Noname manuscript No.
(will be inserted by the editor)

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

Zirui Liu*, **Fenghao Dong**§, **Chengwu Liu***, **Xiangwei Deng***, **Tong Yang***✉, **Yikai Zhao***, **Jizhou Li***, **Bin Cui***, **Gong Zhang**†

Received: date / Accepted: date

Abstract Finding top-k items in data streams is a fundamental problem in data mining. Unbiased estimation is well acknowledged as an elegant and important property for top-k algorithms. In this paper, we propose a novel sketch algorithm, called WavingSketch, which is more accurate than existing unbiased algorithms. We theoretically prove that WavingSketch can provide unbiased estimation, and derive its error bound. WavingSketch is generic to measurement tasks, and we apply it to five applications: finding top-k frequent items, finding top-k heavy changes, finding top-k persistent items, finding top-k Super-Spreaders, and join-aggregate estimation. Our experimental results show that, compared with the state-of-the-art Unbiased Space-Saving, WavingSketch achieves $10\times$ faster speed and $10^3\times$ smaller error on finding frequent items. For other applications, WavingSketch also achieves higher accuracy and faster speed. All related codes are open-sourced at GitHub [4].

Keywords Data Streams; Sketches; Unbiased Estimation; Top-k Frequent Items; Heavy Changes;

* Zirui Liu, Chengwu Liu, Xiangwei Deng, Tong Yang, Yikai Zhao, Jizhou Li, and Bin Cui
Institute: School of Computer Science, Peking University
E-mail: {zirui.liu, liuchengwu, yangtong, zyk, ljzh2014, bin.cui}@pku.edu.cn, dengxiangwei@stu.pku.edu.cn

§ Fenghao Dong
Institute: Carnegie Mellon University
E-mail: fenghaod@andrew.cmu.edu

† Gong Zhang
Institute: Huawei Theory Lab, China
E-mail: nicholas.zhang@huawei.com

✉ Corresponding author: Tong Yang
E-mail: yangtong@pku.edu.cn

1 Introduction

1.1 Background and Motivation

Finding top-k items is a fundamental problem in approximate data stream mining. Nowadays, four kinds of top-k items have attracted wide attention of researchers: 1) top-k *frequent items* [37,26,40,14,57]; 2) top-k *heavy changes* [49,9,33]; 3) top-k *persistent items* [57,18]; and 4) top-k *Super-Spreaders* [54]. *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 that appear in many time windows. *Super-Spreaders* refer to the sources that connect to many distinct destinations. Although these top-k problems have different definition, we find that if an algorithm does well in finding frequent items, it can also well handle the other tasks because these tasks can be converted into the task of finding frequent items (§ 5). Recently, sketches, a kind of probabilistic data structure, have been widely used in finding top-k items, because of their memory efficiency and small error.

Unbiased estimation is well acknowledged as an elegant and important property for top-k algorithms. First, this property is important for distributed measurement, such as the tasks of global heavy hitters detection, global distribution estimation, global entropy estimation, *etc.* To measure the frequency of items in distributed data streams, we can deploy one sketch for each local data stream, and then aggregate the measurement results of all sketches. If the estimations are biased, when the measurement results are aggregated, the error of local sketches will accumulate, leading to large estimation errors. Second, the property of unbiasedness is important for the task of estimating the frequency sum. When estimating the frequency sum of all items in a given set, we accumulate the estimated frequency of each item in this set. Similarly, if the estimated frequency is biased, the errors will accumulate to the final result, thereby significantly degrading the measurement accuracy. Further, unbiased approaches can also stimulate the theoretical progress of sketches. Until now, although numerous sketches have been proposed, only a very few of them (including the biased Count-Min Sketch [17] and the unbiased Count Sketch [14]) have explicit and concise theory bounds and proofs, and most of the other sketches show rather complicated error bounds. One of the key reasons is that their estimations are biased, making the theoretical derivation very complicated.

Among a large number of algorithms for finding frequent items [17,19,14,38,37,50,53], only one recent work, Unbiased Space-Saving (USS) [53], achieves unbiased estimation. Unfortunately, its estimation variance is relatively large and its estimation for top-k items still have overestimated error. As a result, when applied to other applications (*e.g.*, finding heavy changes, or persistent items), the large estimation variance of USS will bring large error. The goal of this paper is to devise a theoretically unbiased sketch algorithm that have high accuracy and generic to different applications.

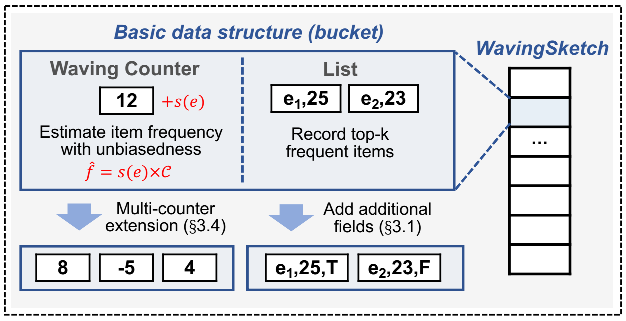

Basic data structure (bucket)
WavingSketch
Waving Counter
12 +s(e)
Estimate item frequency with unbiasedness
f̂ = s(e)×C
List
e1,25 e2,23
Record top-k frequent items
...
Multi-counter extension (§3.4)
Add additional fields (§3.1)
8 -5 4
e1,25,T e2,23,F

Fig. 1 Basic idea of WavingSketch.

1.2 Our Proposed Approach

Towards the above design goal, this paper presents WavingSketch. As shown in Figure 1, we use a simple example to explain the key idea of WavingSketch. To find the top-k frequent items, we maintain a counter $\mathcal{C}$ and a list. The counter, which is called Waving Counter, provides an unbiased estimation for each item's frequency, and the list is used to store k' ($k' > k$) items and their estimated frequencies. For each incoming item e, we first use a hash function $s(e)$ to hash e to $+1$ or -1, and then increase or decrease the Waving Counter by 1. We estimate the frequency of e using the Waving Counter: the estimated frequency is $\hat{f} = s(e) \times \mathcal{C}$. We will prove that the estimated frequency $\hat{f}$ for any item is unbiased (see § 4.1). Afterwards, if the estimated frequency of e is larger than the smallest frequency in the list, we evict the least frequent item from the list, and insert e into the list. In this way, we manage to record frequent items and evict infrequent items in the list. Based on this idea, we encapsulate the aforementioned data structure (Waving Counter and list) into a bucket, and construct our WavingSketch as a bucket array. To achieve higher accuracy, we will also add additional fields in the list (§ 3.1) and extend the Waving Counter in each bucket into multiple Waving Counters (§ 3.4).

Below we explain the rationale of WavingSketch. In practice, the value of the Waving Counter fluctuates over time, which is similar to the waves in the ocean. When the absolute value of the Waving Counter rises sharply, it is likely that a strong flow (frequent item) is driving it. Thus, we expect to catch these frequent items 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 of high probability that the incoming item is more frequent than the least frequent item in the list. Therefore, we replace the least frequent item with the incoming item. In this way, WavingSketch manages to maintain top-k frequent items and their estimated frequencies in the list.

WavingSketch has four advantages. First, WavingSketch can provide unbiased estimation, which is theoretically proved in § 4.1. Second, WavingSketch is accurate. Experimental results show that the error of WavingSketch is much smaller than state-of-the-art Space-Saving and Unbiased Space-Saving. Third, WavingSketch is generic to various applications. We apply WavingSketch to five applications: finding frequent items, finding heavy changes, finding persistent items, finding Super-Spreaders, and join-aggregate estimation. Fourth, WavingSketch is fast. WavingSketch achieves higher insertion throughput than prior art, and it can be further accelerated by SIMD instructions (see § 3.5).

1.3 Key Contributions

- We propose a sketch algorithm called WavingSketch, which can provide unbiased estimation with high accuracy, and is generic to many top-k tasks.
- We theoretically prove that WavingSketch can provide unbiased estimation, and derive its error bound.
- We apply WavingSketch to five applications: finding frequent items, finding heavy changes, finding persistent items, finding Super-Spreaders, and join-aggregate estimation.
- We conduct extensive experiments, and the results show that WavingSketch achieves $10^3\times$ smaller error and $10\times$ faster speed than state-of-the-art Unbiased Space-Saving [53] in finding frequent items.

2 Background and Related Work

2.1 Problem Statement

Data stream: A data stream σ is defined as a sequence $\{e_i\}_{i=1,2,\ldots,n}$ of n items drawn from the universe $[m] := \{1, 2, \ldots, m\}$. Each item e_i in σ is associated with a timestamp t_i indicating its arrival time.

Frequent items: Given a data stream σ of n elements, the frequency of an item $e \in [m]$ is defined as $f = |\{j \in [n] : e_j = e\}|$. Frequent items refer to a set of items $\Psi_F \subseteq [m]$, where the frequency of each item in Ψ_F is larger than a predefined threshold F. Intuitively, frequent items refer to items with large frequencies.

Heavy changes: Given a data stream σ, we divide it into equal-sized and continuous time windows. Consider an item $e \in [m]$ and two adjacent time windows w_1 and w_2. The frequency of e in w_1 (or w_2), namely f' (or f''), is defined as the number of appearances of e in time window w_1 (or w_2). The frequency change of e in w_1 and w_2, namely Δf, is defined as $\Delta f = |f''-f'|$. The heavy changes between w_1 and w_2 refer to a set of items $\Psi_C \subseteq [m]$, where the frequency change of each item in Ψ_C is larger than a predefined threshold C. Intuitively, heavy changes refer to items whose frequencies change drastically over two adjacent time windows.

Persistent items: Given a data stream σ, we divide it into equal-sized and continuous time windows again. Given an item $e \in [m]$, we define its *persistence* p as the number of time windows that e appears. Persistent items refer to a set of items $\Psi_P \subseteq [m]$, where the *persistence* of each item in Ψ_P is larger than a predefined threshold P. Intuitively, persistent items refer to items that appear in many time windows.

Super-Spreaders: We consider a particular kind of data streams: network streams. For a network stream σ_n, each incoming item in σ_n is a packet $e_i \in \Phi$ with a source address src_i and a destination address dst_i, namely $e_i = (src_i, dst_i)$. Here, Φ is the set of all distinct packets (or the set of flows). Given a source address src, we define its number of destinations as $|\{e_j \in \Phi : src_j = src\}|$. Super-Spreaders refer to a set of source addresses Ψ_S, where the number of destinations of each source address in Ψ_S is larger than a predefined threshold S. Intuitively, Super-Spreaders refer to sources that connect to a large number of distinct destinations.

2.2 Related Work

2.2.1 Finding Frequent Items

There are two types of solutions for finding frequent items. The first type, *sketch-based solutions*, records the frequencies of all items by hashing. The second type, *KV-based solutions*, records the $\langle ID, frequency\rangle$ pairs of a subset of items with large frequencies.

Table 1 Symbols frequently used in this paper.

Notation	**Meaning**
σ	A data stream $\{e_i\}_{i=1,2,\ldots,n}$ where $e_i \in [m]$
n	Number of items in σ
m	Number of distinct items in σ
e_i	The i_{th} incoming item in data stream σ
f_i	Real frequency of item e_i
$\hat{f_i}$ (or $\hat{f}$)	Estimated frequency of item e_i
$\mathcal{B}[i]$	The i_{th} bucket of WavingSketch
$\mathcal{B}[i].count$	Waving Counter of bucket $\mathcal{B}[i]$
$\mathcal{B}[i].heavy$	Heavy Part of bucket $\mathcal{B}[i]$
l	Number of buckets in WavingSketch
d	Number of cells in $\mathcal{B}[i].heavy$
$hash(\cdot)$	Raw hash function that uniformly maps items into 32-bit/64-bit integers
$h(\cdot)$	Hash function mapping items into buckets
$s(\cdot)$	Hash function mapping items to $\{+1, -1\}$
c	Number of Waving Counters in each bucket
$g(\cdot)$	Hash function mapping items into Waving Counters
$\mathcal{B}[i].count[j]$	The j_{th} Waving Counter of $\mathcal{B}[i]$
r	Compression ratio or expansion ratio.

Sketch-based solutions: A sketch is an excellent data structure that records the approximate statistics of data streams by maintaining a summary. Typical sketches include CM [17], CU [19], Count [14], ASketch [43], and more [58,11,65]. These sketches often consist of multiple arrays, each of which contains many counters. Each array is associated with a hash function that maps items to the counters. As sketches suffer the error incurred by hash collision, people propose many strategies to reduce this error. However, these strategies are usually memory inefficient for the task of finding top-k frequent items, because they record frequent items and infrequent items simultaneously, while infrequent items are useless for reporting top-k items.

KV-based solutions: Typical KV-based solutions include Space-Saving [38,16], Unbiased Space-Saving [53], Lossy Counting [50], HeavyGuardian [59], Cold filter [67], and LD-Sketch [24]. Space-Saving (SS) and Unbiased Space-Saving (USS) use a list with m buckets to record frequent items and their estimated frequencies. For an incoming item e, if it is recorded in the list, we increment its frequency by one. If e is not in the list and the list is not full, we insert $(e, 1)$ into the list. Otherwise, SS increments the frequency of the least frequent item, and replace the least frequent item with the incoming item. We can see that the estimated frequency recorded in SS are always overestimated. Based on SS, Unbiased Space-Saving (USS) makes small modification by replacing the least frequent item with a certain probability. For each item recorded in the list, USS reports an overestimated frequency, and for each item not recorded in the list, USS estimates its frequency 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. USS proves that its estimated result for any item is unbiased. This is because its overestimated error for recorded items and underestimated error for non-recorded items can just offset each other. However, although USS provides an unbiased estimation for any item, its estimated results for top-k frequent items are also biased upward. In addition, the estimation variance of USS is rather large, which leads to its unsatisfactory accuracy for finding frequent items. Notice that as probabilistic algorithms, when there are more than k buckets equipped (each bucket stores an item), SS and USS cannot guarantee to accurately report all top-k items. LD-Sketch [24] is a KV-based algorithm. The data structure of LD-Sketch is d bucket arrays, where each bucket stores a list of l KV pairs and several counters. The d bucket arrays in LD-sketch operate independently of each other. Therefore, a top-k item might be recorded in each of these d bucket arrays, which leads to a memory waste. To theoretically ensure the accuracy of finding the top-k items, the KV pair list in each bucket of LD-Sketch independently expands its size l. This leads to an uncontrollable total memory usage and unsatisfactory processing speed. In addition, LD-Sketch is an algorithm solely dedicated to the task of finding top-k items. It can only estimate the frequency upper bound and lower bound for each top-k item, and does not provide a method for estimating the exact frequency. Thus, LD-Sketch is not generic to frequency-related tasks like join-aggregate estimation.

2.2.2 **Finding Heavy Changes**

There are two kinds of solutions for finding heavy changes. The first kind is *"record all"* solutions. This kind of solutions builds a data structure to record all items in each time window, and then decodes these data structures and reports heavy changes. Typical algorithms include k-ary [9], reversible sketch [49], and FlowRadar [33]. These solutions are not memory efficient because they record all items, while recording persistently infrequent items are unnecessary for finding heavy changes. By contrast, the other kind, *"record sample"* solutions, only records frequent items. A typical *"record sample"* algorithm is Cold filter [67]. However, in practice, the data structure of Cold filter will be filled up very quickly, and thus needs to be cleaned up periodically. LD-Sketch [24] proposes to build one sketch for each time window to record only top-k items. It reports a top-k item as a heavy change if the difference of its estimated frequency in the two time windows exceeds predefined threshold. However, the accuracy of its frequency estimation has much room for improvement.

2.2.3 **Finding Persistent Items**

Again, two kinds of solutions exist. The first kind, namely *"record all"* solutions, records all items. A typical algorithm is PIE [18]. For each time window, PIE builds a hash table to record the fingerprints of the incoming items. PIE uses the key technique of Raptor codes [8] to generate different fingerprints in different time windows. For a persistent item that appears in many time windows, we can find many of its fingerprints, and these fingerprints can be used to recover the item ID. In this way, persistent items have a higher probability to be successfully recovered. Unfortunately, the accuracy of PIE is also affected by hash collisions between infrequent items and persistent items. The second kind, namely *"record samples"* solutions, records only potential persistent items. Small-Space [29] use sampling techniques to select persistent items, but its sampling error is hard to control. On-Off Sketch [63] combines CM sketch and Space-Saving to build a top-k sketch. It adds a 1-bit marker to each counter in the sketch to decide whether an item first appears in current time window. If so, it increments the frequency of this item. However, the accuracy of this top-k sketch has much room for improvement.

2.2.4 **Finding Super-Spreaders**

There are also two kinds of solutions. The first is *"record all"* solutions. A typical algorithm is OpenSketch [61], which combines CM sketch [17] and bitmap. However, OpenSketch has poor accuracy under tight memory. The second kind is *"record samples"* solutions, which records only potential super-spreaders. One-level filtering and two-level filtering [54] use sampling technique to filter infrequent items. SpreadSketch [52] combines CM sketch [17] and multi-resolution bitmap [20] to achieve theoretial guarantees on accuracy. Its data structure is d independent bucket arrays. Each super-spreader can be recorded in each of these d arrays, which leads to high memory overhead and slow processing speed.

2.3 **Importance of Unbiasedness Property**

In practice, the property of unbiased estimation is important to many applications. In this subsection, we take two typical tasks as examples to explain the importance of unbiasedness property: 1) answering subset query; 2) finding global top-k items in disjoint data streams. Besides these two tasks, unbiased algorithms are also widely applied in other tasks like computing item ranking [66], join-aggregate estimation [55,15], and network packet sampling [36].

Subset query: Given a set of items, the subset query problem estimates the aggregated results over all items in the set. Two typical subset query tasks are subset sum query and subset average query, which estimate the frequency sum and average for a given set respectively. The problem of subset query is of significant importance in data stream analysis. For example, in ad click analysis, each item represents a user's visit to an ad. Operators may wish to query the total number of views for ads from all users belonging to the same company, or from a specific country [39]. In network measurement, each item represents a packet in the network. Users may query for the total number of packets originated from a certain subnet. The above queries can be expressed as subset sums [64].

As pointed out by prior works [53,64,39], the unbiasedness property is crucial for subset query tasks, particularly for the subset sum task. Thanks to the *Law of Larger Numbers*, the unbiasedness property ensures that when aggregating the estimated frequencies of all items in a set, the overestimated errors and underestimated errors in the estimation of individual item frequencies can offset each other. By contrast, applying a biased algorithm to subset sum estimation will result in unacceptably accumulated errors, with larger sets exhibiting more substantial error accumulation. Our experimental results will show that compared to the algorithms with one-sided overestimated error, our unbiased WavingSketch achieves significantly smaller relative error ($> 10^3\times$) on estimating the subset sum/average for top-k items (§ 6.4.3).

Finding global top-k items: Given N disjoint data streams $\mathcal{S}_1, \cdots, \mathcal{S}_N$, the global top-$k$ problem finds the k items with the largest frequency among $\mathcal{S}_1, \cdots, \mathcal{S}_N$. This problem is important in many application scenarios. For example, consider an autonomous system (AS) in a wide-area network (WAN) with multiple border routers [45,51]. All external network packets sent to the AS from the same source IP address must pass through the same border router. If we regard the source IP address of network packets as the key, the network packet streams on different border routers form a group of disjoint data streams. Network operators usually want to monitor the main source of traffic entering the AS, *i.e.* the k source IP addresses that send the most packets. This objective can be cast into our problem of finding global top-k items in disjoint data streams.

To find global top-k items, each data stream $\mathcal{S}_i$ uses a top-k algorithm $\mathcal{B}_i$ to report the set $\mathcal{T}_i$ of local top-k items and their estimated frequency. The central analyzer obtains global top-k items by selecting k items with the k largest estimated frequency from $\cup_{i=1}^{N}\mathcal{T}_i$. As pointed out by prior works [66], the property of unbiasedness is important in ensuring **fairness** in the selection of global top-k items. As an example, suppose we use the well-known Space-Saving (SS) [38] or Unbiased Space-Saving (USS) [53] as the top-k algorithm. Recall that SS and USS always provide overestimated estimation for top-k items (§ 2.2) , and the overestimated error is positively correlated to the size of the data stream. If we directly sort all the selected local top-k items based on their estimated frequency, the result will be significantly related to the items' local environment (size of its data stream) rather than their real frequency. This is because the items in the heavy data streams [1] will be overestimated more and get higher chances to be selected as global top-k items, while the real top-k frequent items in the light data streams will tend to be ignored due to their small overestimated error, which is unfair. By contrast, if the algorithm can always provide unbiased estimation for top-k items, the global top-k results will no longer be influenced by the local environment, thereby achieving top-k fairness [66]. Our experimental results will show that compared to the algorithms with one-sided overestimated error, our unbiased WavingSketch achieves significantly higher F1 score (up to 60%) on finding global top-k items (§ 6.4.4).

3 The WavingSketch Algorithm

3.1 Data Structure

As shown in Figure 2, the data structure of WavingSketch is an array of l buckets. Let $\mathcal{B}[i]$ be the i^{th} bucket. Each incoming item e_i in the data stream is mapped into one bucket $\mathcal{B}[h(e_i)]$ through a hash function $h(\cdot)$. In our implementation, we get the hash function $h(\cdot)$ by modular operation: $h(\cdot) = hash(\cdot)\%l$, where $hash(\cdot)$ is a raw hash function that uniformly maps an item ID into a 32-bit/64-bit integer (*e.g.*, we use 32-bit Murmur Hash [2] in our experiments). We use another hash function $s(\cdot)$ to map each item into $\{+1, -1\}$. Similarly, $s(\cdot)$ is also obtained from modular operation. Each bucket $\mathcal{B}[i]$ consists of two parts: a Waving Counter $\mathcal{B}[i].count$, and a Heavy Part $\mathcal{B}[i].heavy$. 1) The Waving Counter $\mathcal{B}[i].count$ provides an unbiased estimation for the frequency of any item that is mapped into $\mathcal{B}[i]$. 2) The Heavy Part $\mathcal{B}[i].heavy$ consists of d cells. Each cell records a key-frequency pair and a flag $\langle ID, frequency, flag\rangle$, where *key* is the ID of the recorded item, *frequency* is its estimated frequency, and *flag* indicates whether the estimated frequency is accurate (*i.e.*, whether the estimated frequency is the real frequency). All fields in the data structure are initialized to 0 or $Null$.

3.2 Basic Operations

Insertion (Figure 2:) The pseudocode of the insertion operation is shown in Algorithm 1. Given an incoming item e_i, we first compute the hash function $h(e_i)$ to

[1] Notice that in practice the sizes of data streams are often skewed (*e.g.*, power law distribution) [41,66], where *heavy data streams* have more items and *light data streams* have less items.

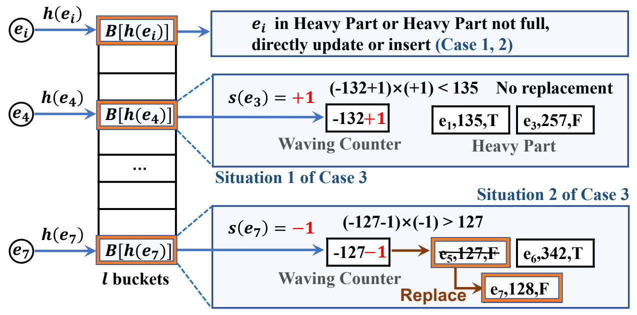


Fig. 2 Data structure and insertion examples of WavingSketch ($d = 2$).

map e_i into bucket $\mathcal{B}[h(e_i)]$ (we call $\mathcal{B}[h(e_i)]$ *the hashed bucket* of e_i in this paper). Below we show how to insert e_i into $\mathcal{B}[h(e_i)]$. There are three cases as follows:

Case 1: (see line 1-4 in Algorithm 1). If e_i is already recorded in $\mathcal{B}[h(e_i)].heavy$, there are two situations. 1) e_i is recorded with a flag of *true*: We just increment its corresponding frequency in the Heavy Part by one; 2) e_i is recorded with a flag of $false$: We not only increment its corresponding frequency by one, but also add $s(e_i)$ to $\mathcal{B}[h(e_i)].count$ ($s(e_i) \in \{-1, +1\}$).

Case 2: (see line 5-6 in Algorithm 1) If e_i is not recorded in $\mathcal{B}[h(e_i)].heavy$ and $\mathcal{B}[h(e_i)].heavy$ is not full, we just insert $\langle e_i, 1, true\rangle$ into $\mathcal{B}[h(e_i)].heavy$.

Case 3: (see line 7-15 in Algorithm 1). If $\mathcal{B}[h(e_i)].heavy$ is full and the item e_i is not recorded in $\mathcal{B}[h(e_i)].heavy$, we first add $s(e_i)$ to $\mathcal{B}[h(e_i)].count$. Next, let $\hat{f}_i = \mathcal{B}[h(e_i)].count \times s(e_i)$ be the estimated frequency of e_i. If $\hat{f}_i$ is larger than the smallest frequency in $\mathcal{B}[h(e_i)].heavy$, we replace the least frequent item in $\mathcal{B}[h(e_i)].heavy$ (denoted as e_r) with e_i as follows: 1) We set the ID field to e_i; 2) We set the frequency field to $\hat{f}_i$; and 3) We set the flag field to $false$ (indicating the frequency of e_i has error). If the flag of the replaced item e_r is *true*, we also need to insert e_r into $\mathcal{B}[h(e_i)].count$ by adding $\hat{f}_r \times s(e_r)$ to $\mathcal{B}[h(e_i)].count$, where $\hat{f}_r$ is the frequency field of e_r before replacement.

Below we use two examples to show how WavingSketch handles Case 3 (see Figure 2). In our examples, we use a WavingSketch with l buckets, and the Heavy Part of each bucket consists of $d = 2$ cells. Suppose we have inserted some items into the WavingSketch.

Example 1: When item e_4 arrives, it is mapped into bucket $\mathcal{B}[h(e_4)]$, and we have $s(e_4) = +1$. As e_4 is not in $\mathcal{B}[h(e_4)].heavy$ and $\mathcal{B}[h(e_4)].heavy$ is full, we first add $s(e_4) = +1$ to $\mathcal{B}[h(e_4)].count$. Then we have $\hat{f}_4 = \mathcal{B}[h(e_4)].count \times s(e_4) = -131$. Since $\hat{f}_4$ is smaller than the smallest frequency 135 in $\mathcal{B}[h(e_4)].heavy$, we do not insert e_4 into the Heavy Part.

Example 2: When item e_7 arrives, it is mapped into bucket $\mathcal{B}[h(e_7)]$, and we have $s(e_7) = -1$. As e_7 is not

Algorithm 1: Insertion of WavingSketch.

```
   Input: An incoming item $e_i$
1  if $e_i$ is recorded in $\mathcal{B}[h(e_i)].heavy$ then
2      increment the frequency of $e_i$ by one;
3      if $e_i.flag = false$ then
4          $\mathcal{B}[h(e_i)].count \leftarrow \mathcal{B}[h(e_i)].count + s(e_i)$;
5  else if $\mathcal{B}[h(e_i)].heavy$ is not full then
6      insert $\langle e_i,\ 1,\ true\rangle$ into $\mathcal{B}[h(e_i)].heavy$;
7  else
8      $\mathcal{B}[h(e_i)].count \leftarrow \mathcal{B}[h(e_i)].count + s(e_i)$;
9      $\hat{f}_i \leftarrow \mathcal{B}[h(e_i)].count \times s(e_i)$;
10     $e_r \leftarrow$ the least frequent item in $\mathcal{B}[h(e_i)].heavy$;
11     $\hat{f}_r \leftarrow e_r.frequency$;
12     if $\hat{f}_i > \hat{f}_r$ then
13         if $e_r.flag = true$ then
14             $\mathcal{B}[h(e_i)].count \leftarrow \mathcal{B}[h(e_i)].count + \hat{f}_r \times s(e_r)$;
15         replace $e_r$ in $\mathcal{B}[h(e_i)].heavy$ with $\langle e_i, \hat{f}_i, false\rangle$;
16 return;
```

in $\mathcal{B}[h(e_7)].heavy$ and $\mathcal{B}[h(e_7)].heavy$ is full, we first add $s(e_7) = -1$ to $\mathcal{B}[h(e_7)].count$. Then we have $\hat{f}_7 = \mathcal{B}[h(e_7)].count \times s(e_7) = 128$. Since $\hat{f}_7$ is larger than the smallest frequency 127 in $\mathcal{B}[h(e_7)].heavy$, we replace the least frequent item in $\mathcal{B}[h(e_7)].heavy$ (namely e_5) with e_7: We set the ID field of that cell to e_7; We set the frequency field to $\hat{f}_7 = 128$; We set the flag to $false$.

Unbiased estimation: Given an item e, to report the unbiased estimation of its frequency, we check the Heavy Part of $\mathcal{B}[h(e)]$. If e is in the Heavy Part with a flag of *true*, we report the frequency field as its estimated frequency. Otherwise, we report the value of $\hat{f} = \mathcal{B}[h(e)].count \times s(e)$. We will theoretically prove that the estimated frequency of e is unbiased in § 4.1.

Top-k query: To report top-k frequent items, we traverse all Heavy Parts of the WavingSketch, and report the items with top-k largest recorded frequencies.

3.3 Elastic Operations

Motivation: In practice, the density of many data streams changes dynamically over time. For example, consider a cache stream formed by many memory access requests where each request is an item. When there arrives an I/O intensive task, the cache stream will become very dense. When applying WavingSketch to measure such data streams, we cannot always set the optimal size of the sketch beforehand. When the sketch is not large enough for current high-density data stream, we should build a larger sketch to avoid poor accuracy. By contrary, when the sketch is too large for current low-density data stream, we can build a smaller sketch to save memory. However, simply building a new sketch will result in the loss of information recorded in the previous sketch. An ideal solution is to make on-the-fly reconfiguration on the sketch size. Towards this goal, we propose two elastic operations of WavingSketch, by which we can dynamically compress and expand the size of WavingSketch by any integer factor.

Compression: The compression operation compresses the size of a WavingSketch by any integer factor r. To compress a WavingSketch $\mathcal{B}$ of l buckets to a WavingSketch $\mathcal{B}'$ of $l' = l/r$ buckets (suppose $l = r \cdot l'$), we take two steps: 1) distribute the l buckets into $l' = l/r$ groups; and 2) merge the buckets in the same group into one bucket. Below we describe the two steps in detail.

To compress a WavingSketch $\mathcal{B}$, we first split the bucket array $\mathcal{B}$ into r equal-sized shards $\mathcal{B}_0, \cdots, \mathcal{B}_{r-1}$, each of which has $l' = l/r$ buckets. We have $\mathcal{B}_i[j] = \mathcal{B}[i \times r + j]$ where $i \in \{0, \cdots, r-1\}$ and $j \in \{0, \cdots, l/r-1\}$. We distribute the buckets with the same index in the r shards into the same group. For example, we distribute $\mathcal{B}_0[0], \mathcal{B}_1[0], \cdots, \mathcal{B}_{r-1}[0]$ into group 0; and we distribute $\mathcal{B}_0[1], \mathcal{B}_1[1], \cdots, \mathcal{B}_{r-1}[1]$ into group 1, *etc.*

Second, we build the compressed sketch $\mathcal{B}'$ by merging the buckets in each group. We have $\mathcal{B}'[i] = OP\{\mathcal{B}_0[i], \cdots, \mathcal{B}_{r-1}[i]\}$, where OP is the merging operator. To merge buckets $\mathcal{B}_0[i], \cdots, \mathcal{B}_{r-1}[i]$ into one bucket $\mathcal{B}'[i]$, we first sum up the Waving Counters of the r buckets to get $\mathcal{B}'[i].count = \sum_{j=0}^{r-1} \mathcal{B}_j[i].count$. Next, we retrieve all items in the Heavy Parts of the r buckets. For the $d \times r$ items, we select d items to store in $\mathcal{B}'[i].heavy$ according to the following rules: 1) First, we prefer the items with flag *true*. 2) For the items with the same flag, we prefer the items with larger frequencies.

Afterwards, for the rest $d \times (r-1)$ items that are not selected, we insert each of them with flag *true* into the Waving Counter $\mathcal{B}'[i].count$. Specifically, for an item e_r with recorded frequency of $\hat{f}_r$, namely $\langle e_r, \hat{f}_r, true\rangle$, we insert it into the Waving Counter by adding $\hat{f}_r \times s(e_r)$ to $\mathcal{B}'[i].count$. Finally, we modify the hash function of the compressed WavingSketch to $h'(\cdot) = hash(\cdot)\%l'$. Recall that $hash(\cdot)$ is the raw hash function that maps an item ID into a 32-bit/64-bit integer, and the hash function of the original WavingSketch is obtained by $h(\cdot) = hash(\cdot)\%l$. We can see that the compression operation needs to check each item stored in WavingSketch, and thus its time complexity is $O(d \cdot l)$.

Discussion: We explain why the design of our compression operation is reasonable. First, we prove that for each item e recorded in $\mathcal{B}[h(e)]$, it can still be retrieved in $\mathcal{B}'[h'(e)]$ after compression. Consider an item e. The index of its hashed bucket in $\mathcal{B}$ is $h(e) = hash(e)\%l$. Note that in our compression operation, we distribute every r buckets into the same group for merging, and each group has $l' = l/r$ buckets. Thus, after compression, e will be recorded in $\mathcal{B}'[h(e)\%l']$. On the other hand, for item e, the index of its hashed bucket in $\mathcal{B}'$ is $h'(e) = hash(e)\%l'$. We have the following lemma.

Lemma 1 *For any integer b, l, and l', if l is divisible by l', then $(b\%l)\%l' = b\%l'$*

For example, $(15\%8)\%4 = 15\%4$. Therefore, we have $(hash(e_i)\%l)\%l' = hash(e)\%l'$, namely $h(e)\%l' = h'(e)$. This property guarantees that each item in the original WavingSketch can be retrieved in the compressed WavingSketch. Note that if we modify the compression operation to simply merging every r consecutive buckets, we must change the hash function of the compressed WavingSketch to $h'(\cdot) = \lfloor h(\cdot)/r \rfloor = \lfloor (hash(\cdot)\%l)/r \rfloor$, which is more complicated.

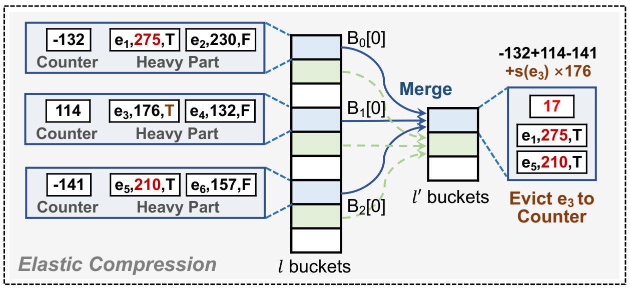


Fig. 3 Example of the elastic compression operation of WavingSketch ($d = 2$, $r = 3$, $l = 9$, and $l' = l/r = 3$).

Example (Figure 3): We use an example to illustrate the elastic compression operation of WavingSketch. To compress a WavingSketch $\mathcal{B}$ by $r = 3$ times, we first split $\mathcal{B}$ into $r = 3$ shards $\mathcal{B}_0$, $\mathcal{B}_1$, and $\mathcal{B}_2$. We distribute the buckets into $l' = l/r = 3$ groups according to their indices in the $r = 3$ shards, where the buckets distributed into the same group are marked with the same color. Next, we build the compressed sketch $\mathcal{B}'$ by merging the buckets in each group. Specifically, we merge $\mathcal{B}_0[0]$, $\mathcal{B}_1[0]$, and $\mathcal{B}_2[0]$ to get $\mathcal{B}'[0]$: First, we sum up the Waving Counters of the 3 buckets to get $\mathcal{B}'[0].count = -132 + 114 - 141 = -159$. Then we check the Heavy Parts of the 3 buckets. Since there are 3 items with the flag of *true*, we select the top-2 items with the largest frequencies (e_1 and e_5), and insert the other item e_3 into the Waving Counter by adding $s(e_3) \times 176 = 176$ to $\mathcal{B}'[0].count$. Finally, we get $\mathcal{B}'[0].count = 17$, and we have e_1 and e_5 recorded in $\mathcal{B}'[0].heavy$. Similarly, we merge the other 2 groups to get $\mathcal{B}'[1]$ and $\mathcal{B}'[2]$.

Expansion: The expansion operation enlarges the size of a WavingSketch by any integer factor r. To expand a WavingSketch $\mathcal{B}$ of l buckets to a WavingSketch $\mathcal{B}'$ of $l' = l \times r$ buckets, we also take two steps: 1) copy the l buckets of $\mathcal{B}$ r times to get $\mathcal{B}'$; 2) mark each bucket in $\mathcal{B}'$ as *redundant*, meaning that there could be redundant items in the Heavy Part of this bucket. Finally, we modify the hash function $h(\cdot)$ to $h'(\cdot) = hash(\cdot)\%l'$.

After expansion, when a *redundant* bucket $\mathcal{B}'[j]$ is accessed during insertion/query, we perform the following *redundancy-clean operation*: For each item in the Heavy Part of $\mathcal{B}'[j]$, we check whether it is a *redundancy*. Specifically, for an item e stored in $\mathcal{B}'[j].heavy$, we check whether $h'(e) = j$. If not, we regard e as a *redundancy* and delete it from $\mathcal{B}'[j]$ by clearing its cell in the Heavy Part. Note that if the redundant item e has a $false$ flag, we subtract $\hat{f} \times s(e)$ from $\mathcal{B}'[j].count$ before clearing its cell. After checking all items in $\mathcal{B}'[j].heavy$, we remove the *redundancy* mark of $\mathcal{B}'[j]$. We can see that the expansion operation copies the l buckets in WavingSketch by r times, and then cleans the redundant items in a lazy manner. Therefore, its time complexity is $O(r \cdot l)$. If we directly check all items in expanded WavingSketch and immediately clean the redundant ones during expansion, the time complexity will be $O(r \cdot l \cdot d)$.

Discussion: We explain the reason why each item e in original WavingSketch $\mathcal{B}$ can still be retrieved in the expanded WavingSketch $\mathcal{B}'$. Consider an item e recorded in $\mathcal{B}$. The index of its bucket is $h(e) = hash(e)\%l$. After expansion, e will exist in each of the following r buckets: $\mathcal{B}'[h(e)], \mathcal{B}'[h(e)+l], \cdots, \mathcal{B}'[h(e)+(r-1)\cdot l]$.

Lemma 2 *For any integer b, l, and r, we have $b\%(r \times l) \in \{b\%l, b\%l + l, \cdots, b\%l + (r-1)\cdot l\}$.*

For example, when $b = 11$, $l = 7$, and $r = 2$. We have $b\%l = 4$, and $b\%(r \times l) = 11\%(2 \times 7) = 11 = 4 + 7 = b\%l + l$. Therefore, when $b = hash(e)$, we have $h'(e) = hash(e)\%l' \in \{hash(e)\%l, hash(e_i)\%l + l, \cdots, hash(e)\%l + (r-1)\cdot l\}$. This property guarantees that e must exist in $\mathcal{B}'[h'(e)]$ after expansion.

In this way, consider a frequent item e in the Heavy Part of bucket $\mathcal{B}[h(e)]$. After expansion, it can be immediately retrieved in $\mathcal{B}'[h'(e)]$. Notice that if e does not appear after the expansion operation, it will first exist in $\mathcal{B}'[h'(e)].heavy$. As more items arrive, item e might no longer be a frequent item, and thus it can be kicked from the Heavy Part $\mathcal{B}'[h'(e)].heavy$ to the Waving Counter $\mathcal{B}'[h'(e)].count$ (*Case 3* in § 3.2).

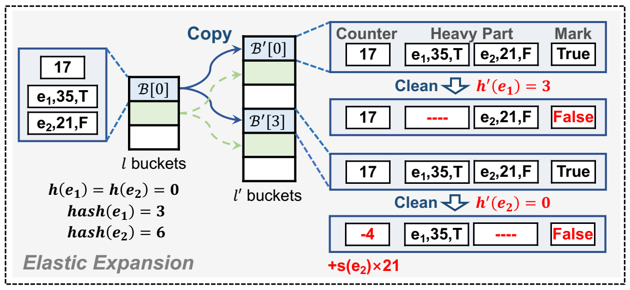


Fig. 4 Example of the elastic expansion operation of WavingSketch ($d = 2$, $r = 2$, $l = 3$, and $l' = l \times r = 6$).

Example (Figure 4): We use an example to illustrate the elastic expansion operation of WavingSketch. To expand a WavingSketch by $r = 2$ times, we copy the $l = 3$ buckets of $\mathcal{B}$ by $r = 2$ times to get $\mathcal{B}'$, where $\mathcal{B}'[0]$ is identical to $\mathcal{B}'[3]$, $\mathcal{B}'[1]$ is identical to $\mathcal{B}'[4]$, *etc.* Then we mark each bucket of $\mathcal{B}'$ as *redundant*, and modify the hash function of the expanded WavingSketch from $h(\cdot) = hash(\cdot)\%l$ to $h'(\cdot) = hash(\cdot)\%l'$.

After expansion, when a *redundant* bucket is first accessed, we perform the *redundancy−clean* operation. Below we take bucket $\mathcal{B}'[0]$ and $\mathcal{B}'[3]$ as examples to illustrate this procedure. 1) When $\mathcal{B}'[0]$ is first accessed, we check all items in its Heavy Part. For item e_1 with the *true* flag, as $h'(e_1) = 3 \neq 0$, we delete it from the Heavy Part. For item e_2, as $h'(e_1) = 0$, we keep it in $\mathcal{B}'[0]$. Finally, we remove the *redundancy* mark of $\mathcal{B}'[0]$. 2) Similarly, when $\mathcal{B}'[3]$ is first accessed, we also check all items in its Heavy Part. For e_2 with the $false$ flag, as $h'(e_2) = 0 \neq 3$, we subtract $\hat{f}_2 \times s(e_2)$ from $\mathcal{B}'[3].count$ and delete e_2 from the Heavy Part. We finally remove the *redundancy* mark of $\mathcal{B}'[3]$.

Automatic Memory Adjustment: We discuss how to leverage the elastic compression/expansion operations to enable one-the-fly memory adjustment of WavingSketch, thereby accommodating the dynamic variations in data stream density (skewness). In practice, the skewness of data streams often varies over time. Ideally, when the skewness of data stream increases, meaning that the frequencies of top-k items also increase, the problem of finding top-k items will become easier. In such case, WavingSketch can use less memory. Conversely, when the skewness of data stream decreases, the top-k problem becomes more challenging, and WavingSketch should use more memory. To achieve the above goal, we periodically calculate the hit rate θ of incoming items in the Heavy Part of WavingSketch. We use this hit rate to reflect the real-time skewness of data stream and control the memory size of WavingSketch. We maintain θ within a range $[\Theta_1, \Theta_2]$ through adjusting the memory of WavingSketch. Specifically, when θ falls below Θ_1, it indicates a decrease in data stream skewness, and in such case, we execute an expansion operation to double the memory. Conversely, when θ exceeds Θ_2, indicating an increase in data stream skewness, we execute a compression operation to halve the memory. Note that the difference $|\Theta_2 - \Theta_1|$ should be sufficiently large to prevent oscillations of repeated compression and expansion operations. We will see that by maintaining the hit rate between $[73\%, 77\%]$, our WavingSketch can automatically adjust its size to adapt to dynamically changing data stream skewness, so as to always achieve $> 97\%$ F1 score (§ 6.3.3).

3.4 Optimization using Multi-Counter Bucket

Motivation and rationale: WavingSketch uses Waving Counter to provide unbiased estimation for the items not recorded in Heavy Part (and the items recorded with flag $false$). In the basic version of WavingSketch, each bucket only has one Waving Counter. When the data stream is of high-density, multiple top-k items will collide into one bucket, and thus collide into one Waving Counter. These collisions significantly degrade the accuracy of WavingSketch. For example, consider two items e_1 and e_2 with frequencies $f_1 = 100$ and $f_2 = 100$. Suppose $s(e_1) = s(e_2) = 1$. When e_1 and e_2 collide into one Waving Counter C, we have $C = f_1 \times s(e_1) + f_2 \times s(e_2) = 200$. In this case, the estimated frequency of e_1 (or e_2) will be $\hat{f}_1 = C \times s(e_1) = 200$, which is significantly larger than its true frequency 100. To tackle this issue, we propose a multi-counter version of WavingSketch, where we extend the Waving Counter in each bucket into an array of c ($c > d$) Waving Counters. We add another hash function $g(\cdot)$ to map each item e_i into one of the c Waving Counters in its hashed bucket $\mathcal{B}[h(e_i)]$. In this way, multi-counter WavingSketch reduces the collisions of top-k items in Waving Counters by c times at the cost of more memory usage. Although simply increasing the number of buckets in basic WavingSketch can also improve the accuracy, we find that using multi-counter WavingSketch is more effective: Both our theoretical analyses (see § 4.2-4.3) and experimental results (see § 6.1) show that under the same memory usage, multi-counter WavingSketch has higher accuracy than basic WavingSketch. Next, we briefly introduce the operations of multi-counter WavingSketch.

Basic operations:

1) Insertion: For an incoming item e_i, we first compute hash function $h(e_i)$ to map it into bucket $\mathcal{B}[h(e_i)]$. If e_i is already recorded in $\mathcal{B}[h(e_i)].heavy$ (*Case 1*) or $\mathcal{B}[h(e_i)].heavy$ is not full (*Case 2*), we perform the same insertion operation as in the basic version. Otherwise (*Case 3*), we compute hash function $g(e_i)$ to map e_i into a Waving Counter $\mathcal{B}[h(e_i)].count[g(e_i)]$, and add $s(e_i)$ to this counter. Let $\hat{f}_i = \mathcal{B}[h(e_i)].count[g(e_i)] \times s(e_i)$. If $\hat{f}_i$ is larger than the smallest frequency in $\mathcal{B}[h(e_i)].heavy$, we replace the least frequent item e_r with e_i. Similar to the basic version, if the flag of the replaced item e_r is $true$, we insert e_r into its Waving Counter by adding $\hat{f}_r \times s(e_r)$ to $\mathcal{B}[h(e_r)].count[g(e_r)]$, where $\hat{f}_r$ is the frequency field of e_r before replacement.

2) Unbiased estimation: Given an item e, we also first check the Heavy Part of $\mathcal{B}[h(e)]$. If e is in the Heavy Part with flag $true$, we report its recorded frequency. Otherwise, we report the value of $\mathcal{B}[h(e)].count[g(e)] \times s(e)$ as its estimated frequency. We theoretically prove that the estimated frequency made by multi-counter WavingSketch is also unbiased in § 4.1.

Elastic operations: Multi-counter WavingSketch also supports the elastic operations in § 3.3. 1) For the compression operation, when merging $\mathcal{B}_0[i], \cdots, \mathcal{B}_{r-1}[i]$ into $\mathcal{B}'[i]$, we get the t_{th} Waving Counter by $\mathcal{B}'[i].count[t] = \sum_{j=0}^{r-1} \mathcal{B}_j[i].count[t]$ for $\forall t \in [0, c)$. Afterwards, we select the d items to be recorded in $\mathcal{B}'[i].heavy$ according to the method described in § 3.3, and evict each not-selected item e_r with flag $true$ into its Waving Counter by adding $\hat{f}_r \times s(e_r)$ to $\mathcal{B}'[i].count[g(e_r)]$. The other procedures of the compression operation are the same as in the basic version. 2) The expansion operation is exactly the same as that in the basic version. We just copy the sketch r times and mark all buckets as *redundant*.

3.5 SIMD Acceleration

Many data streams, such as network packet streams [60] and high-frequency financial transaction streams [27], are generated at extremely high speeds ($>$ 10 million items per second), necessitating that sketch algorithms be sufficiently fast to catch up with this high speed. In addition, many applications like online ML training [62] and real-time anomaly detection [24] also have strict requirements on the latency of sketch algorithms. Therefore, it is of great value to make the speed of WavingSketch as fast as possible. Single instruction, multiple data (SIMD) [21] is a widely used parallel processing technology that can perform the same operation on multiple data points simultaneously. In this subsection, we use SIMD instructions to further accelerate the insertion/query speed of WavingSketch. Nowadays, most modern processors come with built-in SIMD instructions sets (like SSE, AVX on x86 architectures). The utilization of SIMD instructions allows for harnessing the full potential of modern processors. There are also many existing sketches that use these features to accelerate their speed [34,35,60,63,68].

To accelerate WavingSketch with SIMD, we first propose the Heavy Part rearrangement technique to vectorizes the d keys and values in each bucket of WavingSketch, allowing for their parallel processing with SIMD[2]. By utilizing the parallel processing capabilities of SIMD, we further propose two techniques to accelerate the two critical procedures in the insertion/query operation of WavingSketch: 1) finding matched key (used in insertion and query operations); 2) finding the item with the smallest frequency (used in insertion operations). We discuss the implementation details of the above techniques in our supplementary materials

[2] Notice that in our implementation, we always use the Heavy Part rearrangement technique for higher processing speed, even without enabling SIMD acceleration.

[5]. Experimental results show that after using SIMD acceleration, we improve the insertion/query speed of WavingSketch by up to 45%/51% (see § 6.2.3).

4 Mathematical Analysis

4.1 Proof of Unbiasedness

We prove that for an arbitrary item, its estimated frequency made by WavingSketch is unbiased. We first consider the basic version of WavingSketch in Theorem 1. Then we extend the conclusion to multi-counter WavingSketch in Theorem 2. In this subsection, we provide a concise proof framework for Theorem 1 proving the unbiasedness nature of WavingSketch, and uses some examples to explain the different cases in our proof. We provide the detailed proof with rigorous mathematical languages in our supplementary materials [5].

Theorem 1 *Given a data stream σ and an arbitrary item $e \in [m]$ in σ, the estimated frequency of e made by basic WavingSketch, namely $\hat{f}$, is unbiased,* i.e., $\mathbb{E}(\hat{f}) = f$*, where f is the real frequency of e.*

Proof For item e, its estimated frequency $\hat{f}$ is only affected by the items mapped into the same bucket with e, namely the items mapped into $\mathcal{B}[h(e)]$. Thus, we only need to consider the items mapped into $\mathcal{B}[h(e)]$. Consider an incoming item e_k mapped into $\mathcal{B}[h(e)]$. We prove that the expected increment to the estimated frequency $\hat{f}$, namely $\mathbb{E}\left(\Delta\hat{f}\right)$, is 1 if $e_k = e$ and 0 if $e_k \neq e$.

Case 1: $e_k = e$, and e_k is error-free.

In this case, WavingSketch just increments the corresponding frequency of e in the Heavy Part by one. Afterwards, e is still in the Heavy Part and is error-free. Thus, we have $\Delta\hat{f} = \Delta f = 1$.

Example (Figure 5(a)): In Case 1 of Figure 5(a), the incoming item e_1 is in the Heavy Part with the flag *true*. Therefore, we just increment its frequency in the Heavy Part, and the increment of the estimated frequency $\Delta\hat{f}_1 = 1$.

Case 2: $e_k = e$, and e_k is not error-free.

In this case, WavingSketch estimates the frequency of e as $\hat{f} = \mathcal{B}[h(e)].count \times s(e)$. After insertion, if no error-free item is removed from the Heavy Part, we just add $s(e)$ to the Waving Counter. Therefore, we have $\Delta\hat{f} = s(e) \times s(e) = 1$. Otherwise, suppose e_r is the error-free item in Heavy Part that is replaced by e. We have $\mathcal{B}[h(e)].count' = \mathcal{B}[h(e)].count + s(e) + f_r \times s(e_r)$. Therefore, we have $\Delta\hat{f} = s(e) \times (s(e) + f_r \times s(e_r)) = s(e) \times s(e) + f_r \times s(e_r) \times s(e) = 1 + f_r \times s(e_r) \times s(e)$. Since $s(e_r)$ and $s(e)$ are independent, we can prove the expectation $\mathbb{E}\left(s(e_r) \times s(e)\right) = 0$. The detailed proof can be found in our supplementary materials [5]. Therefore, we finally have $\mathbb{E}\left(\Delta\hat{f}\right) = 1$.

Example (Figure 5(a)): In Case 2 of Figure 5(a), the incoming item e_2 is not in the Heavy Part, and after inserting e_2, an error-free item e_6 in the Heavy Part is replaced by the incoming item e_2. In this case, we add $s(e_2)$ and $s(e_6) \times f_6$ to the Waving Counter. We have $\Delta\hat{f}_2 = s(e_2) \times (s(e_2) + f_6 \times s(e_2) \times s(e_6)) = 1 + f_6 \times s(e_6) \times s(e_2)$. Note that for two distinct items e_2 and e_6, $s(e_2)$ and $s(e_6)$ are independent. Based on this property, we can prove that $\mathbb{E}\left(s(e_2) \times s(e_6)\right) = 0$, which indicates that $\mathbb{E}\left(\Delta\hat{f}_2\right) = 1$.

Case 3: $e_k \neq e$, and e_k is error-free.

In this case, WavingSketch just increments the corresponding frequency of e_k in the Heavy Part by one, which does not affect the estimated frequency of e. Thus, we have $\Delta\hat{f} = 0$.

Example (Figure 5(b)): In Case 3 of Figure 5(b), e_3 is an error-free item. The incoming item e_8 just adds $s(e_8)$ to the Waving Counter, which has nothing to do with the estimated frequency of e_3. Thus, we have $\Delta\hat{f}_3 = 0$.

Case 4: $e_k \neq e$, and e_k is not error-free.

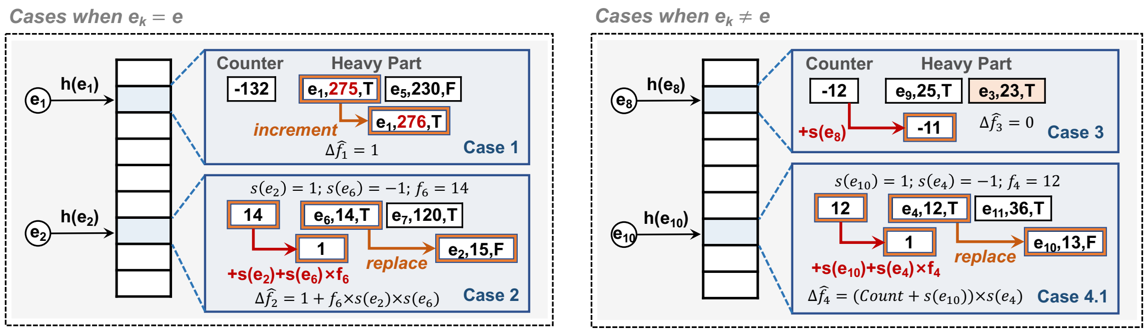
We consider two subcases of Case 4 by discussing whether e is error-free before the insertion of e_k.

Subcase 4.1: e is error-free.

In this subcase, if e is not removed from the Heavy Part by e_k, it will remain error-free after the insertion of e_k. Therefore, we naturally have $\Delta\hat{f} = 0$. Otherwise, e is replaced by e_k, and inserted into $\mathcal{B}[h(e)].count$. We have $\mathcal{B}[h(e)].count' = \mathcal{B}[h(e)].count + s(e_k) + s(e) \times f$, and $\hat{f}' = \mathcal{B}[h(e)].count' \times s(e) = f + (\mathcal{B}[h(e)].count + s(e_k)) \times s(e)$. Notice that e is error-free before the insertion of e_k, meaning that e has not been inserted into $\mathcal{B}[h(e)].count$, and thus the value of $s(e)$ does not affect $\mathcal{B}[h(e)].count$. Therefore, $\mathcal{B}[h(e)].count$ and $s(e)$ are independent. In addition, $s(e_k)$ and $s(e)$ are also independent. Similar to case 2, we can prove $\mathbb{E}\left((\mathcal{B}[h(e)].count + s(e_k)) \times s(e)\right) = 0$. In this way, we have $\mathbb{E}\left(\Delta\hat{f}\right) = 0$.

Example (Figure 5(b)): In Case 4.1 of Figure 5(b), e_4 is not error-free. The incoming item e_{10} adds $s(e_{10})$ to the Waving Counter, and replaces the error-free item e_4 in the Heavy Part. After inserting e_{10}, the estimated frequency of e_4 is $\hat{f}_4 = (\mathcal{B}[h(e_4)].count + s(e_{10}) + f_4 \times s(e_4)) \times s(e_4) = (\mathcal{B}[h(e_4)].count + s(e_{10})) \times s(e_4) + f_4$. Therefore, we have $\Delta\hat{f}_4 = (\mathcal{B}[h(e_4)].count + s(e_{10})) \times s(e_4)$. As e_4 is error-free before inserting e_{10}, it has not yet been inserting into $\mathcal{B}[h(e_4)].count$, meaning that $s(e_4)$ and $\mathcal{B}[h(e_4)].count$ are independent. On the other hand, for two distinct items e_4 and e_{10}, $s(e_4)$ and $s(e_{10})$ are also independent. Based on these properties, we can prove that $\mathbb{E}\left((\mathcal{B}[h(e_4)].count + s(e_{10})) \times s(e_4)\right) = 0$, which indicates that $\mathbb{E}\left(\Delta\hat{f}_4\right) = 0$.

Subcase 4.2: e is not error-free.



(a) Examples when $e_k = e$ (Case 1 and Case 2) (b) Examples when $e_k \neq e$ (Case 3 and Case 4)

Fig. 5 Examples illustrating the unbiasedness of WavingSketch.

In this subcase, WavingSketch estimates the frequency of e as $\hat{f} = \mathcal{B}[h(e)].count \times s(e)$. Similar to Case 2, we discuss whether an error-free item is removed from the Heavy Part after inserting e_k.

4.2.1) If no error-free item is removed from the Heavy Part, we have $\mathcal{B}[h(e)].count' = \mathcal{B}[h(e)].count + s(e_k)$, and $\Delta\hat{f} = s(e_k) \times s(e)$.

4.2.2) If an error-free item e_r is removed from the Heavy Part, we have $\mathcal{B}[h(e)].count' = \mathcal{B}[h(e)].count + s(e_k) + f_r \times s(e_r)$, and $\Delta\hat{f} = s(e_k) \times s(e) + f_r \times s(e_r) \times s(e)$.

For both situations in (4.2.1) and (4.2.2), we can prove that $\mathbb{E}\left(s(e_k) \times s(e)\right) = 0$ and $\mathbb{E}\left(f_r \times s(e_r) \times s(e)\right) = 0$, meaning that $\mathbb{E}\left(\Delta\hat{f}\right) = 0$. The detailed proof can be found in our supplementary materials [5].

It should be noted that in either situation of (4.2.1) and (4.2.2), it is incorrect to assume that $s(e)$ and $s(e_k)$ are independent of each other, as was mistakenly assumed in our conference version [32]. This is because the fact that whether an error-free item is replaced adds conditions on the value of $s(e) \times s(e_k)$. For example, if an error-free item is removed from the Heavy Part (situation 4.2.2), it essentially requires the $s(\cdot)$ values of all erroneous items in the Waving Counter tend to be more uniform. In other words, at this point, the value of $s(e) \times s(e_k)$ has a larger probability of being 1 than 0. Nevertheless, we can circumvent this issue in our proof by using some techniques in measure theory. For more details, please refer to our supplementary materials [5].

In this way, we have proved that the expected increment $\mathbb{E}\left(\Delta\hat{f}\right) = 1$ if $e_k = e$ and $\mathbb{E}\left(\Delta\hat{f}\right) = 0$ if $e_k \neq e$, which means that $\mathbb{E}(\hat{f}) = f$ always holds.

Theorem 2 *Given a data stream σ and an arbitrary item $e \in [m]$ in σ, the estimated frequency of e made by multi-counter WavingSketch is also unbiased.*

This theorem can be proved by making small modifications to the proof of Theorem 1. The detailed proof can be found in our supplementary materials [5].

We have proved that WavingSketch can provide unbiased estimation for any item (or for any top-k item). By contrast, as discussed in § 2.2.1, the estimated frequency made by Space-Saving [38] and Unbiased Space-Saving [53] for top-k items is always biased upward. Therefore, for the tasks where we want to aggregate the estimated results (*e.g.*, subset query, finding global top-k items), SS and USS will suffer significant accumulated errors (§ 2.3). We will see that our unbiased WavingSketch achieves significantly higher accuracy than biased SS and USS in these tasks in § 6.4.3 and § 6.4.4.

4.2 Variance

We derive the variance of the estimated frequency. We first consider the basic version of WavingSketch in Theorem 3. Then we extend the formula to multi-counter WavingSketch in Theorem 4.

Theorem 3 *Given a data stream σ and an arbitrary item $e \in [m]$ in σ (suppose e is not error-free, and let Ω' be the current event). Consider the basic version of WavingSketch. Let $S_1 \subseteq [m]$ be the set of all items mapped into $\mathcal{B}[h(e)]$ that are not error-free. Let $S_1' = S_1 \backslash \{e\}$. The variance of the estimated frequency of e, namely $Var(\hat{f})$, satisfies the following bound: $Var(\hat{f}) \leqslant |S_1'| \times \sum_{e_j \in S_1'} f_j^2$, where $|S_1'|$ is the cardinality of S_1'.*

Theorem 4 *Given a data stream σ and an arbitrary item $e \in [m]$ in σ (suppose e is not error-free). Consider the multi-counter version of WavingSketch. Let $S_2 \subseteq [m]$ be the set of all items mapped into $\mathcal{B}[h(e)].count[g(e)]$ that are not error-free. Let $S_2' = S_2 \backslash \{e\}$. The variance of the estimated frequency of e, namely $Var(\hat{f})$, satisfies the following bound: $Var(\hat{f}) \leqslant |S_2'| \times \sum_{e_j \in S_2'} f_j^2$, where $|S_2'|$ denotes the cardinality of set S_2'.*

The detailed proof to Theorem 3-4 can be found in our supplementary materials [5]. We can see that the variance of multi-counter WavingSketch is smaller than that of the basic WavingSketch because $S_2' \subseteq S_1'$.

4.3 Error Bound

We first derive the general error bound of WavingSketch without distribution assumption in Theorem 5-6. Then we derive the error bound of WavingSketch under Zipf distribution in Theorem 7-8. Finally, we summarize the theoretical results and analyze how the parameters of WavingSketch (l and c) affect its error. We directly consider multi-counter WavingSketch in this subsection. We present the detailed proof and the definition of probability space in our supplementary materials [5].

We first derive the error bound of WavingSketch without distribution assumption. We use L2-norm and L1-norm to derive Theorem 5 and Theorem 6.

Theorem 5 *Given a data stream σ and an arbitrary item $e \in [m]$ in σ. Let $||F_e||_2 = \sqrt{\sum_{e_j \in S_2'} f_j^2}$, where S_2' is defined in Theorem 4. The estimated frequency of item e, namely $\hat{f}$, satisfies the following error bound: $\mathbb{P}\left(\left|\hat{f} - f\right| \geqslant \epsilon\sqrt{|S_2'|} \cdot ||F_e||_2\right) \leqslant \frac{1}{\epsilon^2}$.*

Theorem 6 *Given a data stream σ and an arbitrary item $e \in [m]$ in σ. Let $||F_e||_1 = \left|\sum_{e_j \in S_2'} f_j\right|$, where S_2' is defined in Theorem 4. The estimated frequency of item e, namely $\hat{f}$, satisfies the following error bound: $\mathbb{P}\left(\left|\hat{f} - f\right| \geqslant \epsilon \cdot ||F_e||_1\right) \leqslant \frac{1}{\epsilon}$.*

We then derive the error bound of WavingSketch under Zipf distribution. In a data stream σ from a Zipf [42] distribution, the k_{th} most frequent item in $[m]$ shows up $\frac{n}{k^\alpha \zeta(\alpha)}$ times, where α [3] is the parameter of Zipf distribution and $\zeta(\alpha) = \sum_{i=1}^{m} \frac{1}{i^\alpha}$. Next, we use L2-norm and L1-norm to derive Theorem 7 and Theorem 8.

Theorem 7 *Given a data stream σ that comes from a Zipf distribution with the parameter $\alpha > 1$. Let $||F||_2 = \sqrt{\sum_{e_j \in [m]} f_j^2}$. Let $Z = \left(\frac{m}{\zeta(\alpha)}\right)^{\frac{1}{\alpha}}$ [4], meaning that the frequency of the Z_{th} most frequent items is $\frac{n}{m}$. For an arbitrary item $e \in [m]$ in σ, its estimated frequency $\hat{f}$ has the following error bound:*

$$\mathbb{P}\left(\left|\hat{f} - f\right| \geqslant \epsilon||F||_2\right) \leqslant \frac{Z}{lc} + \frac{4m}{\epsilon^2 l^2 c^2} + \frac{2lc}{m} \tag{1}$$

Theorem 8 *Given a data stream σ that comes from a Zipf distribution with the parameter $\alpha > 1$. Let $||F||_1 = \left|\sum_{e_j \in [m]} f_j\right|$. Let $Z = \left(\frac{m}{\zeta(\alpha)}\right)^{\frac{1}{\alpha}}$, meaning that the frequency of the Z_{th} most frequent items is $\frac{n}{m}$. For an arbitrary item $e \in [m]$ in σ, its estimated frequency $\hat{f}$ has the following error bound:*

$$\mathbb{P}\left(\left|\hat{f} - f\right| \geqslant \epsilon||F||_1\right) \leqslant \frac{Z}{lc} + \frac{\sqrt{2}}{\epsilon\sqrt{lc}} + \frac{2lc}{m} \tag{2}$$

Summary: From Theorem 7, we can see that when $lc \ll m$, we have that $\mathbb{P}\left(|\hat{f} - f| \geqslant \epsilon||F||_2\right) \leqslant \frac{Z}{lc} + \frac{2m}{\epsilon^2 l^2 c^2}$. In practice, we can set l and c so that l^2c^2 is significantly larger than m (*i.e.*, $\sqrt{m} \ll lc \ll m$), in which case the error of any item is theoretically guaranteed to be very small. For example, we can set lc to be $m^{0.8}$ to get a small value in the right side of the inequality in Theorem 7. In addition, we can see that when $lc \ll m$, larger value of lc goes with smaller error of WavingSketch, which proves multi-counter WavingSketch is more memory-efficient than basic WavingSketch. This is because under the same value of lc, the memory usage of multi-counter WavingSketch is smaller. For example, consider a basic WavingSketch W_1 with $2l$ buckets and a multi-counter WavingSketch W_2 with l buckets and $c = 2$ Waving Counters per bucket. We can see that the value of lc of W_1 and W_2 are the same, but the memory usage of W_1 is larger than that of W_2 because W_1 has more Heavy Parts. In other words, under the same memory usage, multi-counter WavingSketch has smaller value of lc than basic WavingSketch, and thus has smaller error.

4.4 Analysis for Elastic Operations

We first prove that the elastic expansion and compression operations do not change the unbiased property of WavingSketch in Theorem 9-10. Then we discuss how the theoretical error bounds of WavingSketch change following the elastic operations. We directly consider multi-counter WavingSketch in this subsection.

Theorem 9 *Given a data stream σ and an arbitrary item $e \in [m]$ in σ. After inserting σ into a WavingSketch $\mathcal{B}$, we perform the elastic expansion operation to expand $\mathcal{B}$ by r times and get $\mathcal{B}'$. Then the estimated frequency made by $\mathcal{B}'$ is unbiased, namely $\mathbb{E}\left(\hat{f}'\right) = f$.*

Proof In § 4.1, we have proved that the estimated frequency made by WavingSketch is unbiased, namely $\mathbb{E}\left(\hat{f}\right) = f$. After the expansion operation, the estimated frequency of e remains unchanged, namely we have $\hat{f} = \hat{f}'$. Thus, we naturally have $\mathbb{E}\left(\hat{f}'\right) = \mathbb{E}\left(\hat{f}\right) = f$.

[3] We assume $\alpha > 1$ so that the series $\sum_{i=1}^{\infty} \frac{1}{i^\alpha}$ converges.

[4] Notice that Z is a constant determined by data stream σ.

Theorem 10 *Given a data stream σ and an arbitrary item $e \in [m]$ in σ. After inserting σ into a WavingSketch $\mathcal{B}$, we perform the elastic compression operation to compress $\mathcal{B}$ by r times and get $\mathcal{B}'$. Then the estimated frequency made by $\mathcal{B}'$ is unbiased, namely $\mathbb{E}\left(\hat{f}'\right) = f$.*

Proof We discuss the following three cases.

Case 1: e is error-free in both $\mathcal{B}$ and $\mathcal{B}'$.

We have $\hat{f}' = f$, which naturally means $\mathbb{E}\left(\hat{f}'\right) = f$.

Case 2: e is error-free in $\mathcal{B}$ but not error-free in $\mathcal{B}'$.

We have $\hat{f} = f$. Let Ψ_1 be the set of all erroneous items mapped into $\mathcal{B}'[h(e)].count[g(e)]$ (except e). In other words, Ψ_1 denotes the items that conflict with item e as a result of the compression operation. We have $\hat{f}' = f + \sum_{e_j \in \Psi_2} f_j \times s(e) \times s(e_j)$. For any item $e_j \in \Psi_1$, $s(e)$ and $s(e_j)$ are independent from each other. Therefore, we have $\mathbb{E}\left(\hat{f}'\right) = f + \sum_{e_j \in \Psi_2} f_j \times \mathbb{E}\left(s(e) \times s(e_j)\right) = f + \sum_{e_j \in \Psi_2} f_j \times \mathbb{E}\left(s(e)\right) \times \mathbb{E}\left(s(e_j)\right) = f + 0 = f$.

Case 3: e is not error-free in both $\mathcal{B}$ and $\mathcal{B}'$.

According to the unbiased property of WavingSketch (§ 4.1), we have $\mathbb{E}\left(\hat{f}\right) = f$. Let Ψ_2 be the set of all erroneous items mapped into $\mathcal{B}'[h(e)].count[g(e)]$ but not mapped into $\mathcal{B}[h(e)].count[g(e)]$. In other words, Ψ_2 denotes the items that conflict with item e as a result of the compression operation. We have $\hat{f}' = \hat{f} + \sum_{e_j \in \Psi_2} f_j \times s(e) \times s(e_j)$. For any item $e_j \in \Psi_2$, $s(e)$ and $s(e_j)$ are independent from each other. Therefore, we have $\mathbb{E}\left(\hat{f}'\right) = \mathbb{E}\left(\hat{f}\right) + \sum_{e_j \in \Psi_2} f_j \times \mathbb{E}\left(s(e) \times s(e_j)\right) = \mathbb{E}\left(\hat{f}\right) + \sum_{e_j \in \Psi_2} f_j \times \mathbb{E}\left(s(e)\right) \times \mathbb{E}\left(s(e_j)\right) = \mathbb{E}\left(\hat{f}\right) + 0 = f$.

We finally discuss how the theoretical error bounds of WavingSketch in § 4.3 change following the elastic operations. For Theorem 5-6, their error bounds are based on set S_2, which is the set of all items mapped into the same Waving Counter $\mathcal{B}[h(e)].count[g(e)]$ as the target item e. According to the definition of S_2, after performing expansion/compression operations, set S_2 will contract/expand to $\overline{S}_2$. By substituting S_2 with the new $\overline{S}_2$ and plugging it into Theorem 5-6, we can get the new error bound after elastic operations. For Theorem 7-8, their error bounds are based on parameters l, c, m, and Z, where m and Z are parameters of data stream, and l and c are parameters of WavingSketch. After performing expansion/compression operations, parameter l (number of the buckets in WavingSketch) will become l', and parameter c (number of Waving Counters in each bucket) remains unchanged. By substituting l with l' and plugging it into Theorem 7-8, we can get the new error bound after elastic operations.

5 Application

5.1 Finding Top-k Frequent Items

Goal: Finding top-k frequent items refers to report the items that have top-k largest frequencies. In other word, the algorithm should report a set of items Ψ_F, where the frequency of each item in Ψ_F should be larger than a predefined threshold F, and F is the real frequency of the k_{th} most frequent item in data stream σ (see § 2.1).
Method: WavingSketch can directly find frequent items. To report the top-k frequent items, we simply traverse the bucket array of WavingSketch and return the IDs of the items that have top-k largest frequencies.

5.2 Finding Top-k Heavy Changes

Goal: Finding top-k heavy changes refers to report the items with top-k largest frequency changes over two adjacent time windows. The algorithm should report a set of items Ψ_C, where the frequency change of each item in Ψ_C should be larger than threshold C, and C is the k_{th} largest frequency change of all items (see § 2.1).
Method: Consider two adjacent time windows w_1 and w_2. We build two WavingSketches $\mathcal{B}_1$ and $\mathcal{B}_2$ for w_1 and w_2. To report the top-k heavy changes between w_1 and w_2, we traverse $\mathcal{B}_1.heavy$ and $\mathcal{B}_2.heavy$ to get the IDs of all items recorded in the Heavy Parts of $\mathcal{B}_1$ and $\mathcal{B}_2$. For each item e, we query it in $\mathcal{B}_1$ and $\mathcal{B}_2$ to get its estimated frequencies in w_1 and w_2: $\hat{f}'$ and $\hat{f}''$. We estimate the frequency change of e by $\Delta\hat{f} = |\hat{f}'' - \hat{f}'|$. Finally, we report the items with top-k largest $\Delta\hat{f}$.

5.3 Finding Top-k Persistent Items

Goal: Finding top-k persistent items refers to report the items with top-k largest *persistence* [5]. In other words, the algorithm should report a set of items Ψ_P, where the *persistence* of each item in Ψ_P should be larger than a predefined threshold P, and P is the k_{th} largest *persistence* of all items (see § 2.1).
Preliminary of Bloom filter [10]: A Bloom filter [10] is a probabilistic data structure used to judge whether an item exists in a set. A Bloom filter consists of z hash functions and one bit array, where all bits are initialized to *zero*. To insert an item, Bloom filter computes the z hash functions to pick z bits in the bit array (called the z hashed bits), and set each of the z bits to *one*. To query an item, Bloom filter checks the z hashed bits. If all the z hashed bits are *one*, Bloom filter reports *true*. Otherwise, it reports *false*. Bloom filter has false positive errors and no false negative errors.
Method: We build one WavingSketch and one Bloom filter, which is used to answer whether an item has appeared in current time window. For each incoming item

[5] We have formally defined the *persistence* of an item as the number of time windows it appears (see § 2.1).

e_i, we first query it in the Bloom filter: 1) If the Bloom filter reports *true*, meaning that e_i has already appeared in current time window, we discard e_i. 2) If the Bloom filter reports *false*, we insert e_i into the Bloom filter and WavingSketch. We periodically clean up the Bloom filter at the end of each time window. In this way, we use the WavingSketch to maintain the *persistencies* of items. To report the top-k persistent items, we traverse the bucket array of WavingSketch and return the IDs of items that have top-k largest frequencies.

5.4 **Finding Top-k Super-Spreaders**

Goal: As stated in § 2.1, we consider a particular kind of data stream, namely network stream, where each item is a source/destination address pair (src_i, dst_i). Finding top-k Super-Spreaders refers to report k source addresses that have top-k largest connections. In other words, the algorithm should report a set of source addresses Ψ_S, where the number of distinct destinations of each source address in Ψ_S should be larger than a predefined threshold S, and S is the k_{th} largest number of destinations of all source addresses (see § 2.1).

Method: We build one WavingSketch and one Bloom filter. The WavingSketch is used to record the number of destinations for source addresses, and the Bloom filter is used to remove duplicated items. For each incoming item $e_i = (src_i, dst_i)$, we first query it in the Bloom filter: 1) If the Bloom filter reports *true*, meaning that e_i has appeared before, we just discard e_i. 2) If the Bloom filter reports $false$, we insert $e_i = (src_i, dst_i)$ into the Bloom filter, and then insert src_i into the WavingSketch. To report the top-k Super-Spreaders, we traverse the bucket array of WavingSketch and return the source addresses with top-k largest frequencies.

5.5 **Performing Join-aggregate Estimation**

Goal: Join-aggregate estimation is an important task in data management society. Given two data streams $\sigma_1 = \{e_i\}_{i=1,2,\dots,n_1}$ and $\sigma_2 = \{e_j\}_{j=1,2,\dots,n_2}$ drawn from the universe $[m] := \{1, 2, \dots, m\}$. For an arbitrary item e_i. Let f_i and g_i denote the frequency of e_i in σ_1 and σ_2, respectively. The result of the join-aggregate query on σ_1 and σ_2 is defined as $J(\sigma_1, \sigma_2) = \sum_{i=1}^{m} f_i \cdot g_i$, where m is the number of distinct items in σ_1 and σ_2.

Background and prior art: Join-aggregate estimation is the base of many data management applications [30,25,31,56]. For example, in many data mining applications [28,44], join-aggregate results are used to measure the cosine similarity of two data streams. For another example, consider the case of distributed multi-way join in DBMS, a good join-aggregate estimation algorithm can guide us to devise an optimal join plan, which minimizes the volume of intermediate relations and the communication time. In some cases, we must treat the attribute values from a large table as a data stream [25], because these tables are so large that we can only process their values in a one-pass manner. As it is impractical and unnecessary to compute the exact join-aggregate results, researchers turn to probabilistic data structures, namely sketches, for fast approximate join-aggregate computation. Typical sketches include AGMS [7], Fast-AGMS (FAGMS) [15], Skimmed sketch [22], JoinSketch [55], and more [23,48,47,12]. Skimmed sketch [22] and JoinSketch [55] propose to separate items into multiple parts according to their frequencies, and record items in different parts with different components. Specifically, Skimmed sketch separates items into two parts: hot items and cold items. JoinSketch separates items into three parts: hot items, medium items, and cold items. In this way, they achieve higher accuracy than traditional FAGMS.

Method: Similar as the separation idea above, we apply WavingSketch to perform join-aggregate estimation by separately considering frequent items and infrequent items. We use basic WavingSketch to explain our join-aggregate procedure. It is straightforward to extend our method to multi-counter WavingSketch.

First, we build two equal-sized WavingSketches $\mathcal{B}_1$ and $\mathcal{B}_2$ for the two data streams σ_1 and σ_2. We calculate $\hat{J}(\sigma_1, \sigma_2)$ by checking every two buckets in the same position of $\mathcal{B}_1$ and $\mathcal{B}_2$. Specifically, for $\mathcal{B}_1[k]$ and $\mathcal{B}_2[k]$, we define Ψ_k as the set of all items mapped into $\mathcal{B}_1[k]$ or $\mathcal{B}_2[k]$, *i.e.*, $\Psi_k := \{e_i : e_i \in [m] \wedge h(e_i) = k\}$. We calculate the join-aggregate value $\hat{J}_k(\sigma_1, \sigma_2)$ of the items in Ψ_k by dividing Ψ_k into the following three parts: 1) Let $\Psi_{k,1} \subseteq \Psi_k$ be the set of the items recorded in both $\mathcal{B}_1[k].heavy$ and $\mathcal{B}_2[k].heavy$ with flag *true*. We calculate $\hat{J}_{k,1}(\sigma_1, \sigma_2) = \sum_{e_i \in \Psi_{k,1}} \hat{f}_i \cdot \hat{g}_i$, where $\hat{f}_i$ and $\hat{g}_i$ are the recorded frequencies in the Heavy Parts. 2) Let $\Psi_{k,2}^1 \subseteq \Psi_k$ be the set of the items that are recorded in $\mathcal{B}_1[k].heavy$ with flag *true*, but are not recorded in $\mathcal{B}_2[k].heavy$ or recorded in $\mathcal{B}_2[k].heavy$ with flag $false$. We calculate the first part of $\hat{J}_{k,2}(\sigma_1, \sigma_2)$ as $\hat{J}_{k,2}^1(\sigma_1, \sigma_2) = \sum_{e_i \in \Psi_{k,2}^1} \hat{f}_i \cdot \hat{g}_i$, where $\hat{f}_i$ is recorded in the Heavy Part of $\mathcal{B}_1[k]$ and $\hat{g}_i = \mathcal{B}_2[k].count \times s(e_i)$. Similarly, let $\Psi_{k,2}^2 \subseteq \Psi_k$ be the set of the items that are recorded in $\mathcal{B}_2[k].heavy$ with flag *true*, but are not recorded in $\mathcal{B}_1[k].heavy$ or recorded in $\mathcal{B}_1[k].heavy$ with flag $false$. We calculate the second part of $\hat{J}_{k,2}(\sigma_1, \sigma_2)$ as $\hat{J}_{k,2}^2(\sigma_1, \sigma_2) = \sum_{e_i \in \Psi_{k,2}^2} \hat{f}_i \cdot \hat{g}_i$, where $\hat{f}_i = \mathcal{B}_1[k].count \times s(e_i)$ and $\hat{g}_i$ is recorded in the Heavy Part of $\mathcal{B}_2[k]$. 3) Let $\Psi_{k,3} = \Psi_k \backslash (\Psi_{k,1} \cup \Psi_{k,2})$ be the set of the other items in Ψ_k. We calculate their join-aggregate value as $\hat{J}_{k,3}(\sigma_1, \sigma_2) = \mathcal{B}_1[k].count \times \mathcal{B}_2[k].count$. Finally, we get $\hat{J}(\sigma_1, \sigma_2) = \sum_{k=0}^{l-1} \hat{J}_k(\sigma_1, \sigma_2) = \sum_{k=0}^{l-1} \sum_{i=1}^{3} \hat{J}_{k,i}(\sigma_1, \sigma_2)$. Next, we theoretically prove that this estimated join-aggregate result made by WavingSketch is unbiased, and uses an example to explain the estimation process and the mathematical proof.

Theorem 11 *Given two data streams σ_1 and σ_2, the estimated result of their join-aggregate query made by WavingSketch is unbiased, namely we have $\mathbb{E}(\hat{J}(\sigma_1,\sigma_2)) = J(\sigma_1,\sigma_2)$.*

Proof This theorem can be proved by separately proving all of the three parts of $\hat{J}_k(\sigma_1,\sigma_2)$ are unbiased. Actually, it is quite straightforward to see that the first two parts of $\hat{J}_k(\sigma_1,\sigma_2)$ are unbiased. We will illustrate this fact in our following example. For the detailed proof, please refer to our supplementary materials [5].

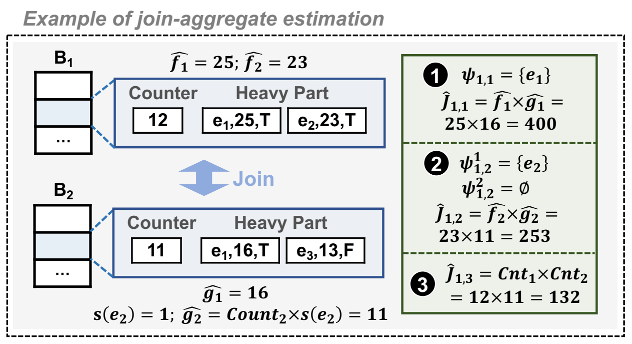

Example of join-aggregate estimation
B_1
$\hat{f_1} = 25$; $\hat{f_2} = 23$
Counter
Heavy Part
12
e_1,25,T
e_2,23,T
Join
B_2
Counter
Heavy Part
11
e_1,16,T
e_3,13,F
$\hat{g_1} = 16$
$s(e_2) = 1$; $\hat{g_2} = Count_2 \times s(e_2) = 11$
1 $\Psi_{1,1} = \{e_1\}$
$\hat{J}_{1,1} = \hat{f_1} \times \hat{g_1} = 25 \times 16 = 400$
2 $\Psi^1_{1,2} = \{e_2\}$
$\Psi^2_{1,2} = \emptyset$
$\hat{J}_{1,2} = \hat{f_2} \times \hat{g_2} = 23 \times 11 = 253$
3 $\hat{J}_{1,3} = Cnt_1 \times Cnt_2 = 12 \times 11 = 132$

Fig. 6 Example of the unbiased join-aggregate estimation.

Example (Figure 6): To calculate the join-aggregate result $\hat{J}(\sigma_1,\sigma_2)$, we check every pair of buckets in the same position of $\mathcal{B}_1$ and $\mathcal{B}_2$. We take the second bucket pair $\mathcal{B}_1[1]$ and $\mathcal{B}_2[1]$ as an example to illustrate the estimation process. We calculate the three parts of $\hat{J}_1(\sigma_1,\sigma_2)$ as follows. 1) We can see that e_1 is the only item that is error-free in both $\mathcal{B}_1[1].heavy$ and $\mathcal{B}_2[1].heavy$, meaning that $\Psi_{1,1} = \{e_1\}$. Therefore, we have $\hat{J}_{1,1}(\sigma_1,\sigma_2) = \hat{f}_1 \cdot \hat{g}_1 = 25 \cdot 16 = 400$. It is straightforward to see that $\hat{J}_{1,1}(\sigma_1,\sigma_2)$ is error-free because both $\hat{f}_1$ and $\hat{g}_1$ are error-free. Therefore, $\hat{J}_{1,1}(\sigma_1,\sigma_2)$ is naturally unbiased. 2) We can see that e_2 is the item that is error-free in $\mathcal{B}_1[1].heavy$ and not error-free in $\mathcal{B}_2$, meaning that $\Psi^1_{1,2} = \{e_2\}$. Similarly, we can see that $\Psi^2_{1,2} = \emptyset$. We calculate the second part as $\hat{J}_{1,2}(\sigma_1,\sigma_2) = \hat{f}_2 \cdot \hat{g}_2 = 23 \cdot 11 = 253$, where $\hat{f}_2$ is recorded in $\mathcal{B}_1[1].heavy$ and $\hat{g}_2 = \mathcal{B}_1[1].count \times s(e_2) = 11 \times 1 = 11$. In § 4.1, we have proved that the estimated frequency made by WavingSketch is unbiased (Theorem 1-2), meaning that $\hat{g}_2$ is unbiased. As $\hat{f}_2$ is error-free and $\hat{g}_2$ is unbiased, we have $\hat{J}_{1,2}(\sigma_1,\sigma_2) = \hat{f}_2 \cdot \hat{g}_2$ is also unbiased. 3) We directly calculate the third part as $\hat{J}_{1,3}(\sigma_1,\sigma_2) = \mathcal{B}_1[1].count \times \mathcal{B}_2[1].count = 12 \times 11 = 132$. We theoretically prove $\hat{J}_{1,3}(\sigma_1,\sigma_2)$ is also an unbiased estimation for the join-aggregate value of the items in $\Psi_{1,3}$ in our supplementary materials [5].

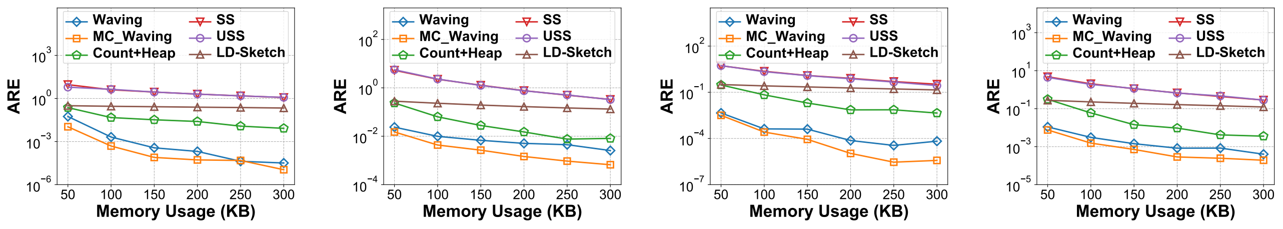
6 Experimental Results

We conduct experiments on one synthetic dataset (Zipf [42]) and three real-world datasets (IP Trace [1], WebPage [3], and Network [6]). For the synthetic dataset, we use Web Polygraph [46], an open-source performance testing tool, to generate 10 synthetic datasets following Zipf [42] distribution with different skewness ($\alpha \in [0, 3]$). By default, we use the dataset with $\alpha = 1.0$. We use the following metrics: Average Relative Error (ARE), Recall Rate (RR), Precision Rate (PR), F1 Score, and Throughput. For details about the platform, implementation, datasets, and metrics, please refer to our supplementary materials [5].

6.1 Experiments on Finding Frequent Items

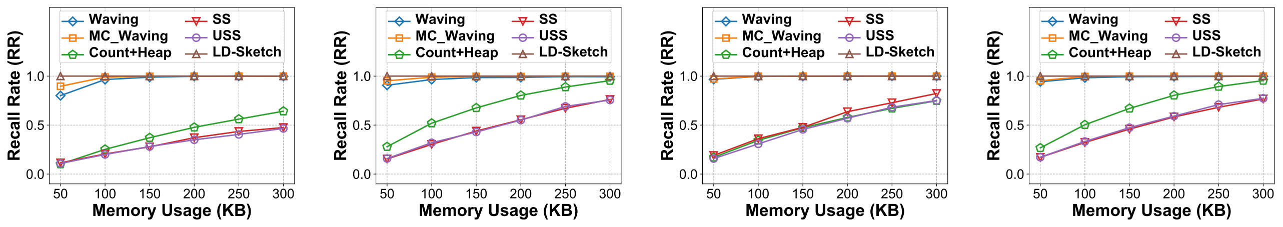
We compare WavingSketch with four algorithms: Count sketch + Max-Heap (Count+Heap) [14], Unbiased Space-Saving (USS) [53], Space-Saving (SS) [38], and LD-Sketch (LD) [24]. We set $k = 2000$ and conduct experiments using both basic WavingSketch ($d = 8$, $c = 1$) and multi-counter WavingSketch ($d = 8$, $c = 16$). For WavingSketch, SS, and USS, we vary their memory by changing bucket number l. For Count+Heap, we vary its memory by changing the number of counters in Count sketch and the Heap size. Recall that the KV-pair list in each bucket of LD-Sketch independently expands its size, and thus the total memory usage of LD-Sketch is uncontrollable (§ 2.2.1). In our experiments, we control the initial memory of LD-Sketch to be the same as the other algorithms, meaning that the actual memory usage of LD-Sketch is significantly larger than that indicated in Figure 7-11 (at least 10× larger).

ARE (Figure 7(a)-7(d)): We find that the ARE of WavingSketch is significantly smaller than that of SS, USS, Count+Heap, and LD-Sketch, and the ARE of multi-counter WavingSketch is smaller than that of basic WavingSketch. We conduct experiments to estimate the frequency of top-k items and report the ARE. Recall that LD-Sketch is an algorithm solely designed for finding top-k items (§ 2.2.1). However, the algorithm of LD-Sketch can provide an upper estimation and a lower estimation for the frequency of each recorded top-k item. In this experiment, we use the upper estimation of LD-Sketch as the estimated frequency because we find that this method yields lower error. On synthetic dataset, when using 200KB memory, the ARE of basic WavingSketch and multi-counter WavingSketch are 2.06×10^{-4} and 5.28×10^{-5}, while that of SS, USS, Count+Heap, and LD-Sketch are 2.02, 2.00, 0.025, and 0.25, respectively. On the other three real-world datasets, the ARE of WavingSketch and multi-counter WavingSketch are also significantly smaller than other algorithms. The ARE of SS and USS is high because of their overes-



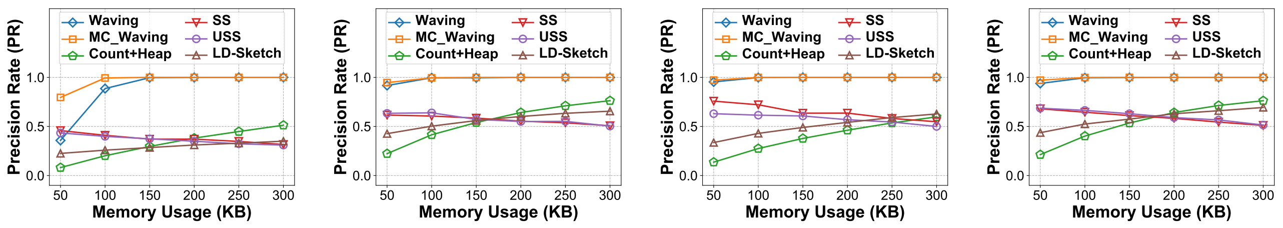
(a) Synthetic (b) IP trace (c) Webpage (d) Network

Fig. 7 Average Relative Error (ARE) on finding frequent items ("MC_Waving" refers to Multi-counter WavingSketch).



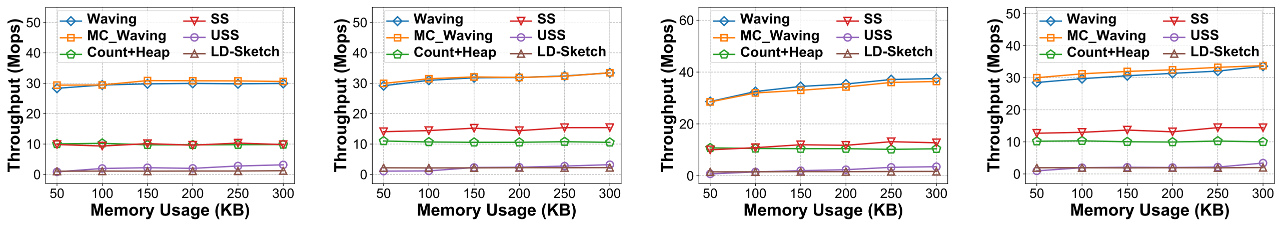
(a) Synthetic (b) IP trace (c) Webpage (d) Network

Fig. 8 Recall Rate (RR) on finding frequent items ("MC_Waving" refers to Multi-counter WavingSketch).



(a) Synthetic (b) IP trace (c) Webpage (d) Network

Fig. 9 Precision Rate (PR) on finding frequent items ("MC_Waving" refers to Multi-counter WavingSketch).



(a) Synthetic (b) IP trace (c) Webpage (d) Network

Fig. 10 Insertion throughput on finding frequent items ("MC_Waving" refers to Multi-counter WavingSketch).

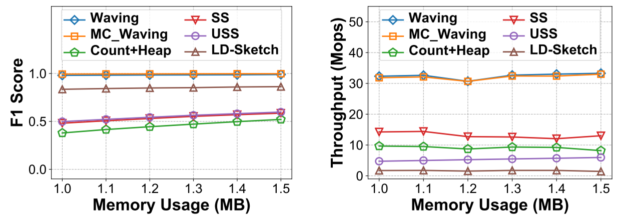
timated errors. The ARE of Count+Heap is high, because the Count sketch has small error only when there is sufficient memory. LD-Sketch has large ARE because its estimation has large upward bias. Actually, its original paper did not conduct experiments on frequency estimation, nor did it specify a method for estimating frequency. Besides, as discussed in § 2.2.1, LD-Sketch is memory inefficient due to the redundancy of the top-k item in its data structure.

RR (Figure 8(a)-8(d)): We find that the RR of WavingSketch is significantly higher than that of SS, USS, and Count+Heap. The RR of multi-counter WavingSketch, basic WavingSketch, and LD-Sketch are almost identical, which are nearly 100%. On synthetic dataset, under the memory usage of 100KB, the RR of basic WavingSketch, multi-counter WavingSketch, Count+Heap, SS, USS, and LD-Sketch are 96.50%, 99.05%, 25.23%, 20.52%, 19.97%, and 100%, respectively. On the other datasets, the RR of WavingSketch and LD-Sketch are also almost always 100%, which are significantly better than the other algorithms. Notice that although LD-Sketch achieves almost 100% RR, its actual memory usage is significantly larger (at least 10× larger) than that indicated in the figures. In addition, we will see that it has poor precision rate.

PR (Figure 9(a)-9(d)): We find that the PR of WavingSketch is significantly higher than that of SS, USS, Count+Heap, and LD-Sketch, and the PR of multi-counter WavingSketch is higher than that of basic WavingSketch. On the synthetic dataset, under the memory usage of 100KB, the PR of basic WavingSketch, multi-counter WavingSketch, Count+Heap, SS, USS, and LD-Sketch are 88.60%, 99.40%, 20.16%, 41.01%, 39.20%, and 25.86%, respectively. On the other datasets, the PR of WavingSketch is also almost 100%, which is significantly better than the other algorithms. LD-Sketch demonstrates a low PR because it reports top-k items based on the upper estimation of item frequency. The overestimated error of upper estimation leads to false positives while minimizes false negatives. In other words, LD-Sketch sacrifices precision rate in favor of high recall rate. We can see that the PR of SS and USS sometimes decreases as memory grows. This is common for algorithms that have overestimated errors. For example, under small memory usage, if the algorithm can only record 200 items, then the frequencies of all the 200 items might exceed the predefined frequency threshold, resulting in a 100% PR. However, under large memory, if 2000 items are recorded, then there might only be 1800 items whose estimated frequency exceeds the predefined threshold, which leads to a 90% PR.

Insertion throughput (Figure 10(a)-10(d)): We find that the insertion throughput of WavingSketch is significantly higher than that of SS, USS, Count+Heap, and LD-Sketch, and the throughput of multi-counter WavingSketch is lower than that of basic WavingSketch. On the synthetic dataset, under the memory usage of 100KB, the throughput of basic WavingSketch, multi-counter WavingSketch, Count+Heap, SS, USS, and LD-Sketch are 29.42 Mops, 29.43 Mops, 9.34 Mops, 1.96 Mops, 10.24 Mops, and 1.01 Mops respectively. On the other datasets, the throughput of WavingSketch is also significantly higher than the other algorithms. SS and USS have slow throughput because of the frequent cache misses. Count+Heap has slow throughput because it needs multiple memory accesses per insertion, and the time complexity of its heap operations is $O(\log k')$ where k' is its heap size. LD-Sketch has slow throughput because of the following reasons. 1) As each item accesses d buckets during insertion, LD-Sketch needs at least d memory accesses per insertion. 2) The KV-pair list in each bucket of LD-Sketch independently expands its size. These frequent dynamic expansion operations are time-consuming. 3) Each bucket in LD-Sketch varies in size, and thus the bucket array of LD-Sketch cannot be implemented in a hardware-friendly manner. It cannot leverage hardware techniques (*e.g.*, SIMD) to accelerate its speed either. By contrast, WavingSketch has always $O(1)$ time complexity, and it only needs one memory access per insertion, which is very fast.

Performance on large-scale dataset (Figure 11): We find that on large-scale dataset with billions of items, WavingSketch also achieves high accuracy and fast speed. As shown in Figure 11(a), when using 1MB memory, the F1 score of basic WavingSketch, multi-counter WavingSketch, Count+Heap, SS, USS, and LD-Sketch are 98.03%, 99.37%, 37.94%, 48.12%, 49.79%, and 83.51%, respectively. The throughput of basic WavingSketch, multi-counter WavingSketch, Count+Heap, SS, USS, and LD-Sketch are 32.32 Mops, 31.84 Mops, 9.67 Mops, 14.23 Mops, 4.73 Mops, and 1.72 Mops, respectively.



(a) F1 Score (b) Throughput

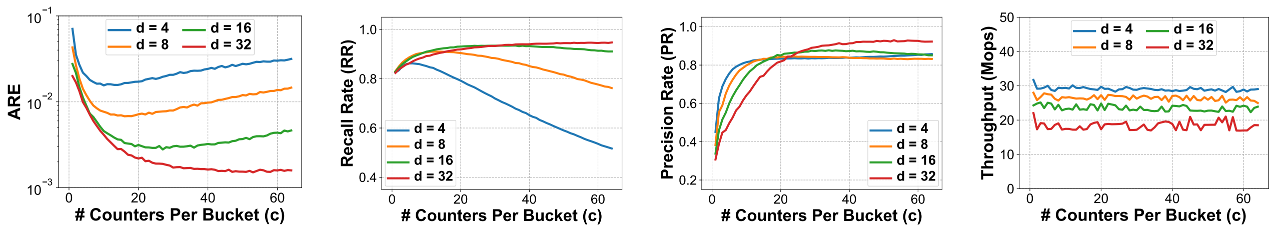
Fig. 11 WavingSketch on large-scale IP trace dataset.

Summary: 1) Under limited memory usage, WavingSketch has higher accuracy than SS, USS, Count+Heap, and LD-Sketch. This is because when using small memory, SS, USS, and LD-Sketch have large overestimated errors, and Count sketch has large variance. 2) WavingSketch is faster than SS, USS, Count+Heap, and LD-Sketch. This is because the time complexity of WavingSketch is always $O(1)$, while that of the other three algorithm is related to their sizes. In addition, when using small d, WavingSketch only needs one memory access per insertion. By contrast, SS, USS, Count+Heap, and LD-Sketch need multiple memory accesses and complex operations. 3) Compared to basic WavingSketch, multi-counter WavingSketch has higher accuracy but slightly slower throughput. As explained in § 3.4, multi-counter WavingSketch has higher accuracy because it reduces the collisions of frequent items in Waving Counters, and as this operation needs extra computation, its insertion speed is slower. 4) WavingSketch scales well to large-scale datasets, where it also achieves higher accuracy and faster speed than prior art.

6.2 Experiments on Different Settings

6.2.1 Impact of WavingSketch Parameters

We evaluate how the two parameters (d and c) of WavingSketch affect its performance in Figure 12. We fix the memory of WavingSketch to 50KB, because such small memory better exposes the impact of different d and c. We vary d from 4 to 32, and vary c from 1 to 64. We further show that multi-counter WavingSketch



(a) ARE (b) RR (c) PR (d) Throughput

Fig. 12 Impact of WavingSketch parameters.

is more memory efficient in Figure 13, where we report the minimal memory of WavingSketch under different ARE/RR requirement. We use the synthetic dataset and set $k = 2000$ by default.

ARE (Figure 12(a)): We find that larger d always goes with smaller ARE, and when d is fixed, the ARE first decreases and then increases as c grows from 1 to 64. For basic WavingSketch ($c = 1$), the ARE of $d = 32$ is about 3.6× smaller than that of $d = 4$. For multi-counter WavingSketch, when $c = 16$, the ARE of $d = 32$ is about 6.2× smaller than that of $d = 4$. When $d = 8$, the optimal value of c is from 14 to 18.

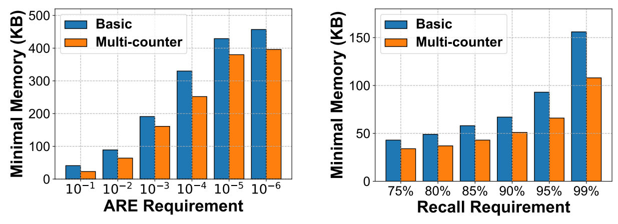
RR (Figure 12(b)): We find larger d goes with higher RR, and when d is fixed, the RR first increases and then decreases as c grows from 1 to 64. For basic WavingSketch, the RR under different d are almost the same. For multi-counter WavingSketch, when $c = 16$, the RR of $d = 16$ is 92.5%, and the RR of $d = 4$ is 81.8%. When $d = 8$, the optimal value of c is from 12 to 16.

PR (Figure 12(c)): We find that under large value of c, larger d always goes with higher PR, and when d is fixed, the PR always increases as c grows from 1 to 64. When $c = 32$, the PR of $d = 32$ is 90.1%, and the PR of $d = 4$ is 83.5%. When $d = 8$, the PR reaches its optimal value when c is larger than 18.

Insertion throughput (Figure 12(d)): We find that smaller d goes with higher insertion throughput. We also find that when d is fixed, the throughput is insensitive to c. Our results show that the throughput of $d = 4$ is around 1.11×, 1.29×, and 1.41× higher than that of $d = 8$, $d = 16$, and $d = 32$. When $d = 8$, the throughput of WavingSketch is about 29.0 Mops.

Minimal memory (Figure 13): We find that multi-counter WavingSketch is 10% $\sim$ 35% more memory efficient than basic WavingSketch. Under the ARE requirement of 10^{-4}, the minimal memory of basic WavingSketch and multi-counter WavingSketch are 330KB and 252KB. Under the RR requirement of 99%, the minimal memory of basic WavingSketch and multi-counter WavingSketch are 158KB and 104KB.

Summary: 1) Multi-counter WavingSketch is more memory efficient than basic WavingSketch. 2) Under fixed c, larger d goes with higher accuracy and lower throughput. The reason is that as d grows, there are more cells in the Heavy Part, and thus WavingSketch has more chance to record the ground-truth top-k items. At the same time, we also need to check more cells at each insertion, which slows down its throughput. 3) Under fixed d, as c grows from 1, the accuracy first increases and then decreases. When c is relatively small (< 100), the throughput is nearly not affected by c. As explained in § 3.4, using multiple Waving Counters reduces the probability of frequent items colliding into the same counter, and thus improves the accuracy. However, under fixed memory usage, if we use too much Waving Counters, the number of cells in the Heavy Part will decreases, which will degrade the accuracy. Our results show that the optimal value of c is from $1.5d$ to $2.5d$.



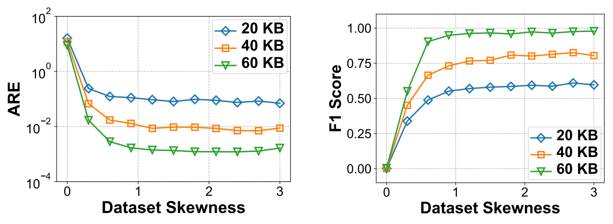
(a) ARE requirement (b) Recall requirement

Fig. 13 Minimal size under different ARE/RR requirement.

Parameter setup methods: We briefly explain how to set the three parameters of WavingSketch (l, d, and c) in practice. 1) First, we set d according to the speed requirement. For applications that prefer higher speed, we can choose a small d, and for applications that prefer higher accuracy, we can choose a large d. In practice, we recommend to use $d = 8$ or $d = 16$. 2) For multi-counter WavingSketch, we recommend to set c to $1.5d \sim 2.5d$. For example, when $d = 8$, we can set $c = 16$. 3) When d and c are fixed, meaning that the size of each bucket is fixed, we set l according to the size of available memory.

6.2.2 Impact of Data Distribution

We evaluate how the data distribution affect the performance of WavingSketch. We use the 10 synthetic datasets that follow the Zipf [42] distribution with the skewness varies from 0.0 to 3.0. We set $k = 2000$, $d = 8$, and $c = 16$, and we vary the memory from 20KB to



(a) ARE (b) F1 score

Fig. 14 Impact of data distribution.

60KB, because such small memory better exposes the difference between different distributions.

ARE (Figure 14(a)): We find that the ARE of WavingSketch decreases as the data skewness grows. For example, when using 60KB memory, the ARE under the skewness of 3.0 is about 10.3× and 1.7× smaller than that under the skewness of 0.3 and 0.6, respectively.

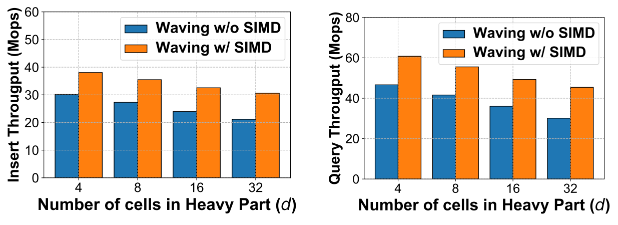
F1 score (Figure 14(b)): We find that the F1 score of WavingSketch increases as the data skewness grows. For example, when using 60KB memory, the F1 score under the skewness of 3.0 is about 1.78× and 1.08× higher than that under the skewness of 0.3 and 0.6 respectively.

Summary and analysis: We find that higher data skewness goes with higher accuracy of WavingSketch, meaning that WavingSketch is good at processing skewed data. These results validate that WavingSketch is most suitable for the application scenarios where the data stream follows a highly skewed distribution.

6.2.3 Impact of SIMD Acceleration

We evaluate the speed improvement of WavingSketch under SIMD acceleration. We use SIMD to accelerate a multi-counter WavingSketch with $c = 16$. We fix the memory usage of WavingSketch to 200KB, and conduct the experiments using the synthetic dataset.

Experimental results (Figure 15): We find SIMD acceleration improves the insertion and query speed of WavingSketch by 27% $\sim$ 45% and 30% $\sim$ 51%. As shown in Figure 15(a), when $d = 32$, the insertion throughput of WavingSketch without and with SIMD are 21.18 Mops and 30.60 Mops. As shown in Figure 15(b), when $d = 32$, the query throughput of WavingSketch without and with SIMD are 30.13 Mops and 45.41 Mops.



(a) Insertion throughput (b) Query throughput

Fig. 15 Impact of SIMD acceleration.

6.3 Experiments on Elastic Operations

6.3.1 Performance of Elastic Compression

We evaluate the accuracy of the compressed WavingSketch and the compression speed under the synthetic dataset. We set $d = 8$ and $c = 16$. In Figure 16(a)-16(c)), we build a 320KB WavingSketch, and compress it by different ratio. For each compression ratio r, we compare the accuracy of the compressed WavingSketch W_1 with the WavingSketch of the same memory as W_1. In Figure 16(d), we evaluate the compression time at different initial memory and compression ratio.

Accuracy (Figure 16(a)-16(c)): We find that under the same memory, the accuracy of WavingSketch compressed from large memory is significantly better than that of the WavingSketch initially using small memory. When $r = 16$, the ARE of the WavingSketch compressed from 320KB to 20KB is 1.0×10^{-6}, which is four orders of magnitude smaller than that of the WavingSketch initially using 20KB memory. When $r = 16$, the RR/PR of the WavingSketch compressed from 320KB to 20KB is 76.9%/100%, while that of the WavingSketch initially using 20KB memory is 52.1%/68.2%.

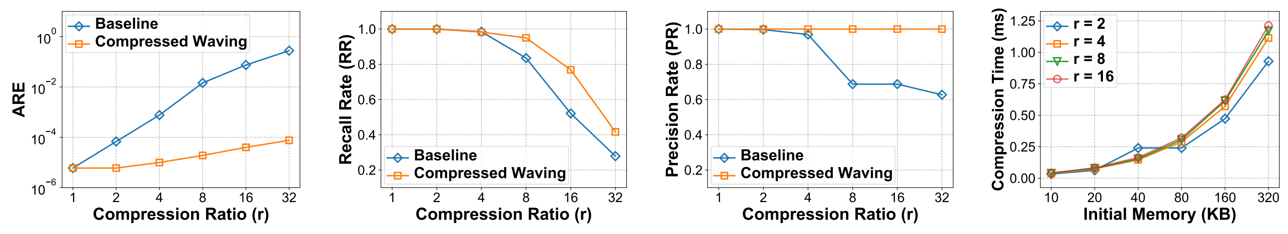
Speed (Figure 16(d)): We find that WavingSketch achieves fast compression speed. It only takes 1.24 millisecond to compress a 320KB WavingSketch by 16 times, which is negligible compared to the sketch building time. In addition, under the same initial memory, larger compression ratio r goes with more compression time.

Summary and analysis: 1) After compression, WavingSketch still has high accuracy on top-k frequent items. This is because in the compression procedure, WavingSketch attempts to maintain the information of frequent items in Heavy Parts, and discards the information of infrequent items into Waving Counters. Thus, as long as $d \times c > k$, the compressed WavingSketch can still make accurate estimation. 2) WavingSketch has fast compression speed. This is because in the compression procedure, we access each bucket only once, and do not need any hash computation. Thus, the compression time is just the time to traverse all buckets in WavingSketch.

6.3.2 Performance of Elastic Expansion

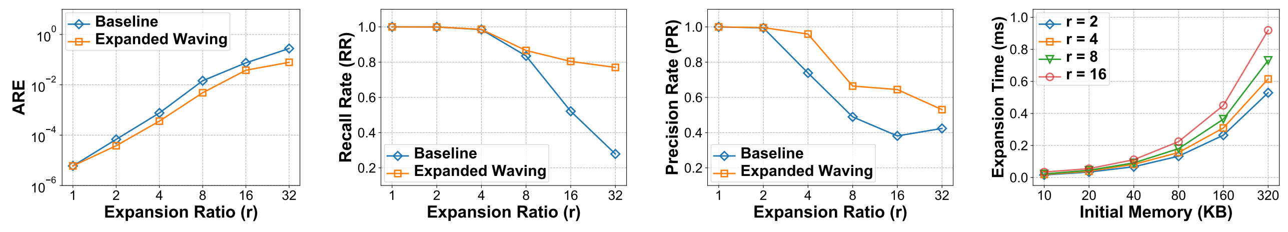
We evaluate the accuracy of the expanded WavingSketch and the expansion speed under the synthetic dataset. We set $d = 8$ and $c = 16$. In Figure 17(a)-17(c), we build WavingSketches of different memory and insert the first $\frac{1}{4}$ dataset into them. For each WavingSketch, we expand it to 320KB, and insert the remaining $\frac{3}{4}$ dataset into it. For each expansion ratio r, we compare the expanded WavingSketch W_1 with the non-expanded WavingSketch. In Figure 17(d), we evaluate the expansion time at different initial memory and expansion ratio r.

Accuracy (Figure 17(a)-17(c)): We find after expansion, the RR and PR are higher than that of the



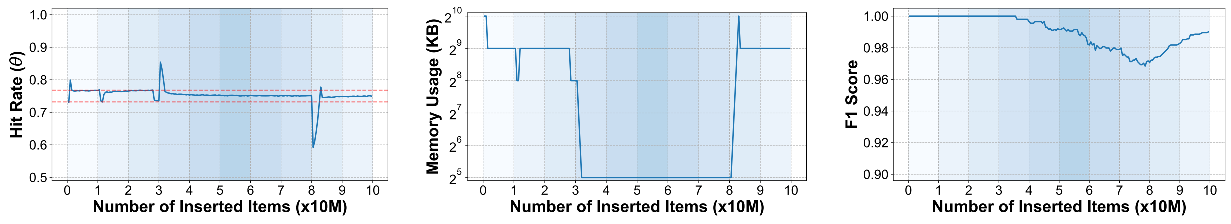
(a) ARE (b) Recall Rate (c) Precision Rate (d) Compression Time

Fig. 16 Performance of elastic compression.



(a) ARE (b) Recall Rate (c) Precision Rate (d) Expansion Time

Fig. 17 Performance of elastic expansion.



(a) Hit Rate (θ) (b) Memory Usage (c) F1 Score

Fig. 18 Performance of automatic memory adjustment.

non-expanded WavingSketch, but ARE does not show much improvement. When $r = 16$, the RR/PR of the WavingSketch expanded from 20KB to 320KB is 80.0%/ 63.2%, while that of the non-expanded WavingSketch is 52.7%/39.6%. When $r = 16$, the ARE of the WavingSketch expanded from 20KB to 320KB is 0.038, while that of the non-expanded WavingSketch is 0.074.

Speed (Figure 17(d)): We find that WavingSketch achieves fast expansion speed. It only takes < 1 millisecond to expand a 320KB WavingSketch by 16 times, which is negligible compared to the sketch building time. We also find that under the same initial memory, larger expansion ratio r goes with longer expansion time.

Summary and analysis: 1) After expansion, the accuracy of WavingSketch is significantly improved. Therefore, in practice, when the density of workload suddenly increases, we can perform the elastic expansion operation to maintain the accuracy of WavingSketch. 2) WavingSketch has fast expansion speed. This is because in the expansion procedure, we just copy the WavingSketch r times, and do not need any computation.

6.3.3 Experiments on Automatic Adjustment

We evaluate the performance of WavingSketch on automatic memory adjustment described in § 3.3. We create a synthetic Zipf [42] data stream of 100M items, and change its skewness every 10M items by varying α from 1.0 to 1.1. In Figure 18(a)-18(c), the darker the background color, the higher the skewness of the data stream. We set $k = 1000$ and set the hit rate range to $[73\%, 77\%]$ (marked as red lines in Figure 18(a)). We will see that under such range, WavingSketch always achieves high accuracy on finding top-1000 items. We set the initial memory of WavingSketch to 1024KB.

Experimental results (Figure 18): We find that WavingSketch can automatically tune its memory to adapt to the dynamic changes of data stream skewness, so that it can always maintain high accuracy ($> 97\%$ F1 score) on finding top-k items. As shown in Figure 18(a)-18(b), as the skewness increases at the 1^{st} and the $30M^{th}$ item, the hit rate θ exceeds the upper bound Θ_2, which triggers the compression operation of WavingSketch to save memory. As the skewness decreases at the $80M^{th}$ item, the hit rate θ drops below the lower bound Θ_1, which triggers the expansion operation of WavingSketch to improve the accuracy. From Figure 18(c), we can see that with the automatic memory adjustment mechanism, WavingSketch can always maintain its F1 score above 97%. In particular, as the skewness drops after the 50Mth item, the F1 score of WavingSketch also gradually drops, and at the 80Mth item, WavingSketch eventually triggers the expansion mechanism, thereby always maintaining the F1 score above a certain level.

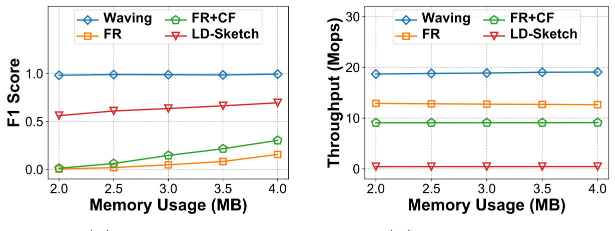
6.4 Experiments on Other Applications

6.4.1 Experiments on Other Top-*k* Applications

We show the performance of WavingSketch on three top-k tasks. On finding heavy changes, we compare WavingSketch with FlowRadar (FR) [33], FlowRadar +Cold filter (FR+CF) [67], and LD-Sketch [24]. On finding persistent items, we compare WavingSketch with PIE [18], Small-Space [29], and On-Off Sketch [63]. On finding Super-Spreaders, we compare WavingSketch with One-level Filtering (OLF) [54], Two-level Filtering (TLF) [54], OpenSketch [61], and SpreadSketch [52].

Settings: For WavingSketch, we set $d = 8$ and $c = 16$. For other algorithms, we set their parameters according to the recommendation of their authors. On finding heavy changes, we set the memory of FR and FR+CF to be 10× larger than that of WavingSketch, as they cannot decode all flows under small memory. As in § 6.1, we configure the initial memory of LD-Sketch to be the same as WavingSketch, meaning that its actual memory usage is significantly larger than that marked in Figure 19 (at least 10× larger). We use the IP Trace dataset, and use the tuple of source and destination IP addresses (4+4 bytes) of each packet as the ID field.

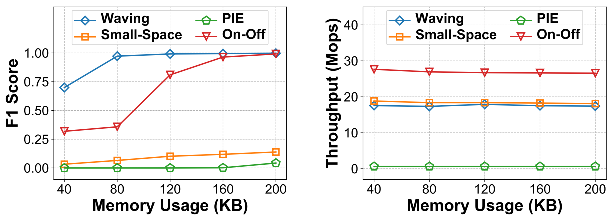
Finding heavy changes (Figure 19(a)-19(b)): We find that even just using 1/10 times of memory, the accuracy and throughput of WavingSketch are still significantly higher than that of FR, FR+CF, and LD-Sketch. When using 2MB of memory, WavingSketch has the F1 score of 98.22%. By contrast, under >20MB memory, the F1 score of FR, FR+CF, and LD-Sketch are 0.40%, 1.17%, and 56.13%, respectively. The throughput of WavingSketch is around 1.44×, 2.05×, and 43.44× higher than that of FR, FR+CF, and LD-Sketch.



(a) F1 score (b) Throughput

Fig. 19 Performance on finding heavy changes (Note that the memory of "FR" and "FR+CF" is 10× larger than that marked in the figure, and the memory of LD-Sketch is > 10× larger than that marked in the figure).

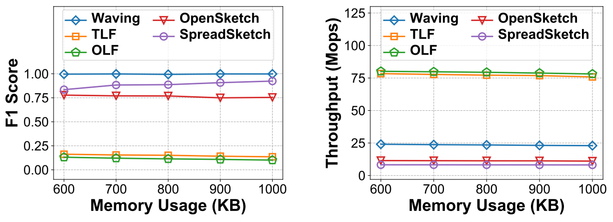
Finding persistent items (Figure 20(a)-20(b)): We find that the accuracy of WavingSketch is significantly higher than that of Small-Space, PIE, and On-Off Sketch, and the throughput of WavingSketch is close to Small-Space, which is slower than On-Off Sketch and faster than PIE. When using 80KB memory, the F1 Score of WavingSketch, Small-Space, PIE, and On-Off Sketch are 97.23%, 6.54%, 0.53%, and 35.80%, respectively. The throughput of WavingSketch is close to that of Small-Space, which is around 1.55× slower than that of On-Off Sketch and 27.95× faster than that of PIE.



(a) F1 score (b) Throughput

Fig. 20 Performance on finding persistent items.

Finding Super-Spreaders (Figure 21(a)-21(b)): We find that the accuracy of WavingSketch is higher than that of OpenSketch, TLF, OLF, and SpreadSketch, and the throughput of WavingSketch is higher than OpenSketch and SpreadSketch but lower than TLF and OLF. When using 600KB memory, the F1 score of WavingSketch, TLF, OLF, OpenSketch, SpreadSketch are 99.57%, 16.23%, 13.11%, 77.75%, and 83.43%. And the throughput of WavingSketch, TLF, OLF, OpenSketch, SpreadSketch are 24.04 Mops, 78.38 Mops, 80.16 Mops, 11.48 Mops, and 8.24 Mops.



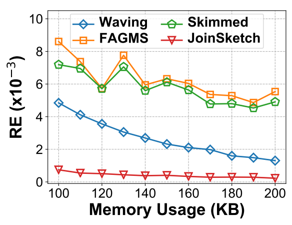
(a) F1 score (b) Throughput

Fig. 21 Performance on finding Super-Spreaders.

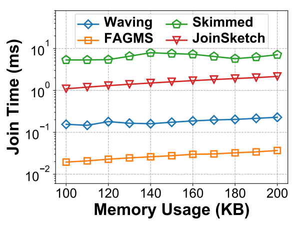
Summary and analysis: 1) On finding heavy changes, WavingSketch achieves higher accuracy than FR, FR+CF, and LD-Sketch while using $< \frac{1}{10}$ memory. This is because finding heavy changes is based on estimating the frequencies of items. Since WavingSketch provides more accurate estimation, it naturally performs better in this task. 2) On finding persistent items, WavingSketch has higher accuracy than Small-Space, PIE, and On-Off Sketch. For Small-Space, when using small memory, the inevitable undersampling magnifies its errors. For PIE, its accuracy is significantly degraded by hash collisions. For On-Off Sketch, it combines CM sketch and Space-Saving to build a top-k sketch and uses this data structure to record item *persistence*. The accuracy of this structure on persistence estimation is lower than WavingSketch, so On-Off sketch has lower F1 score. But On-Off sketch has faster speed. 3) On finding Super-Spreaders, the accuracy of WavingSketch also outperforms OpenSketch, OLF, TLF, and SpreadSketch. One main reason is that prior algorithms use a lot of memory to remove duplicates. For example, OpenSketch uses bitmaps, OLF and TLF use hash-tables. By contrast, our solution uses a Bloom filter, which is more memory-efficient. SpreadSketch also uses multi-resolution bitmap [20] to remove duplicates. Moreover, as discussed in § 2.2.4, each super-spreader can be recorded d times in its data structure, making it not memory efficient.

6.4.2 Experiments on Join-aggregate Estimation

We show the performance of WavingSketch on join-aggregate estimation. The experiments are conducted on two 1-minute IP trace datasets. We use the multi-counter WavingSketch with $d = 16$ and $c = 16$, and compare it with FAGMS [15], Skimmed sketch [22], and JoinSketch [55]. We compare the four algorithms under the same memory usage, and use the relative error (RE) as the metric, which is defined as $|J - \hat{J}|/J$ where J is the join-aggregate result of the two data streams.



(a) Relative Error



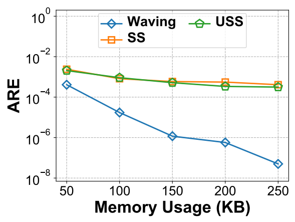
(b) Join Time

Fig. 22 Performance on join-aggregate estimation.

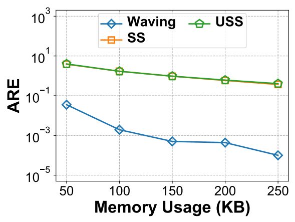
Experimental results (Figure 22): We find that the relative error of WavingSketch is smaller than FAGMS and Skimmed but larger than JoinSketch, and the processing speed of WavingSketch is faster than Skimmed and JoinSketch but slower than FAGMS. When using 100KB memory, the relative error of WavingSketch, FAGMS, Skimmed, and JoinSketch are 4.8×10^{-2}, 8.6×10^{-2}, 7.2×10^{-2}, and 7.4×10^{-3}. And the join time of WavingSketch, FAGMS, Skimmed, and JoinSketch are 0.15 ms, 0.019 ms, 5.33 ms, and 1.10 ms. FAGMS has faster speed because it does not separate frequent and infrequent items, and thus its error is the largest. On the other hand, JoinSketch separate items into three parts, and thus has the smallest error and slow speed. By contrast, WavingSketch separate items into two parts, which strikes an balance between FAGMS and JoinSketch. Therefore, its relative error and processing time are also between that of FAGMS and JoinSketch.

6.4.3 Experiments on Subset Query

We evaluate the accuracy on subset query on the synthetic dataset, and compare WavingSketch with two algorithms with overestimated error for top-k items: SS [38] and USS [53]. We build 100 subsets with a size of 1000. Each subset is built by randomly selecting 1000 items from the top-2000 items. We evaluate the error on reporting subset sum/average, and report the average relative error (ARE) on the 100 subsets.



(a) ARE on subset sum



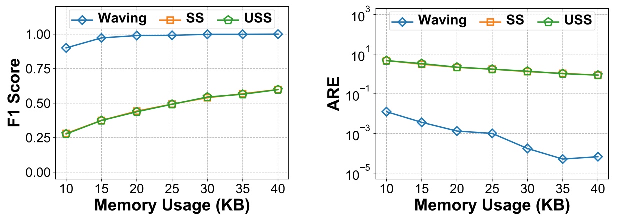
(b) ARE on subset average

Fig. 23 Performance on subset query.

Experimental results (Figure 23): We find that on both subset sum task and subset average task, WavingSketch achieves significantly smaller ARE than SS and USS. When using 200KB memory, WavingSketch, SS, and USS have 5.72×10^{-7}, 5.53×10^{-4}, and 3.38×10^{-4} ARE on subset sum task and 4.36×10^{-4}, 0.58, and 0.61 ARE on subset average task. As discussed in § 2.3, when aggregating the estimated results on a subset, the overestimated error and underestimated error of unbiased WavingSketch can offset each other, which leads to high accuracy. By contrast, the overestimated error of SS and USS will accumulate, resulting in their poor accuracy. Recall that although USS achieves the unbiasedness property for all items, its estimation for frequent items is biased upward. Therefore, when querying the aggregate results on a subset of top-k frequent items, it still suffers large accumulated error.

6.4.4 Experiments on Global Top-k Problem

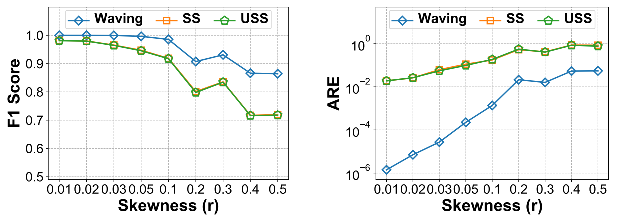
We evaluate the accuracy on finding global top-k items in disjoint data streams, and compare WavingSketch with two algorithms with overestimated error for top-k items: Space-Saving [38] and Unbiased Space-Saving [53]. Following prior work [66], we divide the synthetic dataset into N disjoint data streams $\mathcal{S}_1, \cdots, \mathcal{S}_N$ with size $m_1, \cdots, m_N$. For each algorithm, we deploy one of its copy on each data stream to detect local top-k items, and then aggregate the results to find global top-k items. Memory sizes of all the copies on different data streams are set the same. For Figure 24, we create $N = 10$ data streams of equal size by setting $m_1 = \cdots = m_N = \frac{m}{N}$, where $m = \left|\cup_{i=1}^{N}\mathcal{S}_i\right|$. We set $k = 5000$ and vary the local memory usage of candidate algorithms. For Figure 25, we create $N = 100$ data streams of skewed size by setting $m_1 = r \cdot m$ and $m_i = \frac{1-r}{N-1} \cdot m, \forall i \geqslant 2$ where $r \geqslant \frac{1}{N}$ represents the skewness of the size distribution across different data streams. In this setup, there are one heavy streams $\mathcal{S}_1$ with large size and 99 light streams $\mathcal{S}_2 \sim \mathcal{S}_{100}$ with small size. We set $k = 2000$ and set the memory of each algorithm on each data stream to 10KB.



(a) F1 Score (b) ARE

Fig. 24 Performance of finding global top-k items in data streams of equal size.

Performance on data streams of equal size (Figure 24): We find that compared to the biased algorithms (SS, USS), our unbiased WavingSketch achieves significantly higher F1 score and smaller ARE. When using 15KB local memory, the F1 score of WavingSketch, SS, and USS are 97.27%, 37.49%, and 37.42%, and the ARE for global top-k items of WavingSketch, SS, and USS are 3.6×10^{-3}, 3.09, and 3.32. SS and USS have poor accuracy because they provide highly overestimated frequency estimation under small memory, so they cannot accurately find global top-k items. By contrast, our unbiased WavingSketch maintains high accuracy even under small memory.



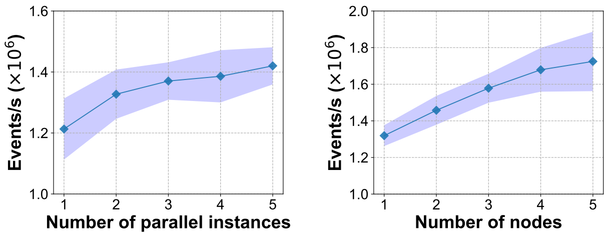
(a) F1 Score (b) ARE

Fig. 25 Performance of finding global top-k items in data streams of skewed size.

Performance on data streams of skewed size (Figure 25): We find that compared to the biased algorithms (SS, USS), our unbiased WavingSketch achieves significantly higher F1 score and smaller ARE. On the skewness $r = 0.1$, the F1 score of WavingSketch, SS, and USS are 98.57%, 91.86%, and 91.47%, and the ARE for global top-k items of WavingSketch, SS, and USS are 1.36×10^{-3}, 0.176, and 0.186. As discussed in § 2.3, SS and USS have poor accuracy because of their overestimated error for local top-k items. In this way, local top-k candidates in heavy streams tend to be highly overestimated, so even if an item in heavy stream has small real frequency, its estimated frequency might still be high enough to be falsely selected as a global top-k item. With items in heavy streams falsely selected as global top-k items and items in light streams ignored, the F1 Scores of SS and USS become unacceptably low when skewness is large. By contrast, for unbiased WavingSketch, the frequency of a local top-k item has the same chance to be overestimated or underestimated, so the global top-k result is no longer influenced by the size of local data stream. Therefore, WavingSketch achieves higher F1 score under data streams of skewed size.

6.5 Experiments on Apache Flink

We implement WavingSketch on Apache Flink [13], showing WavingSketch can be easily integrated into modern stream processing framework. We build a Flink cluster with 1 master node and 5 worker nodes. We deploy one WavingSketch on each of these nodes, and evaluate the streaming processing speed in both local mode and cluster mode. As shown in Figure 26, WavingSketch achieves satisfactory throughput ($1.2 \sim 1.8$ million events per second) in our Flink cluster. We present the details of the experimental setup and discuss the experimental results in our supplementary materials [5].



(a) Local mode (b) Cluster mode

Fig. 26 Throughput of WavingSketch on Apache Flink.

7 Conclusion

In this paper, we propose an algorithm called WavingSketch for finding top-k items. WavingSketch provides unbiased estimation and outperforms the state-of-the-art algorithm on both accuracy and speed. We theoretically prove the unbiasedness property of WavingSketch and analyze its error, and we apply WavingSketch to five applications. Experimental results show that, compared with Unbiased Space-Saving, WavingSketch achieves 10× faster insertion speed and $10^3\times$ smaller error in finding frequent items.

Acknowledgements This work is supported by National Key R&D Program of China (No. 2022YFB2901504), National Natural Science Foundation of China (NSFC) (No. U20A20179, 62372009, 623B2005), and research grant No. SH-2024JK29.

References

1. CAIDA [on line]. Available: `http://www.caida.org/home`.
2. Murmur hashing source codes. `https://github.com/aappleby/smhasher/blob/master/src/MurmurHash3.cpp`.
3. Real-life transactional dataset. `http://fimi.ua.ac.be/data/`.
4. Source code related to WavingSketch. `https://github.com/WavingSketch/Waving-Sketch`.
5. Supplementary materials of wavingsketch. `https://github.com/WavingSketch/Waving-Sketch/blob/master/WavingSketch_Supplementary.pdf`.
6. The Network dataset Internet Traces. `http://snap.stanford.edu/data/`.
7. N. Alon, P. B. Gibbons, Y. Matias, and M. Szegedy. Tracking join and self-join sizes in limited storage. In *Proceedings of the eighteenth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 10–20, 1999.
8. A.Shokrollahi. Raptor codes. *IEEE Transactions Information Theory*, 52(6), 2006.
9. 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.
10. B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, 1970.
11. T. Buddhika, M. Malensek, S. L. Pallickara, and S. Pallickara. Synopsis: A distributed sketch over voluminous spatiotemporal observational streams. *IEEE Transactions on Knowledge and Data Engineering*, 29(11):2552–2566, 2017.
12. W. Cai, M. Balazinska, and D. Suciu. Pessimistic cardinality estimation: Tighter upper bounds for intermediate join cardinalities. In *Proceedings of the 2019 International Conference on Management of Data*, pages 18–35, 2019.
13. P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas. Apache flink: Stream and batch processing in a single engine. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, 36(4), 2015.
14. M. Charikar, K. Chen, and M. Farach-Colton. Finding frequent items in data streams. In *Automata, Languages and Programming*. Springer, 2002.
15. G. Cormode and M. Garofalakis. Sketching streams through the net: Distributed approximate query tracking. In *Proceedings of the 31st international conference on Very large data bases*, pages 13–24, 2005.
16. G. Cormode and M. Hadjieleftheriou. Finding frequent items in data streams. *Proceedings of the VLDB Endowment*, 1(2):1530–1541, 2008.
17. G. Cormode and S. Muthukrishnan. An improved data stream summary: the count-min sketch and its applications. *Journal of Algorithms*, 55(1), 2005.
18. H. Dai, M. Shahzad, A. X. Liu, and Y. Zhong. Finding persistent items in data streams. *Proceedings of the VLDB Endowment*, 10(4):289–300, 2016.
19. C. Estan and G. Varghese. New directions in traffic measurement and accounting. *ACM SIGCOMM CCR*, 32(4), 2002.
20. C. Estan, G. Varghese, and M. Fisk. Bitmap algorithms for counting active flows on high speed links. In *Proceedings of the 3rd ACM SIGCOMM conference on Internet measurement*, pages 153–166, 2003.
21. M. Flynn. Some computer organizations and their effectiveness. ieee trans comput c-21:948. *Computers, IEEE Transactions on*, C-21:948 – 960, 10 1972.
22. S. Ganguly, M. Garofalakis, and R. Rastogi. Processing data-stream join aggregates using skimmed sketches. In *International Conference on Extending Database Technology*, pages 569–586. Springer, 2004.
23. S. Ganguly, D. Kesh, and C. Saha. Practical algorithms for tracking database join sizes. In *International Conference on Foundations of Software Technology and Theoretical Computer Science*, pages 297–309. Springer, 2005.
24. Q. Huang and P. P. Lee. Ld-sketch: A distributed sketching design for accurate and scalable anomaly detection in network data streams. In *IEEE INFOCOM 2014-IEEE Conference on Computer Communications*, pages 1420–1428. IEEE, 2014.
25. Y. Izenov, A. Datta, F. Rusu, and J. H. Shin. Compass: Online sketch-based query optimization for in-memory databases. In *Proceedings of the 2021 International Conference on Management of Data*, pages 804–816, 2021.
26. R. M. Karp, S. Shenker, and C. H. Papadimitriou. A simple algorithm for finding frequent elements in streams and bags. *ACM Transactions on Database Systems (TODS)*, 28(1):51–55, 2003.
27. W. Kim, J. Yun, and H. Jung. Evaluation of high-frequency financial transaction processing in distributed memory systems. In *Proceedings of the 2014 Conference on Research in Adaptive and Convergent Systems*, pages 362–364, 2014.
28. K. Kutzkov, M. Ahmed, and S. Nikitaki. Weighted similarity estimation in data streams. In *Proceedings of the 24th ACM International on Conference on Information and Knowledge Management*, pages 1051–1060, 2015.
29. B. Lahiri, J. Chandrashekar, and S. Tirthapura. Space-efficient tracking of persistent items in a massive data stream. *Statistical Analysis and Data Mining*, 7:70–92, 2011.
30. V. Leis, A. Gubichev, A. Mirchev, P. Boncz, A. Kemper, and T. Neumann. How good are query optimizers, really? *Proceedings of the VLDB Endowment*, 9(3):204–215, 2015.
31. V. Leis, B. Radke, A. Gubichev, A. Mirchev, P. Boncz, A. Kemper, and T. Neumann. Query optimization through the looking glass, and what we found running the join order benchmark. *The VLDB Journal*, 27(5):643–668, 2018.
32. J. Li, Z. Li, Y. Xu, S. Jiang, T. Yang, B. Cui, Y. Dai, and G. Zhang. Wavingsketch: An unbiased and generic sketch for finding top-k items in data streams. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 1574–1584, 2020.

33. Y. Li, R. Miao, C. Kim, and M. Yu. Flowradar: a better netflow for data centers. In *USENIX NSDI*, pages 311–324. USENIX Association, 2016.
34. Y. Li, F. Wang, X. Yu, Y. Yang, K. Yang, T. Yang, Z. Ma, B. Cui, and S. Uhlig. Ladderfilter: Filtering infrequent items with small memory and time overhead. *Proceedings of the ACM on Management of Data*, 1(1):1–21, 2023.
35. Z. Liu, C. Kong, K. Yang, T. Yang, R. Miao, Q. Chen, Y. Zhao, Y. Tu, and B. Cui. Hypercalm sketch: One-pass mining periodic batches in data streams. In *2023 IEEE 39th International Conference on Data Engineering (ICDE). IEEE*, 2023.
36. Z. Liu, A. Manousis, G. Vorsanger, V. Sekar, and V. Braverman. One sketch to rule them all: Rethinking network flow monitoring with univmon. In *Proceedings of the 2016 ACM SIGCOMM Conference*, pages 101–114, 2016.
37. G. Lukasz, D. David, D. E. D, L. Alejandro, and M. J. Ian. Identifying frequent items in sliding windows over on-line packet streams. In *IMC*. ACM, 2003.
38. A. Metwally, D. Agrawal, and A. El Abbadi. Efficient computation of frequent and top-k elements in data streams. In *International Conference on Database Theory*. Springer, 2005.
39. R. Miao, Y. Zhang, G. Qu, K. Yang, T. Yang, and B. Cui. Hyper-uss: Answering subset query over multi-attribute data stream. In *Proceedings of the 29th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, pages 1698–1709, 2023.
40. M. Nishad and P. Themis. Frequent items in streaming data: An experimental evaluation of the state-of-the-art. *Data & Knowledge Engineering*, 2009.
41. K. Park and H. Lee. On the effectiveness of route-based packet filtering for distributed dos attack prevention in power-law internets. *ACM SIGCOMM computer communication review*, 31(4):15–26, 2001.
42. D. M. Powers. Applications and explanations of Zipf's law. In *Proc. EMNLP-CoNLL*. Association for Computational Linguistics, 1998.
43. R. Pratanu, K. Arijit, and A. Gustavo. Augmented sketch: Faster and more accurate stream processing. In *Proc. ACM SIGMOD*, 2016.
44. G. Pruthi, F. Liu, S. Kale, and M. Sundararajan. Estimating training data influence by tracing gradient descent. *Advances in Neural Information Processing Systems*, 33:19920–19930, 2020.
45. Y. Rekhter, T. Li, and S. Hares. A border gateway protocol 4 (bgp-4). Technical report, 2006.
46. A. Rousskov and D. Wessels. High-performance benchmarking with web polygraph. *Software: Practice and Experience*, 2004.
47. F. Rusu and A. Dobra. Statistical analysis of sketch estimators. In *Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, pages 187–198, 2007.
48. F. Rusu and A. Dobra. Sketches for size of join estimation. *ACM Transactions on Database Systems (TODS)*, 33(3):1–46, 2008.
49. R. Schweller, Z. Li, Y. 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.
50. M. G. Singh and M. Rajeev. Approximate frequency counts over data streams. In *Proc. VLDB*, pages 346–357, 2002.
51. J. L. Sobrinho. Network routing with path vector protocols: Theory and applications. In *Proceedings of the 2003 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 49–60, 2003.
52. L. Tang, Q. Huang, and P. P. Lee. Spreadsketch: Toward invertible and network-wide detection of superspreaders. In *IEEE INFOCOM 2020-IEEE Conference on Computer Communications*, pages 1608–1617. IEEE, 2020.
53. D. Ting. Data sketches for disaggregated subset sum and frequent item estimation. In *SIGMOD Conference*, 2018.
54. S. Venkataraman, D. X. Song, P. B. Gibbons, and A. Blum. New streaming algorithms for fast detection of superspreaders. In *NDSS*, 2005.
55. F. Wang, Q. Chen, Y. Li, T. Yang, Y. Tu, L. Yu, and B. Cui. Joinsketch: A sketch algorithm for accurate and unbiased inner-product estimation. *Proceedings of the ACM on Management of Data*, 1(1):1–26, 2023.
56. Y. Wang and K. Yi. Secure yannakakis: Join-aggregate queries over private data. In *Proceedings of the 2021 International Conference on Management of Data*, pages 1969–1981, 2021.
57. Z. Wei, G. Luo, K. Yi, X. Du, and J.-R. Wen. Persistent data sketching. In *Proc. ACM SIGMOD*, pages 795–810. ACM, 2015.
58. D. Yang, B. Li, L. Rettig, and P. Cudré-Mauroux. D ̃22 histosketch: Discriminative and dynamic similarity-preserving sketching of streaming histograms. *IEEE Transactions on Knowledge and Data Engineering*, 31(10):1898–1911, 2018.
59. T. Yang, J. Gong, H. Zhang, L. Zou, L. Shi, and X. Li. Heavyguardian: Separate and guard hot items in data streams. In *SIGKDD*, 2018.
60. T. Yang, J. Jiang, P. Liu, Q. Huang, J. Gong, Y. Zhou, R. Miao, X. Li, and S. Uhlig. Elastic sketch: Adaptive and fast network-wide measurements. In *ACM SIGCOMM 2018*, pages 561–575, 2018.
61. M. Yu, L. Jose, and R. Miao. Software defined traffic measurement with opensketch. In *NSDI 2013*, 2013.
62. H. Zhang, Z. Liu, B. Chen, Y. Zhao, T. Zhao, T. Yang, and B. Cui. Cafe: Towards compact, adaptive, and fast embedding for large-scale recommendation models. *In Proceedings of the 2024 ACM International Conference on Management of Data (SIGMOD)*, 2024.
63. Y. Zhang, J. Li, Y. Lei, T. Yang, Z. Li, G. Zhang, and B. Cui. On-off sketch: A fast and accurate sketch on persistence. *Proceedings of the VLDB Endowment*, 14(2):128–140, 2020.
64. Y. Zhang, Z. Liu, R. Wang, T. Yang, J. Li, R. Miao, P. Liu, R. Zhang, and J. Jiang. Cocosketch: High-performance sketch-based measurement over arbitrary partial key query. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, pages 207–222, 2021.
65. B. Zhao, X. Li, B. Tian, Z. Mei, and W. Wu. Dhs: Adaptive memory layout organization of sketch slots for fast and accurate data stream processing. In *Proceedings of the 27th ACM SIGKDD Conference on Knowledge Discovery & Data Mining*, pages 2285–2293, 2021.
66. Y. Zhao, W. Han, Z. Zhong, Y. Zhang, T. Yang, and B. Cui. Double-anonymous sketch: Achieving top-k-fairness for finding global top-k frequent items. *Proceedings of the ACM on Management of Data*, 1(1):1–26, 2023.
67. Y. Zhou, T. Yang, J. Jiang, B. Cui, M. Yu, X. Li, and S. Uhlig. Cold filter: A meta-framework for faster and more accurate stream processing. In *SIGMOD Conference*, 2018.
68. Y. Zhou, T. Yang, J. Jiang, B. Cui, M. Yu, X. Li, and S. Uhlig. Cold filter: A meta-framework for faster and more accurate stream processing. In *Proceedings of the 2018 International Conference on Management of Data*, pages 741–756, 2018.