A Sketch Framework for Approximate Data Stream Processing in Sliding Windows

Xiangyang Gou, Yinda Zhang, Zhoujing Hu, Long He, Ke Wang, Xilai Liu, Tong Yang, Yi Wang and Bin Cui

Abstract—Data stream processing has become a hot issue in recent years. There are three fundamental stream processing tasks: membership query, frequency query, and Top-K query. While most existing solutions address these queries in fixed windows, this paper focuses on a more challenging task: answering these queries in sliding windows. While most existing solutions address different kinds of queries by using different algorithms, this paper focuses on a generic framework. In this paper, we propose a generic framework, namely the Sliding sketch, which can be applied to many existing solutions for the above three queries, and enable them to support queries in sliding windows. We apply our framework to five state-of-the-art sketches for the above three kinds of queries. Theoretical analysis and extensive experimental results show that the accuracy of existing sketches that do not support sliding windows becomes much higher than the corresponding prior art after using our framework. We released all the source code at Github.

Index Terms—data stream, sliding window, sketch, approximate query

1 Introduction

1.1 Background and Motivations

Data stream processing is a significant issue arising in many applications, like intrusion detection systems [1], [2], financial data trackers [3], [4], sensor networks [5], [6], *etc.* A data stream is composed of an unbounded sequence of items arriving at high speed. Contrary to traditional static datasets, data streams need to be processed in real time, *i.e.*, in one pass, and in $O(1)$ update time. Due to the large volume and high speed, it is difficult and often unnecessary to store the whole data stream. Moreover, large-scale data stream processing applications are usually distributed. Information exchange is needed among multiple hosts which observe local streams. Transporting complete data streams requires lots of bandwidth and is not communication efficient. Instead, one effective choice is to maintain a small summary of the data stream.

Sketches, a kind of probabilistic data structure, achieving memory efficiency at the cost of introducing small errors, have been widely used as the summary of data streams. Sketches only need small memory usage. It is possible to store them in fast memory, such as L2 caches in CPU and GPU chips. Typical sketches include the Bloom filter [7], the CM sketch [8], the CU sketch [9], *etc.* However, these sketches cannot delete outdated items.

In applications of data streams, people usually focus on the most recent items, which reflect the current situation and the future trend. For example, in the financial analysis, people focus on the current finance trend, and in intrusion detection systems, people are mainly concerned about the recent intrusions. It is usually necessary to downgrade the significance of old items and discard them when appropriate. Otherwise, they will bring a waste of memory and also introduce noise to the analysis of recent items. It is important to develop probabilistic data structures which can automatically "forget" old items and focus on recent items.

The most popular model for recording recent items is the sliding window model [10]. It uses a sliding window to include only the most recent items, while the items outside the window are forgotten (deleted). There are various queries that can be implemented in the sliding window model. In this paper, we focus on three kinds of fundamental queries: membership query, frequency query, and Top-K query. Membership query is to check if item e is in the sliding window. Frequency query is to report the frequency of item e in the sliding window. Top-K query is to find all items with top K largest frequencies in the sliding window.

It is challenging to design a probabilistic data structure for the sliding window model. Whenever the window slides, the oldest item needs to be deleted. However, it is challenging to find the oldest item, especially when the demand on memory and speed is high. We have to implement deletions fast enough to catch up with the speed of the data stream. Moreover, we cannot store all items in the sliding window, because the sliding window may be very large and it is memory-consuming to store them.

There have been a few algorithms on approximate queries in sliding windows, like the forgetful Bloom filter [11] supporting membership query and the ECM sketch [12] supporting frequency query. However, existing algorithms

- *Xiangyang Gou, Yinda Zhang, Zhoujing Hu, Long He, Ke Wang, Xilai Liu and Bin Cui are with School of Computer Science and National Engineering Laboratory for Big Data Analysis Technology and Application, Peking University, China. Co-primary authors: Xiangyang Gou, Yinda Zhang, and Zhoujing Hu.*
  E-mail: gxy1995@pku.edu.cn
- *Corresponding author Tong Yang is with School of Computer Science, and National Engineering Laboratory for Big Data Analysis Technology and Application, Peking University, China, and Peng Cheng Laboratory, Shenzhen, China*
  E-mail: yangtongemail@gmail.com
- *Yi Wang is with Institute of Future Networks, Southern University of Science and Technology and Peng Cheng Laboratory, Shenzhen, China.*

- *This work is supported by Key-Area Research and Development Program of Guangdong Province 2020B0101390001, Peng Cheng Laboratory The Major Key Project of PCL (PCL2021A08), National Natural Science Foundation of China (NSFC) (No. U20A20179, 61832001), PKU-Baidu Fund 2019BD006 and Beijing Academy of Artificial Intelligence (BAAI).*

have two main limitations. First, these algorithms usually need a lot of memory to achieve fine-grained deletions. When the space limitation is tight, the accuracy of these algorithms is poor. Second, most existing algorithms only handle one specific query in sliding windows. However, various kinds of queries are usually needed in applications. Thus a general framework is more preferred.

1.2 Our Proposed Solution

In this paper, we propose a framework, namely **the Sliding sketch**. It can be applied to most of the existing sketches and adapt them to the sliding window model. We apply our framework to the Bloom filter [7], the CM sketch [8], the CU sketch [9], the Count sketch [13], and the HeavyKeeper [14] for experimental evaluation in Section 8.

Before we give a brief introduction to the basic idea of our algorithm, we first introduce the common model of sketches. A typical sketch uses an array composed of elements like counters or bits. We call each element in the array a **bucket** in general. This array is divided into k equal-sized segments. Each segment is associated with one hash function. When an item e arrives, the sketch maps it into k buckets using the k hash functions, one in each segment, and records the information of e, like frequency or presence in these buckets. We call these k buckets **the k mapped buckets**. These k mapped buckets usually store k copies of the desired information of e. They have different accuracy because of hash collisions. The hash collision means that multiple items are mapped to the same bucket, and their information is mixed up. In queries, we report the most accurate one in the k mapped buckets. For example, in CM sketches, each bucket is a counter and stores the summary of the frequencies of all items mapped into it. Each item is mapped to k buckets, and these buckets all contain counters larger than or equal to its frequency. We report the smallest one of these k counters as result in queries.

Most existing algorithms keep the basic structure of these sketches and introduce different improvements to apply them to sliding windows. It is difficult to store exactly the information in the sliding window, because it is difficult to delete all outdated information. Therefore, most existing algorithms choose to store a recent slice of the data stream, ω, which approximates to the sliding window. Recall that in the common sketch model, each item has k mapped buckets. In prior algorithms, these k mapped buckets work *synchronously*. These mapped buckets store the information mapped to them in the same slice ω. The difference between ω and the sliding window varies with time, and can be large at some time points. We notice that in queries, the most accurate one in these k mapped buckets is reported. Therefore, we come up with a new idea. **These k mapped buckets can work asynchronously, which means they store information in different slices $\omega_0, \omega_1 ... \omega_{k-1}$. In this way, we can achieve that whenever we query, there is always a mapped bucket that records the frequency or presence of queried items in a slice very close to the sliding window.** We design an algorithm called **the scanning operation** to achieve this. As a result, our algorithm has a much lower error compared with the prior art. In Section 7, we further propose several techniques to accelerate the Sliding sketch and improve the accuracy.

Extensive experiments and theoretical analysis show that the Sliding sketch has high accuracy with small memory usage. Experimental results show that after using our framework, the accuracy of existing sketches that do not support sliding windows becomes much higher than the algorithms for sliding windows. In membership query, the error rate of the Sliding sketch is up to 9 times lower than that of the state-of-the-art (Figure 5). In frequency query, ARE of the Sliding sketch is up to 161 times lower than that of the state-of-the-art (Figure 7). In Top-K query, the error rate of the Sliding sketch is below 5% (Figure 9), and ARE of the frequencies of reported items in the Sliding sketch is up to 15 times lower than that of the state-of-the-art (Figure 10).

1.3 Key Contribution

Our key contributions are as follows:

1) We propose a generic framework named the Sliding sketch, which can be applied to most existing sketches and adapt them to the sliding window model.

2) We propose several techniques to further accelerate the Sliding sketch and improve its accuracy in Section 7, including FPGA acceleration, accelerating mapped bucket address computation and result correction based on jet lag.

2) We apply our framework to three typical kinds of queries in sliding windows: membership query (the Bloom filter), frequency query (sketches of CM, CU, and Count), and Top-K query (the HeavyKeeper). Mathematical analysis and experiments show that the Sliding sketch achieves much higher accuracy than the state-of-the-art. We have released all the source code at Github [15].

2 Related Work

In this section, we introduce different kinds of sketches that can be used in our framework and prior art of probabilistic data structures for sliding windows.

2.1 Different Kinds of Sketches

Sketches are a kind of probabilistic data structure for data stream summarization. Classic sketches support queries in the whole data stream or a fixed slice, but do not support the sliding window model. According to the queries they support, we illustrate three kinds of sketches in this paper: sketches for membership query, sketches for frequency query, and sketches for Top-K query.

2.1.1 Sketches for Membership Query

Membership query is to check if an item is in a set or not. The most well-known sketch for membership query is the Bloom filter [7]. It is composed of an array of m bits. When inserting an item e, the Bloom filter maps it to k bits with k hash functions, and sets these bits to 1. When querying an item e, the Bloom filter checks the k mapped bits and reports false if any of them is 0, true otherwise. The Bloom filter has the property of one-side error. It only has false positives and no false negatives. In other words, if item e is in set s, it will definitely report true, but if e is not in the set, it still has a probability to report true due to hash collisions. In recent years, many variants of Bloom filters have been proposed to meet the requirements of different applications, like the Bloomier filter [16], the Dynamic count filter [17], COMB [18], the shifting Bloom filter [19].

2.1.2 Sketches for Frequency Query

Frequency query is to report the frequency of an item. There are several well-known sketches for frequency queries, like the CM sketch [8], the CU sketch [9], and the Count sketch [13]. The CM sketch is composed of a counter array with k equal-sized segments. All the counters are initialized to 0. When inserting an item e, the CM sketch maps it to k counters with k hash functions, one in each segment, and increases these counters by 1. When querying the frequency of an item e, it finds the k mapped counters with the k hash functions and reports the minimum value among them. The CM sketch only has over-estimation error, which means the reported frequency is no less than the true value. The CU sketch and the Count sketch have the same structure as the CM sketch, but different update and query strategies. They have higher accuracy but suffer from different problems. The CU sketch does not support deletions, and the Count sketch has two-side error, which means the query result may be either bigger or smaller than the true value. Sophisticated sketches for frequency query include the Pyramid sketch [20], the Augmented sketch [21], and [22], [23], [24].

2.1.3 Sketches for Top-K Query

In some applications, we only care about frequent items. Top-K query and heavy hitter query are query models proposed for these applications. Top-K query is to find items with the top K largest frequencies. Heavy hitter query is to find all the items with frequencies exceeding a threshold. Most sketches proposed for recording frequent items support both Top-K query and heavy hitter query. In this paper, we focus on Top-K query model as it is more popular. The state-of-the-art method for Top-K query in data streams is the HeavyKeeper [14]. It provides frequency queries for items in the data stream and gives a guarantee to the accuracy of frequent items' query results. It uses a strategy called count-with-exponential-decay to remove items with small frequencies through decaying, and minimize the impact on frequent items. It reaches very high accuracy in heavy hitter query and Top-K query. Other algorithms for Top-K query include Frequent [25], Lossy counting [26], Space-Saving [27] and unbiased space saving [28].

2.2 Probabilistic Data Structures for Sliding Windows

We divide the prior art of probabilistic data structures for sliding windows into three kinds according to the queries they support. The first kind supports membership query, like the double buffering Bloom filter [29], the A^2 buffering Bloom filter [30] and the Forgetful Bloom filter [11]. The second kind is designed for frequency query, like the ECM sketch [12], the splitter windowed count-min sketch [31], and [32], [33]. The third kind supports Top-K query. This kind includes the window compact space saving (WCSS) [34], and [35], [36]. Unfortunately, none of these algorithms has high accuracy with limited memory. Moreover, most of them are specific to limited kinds of queries.

3 Problem Definition

3.1 Definitions of Data Streams

We give a formal definition of the data stream as follows:

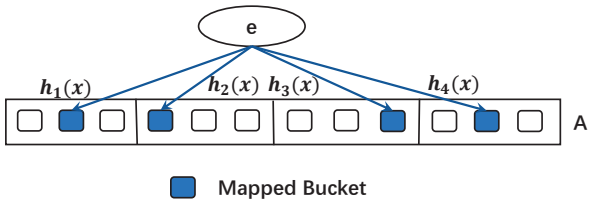


Fig. 1. Structure of the k-hash model

Definition 1. ***Data stream.*** *A data stream is an unbounded sequence of items $S = \{e_1^{t_1}, e_2^{t_2}, e_3^{t_3} ... e_i^{t_i} ...\}$. Each item e_i has a timestamp t_i which indicates its arriving time. In a data stream, the same item may appear more than once.*

3.2 Definitions of Sliding Windows

There are two kinds of sliding windows: time-based sliding windows and count-based sliding windows. The definitions of them are as follows:

Definition 2. ***Time-based sliding window.*** *In data stream S, a time-based sliding window with length N means the union of items that arrive in the last N time units.*

Definition 3. ***Count-based sliding window.*** *In data stream S, a count-based sliding window with length N means the union of the last N items.*

Our framework can be applied in both kinds of sliding windows. For simplicity of presentation, we focus on time-based sliding windows in the majority of the paper. We also use *sliding window* as an abbreviation of the time-based sliding window. In Section 4.2, we will extend our algorithm to count based sliding windows. We use T to denote current time and W to denote current sliding window, namely the union of items arriving from time $T - N$ to current time T. More generally, we use $W_{t_1}^{t_2}$ to denote the union of items arriving from time t_1 to t_2.

3.3 Definitions of Stream Processing Tasks

In a sliding window, there are 3 fundamental queries:

Definition 4. ***Membership query.*** *Given a sliding window W, we want to find out whether an item e is in it.*

Definition 5. ***Frequency query.*** *Given a sliding window W, we want to find out how many times an item e shows up in W and return the number. We call this number the frequency of item e.*

Definition 6. ***Top-K query.*** *Given a sliding window W, we want to find out the items with top K largest frequencies in the sliding window. The frequency is defined as the number of arrivals of an item in the sliding window W.*

The symbols we use in this paper, and their meanings are shown in table 1.

4 Sliding Sketch: Basic Version

In this section, we propose a generic framework for typical data stream processing tasks in sliding windows. First, we introduce a model that many sketches use. Second, based on this common model, we present a basic version of our framework.

TABLE 1
Notation table

Notation	Meaning
S	A data stream
T	Current time.
W	The sliding window.
N	The length of the sliding window.
$W_{t_1}^{t_2}$	Union of items from time t_1 to t_2.
A	The array in the Sliding Sketch
m	The number of buckets in array A
k	The number of segments in array A
$A[i]$	A bucket in array A
d	The number of fields in each bucket
$\{A[i][j]\|0 \leqslant j < d\}$	The fields in bucket $A[i]$
$z[j]\|j = 0, 1, 2....$	the sequence of zero times in bucket $A[i]$
δ	The jet lag in a bucket
$\{A[h_i]\|0 \leqslant i < k\}$	The mapped buckets of the updated/queried item
St_u	The update strategy of the base sketch
St_q	The query strategy of the base sketch

4.1 A Common Sketch Model

This paper focuses on three stream processing tasks: membership query, frequency query, and Top-K query. The state-of-the-art sketches for these tasks use a common model, namely k**-hash model** in this paper. The details of this model are as follows:

Data structure: As shown in Figure 1, the data structure of the k-hash model is an array which is composed of simple and small data structures, like counters or bits. We call each element in the array a **bucket** in general. The array is divided into k equal-sized segments, each associated with a hash function.

Update: To insert an item e, it maps e to k buckets with the k hash functions, one in each segment. We call them the k **mapped buckets**. It updates the k mapped buckets with an **update strategy** St_u, which varies according to the specific sketch.

Query: To query an item e, it computes the k functions and gets the k mapped buckets. The reported result is computed from the values in the k mapped buckets with a **query strategy** St_q. The query strategy also varies according to the specific sketch.

An example using the CM sketch: Different sketches use different update and query strategies. Take the CM sketch [8] as an example. Each bucket in the CM sketch is a counter. Its update strategy St_u increases all the k mapped counters by 1, while its query strategy St_q reports the minimum value among the k mapped counters.

4.2 The Sliding Sketch Model

In this paper, we propose a framework named the **Sliding sketch**, which can be applied to all sketches consistent with the k-hash model and adapt them to the sliding window model. In the following sections, when we apply Sliding sketch technique to a specific scheme, we call the basic sketch before enhancement **base sketch**, and call the algorithm after enhancement "Sliding" + base sketch name for short. For example, when applying Sliding sketch technique to the CM sketch, we call the result algorithm the Sliding CM sketch, and the CM sketch is its base sketch.

Data structure: In the Sliding sketch, we build an array A with m buckets, which are divided into k equal-sized segments. Every bucket $A[i]$ has two fields $A[i][0]$ and $A[i][1]$. Each field is a counter or a bit, depending on the base sketch.

In the Sliding sketch, we have the following 3 operations: update operation, scanning operation, and query operation,

Update operation: When an item e arrives, we use the k hash functions $\{h_i(\cdot)|0 \leqslant i < k\}$ to map the item into k buckets $\{A[h_i]|0 \leqslant i < k\}$, one in each segment. We update $A[h_i][0]$ filed in these k mapped buckets with the update strategy St_u of the base sketch.

Scanning operation: We use a *scanning operation* to delete outdated information. We use a scanning pointer to go through A repeatedly with a constant speed. Every time it reaches the end of the array, it returns to the beginning and starts a new scan. The cycle that the pointer scans the entire array once is equal to the length of the sliding window. In other words, for a sliding window with length N, the scanning pointer goes through $\frac{m}{N}$ buckets in each time unit (if $m < N$, the pointer goes through 1 bucket in every $\frac{N}{m}$ time units). Every time the scanning pointer arrives at a bucket $A[i]$, it is a **zero time** of this bucket. At the zero times of a bucket, we delete the value in $A[i][1]$. Then we copy the value in $A[i][0]$ to $A[i][1]$, and set $A[i][0]$ to 0.

Query operation: When querying an item e in a Sliding sketch, we find the k mapped buckets $\{A[h_i]|0 \leqslant i < k\}$ with the k hash functions. In each mapped bucket, we add the values in two fields and get k sums $\{Sum[i] = A[h_i][0]+ A[h_i][1]|0 \leqslant i < k\}$. We use the query strategy St_q of the base sketch to get the result from these k sums.

Example of the Sliding CM sketch: We take the Sliding CM sketch as an example. In the Sliding CM sketch, each field is a counter. In the update operation, we find the k mapped buckets $\{A[h_i]|0 \leqslant i < k\}$. In each mapped bucket $A[h_i]$, we add $A[h_i][0]$ by 1. The scanning operation is the same as discussed above. In the query operation, we find the k mapped buckets and add up two counters in each bucket to get k sums. We return the minimum value among the k sums. In Appendix A of the supplementary materials, we will introduce the other 4 Sliding sketches we use in our experiments.

Extend to count-based sliding windows: In the count-based sliding window, the only change lies in the scanning operation. For a count-based sliding window with length N, we move the scanning pointer upon each item arrival rather than each time unit. Suppose the array length is m, we move the pointer $\frac{N}{m}$ buckets forward upon each item arrival. The other operations are the same as the time-based sliding window. Similarly, in the following analysis and optimizations, we just need to replace time units with item arrivals for the count-based sliding window, and all the theoretical results and optimization techniques are the same.

The scanning operation is our major novelty, which generates an array of zero times for each bucket. These zero times split the data stream into multiple slices, which we call **Days**. The two fields record information of items mapped to this bucket in the most recent two Days. The arrays of zero times are different in different buckets, because the scanning pointer scans them at different times. They are like different **time zones** with **time differences**. This asynchrony is our major difference with former algorithms with data stream

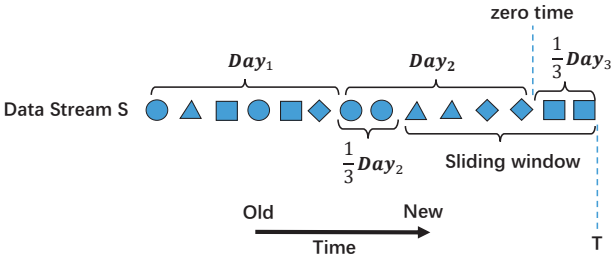


Fig. 2. Example of the sliding window and the Days

slicing like [37]. We will discuss this asynchrony and its benefit in detail in Section 4.3.

4.3 Analysis of the Sliding Sketch Model

The key technique of the Sliding sketch is the scanning operation. It controls the aging procedure of the array. In this section, we analyze this operation in detail.

First we analyze the slices we record in the Sliding sketch. Due to the scanning operation, each bucket $A[i]$ has an array of zero times $\{z[j] \mid j = 0, 1, 2....\}$. For any integer $j \geqslant 0$, we have $z[j+1] - z[j] = N$. These zero times split the data stream into multiple **Days** $\{Day_j = W_{z[j]}^{z[j+1]} \mid j = 0, 1, 2....\}$. Suppose for current time T, we have $z[t] < T \leqslant z[t+1]$, namely current active Day is Day_t. We suppose that $\frac{T - z[t]}{N} = \delta (0 < \delta \leqslant 1)$. In other words, only δ of Day_t has passed at time T. Notice that different buckets have different δ, as they have different zero time arrays. We call δ the **jet lag** in this bucket.

We use $A[i][0]$ to store the information (frequency or presence) of items mapped to $A[i]$ in Day_t, and the previous Day, Day_{t-1} is stored in $A[i][1]$ field. By adding $A[i][0]$ and $A[i][1]$ together, we get the information of items mapped to $A[i]$ in $Day_t + Day_{t-1} = W_{z[t-1]}^{T}$. Because $z[t-1] < T - N \leqslant z[t] < T$, $W_{z[t-1]}^{T}$ is a superset of the sliding window, and is $1 + \delta$ times larger than it. We can use $A[i][0] + A[i][1]$ as an approximation of the information of items mapped to $A[i]$ in the sliding window. Moreover, this approximation only introduces over-estimation error, and thus keeps the over-estimation-only property of many sketches including the CM sketch, the CU sketch and the Bloom filter. For other sketches with under-estimation error, we can vary the scanning speed to fit with them, as will be discussed in Section 7.1.

Second, we use an example to explain the relationship between the Days and the sliding window.

Example 1. *An example of the Days in bucket $A[i]$ and the sliding window of the data stream is shown in Figure 2. In this example, current time T is in the 3_{rd} Day, namely $t = 3$. At time T, $\frac{1}{3}$ of Day_3 has passed. The length of the sliding window is equal to one Day, thus Day_3 is only $\frac{1}{3}$ of the sliding window, and the other $\frac{2}{3}$ is in Day_2. We record both the information in Day_3 and Day_2 to estimate the sliding window.*

At last we analyze the influence of jet lag δ and the value range of δ in the k mapped buckets of an item. Obviously, δ influences the accuracy of our estimation. In each bucket $A[i]$, $A[i][0] + A[i][1]$ records a slice with length $N + \delta N$, which is $(1 + \delta)$ times of the sliding window. Therefore, when we use $A[i][0] + A[i][1]$ to approximate the information in the sliding window, the smaller δ is, the more accurate this approximation is. Because the scanning pointer goes through the array at a constant speed, δ depends on the distance between the bucket and the scanning pointer. Assuming the scanning pointer is at the q_{th} bucket at time T, for a bucket $A[i]$ with index i, δ in this bucket can be computed as follows:

$$\begin{cases} \delta = \dfrac{q - i}{m} & (i < q) \\ \delta = 1 - \dfrac{i - q}{m} & (i \geqslant q) \end{cases} \tag{1}$$

Derivation of the equation is shown in the Section 6.2.

In the Sliding sketch, each item is mapped to k buckets. These mapped buckets have different δ because of the scanning operation. We can prove that for each item e, there must be a mapped bucket $A[h_{j_1}]$ where $0 < \delta < \frac{2}{k}$, and a mapped bucket $A[h_{j_2}]$ where $\frac{k-2}{k} < \delta \leqslant 1$. For other $k - 2$ mapped buckets, the value range is $0 < \delta \leqslant 1$. Detailed analysis is shown in the Section 6.2.

The value range of δ in the k mapped buckets gives a guarantee to the accuracy. It guarantees that whenever we query, there is always at least one mapped bucket which records a slice very close to the sliding window. For example, in the Sliding CM sketch, there is always one mapped bucket which records the frequency of the queried item in the most recent slice with length $N \sim \frac{k+2}{k} N$[1]. This slice is no larger than $\frac{k+2}{k}$ times of the sliding window. Because the query strategy of the CM sketch is to find the smallest mapped counter, the existence of such bucket guarantees the quality of the query result. Detailed accuracy analysis of the sliding CM and other sketches is shown in Appendix B of the supplementary materials.

5 Sliding Sketch: d-field version

More generally, when the memory is sufficient, we can use $d (d \geqslant 2)$ fields $\{A[i][j] | 0 \leqslant j < d\}$ in each bucket $A[i]$. These d fields record the information in the last d Days. If we suppose that current Day is Day_t, $A[i][j]$ records information in Day_{t-j}. In this case, the length of each Day should be $\frac{1}{d-1}$ of the sliding window. The basic operations in the d-field version are as follows:

Update operation: When an item e arrives, we use the k hash functions to map e to k buckets $\{A[h_i] | 0 \leqslant i < k\}$, one in each segment. We update the $A[h_i][0]$ field in these k mapped buckets with strategy St_u of the base sketch.

Scanning operation: The scanning pointer scans $\frac{(d-1) \times m}{N}$ buckets in each time unit. When the scanning pointer arrives at bucket $A[i]$, we set $A[i][j] = A[i][j-1] (1 \leqslant j < d)$ and $A[i][0] = 0$. Because a new Day starts, all the stored information becomes one Day older.

Query operation: When querying an item e, we find the k mapped buckets $\{A[h_i] | 0 \leqslant i < k\}$ for e with the k hash functions. Then we compute k sums $\{Sum(A[h_i]) = \sum_{j=0}^{d-1} A[h_i][j] | 0 \leqslant i < k\}$. At last, we get the query result based on these k sums with strategy St_q of the base sketch.

Next we use an example to illustrate the Days in the d-field version:

1. Error brought by hash collisions also exits. But for simplicity of presentation, we focus on error brought by summarizing a slice larger than the sliding window in this section.

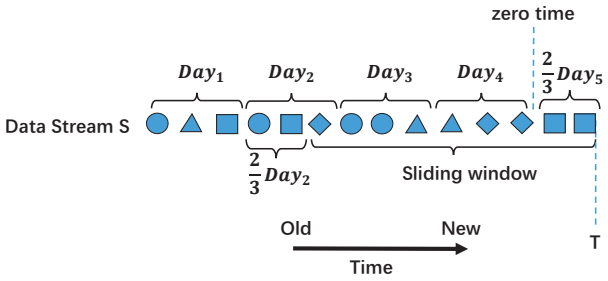


Fig. 3. Sliding window and Days in the d-field version

Example 2. *An example of the Days of bucket $A[i]$ in the d-field version is shown in Figure 3. In this example, we set $d = 4$. Each bucket contains 4 fields, and the length of each Day is $\frac{1}{3}$ of the sliding window. At time T, bucket $A[i]$ records $Day_2 \sim Day_5$ and $\delta = \frac{2}{3}$. From the figure, we can see that $\frac{2}{3}$ of the oldest Day, Day_2 is not in the sliding window, and $A[i]$ records a slice which is $\frac{11}{9}$ times of the sliding window. Notice that different buckets have asynchronous Days.*

When we use multiple fields, the accuracy may become higher. The jet lag δ is still the same as the basic version. As Days in each bucket are $\frac{1}{d-1}$ of the sliding window, the error brought by the approximation of the sliding window also becomes $\frac{1}{d-1}$. To be specific, the d fields in each bucket record the most recent d Days. Current Day, namely Day_t, only has the length of $\frac{\delta N}{d-1}$, and $Day_{t-d+1} \sim Day_{t-1}$ has the length of $\frac{N}{d-1}$. Therefore, $Sum(A[i])$ summarizes a slice of $\frac{N}{d-1} \times (d-1) + \frac{\delta N}{d-1} = (1 + \frac{\delta}{d-1})N$. The excess part is $\frac{\delta}{d-1}N$, smaller than δN in the basic version if $d > 2$. However, increasing d does not necessarily bring improvements in accuracy. When using the same amount of memory, enlarging d means the length of the array becomes smaller, and the error brought by hash collisions will increase. The trade off among the number of fields d, the length of the array m, and the number of segments k depends on different base sketches, and experimental attempt is recommended in applications. We carry out an experiment of parameter settings in the Sliding CM sketch and the Sliding CU sketch. The result is shown in Section 8.5.

6 Mathematical Analysis

In this section, we analyze the memory and time cost of the Sliding sketch, and the value range of δ. Accuracy of the Sliding sketch depends on the base sketch, and we analyze the accuracy of 3 kinds of Sliding sketch applications as examples, namely the Sliding Bloom filter, the Sliding CM sketch, and the Sliding HeavyKeeper. The accuracy analysis is given in Appendix B of the supplementary materials due to space limitation.

6.1 Analysis of Memory and Time Cost

The space cost of the Sliding sketch is $O(md)$, where m is the length of the bucket array, and d is the number of fields in each bucket. In applications, m is usually linear correlated with the length of the sliding window, while d is a small constant. Therefore the space cost is also $O(N)$. More specifically, The memory usage of a Sliding sketch is no larger than d times of the base sketch. In most Sliding sketch applications, each bucket is split into d fields, and each field has the same structure as a bucket in the base sketch. Therefore the memory consumption of the Sliding sketch is d times of a base sketch with the same length. However, in some kinds of Sliding sketches, the memory consumption is smaller. For example, each bucket in the HeavyKeeper is a key-value pair, where the key is an item's ID and the value is its frequency. In the Sliding HeavyKeeper, each bucket contains 1 key and d counters. Thus the memory consumption is smaller than d times of a HeavyKeeper with the same length.

The time cost of the update operation and the query operation is $O(k)$. We need to find the k mapped buckets of the new item (or the queried item), and update these buckets or retrieve the information in these buckets. The cost of the scanning operation is $O(\frac{(d-1)\times m}{N})$, because $\frac{(d-1)\times m}{N}$ buckets need to be scanned in each time unit. In applications, k and d are small constants set by users, and the length of the array m is linear correlated with the window length N. Therefore, the time cost of the update operation, the query operation and the scanning operation can be seen as $O(1)$.

6.2 Analysis of the Jet Lag δ

6.2.1 Computation of δ

For each bucket $A[i]$ in the array, the jet lag δ, which represents how much of current Day has passed by current time T, can be computed with the distance between the bucket and the scanning pointer. Suppose the index of the bucket in the array is i, and the position of the scanning pointer is q. The scanning pointer moves in a constant speed and scans each bucket in $\frac{1}{m}$ Day. There are two kinds of situations:

1) When $i < q$, the scanning pointer has scanned $q - i$ buckets after its last arrival at $A[i]$. Therefore

$$\delta = \frac{q-i}{m} \ (i < q) \tag{2}$$

2) When $i \geqslant q$, the scanning pointer has scanned $(m - i) + q$ buckets after its last arrival at $A[i]$. Therefore

$$\begin{aligned} \delta &= \frac{m-i+q}{m} \\ &= 1 - \frac{i-q}{m} \ (i \geqslant q) \end{aligned} \tag{3}$$

If the scanning pointer is just in $A[i]$, we define $\delta = 1$.

6.2.2 Value Range of δ

Theorem 1. *Given an item e with k mapped buckets in the Sliding sketch, the jet lags in all these mapped buckets are in range $(0, 1]$. Moreover, There must be at least one mapped bucket with a jet lag δ smaller than $\frac{2}{k}$, and at least one mapped bucket with a δ larger than $1 - \frac{2}{k}$.*

Theorem 2. *Given an item e with k mapped buckets, there are at least $i - 1$ mapped buckets where $\delta < \frac{i}{k}$ $(2 \leqslant i \leqslant k - 1)$ and there are k mapped buckets where $\delta \leqslant 1$.*

Theorem 3. *Given an item e with k mapped buckets, there are at least $i-1$ mapped buckets where $\delta > 1-\frac{i}{k}$ $(2 \leqslant i \leqslant k-1)$, and there are k mapped buckets where $\delta > 0$.*

Proof. For each item e in the data stream, the Sliding sketch maps it to k mapped buckets, one in each segment. Therefore there must be a mapped bucket $A[h_{j_1}]$ which is in the same segment with the scanning pointer. There are two kinds of situations:

1) When $h_{j_1} < q$, $A[h_{j_1}]$ has the smallest δ among the k mapped buckets. $q - h_{j_1}$ is less than the length of the segment, which is $\frac{m}{k}$. In bucket $A[h_{j_1}]$, we have

$$\delta = \frac{q-h_{j_1}}{m} \quad < \frac{\frac{m}{k}}{m} = \frac{1}{k} \tag{4}$$

Therefore in this bucket $A[h_{j_1}]$ we have $0 < \delta < \frac{1}{k}$. In this situation the largest δ appears in the next segment, we represent the mapped bucket in this segment with $A[h_{j_2}]$. Then $h_{j_2} - q$ is less than the length of two segments, which is $\frac{2\times m}{k}$. In bucket $A[h_{j_2}]$, we have

$$\delta = 1-\frac{h_{j_2}-q}{m} \quad > 1-\frac{\frac{2\times m}{k}}{m} = 1-\frac{2}{k} \tag{5}$$

Therefore in this bucket $A[h_{j_2}]$ we have $1-\frac{2}{k} < \delta < 1$. For the other $k-2$ mapped buckets, as they are mapped to different segments and each segment has the same length, value ranges of δ in them form an arithmetic sequence $\{(\frac{j-1}{k}, \frac{j+1}{k}) | 1 \leqslant j \leqslant k-2\}$.

2) When $h_{j_1} \geqslant q$, $A[h_{j_1}]$ has the largest δ among the k mapped buckets. $h_{j_1} - q$ is less than the length of the segment, which is $\frac{m}{k}$. In bucket $A[h_{j_1}]$, we have

$$\delta = 1-\frac{h_{j_1}-q}{m} \quad > 1-\frac{\frac{m}{k}}{m} = 1-\frac{1}{k} \tag{6}$$

Therefore, in this bucket $A[h_{j_1}]$, we have $\frac{k-1}{k} < \delta \leqslant 1$. In this situation the smallest δ appears in the last segment, we represent the mapped bucket in this segment with $A[h_{j_2}]$. Then $q - h_{j_2}$ is less than the length of two segments, which is $\frac{2\times m}{k}$. In bucket $A[h_{j_2}]$, we have

$$\delta = \frac{q-h_{j_2}}{m} \quad < \frac{\frac{2m}{k}}{m} = \frac{2}{k} \tag{7}$$

Therefore, in this bucket $A[h_{j_2}]$, we have $0 < \delta < \frac{2}{k}$. For the other $k-2$ mapped buckets, as they are mapped to different segments and each segment has the same length, value ranges of δ in them form an arithmetic sequence $\{(\frac{j}{k}, \frac{j+2}{k}) | 1 \leqslant j \leqslant k-2\}$.

Combining the value ranges in these 2 kinds of situations, we can easily get Theorem 1, 2 and 3.

□

7 More Optimizations

7.1 Varying the Scanning Speed

In the sections above, for a Sliding sketch with d fields in each bucket ($d \geqslant 2$), we let the scanning pointer scan $\frac{(d-1)\times m}{N}$ buckets in each time unit. As a result, each Day is $\frac{1}{d-1}$ of the sliding window. This scanning speed keeps the over-estimation-only property of many sketches, like the Bloom filter, the CM sketch and the CU sketch. We use it as a basic case. However, for other sketches with under-estimation error, we need to use different scanning speed.

Generally, we suppose that each Day has length θN, namely θ times of the sliding window. The d fields in each bucket record the most recent d Days, which are $(d-1+\delta)\theta N$. For the Sliding Bloom filter, the Sliding CM sketch and the Sliding CU sketch, we set $\theta = \frac{1}{d-1}$ as discussed above. In the follows, we analyze how to set θ for the other two Sliding sketches we use in experiments, the Sliding HeavyKeeper and the Sliding Count sketch.

The HeavyKeeper only has under-estimation error, and it returns the largest value among the k mapped buckets as the estimated frequency of the queried item. We want to keep the under-estimation-only property in the Sliding HeavyKeeper. Therefore, we hope that the slice we record in each bucket is no larger than the sliding window. The d fields in a bucket record the item frequency in a slice of $d-1+\delta$ Days. Because $0 < \delta \leqslant 1$, $d-1 < (d-1+\delta) \leqslant d$. We set $\theta = \frac{1}{d}$ to keep $(d-1+\delta)\theta N \leqslant N$. In other words, each Day is $\frac{1}{d}$ of the sliding window, and the scanning pointer need to scan $\frac{d\times m}{N}$ buckets in each time unit. Because the one-side error property is kept in the Sliding HeavyKeeper, we can sum up the fields in each bucket and select the largest value among the k mapped buckets as the most accurate estimation. As analyzed in Section 6, there are always buckets with large enough δ, which guarantee the accuracy. Detailed implementation and analysis of the Sliding HeavyKeeper is shown in Appendix A.5 and B.3 of the supplementary materials.

The Count sketch has two-side errors, and it selects the medium value among the k mapped buckets as the estimated frequency of the queried item. In the Sliding Count sketch, we hope that the medium value in the k mapped buckets records a slice close to the sliding window. In each mapped bucket, the sum of the d fields record a slice of $(d-1+\delta)$ Days. According to the analysis in Section 6.2.2, the medium value of δ in the k mapped buckets of each item is $(\frac{1}{2}-\frac{3}{2k}, \frac{1}{2}+\frac{1}{2k})$ or $(\frac{1}{2}-\frac{1}{2k}, \frac{1}{2}+\frac{3}{2k})$. We approximately take the value of $\frac{1}{2}$. Therefore, the medium value of the k mapped buckets is expected to record a slice of $(d-\frac{1}{2})\theta N$. We let $(d-\frac{1}{2})\theta N = N$, and $\theta = \frac{2}{2d-1}$. In other words, each Day is $\frac{2}{2d-1}$ of the sliding window, and the scanning pointer need to scan $\frac{(2d-1)\times m}{2N}$ buckets in each time unit. Detailed implementation of the sliding Count sketch is shown in Appendix A.3 of the supplementary materials.

7.2 Accelerating Mapped Bucket Address Computation

Sliding sketches need a large number of hash computations to get high accuracy. In applications, we usually need the number of mapped buckets of each item, k, to be set to $5 \sim 10$. If we compute the address of these mapped buckets as $\{h_i = H_i(e)\%\frac{m}{k} | 0 \leqslant i < k\}$ where $\{H_i(\cdot)\}$ are independent hash functions, computation of these hash functions will be time consuming. In order to solve this problem, we propose to use pseudo random algorithm to accelerate the computation. The details are as follows:

For each item e, we only compute two hash functions, $H_1(\cdot)$ and $H_2(\cdot)$, and set two variables $g_1(e) = H_1(e)\%\frac{m}{k}$ and $g_2(e) = H_2(e)\%\frac{m}{k}$. We use $g_1(e)$ as a seed to generate pseudo random sequences with linear congruence generator

[38], and use $g_2(e)$ as an offset. We generate the pseudo random sequence $\{r_i(e)|0 \leqslant i < k\}$ as follows:

$$\begin{cases} r_i(e) = g_1(e) \ \ (i = 0) \\ r_i(e) = (a \times r_{i-1}(e) + b)\%\dfrac{m}{k} \ \ (1 \leqslant i < k) \end{cases} \tag{8}$$

a and b are two constants which satisfies:

1) b and $\frac{m}{k}$ are coprime;
2) All the prime factors of $\frac{m}{k}$ can divide $a-1$.
3) If $\frac{m}{k}$ is the multiple of 4, so is $a-1$;
4) Both a and b are smaller than $\frac{m}{k}$.

The property of linear congruence generator guarantees that it can generate independent random sequences with different seeds. However, if two items e and e' has the same seed $g_1(e)$ and $g_1(e')$, they will get the same pseudo random sequence. In order to avoid heavy hash collisions, we add an offset $g_2(e)$ to the pseudo random sequence to get the mapped bucket addresses: $\{h_i(e) = (r_i(e)+g_2(e))\%\frac{m}{k} \mid 0 \leqslant i < k\}$. In this method, two items will get the same set of mapped bucket addresses only when they have the same seed and the same offset. The probability of this case is only $\frac{k^2}{m^2}$, which is extremely small. Experimental result in Section 8.6 shows that with pseudo random algorithm, we can accelerate the update and query of the Sliding sketch by up to 1.68 times without influence on the accuracy.

7.3 Accelerating with FPGA

FPGA (Field Programmable Gate Array) is a kind of widely used acceleration hardware. It has high parallelism and low energy consumption. An FPGA acceleration board is composed of a chip and multiple memory banks. Each memory bank has multiple ports connected with the chip, allowing memory access in parallel. Nowadays, the SRAM memory on FPGA chip can be as large as tens of million bytes, allowing us to store data structures like sketches in it. We can use FPGA to accelerate the Sliding sketch.

When implementing Sliding sketch with FPGA, we store the sketch in on-chip SRAM of FPGA. Items in the data stream are transported from the CPU host to FPGA through PCI bus and processed by FPGA kernels. Compared with the CPU implementation, the acceleration lies in three aspects. First, the locality of the scanning operation is improved. Second, update and query operations in different segments are paralleled. Third, updates and queries of different items are pipelined. We made the following changes to the sliding sketch model:

Array reshaping: We reshape the array A of the Sliding sketch into a matrix of size $m \times d$. Each bucket is organized as a column, and the $i_{th}(0 \leqslant i < d)$ row is composed of the i_{th} field in all m buckets. We store the matrix with row-major order. In other words, fields in the same row are stored together. Besides, like the CPU version, the matrix is partitioned into k equal-sized segments. Different segments are stored separately so that we can access them in parallel. The organization of the FPGA-implementation of the Sliding sketch is shown in Figure 4.

This change intends to improve locality in the scanning operation. FPGA can read or write 64 bytes in one memory request, and each field of the Sliding sketch is usually as small as 4 bytes or less. Therefore, after reshaping, we can copy several adjacent values in the i_{th} row to the $(i+1)_{th}$ row in one memory access.

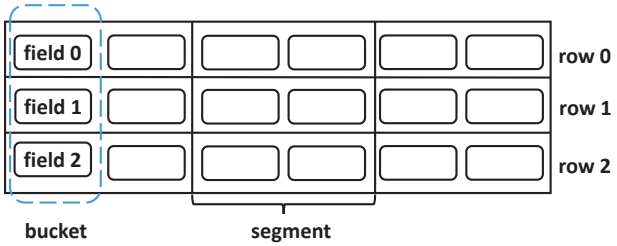


Fig. 4. Sliding sketch in FPGA-implementation

To be specific, in each time unit, we move the scanning pointer $C = \frac{(d-1)m}{N}$ buckets forward[2]. Suppose each field of the Sliding sketch is l bytes. We can divide the C buckets into multiple groups with size $\frac{64}{l}$. In each group, we carry out $k-1$ data transports, and in the $i_{th}(1 \leqslant i \leqslant k-1)$ transport, we copy the $(k-i-1)_{th}$ field of the buckets in this group to the $(k-i)_{th}$ field. As stated above, this transport can be conducted with one read request and one write request. At last, we set the 0_{th} field of these buckets to 0. We scan groups of buckets recursively until all the C buckets are scanned. In order to make full use of such locality, we recommend to carry out scanning operations in small batches. In other words, we scan $C = B \times \frac{(d-1)m}{N}$ buckets together in each B time units.

Inter-segment parallelism: In the update or query operation of an item e, computation of the mapped bucket addresses and access of the mapped buckets in different segments can be paralleled.

In computations of mapped bucket addresses, we use the pseudo random function as discussed in Section 7.2. We compute the k mapped bucket addresses $\{h_i(e)\}$ as

$$\begin{cases} r_i(e) = g_1(e) \ \ (i = 0) \\ r_i(e) = (a^i \times g_1(e) + \sum_{j=0}^{i-1} a^j \times b)\%\dfrac{m}{k} \ \ (1 \leqslant i < k) \end{cases} \tag{9}$$

and $\{h_i(e) = (g_2(e) + r_i(e))\%\frac{m}{k}\}$. Equation 9 is an expansion of Equation 8, which removes the recursive part. We can first compute hash functions $g_1(e)$ and $g_2(e)$, and then compute the k addresses in k channel parallel. Parameters including $\{a^i\}$ and $\{\sum_{j=0}^{i-1} a^j \times b\}$ can be pre-computed and stored, so that they can be directly used in address computations.

The update or query of the k mapped buckets can also be carried out in parallel. As discussed above, different segments of the Sliding sketch are stored separately, and k mapped buckets of an item are located in different segments. Therefore, in the update operation, we can use k parallel units to read the mapped buckets of an item, update the value, and write them back in k channels. In query operation, we can also use k parallel units to read the mapped buckets and add up the fields. Then we merge the result of the k channels and get the final query result.

2. In different Sliding sketches this speed may be different, as discussed in Section 7.1

Pipeline in batches: We can combine multiple update operations, like 400, into a batch and pipeline them. The initiation interval of the pipeline [3] depends on the base sketch we choose. In the update operation, we need to first read the mapped buckets of an item, and then write them back after updating them. An update operation cannot read the Sliding sketch until the last operation finishes writing. Therefore the complexity of the update strategy decides the initiation interval. For the CM sketch and the Bloom filter which have a simple update strategy, the initiation interval is 2 cycles.

In the query operation, we can also pipeline the queries in batches. Different from update, a query operation only needs 1 cycle to read the Sliding sketch, and then it releases the sketch for later operations to read. No writing back is needed. Therefore, the initiation interval of the query pipeline is 1 cycle.

With the above techniques, we can accelerate the update and query speed of Sliding sketch to at most 45.8 times without influence the accuracy. In Section 8.6, we will show the performance of FPGA implementation of the Sliding CM sketch as an example.

7.4 δ-based Correction

In the Sliding CM sketch and the Sliding CU sketch which evaluate item frequencies and have one-side error, we can use jet lag δ to help us improve the accuracy. We call this strategy δ-based correction. For each mapped bucket $A[h_i]$ of the queried item e, we find another mapped bucket $A[h_j]$, which has larger jet lag than $A[h_i]$ and the minimum 0_{th} field. The slice record by $A[h_j][0]$ is a superset of the slice record by $A[h_i][0]$, and the frequency of e in the slice record by $A[h_i][0]$ should be no larger than $A[h_j][0]$. If $A[h_i][0] > A[h_j][0]$, $A[h_i][0]$ must suffers from heavier hash collisions. We can correct the value of $A[h_i][0]$ to $A[h_j][0]$ during the query procedure to improve accuracy. Note that this correction is temporary and only has effect in the query of e. After the query $A[h_i][0]$ need to recover to its original value, because it also records the frequency of other items.

8 Performance Evaluation

In this section, we apply the Sliding sketch to five kinds of sketches: the Bloom filter [7], the CM sketch [8], the CU sketch [9], the Count sketch [13], and the HeavyKeeper [14]. We call these specific schemes the Sliding Bloom filter, the Sliding CM sketch, the Sliding CU sketch, the Sliding Count sketch, and the Sliding HeavyKeeper, respectively. We compare them with the state-of-the-art sliding window algorithms in different queries under the same memory usage in Section 8.2, 8.3, and 8.4. We also analyze the impact of the number of fields d and the number of hash functions k in the Sliding sketch in Section 8.5. The scanning speed computed for Sliding HeavyKeeper and Sliding Count sketch in Section 7.1 are already used in the experiments in the above Sections. In Section 8.6, we evaluate the effect of improving techniques proposed in Section 7.2, 7.3, and 7.4.

3. The initiation interval indicates the latency of the pipeline. If we denote it with II, an update operation has to wait II cycles after the last operation starts.

TABLE 2
Abbreviations of algorithms in experiments

Abbreviation	Full name
Sl-BF	Sliding Bloom Filter
FBF	Forgetful Bloom Filter[11]
SWBF	Technique in [37] applied to the Bloom filter
Sl-CM	Sliding CM Sketch
Sl-CU	Sliding CU Sketch
Sl-Count	Sliding Count Sketch
ECM	Exponential Count-Min Sketch[12]
SWCM	Splitter Windowed Count-Min Sketch[31]
Sl-HK	Sliding HeavyKeeper
λ-sampling	λ-sampling Algorithm[36]
WCSS	Window Compact Space-Saving[34]

8.1 Experimental Setup

Datasets:

1) IP Trace dataset: IP trace dataset contains anonymized IP trace streams collected in 2016 from CAIDA [4]. Each item is identified by its source IP address (8 bytes).

2) Web Page dataset: We download Web page dataset from the website [5]. Each item (4 bytes) represents the number of distinct terms in a web page.

3) Synthetic dataset: By using Web Polygraph [39], an open source performance testing tool, we generate the synthetic dataset, which follows the Zipf [40] distribution. This dataset has 32M items, and the skewness is 1.5. The length of each item is 4 bytes.

4) StackOverflow dataset: This is a dataset of interactions on the stack exchange website StackOverflow [6]. Each item has three values u, v, t, which means user u answered user v's question at time t. We use u as the ID and t as the timestamp of an item.

For the first 3 datasets with no timestamps, we carry out experiments in count-based sliding windows. For StackOverflow dataset, we use time-based sliding windows. The unit of window size in count-based sliding windows is item arrival. In time-based sliding windows, the unit is the average time span between item arrivals, namely the total time span divided by the number of item arrivals.

Implementation: We implemented the algorithms in C++ and made them open sourced [15]. The hash functions are 32-bit Bob Hash (obtained from the open source website [7]) with different initial seeds. Abbreviations of algorithms we use in experiments and their full name are shown in Table 2. We conducted the experiments on a machine with two 6-core processors (12 threads, Intel Xeon CPU E5-2620 @2 GHz) and 62 GB DRAM memory. Each processor has three levels of cache: one 32KB L1 data cache and one 32KB L1 instruction cache for each core, one 256KB L2 cache for each core, and one 15MB L3 cache shared by all cores.

Metrics: In experiments, we measure the metrics whenever the window slides $\frac{1}{10}N$ and compute the average value (N is the length of the sliding window). We use the average value as the experiment result. We use the following metrics to evaluate the performance of our algorithms:

4. http://www.caida.org/data/overview/
5. http://fimi.ua.ac.be/data/
6. http://snap.stanford.edu/data/
7. burtleburtle.net/bob/hash/evahash.html

1) Error rate in membership estimation: Ratio of the number of incorrectly reported queries to all queries. We use error rate because FBF and SWBF have two-side error. The query set includes all the n distinct items in the sliding window and n items not in the sliding window.

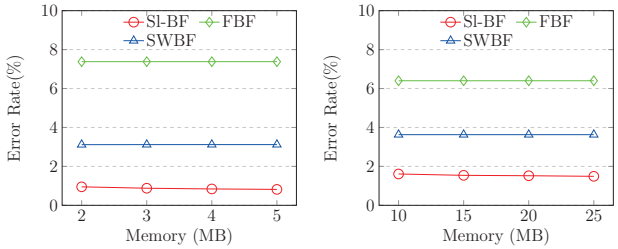
2) Average relative error (ARE) in frequency estimation: $\frac{1}{|\mathbf{\Psi}|}\sum_{e_i\in\mathbf{\Psi}}\frac{|f_i-\widehat{f}_i|}{f_i}$, where f_i is the real frequency of the queried item e_i, and $\widehat{f}_i$ is its estimated frequency. $\mathbf{\Psi}$ is the query set. We query each distinct item once in the sliding window.

3) Error rate in finding Top-K items: Ratio of the number of wrongly reported Top-K items to the total number of Top-K items, namely K.

4) Average relative error (ARE) in finding Top-K items: $\frac{1}{|\mathbf{\Psi}|}\sum_{e_i\in\mathbf{\Psi}}\frac{|f_i-\widehat{f}_i|}{f_i}$, where f_i is the real frequency of the queried item e_i, and $\widehat{f}_i$ is its estimated frequency. $\mathbf{\Psi}$ is the set of reported Top-K items.

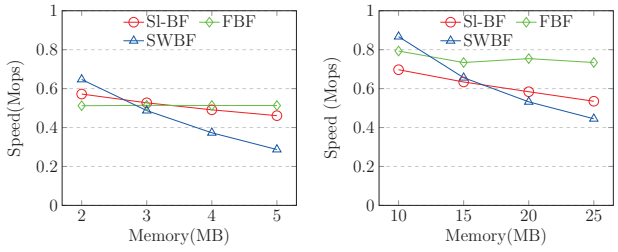
5) Speed: Million operations (insertions / queries) per second (Mops). We repeat speed experiments 100 times and compute the average value to avoid the influence of instability of system performance.

8.2 Evaluation on Membership Query



(a) IP trace. (b) StackOverflow dataset.

Fig. 5. Error rate of membership query.



(a) IP trace. (b) StackOverflow dataset.

Fig. 6. Insertion speed of membership query.

Parameter setting: We compare 3 approaches: Sl-BF, FBF, and the SWBF. For our Sl-BF, we set the number of segments $k = 15$, and the number of fields in each bucket $d = 2$. For FBF, we set the total number of Bloom filters to 10, and each Bloom filter uses 8 hash functions. For SWBF, we use a 2-level structure. In the first level, we split the sliding window into 16 blocks, and in the second level, we split the sliding window into 8 blocks. For each block, we use a small Bloom filter with 3 hash functions. Details of the algorithms can be seen in the original papers [37], [11]. We control the memory usage by changing the size of Bloom filters in these data structures. We use IP trace and StackOverflow dataset in this experiment. For IP trace, we set window length $N = 1M$, and read $8M$ items. For StackOverflow, we set $N = 5M$, and read $40M$ items.

Error rate (Figure 5(a)-5(b)): Our results show that the error rate of Sl-BF is up to 9 and 3.8 times lower than FBF and SWBF, respectively. Both FBF and SWBF follow the idea of splitting the sliding window into small slices, and use union of slices to approximate the sliding window. However, the quality of such approximation in these algorithms varies with time. On the other hand, Sl-BF can always get a guaranteed approximation with the asynchronous mapped buckets. As a result, their average accuracy is not as good as our Sl-BF. The error of these algorithms is composed of 2 parts, error brought by hash collisions in the Bloom filters, and error brought by approximation of the sliding windows. Enlarging the Bloom filters in these data structures decreases the first part. But the second part is not changed, because parameters like k are fixed. Therefore, the error rate of these algorithms slightly decreases as the memory increases, but the trend is not significant.

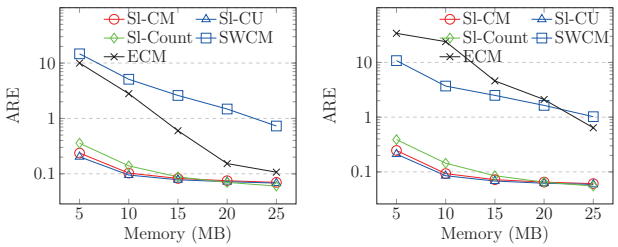
Insertion speed (Figure 6(a)-6(b)): Our results show that the insertion speed of Sl-BF is competitive with FBF and SWBF. When the memory increases, the speed of Sl-BF decreases, because we need to scan more buckets in each time unit in the scanning operation. The speed of Sliding sketches in the following sections has the same trend. The speed of SWBF and FBF also decreases with increasing memory, because these algorithms need to frequently empty old Bloom filters so that they can be used for new slices. Larger Bloom filters make this procedure slower.

8.3 Evaluation on Frequency Query

Parameter setting: We compare 5 approaches: Sl-CM, Sl-CU, Sl-Count, ECM, and SWCM. All the 5 approaches have similar data structures as the CM sketch, namely bucket arrays separated into segments. We set the number of segments $k = 5$ for all the 3 data structures, and vary the array length to control their memory usage. For the Sliding sketches, we set the number of fields in each bucket $d = 3$. For ECM, we set the threshold of triggering exponential histogram merge to 3. For SWCM, we set $\tau = 0.2$ and $\mu = 1.5$. The datasets used in experiments are IP trace dataset and StackOverflow dataset. For each dataset, we set the length of the sliding window $N = 1M$, and read $8M$ items.

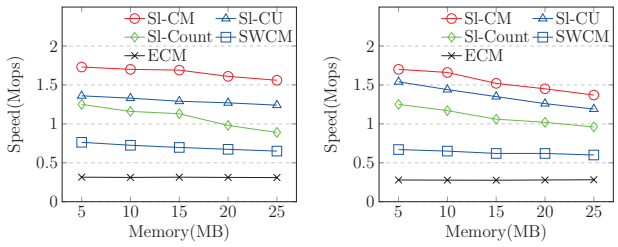
ARE (Figure 7(a)-7(b)): The figures use log-tick due to the large gap between different algorithms. Our results show that the ARE of Sl-CU, which is the Sliding sketch with the best performance, is up to 72 and 161 times lower than SWCM and ECM, respectively. In order to deal with the sliding window model, ECM and SWCM replace counters in the CM sketch with complicated data structures like histograms or deques of slice summarizations. As a result, they need a large quantity of memory to get good performance. Therefore, the gap between prior work and our algorithm is particularly large when the memory usage is small. Among the 3 kinds of Sliding sketches, Sl-CU has the highest accuracy due to its well-designed update strategy. In Sl-CU, each item insertion changes as few counters as possible while keeping the one-side error property, introducing less

error compared with the other 2 Sliding sketches. Details about the implementation of these algorithms are provided in Appendix A of the supplementary materials.



(a) IP trace (b) StackOverflow dataset

Fig. 7. ARE of frequency query.



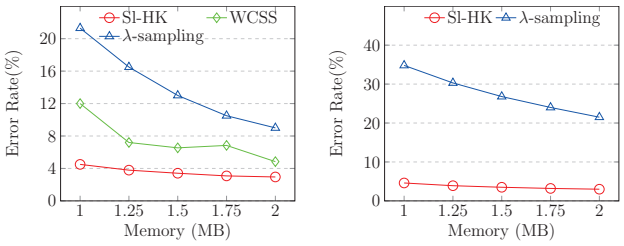
(a) IP trace (b) StackOverflow dataset

Fig. 8. Insertion speed of frequency query.

Insertion speed (Figure 8(a)-8(b)): Our results show that the insertion speed of Sliding sketches is much higher than prior work. Particularly, the fastest one among the three kinds of Sliding Sketches, Sl-CM, is up to 2.54 and 6.07 times faster than SWCM and ECM, respectively. When a new item arrives and is inserted into the data structure, ECM and SWCM may need to merge histograms or summarizations to keep memory in budget, introducing additional cost. Maintaining structures like deques in these algorithms also brings cost. On the contrary, our update operation and scanning operation only operate a small number of counters, which is much simpler and faster. Among the 3 kinds of Sliding sketches, Sl-CM has the highest speed, because its update strategy is the simplest. Sl-CU needs a comparison among the mapped buckets before update them, and Sl-Count needs to carry out additional hash function computations. Thus they are slower than Sl-CM.

8.4 Evaluation on Top-K Query

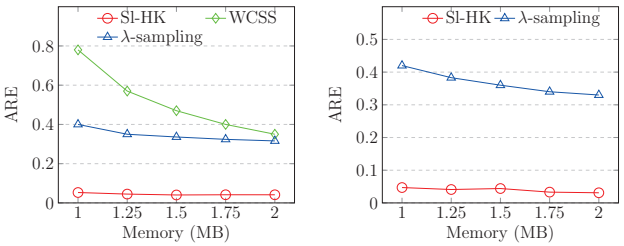
Parameter setting: We compare 3 approaches: Sl-HK, λ-sampling, and WCSS. For Sl-HK, we set the number of segments $k = 5$, and the number of fields in each bucket $d = 4$. We vary the length of the bucket array to change memory usage. For λ-sampling, we set the sample threshold $\lambda = 20$, and vary the number of window counters to change memory usage. For WCSS, we vary the number of blocks to control memory usage. We use IP trace dataset and StackOverflow dataset in this experiment. For each dataset, we set the length of the sliding window $N = 1M$, and read $8M$ items. We find the items with top 3000 frequencies in the sliding window. In experiments of time-based sliding windows in StackOverflow dataset, we only evaluate λ-sampling and Sl-HK, as WCSS does not support time-based sliding windows. We compare error rate, ARE, and insertion speed of the 3 approaches.



(a) IP trace (b) StackOverflow dataset

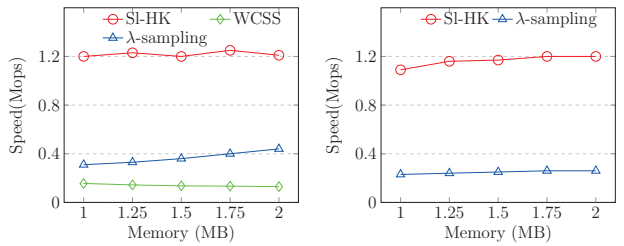
Fig. 9. Error rate of Top-K query.

Error rate (Figure 9(a)-9(b)) and ARE (Figure 10(a)-10(b)): Our results show that our Sl-HK has an error rate below 5% in most experimental settings, better than WCSS and λ-sampling. ARE of Sl-HK also has superiority, which is up to 15 and 9 times lower than WCSS and λ-sampling. Sl-HK has much higher accuracy due to 2 reasons. First, the HeavyKeeper algorithm can get much higher accuracy in Top-K query compared with the prior art. By adapting it to sliding windows, we obtain its superiority. Second, in the Sliding sketch, we can achieve high-quality sliding window approximation with additional counters, which is simple and efficient. On the contrary, these 2 prior algorithms answer sliding window queries based on maintaining sampled arrivals of frequent items. Such sample-based method is memory consuming, because items with high frequencies produce large quantities of samples, and maintaining these sampled arrivals with linked lists or queues introduces additional memory usage. As a result, when the memory is tight, these algorithms are heavily coarsened.



(a) IP trace (b) StackOverflow dataset

Fig. 10. ARE of Top-K query.

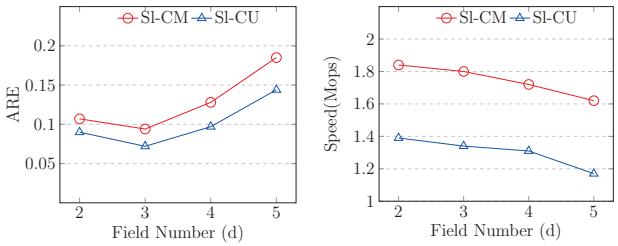


(a) IP trace (b) StackOverflow dataset

Fig. 11. Insertion speed of Top-K query.

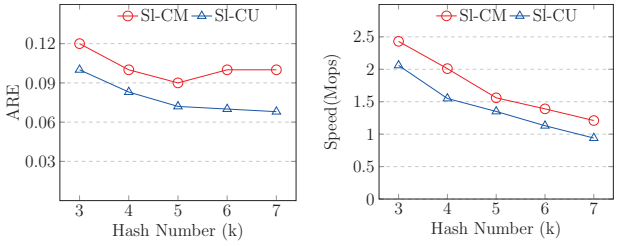
Insertion speed (Figure 11(a)-11(b)): Our results show that the insertion speed of Sl-HK is up to 4 times faster than λ-sampling and 8 times faster than WCSS. WCSS has the

lowest speed as it uses cuckoo hash tables. Cuckoo hash tables need many iterations of kick to insert an item when the memory is tight, resulting into low speed. λ-sampling has higher speed, but it is still not as fast as Sl-HK. Because it still needs complicated operations like querying maps and modifying double lists in the update operation. Moreover, λ-sampling does not have a fixed memory usage, and it needs to frequently clean up records of infrequent items to keep the memory upper-bounded, which also costs time. On the contrary, our Sl-HK only need to operate several counters upon each item arrival. Thus it has the highest speed.



(a) ARE (b) Insertion Speed

Fig. 12. Varying the number of fields in each bucket.



(a) ARE (b) Insertion Speed

Fig. 13. Varying the number of Segments.

8.5 Sensitivity Analysis of the Sliding Sketch

In this section, we evaluate the impact of the number of fields, d, and the number of segments, k on the performance of the Sliding sketch. We use Sl-CM and Sl-CU as case studies. We observe the impact of d and k on ARE and the insertion speed of Sl-CM and Sl-CU. We use Synthetic dataset in this experiment. When varying one parameter, we fix the other. We set $k = 5$ when varying d, and set $d = 3$ when varying k. The length of the sliding window is set $N = 1M$, and the memory usage is $15MB$.

Impact of the number of fields in each bucket (Figure 12(a) and 12(b)): We observe that when increasing the number of fields in each bucket, d, of Sl-CM and Sl-CU, ARE first decreases and then increases, and $d = 3$ brings the best performance. Enlarging d gives a better approximation of the sliding window, but too large d will decrease the number of buckets in the Sliding sketch array and increase hash collision errors. On the other hand, with the increasing of d, the insertion speed keeps decreasing, as we need to scan more buckets in the same time interval.

Impact of the number of segments (Figure 13(a) and 13(b)): We observe that when increasing the number of segments, k, of Sl-CM and Sl-CU, ARE first decreases and then increases, and $k = 5$ brings the best performance. Like d, enlarging k gives a better approximation of the sliding window, but too large k will increase the hash collision error, as each item is mapped to more buckets and has a higher probability to collide with each other. On the other hand, with the increasing of k, the insertion speed keeps decreasing, as we need to operate more mapped buckets for each inserted item.

8.6 Evaluation on Improving Techniques

Accelerations: We use Sl-BF as a case study. We implement 3 versions: the basic CPU version, CPU version accelerated with technique in Section 7.2, and FPGA accelerated version, denoted as CPU-Base, CPU-Acc, and FPGA-Acc. The dataset we use is the IP trace dataset. The window length is $1M$. We use $2MB$ memory and set $d = 2$ and $k = 15$. The FPGA acceleration board we use is Xilinx Alveo U280. We evaluate the insertion speed, query speed and error rate of different versions. The results are shown in Table 3.

From the tables, we can see that the 3 versions have nearly the same error rate, revealing that our accelerating technique does not harm the accuracy. CPU-Acc is about 1.7 times faster than CPU-base due to the decrement of hash computation cost. FPGA-Acc is further 30 times faster because of the parallelism and pipeline of FPGA implementation. In all 3 versions, the query speed is higher than the insertion speed, because both the query operation and the update operation have a time cost of $O(k)$, but the insertion speed also includes the cost of the scanning operation. The gap between the query speed and the update speed is larger in FPGA-Acc. As discussed in Section 7.3, the query operation can be fully pipelined with interval of 1 clock cycle in FPGA implementation, but the update operation can only achieve pipeline interval of 2 cycles due to the conflict of read and write in the bucket array.

TABLE 3
Performance of different versions of Sl-BF

Version	Error Rate(%)	Insertion Speed (Mops)	Query Speed (Mops)
CPU-Base	0.95	0.57	0.87
CPU-Acc	0.95	0.96	1.53
FPGA-Acc	0.92	29.5	70

Accuracy improvements: We use Web page dataset to evaluate the effect of δ-based correction strategy. The window length is set to $1M$. We denote the basic version of Sl-CM as CM-Base, and the version with δ-based correction as CM-Impr, similar for Sl-CU. For both Sl-CM and Sl-CU, we set $k = 5$ and $d = 3$ and use $5MB$ memory. The result is shown in Table 4. We can see that the ARE of Sl-CM can decrease 21% with δ-based correction. The effect on Sl-CU is smaller, as it already has a small hash collision error. But the ARE of Sl-CU still decreases by 12% with δ-based correction.

9 Conclusion

Data stream processing in sliding windows is an important and challenging work. We propose a generic framework

TABLE 4
Effect of δ-based correction

Version	ARE
CM-Base	0.203
CM-Impr	0.16
CU-Base	0.116
CU-Impr	0.102

in this paper, namely the Sliding sketch, which can be applied to most existing sketches and answer various kinds of queries in sliding windows. We use our framework to address three fundamental queries in sliding windows: membership query, frequency query, and Top-K query. Theoretical analysis and experimental results show that our algorithm has much higher accuracy than the prior art.

We believe our framework is suitable for all sketches that use the common sketch model.

REFERENCES

[1] S. H. Oh, J. S. Kang, Y. C. Byun, T. T. Jeong, and W. S. Lee, "Anomaly intrusion detection based on clustering a data stream," in *Acis International Conference on Software Engineering Research, Management and Applications*, pp. 220–227, 2006.

[2] M. A. Faisal, Z. Aung, J. R. Williams, and A. Sanchez, "Securing advanced metering infrastructure using intrusion detection system with data stream mining," in *Pacific Asia Conference on Intelligence and Security Informatics*, pp. 96–111, 2012.

[3] B. Ball, M. Flood, H. V. Jagadish, J. Langsam, L. Raschid, and P. Wiriyathammabhum, "A flexible and extensible contract aggregation framework (caf) for financial data stream analytics," pp. 1–6, 2014.

[4] L. G. Gyurkó, T. Lyons, M. Kontkowski, and J. Field, "Extracting information from the signature of a financial data stream," *Quantitative Finance*, 2013.

[5] R. Hu, "Stability analysis of wireless sensor network service via data stream methods," *Applied Mathematics & Information Sciences*, vol. 6, no. 3, pp. 793–798, 2012.

[6] C. M. S. Figueiredo, C. M. S. Figueiredo, E. F. Nakamura, L. S. Buriol, A. A. F. Loureiro, A. O. Fernandes, and C. J. N. J. Coelho, "Data stream based algorithms for wireless sensor network applications," in *International Conference on Advanced Information NETWORKING and Applications*, pp. 869–876, 2007.

[7] B. H. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Communications of the ACM*, vol. 13, no. 7, pp. 422–426, 1970.

[8] G. Cormode and S. Muthukrishnan, "An improved data stream summary: the count-min sketch and its applications," *Journal of Algorithms*, vol. 55, no. 1, pp. 58–75, 2005.

[9] C. Estan and G. Varghese, "New directions in traffic measurement and accounting," *ACM SIGMCOMM CCR*, vol. 32, no. 4, 2002.

[10] M. Datar, A. Gionis, P. Indyk, and R. Motwani, "Maintaining stream statistics over sliding windows," *Siam Journal on Computing*, vol. 31, no. 6, pp. 1794–1813, 2002.

[11] R. Subramanyam, I. Gupta, L. M. Leslie, and W. Wang, "Idempotent distributed counters using a forgetful bloom filter," *Cluster Computing*, vol. 19, no. 2, pp. 879–892, 2016.

[12] O. Papapetrou, M. Garofalakis, and A. Deligiannakis, "Sketch-based querying of distributed sliding-window data streams," *Proceedings of the VLDB Endowment*, vol. 5, no. 10, pp. 992–1003, 2012.

[13] M. Charikar, K. Chen, and M. Farach-Colton, "Finding frequent items in data streams," in *Automata, Languages and Programming*, Springer, 2002.

[14] J. Gong, T. Yang, H. Zhang, H. Li, S. Uhlig, S. Chen, L. Uden, and X. Li, "Heavykeeper: An accurate algorithm for finding top-k elephant flows," in *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, (Boston, MA), pp. 909–921, USENIX Association, 2018.

[15] ""source code of sliding sketches and other sketches"." https://github.com/sliding-sketch/Sliding-Sketch.

[16] D. Nelson, "The bloomier filter: An efficient data structure for static support lookup tables," *Proc Symposium on Discrete Algorithms*, 2004.

[17] J. Aguilar-Saborit, P. Trancoso, V. Muntes-Mulero, and J. L. Larriba-Pey, "Dynamic count filters," *Acm Sigmod Record*, vol. 35, no. 1, pp. 26–32, 2006.

[18] F. Hao, M. Kodialam, T. V. Lakshman, and H. Song, "Fast multiset membership testing using combinatorial bloom filters," in *INFOCOM*, pp. 513–521, 2009.

[19] T. Yang, A. X. Liu, M. Shahzad, Y. Zhong, Q. Fu, Z. Li, G. Xie, and X. Li, "A shifting bloom filter framework for set queries," *Proceedings of the Vldb Endowment*, vol. 9, no. 5, pp. 408–419, 2016.

[20] T. Yang, Y. Zhou, H. Jin, S. Chen, and X. Li, "Pyramid sketch: a sketch framework for frequency estimation of data streams," *Proceedings of the Vldb Endowment*, vol. 10, no. 11, 2017.

[21] P. Roy, A. Khan, and G. Alonso, "Augmented sketch: Faster and more accurate stream processing," in *International Conference on Management of Data*, pp. 1449–1463, 2016.

[22] J. Chen and Q. Zhang, "Bias-aware sketches," *Proceedings of the VLDB Endowment*, vol. 10, no. 9, pp. 961–972, 2017.

[23] 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*.

[24] 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*, SIGMOD '18, (New York, NY, USA), pp. 741–756, ACM, 2018.

[25] E. D. Demaine, A. López-Ortiz, and J. I. Munro, "Frequency estimation of internet packet streams with limited space," in *European Symposium on Algorithms*, pp. 348–360, Springer, 2002.

[26] G. S. Manku and R. Motwani, "Approximate frequency counts over data streams," in *VLDB'02: Proceedings of the 28th International Conference on Very Large Databases*, pp. 346–357, Elsevier, 2002.

[27] 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*, pp. 398–412, Springer, 2005.

[28] D. Ting, "Data sketches for disaggregated subset sum and frequent item estimation," 2017.

[29] F. Chang, W. C. Feng, and K. Li, "Approximate caches for packet classification," in *Joint Conference of the IEEE Computer and Communications Societies*, pp. 2196–2207 vol.4, 2004.

[30] Yoon, "Aging bloom filter with two active buffers for dynamic sets," *IEEE Transactions on Knowledge & Data Engineering*, vol. 22, no. 1, pp. 134–138, 2009.

[31] N. Rivetti, Y. Busnel, and A. Mostefaoui, *Efficiently Summarizing Distributed Data Streams over Sliding Windows*. PhD thesis, LINA-University of Nantes; Centre de Recherche en Économie et Statistique; Inria Rennes Bretagne Atlantique, 2015.

[32] H. L. Chan, T. W. Lam, L. K. Lee, and H. F. Ting, *Continuous Monitoring of Distributed Data Streams over a Time-Based Sliding Window*. 2009.

[33] G. Cormode and K. Yi, "Tracking distributed aggregates over time-based sliding windows," in *ACM Sigact-Sigops Symposium on Principles of Distributed Computing*, pp. 213–214, 2011.

[34] B. B. Ran, G. Einziger, R. Friedman, and Y. Kassner, "Heavy hitters in streams and sliding windows," in *IEEE INFOCOM 2016 - the IEEE International Conference on Computer Communications*, pp. 1–9, 2016.

[35] L. K. Lee and H. F. Ting, "A simpler and more efficient deterministic scheme for finding frequent items over sliding windows," in *ACM Sigmod-Sigact-Sigart Symposium on Principles of Database Systems*, pp. 290–297, 2006.

[36] Hung, Y. S. Regant, Lee, Lap-Kei, Ting, and H.F, "Finding frequent items over sliding windows with constant update time," *Information Processing Letters*, vol. 110, no. 7, pp. 257–260, 2010.

[37] A. Arasu and G. S. Manku, "Approximate counts and quantiles over sliding windows," in *ACM Sigmod-Sigact-Sigart Symposium on Principles of Database Systems*, pp. 286–296, 2004.

[38] P. L'Ecuyer, "Tables of linear congruential generators of different sizes and good lattice structure," *Mathematics of Computation*, vol. 68, no. 225, pp. 249–260, 1999.

[39] A. Rousskov and D. Wessels, "High-performance benchmarking with web polygraph," *Software: Practice and Experience*, vol. 34, no. 2, pp. 187–211, 2004.

[40] D. M. Powers, "Applications and explanations of Zipf's law," in *Proc. EMNLP-CoNLL*, Association for Computational Linguistics, 1998.

[41] M. Mitzenmacher, "Network applications of bloom filters: A survey," *Internet Mathematics*, vol. 1, no. 4, pp. 485–509, 2004.

[42] W. C. Feng, "Blue : A new class of active queue management algorithms," *Tech Rep*, vol. 18, no. 3, pp. 298–312, 1999.

Xiangyang Gou is a Ph.D. student in the School of Computer Science of Peking University. His research interests include data structures and algorithms in data streams and graph streams.

Yinda Zhang has graduated with a bachelor's degree from Peking University, and graduated with a master's degree from the University of Chicago. His research interests include network measurements and streaming algorithms.

Zhoujing Hu is currently an undergraduate student of Peking University majoring in Computer Science. His research interests include data sketches, and data stream processing systems.

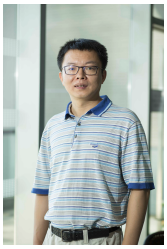
Long He has graduated with a bachelor's degree from Peking University. His research interests include Network Big Data and Network Measurement.

Ke Wang has graduated with a bachelor's degree from Peking University. He is currently a Ph.D. student in Yale University. His research interests lie in the fields of computer system, network and database, including efficient memory architectures, network telemetry, programmable network and database engine.

Xilai Liu has graduated with a bachelor's degree from Peking University. He is currently a Ph.D. student in the Institute of Computing Technology, Chinese Academy of Sciences (CAS), and the University of Chinese Academy of Sciences (UCAS). His research interests include data sketches, in-network computing, and data stream processing systems.

Tong Yang received the Ph.D. degree in computer science from Tsinghua University in 2013. He visited the Institute of Computing Technology, Chinese Academy of Sciences (CAS). He is currently an Associate Professor with the Department of Computer Science, Peking University. His research interests include network measurements, sketches, IP lookups, Bloom filters, sketches, and KV stores.

Yi Wang is a Research Professor in the Sustech Institute of Future Networks, Southern University of Science and Technology. He received the Ph.D. degree in Computer Science and Technology from Tsinghua University in July 2013. His research interests include Future Network Architectures, Information Centric Networking, Software-defined Networks, and the design and implementation of high-performance network devices. He has published more than 80 peer-reviewed papers in SIGCOMM, NSDI, MOBICOM, INFOCOM, ToN, etc..



Bin Cui is a professor in the School of Computer Science and Director of Institute of Network Computing and Information Systems, at Peking University. His research interests include database system architectures, query and index techniques, big data management and mining. He is a senior member of IEEE, member of ACM and distinguished member of CCF.

SUPPLEMENTARY MATERIALS

APPENDIX A
SLIDING SKETCH APPLICATIONS

In this section, we apply the Sliding sketch technique to the Bloom filter, the CM sketch, the Count sketch, the CU sketch, and the HeavyKeeper as examples to show how it works explicitly. In these examples, we use d fields in each bucket, where d is a parameter that can be adjusted.

A.1 Apply to the Bloom filter

A.1.1 Standard Bloom filter

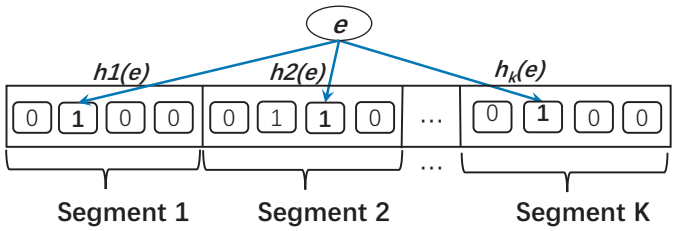


Fig. 14. Structure of the partition Bloom filter

There are 2 versions of Bloom filters [7], the partition version and the standard version. They have similar error rate [41], and the partition Bloom filter is more popular because it is suitable for parallel implementations [42]. In this paper, we apply the Sliding sketch technique to the partition Bloom filter. A partition Bloom filter is composed of an array of m bits, separated into k equal-sized segments, and each segment is associated with a hash function, as shown in Figure 14.

Update operation: When inserting an item e, we map e into k bits with the k hash functions, one in each segment, and set these mapped bits to 1.

Query operation: When querying an item e, we find the k mapped bits of e with the hash functions. If any of them is 0, we report false. Otherwise we report true.

A.1.2 Sliding Bloom filter

In the Sliding Bloom filter, A is a bit array with m buckets, separated into k equal-sized segments. Each bucket $A[i]$ contains d bits $\{A[i][j]|0 \leqslant j < d\}$. Initially, all the bits are set to 0.

Update operation: When an item e arrives, we find the k mapped buckets $\{A[h_i]|0 \leqslant i < k\}$with the k hash functions, and in each mapped bucket, we set the first bit $A[h_i][0]$ to 1.

Scanning operation: In the scanning operation, a scanning pointer goes through the buckets one by one in A repeatedly. The scanning pointer goes through $\frac{(d-1)\times m}{N}$ buckets in each time unit, where N is the length of the sliding window. When the scanning pointer arrives at a bucket $A[i]$, we set $A[i][j] = A[i][j-1](1 \leqslant j \leqslant d-1)$ and $A[i][0] = 0$.

Query operation: When querying an item e, we find the k mapped buckets $\{A[h_i]|0 \leqslant i < k\}$ with the k hash functions, and compute k values $\{Sum(A[h_i]) = \sum_{j=0}^{d-1} A[h_i][j]|0 \leqslant i < k\}$. If any of them is 0, we return false. Otherwise, we return true.

A.2 Apply to the CM sketch

A.2.1 Standard CM sketch

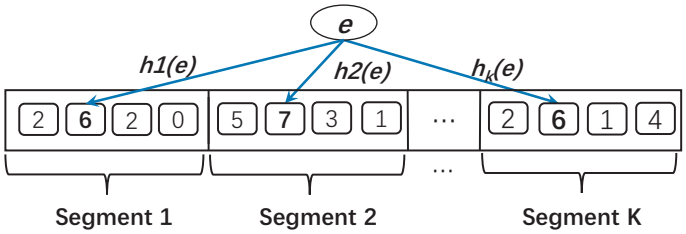


Fig. 15. Structure of the CM sketch

The CM sketch [8] is composed of a counter array with m counters which are separated into k equal-sized segments. These k segments are associated with k hash functions $\{h_i(\cdot)|0 \leqslant i < k\}$, as shown in Figure 15. All the counters are set to 0 initially.

Update operation: When inserting an item e, we map it to k counters with the k hash functions, one in each segment. We increase all the k mapped counters by 1.

Query operation: When querying an item e, we find the k mapped counters, and report the minimum one as the frequency.

A.2.2 Sliding CM sketch

In the Sliding CM sketch, A is a counter array with m buckets, separated into k equal-sized segments. Each bucket $A[i]$ contains d counters $\{A[i][j]|0 \leqslant j < d\}$. Initially, all the counters are set to 0.

Update operation: When an item e arrives in the data stream, we map it to k mapped buckets $\{A[h_i]|0 \leqslant i < k\}$ with k hash functions, one in each segment. In each mapped bucket, we increase the first counter $A[i][0]$ by 1.

Scanning operation: The scanning operation is the same as the Sliding Bloom filter.

Query operation: When querying an item e, we find the k mapped buckets $\{A[h_i]|0 \leqslant i < k\}$, and compute k values $\{Sum(A[h_i]) = \sum_{j=0}^{d-1} A[h_i][j]|0 \leqslant i < k\}$. We report the minimum one as the frequency.

A.3 Apply to the Count sketch

A.3.1 Standard Count Sketch

The Count sketch [13] has the same data structure as the CM sketch, but it has another set of k hash functions $\{g_i(\cdot)|0 \leqslant i < k\}$ besides $\{h_i(\cdot)|0 \leqslant i < k\}$. Each $g_i(\cdot)$ maps the input to value range $\{-1, 1\}$.

Update operation: When inserting an item e, we use $\{h_i(\cdot)|1 \leqslant i \leqslant k\}$ to map the item into k mapped counters just like the CM sketch, but instead of increasing the

counters by 1, we increase the i_{th} mapped counter by $g_i(e)$ ($g_i(e) = -1$ or 1).
Query operation: When querying an item e, we find the k mapped counters, and report the median value of the k value $\{g_i(e) \times C_i | 1 \leqslant i \leqslant k\}$ where C_i is the i_{th} mapped counter.

A.3.2 Sliding Count Sketch

The Sliding Count sketch has the same data structure as the Sliding CM sketch. The update operation, the scanning operation, and the query operation are detailed as follows.
Update operation: When an item e arrives, we find the k mapped buckets $\{A[h_i]|0 \leqslant i < k\}$. Then we compute another set of k hash functions $\{g_i(e)|0 \leqslant i < k\}$ which map item e to the range $\{-1, 1\}$. In each mapped bucket, we update the first counter $A[h_i][0]$ as $A[h_i][0] = A[h_i][0] + g_i(e)$.
Scanning operation: In the scanning operation, a scanning pointer goes through the buckets in A repeatedly. The scanning pointer goes through $\frac{m \times (2d-1)}{2N}$ buckets in each time unit, where N is the length of the sliding window. When it arrives at a bucket $A[[i]$, we set $A[i][j] = A[i][j-1] (1 \leqslant j \leqslant d-1)$ and set $A[i][0] = 0$.
Query operation: When querying an item e, we find the k mapped buckets $\{A[h_i]|0 \leqslant i < k\}$, and compute k values $\{Sum(A[h_i]) = g_i(e) \times \sum_{j=0}^{d-1} A[h_i][j] | 0 \leqslant i < k\}$. Then we return the median value of them.

A.4 Apply to the CU sketch

A.4.1 Standard CU sketch

The CU sketch [9] has the same data structure as the CM sketch.
Update operation: When inserting an item e, it only increases the minimum mapped counter by 1. As the mapped counters are no smaller than the accurate frequency, we only need to increase the minimum one to keep such one-side error property.
Query operation: When querying an item e, the CU sketch reports the minimum value of the k mapped counters as the frequency.

The CU sketch is more accurate than the CM sketch when using the same parameters. The CU sketch also only has over-estimation error.

A.4.2 Sliding CU sketch

The Sliding CU sketch has the same data structure as the Sliding CM sketch. The update operation, the scanning operation, and the query operation are detailed as follows.
Update operation: When an item e arrives, we find the k mapped buckets$\{A[h_i]|0 \leqslant i < k\}$ with k hash functions. Then we go through these k mapped buckets by the descending order of jet lag, where jet lag can be computed according to equation 1. For each mapped bucket $A[h_i]$, if there is another mapped bucket $A[h_j]$ with larger jet lag but $A[h_j][0] < A[h_i][0]$, we do not increase $A[h_i][0]$. Field $A[h_i][0]$ records a smaller slice than $A[h_j][0]$. If it still has a larger value, it must suffer from hash collisions, and we do not need to increase it. Otherwise we increase $A[h_i][0]$ by 1.

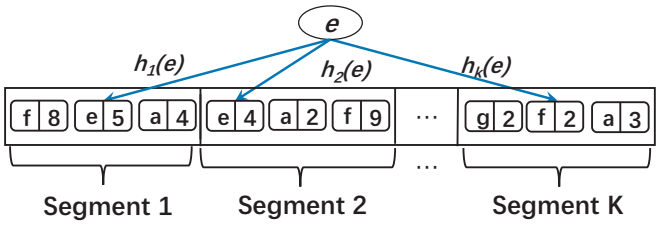


Fig. 16. Structure of the HeavyKeeper

Scanning operation: The scanning operation is the same as the Sliding Bloom filter.
Query operation: When querying an item e, we find the k mapped buckets $\{A[h_i]|0 \leqslant i < k\}$, and compute k values $\{Sum(A[h_i]) = \sum_{j=0}^{d-1} A[h_i][j] | 0 \leqslant i < k\}$. We report the minimum one as the frequency.

A.5 Apply to the HeavyKeeper

A.5.1 Standard HeavyKeeper

The HeavyKeeper [14] is composed of an array of buckets with k segments and k hash functions, as shown in Figure 16. Each bucket is composed of an ID and a counter. All IDs and the counters are set to 0 initially.
Update operation: When inserting an item e, we map the item to k buckets with the k hash functions, one in each segment. We check these k mapped buckets. There are 3 kinds of situations.

1) If the ID is 0 in a mapped bucket, which means this bucket is empty, we set the ID to e, and set the counter to 1.

2) If the ID is equal to e in a mapped bucket, we increase the counter by 1.

3) If the ID is not e or 0 in a mapped bucket, it means that this bucket is occupied by another item. Then we decrease the counter in it by 1 with a probability b^{-C} where C is the value of the counter and b is a constant suggested to be 1.08. If the counter is 0 after decrement, we replace the ID with e, and set the counter to 1.

Query the frequency of an item: When querying the frequency of an item e, we check the counters in the mapped buckets which contain ID equal to e, and report the maximum one as the frequency.
Find the Top-K items: When the we need to find the Top-K items, we scan the array and get the frequency of all stored items, and find the ones with top K largest frequencies.

In the HeavyKeeper, items with high frequencies are hardly decreased, as b^{-C} is very small when C is large. While items with small frequencies can hardly stay in the array. In this way, it achieves high accuracy in Top-K query. It only has under-estimation error for the frequencies of items.

A.5.2 Sliding HeavyKeeper

In the Sliding HeavyKeeper, A is an array with m buckets which are divided into k equal-sized segments. Each bucket $A[i]$ contains an ID and d counters $\{A[i][j]|0 \leqslant j < d\}$. Initially, all IDs and counters are set to 0. The update

operation, the scanning operation, and the query operation are detailed as follows.

Update operation: When an item e arrives in the data stream, we find k mapped buckets $\{A[h_i]|0 \leqslant i < k\}$ with k hash functions. Then we check these k mapped buckets. There are 3 kinds of situations:

1) If the ID is 0 in a mapped bucket $A[h_i]$, we set the ID to e and set the first counter $A[h_i][0] = 1$.

2) If the ID is equal to e in a mapped bucket $A[h_i]$, we increase the first counter $A[h_i][0]$ by 1.

3) If the ID is not 0 or e in a mapped bucket $A[h_i]$, the update procedure is as follows. First, we compute a sum $Sum(A[h_i]) = \sum_{j=0}^{d-1} A[h_i][j]$. Second, we find the smallest j where $A[h_i][j] > 0$ and decrease it by 1 with probability $b^{Sum(A[h_i])}$, where b is suggested to be 1.08. Third, if after the decrement $Sum(A[h_i]) = 0$, we set the ID to e and $A[h_i][0] = 1$, $A[h_i][j] = 0$ $(1 \leqslant j \leqslant d-1)$.

Scanning Operation: In the scanning operation, a scanning pointer goes through the buckets in A repeatedly. The scanning pointer goes through $\frac{m \times d}{N}$ buckets in each time unit, where N is the length of the sliding window. When it arrives at a bucket $A[[i]$, we set $A[i][j] = A[i][j-1] (1 \leqslant j \leqslant d-1)$ and set $A[i][0] = 0$.

Query Operation: We check all the items in the Sliding HeavyKeeper. For each item e, we find the k mapped buckets $\{A[h_i]|0 \leqslant i < k\}$. Then we check the ID in these k mapped buckets. If in a mapped bucket $A[h_i]$ the ID is equal to e, we compute value $Sum(A[h_i]) = \sum_{j=0}^{d-1} A[h_i][j]$. We report the maximum one among these computed values as the frequency of e. At last we find the items with top K largest frequencies and report them.

APPENDIX B
ANALYSIS OF ACCURACY

The accuracy of the Sliding sketch is influenced by the base sketch. We analyze the accuracy of the Sliding Bloom filter, the Sliding CM sketch and the Sliding HeavyKeeper as examples.

B.1 Accuracy of the Sliding Bloom Filter

For simplicity, in the following analysis, we use N_i to denote $(1 + \frac{i}{(d-1)\times k})N$, where N is the length of the sliding window. We denote current time with T. We use n_i to represent the number of distinct items in $W^T_{T-N_i}$.

Theorem 4. *The Sliding Bloom filter only has false positives and no false negatives.*

Proof. In each mapped bucket $A[h_i]$ of item e in the Sliding Bloom filter, $Sum(A[h_i]) = \sum_{j=0}^{d-1} A[h_i][j]$ records the presence of items mapped to $A[h_i]$ in $W^T_{T-(1+\frac{\delta}{d-1})N}$. This slice is a superset of the sliding window. Combining this property with the one-side error property of the Bloom filter, we know that if an item e shows up in sliding window, it must show up in $W^T_{T-(1+\frac{\delta}{d-1})N}$, and $Sum(A[h_i]) > 0$. When in all the k mapped buckets $Sum(A[h_i]) > 0$, we will report true. However, when e is not in the sliding window, we may still report true because we record a larger slice, and hash collisions may happen. □

Theorem 5. *In the Sliding Bloom filter, suppose we query an item e which is not in the sliding window. We use Pr_i to represent probability that we correctly get a negative report when e does not present in $W^T_{T-N_i}, (2 \leqslant i \leqslant k)$. Then we have*

$$\begin{aligned} Pr_i &\geqslant 1 - (1 - (1 - \frac{k}{m})^{n_i})^{i-1} \\ &\approx 1 - (1 - e^{-\frac{n_i k}{m}})^{i-1}, \quad (2 \leqslant i \leqslant k-1) \end{aligned} \tag{10}$$

and

$$\begin{aligned} Pr_k &\geqslant 1 - (1 - (1 - \frac{k}{m})^{n_k})^{k} \\ &\approx 1 - (1 - e^{-\frac{n_k k}{m}})^{k} \end{aligned} \tag{11}$$

Proof. In [7] the authors have proved that given a set with n distinct items, when querying for an item e which is not in the set with the Bloom filter, the probability that there is at least one bit correct among j mapped bits is

$$\begin{aligned} Pr &= 1 - (1 - (1 - \frac{k}{m})^{n})^{j} \\ &\approx 1 - (1 - e^{-\frac{nk}{m}})^{j} \end{aligned} \tag{12}$$

m is the length of the array and k is the number of segments. This is the probability that we get a correct answer with the j mapped bits, because as long as one bit is 0, we will give a negative report. In the following part we analyze the accuracy of the Sliding Bloom filter with this result.

In the Sliding Bloom filter, when querying an item e which is not in the sliding window, the value ranges of δ in the k mapped buckets are shown in Theorem 1, 2 and 3. We use Pr_i to represent probability that we get a negative report when e does not present in $W^T_{T-N_i}, (2 \leqslant i \leqslant k)$, and use n_i to represent the number of items in this slice. For $2 \leqslant i \leqslant (k-1)$, there are at least $i-1$ buckets where $\delta \leqslant \frac{i}{k}$, as shown in Theorem 2. These buckets record slices shorter than $(1 + \frac{i}{(d-1)\times k})N$. Therefore we have

$$\begin{aligned} Pr_i &\geqslant 1 - (1 - (1 - \frac{k}{m})^{n_i})^{i-1} \\ &\approx 1 - (1 - e^{-\frac{n_i k}{m}})^{i-1}, \quad (2 \leqslant i \leqslant k-1) \end{aligned} \tag{13}$$

There are k buckets which record slices shorter than $(1 + \frac{1}{(d-1)})N$, therefore we have

$$\begin{aligned} Pr_k &\geqslant 1 - (1 - (1 - \frac{k}{m})^{n_k})^{k} \\ &\approx 1 - (1 - e^{-\frac{n_k k}{m}})^{k} \end{aligned} \tag{14}$$

□

B.2 Accuracy of the Sliding CM sketch

In the following analysis, we use N_i to denote $(1 + \frac{i}{(d-1)\times k})N$, where N is the length of the sliding window. We denote current time with T. We use $|W^T_{T-N_i}|$ to represent the number of items with duplication in $W^T_{T-N_i}$, and use f_i to denote the frequency of e in $W^T_{T-N_i}$.

Theorem 6. *The Sliding CM sketch only has over-estimation error and no under-estimation error.*

Proof. In each mapped bucket $A[h_i]$ of item e in the Sliding CM sketch, $Sum(A[h_i]) = \sum_{j=0}^{d-1} A[h_i][j]$ records the sum of frequencies of items mapped to $A[h_i]$ in $W^T_{T-(1+\frac{\delta}{d-1})N}$. This slice is a superset of the sliding window. Combining this property with the one-side error property of the CM sketch, we know that if an item e has frequency f in sliding window, it must shows up no less than f times in the data stream in $W^T_{T-(1+\frac{\delta}{d-1})N}$, and $Sum(A[h_i]) \geqslant f$. When in all the k mapped buckets $Sum(A[h_i]) \geqslant f$, the reported value $\hat{f}$ will be no smaller than f. □

Theorem 7. *In the Sliding CM sketch, suppose the reported frequency is $\hat{f}$. We use Pr_i to represent probability that $\hat{f} \leqslant f_i + \epsilon|W^T_{T-N_i}|$, $(1 \leqslant i \leqslant k)$. $\epsilon = \frac{ke}{m}$. Then we have*

$$Pr_i \geqslant 1 - e^{-i+1}, \quad (2 \leqslant i \leqslant k-1) \tag{15}$$

and

$$Pr_k \geqslant 1 - e^{-k} \tag{16}$$

Proof. In [8], the authors have proved that when querying the frequency of an item e in set s with the CM sketch, we have

$$Pr(\hat{f} \leqslant f + \epsilon|s|) \geqslant 1 - e^{-k} \tag{17}$$

f is the accurate frequency. $\hat{f}$ is the query result, *i.e.*, the minimum value among the k mapped counters. $\epsilon = \frac{ke}{m}$, where m is the length of the array, and k is the number of segments. $|s|$ is the number of items with duplication in s.

When we only use j mapped counters and return the minimum value among them as $\hat{f}$, we have

$$Pr(\hat{f} \leqslant f + \epsilon|s|) \geqslant 1 - e^{-j} \tag{18}$$

In the following part we analyze the accuracy of the Sliding CM sketch with these results.

For $2 \leqslant i \leqslant (k-1)$, there are at least $i-1$ buckets where $\delta \leqslant \frac{i}{k}$, as shown in Theorem 2. These buckets record recent slices no longer than $N_i = (1 + \frac{i}{(d-1)\times k})N$. Therefore we have

$$\begin{aligned} Pr_i &= Pr(\hat{f} \leqslant f_i + \epsilon|W^T_{T-N_i}|) \\ &\geqslant 1 - e^{-i+1}, \quad (2 \leqslant i \leqslant k-1) \end{aligned} \tag{19}$$

There are k buckets which record slices no longer than $N_k = (1 + \frac{1}{(d-1)})N$. Therefore we have

$$Pr_k \geqslant 1 - e^{-k} \tag{20}$$

□

B.3 Accuracy of the Sliding HeavyKeeper

In the following analysis, we use N_i to denote $(1 - \frac{i}{d\times k})N$, where N is the length of the sliding window. We denote current time is T. We use $|W^T_{T-N_i}|$ to represent the number of items with duplication in $W^T_{T-N_i}$. f_i denotes the frequency of e in $W^T_{T-N_i}$

Theorem 8. *For the frequency of any item, The Sliding HeavyKeeper only has under-estimation error and no over-estimation error.*

In each mapped bucket $A[h_i]$ of item e in the Sliding HeavyKeeper, with the scanning speed recommended in Section 7.1, $Sum(A[h_i]) = \sum_{j=0}^{d-1} A[h_i][j]$ records the frequencies of the item stored in $A[h_i]$ in $W^T_{T-(1-\frac{1-\delta}{d})N}$. This slice is a subset of the sliding window. On the other hand, the HeavyKeeper only has under-estimation for the frequencies, because when an item collides with other items, its frequency may be decreased. Therefore, we know that if an item e has frequency f in sliding window, it must shows up no more than f times in $W^T_{T-(1-\frac{1-\delta}{d})N}$, and because of hash collisions, the recorded frequency may be smaller.

Theorem 9. *In the Sliding HeavyKeeper, suppose the reported frequency of a Top-K item if $\hat{f}$. If we use Pr_i to represent probability that $\hat{f} \geqslant f_i - \epsilon|W^T_{T-N_i}|$ where ϵ is any small positive number, we have*

$$Pr_i \geqslant 1 - (\frac{k}{\epsilon m f(b-1)})^{i-1}, \quad (2 \leqslant i \leqslant k-1) \tag{21}$$

$$Pr_k \geqslant 1 - (\frac{k}{\epsilon m f(b-1)})^{k} \tag{22}$$

b is the constant for probabilistic decay. m is the length of the array, and k is the number of segments.

In the Sliding HeavyKeeper, we concentrate on the items with high frequencies. Specifically, in Top-K query, we only care about the items with top K frequencies. The mapped buckets of a top K item are all occupied by it in most cases. Therefore, in the following analysis, we assume that when querying an item e, the k mapped buckets of e are all occupied by it. In [14], the authors have proved that when querying an item e in set s with the HeavyKeeper, in each mapped bucket occupied by e we have:

$$Pr(\hat{f} \leqslant f - \epsilon|s|) \leqslant \frac{k}{\epsilon m f(b-1)} \tag{23}$$

where f is the accurate frequency of e. $\hat{f}$ is the frequency stored in this mapped bucket. ϵ is any small positive number. m is the length of the array, and k is the number of segments. $|s|$ is the number of items with duplication in s.

When we use j mapped buckets and return the maximum value among them as $\hat{f}$, we have $\hat{f} \geqslant f - \epsilon|s|$ unless all the j mapped buckets have counter smaller than $f - \epsilon|s|$. Therefore:

$$Pr(\hat{f} \geqslant f - \epsilon|s|) \geqslant 1 - (\frac{k}{\epsilon m f(b-1)})^{j} \tag{24}$$

In this following part we analyze the accuracy of the Sliding HeavyKeeper with these results.

For $2 \leqslant i \leqslant (k-1)$, there are at least $i-1$ buckets where $\delta \geqslant 1 - \frac{i}{k}$, as shown in Theorem 3. These buckets record slices which are superset of $W^T_{T-N_i}$, and subset of the sliding window. Therefore we have

$$\begin{aligned} Pr_i &= Pr(\hat{f} \geqslant f_i - \epsilon|W^T_{T-N_i}|) \\ &\geqslant 1 - (\frac{k}{\epsilon m f(b-1)})^{i-1}, \quad (2 \leqslant i \leqslant k-1) \end{aligned} \tag{25}$$

There are k buckets recording slices which are superset of $W^T_{T-N_k}$. Therefore we have

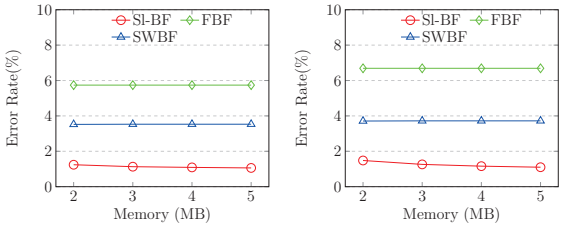
$$Pr_k \geqslant 1 - (\frac{k}{\epsilon m f(b-1)})^{k} \tag{26}$$

APPENDIX C
SUPPLEMENTARY EXPERIMENTS

In this section, we show the experimental results of more datasets as supplement to Section 8.

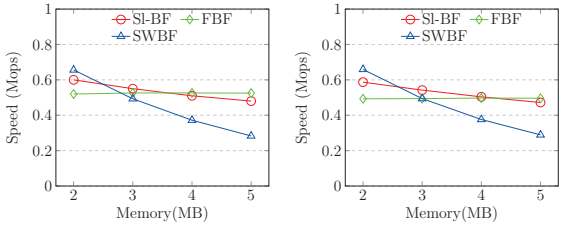
C.1 Evaluation on Membership Query

We show the error rate and speed of Sl-BF, FBF and SWBF in Web Page dataset and Synthetic dataset here. The window length is set to $1M$ and we read $8M$ items. The parameter settings are the same as Section 8.2.



(a) Web Page. (b) Synthetic dataset.

Fig. 17. Error rate of membership query.

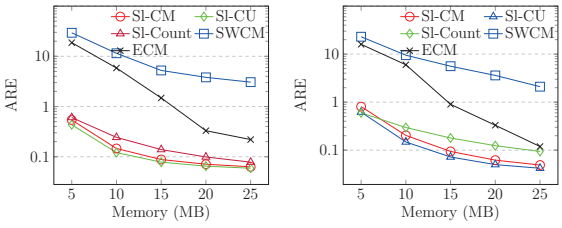


(a) Web page. (b) Synthetic dataset.

Fig. 18. Insertion speed of membership query.

C.2 Evaluation on Frequency Query

We show the ARE and speed of Sl-CM, Sl-CU, Sl-Count, ECM and SWCM in Web Page dataset and Synthetic dataset here. The window length is set to $1M$ and we read $8M$ items. The parameter settings are the same as Section 8.3.

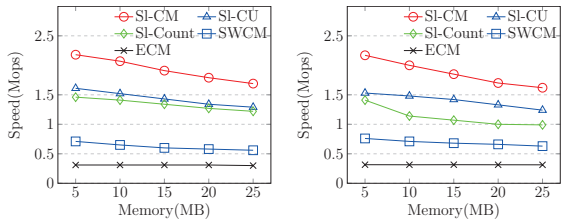


(a) Web Page (b) Synthetic dataset

Fig. 19. ARE of frequency query.

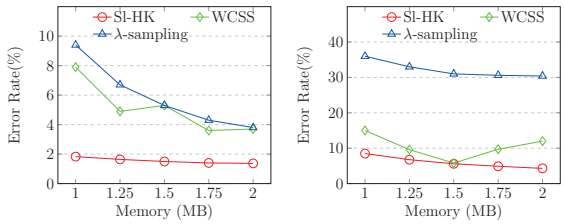
C.3 Evaluation on Top-K Query

We show the error rate, ARE and speed of Sl-HK, λ-sampling and WCSS in Web Page dataset and Synthetic dataset here. The window length is set to $1M$ and we read $8M$ items. The parameter settings are the same as Section 8.4.



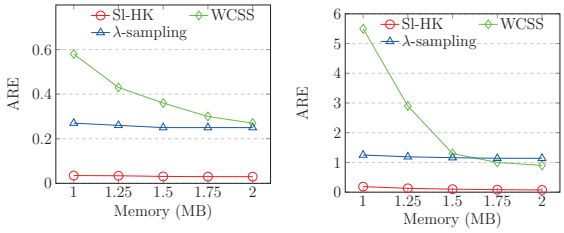
(a) Web page (b) Synthetic dataset

Fig. 20. Insertion speed of frequency query.



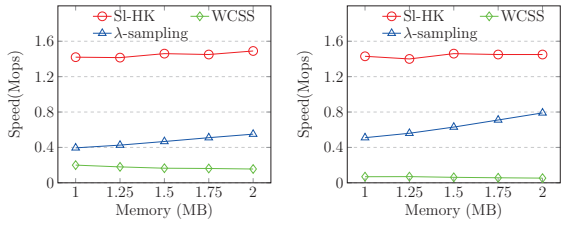
(a) Web page (b) Synthetic dataset

Fig. 21. Error rate of Top-K query.



(a) Web page (b) Synthetic dataset

Fig. 22. ARE of Top-K query.



(a) Web page (b) Synthetic dataset

Fig. 23. Insertion speed of Top-K query.

C.4 Evaluation on Acceleration Techniques

In this section we use the Sliding CM sketch as another example to show the effect of acceleration techniques. We use $5MB$ memory and set $d = 3$ and $k = 5$. The dataset and other settings are the same as Section 8.6.

TABLE 5
Performance of Different Versions of Sl-CM

Version	ARE	Insertion Speed (Mops)	Query Speed (Mops)
CPU-Base	0.24	1.7	1.36
CPU-Acc	0.24	2.4	2.37
FPGA-Acc	0.27	32	64