Sliding Sketches: A Framework using Time Zones for Data Stream Processing in Sliding Windows

ABSTRACT

Data stream processing has become a hot issue in recent years due to the arrival of big data era. There are three fundamental stream processing tasks: membership query, frequency query and heavy hitter 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 Sliding sketches, 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 after using our framework, the accuracy of existing sketches that do not support sliding windows becomes much higher than the corresponding best algorithms in sliding windows. We have released all the source code at Github [1] without any identity information.

KEYWORDS

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 [2, 3], financial data trackers [4, 5], sensor networks [6, 7], *etc.* A data stream is composed of an unbounded sequence of items arriving at high speed. Contrary to traditional static datasets, data streams require 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 completely. 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 a lot of bandwidth and is far from communication efficient. Instead, one effective choice is to maintain a small summary of the data stream.

Sketches, a kind of probabilistic data structures, 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, which makes it possible to store them in fast memory, such as L2 caches in CPU and GPU chips, and Block RAM in FPGA [8]. Typical sketches include the Bloom filter [9], the CM sketch [10], the CU sketch [11], *etc.* However, these sketches cannot delete the out-dated 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 financial analysis, people focus on the current finance trend, and in intrusion detection systems, people are mainly concerned about the recent intrusions. Therefore, 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 an important issue to develop probabilistic data structures which can automatically "forget" old items and focus on recent items.

The most widely used model for recording recent items is the sliding window model [12]. In this model, there is a sliding window including only the most recent items, while the items outside are forgotten (deleted).

There are various queries which can be implemented in the sliding window model. In this paper, we focus on three kinds of fundamental queries: membership query, frequency query, and heavy hitter query. Membership query is to check if an item e is in the sliding window. Frequency query is to report the frequency of an item e. Heavy hitter query is to find all the items with frequencies exceeding a threshold.

It is challenging to design a probabilistic data structure for the sliding window model. Whenever a new item arrives, 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 in $O(1)$ time to catch up with the speed of the data stream. Moreover, we cannot store all items' ID in the sliding window, because the sliding window may be very large and it is memory-consuming to store them.

1.2 Prior Art and Their Limitations

There have been quite a few algorithms on approximate queries in sliding windows. These algorithms can be divided into three kinds according to the queries they can support. The first kind supports membership queries, like the double buffering Bloom filter [13], the A^2 buffering Bloom filter [14], the forgetful Bloom filter [15] and so on. The second

kind is designed for frequency queries, like the ECM sketch [16], the splitter windowed count-min sketch [17] and so on [18, 19]. The third kind supports heavy hitter queries. This kind includes λ−sampling algorithm [20], the window compact space saving (WCSS) [21] and so on [22].

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

1.3 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 [9], the CM sketch [10], the CU sketch [11], the Count sketch [23], and the HeavyKeeper [24] for experimental evaluation in Section 7.

Before we give a brief introduction of the basic idea of our algorithm, we first introduce the common model of sketches. A typical sketch uses an array with length m, composed of elements like counters, bits or other data structures. 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 multiple items are mapped to the same bucket, and their information is stored together, resulting in errors. 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, and we report the smallest one in these k counters as the frequency in query.

Most existing algorithms keeps the basic structure of the sketches and introduce different improvements to apply them to sliding windows. It is difficult to store exactly the information in the current sliding window in sketches, because it is difficult to delete all the out-dated information. Therefore, most existing algorithms choose to store a recent period, ω, which approximates to the sliding window. Recall that in the common sketch model, each item has k mapped buckets and k copies of its information. In prior work these k mapped buckets work *synchronously*. In other words, these mapped buckets stores the information in the same period ω. The difference between this period and the actual sliding window varies at different query times, and can be large at some time points. We notice that in queries we report the most accurate one in these k mapped buckets. Therefore, we come up with a new idea. **These k mapped buckets can work asynchronously, which means they store different periods. In this way, we can achieve that whenever we query, there is a mapped bucket which records a period very close to the sliding window.** We design an algorithm called a **scanning operation** to achieve this. As a result, our algorithm has a much lower error compared to prior art.

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 $10 \sim 50$ times lower than that of the state-of-the-art sliding window algorithm. In frequency query, the ARE of the Sliding sketch is $40 \sim 50$ times lower than that of the state-of-the-art sliding window algorithm. In heavy hitter query, the precision and recall of the Sliding sketch are near to 100% and better than the state-of-the-art, and the ARE of the frequencies of heavy hitters in the Sliding sketch is $3 \sim 5.6$ times lower than that of the state-of-the-art sliding window algorithm.

1.4 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 apply our framework to three typical kinds of queries in sliding windows: membership query (Bloom filters), frequency query (sketches of CM, CU, and Count), and heavy hitter query (HeavyKeeper). Mathematical analysis and experiments show that the Sliding sketch achieves much higher accuracy than the state-of-the-art.

2 RELATED WORK

In this section, we introduce different kinds of sketches which 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 structures for data stream summarization. Classic sketches support queries in the whole data stream or a fixed period, 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 queries, sketches for frequency queries, and sketches for heavy hitter queries.

2.1.1 Sketches for Membership Queries.

Membership query is to check if an item e is in a set s or not. The most well-known sketch for membership query is the Bloom filter [9]. 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 true only if they are all 1. The Bloom filter has the property of one-side error: it only has false positives, and no false negatives. In other words, if an item e is in set s, it will definitely report true, but if e is not in the set, it still has probability to report true due to hash collisions. In recent years many variants of the Bloom filter have been proposed to meet the requirements of different applications, like the Bloomier filter [25], the Dynamic count filter [26], COMB [27] the shifting Bloom filter [28] and so on.

2.1.2 Sketches for Frequency Queries.

Frequency query is to report the frequency of an item e in a set s. There are several well-known sketches for frequency queries, like the CM sketch [10], the CU sketch [11] and the Count sketch [23]. The CM sketch is composed of a counter array with k segments. 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 for 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 frequency it reports 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 larger or smaller than the true value. Sophisticated sketches for frequency queries include the Pyramid sketch [29], the cold filter[30], the Augmented sketch [31], the bias-aware sketch [32] and so on.

2.1.3 Sketches for Heavy Hitter Queries.

Heavy hitter query is to find all the items with frequencies exceeding a threshold in a data stream. The state-of-the-art method of the heavy hitter query in data streams is the HeavyKeeper [24]. It uses a strategy called count-with-exponential-decay to actively remove items with small frequencies through decaying, and minimize the impact on heavy hitters. It reaches very high accuracy in heavy hitter queries and top$-k$ queries. Experimental results in the literature [24] show that it reduces the error rate *by about 3 orders of magnitude* in average compared to other algorithms. Other algorithms for heavy hitter queries include Frequent[33], Lossy counting [34], Space-Saving [35], unbiased space saving [36], *e.t.c.*

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 can support. The first kind supports membership queries, like the double buffering Bloom filer [13], the A^2 buffering Bloom filter[14], the Forgetful Bloom filter [15] and so on. The second kind is designed for frequency queries, like the ECM sketch[16], the splitter windowed count-min sketch [17] and so on [18, 19]. The third kind supports heavy hitter queries. This kind includes the $\lambda-$sampling algorithm [20], the window compact space saving (WCSS) [21] and so on [22]. 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

In this section, we present some important definitions about sliding windows and stream tasks, which are frequently used in this paper.

3.1 Definitions of Data Streams

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

Definition 1. ***Data Stream****: A data stream is an unbounded sequence of items* $S = \{e_1^{t_1}, e_2^{t_2}, e_3^{t_3} \ldots e_i^{t_i} \ldots\}$. *Each item* e_i *has a time stamp* 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: the time-based sliding windows and the count-based sliding windows. The definitions of them are as follows:

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

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

Both kinds of the sliding windows contain only the most recent items and ignore the old ones. Our algorithm can be used in both count-based sliding windows and time-based sliding windows. The difference between the scheme for the time-based sliding window and the scheme for the count-based sliding window is in the operation called " scanning operation", which will be shown in Section 4.2. Other operations stay the same.

Table 1: Notation Table

Notation	Meaning
S	A data stream S
$e_i^{t_i}$	A data item e_i which arrives at time t_i.
N	The length of the sliding window.
f	The frequency of an item e
$\hat{f}$	The reported frequency of an item e
A	The array in the Sliding Sketch
m	The number of buckets in array A
k	The number of segments in array A
B	A bucket in array A
p	The index of bucket B in array A
d	The number of fields in bucket B
$\{B^j \mid 1 \leqslant j \leqslant d\}$	The fields in bucket B
$\{B^{new}, B^{old}\}$	The 2 fields in the basic version
δ	The jet lag in a bucket
$\{B_i \mid 1 \leqslant i \leqslant k\}$	The mapped buckets of item e
q	The position of the scanning pointer
St_u	The update strategy of a specific sketch
St_q	The query strategy of a specific sketch

3.3 Definitions of Stream Processing Tasks

Given a sliding window, There are 3 kinds of fundamental queries which are as follows:

Definition 4. ***Membership query**: Given a sliding window W, we want to find out if item e is in it. If the answer is yes, return true, otherwise false.*

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. ***Heavy Hitter query**: Given a sliding window W, we want to find out the items with frequencies exceeding a threshold.*

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

4 SLIDING SKETCHES: 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.

4.1 A Common Sketch Model

This paper focuses on three stream processing tasks: membership query, frequency query, heavy hitter query. Recently, a kind of probabilistic data structures called sketches have been the best choice. The state-of-the-art sketches for these tasks use a common model, namely $k-$**hash model** in this paper. Next we show the details of this model: the data structure, the update operation and the query operation.

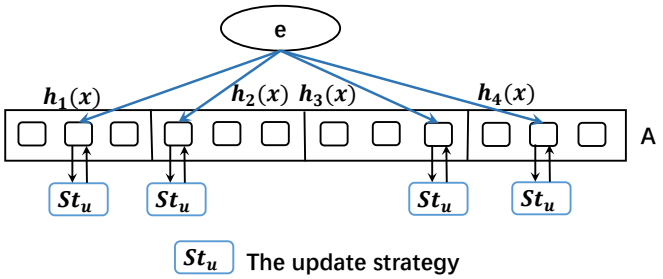


Figure 1: Update operation in the k-hash Model

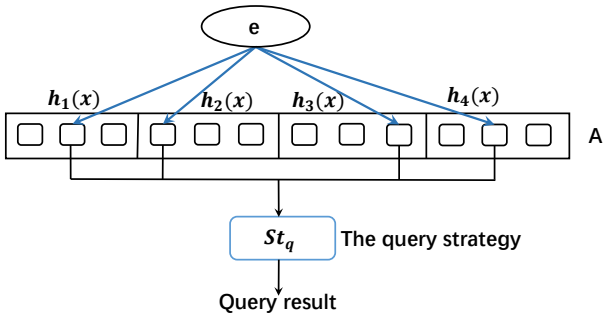


Figure 2: Query operation in the k-hash Model

Data structure: As shown in Figure 1 and 2, the $k-$hash model is an array which is composed of simple and small data structures, like counters , bits or key-value pairs. We call each element in the array a **bucket** in general. The array is divided into k equal-sized segments, each of which is 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 d mapped buckets. Then it reports a result computed from the values in the k mapped buckets with a query strategy St_q, which also varies according to the specific sketch.

An example using CM sketches: Different sketches use different update and query strategies. Take the CM sketch [10] 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 which are consistent with the k−hash model, and adapt them to the sliding window model.

Data Structure: In the Sliding sketch, we build an array A with m buckets. The array is divided into k equal-sized segments, and each of them has $\frac{m}{k}$ buckets. In the basic version, every bucket B has two fields B^{new} and B^{old}. Each field is a counter or a bit, or a key-value pair, depending on the specific sketch we choose. B^{new} stores the information mapped to bucket B in a recent period, which we call the active **Day**, or **Today**. B^{old} stores the information in the previous Day, which is called **Yesterday** in this paper. The length of each Day is equal to the length of the sliding window in this version. In other words, a Day is a period of N time units (for time-based sliding windows) or N new item arrivals (for count-based sliding windows), and δ Day ($0 < \delta \leqslant 1$) means a period of $\delta \times N$ time units (for time-based sliding windows) or $\delta \times N$ new item arrivals (for count-based sliding windows). We use information in these 2 Days to estimate the sliding window. In Section 5.2, we will extend the Sliding sketch and use d smaller fields instead of 2 fields in each bucket.

In the Sliding sketch, we have 3 kinds of operations: the update operation, the scanning operation and the query operation, which are as follows.

Update Operation: When an item e arrives, we use the k hash functions to map the item into k buckets $\{B_i | 1 \leqslant i \leqslant k\}$, one in each segment. We update the B_i^{new} filed in these k mapped buckets with the update strategy St_u of the specific sketch.

Scanning Operation: Besides the update process, we use a *scanning operation* to delete the out-dated information. We use a scanning pointer to go through A one bucket by one bucket repeatedly. Every time it reaches the end of the array, it returns to the beginning and starts a new scan. The scan speed is determined by the length of the sliding window. Specifically, for a count-based sliding window with width N, the scanning pointer goes through $\frac{m}{N}$ buckets whenever a new item arrives. In other words, the cycle of the scanning pointer is equal to the period that N items arrive in the data stream. It is the same in the time-based sliding windows. The scanning pointer scans the array in the cycle of N time units at a constant speed, which means it scans $\frac{m}{N}$ buckets in each time unit. Every time when the scanning pointer arrives at a bucket B, we call it the **zero time** of the bucket. At the zero time of a bucket, we delete the value in the B^{old} field. Then we copy the value in B^{new} to B^{old}, and set B^{new} to 0. In other words, a new Day starts. Today becomes Yesterday, and the information in Yesterday is out-dated, and is deleted. The scanning operation makes different buckets have asynchronous time, like different **time zones**.

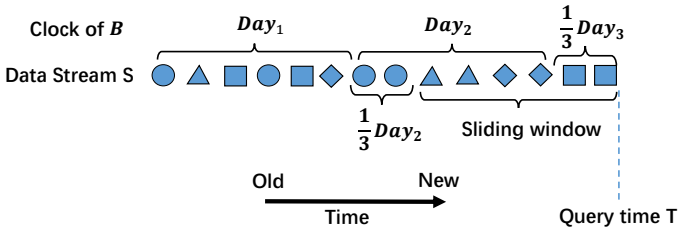


Figure 3: Days in the Sliding Sketch

Query Operation: When querying for an item e in a Sliding sketch, we find the k mapped buckets $\{B_i | 1 \leqslant i \leqslant k\}$ with the k hash functions. Then we get k value pairs: $\{({B_i}^{new}, {B_i}^{old}) | 1 \leqslant i \leqslant k\}$. At last, we get the query result based on these k value pairs with a strategy which depends on the need of applications and the specific sketch. For example, we can use the following strategy.

The Sum Strategy: We compute k sums $\{Sum(B_i) = {B_i}^{new} + {B_i}^{old} | 1 \leqslant i \leqslant k\}$, and use the query strategy St_q of the specific sketch to get the result from these k sums. For example, if the specific sketch is the CM sketch [10], we report the minimum value among the k sums, and if the specific sketch is the Bloom filter [9], we return false if any of these k sums is 0 and return true otherwise.

This strategy returns the information in both Yesterday and Today as an approximation of the sliding window. It only has over-estimation error, but no under-estimation error. Therefore this strategy can be applied to the sketches which only have over-estimation error to keep the one-side error property, such as the Bloom filter, the CM sketch, and the CU sketch. More strategies will be discussed in Section 5.1.

4.3 The 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 will analyze this operation in detail, and give a brief analysis about the accuracy of the Sliding sketch.

First we analyze the period we record in the Sliding Sketch. In each bucket B, we store the information of the items mapped to it in the active Day, or **Today** in B^{new} field, and the previous Day, or **Yesterday** in B^{old} field. In the basic version, each Day is equal to the length of the sliding window, while at query time T, only $\delta (0 < \delta \leqslant 1)$ of Today has passed. Today is shorter than the sliding window. Therefore, both the information in Today and Yesterday is needed in sliding window estimation. Notice that different buckets have different δ because of time difference.

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 B and the sliding window of the data stream is shown in Figure 3. In this example, the query time T is in the* 3_{rd} *Day in bucket B. At query time,* $\frac{1}{3}$ *of Today 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 Yesterday* Day_2*. We record both the information in Yesterday and Today to estimate the Sliding window. However, it should be noted that* $\frac{1}{3}$ *of Yesterday is out-dated.*

Next we analyze the influence of the jet lag δ and the value ranges of δ in the k mapped buckets of an item. Obviously, δ will influence the accuracy of our estimation a lot. The smaller δ is, the more accurate $B^{old} + B^{new}$ is, as it has smaller over-estimation error and is more close to the true answer. On the other hand, the bigger δ is, the more accurate B^{new} is, as it has smaller under-estimation error. Because the scanning pointer goes through the array at constant speed, δ is depended on the distance between the bucket and the scanning pointer. Assuming the scanning pointer is at the q_{th} bucket at query time, for a bucket with index p, δ in this bucket can be computed as follows:

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

Derivation of the equation is shown in 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 B' where $0 < \delta < \frac{2}{k}$, and a mapped bucket B'' 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 Section 6.2.

At last we give a brief analysis of the accuracy of the basic version. The value ranges of δ in the k mapped buckets give a guarantee of the accuracy. For example, when we use the sum strategy in the Sliding CM sketch, it only has over-estimation error and the result it returns is summarization in a period of $1 \sim \frac{k+2}{k}$ sliding windows. The analysis is as follows. When querying an item e, $Sum(B_i)$ in each mapped bucket B_i summarizes the frequency of items mapped to it in Today and Yesterday, which is $1+\delta$ times of the sliding window. Because this period is larger than the sliding window, and the CM sketch only has over-estimation error, we know that the query result is no less than the true value. As stated above, there must be a mapped bucket B' where $0 < \delta \leqslant \frac{2}{k}$. In this bucket, $Sum(B')$ contains the summarization of $1 \sim \frac{k+2}{k}$ sliding windows. Because the query strategy of in the CM sketch is to find the smallest mapped counter, B' guarantees that the final result will be near to the frequency of e in $1 \sim \frac{k+2}{k}$ sliding windows. Detailed accuracy analysis is shown in Section 6.3.

5 SLIDING SKETCH OPTIMIZATIONS

5.1 More query strategies

As stated above, there are many strategies to get the query result for an item e with the k value pairs $\{({B_i}^{new}, {B_i}^{old}) | 1 \leqslant i \leqslant k\}$. Below are a few examples:

Under-estimation Strategy: We can also only use information in Today in the k mapped buckets to get an under-estimation of the result. In this strategy, we find the k mapped buckets $\{B_i | 1 \leqslant i \leqslant k\}$, and get k values $\{{B_i}^{new} | 1 \leqslant i \leqslant k\}$. Then we use the query strategy St_q of the specific sketch to get a result from these k values. As Today is only δ of the sliding window and $0 < \delta \leqslant 1$, this usually gives an under-estimation. This strategy is suitable for the sketches which have under-estimation error or have two-side error, like the Count sketch [23] and the HeavyKeeper [24].

Corrected Result: In each mapped bucket, we can compute the jet lag in it. Therefore we know the approximate ratio of the length of the Today and Yesterday against the sliding window. We can divide the queried results of the sum strategy or the under-estimation strategy with the corresponding ratio to correct the result. This strategy can be used when the stream has a nearly constant speed and the items in it spread uniformly.

5.2 Using more fields

In the basic version, we set 2 fields in each bucket. When the memory is sufficient, we can use d fields $\{B^j | 1 \leqslant j \leqslant d\}$ in each bucket B. These d fields record the information in the last d Days. If we suppose that Today is Day_t, B^j records information in Day_{t-j+1}. In this case, 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 the item into k buckets $\{B_i | 1 \leqslant i \leqslant k\}$, one in each segment. We update the ${B_i}^1$ field in these k mapped buckets with strategy St_u of the specific sketch.

The Scanning Operation. The scanning pointer scans $\frac{(d-1)\times m}{N}$ buckets each time an item arrives (count based sliding window) or in each time unit (time based sliding window). When the scanning pointer arrives at bucket B, we set $B^j = B^{j-1} (2 \leqslant j \leqslant d)$ and $B^1 = 0$, because a new Day starts, and all the stored information becomes one Day older.

The Query Operation. When querying an item e, we find the k mapped buckets $\{B_i | 1 \leqslant i \leqslant k\}$ for the queried item e with the k hash functions. Then we get k sets of values: $\{{B_i}^j | \ 1 \leqslant j \leqslant d, \ 1 \leqslant i \leqslant k\}$. At last, we get the query result based on these k sets of values with a strategy which depends

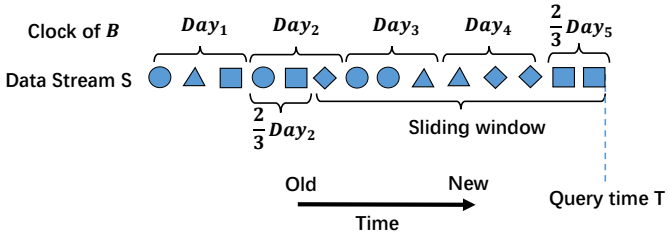


Figure 4: Days in the Multi-field Version

on the need of applications and the specific sketch. The strategies mentioned in Section 5.1 can all be used, but slight adjustment is needed. For example, the under-estimation strategy becomes as follows:

Under-estimation Strategy: For each queried item e we find the k mapped buckets $\{B_i|1 \leqslant i \leqslant k\}$ and compute k values $\{UE(B_i) = \sum_{j=1}^{d-1} B_i^j | 1 \leqslant i \leqslant k\}$, and use the query strategy St_q of the specific sketch to get the result from these k values.

We do not discuss all the strategies in detail here due to space limitation.

Next we use an example to show the relationship between the Days and the sliding window.

Example 2. *An example of the Days in a bucket B and the sliding window is shown in Figure 4. In this example, we set $d = 4$. Each bucket contains 4 fields, and each Day is $\frac{1}{3}$ of the sliding window. At the query time T, bucket B 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 B records a period which is $\frac{11}{9}$ of the sliding window.*

When we use multiple fields, the accuracy will become higher. The jet lag δ can still be computed with equation 1. For each item e, the value ranges of δ in the k mapped buckets are also the same as the basic version. However, as Days in each bucket become $\frac{1}{d-1}$ of the sliding window, the error brought by approximation of the sliding window also becomes $\frac{1}{d-1}$. However, increasing d does not necessarily bring improvements in accuracy if the memory is insufficient. When using the same memory, enlarging d means the length of the array has to become smaller, and the error brought by hash collisions will increase.

6 MATHEMATICAL ANALYSIS

In this part we analyze the memory and time cost of the Sliding sketch, and analyze the value range of δ. The accuracy of the Sliding sketch depends on the specific sketch and the query strategy, and we give the analysis of the the accuracy of 3 kinds of Sliding sketch applications as examples, namely the Sliding Bloom filter for the membership query, the Sliding CM sketch for the frequency query, and the Sliding HeavyKeeper for the heavy hitter query.

6.1 Analysis of memory and time cost

The space cost of the Sliding sketch is d times of the specific sketch, where d is the number of fields in each bucket. This memory cost is better than most prior art of algorithms for sliding windows. The time cost of update in every item arrival is $O(1)$. The move of the scanning pointer can be implemented in another thread, or in the inserting process of each item. As whenever an item arrives or the clock increases, $\frac{(d-1)\times m}{N}$ buckets need to be scanned, the time cost of scanning buckets is $O(\frac{(d-1)\times m}{N})$, which is usually a small constant. For example, in the experiment of Sliding HeavyKeeper, the length of the sliding window $N = 1M$, the length of the array $m = 40k$, and the number of fields $d = 4$. In this case $\frac{(d-1)\times m}{N}$ is only 1.2. Because these buckets are adjacent, reading or writing them is usually very fast.

6.2 Analysis of the jet lag δ

6.2.1 *The Computation of δ.*

For each bucket/time zone B in the array, the jet lag δ, which represents how much of the Today has passed by query 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 p, 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 $p \leqslant q$, the scanning pointer has scanned $q - p$ buckets after its arrival at B. Therefore

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

2) When $p > q$, the scanning pointer has scanned $(m-p)+q$ buckets after its arrival at B. Therefore

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

For the bucket where the pointer is in, we define $\delta = 1$.

6.2.2 *The Value Range of δ.*

Theorem 1. *Given an item e with k mapped buckets in the Sliding sketch, The jet lags δ in all these mapped buckets are in range $(0, 1]$. Moreover, There must be at least one mapped bucket with a 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 $\delta < \frac{i}{k}$, $\forall 2 \leqslant i \leqslant k-1$ 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 $\delta > 1 - \frac{i}{k}$, $\forall 2 \leqslant i \leqslant k-1$, 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 B' which is in the same segment with the scanning pointer, and we represent its index with p'. There are two kinds of situations:

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

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

Therefore in this bucket B' 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 B'', whose index in the array is p''. Then $p'' - q$ is less than the length of two segments, which is $\frac{2\times m}{k}$. In bucket B'', we have

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

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

2) When $p' \geqslant q$, B' has the largest δ among the k mapped buckets. $p' - q$ is less than the length of the section, which is $\frac{m}{k}$. In bucket B', we have

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

Therefore, in this bucket B', we have $\frac{k-1}{k} < \delta \leqslant 1$. In this situation the smallest δ appears in the last section, we represent the mapped bucket in this section with B'', whose index in the array is p''. Then $q - p''$ is less than the length of two sections, which is $\frac{2\times m}{k}$. In bucket B'', we have

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

Therefore, in this bucket B'', we have $0 < \delta < \frac{2}{k}$. For the other $k-2$ mapped buckets, as they are mapped to different sections and each section has the same length, the δ in them has value ranges which form an arithmetic sequence, which is : $\{(\frac{j}{k}, \frac{j+2}{k}) | 1 \leqslant j \leqslant k-2\}$.

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

6.3 Analysis of the Accuracy

The accuracy of the Sliding sketch is influenced by the specific sketch and the strategy we use. We analyzed the accuracy of the sliding Bloom filter, the sliding CM sketch and the sliding HeavyKeeper as examples. Analysis of other sliding sketches is similar.

6.3.1 *Accuracy of the Sliding Bloom Filter.*

Theorem 4. *The Sliding Bloom filter only has false positives and no false negatives if we use the sum strategy.*

Proof. In each mapped bucket B_i of item e in the Sliding Bloom filter, $Sum(B_i) = \sum_{j=1}^{d} B_i^j$ records the presence of items mapped to B in the last $1 + \frac{\delta}{d-1}$ times of the sliding window. This period is larger than 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 the data stream in the last $1 + \frac{\delta}{d-1}$ times of the sliding window, and $Sum(B_i) > 0$. When in all the k mapped buckets $Sum(B_i) > 0$, we will report true. However, when e is not in the sliding window, we may still report true because we record a longer period, and hash collisions may happen. □

Theorem 5. *In the Sliding Bloom filter with the sum strategy, 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 the most recent period of $1 + \frac{i}{(d-1)\times k} (2 \leqslant i \leqslant k)$ sliding windows, and use n_i to represent the number of items in this period. Then we have*

$$\begin{aligned} Pr_i &\geqslant 1 - (1 - (1 - \frac{k}{m})^{n_i})^{i-1} \\ &\approx 1 - (1 - e^{-\frac{n_i k}{m}})^{i-1}, \quad (2 \leqslant i \leqslant k-1) \end{aligned} \tag{8}$$

and

$$\begin{aligned} Pr_k &\geqslant 1 - (1 - (1 - \frac{k}{m})^{n_k})^{k} \\ &\approx 1 - (1 - e^{-\frac{n_k k}{m}})^{k} \end{aligned} \tag{9}$$

Proof. In the Bloom filter, we analyze the false positive rate, as it does not have false negatives. In [9] the author has proved that given a set with n distinct items, when querying for an item e which is not in the set with a partition Bloom filter, the probability that there is at one bit correct among j mapped bits is

$$\begin{aligned} Pr &= 1 - (1 - (1 - \frac{k}{m})^{n})^{j} \\ &\approx 1 - (1 - e^{-\frac{nk}{m}})^{j} \end{aligned} \tag{10}$$

where m is the length of the array and k is the number of segments. This is the probability that we can get a correct

answer with j mapped bits, because as long as one bit is 0, we will give a negative report. In this following part we analyze the accuracy of the Sliding Bloom filter with this result.

In the sliding Bloom filter, suppose we use the sum strategy. When querying an item e 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 the most recent period of $1 + \frac{i}{(d-1)\times k} (2 \leqslant i \leqslant k)$ sliding windows, and use n_i to represent the number of items in this period. For $2 \leqslant i \leqslant (k-1)$, there are at least $i-1$ buckets where $\delta < \frac{i}{k}$, as shown in Theorem 2. These buckets record periods smaller than $1 + \frac{i}{(d-1)\times k}$ sliding windows. Therefore we have

$$\begin{aligned} Pr_i &\geqslant 1 - (1 - (1 - \frac{k}{m})^{n_i})^{i-1} \\ &\approx 1 - (1 - e^{-\frac{n_i k}{m}})^{i-1}, \quad (2 \leqslant i \leqslant k-1) \end{aligned} \tag{11}$$

There are k buckets which record periods smaller than 2 sliding windows, therefore we have

$$\begin{aligned} Pr_k &\geqslant 1 - (1 - (1 - \frac{k}{m})^{n_k})^{k} \\ &\approx 1 - (1 - e^{-\frac{n_k k}{m}})^{k} \end{aligned} \tag{12}$$

□

6.3.2 Accuracy of the Sliding CM sketch.

Theorem 6. *The Sliding CM sketch only has over-estimation error and no under-estimation error if we use the sum strategy.*

Proof. In each mapped bucket B_i of item e in the Sliding CM sketch, $Sum(B_i) = \sum_{j=1}^{d} B_i^j$ records the sum of frequencies of items mapped to B in the last $1 + \frac{\delta}{d-1}$ sliding windows. This period is larger than 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 more than f times in the data stream in the last $1 + \frac{\delta}{d-1}$ sliding windows, and $Sum(B_i) > f$. When in all the k mapped buckets $Sum(B_i) > f$, the reported value $\hat{f}$ will be larger than f. □

Theorem 7. *In the Sliding CM sketch, suppose we use the sum strategy, and the accurate frequency of an item is f, and the reported frequency is $\hat{f}$. If we use Pr_i to represent probability that $\hat{f} \leqslant f_i + \epsilon N_i, \ (2 \leqslant i \leqslant k)$ where N_i is the number of items in the most recent period of $(1 + \frac{i}{k\times(d-1)})$ of the sliding window, and f_i is the frequency of the item in this period. $\epsilon = \frac{ke}{m}$. We have*

$$\begin{aligned} Pr_i &= Pr(\hat{f} \leqslant f_i + \epsilon N_i) \\ &\geqslant 1 - e^{-i+1}, \quad (2 \leqslant i \leqslant k-1) \end{aligned} \tag{13}$$

$$\begin{aligned} Pr_k &= Pr(\hat{f} \leqslant f_k + \epsilon N_k) \\ &\geqslant 1 - e^{-k} \end{aligned} \tag{14}$$

Notice that for count based sliding window $N_i = (1 + \frac{i}{k\times(d-1)})N$ where N is the length of the sliding window, but for time based sliding window this is no longer true.

Proof. In [10], the author has proved that when querying for an item e with the CM sketch, we have

$$Pr(\hat{f} \leqslant f + \epsilon N) \geqslant 1 - e^{-k} \tag{15}$$

where 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. N is the number of items in the set.

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

$$Pr(\hat{f} \leqslant f + \epsilon N) \geqslant 1 - e^{-j} \tag{16}$$

In this following part we analyze the accuracy of the Sliding CM sketch with these results. We use Pr_i to represent probability that $\hat{f} \leqslant f_i + \epsilon N_i, \ (2 \leqslant i \leqslant k)$ where N_i is the number of items in the most recent period of $(1 + \frac{i}{k\times(d-1)})$ of the sliding window, and f_i is the accurate frequency of the item in this period. $\epsilon = \frac{ke}{m}$. $\hat{f}$ represents the reported frequency.

For $2 \leqslant i \leqslant (k-1)$, there are at least $i-1$ buckets where $\delta < \frac{i}{k}$, as shown in Theorem 2. These buckets record periods which are $1 \sim 1 + \frac{i}{(d-1)\times k}$ sliding windows. The reported value of the sliding sketch will be no larger than the minimum one in these $i-1$ buckets. Therefore we have

$$\begin{aligned} Pr_i &= Pr(\hat{f} \leqslant f_i + \epsilon N_i) \\ &\geqslant 1 - e^{-i+1}, \quad (2 \leqslant i \leqslant k-1) \end{aligned} \tag{17}$$

There are k buckets which record periods which are $1 \sim 1 + \frac{k}{(d-1)\times k}$ sliding windows. Therefore we have

$$\begin{aligned} Pr_k &= Pr(\hat{f} \leqslant f_k + \epsilon N_k) \\ &\geqslant 1 - e^{-k} \end{aligned} \tag{18}$$

□

6.3.3 Accuracy of the Sliding HeavyKeeper.

Theorem 8. *For the frequency of any heavy hitter, The Sliding HeavyKeeper only has under-estimation error and no over-estimation error if we use the under-estimation strategy.*

Proof. In each mapped bucket B_i of item e in the Sliding CM sketch, $UE(B_i) = \sum_{j=1}^{d-1} B_i^j$ records the frequencies of the item stored in B in the last $1 - \frac{1-\delta}{d-1}$ sliding windows. This period is smaller than the sliding window. On the other hand, the HeavyKeeper only has under-estimation for the

frequencies of heavy hitters, as when a heavy hitter collides with other items, its frequency may be decreased. Therefore, we know that if a heavy hitter e has frequency f in sliding window, it must shows up less than f times in the data stream in the last $1 - \frac{1-\delta}{d-1}$ sliding windows, and because of hash collisions, the recorded frequency may be smaller. □

Theorem 9. *In the Sliding Heavy Keeper, suppose we use the under-estimation strategy, and the accurate frequency of an item is f, and the reported frequency if $\hat{f}$. If we use Pr_i to represent probability that $\hat{f} \geqslant f_i - \epsilon N_i$, $(2 \leqslant i \leqslant k)$ where N_i is the number of items in the most recent period of $(1 - \frac{i}{k\times(d-1)})$ of the sliding window, and f_i is the frequency of the item in this period. ϵ is any small positive number, we have*

$$\begin{aligned} Pr_i &= Pr(\hat{f} \geqslant f_i - \epsilon N_i) \\ &\geqslant 1 - (\frac{k}{\epsilon m f(b-1)})^{i-1}, \quad (2 \leqslant i \leqslant k-1) \end{aligned} \tag{19}$$

$$\begin{aligned} Pr_k &= Pr(\hat{f} \geqslant f_k - \epsilon N_k) \\ &\geqslant 1 - (\frac{k}{\epsilon m f(b-1)})^{k} \end{aligned} \tag{20}$$

where b is the constant for probabilistic decay and m is the length of the array and k is the number of segments. Similar to the sliding CM sketch, for count based sliding window $N_i = (1 - \frac{i}{k\times(d-1)})N$ where N is the length of the sliding window, but for time based sliding window this is no longer true.

Proof. In [24], the author has proved that when querying for an item e with the HeavyKeeper, in each segment we have:

$$Pr(\hat{f} \leqslant f - \epsilon N) \leqslant \frac{k}{\epsilon m f(b-1)} \tag{21}$$

where f is the accurate frequency of a heavy hitter e, $\hat{f}$ is the frequency stored in this segment. ϵ is any small positive number. m is the length of the array, and k is the number of segments. b is the constant for probabilistic decay. N is the number of items in the set.

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

$$Pr(\hat{f} \geqslant f - \epsilon N) \geqslant 1 - (\frac{k}{\epsilon m f(b-1)})^{j} \tag{22}$$

In this following part we analyze the accuracy of the Sliding HeavyKeeper with these results. We use Pr_i to represent probability that $\hat{f} \geqslant f_i - \epsilon N_i$, $(2 \leqslant i \leqslant k)$ where N_i is the number of items in the most recent period of $(1 - \frac{i}{k\times(d-1)})$ of the sliding window, and f_i is the frequency of the item in this period. ϵ is any small positive number. $\hat{f}$ represents the reported frequency.

For $2 \leqslant i \leqslant (k-1)$, there are at least $i-1$ buckets where $\delta > 1 - \frac{i}{k}$, as shown in Theorem 3. These buckets record periods which are $1 - \frac{1-\delta}{d-1} \sim 1$ sliding windows, larger than $(1 - \frac{i}{k\times(d-1)})$ sliding windows. Therefore we have

$$\begin{aligned} Pr_i &= Pr(\hat{f} \geqslant f_i - \epsilon N_i) \\ &\geqslant 1 - (\frac{k}{\epsilon m f(b-1)})^{i-1}, \quad (2 \leqslant i \leqslant k-1) \end{aligned} \tag{23}$$

There are k buckets which record periods which are $1 - \frac{1}{d-1} \sim 1$ sliding windows. Therefore we have

$$\begin{aligned} Pr_k &= Pr(\hat{f} \geqslant f_k - \epsilon N_k) \\ &\geqslant 1 - (\frac{k}{\epsilon m f(b-1)})^{k} \end{aligned} \tag{24}$$

□

7 PERFORMANCE EVALUATION

In this section, we apply the Sliding sketch to five kinds of sketches: the Bloom filter [9], the CM sketch [10], the CU sketch [11], the Count sketch [23] and the HeavyKeeper [24]. 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 for experimental evaluation.

7.1 Experimental Setup

Datasets:

1) IP Trace Dataset: IP trace dataset contains anonymized IP trace streams collected in 2016 from CAIDA [37]. Each item is identified by its source IP address (4 bytes).

2) Web Page Dataset: We download Web page dataset from the website [38]. Each item (4 bytes) represents the number of distinct terms in a web page.

3) Network Dataset: The network dataset contains users' posting history on stack exchange website [39]. 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.

4) Synthetic Dataset: By using Web Polygraph [40], an open source performance testing tool, we generate the synthetic dataset, which follows the Zipf [41] distribution. This dataset has 32M items, and the skewness is 1.5. The length of each item is 4 bytes.

Implementation: We implemented all the algorithms in C++ and made them open sourced [1]. The hash functions are 32-bit Bob Hash (obtained from the open source website [43]) with different initial seeds. All of the abbreviations of

Table 2: Abbreviations of algorithms in experiments

Abbreviation	Full name
Sl-BF	Sliding Bloom Filter
FBF	Forgetful Bloom Filter[15]
SWBF	Technique in [42] 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[16]
SWCM	Splitter Windowed Count-Min Sketch[17]
Sl-HK	Sliding HeavyKeeper
λ-sampling	λ-sampling Algorithm[20]
WCSS	Window Compact Space-Saving[21]

algorithms we use in the evaluation and their full name are shown in Table 2. We conducted all 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 experiment, we discover that after reading enough items (usually $1 \sim 2$ window sizes), the experiment result will become stable. We measure the metrics whenever the window slides $\frac{1}{5}N$ and compute the average value (N is the length of the sliding window). We use the average value to represent the experiment result at given parameter setting. The error bar represents the minimal value and the maximum value. We use the following metrics to evaluate the performance of our algorithms:

1) Error Rate in Membership Estimation: Ratio of the number of incorrectly reported instances to all instances being queried. We use error rate because FBF and SW-BF have two-side error. The query set we use include all the n distinct items in the present sliding window and n items which are not in the sliding window.

2) Average Relative Error (ARE) in Frequency Estimation: $\frac{1}{|\Psi|}\sum_{e_i\in\Psi}\frac{|f_i-\widehat{f_i}|}{f_i}$, where f_i is the real frequency of item e_i, $\widehat{f_i}$ is its estimated frequency, and Ψ is the query set.

3) Average Absolute Error (AAE) in Frequency Estimation: $\frac{1}{|\Psi|}\sum_{e_i\in\Psi}|f_i-\widehat{f_i}|$, where f_i is the real frequency of item e_i, $\widehat{f_i}$ is its estimated frequency, and Ψ is the query set.

Here, we query the dataset by querying each distinct item once in the present sliding window.

4) Precision Rate in finding Heavy Hitter: Ratio of the number of correctly reported instances to the number of reported instances.

5) Recall Rate in finding Heavy Hitter: Ratio of the number of correctly reported instances to the number of correct instances.

6) Average Relative Error (ARE) in finding Heavy Hitter: $\frac{1}{|\Psi|}\sum_{e_i\in\Psi}\frac{|f_i-\widehat{f_i}|}{f_i}$, where f_i is the real frequency of item e_i, $\widehat{f_i}$ is its estimated frequency, and Ψ is the real heavy hitters set in the present sliding window.

7) Speed: Million operations (insertions) per second (Mops). All the experiments about speed are repeated 100 times to ensure statistical significance.

7.2 Evaluation on Membership Query

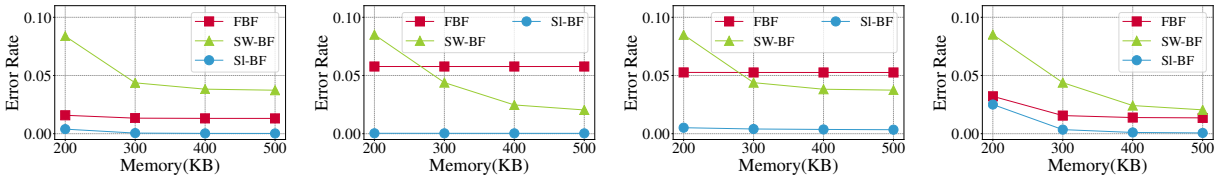
Parameter Setting: We compare 3 approaches: Sl-BF, FBF, and the SWBF. Let k be the number of hash functions. For our Sl-BF, we set $k = 10$. Let d be the number of fields in each bucket. For our Sl-BF, we set $d = 2$. For FBF, the parameters are set according to the recommendation of the authors. For 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 function. Details of the algorithm can be seen in the original paper [42]. For each dataset, we read 500k items. We set the length of the sliding window $N = 100$k. In the experiment, we compare error rate and insertion speed.

Error Rate (Figure 5(a)-5(d)): Our results show that the Error Rate of Sl-BF is about 10 times lower than the prior arts when the memory is set to 200KB on three real-world datasets and one synthetic dataset. When the memory is increased to 500KB, the Error Rate of Sl-BF is up to 50 times lower than the state-of-the-art. This difference is because we can record the presence of items in a period very close to the sliding window with only one extra bit in each bucket. On the other hand, prior algorithms need more complicated structure to achieve a good approximation of the sliding window, which is still not as good as ours. This limits the length of Bloom filters when the memory usage is fixed and enlarge the influence of hash collisions. It is similar in the experiments of frequency queries.

Insertion Speed (Figure 6(a)-6(d)): Our results show that the Insertion Speed of Sl-BF is about $2 \sim 3$ times faster than FBF. The speed of the SWBF is higher than our algorithm, but its accuracy is much poorer.

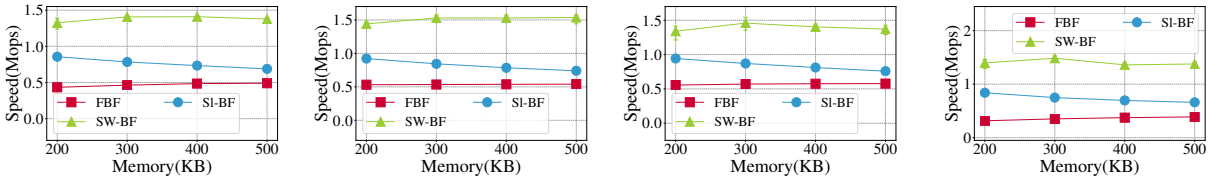
7.3 Evaluation on Frequency Query

Parameter Setting: We compare 5 approaches: Sl-CM, Sl-CU, Sl-Count, ECM and SWCM. Let k be the number of hash functions, and let d be the number of fields in each bucket. For our Sliding sketch, we set $k = 10$, $d = 2$. For ECM and SWCM, the parameters are set according to the recommendation of



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

Figure 5: Error Rate of Membership Query.



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

Figure 6: Insertion Speed of Membership Query.

the authors. For each dataset, we read 100k items. We set the length of the sliding window N = 50k. In the experiment, we compare ARE, AAE and insertion speed.

ARE (Figure 7(a)-7(d)): Our results show that the ARE of Sl-CM is about 150 and 40 times lower than ECM and SWCM respectively when the memory is set to 2MB on three real-world datasets and one synthetic dataset. The ARE of Sl-CU is about 200 and 50 times lower than ECM and SWCM respectively in the same condition. The ARE of Sl-Count is about 150 and 40 times lower than ECM and SWCM respectively in the same condition.

AAE (Figure 8(a)-8(d)): Our results show that the AAE of Sl-CM is about 70 and 10 times lower than ECM and SWCM respectively when the memory is set to 2MB on three real-world datasets and one synthetic dataset. The AAE of Sl-CU is about 150 and 20 times lower than ECM and SWCM respectively. The AAE of Sl-Count is about 70 and 10 times lower than ECM and SWCM respectively.

Insertion Speed (Figure 9(a)-9(d)): Our results show that the insertion speed of Sl-CM is about 25 and 3.9 times faster than ECM and SWCM respectively when memory is set to 2MB. The insertion speed of Sl-CU is about 18.6 and 3.2 times faster than ECM and SWCM respectively. The insertion speed of Sl-Count is about 20 and 3.4 times faster than ECM and SWCM respectively.

7.4 Evaluation on Heavy Hitter Query

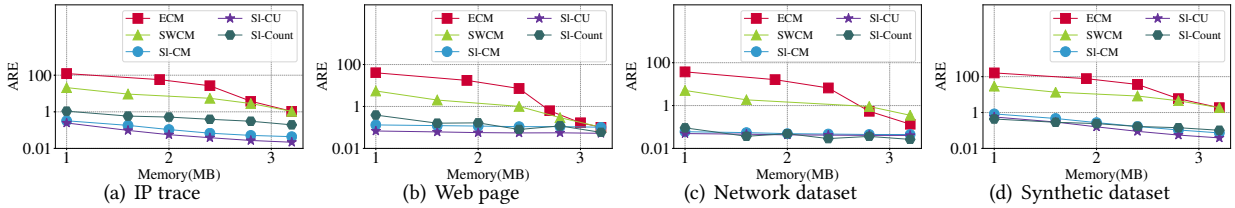
Parameter Setting: We compare 3 approaches: Sl-HK, λ-sampling and WCSS. Let k be the number of hash functions, and let d be the number of fields in each bucket. For our Sliding sketch, we set k = 10, d = 4. For λ-sampling and WCSS, the parameters are set according to the recommendation of the authors. For each dataset, we read 10M items. We set the length of the sliding window N = 1M. When the frequency of an item in the present sliding window is more than 1000, we consider it as a heavy hitter. We compare precision rate, recall rate, ARE and insertion speed between the 3 approaches.

Precision Rate and Recall Rate (Figure 10(a)-11(d)): In our experiment, we set the size of memory between 100KB and 200 KB. Our results show that for our Sl-HK, both precision rate and recall rate achieve nearly 100% on three real-world datasets and one synthetic dataset. For λ-sampling, it starts to work only after memory size is more than 160KB. The precision rate and recall rate is 0% before. For WCSS, although the recall rate can reach nearly 100%, the precision rate is much lower than Sl-HK. Our results show that the precision rate of Sl-HK is about 1.7 times higher than WCSS on three real-world datasets when memory is set to 200KB. The recall rate is about 2.5 times higher than λ-sampling.

ARE (Figure 12(a)-12(d)): Our results show that the ARE of Sl-HK is about 17.4 and 5.6 times lower than λ-sampling and WCSS on three real-world dataset when memory is set to 200KB. The ARE of Sliding HeavyKeepr is about 2.4 and 3.0 times lower than λ-sampling and WCSS on synthetic dataset when memory is set to 200KB.

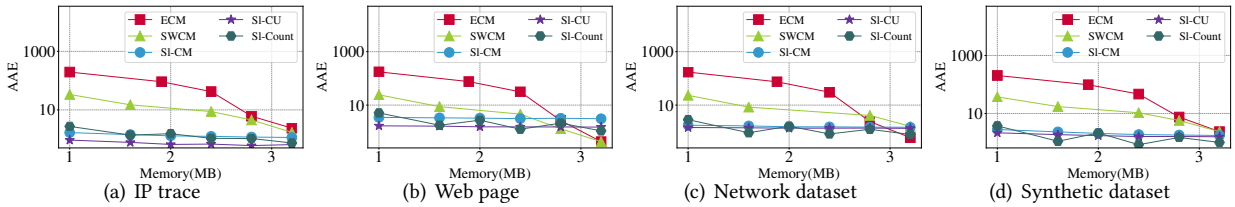
Insertion Speed (Figure 13(a)-13(d)): Our results show that the insertion speed of Sl-HK is about 1.42 times faster than λ-sampling. Although the insertion speed of Sl-HK is slightly slower than WCSS, the accuracy performance is much better than WCSS.

The superiority of Sl-HK is due to 2 reasons. First, HeavyKeeper algorithm can get a much higher accuray in heavy hitter queries compared to prior arts. By adapting it to sliding windows, we obtain its superiority. Second, our Sliding sketch framework has a better approximation of the sliding



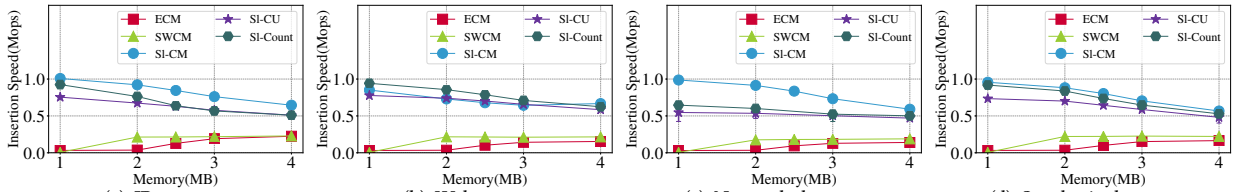
(a) IP trace (b) Web page (c) Network dataset (d) Synthetic dataset

Figure 7: ARE of Frequency Query.



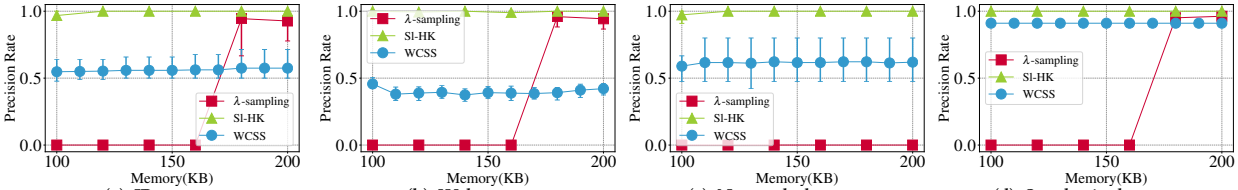
(a) IP trace (b) Web page (c) Network dataset (d) Synthetic dataset

Figure 8: AAE of Frequency Query.



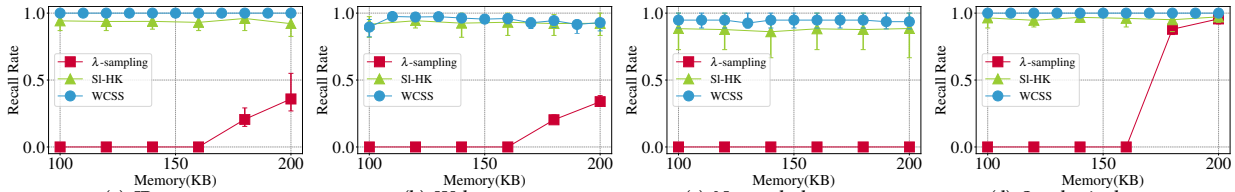
(a) IP trace (b) Web page (c) Network dataset (d) Synthetic dataset

Figure 9: Insertion Speed of Frequency Query.



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

Figure 10: Precision Rate of Heavy Hitter Query.



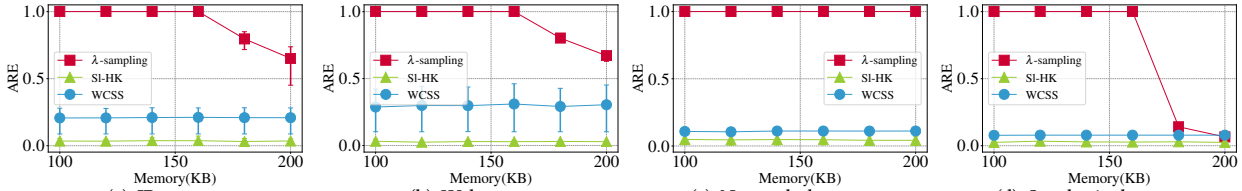
(a) IP trace (b) Web page (c) Network dataset (d) Synthetic dataset

Figure 11: Recall Rate of Heavy Hitter Query.

window compared to other algorithms. These 2 strengths help Sl-HK out-perform other algorithms.

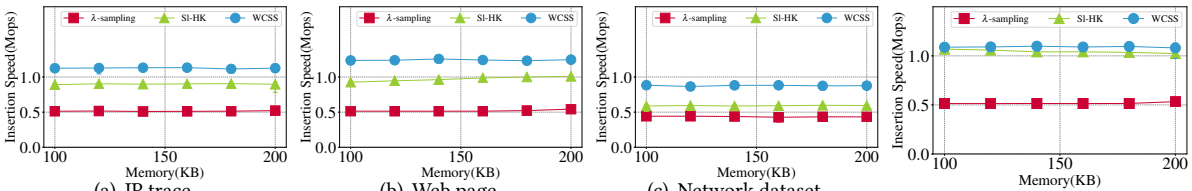
7.5 Sensitivity Analysis

In this section, we evaluate the impact of the number of fields, d on the performance of the Sliding sketch. We only show the result of the Sl-HK on heavy hitter queries here due to space limitation. In the experiment we observe the impact of the number of fields in each bucket on precision rate, recall rate, ARE and insertion speed for Sl-HK. We use IP trace dataset in this experiment. Let k be the number of hash functions, we set $k = 10$. The length of the sliding window



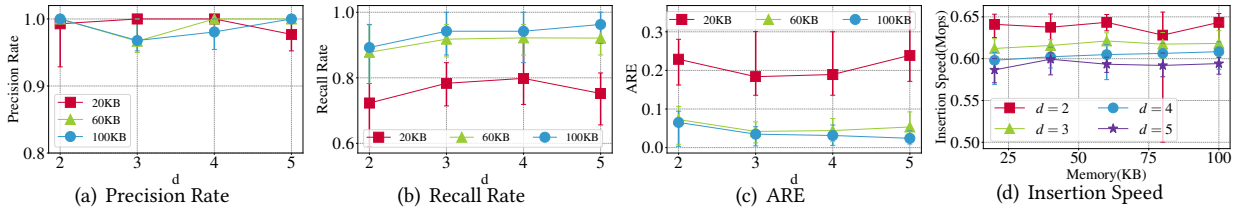
(a) IP trace (b) Web page (c) Network dataset (d) Synthetic dataset

Figure 12: ARE of Heavy Hitter Query.



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

Figure 13: Insertion Speed of Heavy Hitter Query.



(a) Precision Rate (b) Recall Rate (c) ARE (d) Insertion Speed

Figure 14: Varying the number of the field in each bucket of HeavyKeeper.

is set $N = 1M$. When the frequency of an item exceeds 1000, we reprot it as a heavy hitter.

`Impact of the number of fields in each bucket on Precision Rate (Figure 14(a))`: We observe that the impact of d on precision rate is little. The precision rate can always reach nearly 100% when changing d.

`Impact of the number of fields in each bucket on Recall Rate (Figure 14(b))`: We observe that the best parameter setting of the number of fields in each bucket is $d = 4$ when the memory is set to 20KB. When increasing the memory size, a larger d can provide a higher recall rate.

`Impact of the number of fields in each bucket on ARE (Figure 14(c))`: We observe that the best parameter setting of the number of fields in each bucket is $d = 4$ when the memory is set to 20KB. When increasing the memory size, a larger d can provide a lower ARE.

`Impact of the number of fields in each bucket on Insertion Speed (Figure 14(d))`: We observe that the insertion speed will decrease when increasing d. For different memory size, the insertion speed is very close.

`Summary`: From the experiments we can see that by adjusting the number of fields in each bucket, we may get a better accuracy. However, increasing the number of fields does not necessarily brings improvement. Because when the memory is limited, increasing the number of fields means decreasing the number of buckets, and the error brought by hash collisions will increase. We can also see that increasing d will bring decrease in update speed. Therefore, a careful trade off is needed.

8 CONCLUSION

Data stream processing in sliding windows is an important and challenging work. While existing work uses one sketch to address one single query, we propose a generic framework in this paper, namely the Sliding sketch which can be applied to most existing sketches and answer various kinds of queries in sliding windows. The key novelty in our framework is to introduce time zones to sketches, and to guarantee that at least one mapped bucket has small enough jet lag. We use our framework to address three fundamental queries in sliding windows: membership query (the Bloom filter), frequency query (the CM sketch, the CU sketch, and the Count sketch) and heavy hitter query (HeavyKeeper). Theoretical analysis and experimental results show that after using our framework, the above five sketches that do not support sliding windows achieve much higher than the corresponding best algorithms in sliding windows. We believe our framework is suitable for all sketches that use the common sketch model.

REFERENCES

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

[2] Sang Hyun Oh, Jin Suk Kang, Yung Cheol Byun, Taikyeong T Jeong, and Won Suk Lee. Anomaly intrusion detection based on clustering a data stream. In *Acis International Conference on Software Engineering Research, Management and Applications*, pages 220–227, 2006.

[3] Mustafa Amir Faisal, Zeyar Aung, John R. Williams, and Abel Sanchez. Securing advanced metering infrastructure using intrusion detection system with data stream mining. In *Pacific Asia Conference on Intelligence and Security Informatics*, pages 96–111, 2012.

[4] Bryan Ball, Mark Flood, H. V. Jagadish, Joe Langsam, Louiqa Raschid, and Peratham Wiriyathammabhum. A flexible and extensible contract aggregation framework (caf) for financial data stream analytics. pages 1–6, 2014.

[5] Lajos Gergely GyurkÃş, Terry Lyons, Mark Kontkowski, and Jonathan Field. Extracting information from the signature of a financial data stream. *Quantitative Finance*, 2013.

[6] Ruo Hu. Stability analysis of wireless sensor network service via data stream methods. *Applied Mathematics Information Sciences*, 6(3):793–798, 2012.

[7] Carlos M. S. Figueiredo, Carlos M. S. Figueiredo, Eduardo F. Nakamura, Luciana S. Buriol, Antonio A. F. Loureiro, Antnio Otvio Fernandes, and Claudionor J. N. Jr Coelho. Data stream based algorithms for wireless sensor network applications. In *International Conference on Advanced Information NETWORKING and Applications*, pages 869–876, 2007.

[8] FPGA data sheet [on line]. http://www.xilinx.com/support/documentation/data_sheets/ds180_7Series_Overview.pdf.

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

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

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

[12] Mayur Datar, Aristides Gionis, Piotr Indyk, and Rajeev Motwani. Maintaining stream statistics over sliding windows. *Siam Journal on Computing*, 31(6):1794–1813, 2002.

[13] F. Chang, Wu Chang Feng, and Kang Li. Approximate caches for packet classification. In *Joint Conference of the IEEE Computer and Communications Societies*, pages 2196–2207 vol.4, 2004.

[14] Yoon. Aging bloom filter with two active buffers for dynamic sets. *IEEE Transactions on Knowledge Data Engineering*, 22(1):134–138, 2009.

[15] Rajath Subramanyam, Indranil Gupta, Luke M. Leslie, and Wenting Wang. Idempotent distributed counters using a forgetful bloom filter. *Cluster Computing*, 19(2):879–892, 2016.

[16] Odysseas Papapetrou, Minos Garofalakis, and Antonios Deligiannakis. Sketch-based querying of distributed sliding-window data streams. *Proceedings of the VLDB Endowment*, 5(10):992–1003, 2012.

[17] Nicoló Rivetti, Yann Busnel, and Achour 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.

[18] Ho Leung Chan, Tak Wah Lam, Lap Kei Lee, and Hing Fung Ting. *Continuous Monitoring of Distributed Data Streams over a Time-Based Sliding Window*. 2009.

[19] Graham Cormode and Ke Yi. Tracking distributed aggregates over time-based sliding windows. In *ACM Sigact-Sigops Symposium on Principles of Distributed Computing*, pages 213–214, 2011.

[20] Hung, Y. S Regant, Lee, Lap-Kei, Ting, and H.F. Finding frequent items over sliding windows with constant update time. *Information Processing Letters*, 110(7):257–260, 2010.

[21] Ben Basat Ran, Gil Einziger, Roy Friedman, and Yaron Kassner. Heavy hitters in streams and sliding windows. In *IEEE INFOCOM 2016 - the IEEE International Conference on Computer Communications*, pages 1–9, 2016.

[22] 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*, pages 290–297, 2006.

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

[24] Junzhi Gong, Tong Yang, Haowei Zhang, Hao Li, Steve Uhlig, Shigang Chen, Lorna Uden, and Xiaoming Li. Heavykeeper: An accurate algorithm for finding top-k elephant flows. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 909–921, Boston, MA, 2018. USENIX Association.

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

[26] J. Aguilar-Saborit, P. Trancoso, V. Muntes-Mulero, and J. L. Larriba-Pey. Dynamic count filters. *Acm Sigmod Record*, 35(1):26–32, 2006.

[27] Fang Hao, M Kodialam, T. V Lakshman, and Haoyu Song. Fast multiset membership testing using combinatorial bloom filters. In *INFOCOM*, pages 513–521, 2009.

[28] Tong Yang, Alex X. Liu, Muhammad Shahzad, Yuankun Zhong, Qiaobin Fu, Zi Li, Gaogang Xie, and Xiaoming Li. A shifting bloom filter framework for set queries. *Proceedings of the Vldb Endowment*, 9(5):408–419, 2016.

[29] Tong Yang, Yang Zhou, Hao Jin, Shigang Chen, and Xiaoming Li. Pyramid sketch: a sketch framework for frequency estimation of data streams. *Proceedings of the Vldb Endowment*, 10(11), 2017.

[30] Yang Zhou, Tong Yang, Jie Jiang, Bin Cui, Minlan Yu, Xiaoming Li, and Steve Uhlig. Cold filter: A meta-framework for faster and more accurate stream processing. In *Proceedings of the 2018 International Conference on Management of Data*, SIGMOD '18, pages 741–756, New York, NY, USA, 2018. ACM.

[31] Pratanu Roy, Arijit Khan, and Gustavo Alonso. Augmented sketch: Faster and more accurate stream processing. In *International Conference on Management of Data*, pages 1449–1463, 2016.

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

[33] Erik D Demaine, Alejandro López-Ortiz, and J Ian Munro. Frequency estimation of internet packet streams with limited space. In *European Symposium on Algorithms*, pages 348–360. Springer, 2002.

[34] Gurmeet Singh Manku and Rajeev Motwani. Approximate frequency counts over data streams. In *VLDB'02: Proceedings of the 28th International Conference on Very Large Databases*, pages 346–357. Elsevier, 2002.

[35] Ahmed Metwally, Divyakant Agrawal, and Amr El Abbadi. Efficient computation of frequent and top-k elements in data streams. In *International Conference on Database Theory*, pages 398–412. Springer, 2005.

[36] Daniel Ting. Data sketches for disaggregated subset sum and frequent item estimation. 2017.

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

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

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

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

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

[42] Arvind Arasu and Gurmeet Singh Manku. Approximate counts and quantiles over sliding windows. In *ACM Sigmod-Sigact-Sigart Symposium on Principles of Database Systems*, pages 286–296, 2004.

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