*, pages 561–575, 2018.
[22] Tong Yang, Yang Zhou, Hao Jin, Shigang Chen, and Xiaoming Li. Pyramid sketch: A sketch framework for frequency estimation of data streams. *Proc. VLDB Endow.*, 10(11):1442–1453, August 2017.
[23] Tong Yang, Alex X Liu, Muhammad Shahzad, Yuankun Zhong, Qiaobin Fu, Zi Li, Gaogang Xie, and Xiaoming Li. A shifting bloom filter framework for set queries. *Proceedings of the VLDB Endowment*, 9(5):408–419, 2016.
[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. In *SIGMOD Conference*, 2018.
[25] Bofang Li, Aleksandr Drozd, and et al. Scaling word2vec on big corpus. *Data Science and Engineering*, pages 1–19, 2019.
[26] Stephen Bonner, Ibad Kureshi, and et al. Exploring the semantic content of unsupervised graph embeddings: An empirical study. *Data Science and Engineering*, 4(3):269–289, 2019.
[27] Yinghui Wang, Peng Lin, and Yiguang Hong. Distributed regression estimation with incomplete data in multi-agent networks. *Science China Information Sciences*, 61(9):092202, 2018.
[28] Tongya Zheng, Gang Chen, and et al. Real-time intelligent big data processing: technology, platform, and applications. *Science China Information Sciences*, 62(8):82101, 2019.
[29] R. Pratanu, K. Arijit, and A. Gustavo. Augmented sketch: Faster and more accurate stream processing. In *Proc. ACM SIGMOD*, 2016.
[30] C. Graham and H. Marios. Finding frequent items in data streams. *VLDB*, 2008.
[31] Tong Yang, Junzhi Gong, Haowei Zhang, Lei Zou, Lei Shi, and Xiaoming Li. Heavyguardian: Separate and guard hot items in data streams. In *SIGKDD*, 2018.
[32] A.Shokrollahi. Raptor codes. *IEEE Transactions Information Theory*, 52(6), 2006.
[33] Bibudh Lahiri, Jaideep Chandrashekar, and Srikanta Tirthapura. Space-efficient tracking of persistent items in a massive data stream. *Statistical Analysis and Data Mining*, 7:70–92, 2011.
[34] Minlan Yu, Lavanya Jose, and Rui Miao. Software defined traffic measurement with opensketch. In *NSDI 2013*, 2013.
[35] Michael Flynn. Some computer organizations and their effectiveness. ieee trans comput c-21:948. *Computers, IEEE Transactions on*, C-21:948 – 960, 10 1972.
[36] Burton H Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, 1970.
[37] Hash website. http://burtleburtle.net/bob/hash/evahash.html.
[38] David MW Powers. Applications and explanations of Zipf's law. In *Proc. EMNLP-CoNLL*. Association for Computational Linguistics, 1998.
[39] Alex Rousskov and Duane Wessels. High-performance benchmarking with web polygraph. *Software: Practice and Experience*, 2004.
[40] The caida anonymized 2016 traces. http://www.caida.org/data/overview/.
[41] Real-life transactional dataset. http://fimi.ua.ac.be/data/.
[42] The Network dataset Internet Traces. http://snap.stanford.edu/data/.

A PSEUDO CODE

Algorithm 1: Insertion of WavingSketch.

```
Input: An item $e_i$
$\hat{f}_i = B[h(e_i)].count * s(e_i)$;
if $e_i \in L_{e_i}$ then
    update the frequencies;
    if flag for $e_i$ is false then
        $B[h(e_i)].count \leftarrow B[h(e_i)].count + s(e_i)$;
else if $L_{e_i}$ is not full then
    insert the item with $<e_i,\ 1,\ true>$;
else
    $B[h(e_i)].count \leftarrow B[h(e_i)].count + s(e_i)$;
    $e_r \leftarrow L_{e_i}.smallestItem()$;
    $\hat{f}_r \leftarrow e_r.frequency$;
    if $\hat{f}_i \geq \hat{f}_r$ then
        if $e_r.flag = true$ then
            $B[h(e_i)].count \leftarrow B[h(e_i)].count + \hat{f}_r * s(e_r)$;
        $L_{e_i}.replaceSmallestItem(e_i, \hat{f}_i + 1, false)$;
return;
```

B DATASETS

1) Synthetic Datasets: We generate 10 synthetic datasets that follow the Zipf [38] distribution by using Web Polygraph [39], an open-source performance testing tool. Each dataset has 32 million items, and the skewness of datasets varies from 0.3 to 3.0. The length of each item ID is 4 bytes. The synthetic datasets can be used to experiment the influence of the distribution of datasets (skewness varying from 0.3 to 3.0). We also use the dataset with skewness of 1.5 as the synthetic dataset for experiments on the four applications, because this skewness provides an appropriate difficulty of distinguishing items by frequency.
2) IP Trace Dataset: The IP Trace Dataset is streams of anonymized IP traces collected in 2016 by CAIDA [40]. Each item contains a source IP address (4 bytes) and a destination IP address (4 bytes), 8 bytes in total.
3) Web Page Dataset: The Web page dataset is built from a collection of web pages, which were downloaded from the website [41]. Each item (4 bytes) represents the number of distinct terms in a web page.
4) Network Dataset: The network dataset contains users' posting history on the stack exchange website [42]. Each item has three values u, v, t, which means user u answered user v's question at time t. We use u as the ID.

C PARAMETER SETTINGS

C.1 Parameter Settings for Finding Frequent Items

Let d be the number of cells in the Heavy Part of a bucket. For WavingSketch, we set $d = 16$. For other sketches, the parameters are set according to the recommendation of their authors. The memory size ranges from 0.2MB to 1MB. We choose such a small memory for the following two reasons.

- When using sketches, it is often desired that they fit in the cache to make them fast enough.
- Sketches are often sent across the network, and the small size of sketches can significantly save the bandwidth.

C.2 Parameter Settings for Other Applications

For WavingSketch, we set $d = 16$, which means there are 16 cells in the Heavy Part. For other sketches, the parameters are set according to their authors' recommendations. For finding heavy changes, the memory size ranges from 4MB to 8MB, because FR cannot decode with less memory, and WavingSketch_C only uses $\frac{1}{10}$ the memory of FR and FR+CF. For finding persistent items, the memory size ranges from 0.2MB to 1MB. Because PIE cannot decode with small amounts of memory, it will use 200 times more memory as Small-Space and WavingSketch_P. For finding Super-Spreaders, the memory size ranges from 0.6MB to 1MB, because algorithms on this application often need more memory to remove duplicates. We use the IP Trace dataset to evaluate the performance of other applications because only IP Trace datasets can be used to find Super-Spreaders.

C.3 Parameter Settings for Experiments on Parameter Settings

As shown in Section 3, we have two parameters in WavingSketch: l and d. To evaluate the influence of parameter setting, we fix the memory usage of WavingSketch and vary the value of d. We use finding frequent items as the considered application in this section to avoid the influence brought by the Bloom filter. The memory size ranges from 20KB to 100KB, because such little memory better exposes the differences between different values of d. As shown in Section 6.2, WavingSketch performs better than prior algorithms with only 200KB memory. We vary d from 8 to 64 and use the synthetic dataset to evaluate the influence of the parameter setting.

C.4 Parameter Settings for Experiments on Distributions

To evaluate the impact of the item distribution, we use the synthetic datasets whose skewness ranges from 0.3 to 3.0. For WavingSketch, we set $d = 16$, which means there are 16 cells in the Heavy Part. The memory size ranges from 20KB to 60KB, because such little memory better exposes the difference between different distributions.

D MATHEMATICAL ANALYSIS

D.1 Variance and Error Bound

Here, we show the variance and the error bound of our estimation for each item e_i.

Theorem D.1. *Let $e_1, e_2, \cdots, e_n$ be the items inserted to $B[h(e_i)]$. We can get the bound of the variance of our estimation that*

$$Var(\hat{f}_i) \leqslant \sum_{e_j \neq e_i} (f_j)^2 \tag{2}$$

Proof. If e_i is error-free, then $\hat{f}_i = f_i$. Otherwise, we have $\hat{f}_i = \left(\sum_{e_j \in S_e} f_j \cdot s(e_j)\right) \cdot s(e_i)$, where S_e is the set of the items that are not error-free. According to 5.1, $E(\hat{f}_i) = f_i$, so we get the variance of

$\hat{f}_i$ that $Var(\hat{f}_i) = E_{s(e_j)\in\{1,-1\}}\left(\left(\sum_{e_j\in S_e, j\neq i} f_j \cdot s(e_j)\right)\cdot s(e_i)\right)^2 = E_{s(e_j)\in\{1,-1\}}\left(\sum_{e_j\in S_e, j\neq i} f_j \cdot s(e_j)\right)^2$

From the analysis in 5.1, we find that $s(e_i)$ and whether e_i is error-free is independent. Thus, the cross terms have the same chance to be 1 and -1, so the expectation of their sum is 0. Therefore, we have $Var(\hat{f}_i) = E_{s(e_j)\in\{1,-1\}}\left(\sum_{e_j\in S_e, j\neq i}(f_j)^2\right) \leqslant \sum_{e_j\neq e_i}(f_j)^2$ □

According to the variance, we can derive an error bound for $\|f\|_2$.

Theorem D.2. *Let* $l = \frac{e}{\epsilon^2}$*, then* $P\left(\left|\hat{f}_i - f_i\right| \geqslant \epsilon\|f\|_2\right) \leqslant \frac{1}{e}$

Proof. Based on Chebyshev's theorem, we can get that $P\left(\left|\hat{f}_i - f_i\right| \geqslant \sqrt{e\sum_{e_j\neq e_i}(f_j)^2}\right) \leqslant \frac{Var(\hat{f}_i)}{\left(\sqrt{e\sum_{e_j\neq e_i}(f_j)^2}\right)^2} \leqslant \frac{1}{e}$.

For items in $B[h(e_i)]$, we can have an estimation that $\sum_{e_j}(f_j)^2 = \frac{1}{l}(\|f\|_2)^2$. Therefore, we can get
$P\left(\left|\hat{f}_i - f_i\right| \geqslant \epsilon\|f\|_2\right) \leqslant P\left(\left|\hat{f}_i - f_i\right| \geqslant \epsilon\sqrt{l\cdot\sum_{h(e_j)=h(e_i)} f_j^2}\right) \leqslant P\left(\left|\hat{f}_i - f_i\right| \geqslant \sqrt{e\sum_{e_j\neq e_i} f_j^2}\right) \leqslant \frac{1}{e}$ □

We can find that this bound is relatively loose because it also takes effect on the items in the Waving Counter. However, for items in the Heavy Part, $\sqrt{l\cdot\sum_{e_j\neq e_i}(\Delta_i f_j)^2}$ is often much smaller than $\|f\|_2$.

We can also derive an error bound of $\|f\|_1$.

Theorem D.3. *Let* $l = \frac{e}{\epsilon}$*, we have*

$$P\left(\left|\hat{f}_i - f_i\right| \geqslant \epsilon\|f\|_1\right) \leqslant \frac{1}{e}$$

Proof. For our WavingSketch, we have

$$\mathbb{E}\left[\left|\hat{f}_i - f_i\right|\right] = \mathbb{E}\left[\left|\sum_{e_j\neq e_i} f_j\cdot s(e_j)\right|\right] \leqslant \mathbb{E}\left[\left|\sum_{e_j\neq e_i} f_j\right|\right] \leqslant \frac{\epsilon\|f\|_1}{e}$$

By the Markov inequality,

$$P\left(\left|\hat{f}_i - f_i\right| \geqslant \epsilon\|f\|_1\right) \leqslant P\left(\left|\hat{f}_i - f_i\right| \geqslant e\mathbb{E}\left[\left|\hat{f}_i - f_i\right|\right]\right) \leqslant \frac{1}{e}$$

□

D.2 Parameter Analysis

We analyze the influence of parameters in WavingSketch. We use $c = dl$ to denote the number of cells in WavingSketch. Then we show that for fixed c, how d influences the performance of our WavingSketch.

Theorem D.4. *Let e_i be the i_{th} most frequent item in the data stream. The probability that its frequency f_i is among top-d largest frequencies in bucket $B[h(e_i)]$ is at least* $1 - \frac{d^d}{d!}\cdot\left(\frac{i-1}{c}\right)^d$

Proof. Let P_i be the probability that $B[h(e_i)]$ contains at least d items whose frequency is higher than e_i. When $i \leqslant d$, $P_i = 0$. So we only need to discuss the case that $i > d$. When $i > d$, we have $P_i \leqslant \binom{i-1}{d}\cdot\left(\frac{1}{l}\right)^d \leqslant \frac{d^d}{d!}\cdot\left(\frac{i-1}{c}\right)^d$. Therefore, the probability that f_i is among top-d largest frequencies in bucket $B[h(e_i)]$ is at least $1 - \frac{d^d}{d!}\cdot\left(\frac{i-1}{c}\right)^d$. □

We can find that, when i decreases, P_i decreases sharply, which indicates that the probability that e_i is top-d items in $B[h(e_i)]$ becomes much higher. According to Stirling's approximation,

$$1 - \frac{d^d}{d!}\cdot\left(\frac{i-1}{c}\right)^d \approx 1 - \frac{1}{\sqrt{2\pi d}}\cdot\left(\frac{e(i-1)}{c}\right)^d \quad (3)$$

We can also find that, when $i < \frac{c}{e} + 1$, the probability that e_i is top-d items in $B[h(e_i)]$ increases with d increasing.