Finding Interesting Items in Data Streams

Anonymous Author(s)

ABSTRACT

In high-speed data streams, a small fraction of items that have specific characteristics are often the focus, such as frequent items, heavy changes, Super-Spreaders, or persistent items. We call them interesting items. Most existing algorithms are designed for only one specific characteristic/interest, and use different data structures for different interests. Existing algorithms can be divided into two kinds. The first kind records all information of the stream, and thus is not memory efficient. The second kind manages to only record the interesting items, but it is challenging to differentiate interesting items from others.

In this paper, we generalize these characteristics into one aspect, which we call finding interesting items in data streams. To find interesting items, we propose a generic framework, InterestSketch. InterestSketch manages to record only interesting items. To address the challenge of differentiating interesting items from others, we propose a key technique called Probabilistic Replacement and then Increment (PRI). PRI is designed to record only interesting items with high accuracy and high speed, using limited memory space. The key idea of PRI is as follows. To insert a new item, we *replace the current smallest item in the min-heap with a dynamic probability $\mathcal{P}$. If the replacement is successful, the smallest interest (e.g., frequency) is incremented.* Otherwise, we do not change the smallest item, but increase the probability $\mathcal{P}$. Our theoretical proofs show that when replacement is successful, with high probability, the new item has a higher interest than the current smallest interest.

We conduct extensive experiments on three real datasets and one synthetic dataset, on four definitions of interesting items. Our experimental results show that compared with the state-of-the-art, for each definition of interest, our algorithm increases the insertion speed 2.2 ∼ 7.7 times and decreases the error 74 ∼ 3207 times. All related code and datasets are open-source and available at Github anonymously [1].

1 INTRODUCTION

1.1 Motivation

Big data often comes in the form of high-speed data streams. A data stream is made of continuously arriving items, each of which appearing one or multiple times. Although there are typically many items, people are often interested in a very limited number of them with some special characteristics, such as frequent items [2–5], heavy changes[6–8], Super-Spreaders [9], or persistent items [10, 11]. In this paper, we call them *interesting items*.

Finding interesting items is not new. For instance, the problem of finding frequent items is an old but still active topic in fields such as data bases and data mining. Finding Super-Spreaders and heavy changes are critical problems in security [12]. Finding persistent items on the other hand is relatively recent. Note that interest can be defined in other ways. For example, to find the destination IP addresses with a large number of source IP addresses is fundamental in DDoS detection.

Finding interesting items is challenging, mainly because of the high speed of the data streams. Therefore, it is often impossible to find interesting items without errors. Fortunately, small and controllable errors are often acceptable in practice. This is why sketches, a type of probabilistic data structure, have recently been used for such problems [10, 13–17].

1.2 Prior Art and Their Limitations

Existing sketches often focus on one specific type of interest. For different interests, existing solutions use different data structures. Four common definitions of interest exist, corresponding to four tasks. For each of the four tasks, existing algorithms can be divided into two types. The first is called `record all`: recording all information of the stream. The second records a part of information of the stream, recording only ***hot items*** or relying on sampling. In this paper, we call the items with high interest ***hot items*** and the items with low interest ***cold items***.

1) Finding Frequent Items: Interest is defined as *frequency*, *i.e.*, the number of appearances of an item. The task is to find items with large frequencies. To find frequent items, two types of solutions exist. The first type, `record all`, records the frequencies of all items. Typical algorithms are made of a sketch (*e.g.*, sketches of CM [18], CU [19], and Count [5]) plus a min-heap, and ASketch [13]. Recording the frequencies of cold items is unnecessary. The second kind only records hot items: the information of items with large frequencies. Typical algorithms are SpaceSaving [20], Unbiased SpaceSaving [15], Lossy Counting [21], and Cold filter [16]. SpaceSaving and the Unbiased SpaceSaving use a min-heap-like data structure to record hot items. When the min-heap is full, the smallest frequency is incremented by one irrespective of the incoming item being cold or hot. However, in practice, there are many cold items that have a *negative impact* on the recorded frequencies, leading to relatively poor accuracy. Our algorithm belongs to the second kind, but manages to minimize this negative impact without using any additional data structure.

2) Finding Heavy Changes: Here, a data stream is equally divided into n periods (also known as time windows or intervals). We define interest as *change of frequency*, *i.e.*, the difference of frequency of an item in two adjacent periods. Again, the first type of solution is to `record all`, including k-ary [6], the reversible sketch [7], and FlowRadar [8]. They build one data structure to record all items in each period, and then manage to decode and report heavy changes. The second kind manages to record only hot items, and the typical algorithm being Cold filter [16]. Cold filter first uses a filter to filter cold items, and then focuses on hot items. We aim to achieve a higher accuracy than Cold filter without any additional data structure.

3) Finding Super-Spreaders: Each item is a packet with a source IP address and a destination IP address. We define interest as *connections*, *i.e.*, the number of destination IP addresses for a given source IP address. The problem is to find source IP addresses with large number of connections. Again, two kinds of solutions exist. The first, `record all`, records the information of all packets. Typical algorithms are Two-dimension bitmap [22] and OpenSketch [14]. The second kind, `record samples`, samples packets before recording the information of packets. Sampling achieves memory efficiency at the cost of poor accuracy. The typical algorithms here are called one-level filtering [9] and two-level filtering [9]. We aim to record only Super-Spreaders without sampling, to achieve a higher accuracy.

4) Finding Persistent Items: Here, a data stream is equally divided into n periods. We define interest as *persistency*, *i.e.*, the number of periods in which the item appears. In each period of the stream, the persistency of an item is incremented only once or not changed. The problem is to find items with high persistency. Again, two types of solutions exist. The first, `record all`, records the information of all packets. The typical algorithm is PIE [11]. The second kind, `record samples`, samples before recording the information of items. The typical algorithm is small-space [23]. We aim to record only persistent items to achieve a higher accuracy.

1.3 Our Solution

In this paper, we propose a generic framework, named InterestSketch, which can be used for finding interesting items, such as frequent items, heavy changes, Super-Spreaders, and persistent items.

Now we use a simple example to explain our key idea of differentiating hot items from others. Let us consider the following problem: given a data stream, how to find the most frequent item with only one bucket? The bucket has two fields: item ID and frequency. The key operation lies in the following situation: the incoming item e_1 is different from `the original item` e_0 with frequency f_0 kept in the bucket. The most widely used algorithm, SpaceSaving [20], just replaces e_0 with e_1, and increments the frequency from f_0 to $f_0 + 1$. In contrast, our technique is called ***Probabilistic Replace then Increment (PRI)***: replacing e_0 with the incoming item e_1 with a replacement probability $\mathcal{P}$. If e_1 successfully replaces e_0, we increment f_0 by $\lfloor t_{fail}/f_0 \rfloor$ and reset t_{fail} to 0, where t_{fail} is the number of replacement failures. Otherwise, we increment t_{fail} by 1. After each unsuccessful replacement, the replacement probability $\mathcal{P}$ increases. Further, to make replacement as correct as possible, i.e., given that we want only hot items to replace cold items, *the value of* $\mathcal{P}$ *decreases as* f *increases*. More details about $\mathcal{P}$ are provided in Section 3.1.

Analysis: SpaceSaving supposes the incoming item is always hotter than the original item in the bucket. At the end, the item kept in the bucket must be the last incoming one and the interest is the sum of the interests of all distinct items in the data stream. The unbiased SpaceSaving [15] first increments the frequency and then tries replacement, achieving unbiased error at the cost of poor accuracy. In contrast, we differentiate hot and cold items by using PRI. Compared with cold items, hot items have a higher probability of replacing the original item. After each successful replacement, incrementing the frequency is reasonable.

Main Experimental Results: We compare our InterestSketch with the state-of-the-art algorithms in each of the aforementioned four tasks. In finding frequent items, InterestSketch reduces the error 74 ~ 3207 times and improves the insertion speed 2.2 ~ 7.7 times. In finding heavy changes, InterestSketch improves the precision 3.6 ~ 6.2 times when using only 1/20 of the memory size of other algorithms and improves the insertion speed 2.1 ~ 3.0 times. In finding Super-Spreaders, InterestSketch reduces the error 18 ~ 31 times. In finding persistent items, InterestSketch reduces the error 115 ~ 50212 times when using only 1/20 of the memory size of other algorithms and increases the insertion speed 1.4 ~ 4 times.

1.4 Key Contribution

- We propose a generic framework called InterestSketch, which can find interesting items with high accuracy and high speed using small memory.
- To verify the generality of our framework, we apply the framework to four specific tasks, including finding frequent items, finding heavy changes, finding Super-Spreaders, and finding persistent items.
- We derive the error bounds and several theoretical properties of InterestSketch.
- We conduct extensive experiments on three real datasets and one synthetic dataset. Our results show that InterestSketch significantly outperforms all existing algorithms in terms of both accuracy and speed.

2 PROBLEM STATEMENT

This section show the definitions of typical tasks, and the related work briefly introduced in the Introduction is described in detail in Appendix A.

Finding Interesting Items: Given a data stream $\mathcal{S}$ consisting of items, each item can appear more than once. We use "interest" to describe one property of every item. Finding interesting items means to report all the items with interests larger than a given threshold. Different definitions of interest correspond to different tasks of data streams. Next, we show four typical tasks.

Finding Frequent Items: In finding frequent items, the interest is defined as **frequency**, *i.e.*, the number of appearances of each item. Finding frequent items consists in finding items whose frequencies are larger than a given threshold.

Finding Heavy Changes: Given a data stream, we divide it into equal-sized periods. In finding heavy changes, the interest is defined as **change of frequency**, *i.e.*, the change of frequency of an item in two adjacent periods. The frequencies of some items could drastically change between two adjacent periods. Analyzing such changes is important in security [7, 8]. Finding heavy changes consists in finding items whose changes of frequency are larger than a given threshold.

Finding Super-spreaders: In computer networks, each item is a packet with a source IP address and a destination IP address. A specific source IP address can send packets to many destination IP addresses. When the interest is defined as **connections**, i.e., the number of destination IP addresses for a given source IP address, the problem will be about finding super-spreaders.

Finding Persistent Items: Again, given a data stream, we equally divide it into equal size periods. For a specific item, its **persistency** indicates the number of periods in which the item appears. In each period, if an item appears, either once or more, its persistency is incremented by one. When we define interest as persistency, the problem will be to find persistent items.

3 INTERESTSKETCH: BASICS

In this section, we first present our generic framework – InterestSketch, and then show how to use our framework to find frequent items, heavy changes, super-spreaders, and persistent items.

3.1 The InterestSketch Framework

To make our novelty clear, we first show the basic framework. It is simple: a filter (optional) and a min-heap. "Optional" means that some tasks do not need the filter. Our key novelty is ***the PRI technique***. In next Section (Section 4), we will replace the min-heap to minimize the overhead of both time and space.

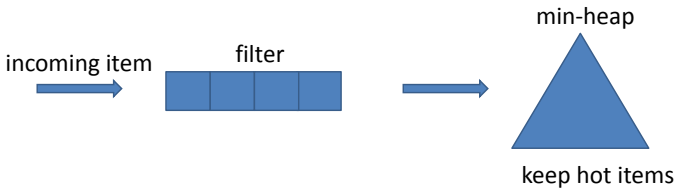


Figure 1: The InterestSketch framework.

Data Structure (Figure 1): The InterestSketch framework consists of two parts. The first is a Bloom filter [24], used to remove duplicates in the incoming items. Duplicates removal is necessary because the interest $\mathcal{I}$ might not be incremented for every incoming item. For example, when finding persistent items, in each period, each incoming item can only increment the corresponding persistency by one, even though it might appear more than once in that period. The second part is a min-heap keeping hot items. In the min-heap, each node stores the information of an item, including item ID and interest $\mathcal{I}$. The root node stores the item with the smallest interest.

A Bloom filter [24] is a compact data structure consisting of a number of bits and is often used when judging whether an item exists in a set or not. It is associated with z hash functions. There are mainly two operations for this data structure. The first is to insert an item e. The z hash functions are first computed to pick out z bits in the Bloom filter, and then all the z bits are set to 1. The second operation is to judge whether an item belongs to the set. The same z hash functions are first computed to get the z bits, and only if all the z bits are 1, the Bloom filter reports true; otherwise, it reports false. If the item is indeed in the set, true is always reported, *i.e.*, it has no false negative error. In some cases, when an item does not belong to the set, the Bloom filter might also report true, which is called as false positive. However, the probability of false positive is often small enough to be acceptable in practice. Therefore, Bloom filters are used in various fields, especially in approximate data stream processing.

Insertion: Given an incoming item e, we first check the Bloom filter to judge whether e is a duplicate: if the Bloom filter reports true, which means e is a duplicate, and then e is discarded. Otherwise, e is inserted into the Bloom filter, and then we insert e in the min-heap. There are two cases:

Case 1: The item e is in the min-heap. In this case, we increment the interest $\mathcal{I}$ by one.

Case 2: e is not in the min-heap. If the min-heap is not full, we insert e into the min-heap. If the min-heap is full, we propose a technique named Probabilistic Replacement and then Increment (PRI for short) to *probabilistically judge whether the incoming item is hot or cold.*

Probabilistic Replacement and then Increment (PRI): This technique is the key novelty in this paper. Our PRI works as follows. Suppose that the root node of the min-heap stores item e_{min} with interest $\mathcal{I}_{min}$. Given an incoming item e which is not in the min-heap, we replace e_{min} with incoming item e with a probability

$$\mathcal{P} = \frac{1}{2 * \mathcal{I}_{min} - t_{fail} + 1}$$

where t_{fail} is the number of replacement failures. If e_{min} is successfully replaced by e, indicating that the interest of e is likely to be larger than e_{min}, we increment the interest of e_{min} from $\mathcal{I}_{min}$ to $\mathcal{I}_{min} + \lfloor t_{fail}/\mathcal{I}_{min} \rfloor$ and set t_{fail} to 0. Otherwise, t_{fail} is incremented by 1. For convenience, $\mathcal{I}_{min} + \lfloor t_{fail}/\mathcal{I}_{min} \rfloor$ is abbreviated to $\mathcal{I}_{min} + t_{fail}/\mathcal{I}_{min}$ in this paper.

Designing the Expression of $\mathcal{P}$: We carefully design the expression of $\mathcal{P}$ to meet the following five properties. 1) To successfully replace the original item, the value of t_{fail} should reach an expectation of $\mathcal{I}_{min}$. 2) The larger $\mathcal{I}_{min}$ is, the less likely it is to be replaced. 3) The more replacement failures there are, the more likely the replacement will happen. 4) When the number of replacement failures reaches $2 * \mathcal{I}_{min}$, the probability $\mathcal{P}$ increases to 1. This can avoid too many replacement failures. 5) When the replacement happens when number of replacement failures is small, $t_{fail}/\mathcal{I}_{min}$ =0, we do not increase the smallest frequency. When the replacement happens when number of replacement failures is large, $t_{fail}/\mathcal{I}_{min}$ =2, we increase the smallest frequency by 2. In other cases, we increase the smallest frequency by 0 or 1.

Report: To report the items above a given threshold $\mathcal{T}$, we simply traverse the min-heap to return the items with interests larger than $\mathcal{T}$ (see Algorithm 1).

Every time the interest of the item changes in the min-heap, we should adjust the min-heap. Therefore, we adjust the min-heap in lines 8^{th}, 12^{th}, and 18^{th}. Lines 14 to 20 of Algorithm 1 are the implementation of the PRI technique. Besides, in line 14, we calculate the probability $\mathcal{P}$ by producing a random number between 0 and 1, because the probability that the random number is less than $\frac{1}{2*f_{min}-t_{fail}+1}$ is approximately equal to $\mathcal{P}$.

Next, we apply our generic framework to four specific problems in the following sections.

3.2 Finding Frequent Items

Data Structure (Figure 2): InterestSketch for finding frequent items only uses a min-heap. No filter is necessary since there is no need to remove duplicates. Here the interest is the frequency, *i.e.*, the number of appearances of an item.

Insertion: Given an incoming item, we check whether it is in the min-heap. If the item is in the min-heap, we increment the corresponding frequency by one. Otherwise, if the min-heap is not full, we insert the item into the min-heap. If the min-heap is full, we use the PRI technique.

Algorithm 1: Insertion process of InterestSketch.

```
   Input: An item e_i
1  random() ∈ [0, 1]
2  if e_i ∈ Bloom_filter then
3  |  return;
4  else
5  |  Bloom_filter.insert(e_i);
6  |  if e_i ∈ min_heap then
7  |  |  Interest(e_i) + +;
8  |  |  min_heap.adjust();
9  |  else
10 |  |  if min_heap has empty buckets then
11 |  |  |  min_heap.insert(e_i);
12 |  |  |  min_heap.adjust();
13 |  |  else
14 |  |  |  if random() ⩽ 1/(2*I_min - t_fail + 1) then
15 |  |  |  |  v_min ← e_i;
16 |  |  |  |  I_min ← I_min + t_fail/I_min;
17 |  |  |  |  t_fail ← 0;
18 |  |  |  |  min_heap.adjust();
19 |  |  |  else
20 |  |  |  |  t_fail + +;
```

Report: To report the items above a given threshold $\mathcal{T}$, we traverse the min-heap to return the items with frequencies larger than $\mathcal{T}$.

Example 1 (Figure 2): Given a data stream (stream 1): q, e, g, e, g, g, v, for the first incoming item q, we increment the frequency of q in the min-heap by one. For the following two items e and g, replacements fail, and thus t_{fail} is incremented to 1 and then 2. The probability increases to 1/3 for the fourth item e. Then e_{min} is successfully replaced by the fourth item e in the root node, and the frequency f_{min} is incremented by $\frac{t_{fail}}{f_{min}} = 1$, from 2 to 3, and t_{fail} is reset to 0. Since the actual frequency of item e is 2 at present but we record it as 3, the frequency is slightly overestimated. Then the following 3 items (g, g, v) arrive and replace e with probabilities 1/7, 1/6, and 1/5, but all fail. Finally, the eighth item g successfully replaces e and the frequency f_{min} is incremented from 3 to 4, which is exactly the frequency of item g by now in data stream 1.

Example 2 (Figure 2): Given another data stream (stream 2): $g, r, v, e, e, e, e, e, e$, suppose that each of the first three items successfully replaces the root node with the same probability

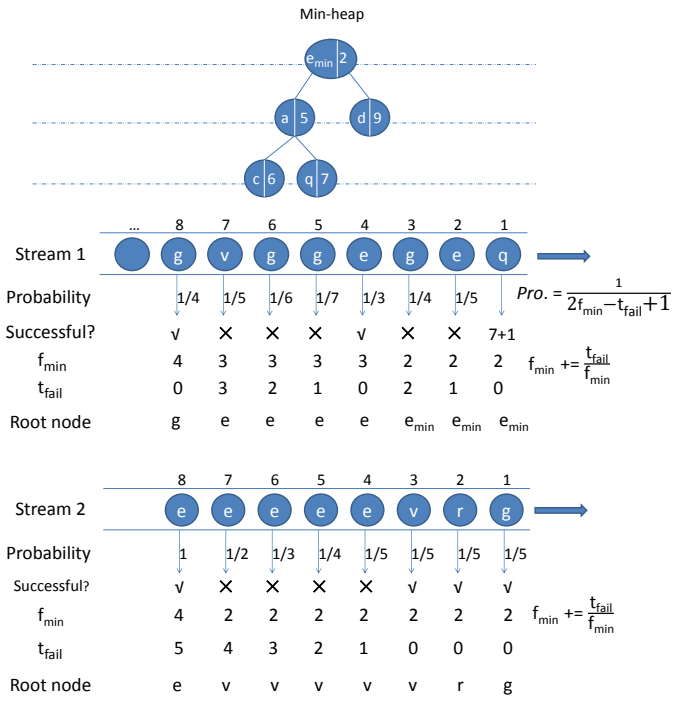


Figure 2: InterestSketch for finding frequent items.

of 1/5, and increments the frequency f_{min} by $\frac{t_{fail}}{f_{min}}(=0)$ each time. Then item e arrives five times in succession. After four unsuccessful replacements with probabilities 1/5, 1/4, 1/3, and 1/2, the eighth item e replaces item v in the root node with probability 1, and increments the frequency f_{min} by $\frac{t_{fail}}{f_{min}}(=2)$, from 2 to 4. The frequency f_{min} is slightly underestimated, since item e has appeared 5 times in the data stream by now. The first three replacements are wrong, but do not bring a large error to the final value of f_{min}.

From the two examples, we see that in our algorithm, the frequency f_{min} might be overestimated or underestimated. Fortunately, an item might be overestimated at first, but it could be underestimated later, and finally the estimate of the item is probably very close to its true value. Moreover, successful replacement of infrequent items hardly impact the final result of f_{min}.

3.3 Finding Heavy Changes

Rationale: Both heavy changes and frequent items are related to the frequencies of items. One typical approach for finding heavy changes is to build two data structures for frequent items in two periods respectively. Our algorithm also uses this approach.

Data Structure: For each period, we only build a min-heap, and do not use the filter. This min-heap is used to record frequent items.

Insertion: For each period, the insertion process is exactly the same as for finding frequent items (see Section 3.2).

Report: For two adjacent periods, we traverse all items: for each item, we query its frequency in the two min-heaps, and get two frequencies. If the difference of the two frequencies is larger than a predefined threshold, the item is reported as a heavy change. Note that if an item only appears in one min-heap, the queried frequency of the other min-heap is 0.

Analysis: All algorithms for finding frequent items can be used for finding heavy changes. There are three metrics for finding frequent items: precision, recall, and the accuracy of the estimate of the reported items. Only if all three metrics are high, the error of heavy changes is small. Fortunately, InterestSketch is accurate in terms of all three metrics, and thus can achieve very high accuracy for finding heavy changes.

3.4 Finding Super-spreaders

Data Structure (Figure 3): In the data structure of InterestSketch for finding super-spreaders, both of the filter and the min-heap are used.

For a given source IP address, we need to count the number of connections, *i.e.*, the number of different destination IP addresses. If the source IP address sends more than one packet to a specific destination IP address, we increment the value of connection for only the first packet (but not for the other packets as these are duplicates in this case). The Bloom filter is used for de-duplication before inserting items into the min-heap. We remind the reader that details about Bloom filters are provided in Section 3.1.

Each node in the min-heap stores two fields: the source IP address (ID) as key, and the *connection* as value. The root node in the min-heap stores the source IP with the smallest connections.

Insertion: Given an incoming packet e, there are two steps. First we check the Bloom filter to judge whether e is a duplicate. Second, if the Bloom filter reports true, e is discarded. Otherwise, we try to insert e into the min-heap.

Note that when checking or querying the Bloom filter, the item ID is the source IP address plus the destination IP address. On the other hand, when checking or updating the min-heap, the item ID is the source IP address only. Specifically, given an incoming packet e (with ID of source+destination), we first check the Bloom filter by calculating the z hash functions and get z ***hashed bits***. If all the z hashed bits are 1, the packet is probably not the first one sent from the source to the specific destination, and it is discarded. If the z hashed bits are not all 1s, e is definitely the first packet from the source to the destination. In this case, we first set all the z bits to 1, and then try to insert e (with ID of source only) into the min-heap. There are two cases: 1) If the source IP address of the packet is in the min-heap, we increment the corresponding connection. 2) Otherwise, if the min-heap

is not full, we simply insert this source IP address into the min-heap. If the min-heap is full, we try to insert the source IP address into the root node by using our PRI algorithm.

Report: To report the source IP addresses above a given threshold $\mathcal{T}$, we traverse the min-heap to pick the source IP addresses with connection larger than $\mathcal{T}$.

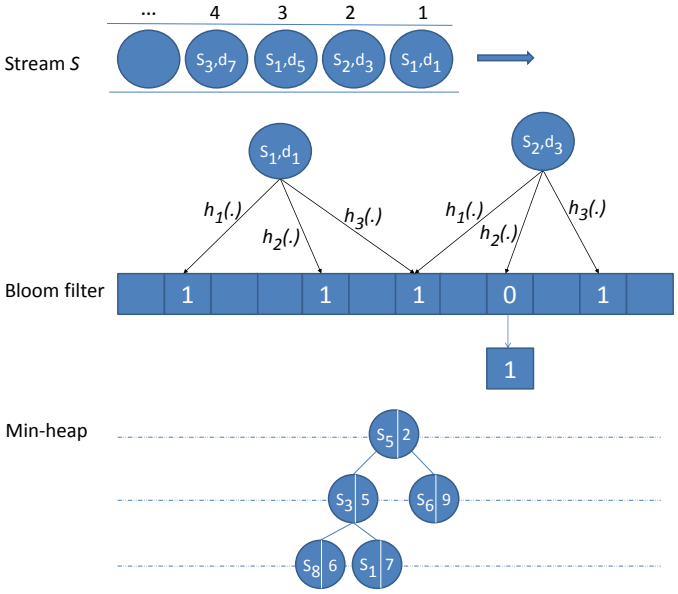


Figure 3: InterestSketch for finding super-spreaders.

Example: As shown in Figure 3, in the min-heap, the source IP address is stored in each node as the item ID. Given a network stream, each packet has a source IP address (s_i) and a destination IP address (d_i). For the first incoming packet with (s_1, d_1), we calculate three hash functions $h_1(s_1, d_1), h_2(s_1, d_1)$, *and* $h_3(s_1, d_1)$. If all of three *hashed bits* are 1, indicating that this packet is not the first one sent from s_1 to d_1, we discard it. When the second packet with (s_2, d_3) arrives, one of the three hashed bits is 0, indicating that this packet is the first one sent from s_1 to d_3 and s_1 is not in the min-heap. We change the second hashed bit from 0 to 1 in the Bloom filter and then insert s_2 into the min-heap, by applying our PRI algorithm (see Section 3.2). In this way, the source IP addresses with the largest connection (*i.e.*, number of destination IP addresses) are recorded in the min-heap. The detailed description of the algorithm is similar to Algorithm 1.

3.5 Finding Persistent Items

Data Structure (Figure 4): The data structure of InterestSketch in finding persistent items uses a Bloom filter and a min-heap. The Bloom filter is used to remove duplicates in each period. Each node of the min-heap stores an item ID and the corresponding persistency. The root node stores the item with the smallest persistency.

Insertion: Given an incoming item, we first check the Bloom filter by calculating the d hash functions. If the d hashed bits are all 1, indicating that the item has probably been recorded in this period, we just discard the item. If the d hashed bits are not all 1, we first set all the d bits to 1. Then, there are two cases: 1) If the item is in the min-heap, we increment the corresponding persistency by one. 2) Otherwise, the item is not in the min-heap, and if the min-heap is not full, we insert this item into the min-heap. If the min-heap is full, we try to insert the item into the root node using our PRI algorithm.

Periodically Emptying: At the end of each period, we empty the Bloom filter by setting all bits to 0. In other words, the Bloom filter is used to indicate whether an item appears in the current period only.

Report: To report all the items above a given threshold $\mathcal{T}$, we traverse the min-heap to pick out the items with persistencies larger than $\mathcal{T}$.

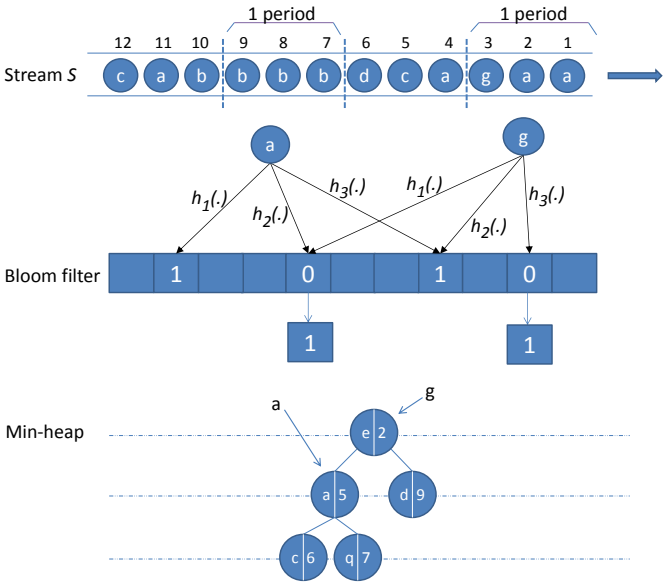


Figure 4: InterestSketch for finding persistent items.

Example: As shown in Figure 4, for a given data stream, we divide them into multiple equal-sized periods. For the first incoming item a, we get the three hashed bits $(1, 0, 1)$, which means that item a has not been recorded in the min-heap in this period. Therefore, we change the hashed bit from 0 to 1 in the Bloom filter and increment the persistency of a by one in the min-heap. Next we calculate three hash functions $h_1(a), h_2(a), h_3(a)$ for the second incoming item a. The three hashed bits are all 1, which indicates that a has been recorded in this period and thus the second item a is discarded. When the third item g arrives, the third hashed bit is 0, so we insert g into the min-heap and change the third hashed bit from 0

to 1. Since g is not in the min-heap, we try to insert it into the root node by applying our PRI algorithm. At the end of each period, we empty the Bloom filter by setting all bits to 0. In this way, when the fourth item a arrives in the next period, the persistency of a in the min-heap can be incremented by one regardless of the record in the previous period.

3.6 Shortcomings of the Basic Version

Our basic version has the following two shortcomings. First, we need to check whether every incoming item is in the min-heap, and the time complexity is $O(k)$, where k is the number of items in the min-heap. This problem can be solved by adding a hash table. However, the hash table will inevitably bring hash collisions and a waste of memory space. Second, when updating the min-heap, the time complexity is $O(logk)$. Since the speed of data streams is often high, we wish to reduce the time complexity from $O(logk)$ to $O(1)$.

4 OPTIMIZATIONS

To address the two shortcomings of our basic version, we propose three optimization methods. For convenience, and without loss of generality, *we assume the interest is frequency in this section.* When interest is defined as change of frequency, persistency or connection, the optimizations remain unchanged. In practice, we recommend using our final version of storing multiple items in one bucket (see Section 4.3). Related experiments are shown in Appendix C.

4.1 Using Stream-Summary

The basic version of our algorithm is slow with insertions. To address this, we accelerate our algorithm by replacing the min-heap with Stream-Summary[20] and a hash table. Stream-Summary is a data structure proposed by the authors of SpaceSaving [20]. It can achieve $O(1)$ time complexity for both updating the heap and returning the smallest item. Stream-Summary is a double-directional linked list consisting of nodes. Each node stores a unique frequency. For a given node with frequency f, it has a linked list storing all the items whose frequency is exactly f. In the hash table, the key is the flow ID, and the value is a pointer to the corresponding node in the min-heap. The hash table is used to check whether the incoming flow is in the Stream-Summary, and if so, the corresponding frequency will be incremented by 1.

However, this optimization has inevitable drawbacks. First, Stream-Summary consumes a relatively large amount of memory, since it is a double-directional linked list and each node has four or six pointers. Second, the hash table also requires a significant amount of memory space to minimize the hash collision rate. Third, the update process is relatively slow because multiple pointers need to be modified for each update.

4.2 Using Multiple Arrays

To solve the problem of memory space consumption in the previous optimization, we propose another optimization using multiple arrays.

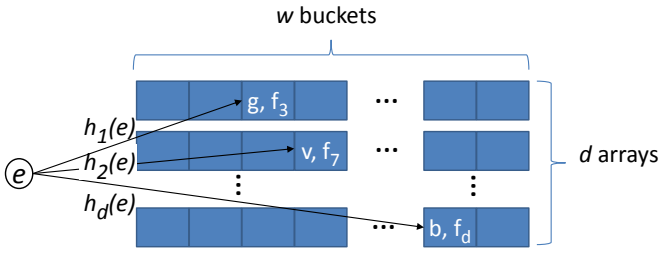


Figure 5: InterestSketch using multiple arrays.

Data Structure (Figure 5): There are d arrays in total and each array corresponds to a hash function. Each array consists of w buckets and each bucket stores the ID and frequency of an item. Every bucket has its own t_{fail}.
Insertion: For an incoming item e, we calculate d hash functions and pick the corresponding d buckets. There are three cases. First, if one of the d buckets stores the item with the same ID as the incoming item e, we increment the frequency in that bucket. Second, if there is no item with the same ID as e but there are empty buckets, we just store e in the first empty bucket. Third, if there is neither item with the same ID nor empty buckets, we select the bucket with the smallest frequency and replace the original item with the incoming one using our PRI algorithm.
Report: To return the items with frequency larger than a predefined threshold $\mathcal{T}$, we traverse the d arrays. If the frequency stored in a bucket is larger than $\mathcal{T}$, the corresponding item is reported.
Advantages and Disadvantages: Compared with the memory hungry Stream-Summary, using multiple arrays minimizes the memory usage thanks to the following two reasons. First, there is no pointer in this data structure. Second, since the number of items is far larger than that of buckets, there are few, if any, empty buckets. Given the same memory space, using multiple arrays outperforms Stream-Summary to a large extent in terms of accuracy. However, the time complexity of insertions is $O(d)$, where d is the number of arrays. Although d can be small (*e.g.*, 3 or 4), we still wish to reduce the time complexity to $O(1)$, without sacrificing accuracy or memory efficiency. Therefore, we propose the final optimization in the following subsection.

4.3 Final Version: Storing Multiple Items in one Bucket

We recommend this final version for finding interesting items, because it overcomes all shortcomings of the previous versions.

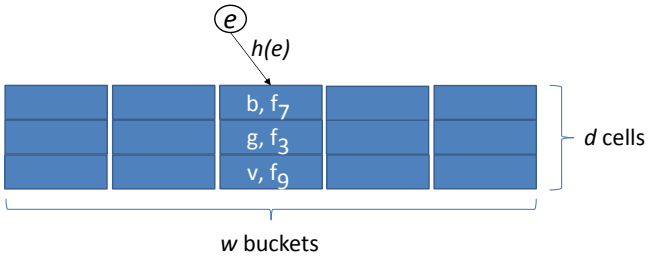


Figure 6: InterestSketch using multiple cells in every bucket.

Data Structure: As shown in Figure 6, there is one array which consists of w buckets. Each bucket has d cells storing the ID and frequency of an item. Every bucket has also its own t_{fail}.

Insertion: Given an incoming item e, we calculate the hash function and pick the corresponding bucket. There are three cases. First, if one of the items in the bucket has the same ID as the incoming item e, we increment the frequency of that item by 1. Second, if there is no item with the same ID as e but the bucket is not full, we store e in the first empty cell in that bucket. Third, if there is no item with the same ID nor empty cell, we select the item e_{min} with the smallest frequency f_{min} in the corresponding bucket and replace e_{min} with the incoming one e using our PRI algorithm: we replace e_{min} with the item e with a probability of $1/(2f_{min} - t_{fail} + 1)$. If replacing is successful, we increment the frequency from f_{min} to $f_{min} + t_{fail}/f_{min}$. Otherwise, the frequency remains unchanged.

Report: To return the items with frequency larger than a predefined threshold $\mathcal{T}$, we traverse the w buckets and d cells in each bucket, and report all appropriate items.

Analysis: The final version not only inherits the advantages of using multiple arrays, but overcomes its disadvantage in terms of time complexity as well. First, this data structure has high memory efficiency, since it neither applies pointers nor has many empty cells. Second, by applying our PRI algorithm, it reaches a much higher accuracy compared with SpaceSaving. Third, the time complexity for processing each packet is reduced from $O(d)$ to $O(1)$, because each insertion only needs to probe one bucket. The algorithm is shown in Algorithm 2

Algorithm 2: Insertion process for the final version.

Input: An item e_i

```
random() ∈ [0, 1]
b_i ← b[h(e_i)%w];
if e_i ∈ b_i then
    Interest(e_i) + +;
else
    if b_i has empty cells then
        b_i.insert(e_i);
    else
        if random() ⩽ 1/(2∗I_min − t_fail+1) then
            e_min ← e_i;
            I_min ← I_min + t_fail/I_min;
            t_fail ← 0;
        else
            t_fail + +;
```

5 PROOFS

In this section, we first derive the error bounds of the basic version of InterestSketch, and then derive some theoretical properties of our technique – PRI.

5.1 Error Bounds of InterestSketch

For the basic version of InterestSketch, we first prove that t_{fail}, the number of replacement failures defined in Section 3.1, is bounded.

Theorem 5.1. $0 \leqslant t_{fail} \leqslant 2 * \mathcal{I}_{min}$.

Proof. When there are still empty buckets in the min-heap, we do not use PRI. At that time, $t_{fail} = 0, \mathcal{I}_{min} = 0$.

When there is no empty bucket in the min-heap, it is obvious that the t_{fail} increases by 1 for each unsuccessful replacement. When t_{fail} reaches $2 * \mathcal{I}_{min}$, the probability $\mathcal{P} = \frac{1}{(2*\mathcal{I}_{min} - t_{fail}+1)}$ increases to 1, and then the coldest item e_{min} is replaced, and t_{fail} is set to 0. Therefore, $0 \leqslant t_{fail} \leqslant 2 * \mathcal{I}_{min}$.

□

Now we derive the deterministic upper bounds for the estimation error of InterestSketch.

Theorem 5.2. *Let $\mathcal{I}_x$ be the true interest of x, let $\hat{\mathcal{I}}_x$ be the estimated interest of $\mathcal{I}_x$, and let $\mathcal{I}_{min}$ be the minimum counter in the min-heap. We have*

$$\hat{\mathcal{I}}_x \leqslant \mathcal{I}_x + \mathcal{I}_{min} + 1 \tag{1}$$

Proof. Let $\mathcal{I}'_x$ be the interest of x after processing by the Bloom filter. Because of the false positives of the Bloom filter, some items which are supposed to be inserted to the min-heap will be discarded, which makes $\mathcal{I}'_x \leqslant \mathcal{I}_x$.

We assume that x was inserted to the min-heap at time t. Let $\mathcal{I}^t_{min}$ be the minimum counter in the min-heap before time t, t^t_{fail} the number of replacement failures before time t, and $\hat{\mathcal{I}_x}^t$ the estimated interest of x at time t. If there are still empty buckets in the min-heap before time t, $\hat{\mathcal{I}_x}^t = 1, \mathcal{I}^t_{min} = 0$. Otherwise, $\hat{\mathcal{I}_x}^t = \mathcal{I}^t_{min} + \frac{t^t_{fail}}{\mathcal{I}^t_{min}}$. According to Theorem 5.1, $t^t_{fail} \leqslant 2 * \mathcal{I}^t_{min}$. Therefore, at that point, $\hat{\mathcal{I}_x}^t \leqslant \mathcal{I}^t_{min} + 2$.

Since the value of the minimum counter monotonically increases over time, which means that $\mathcal{I}^t_{min} \leqslant \mathcal{I}_{min}$. Assume x arrived n times after t. n must be no larger than $\mathcal{I}'_x - 1$. Therefore,

$$\begin{aligned}\hat{\mathcal{I}_x} &= \hat{\mathcal{I}_x}^t + n \\ &\leqslant \mathcal{I}^t_{min} + 2 + \mathcal{I}'_x - 1 \\ &= \mathcal{I}^t_{min} + \mathcal{I}'_x + 1 \\ &\leqslant \mathcal{I}_x + \mathcal{I}_{min} + 1\end{aligned} \tag{2}$$

□

5.2 Theoretical Properties of PRI

Below, we provide some theoretical properties of PRI during replacements.

Theorem 5.3. *Assume that there is no empty bucket in the min-heap. At this time, we begin to use PRI. We also assume that the minimum counter does not increase during the replacement and $t_{fail} = 0$. Let $\mathcal{I}_{min}$ be the the minimum counter in the min-heap, let C_i be the collections of the next $2 * \mathcal{I}_{min} + 1$ repeatable items which are not in the min-heap and not discarded, and let P_i be the probability that e_{min} will be replaced by the i_{th} item in C_i. Then,*

$$P_i = \frac{1}{2 * \mathcal{I}_{min} + 1} \ (1 \leqslant i \leqslant 2 * \mathcal{I}_{min} + 1) \tag{3}$$

This holds irrespective of the stream permutation.

Proof. When e_{min} is replaced by the i_{th} item in C_i, $t_{fail} = i - 1$, which means that the first $i - 1$ items in C_i all fail to replace e_{min}. Therefore,

$$\begin{aligned}P_i &= \frac{1}{2 * \mathcal{I}_{min} - (i-1) + 1} * \prod_{j=0}^{i-2} \frac{2 * \mathcal{I}_{min} - j}{2 * \mathcal{I}_{min} - j + 1} \\ &= \frac{1}{2 * \mathcal{I}_{min} - i + 2} * \frac{2 * \mathcal{I}_{min} - i + 2}{2 * \mathcal{I}_{min} - i + 3} * \cdots * \frac{2 * \mathcal{I}_{min}}{2 * \mathcal{I}_{min} + 1} \\ &= \frac{1}{2 * \mathcal{I}_{min} + 1} \ (1 \leqslant i \leqslant 2 * \mathcal{I}_{min} + 1)\end{aligned} \tag{4}$$

□

Now, we show that the probability of replacing e_{min} with each distinct item in C_i is proportional to the frequency of each distinct item in C_i.

Theorem 5.4. *The conditions are the same as those of Theorem 5.3. If item e_i appears f_{e_i} times in C_i, e_i will replace e_{min} with probability $\frac{f_{e_i}}{2 * \mathcal{I}_{min} + 1}$.*

Proof. According to Theorem 5.3, for any e_i in C_i, e_i replaces e_{min} with probability $\frac{1}{2 * \mathcal{I}_{min} + 1}$. Therefore, e_i will replace e_{min} with probability $f_{e_i} * \frac{1}{2 * \mathcal{I}_{min} + 1} = \frac{f_{e_i}}{2 * \mathcal{I}_{min} + 1}$.

□

Finally, we show that the number of replacement failures before replacing e_{min} is equal to $\mathcal{I}_{min}$ in expectation.

Theorem 5.5. *We assume e_{min} is replaced at time t, and that the minimum counter does not increase during the replacement. Let $\mathcal{I}^t_{min}$ be the the minimum counter in the min-heap before time t and t^t_{fail} the number of replacement failures before time t. We have $E[t^t_{fail}] = \mathcal{I}^t_{min}$.*

Proof. Assume that the current incoming item is the i^{th} item. The number of replacement failures is equal to i before e_{min} is replaced, which is equivalent to the statement that e_{min} is replaced by the $i + 1_{th}$ item in C_i. According to Theorem 5.3, we have

$$\begin{aligned}P(t^t_{fail} = i) &= P_{i+1} \\ &= \frac{1}{2 * \mathcal{I}_{min} + 1} \ (0 \leqslant t^t_{fail} \leqslant 2 * \mathcal{I}^t_{min})\end{aligned} \tag{5}$$

Therefore,

$$\begin{aligned}E[t^t_{fail}] &= \sum_{i=0}^{2*\mathcal{I}^t_{min}} i * P(t^t_{fail} = i) \\ &= \frac{1}{2 * \mathcal{I}_{min} + 1} * \sum_{i=0}^{2*\mathcal{I}^t_{min}} i \\ &= \frac{1}{2 * \mathcal{I}_{min} + 1} * \frac{2 * \mathcal{I}^t_{min} * (2 * \mathcal{I}^t_{min} + 1)}{2} \\ &= \mathcal{I}^t_{min}\end{aligned} \tag{6}$$

□

6 EXPERIMENTAL RESULTS

In this section, we provide experimental results where we compare the final version of InterestSketch with the state-of-the-art algorithms for different definition of interests. Due to space limitation, the experimental figures on AAE and CR are provided in Appendix B. Experimental results of comparing different versions of InterestSketch are provided in the end of our technical report [1] without identifying information.

6.1 Experimental Setup

Datasets:

1) IP Trace Dataset: The IP Trace Dataset are streams of anonymized IP traces collected in 2016 by CAIDA [25]. Each item is identified by its source IP address (4 bytes).

2) Web Page Dataset: The Web page dataset is built from a collection of web pages, which were downloaded from the website [26]. Each item (4 bytes) represents the number of distinct terms in a web page.

3) Synthetic Dataset: We generate the synthetic dataset which follows the Zipf [27] distribution by using Web Polygraph [28], an open source performance testing tool. This dataset has 32 million items. The length of each item ID is 4 bytes. The skewness of this dataset is 1.5.

4) Network Dataset: The network dataset contains users' posting history on the stack exchange website [29]. Each item has three values u, v, t, that mean user u answered user v's question at time t. We use u as the ID and t as the time stamp of an item.

Implementation: We have implemented InterestSketch in C++. The hash functions are implemented using the 32-bit Bob Hash (obtained from the open source website [30]) with different initial seeds. The random function is implemented from the random library in C++. We produce the random number by using the random_device from the <random> header file of the C++ standard library. All of the abbreviations used in the evaluation and their full name are shown in Table 1.

Computation Platform: We conducted all experiments on a machine with a 2-core processor (4 threads, 6th Gen Intel Core i7-6600U @2.60 GHz) and 16 GB DRAM memory. The processor has three levels of cache: one 128KB L1 cache, one 512KB L2 cache, and one 4MB L3 cache.

Metrics:

1) Average Absolute Error (AAE): $\frac{1}{|\Psi|}\sum_{e_i\in\Psi}|\mathcal{I}_i - \widehat{\mathcal{I}}_i|$, where $\mathcal{I}_i$ is the real interest of item e_i, $\widehat{\mathcal{I}}_i$ is its estimated interest, and Ψ is the query set. Here, we query the dataset by querying every distinct item once in the sketch.

2) Average Relative Error (ARE): $\frac{1}{|\Psi|}\sum_{e_i\in\Psi}\frac{|\mathcal{I}_i - \widehat{\mathcal{I}}_i|}{\mathcal{I}_i}$, where $\mathcal{I}_i$ is the real interest of item e_i, $\widehat{\mathcal{I}}_i$ is its estimated interest, and Ψ is the query set. Here, we query the dataset by querying each correct instance once in the sketch.

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

4) Recall Rate (CR): Ratio of the number of correctly reported items to the number of correct items.

5) Speed: Million operations (insertions) per second (Mops). All the experiments about speed are repeated 10 times and the average speed is reported.

6.2 Evaluation on Finding Frequent Items

Parameter Setting: We compare 6 algorithms: InterestSketch, CM with heap[18], CM-CU with heap[19], SS with CF[16], SS[20], and Unbiased SS[15]. Let d be the number of cells in each bucket. For InterestSketch, we set $d = 8$.

Table 1: Abbreviations of algorithms in experiment

Abbreviation	Full name
CM	Count-Min Sketch[18]
FR	Flow[8]
SS	SpaceSaving[20]
CF	Cold Filter[16]
OLF	One-level Filtering[9]
TLF	Two-level Filtering[9]
IttSketch	The final version of InterestSketch in §4.3

For CM with heap, CM-CU with heap, and SS with CF, the parameters are set according to the recommendation of the authors. In this experiment, we compare AAE, ARE, PR, CR, and insertion speed among the 6 algorithms. The size of memory used ranges from 200KB to 400KB. We choose this range because: 1) InterestSketch has performed well enough in this range and 2) relying on little memory will expose the difference between the algorithms.

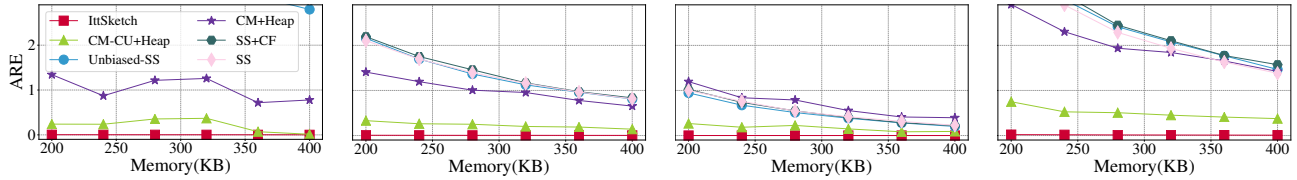
ARE (Figure 7(a)-7(d)): We find that, on three real-world datasets, the ARE of InterestSketch is around 3207 times, 708 times, 2735 times, 2707 times, and 2576 times lower than CM with heap, CM-CU with heap, SS with CF, SS, and Unbiased SS, respectively. On the synthetic dataset, the ARE of InterestSketch is around 416 times, 74 times, 1851 times, 1866 times, and 1820 times lower than CM with heap, CM-CU with heap, SS with CF, SS, and Unbiased SS, respectively.

PR (Figure 8(a)-8(d)): We find that on three real-world datasets, the PR of InterestSketch is around 11.2 times, 9.9 times, 2.8 times, 3.4 times, and 2.7 times higher than CM with heap, CM-CU with heap, SS with CF, SS, and Unbiased SS respectively. On the synthetic dataset, the PR of InterestSketch is around 30.3 times, 33.8 times, 5.9 times, 5.5 times, and 5.8 times higher than CM with heap, CM-CU with heap, SS with CF, SS, and Unbiased SS, respectively.

Speed (Figure 9(a)-9(d)): We find that, on three real-world datasets and one synthetic dataset, the insertion speed of InterestSketch is around 2.2 times, 2.7 times, 7.7 times, 5.48 times, and 6.7 times faster than CM with heap, CM-CU with heap, SS with CF, SS, and Unbiased SS, respectively.

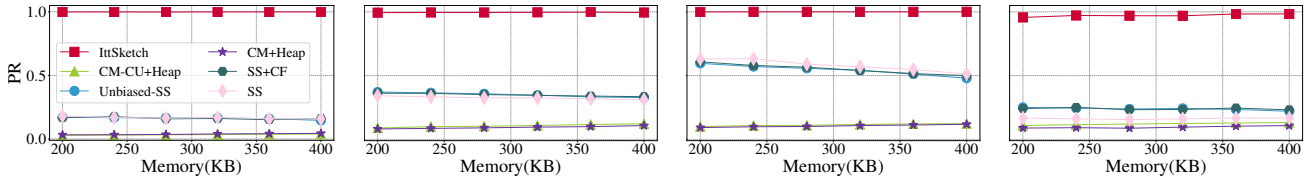
AAE (Figure 18(a)-18(d)) in Appendix B: We find that, on three real-world datasets, the AAE of InterestSketch is around 7309 times, 995 times, 3860 times, 3701 times, and 3549 times lower than CM with heap, CM-CU with heap, SS with CF, SS, and Unbiased SS, respectively. Besides, on the synthetic dataset, the AAE of InterestSketch is around 2278 times, 80 times, 3247 times, 3278 times, and 3205 times lower than CM with heap, CM-CU with heap, SS with CF, SS, and Unbiased SS, respectively.

CR (Figure 19(a)-19(d)) in Appendix B: We find that, on three real-world datasets, the CR of InterestSketch is around



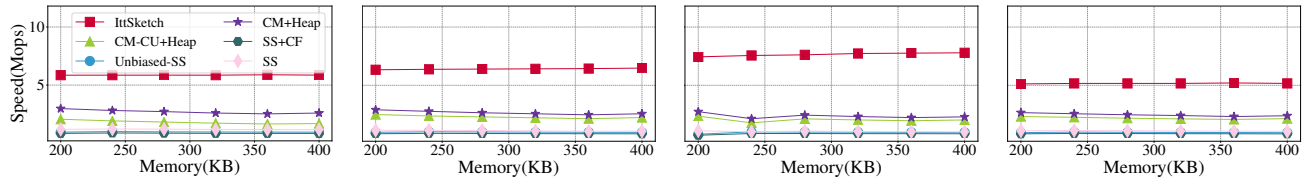
(a) Synthetic dataset (b) IP trace (c) Web page (d) Network dataset

Figure 7: ARE of finding Frequent Items.



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

Figure 8: PR of finding Frequent Items.



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

Figure 9: Speed of finding Frequent Items.

5.6 times, 5 times, 1.7 times, 2 times, and 1.6 times higher than CM with heap, CM-CU with heap, SS with CF, SS, and Unbiased SS, respectively. On the synthetic dataset, the CR of InterestSketch is around 17 times, 18 times, 5 times, 4.7 times, and 5.3 times higher than CM with heap, CM-CU with heap, SS with CF, SS, and Unbiased SS, respectively.

Summary: 1) InterestSketch can achieve high accuracy with limited memory. The ARE of InterestSketch is lower than 0.01 when the memory is set to 200KB, while the ARE of the other algorithms is often higher than 1. As seen in the figures, the ARE of SS, SS with CF, and Unbiased SS often exceed the range of the plots.

2) InterestSketch can report more correct instances than other approaches. InterestSketch often reports more than 99 percent of the correct instances, while the other approaches report less than 40 percent, because they often consume too much memory on the hash table or Stream-Summary.

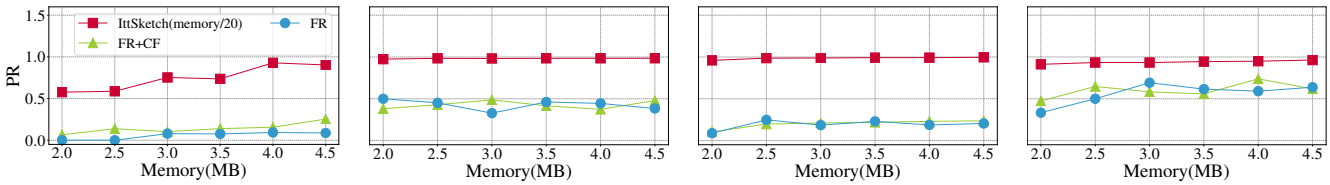
3) InterestSketch achieves higher precision in reported instances. The PR of InterestSketch is often higher than 0.99, while the PR of other approaches is often less than 0.6 because they overestimate the results.

4) The insertion speed of InterestSketch is also faster than the other approaches for the same memory consumption on three real-world datasets and one synthetic dataset.

6.3 Evaluation on Finding Heavy Changes

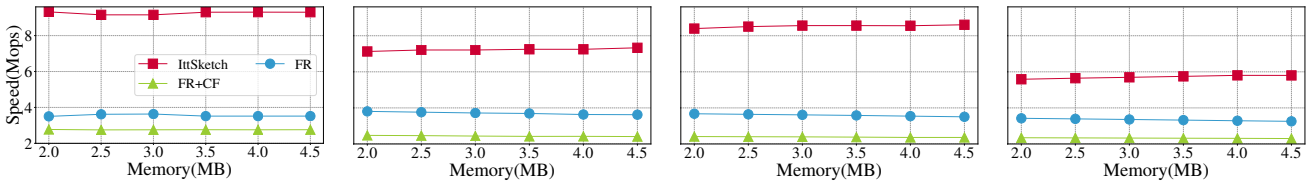
Parameter Setting: We compare 3 algorithms: InterestSketch, FR[8], and FR with CF[16]. Let d be the number of cells in each bucket. For InterestSketch, we set $d = 8$. For FR and FR with CF, the parameters are set according to the recommendation of the authors. In the experiments, we compare PR, CR, and the insertion speed among the 3 algorithms. We vary the amount of memory from from 2MB to 4.5MB. We choose this range because FR cannot report any heavy changes if memory size is smaller. Also, we set the memory size of InterestSketch to 1/20 of the memory size of the other algorithms when we compare PR and CR. The reason for this is that when memory is larger than 2MB, the CR and PR of InterestSketch are 1.

PR (Figure 10(a)-10(d)): We find that on three real-world datasets, the PR of InterestSketch is around 4.1 times and 3.6 times higher than FR and FR with CF. On the synthetic



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

Figure 10: PR of finding Heavy Changes.



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

Figure 11: Speed of finding Heavy Changes.

dataset, the PR of FR is 0 when its memory size is less than 2.5 MB. The PR of InterestSketch is around 6.2 times higher than FR with CF.

Speed (Figure 11(a)-11(d)): We find that the insertion speed of InterestSketch is around 2.1 times and 3 times faster than FR and FR with CF on three real-world datasets and one synthetic dataset.

CR (Figure 20(a)-20(d)) in Appendix B: We find that, on three real-world datasets, the CR of InterestSketch is around 64 times and 35 times higher than FR and FR with CF. On the synthetic dataset, the CR of FR is 0 when its memory size is less than 2.5 MB. The CR of InterestSketch is around 16 times higher than FR with CF.

Summary: 1) Although the memory size of InterestSketch is only 1/20 of other algorithms, the CR of InterestSketch is often more than 0.98 when its memory size is more than 0.98 on three real-world datasets and one synthetic dataset. In contrast, the CR of FR is often lower than 0.2 because it will decode few items if the memory size is too small.

2) The PR of InterestSketch is lower than the PR of InterestSketch in other datasets, for there are less differences between the two periods in the synthetic dataset. Therefore, we have to spend more space on finding heavy changes in the synthetic dataset. However, InterestSketch still performs much better than FR and FR with CF.

6.4 Evaluation on Finding Super-Spreaders

Parameter Setting: We compare 4 algorithms: InterestSketch, OLF[9], TLF[9], and OpenSketch[14]. Let d be the number of cells in each buckets. For InterestSketch, we set $d = 8$. Let z be the number of hash functions for the Bloom filter. For InterestSketch, we set $z = 4$. For OLF, TLF, and OpenSketch, the parameters are set according to the recommendation of the authors. In the experiment, we compare AAE, ARE, PR, CR, and insertion speed among the 4 algorithms. The memory size ranges from 500KB to 750KB. Because other algorithms often spend much memory on the hash table or bitmap, more memory is required for the experiment. Also, we split the IP Trace Dataset into 4 sub-datasets.

ARE (Figure 12(a)-12(d)): We find that the ARE of InterestSketch is around 31 times, 30 times, and 18 times lower than OLF, TLF, and OpenSketch, respectively.

PR (Figure 13(a)-13(d)): We find that InterestSketch achieves a PR above 0.99 when the memory is more than 500KB. The PR of InterestSketch is around 4.1 times, 3.4 times, and 1.9 times higher than OLF, TLF, and OpenSketch, respectively.

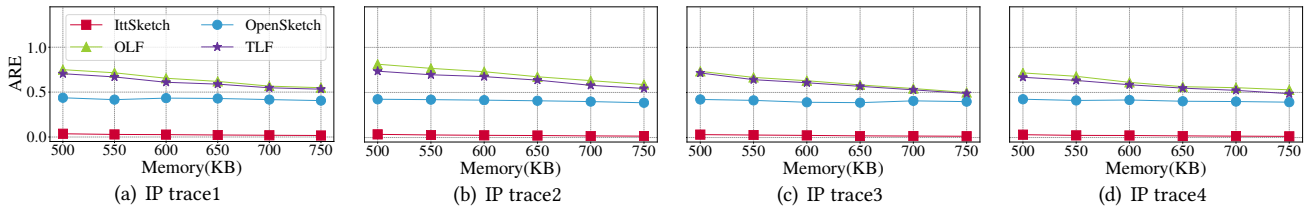
Speed (Figure 14(a)-14(d)): We find that, on IP Trace datasets, the insertion speed of InterestSketch is slower than OLF and TLF but is faster than OpenSketch.

AAE (Figure 21(a)-21(d)) in Appendix B: We find that the AAE of InterestSketch is around 14 times, 13.3 times, and 40 times lower than OLF, TLF, and OpenSketch, respectively.

CR (Figure 22(a)-22(d)) in Appendix B: We find that the CR of InterestSketch is around 1.4 times, 1.6 times, and 1.8 times higher than OLF, TLF, and OpenSketch, respectively.

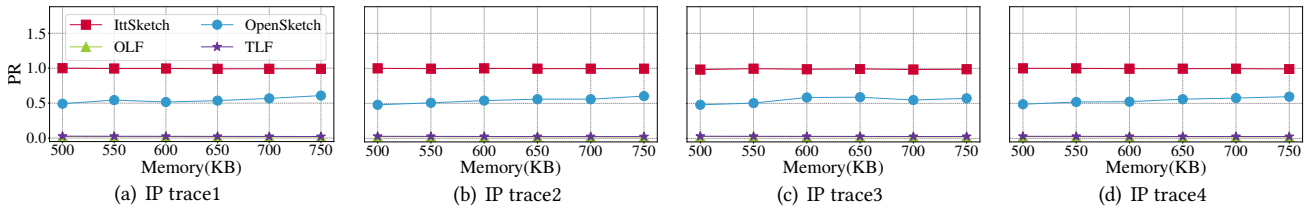
Summary: 1) InterestSketch is more accurate than other algorithms because other algorithms often spend much memory on the hash table or bitmap, which makes them unable to count items precisely.

2) Our results show that InterestSketch achieves both high recall rate and high precision rate. Though OLF and TLF



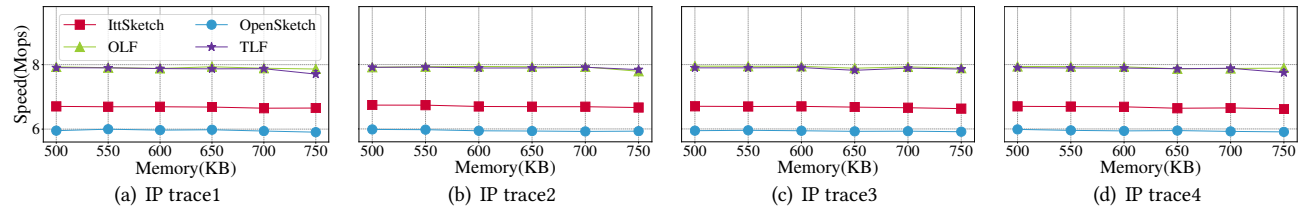
(a) IP trace1 (b) IP trace2 (c) IP trace3 (d) IP trace4

Figure 12: ARE of finding Super-Spreaders.



(a) IP trace1 (b) IP trace2 (c) IP trace3 (d) IP trace4

Figure 13: PR of finding Super-Spreaders.



(a) IP trace1 (b) IP trace2 (c) IP trace3 (d) IP trace4

Figure 14: Speed of finding Super-Spreaders.

report more correct instances than OpenSketch, they also report more wrong instances due to their low sample rate, making their precision rate lower than 0.03.

6.5 Evaluation on Finding Persistent Items

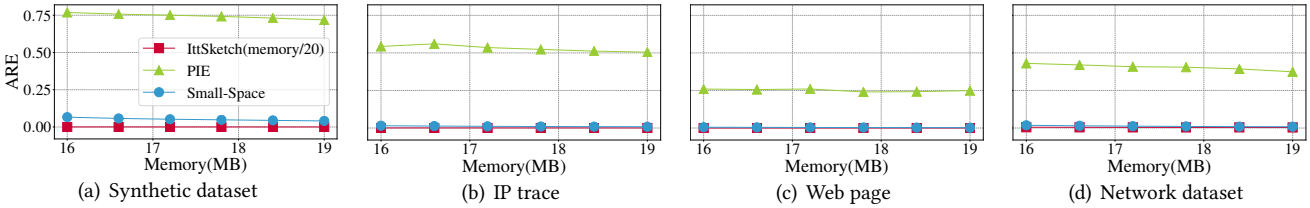
Parameter Setting: We compare 3 algorithms: InterestSketch, PIE[11], and Small-Space[23]. Let d be the number of cells in each bucket. For InterestSketch, we set $d = 8$. Let z be the number of hash functions for the Bloom filter. For InterestSketch, we set $z = 3$. For PIE and Small-Space, the parameters are set according to the recommendation of the authors. In the experiment, we compare AAE, ARE, PR, CR, and insertion speed among the 3 algorithms. The memory size ranges from 16MB to 19MB. We choose this range because PIE cannot report any interesting items on the synthetic dataset if memory size is less than 16MB. Also, we set the memory size of InterestSketch to 1/20 of the memory size of the other algorithms when we compare AAE, ARE, PR, and CR. We do this because when memory size is more than 16MB, the AAE and ARE of InterestSketch are 0, and the CR and PR of InterestSketch are 1. To compare these algorithms more conveniently, we reduce the memory size of InterestSketch.

ARE (Figure 15(a)-15(d)): We find that, on three real-world datasets, the ARE of InterestSketch is around 771 times and 50212 times lower than Small-Space and PIE. On the synthetic dataset, the ARE of InterestSketch is around 115 times and 1655 times lower than Small-Space and PIE.

PR (Figure 16(a)-16(d)): We find that, on three real-world datasets, the PR of InterestSketch is around 1.01 times and 1.23 times higher than Small-Space and PIE. On the synthetic dataset, the PR of InterestSketch is around 1.08 times and 6 times higher than Small-Space and PIE.

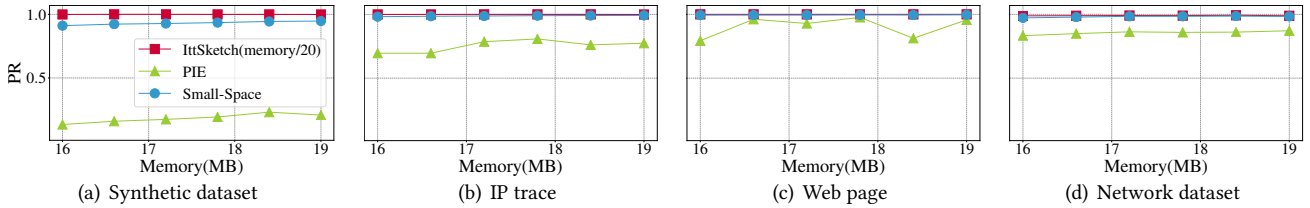
Speed (Figure 17(a)-17(d)): We find that the insertion speed of InterestSketch is around 1.4 times and 4 times faster than Small-Space and PIE on three real-world datasets and one synthetic dataset.

AAE (Figure 23(a)-23(d)) in Appendix B: We find that, on three real-world datasets, the AAE of InterestSketch is around 698 times and 53543 times lower than Small-Space and PIE. On the synthetic dataset, the AAE of InterestSketch



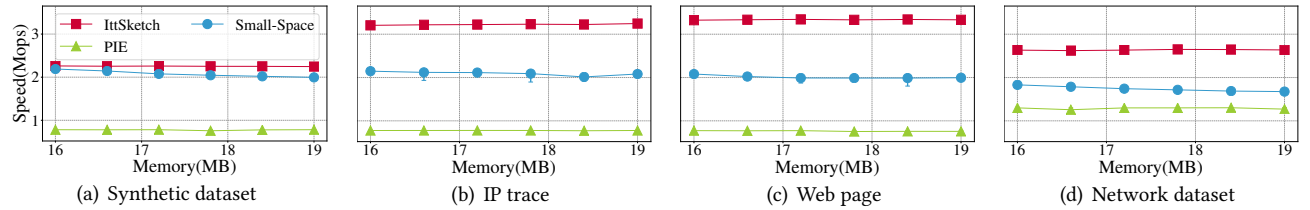
(a) Synthetic dataset (b) IP trace (c) Web page (d) Network dataset

Figure 15: ARE of finding Persistent Items.



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

Figure 16: PR of finding Persistent Items.



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

Figure 17: Speed of finding Persistent Items.

is around 167 times and 6095 times lower than Small-Space and PIE.

CR (Figure 24(a)-24(d)) in Appendix B: We find that, on three real-world datasets, the CR of InterestSketch is around 1.03 times and 33 times higher than Small-Space and PIE. On the synthetic dataset, the CR of InterestSketch is around 1.05 times and 33 times higher than Small-Space and PIE.

Summary: 1) Although the memory size of InterestSketch is only 1/20 of the other algorithms, it still performs much better. The ARE of InterestSketch is lower than 0.005 when its memory size is more than 800KB on three real-world datasets and one synthetic dataset.

2) The PR and CR of InterestSketch are often more than 0.99 when its memory size is more than 800KB on three real-world datasets and one synthetic dataset. The PR and CR of Small-Space are often more than 0.95 when its memory size is more than 16MB on three real-world datasets and one synthetic dataset. However, the CR of PIE is often lower than 0.2 because it wastes much of the space to record the items in every period.

3) InterestSketch can achieve higher insertion speed compared to Small-Space and PIE. Also, InterestSketch will not be slower when its memory use increases. In contrast, Small-Space slows down as memory use increases.

6.6 Limitations of Our Final Version

The above experimental results show that the final version of InterestSketch can achieve high accuracy with small memory. However, the final version cannot guarantee that it will report all the interest items correctly even with larger amount of memory. We can only claim that as the memory increases, the final version has a higher probability to be correct. In contrast, the basic version of InterestSketch can record all the flows if it has enough memory. Though it may use more memory and be slower, its correctness can be guaranteed when the memory size is large enough. When compared with the Unbiased SpaceSaving [15], another shortcoming of InterestSketch is that the InterestSketch is biased. However, InterestSketch achieves 1820~2576 smaller error rate than the Unbiased SpaceSaving (see Figure 7).

7 CONCLUSION

This paper addresses the issue of finding interesting items in data streams. Interesting items can be frequent items, heavy changes, super-spreaders, or persistent items. While most existing algorithms focus on one specific definition of interest and uses different data structures, we propose a generic framework which can find interesting items for different definitions of interest. Our main technique is called PRI, which is able to differentiate interesting items from others with high probability, in limited memory space. The idea of PRI is *to replace the current smallest item with a probability and then increment*. We theoretically prove that when replacement is successful, with high probability, the new item has a higher interest than the current smallest interest. We use our framework to find frequent items, heavy changes, super-spreaders, and persistent items. We conduct extensive experiments on three real datasets and one synthetic dataset. Our experimental results show that compared with the state-of-the-art, for each definition of interest, our algorithm improves the insertion speed $2.2 \sim 7.7$ times and the accuracy $74 \sim 3207$ times.

REFERENCES

[1] Source code related to interest sketch. https://github.com/IttSketch/IttSketch.

[2] Lukasz Golab, David DeHaan, Erik D Demaine, Alejandro Lopez-Ortiz, and J Ian Munro. Identifying frequent items in sliding windows over on-line packet streams. In *Proc. ACM IMC*, pages 173–178. ACM, 2003.

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

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

[5] Moses Charikar, Kevin Chen, and Martin Farach-Colton. Finding frequent items in data streams. In *Automata, Languages and Programming*. Springer, 2002.

[6] Balachander Krishnamurthy, Subhabrata Sen, Yin Zhang, and Yan Chen. Sketch-based change detection: methods, evaluation, and applications. In *Proceedings of the 3rd ACM SIGCOMM conference on Internet measurement*, pages 234–247. ACM, 2003.

[7] Robert Schweller, Zhichun Li, Yan Chen, et al. Reversible sketches: enabling monitoring and analysis over high-speed data streams. *IEEE/ACM Transactions on Networking (ToN)*, 15(5):1059–1072, 2007.

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

[9] Shobha Venkataraman, Dawn Xiaodong Song, Phillip B. Gibbons, and Avrim Blum. New streaming algorithms for fast detection of superspreaders. In *NDSS*, 2005.

[10] Zhewei Wei, Ge Luo, Ke Yi, Xiaoyong Du, and Ji-Rong Wen. Persistent data sketching. In *Proc. ACM SIGMOD*, pages 795–810. ACM, 2015.

[11] Haipeng Dai, Muhammad Shahzad, Alex X Liu, and Yuankun Zhong. Finding persistent items in data streams. *Proceedings of the VLDB Endowment*, 10(4):289–300, 2016.

[12] Jake D Brutlag. Aberrant behavior detection in time series for network monitoring. In *Usenix Conference on System Administration*, pages 139–146, 2000.

[13] Pratanu Roy, Arijit Khan, and Gustavo Alonso. Augmented sketch: Faster and more accurate stream processing. In *Proc. ACM SIGMOD*, pages 1449–1463, 2016.

[14] Minlan Yu, Lavanya Jose, and Rui Miao. Software defined traffic measurement with opensketch. In *NSDI 2013*, 2013.

[15] Daniel Ting. Data sketches for disaggregated subset sum and frequent item estimation. In *SIGMOD Conference*, 2018.

[16] Yang Zhou, Tong Yang, Jie Jiang, Bin Cui, Minlan Yu, Xiaoming Li, and Steve Uhlig. Cold filter: A meta-framework for faster and more accurate stream processing. In *SIGMOD Conference*, 2018.

[17] Peixiang Zhao, Charu C Aggarwal, and Min Wang. gsketch: on query estimation in graph streams. *Proc. VLDB*, 2011.

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

[19] Cristian Estan and George Varghese. New directions in traffic measurement and accounting. *ACM SIGMCOMM CCR*, 32(4), 2002.

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

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

[22] Q. Zhao, J. Xu, and A. Kumar. Detection of super sources and destinations in high-speed networks: Algorithms, analysis and evaluation. *IEEE JSAC*, 24(10):1840–1852, 2006.

[23] Bibudh Lahiri, Jaideep Chandrashekar, and Srikanta Tirthapura. Space-efficient tracking of persistent items in a massive data stream. *Statistical Analysis and Data Mining*, 7:70–92, 2011.

[24] Burton H Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, 1970.

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

[26] Real-life transactional dataset. http://fimi.ua.ac.be/data/.

[27] David MW Powers. Applications and explanations of Zipf's law. In *Proc. EMNLP-CoNLL*. Association for Computational Linguistics, 1998.

[28] Alex Rousskov and Duane Wessels. High-performance benchmarking with web polygraph. *Software: Practice and Experience*, 34(2):187–211, 2004.

[29] The Network dataset Internet Traces. http://snap.stanford.edu/data/.

[30] Hash website. http://burtleburtle.net/bob/hash/evahash.html.

[31] Ben Basat Ran, Gil Einziger, Roy Friedman, and Yaron Kassner. Heavy hitters in streams and sliding windows. In *Proc. IEEE INFOCOM 2016*, 2016.

[32] Michael T Goodrich and Michael Mitzenmacher. Invertible bloom lookup tables. In *Proceedings of the 49th Annual Allerton Conference on Communication, Control, and Computing*, pages 792–799. IEEE, 2011.

[33] A.Shokrollahi. Raptor codes. *IEEE Transactions Information Theory*, 52(6):2551–2567, 2006.

[34] Graham Cormode, Minos Garofalakis, Peter J Haas, Chris Jermaine, et al. Synopses for massive data: Samples, histograms, wavelets, sketches. *Foundations and Trends® in Databases*, 4(1–3):1–294, 2011.

[35] Nan Tang, Qing Chen, and Prasenjit Mitra. Graph stream summarization: From big bang to big crunch. In *Proc. ACM SIGMOD*, pages 1481–1496, 2016.

[36] Zengfeng Huang, Xuemin Lin, Wenjie Zhang, and Ying Zhang. Efficient matrix sketching over distributed data. In *ACM Sigmod-Sigact-Sigai Symposium on Principles of Database Systems*, pages 347–359, 2017.

[37] Haida Zhang, Zengfeng Huang, Zhewei Wei, Wenjie Zhang, and Xuemin Lin. Tracking matrix approximation over distributed sliding windows. In *Proc. IEEE ICDE 2017*, 2017.

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

[39] Anshumali Shrivastava, Arnd Christian Konig, and Mikhail Bilenko. Time adaptive sketches (ada-sketches) for summarizing data streams. In *Proc. ACM SIGMOD*, pages 1417–1432, 2016.

[40] Huanchen Zhang, Hyeontaek Lim, Viktor Leis, David G Andersen, Michael Kaminsky, Kimberly Keeton, and Andrew Pavlo. Surf: practical range query filtering with fast succinct tries. In *Proc. ACM SIGMOD*, pages 323–336, 2018.

[41] Prashant Pandey, Michael A Bender, Rob Johnson, and Rob Patro. A general-purpose counting filter: Making every bit count. In *Proc. ACM SIGMOD*, pages 775–787, 2017.

[42] Yanqing Peng, Jinwei Guo, Feifei Li, Weining Qian, and Aoying Zhou. Persistent bloom filter: Membership testing for the entire history. In *Proc. ACM SIGMOD*, pages 1037–1052. ACM, 2018.

[43] Jiawei Jiang, Fangcheng Fu, Tong Yang, and Bin Cui. Sketchml: Accelerating distributed machine learning with data sketches. In *Proc. ACM SIGMOD*, pages 1269–1284. ACM, 2018.

[44] Kai Sheng Tai, Vatsal Sharan, Peter Bailis, and Gregory Valiant. Sketching linear classifiers over data streams. In *Proc. ACM SIGMOD*, pages 757–772, 2018.

A RELATED WORK

A.1 Prior Art for Finding Frequent Items

To find frequent items, as mentioned above, existing solutions can be divided into two kinds. The first kind is called `record all`, which records the information of all items. This kind uses a sketch plus a min-heap or an array. Typically, we can use sketches of CM [18], CU [19], and Count[5]. Since the three sketches are similar to a large extent, we only introduce the CM sketch, as it is the most widely used. In a CM sketch, there are d arrays corresponding to d hash functions. For an incoming item, it first calculates d hash functions and gets d hashed counters, each of which is incremented by one. When reporting the frequency of a specific item, the d hash functions of the ID are first calculated and the smallest hashed counter is returned. When inserting an item, if its estimated frequency is larger than the smallest frequency in the min-heap, the item is inserted into the min-heap. A sophisticate sketch, ASketch [13], uses a small filter acting as a cache to capture hot items, and records other items in a CM sketch, achieving higher speed and accuracy than only using a CM sketch. The authors recommend storing only 32 hot items in the filter, and thus ASketch focuses on frequency estimation rather than frequent items.

Realizing that it is unnecessary and harmful to record frequencies of cold items, the idea of the second kind of solution, `record part`, records the information of only hot items. For this kind of solution, the most well-known algorithm is SpaceSaving [20]. It has various variants: the Unbiased SpaceSaving [15], Cold filter, and more [31]. The key operation of SpaceSaving is as follows. It uses a min-heap-like data structure – Stream-Summary. When a new item e arrives, it increments the smallest frequency f_{min} in Stream-Summary by 1, and then replaces the smallest item with e. SpaceSaving has only over-estimation error. Based on SpaceSaving, the Unbiased SpaceSaving [15] makes a small modification: when a new item e arrives, it also increments the smallest frequency f_{min} by 1, but then replaces the smallest item with e with a probability. The Unbiased SpaceSaving is proved theoretically to be unbiased. For SpaceSaving and the Unbiased SpaceSaving, every cold item will increment the smallest frequency, and since most items are cold in practice, leads to poor accuracy. Noticing the negative effect of cold items, Cold filter [16] filters cold items using a small CU sketch, achieving a much higher accuracy. In contrast, we aim to minimize the negative effect above with no auxiliary data structures.

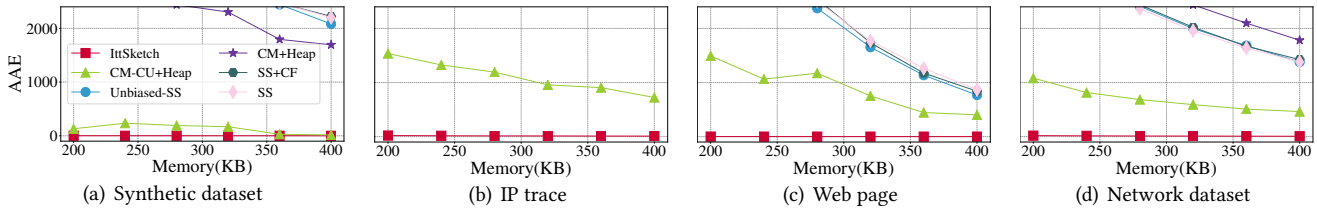
A.2 Prior Art for Finding Heavy Changes

To find frequent items, there are also two kinds of algorithms. The first kind is `record all`, recording the information of all items. Typical algorithms include k-ary [6], the reversible sketch [7], and FlowRadar [8]. The k-ary sketch [6] can only record the difference of frequencies, but cannot record item IDs. To overcome this shortcoming, the reversible sketch [7] improves the k-ary sketch by estimating the item IDs with time complexity of $O(2^{0.75L})$ (L is the number of bits of item ID), which can be very large. FlowRadar [8] uses a Bloom filter to remove duplicates, and records each item in a modified Invertible Bloom Lookup Table (IBLT) [32]. The time complexity to decode item ID is $O(n)$ (n is the number of distinct items). When IBLT uses the appropriate parameters, FlowRadar can successfully decode all the items with high probability. As FlowRadar records and decodes all items, it is not memory efficient. The second kind of solutions manages to record only the hot items, and the typical algorithm is Cold filter [16]. Based on FlowRadar, Cold filter [16] uses an additional filter to filter cold items, and only inserts hot items into FlowRadar, thus achieving a higher accuracy. However, as FlowRadar is inherently not a compact data structure, it is still not memory efficient despite using Cold filter.

A.3 Prior Art for Finding Super-Spreaders

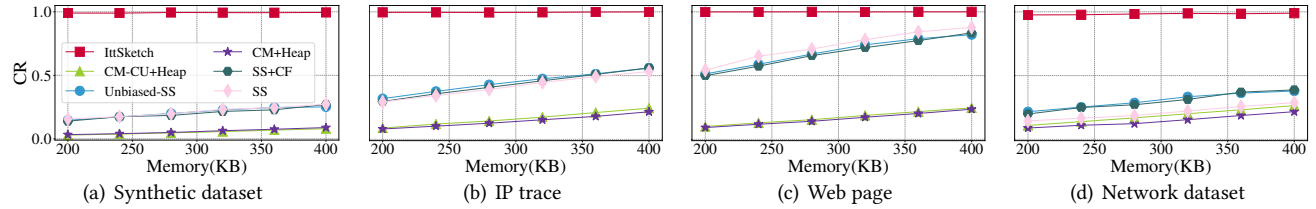
Again, there are two kinds of solutions for finding Super-Spreaders. The first is `record all`, which records the information of all packets. Typical algorithms are Two-dimension bitmap [22] and OpenSketch [14]. Two-dimension bitmap [22] uses a matrix of bits. To insert a packet, the source IP address is hashed to a row of the matrix while the destination IP address is hashed to a column. Then a bit is located and set to 1. For a source IP address, the number of 1s in the mapped row can be used to estimate the number of connections. By adding a min-heap, Super-Spreaders can be found. As the number of Super-Spreaders are often small, most rows are empty, which is a waste of memory. To find Super-Spreaders, OpenSketch [14] combines the techniques of CM sketches (presented above) and bitmaps. It replaces each counter in the CM sketch by a bitmap. The number of 1s in each bitmap is used to estimate the number of connections. The bitmap needs to be large enough to accurately estimate the connections. Therefore, OpenSketch is accurate when using large amounts of memory.

The second kind is again `record samples`. As there are many items, even simple sampling can automatically filter/discard many cold items, saving memory. The typical algorithms are called one-level filtering [9] and two-level filtering [9]. Besides sampling, they remove duplicates by using a hash function. Specifically, given an incoming packet e, they hash the combination of source and destination IP into the range $[0, 1]$. If the hash value is larger than the given threshold, e is chosen; otherwise, e is discarded. Then, it uses a hash table to record all samples before it reports Super-Spreaders. Indeed, sampling can save memory, but the incurred error is hard to reduce. Our method on the other



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

Figure 18: AAE of finding Frequent Items.



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

Figure 19: CR of finding Frequent Items.

hand can achieve a higher accuracy, and can be used together with the sampling method.

A.4 Prior Art for Finding Persistent Items

Again, two types of solutions exist to find persistent items. The first, `record all`, records all packets. The typical algorithm is PIE [11]. For each time period, PIE uses a hash table to record the incoming items fingerprints. When a collision happens, the fingerprint is set to invalid. The key technique is to use Raptor codes [33] to generate different fingerprints in different periods. For a persistent item occurring in many periods, many valid fingerprints can be found, and these can be used to recover the item ID. Therefore, persistent items have a higher probability to be successfully recovered. However, due to the recording of cold items and collisions, the accuracy of PIE is not high.

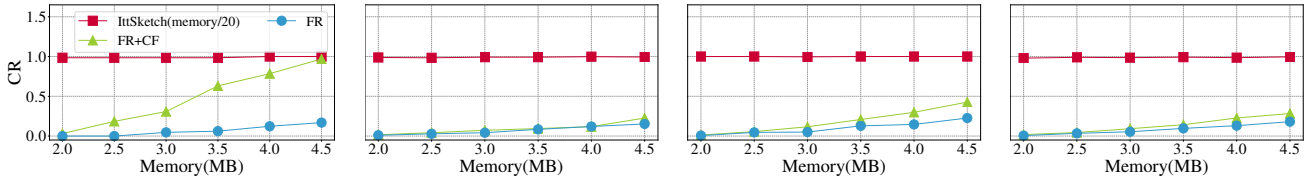
The second kind, `record samples`, samples and removes duplicates before recording packets. The typical algorithm is called small-space [23]. As mentioned earlier, sampling can save memory. The algorithm of small-space is similar to one-level filtering [9]. The only difference is that it hashes the item ID plus the index of period for removing duplicates. Our method can achieve a higher accuracy, and can be used together with the sampling method.

A.5 Other Sketches

Except for the above four tasks, sketch techniques also attract the attention of researchers in other applications. Here we give a brief overview of recent works. Interested readers please refer to the excellent book [34].

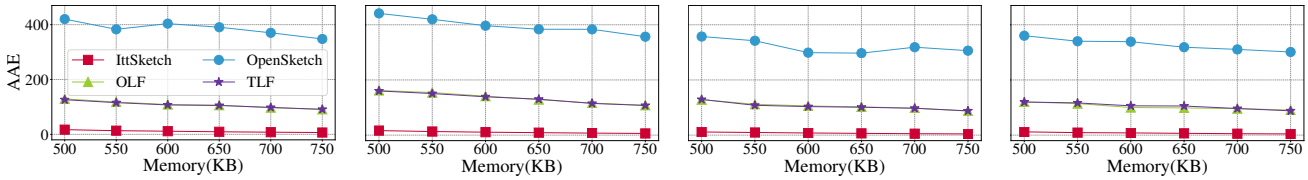
TCM [35] is a sketch for graph streams, which supports a wide range of graph queries. SVS [36] is an algorithm to output a covariance sketch over massive distributed data metrics. DA2 [37] is another algorithm to track covariance sketch over distributed sliding windows.

For the task of frequency estimation, bias-aware sketches [38] outperforms other sketches when there is a bias in the frequency vector of the data stream. Ada-sketches [39] supports queries by time ranges as well as time points, and provide better accuracy for queries on recent time than that on old time. For membership query, SuRF [40] supports approximate range membership query, besides the point query which is supported by most filters. Counting Quotient Filter [41] is designed to support deletion and counting, and has good data locality on SSDs. Persistent Bloom filters [42] can be used for temporal membership queries. Sketches can also be used in machine learning. SketchML [43] tries to use sketches to compress the gradients exchanged in the network, so as to reduce the training time. The authors in [44] use a sketch to express the model of the linear classifier, to support model training on memory-limited devices.



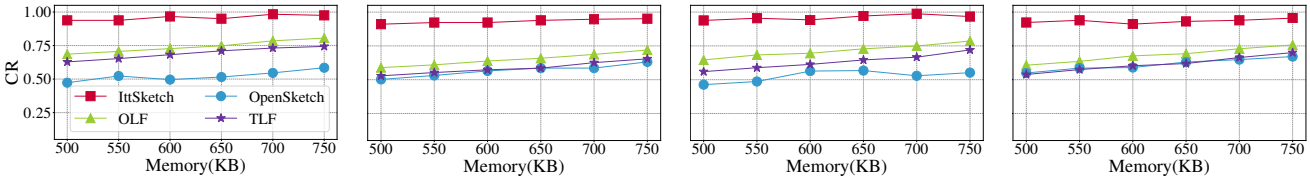
(a) Synthetic dataset (b) IP trace (c) Web page (d) Network dataset

Figure 20: CR of finding Heavy Changes.



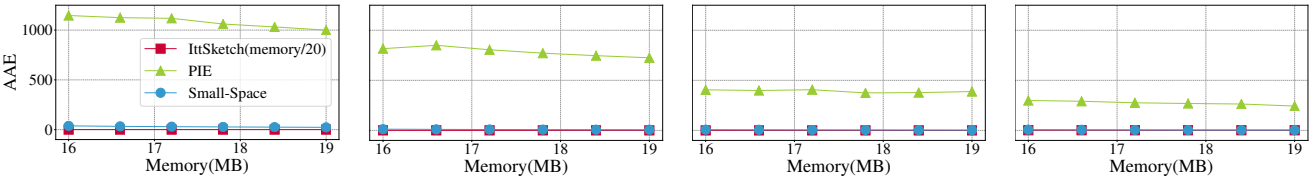
(a) IP trace1 (b) IP trace2 (c) IP trace3 (d) IP trace4

Figure 21: AAE of finding Super-Spreaders.



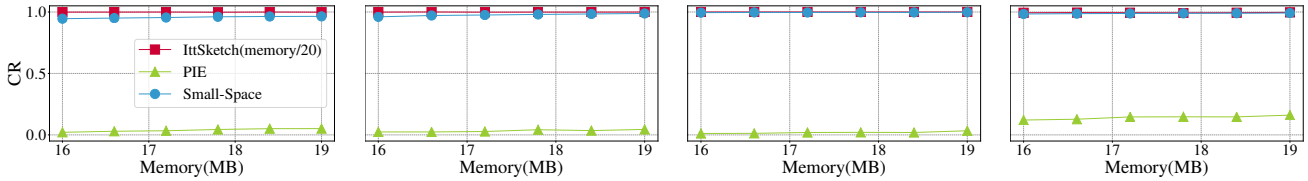
(a) IP trace1 (b) IP trace2 (c) IP trace3 (d) IP trace4

Figure 22: CR of finding Super-Spreaders.



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

Figure 23: AAE of finding Persistent Items.



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

Figure 24: CR of finding Persistent Items.

B EXPERIMENTAL FIGURES ON AAE AND CR

C EVALUATION OF DIFFERENT VERSIONS OF INTERESTSKETCH

Parameter Setting:

We compare 4 versions of InterestSketch, using min-heap, using Stream-Summary, using Multi-Array, and the final version. These different versions are introduced in Section 4. Let c be the number of arrays. In the version using Multi-Array, we set $c = 4$. Let d be the number of cells in each bucket. For the final version of InterestSketch, we set $d = 8$. In this experiment, we compare AAE, ARE, PR, CR, and insertion speed among the 4 versions. We focus on finding the frequent items to ignore the errors brought by the Bloom filter.

AAE (Figure 25(a)-25(d)): We find that, on three real-world datasets and one synthetic dataset, the AAE of the final version of InterestSketch is around 10 times, 20 times, and 1.3 times lower than using min-heap, using Stream-Summary, and using Multi-Array, respectively.

ARE (Figure 26(a)-26(d)): We find that, on three real-world datasets and one synthetic dataset, the ARE of the final version of InterestSketch is around 9 times, 14 times, and 1.3 times lower than using min-heap, using Stream-Summary, and using Multi-Array, respectively.

CR (Figure 27(a)-27(d)): We find that, on three real-world datasets and one synthetic dataset, the CR of the final version of InterestSketch is around 1.1 times and 1.2 times lower than using min-heap and using Stream-Summary. Also, using Multi-Array and the final version have nearly the same CR.

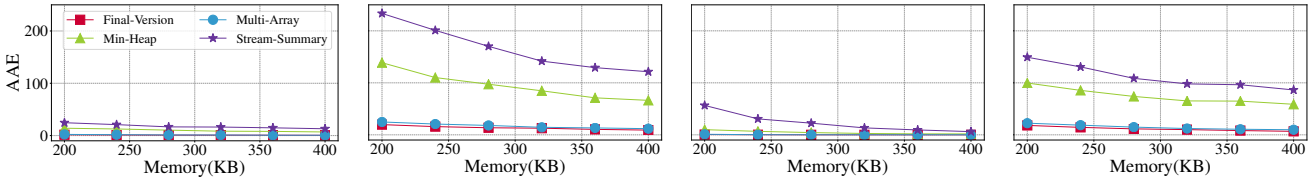
PR (Figure 28(a)-28(d)): We find that, on the Synthetic Dataset and the Web Page Dataset, these four version have nearly the same PR. On the IP Trace Dataset and the Network Dataset, using Stream-Summary and using min-heap perform worse than using Multi-Array and the final version.

Speed (Figure 29(a)-29(d)): We find that, on three real-world datasets and one synthetic dataset, the AAE of the final version of InterestSketch is around 1.7 times, 2 times, and 1.4 times lower than using min-heap, using Stream-Summary, and using Multi-Array, respectively.

Summary:

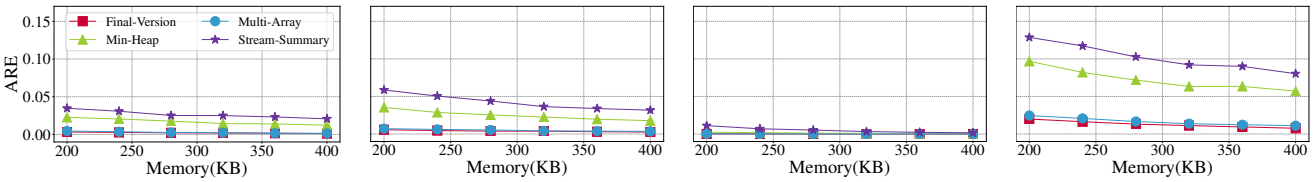
1) The final version and using Multi-Array achieve high accuracy with limited memory. Due to the memory cost of pointers and hash tables, using Stream-Summary and using min-heap cannot record as many items as using Multi-Array and the final version. As a result, they will perform worse.

2) Though using Stream-Summary achieves $O(1)$ time complexity, its insertion speed is the slowest among these 4 versions, because it needs to modify many pointers for each update. In contrast, though the time complexity of using min-heap is $O(logk)$, its insertion speed is often faster than the one of using Stream-Summary, because having too many items in the min-heap is rare. using Multi-Array is slower than the final version because it needs to calculate more hash functions and access memory discontinuously.



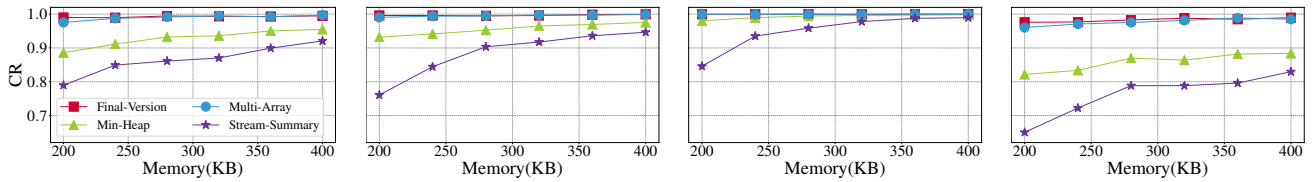
(a) Synthetic dataset (b) IP trace (c) Web page (d) Network dataset

Figure 25: AAE on Different Versions.



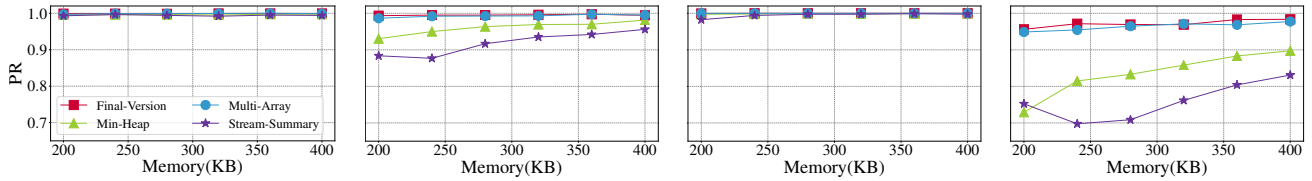
(a) Synthetic dataset (b) IP trace (c) Web page (d) Network dataset

Figure 26: ARE on Different Versions.



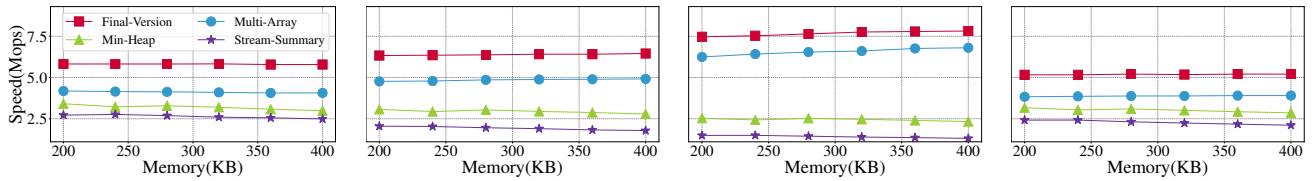
(a) Synthetic dataset (b) IP trace (c) Web page (d) Network dataset

Figure 27: CR on Different Versions.



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

Figure 28: PR on Different Versions.



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

Figure 29: Speed on Different Versions.