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

Tong Yang, Xiangyang Gou, Long He, Yinda Zhang, Ke Wang, Xilai Liu, Bin Cui
Peking University, China

***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. Our key idea is to *introduce time zones to sketches for the first time, and we achieve that for a query at any time, from multiple time zones we can get at least one accurate result, which has a small jet lag.* 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].**

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

I. Introduction

A. 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 streams 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 items 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.

B. 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 $\lambda-$sampling algorithm [20], the window compact space saving (WCSS) [21] and so on [22].

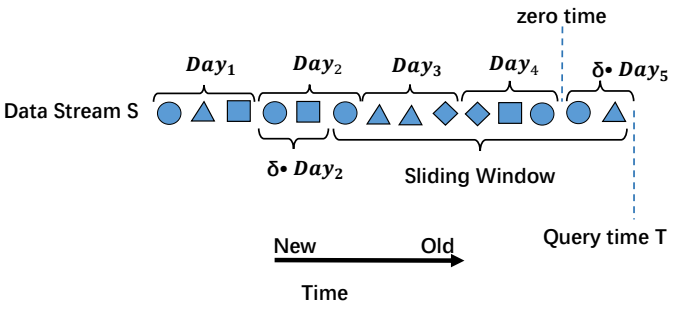


Fig. 1. Illustration of the sliding window and the Days

However, existing algorithms have two main limitations. First, these algorithms 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 handles one specific query in sliding windows. However, in applications various kinds of queries are usually needed, which makes a general framework more preferred.

C. 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 VIII.

In the following, first, we introduce the common sketch model. Second, we use a classic method to make this model support deletions of out-dated information. Third, we explain our key observation. Fourth, based on the observation, we present our key technique. Fifth, we explain why this technique works. Finally, we give a summary of our experimental results.

1) Sketch model: A typical sketch uses m buckets, which are 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 in these buckets. We call these k buckets **the k mapped buckets**. For queries, the sketch returns the most accurate value in the k mapped buckets.

2) Deleting out-dated information: We split the data stream into small periods. The length of each period is $\frac{1}{d-1}$ of the sliding window. We call each period a **Day**. In each bucket, we use d fields to record the most recent d periods, in order to estimate the information in the sliding window. Whenever a field is out-dated, we delete it. This method is classic and intuitive, and has been used in many prior works.

Take an example in Figure 1. We set $d = 4$. Each Day is $\frac{1}{3}$ of the sliding window. We record the most recent 4 Days, *i.e.*, $Day_2 \sim Day_5$. Notice Day_5 is not a complete Day, and only $\delta(0 \leqslant \delta < 1)$ of it has passed at the query time. For convenience, we call δ the **jet lag**.

3) Observation: We observe that when the query time is close to the nearest **zero time**, *i.e.*, the start time of the nearest Day, the d Days we record will be very close to the sliding window. In other words, we record additional δ of the oldest Day, which is out-dated. The smaller δ is, the higher accuracy we achieve. In prior works, the entire data structure works synchronously, which means all the buckets always have the same jet lag. This makes the accuracy of the data structure fluctuates according to the query time. In the worst case, the error can be $\frac{1}{d-1}$ of the sliding window plus the errors of hash collisions. Note that in the sketch model mentioned above, each item is mapped to k buckets, and the most accurate value will be reported in queries. We come up with an idea: *is it possible to make different buckets asynchronous and achieve that for each item there is at least one mapped bucket with a small jet lag?*.

4) Our Key technique: Based on the above observation, we use a scanning pointer to scan all the buckets one by one, again and again at a constant speed. When the scanning pointer points to a bucket, we call this time the **zero time** of the bucket. In this way, the zero time of different buckets is asynchronous. At the zero time, we delete the oldest field, and insert a new field into the bucket as the unique active field. It means a new Day just started.

The above process is just like the sun and Earth. The array/Earth is split into many buckets/time zones. Every time when the scanning pointer/sun reaches a bucket/time zone, a new Day starts, and the out-dated information is deleted. Every two buckets/time zones have different time.

5) Why our framework works? The k mapped buckets of an item are separated and distributed in the k equal-sized segments/*continent*. It is like that the item is mapped to k time zones, one in a continent. Whenever we query, there must be a continent which has just passed the zero time. The bucket in this 'continent' has a small enough jet lag, to be precise, less than $\frac{2}{k}$ (proofs are provided in the technical report [1] due to space limitation). In this way, the error is bounded to be $\frac{2}{k\times(d-1)}$ of the sliding window. In contrast, existing related algorithms use the same zero time for all buckets, and thus the error can only be bounded as $\frac{1}{d-1}$ of the sliding window.

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.

D. 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 sliding window models.

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 and show that the Sliding sketch achieves much higher accuracy than the state-of-the-art.

II. 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.

A. 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.

1) Sketches for Membership Queries:

Membership query is to check if an item e is in a set s or not. The most well-know 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) 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 bigger 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.

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 on 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] and so on.

B. 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 works has high accuracy with limited memory. Moreover, most of them are specific to a limited kinds of queries.

III. Problem Definition

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

A. 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 $S = \{e_1^{t_1}, e_2^{t_2}, e_3^{t_3} ... e_i^{t_i} ...\}$. 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.

B. 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.

TABLE I
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

Either the time-based or count-based sliding windows contain only the most recent items and ignore the old ones. In this paper, we mainly discuss the count-based sliding windows, but our algorithm can be easily extended to time-based sliding windows. [1]

C. 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.

There have been several prior algorithms supporting these queries in data streams, like the Bloom filter which supports membership queries, the CM sketch, the CU sketch, and the Count sketch which support frequency queries, and the Heavy-Keeper which supports heavy hitter queries. However, they can not delete out-dated information and can not be applied in sliding windows. In this paper, we propose a framework named the *Sliding sketch* which extends these algorithms to the sliding windows.

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

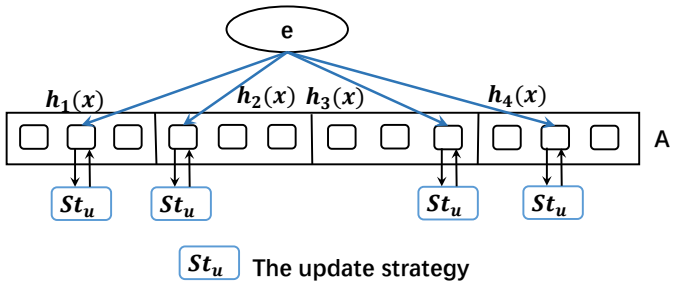


Fig. 2. Update Operation in the k-hash model

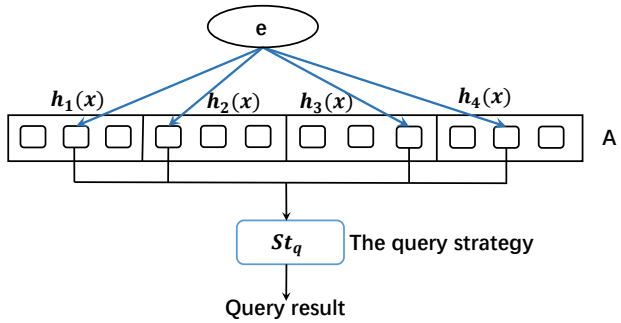


Fig. 3. Query Operation in the k-hash model

IV. 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.

A. 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.

Data structure: As shown in Figure 2 and 3, the $k-$hash model is composed of an array of k equal-sized segments, each of which is associated with a hash function. The array is composed of buckets, while each bucket contains a bit, a counter, or a key-value pair.

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 get the d mapped buckets. Then it reports a result computed

[1]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 IV-C

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.

B. Rationale

The key technique of the Sliding sketch is to introduce time zones into the sketch model. Each bucket is like a time zone, and every two adjacent buckets have the same time difference. Whenever we query, only one bucket has no jet lag, and other buckets have different jet lags. In this way, all buckets in the Sliding sketch work asynchronously. An item e is mapped to k buckets, each of which is in a segment. The use of segments in existing sketches provides an opportunity to guarantee that at least one mapped bucket has a small enough jet lag at any query time. Our method is to use a novel operation, namely the **scanning operation** to guarantee that one of the mapped bucket has a small jet lag.

C. 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, which are divided into k equal-sized segments. 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 the most recent small 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. We use these the information in these 2 Days to estimate the sliding window. In Section V-B, 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.

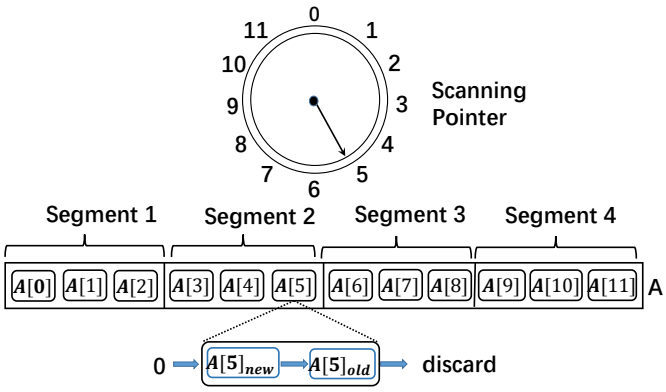


Fig. 4. Example of the scanning operation

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. Every time when the scanning pointer arrives at a bucket B, it is 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**.

Next we give an example of the scanning operation:

Example 1:

An example of the scanning operation is shown in Figure 4. In this figure we show the scanning pointer as a ring to make it easy to understand. In this example, A is an array with length 12 and 4 segments. The scanning pointer goes through all the 12 buckets cycle by cycle. In this figure, it arrives at bucket $A[5]$, and we delete the value in $A[5]^{old}$. Then we move the value in $A[5]^{new}$ to $A[5]^{old}$, and set $A[5]^{new}$ to 0.

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

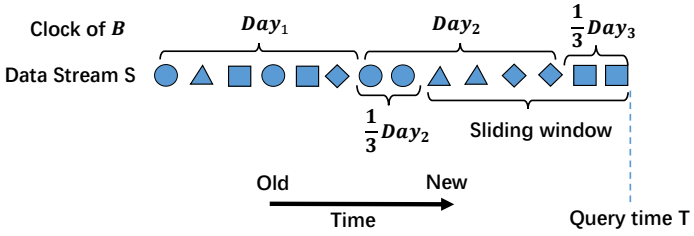


Fig. 5. Example of the sliding window and the Days

as the Bloom filter, the CM sketch, and the CU sketch. More strategies will be discussed in Section V-A.

D. 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. Because in the basic version, each Day is equal to the length of the sliding window, query in the sliding window is equal to query in the last Day before query time T. At query time T, only $\delta(0 \leqslant \delta < 1)$ of Today has passed. 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 2:

An example of the Days in bucket B and the sliding window of the data stream is shown in Figure 5. 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:

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

Derivation of the equation will be shown in in Section VII.

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 $0 < \delta \leqslant \frac{2}{k}$, and a mapped bucket B'' where $\frac{k-2}{k} \leqslant \delta \leqslant 1$. For other $k-2$ mapped buckets, the value range is $0 < \delta \leqslant 1$. Detailed analysis is shown in Section VII.

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 VII.

V. Sliding Sketch Optimizations

A. 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:

Corrected Sum Strategy: To query an item, we compute $\{Sum(B_i) = B_i^{new} + B_i^{old} | 1 \leqslant i \leqslant k\}$ for the k mapped buckets. $Sum(B_i)$ summarizes the frequency of the items mapped to bucket B_i in the last $1+\delta$ times of the sliding windows. Therefore, we correct each $Sum(B_i)$ as $Sum(B_i) = \frac{Sum(B_i)}{1+\delta_i}$ where δ_i is computed according to equation 1. Then we use the query strategy St_q of the specific sketch to get the result from these k values. This strategy can be applied when the following assumption holds:

Definition 7: Stable stream assumption: We assume that in the data stream each item arrives at a constant speed. In other words, in the data stream for any two windows $W1$ and $W2$ with the same length, each item e has similar frequencies in them.

This strategy can return more accurate results when the *stable stream assumption* holds. However, if we use this strategy when applying the Sliding sketch technique to the

CM sketch or the CU sketch, even though we may get a more accurate value, we can on longer guarantee their property of one-side error. Therefore, a prudent trade-off is needed before applying this strategy.

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 only have under-estimation error or have two-side error, like the Count sketch [23] and the HeavyKeeper [24].

Corrected Under-estimation Summary: We can use δ to correct the ${B_i}^{new}$ field in the k mapped buckets when the *stable stream assumption* holds. In this strategy, we find k buckets $\{B_i|1 \leqslant i \leqslant k\}$, and get k values $\{{B_i}^{new}|1 \leqslant i \leqslant k\}$. We get k corrected results as $\{\frac{{B_i}^{new}}{\delta_i}|1 \leqslant i \leqslant k\}$ where δ_i is computed with equation 1. We use the query strategy St_q of the specific sketch to get a result from these k corrected values.

B. 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. 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 on the need of applications and the specific sketch. The strategies mentioned in Section V-A can all be used, which are implemented as follows:

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

Corrected Sum Strategy: For each queried item e we find the k mapped buckets $\{B_i|1 \leqslant i \leqslant k\}$ and compute k values $\{Sum(B_i) = \frac{\sum_{j=1}^{d} B_i^j}{1+\frac{\delta_i}{d-1}}|1 \leqslant i \leqslant k\}$ where δ_i in each bucket can be computed following equation 1. Then we use the query strategy St_q of the specific sketch to get the result from these k sums.

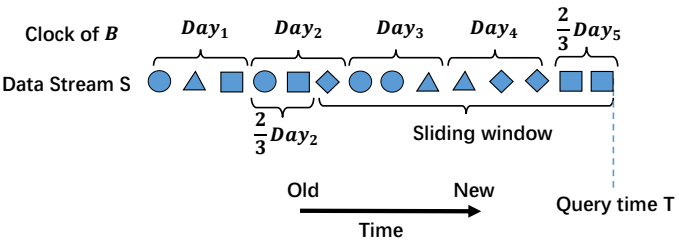


Fig. 6. Example of the sliding window and the Days

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.

Corrected 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 $\{UE(B_i) = \frac{\sum_{j=1}^{d-1} B_i^j}{1-\frac{1-\delta_i}{d-1}}|1 \leqslant i \leqslant k\}$ where δ_i in each bucket can be computed following equation 1. Then we use the query strategy St_q of the specific sketch to get the result from these k values.

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

Example 3: An example of the Days in a bucket B and the sliding window is shown in Figure 6. 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.

VI. 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. Apply to Bloom filters

1) Standard Bloom filters:

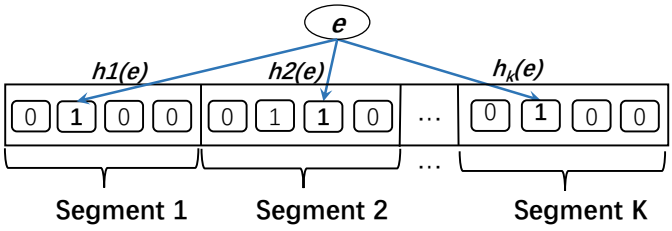


Fig. 7. Structure of the partition Bloom filter

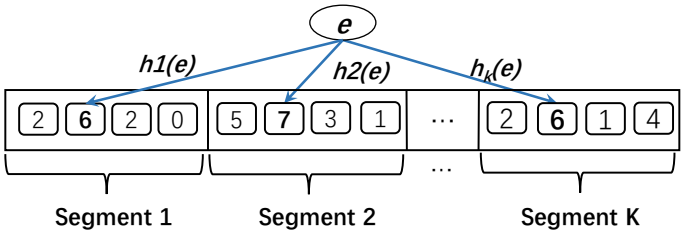


Fig. 8. Structure of the CM sketch

There are 2 versions of Bloom filters [9], the partition version and the standard version. They have similar error rate [37], and the partition Bloom filter is more popular because it is suitable for parallel implementations [38]. 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 7.

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 reports false, otherwise we reports true.

The Bloom filter has the property of one-side error. It only has false positives, but no false negatives. In other words, if an item e is in the data stream, it will definitely report true, but if e is not in the set, it has a small probability to report true due to hash collisions.

2) Sliding Bloom filters:

In the Sliding Bloom filter, A is a bit array with m buckets, separated into k equal-sized segments. Each bucket B contains d bits $\{B^j|1 \leqslant j \leqslant d\}$. Initially, all the bits are set to 0.

Update Operation: When an item e arrives, we find the k mapped buckets $\{B_i|1 \leqslant i \leqslant k\}$with the k hash functions, and in each mapped bucket B_i, we set the first bit B_i^1 to 1.

Scanning Operation: In the scanning operation, a scanning pointer goes through the buckets one by one in A repeatedly. For a count-based sliding window with length N, the scanning pointer goes through $\frac{(d-1)\times m}{N}$ buckets whenever a new item arrives. When the scanning pointer arrives at a bucket B, it sets $B^j = B^{j-1}(2 \leqslant j \leqslant d)$ and $B^1 = 0$.

Query Operation: We use the sum strategy as an example. When querying an item e, we find the k mapped buckets $\{B_i|1 \leqslant i \leqslant k\}$ with the k hash functions, and computes k values $\{Sum(B_i) = \sum_{j=1}^{d} B_i^j|1 \leqslant i \leqslant k\}$. If any of them is 0, we return false. Otherwise, we return true.

B. Apply to CM sketches

1) Standard CM sketches:

The CM sketch [10] is composed of a counter array with m counters which are separated into k equal-sized segments and k hash functions $\{h_i(\cdot)|1 \leqslant i \leqslant k\}$, as shown in Figure 8. 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 call these counters the mapped counters of e for convenience. 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. The CM sketch only has over-estimation error, which means the reported frequency is no less than the true value.

2) Sliding CM sketches:

In the Sliding CM sketch, A is a counter array with m buckets, separated into k equal-sized segments. Each bucket B contains d counters $\{B^j|1 \leqslant j \leqslant d\}$. 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 $\{B_i|1 \leqslant i \leqslant k\}$ with k hash functions, one in each segment. In each mapped bucket B_i, we increase the first counter B_i^1 by 1.

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

Query Operation: We use the sum strategy as an example. When querying an item e, we find the k mapped buckets $\{B_i|1 \leqslant i \leqslant k\}$, and compute k values $\{Sum(B_i) = \sum_{j=1}^{d} B_i^j|1 \leqslant i \leqslant k\}$. We report the minimum one as the frequency.

C. Apply to Count sketches

1) Standard Count Sketches:

The Count sketch [23] has the same data structure as the CM sketch which can be referred in Figure 8, but it has another set of k hash functions $\{s_i(\cdot)|1 \leqslant i \leqslant k\}$ besides $\{h_i(\cdot)|1 \leqslant i \leqslant k\}$. Each $s_i(\cdot)$ 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 $s_i(e)$ ($s_i(e) = -1$ or 1).

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

2) Sliding Count Sketches:

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 $\{B_i|1 \leqslant i \leqslant k\}$. Then we compute another set of k hash functions $\{s_i(e)|1 \leqslant i \leqslant k\}$ which map item e to the range $\{-1, 1\}$. In each mapped bucket B_i, we update the first counter B_i^1 as $B_i^1 = B_i^1 + s_i(e)$.

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

Query Operation: We use the corrected sum strategy as an example. When querying an item e, we find the k mapped buckets $\{B_i|1 \leqslant i \leqslant k\}$, and compute k values $\{Sum(B_i) = s_i(e) \times \frac{\sum_{j=1}^{d} B_i^j}{1+\delta_i}|1 \leqslant i \leqslant k\}$ where δ_i can be computed following equation 1. Then we return the median value of them.

D. Apply to CU sketches

1) Standard CU sketches:

The CU sketch [11] has the same data structure as the CM sketch, which can be referred to Figure 8.

Update Operation: When inserting an item e, it only increases the minimum one among the k mapped counters by 1. Because the mapped counters may be larger than the accurate frequency due to hash collisions, but never smaller. We only need to increase the smallest one to make sure it is no less than the accurate frequency.

Query Operation: When querying an item e, the CU sketch reports the minimum value in the k mapped counters as the frequency.

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

2) Sliding CU sketches:

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 $\{B_i|1 \leqslant i \leqslant k\}$ with k hash functions. Then we find the bucket B' with index p' which has the smallest $Sum(B) = \sum_{j=1}^{d} B^j$ in the k mapped buckets. Suppose the scanning pointer is at the q_{th} bucket. Then we increase the first counter in B' and all the buckets B'' with a larger δ than B'. The index p'' of such a bucket B'' satisfies the following equation:

$$\begin{cases} p'' \leqslant p' \ || \ p'' > q & (p' \leqslant q) \\ q \leqslant p'' < p' & (p' > q) \end{cases} \tag{2}$$

These buckets have larger δ and summarize a longer period than B'. If the counters in these buckets have a larger value, we can not determine that they suffer from hash collisions, and we should also increase the first counter in each of them by 1. Other mapped buckets summarize a shorter period than B'. If they still have a larger counter, they must suffer from hash collisions, and we do not need to increase them.

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

Query Operation: We use the sum strategy as an example. When querying an item e, we find the k mapped buckets $\{B_i|1 \leqslant i \leqslant k\}$ with the k hash functions, and compute the k values $\{Sum(B_i) = \sum_{j=1}^{d} B_i^j|1 \leqslant i \leqslant k\}$. Then we return the minimum one as the frequency.

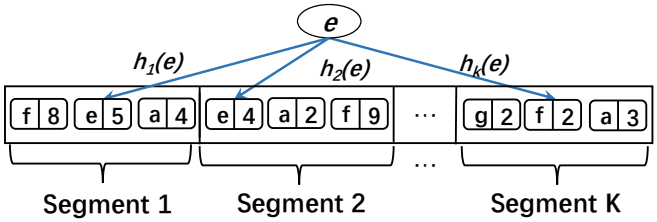


Fig. 9. Structure of the HeavyKeeper

E. Apply to HeavyKeeper

1) Standard HeavyKeeper:

The HeavyKeeper [24] is composed of an array of buckets with k segments and k hash functions, as shown in Figure 9. Each bucket is composed of an ID and a counter. All the ID 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 call these buckets the mapped buckets for convenience. We check these k mapped buckets. There are 3 kinds of situations.

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

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

3) If in a mapped bucket the ID is not e or 0, it means the 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 for 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 Heavy Hitters: When the we need to find the heavy hitters, we scan the array and find all the items with frequencies larger than the threshold.

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

2) Sliding HeavyKeeper:

In the Sliding HeavyKeeper, A is an array with m buckets which are divided into k equal-sized segments, and each bucket B contains an ID and d counters $\{B^j|1 \leqslant j \leqslant d\}$. Initially, all the ID 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 $\{B_i|1 \leqslant i \leqslant k\}\}$ with k hash functions. Then we check these k mapped buckets. There are 3 kinds of situations:

1) If in mapped bucket B_i the ID is 0, we set the ID to e and set the first counter ${B_i}^1 = 1$.

2) If in a mapped bucket B_i the ID is equal to e, we increase the first counter ${B_i}^1$ by 1.

3) If in bucket B_i the ID is not 0 nor e, the update procedure is as follows. First, we compute a sum $S(B_i) = \sum_{j=1}^{d-1} B_i^j$. Second, we find the smallest j where $B_i^j > 0$ and decrease it by 1 with probability $b^{-S(B_i)}$, where b is suggested to be 1.08. Third, if after the decrement $S(B_i) = 0$, we set the ID to e and ${B_i}^1 = 1$, ${B_i}^j = 0$ $(2 \leqslant j \leqslant d)$.

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

Query Operation: We use the strategy of under-estimation as an example. We check all the items in the Siding HeavyKeeper. For each item e, we find the k mapped buckets $\{B_i|1 \leqslant i \leqslant k\}$ with the k hash functions. Then we check the ID in these k mapped buckets. If in a mapped bucket B_i the ID is equal to e, we compute value $UE(B_i) = \sum_{j=1}^{d-1} B_i^j$. We report the maximum one among these computed values as the frequency of e. If its frequency exceeds the threshold, we report the item as a heavy hitter.

VII. 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 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.

A. 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, $\frac{(d-1)\times m}{N}$ buckets need to be scanned, the time cost of scanning buckets in every item arrival 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.

B. Analysis of the jet lag δ

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

$$\delta = \frac{q-p}{m} \ (p \leqslant q) \tag{3}$$

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

$$\begin{aligned} \delta &= \frac{m-p+q}{m} \\ &= 1 - \frac{p-q}{m} \ (p > q) \end{aligned} \tag{4}$$

2) The 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 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 $\delta \leqslant \frac{i}{k}$, $\forall 2 \leqslant i \leqslant k-1$ 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 $\delta \geqslant 1 - \frac{i}{k}$, $\forall 2 \leqslant i \leqslant k-1$, and there are k mapped buckets where $\delta > 0$.

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 $p' \leqslant q$, 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

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

Therefore in this bucket B' we have $0 < \delta \leqslant \frac{1}{k}$. 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

$$\delta = 1 - \frac{p''-q}{m} \quad \geqslant 1 - \frac{\frac{2\times m}{k}}{m} = 1 - \frac{2}{k} \tag{6}$$

Therefore in this bucket B'' we have $1 - \frac{2}{k} \leqslant \delta < 1$. 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 forms a arithmetic sequence, which is : $\{[\frac{j-1}{k}, \frac{j+1}{k}]|1 \leqslant j \leqslant k-2\}$.

2) When $p' > q$, 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

$$\delta = 1 - \frac{p'-q}{m} \quad \geqslant 1 - \frac{\frac{m}{k}}{m} = 1 - \frac{1}{k} \tag{7}$$

Therefore, in this bucket B', we have $\frac{k-1}{k} \leqslant \delta < 1$. 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

$$\delta = \frac{q-p''}{m} \quad \leqslant \frac{\frac{m}{k}}{m} = \frac{2}{k} \tag{8}$$

Therefore, in this bucket B'', we have $0 \leqslant \delta \leqslant \frac{2}{k}$. 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 a 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.

C. Analysis of the Accuracy

The accuracy of the Sliding sketch is influenced by the specific sketch and the strategy we use. In this section we analysis the accuracy of the sliding Bloom filter, the sliding CM sketch and the sliding HeavyKeeper as examples.

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.

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

$$\begin{aligned} Pr_i &\geqslant 1-(1-(1-\frac{k}{m})^{n_i})^{i-1} \\ &\approx 1-(1-e^{-\frac{n_i k}{m}})^{i-1}, \quad (2 \leqslant i \leqslant k-1) \end{aligned} \tag{9}$$

and

$$\begin{aligned} Pr_k &\geqslant 1-(1-(1-\frac{k}{m})^{n_k})^{k} \\ &\approx 1-(1-e^{-\frac{n_k k}{m}})^{k} \end{aligned} \tag{10}$$

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

$$\begin{aligned} Pr &= 1-(1-(1-\frac{k}{m})^{n})^{j} \\ &\approx 1-(1-e^{-\frac{nk}{m}})^{j} \end{aligned} \tag{11}$$

where m is the length of the array and k is the number of segments. This is the probability that we can 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 $\leqslant \frac{i}{k}$, as shown in Theorem 2. These buckets record periods smaller than $1+\frac{i}{(d-1)\times k}$ sliding windows. Therefore we have

$$\begin{aligned} Pr_i &\geqslant 1-(1-(1-\frac{k}{m})^{n_i})^{i-1} \\ &\approx 1-(1-e^{-\frac{n_i k}{m}})^{i-1}, \quad (2 \leqslant i \leqslant k-1) \end{aligned} \tag{12}$$

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

$$\begin{aligned} Pr_k &\geqslant 1-(1-(1-\frac{k}{m})^{n_k})^{k} \\ &\approx 1-(1-e^{-\frac{n_k k}{m}})^{k} \end{aligned} \tag{13}$$

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. In each mapped bucket B_i of item e in the Sliding CM sketch, $Sum(B_i) = \sum_{j=1 dB_i^j}$ 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 $\hat{f} \leqslant (1+\frac{i}{k\times(d-1)})f+(1+\frac{i}{k\times(d-1)})\epsilon N, \ (1 \leqslant i \leqslant k)$ where N is the length of the sliding window and $\epsilon = \frac{ke}{m}$, we have

$$\begin{aligned} Pr_i &= Pr(\hat{f} \leqslant (1+\frac{i}{k\times(d-1)})f + (1+\frac{i}{k\times(d-1)})\epsilon N) \\ &\geqslant 1-e^{-i+1}, \quad (2 \leqslant i \leqslant k-1) \end{aligned} \tag{14}$$

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

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

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

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

$$Pr(\hat{f} \leqslant f + \epsilon N) \geqslant 1 - e^{-j} \tag{17}$$

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

For $2 \leqslant i \leqslant (k-1)$, there are at least $i-1$ buckets where $\leqslant \frac{i}{k}$, as shown in Theorem 2. These buckets record periods which are $1 \sim 1 + \frac{i}{(d-1)\times k}$ sliding windows, in other words they record the most recent $N \sim (1 + \frac{i}{(d-1)\times k}) \times N$ items. Therefore we have

$$Pr_i = Pr(\hat{f} \leqslant (1+\frac{i}{k\times(d-1)})f + (1+\frac{i}{k\times(d-1)})\epsilon N) \geqslant 1 - e^{-i+1},\quad (2 \leqslant i \leqslant k-1) \tag{18}$$

There are k buckets which record periods which are $1 \sim 2$ sliding windows. In other words they record the most recent $N \sim (1 + \frac{1}{d-1}) \times N$ items. Therefore we have

$$Pr_k = Pr(\hat{f} \leqslant (1+\frac{1}{d-1})f + (1+\frac{1}{d-1})\epsilon N) \geqslant 1 - e^{-k} \tag{19}$$

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. In each mapped bucket B_i of item e in the Sliding CM sketch, $UE(B_i) = \sum_{j=1} d - 1B_i^j$ 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 dereased. 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 $\hat{f} \geqslant (1 - \frac{i}{k\times(d-1)})f + (1 - \frac{i}{k\times(d-1)})\epsilon N,\ (1 \leqslant i \leqslant k)$ where N is the length of the sliding window and ϵ is any small positive number, we have

$$Pr_i = Pr(\hat{f} \geqslant (1-\frac{i}{k\times(d-1)})f - (1-\frac{i}{k\times(d-1)})\epsilon N) \geqslant 1 - (\frac{k}{\epsilon m f(b-1)})^{i-1},\quad (2 \leqslant i \leqslant k-1) \tag{20}$$

$$Pr_k = Pr(\hat{f} \geqslant (1-\frac{1}{d-1})f - (1-\frac{1}{d-1})\epsilon N) \geqslant 1 - (\frac{k}{\epsilon m f(b-1)})^{k} \tag{21}$$

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

In the Sliding HeavyKeeper we analyze the frequencies of the heavy hitters. As heavy hitters have very high frequencies, the mapped buckets of a heavy hitter are all occupied by it in most cases. Therefore, in the following queries we assume that when querying a heavy hitter e, the k mapped buckets of e are all occupited by it. In [24], the author has proved that when querying for an item e with the HeavyKeeper, in each segment we have:

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

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. 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:

$$Pr(\hat{f} \geqslant f + \epsilon N) \geqslant 1 - (\frac{k}{\epsilon m f(b-1)})^{j} \tag{23}$$

In this following part we analyze the accuracy of the Sliding HeavyKeeper with these results. We use Pr_i to represent probability that $\hat{f} \geqslant (1-\frac{i}{k\times(d-1)})f + (1-\frac{i}{k\times(d-1)})\epsilon N,\ (1 \leqslant i \leqslant k)$ where N is the length of the sliding window and ϵ is any small positive number, f represents the accurate frequency of the queried heavy hitter, and $\hat{f}$ represents the reported frequency.

For $2 \leqslant i \leqslant (k-1)$, there are at least $i-1$ buckets where $\geqslant 1 - \frac{i}{k}$, as shown in Theorem 3. These buckets record periods which are $1 - \frac{1-\delta}{d-1} \sim 1$ sliding windows, in other words they

record the most recent $(1-\frac{1-\delta}{d-1})N \sim N$ items. Therefore we have

$$\begin{aligned} Pr_i &= Pr(\hat{f} \geqslant (1-\frac{i}{k\times(d-1)})f - (1-\frac{i}{k\times(d-1)})\epsilon N) \\ &\geqslant 1-(\frac{k}{\epsilon m f(b-1)})^{i-1}, \quad (2 \leqslant i \leqslant k-1) \end{aligned} \tag{24}$$

There are k buckets which record periods which are $1-\frac{1}{d-1} \sim 1$ sliding windows. In other words they record the most recent $(1-\frac{1}{d-1}) \times N \sim N$ items. Therefore we have

$$\begin{aligned} Pr_k &= Pr(\hat{f} \geqslant (1-\frac{1}{d-1})f - (1-\frac{1}{d-1})\epsilon N) \\ &\geqslant 1-(\frac{k}{\epsilon m f(b-1)})^{k} \end{aligned} \tag{25}$$

VIII. 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. After introducing the experimental setup in Section VIII-A, we provide results of Membership Query (Section VIII-B), Frequency Query (Section VIII-C) and Heavy Hitter Query (Section VIII-D) separately.

A. Experimental Setup

Datasets:

1) IP Trace Dataset: IP trace dataset contains anonymized IP trace streams collected in 2016 from CAIDA [39]. Each item is identified by its source IP address (4 bytes).

2) Web Page Dataset: We download Web page dataset from the website [40]. 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 [41]. 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 [42], an open source performance testing tool, we generate the synthetic dataset, which follows the Zipf [43] distribution. This dataset has 32M items. The length of each item is 4 bytes. The skewness of this dataset is 1.5.

Implementation: We have implemented all the algorithms in C++ and made them open sourced [1]. The hash functions are implemented from the 32-bit Bob Hash (obtained from the open source website [44]) with different initial seeds. All of the abbreviations of algorithms we use in the evaluation and their full name are shown in table II.

TABLE II
Abbreviations of algorithms in performance evaluation

Abbreviation	Full name
Sl-BF	Sliding Bloom Filter
FBF	Forgetful Bloom Filter[15]
SW-BF	sliding window sketch[45] 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]

Computation Platform: 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:

We use the following metrics (including accuracy metrics and insertion speed) to evaluate the performance of our algorithms. 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 in different windows (after the first window), and compute the average value. 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.

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 Absolute Error (AAE) in Frequency Estimation: $\frac{1}{|\mathbf{\Psi}|}\sum_{e_i\in\mathbf{\Psi}}|f_i-\widehat{f_i}|$, where f_i is the real frequency of item e_i, $\widehat{f_i}$ is its estimated frequency, and $\mathbf{\Psi}$ is the query set. Here, we query the dataset by querying each distinct item once in the present sliding window.

3) 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 item e_i, $\widehat{f_i}$ is its estimated frequency, and $\mathbf{\Psi}$ 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: $\frac{1}{|\mathbf{\Psi}|}\sum_{e_i\in\mathbf{\Psi}}\frac{|f_i-\widehat{f_i}|}{f_i}$, where f_i is the real frequency

of item e_i, $\widehat{f_i}$ is its estimated frequency, and $\mathbf{\Psi}$ 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.

B. Evaluation on Membership Query

Parameter Setting:

We compare 3 approaches: Sl-BF, FBF, and the SW-BF. 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 buckets. For our Sl-BF, we set $d = 2$. For FBF, the parameters are set according to the recommendation of the authors. For SW-BF, we use 3 hash functions in each Bloom filter and 2 levels. 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. Details of the algorithm can be seen in the original paper [45]. 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 between the 2 approaches.

Error Rate (Figure 10(a)-10(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.

Insertion Speed (Figure 11(a)-11(d)): Our results show that the Insertion Speed of Sl-BF is about $2 \sim 3$ times faster than FBF. The speed of the Sliding Window Bloom filter is higher than our algorithm, but its accuracy is much poorer.

Summary: 1)The accuracy of our Sl-BF is about 30 times better than the prior arts on the same memory consumption on three real-world datasets and one synthetic dataset. 2)The insertion speed of our Sl-BF is about $2 \sim 3$ times faster than FBF on the same memory consumption on three real-world datasets and one synthetic dataset. The Sliding Window Bloom filter is faster than our algorithm, but the accuracy of Sl-BF has significant superiority.

C. 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 buckets. For our Sliding sketch, we set $k = 10$, $d = 2$. For ECM and SWCM, the parameters are set according to the recommendation of the authors. For each dataset, we read 100k items. We set the length of the sliding window $N = 50$k.

In the experiment we compare ARE, AAE and insertion speed between the 5 approaches.

ARE (Figure 12(a)-12(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 when the memory is set to 2MB on three real-world datasets and one synthetic dataset. The ARE of Sl-Count 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.

AAE (Figure 13(a)-13(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 Flower 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-Count 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.

Insertion Speed (Figure 14(a)-14(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 on three real-world datasets and one synthetic dataset. The insertion speed of Sl-CU is about 18.6 and 3.2 times faster than ECM and SWCM respectively when memory is set to 2MB on three real-world datasets and one synthetic dataset. The insertion speed of Sl-Count is about 20 and 3.4 times faster than ECM and SWCM respectively when memory is set to 2MB.

Summary: 1)The accuracy of our Sliding sketch (for example, Sl-CM) is about 150 and 40 times better than ECM and SWCM respectively on the same memory consumption (for example, 2MB) on three real-world datasets and one synthetic dataset. 2)The insertion speed of our Sliding sketch (for example, Sl-CM) is about 20 and 4 time faster than ECM and SWCM respectively on the same memory consumption (for example, 2MB) on three real-world datasets and one synthetic dataset.

D. Evaluation on Heavy Hitter Query

Parameter Setting:

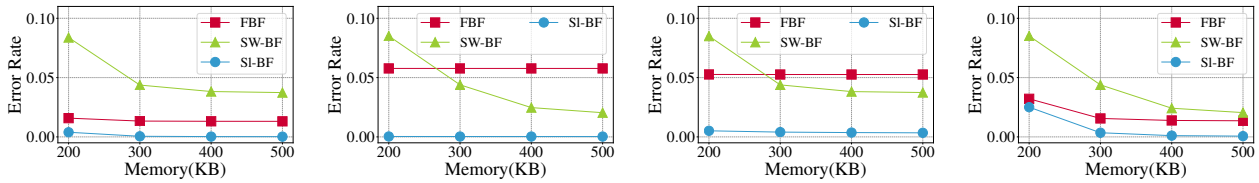
We compare 3 approaches: Sl-HK, λ-sampling and WCSS.

Let k be the number of hash functions, let d be the number of fields in each buckets. 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.

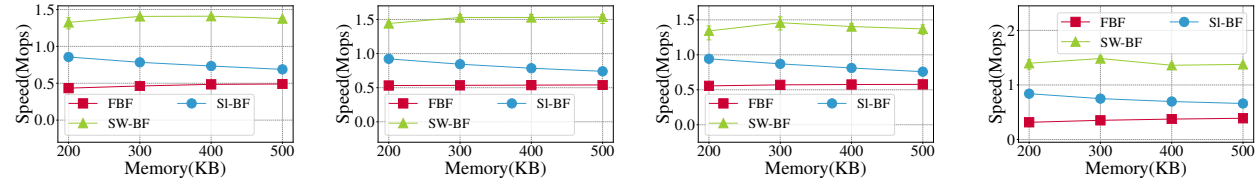
In the experiment we compare precision rate, recall rate, ARE and insertion speed between the 3 approaches.

Precision Rate and Recall Rate (Figure 15(a)-16(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. But for λ-sampling, only after memory size is more than 160KB, will



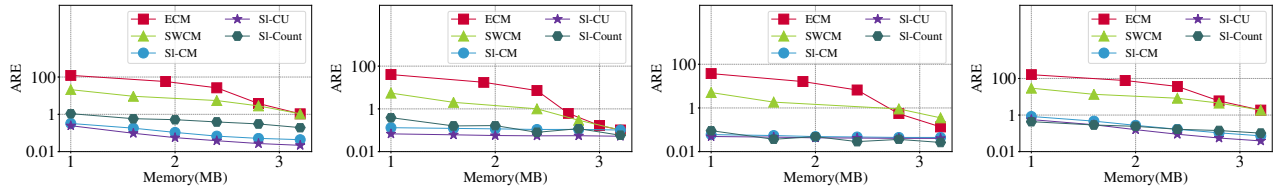
(a) IP trace. (b) Web page. (c) Network dataset. (d) Synthetic dataset.

Fig. 10. Error Rate of Membership Query.



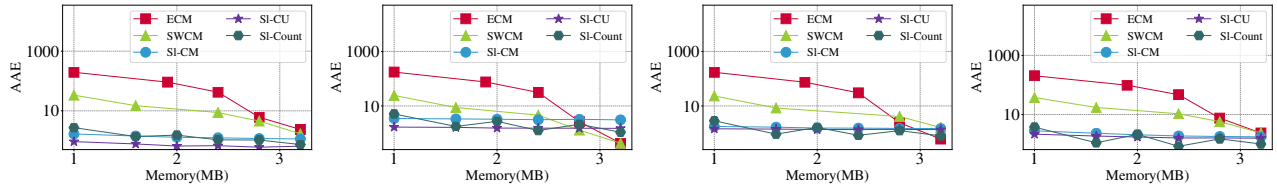
(a) IP trace. (b) Web page. (c) Network datset. (d) Synthetic dataset.

Fig. 11. Insertion Speed of Membership Query.



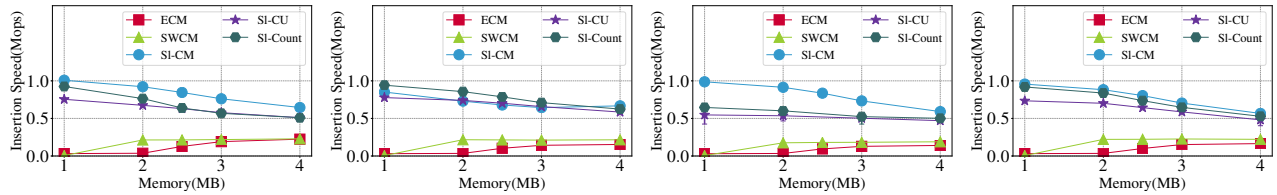
(a) IP trace (b) Web page (c) Network dataset (d) Synthetic dataset

Fig. 12. ARE of Frequency Query.



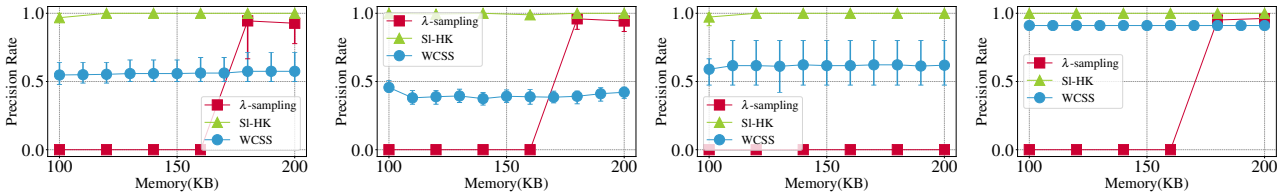
(a) IP trace (b) Web page (c) Network dataset (d) Synthetic dataset

Fig. 13. AAE of Frequency Query.



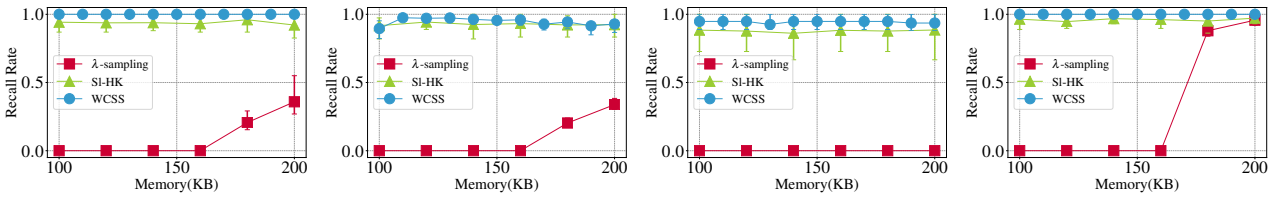
(a) IP trace (b) Web page (c) Network dataset (d) Synthetic dataset

Fig. 14. Insertion Speed of Frequency Query.



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

Fig. 15. Precision Rate of Heavy Hitter Query.



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

Fig. 16. Recall Rate of Heavy Hitter Query.

λ-sampling start to work; Before then, the precision rate and recall rate is 0%. 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 on three real-world datasets when memory is set to 200KB.

ARE (Figure 17(a)-17(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 18(a)-18(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.

IX. Experiments on Parameters

In this part we implement experiments to analyze the influence of different parameters in the Sliding sketch, *i.e.*, k and d. The experimental setup and evaluation metrics are the same as Section VIII.

1).Sensitivity Analysis on Membership Query

In the experiment we observe the impact of the number of hash functions in each buckets on Error Rate of the Sl-BF. We use IP trace dataset in this experiment. The length of the sliding window is set $N = 100k$.

`Impact of the number of hash functions on Error Rate (Figure 19):` We change the memory size from 200KB to 500KB. Let d be the number of fields in each buckets, for our Sl-BF, we set $d = 2$. We observe that the best parameter setting of the number of hash functions is $k = 10$ when we set memory from 300KB to 500KB on Sl-BF. When we set memory to 200KB, the best parameter setting of the number of hash functions is $k = 8$.

2).Sensitivity Analysis on Frequency Query

In the experiment we observe the impact of the number of fields in each buckets and the number of hash functions on ARE of the frequency query. We use IP trace dataset in this experiment. The length of the sliding window is set $N = 50k$.

`Impact of the number of fields in each bucket on ARE (Figure 20(a)-20(c)):` Let k be the number of hash functions, for our Sliding sketch, we set $k = 10$. We observe that the best parameter setting of the number of fields in each buckets is $d = 2$ when we set memory between 1MB and 5MB on Sl-CM, Sl-CU and Sl-Count.

`Impact of the number hash functions on ARE (Figure 21(a)-21(c)):` Let d be the number of fields in each buckets, for our Sliding sketch, we set $d = 2$. We observe that the best parameter setting of the number of hash functions is $k = 10$ when we set memory to 3MB or 5MB on Sl-CM and Sl-CU . When we set memory to 1MB, the ARE is more than 10%, which means 1MB memory is too small for Sliding Sketch to work on the ideal status under our experiment setting. For Sl-Count, when we set memory to 5MB, the best parameter setting is $k = 8$.

`Summary:` For our experimental parameter setting (the length of the sliding window $N = 50$k), the number of fields in each buckets is recommended to $d = 2$, the number of hash functions is recommended to $k > 10$, and the memory size is about 3MB$\sim$5MB.

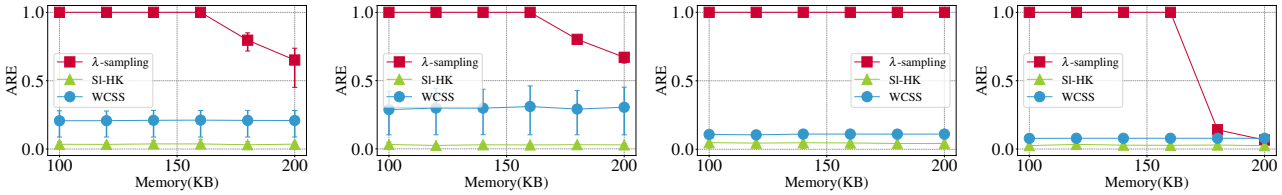
3).Sensitivity Analysis on Heavy Hitter Query

In the experiment we observe the impact of the number of fields in each buckets 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 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 buckets on Precision Rate (Figure 22(a)):` We observe that the impact of d on precision rate is little. The precision rate can always reach nearly 100% when changing d to different values.

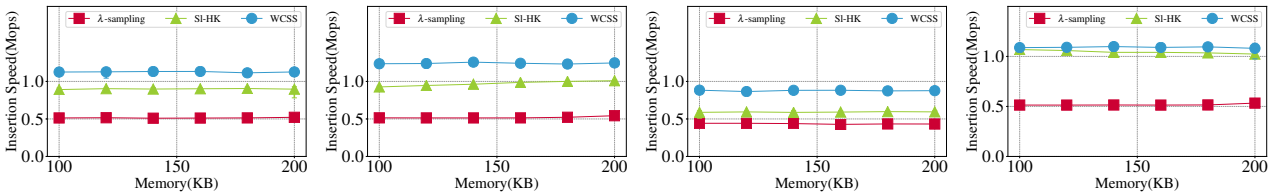
`Impact of the number of fields in each buckets on Recall Rate (Figure 22(b)):` We observe that the best parameter setting of the number of fields in each buckets is $d = 4$ when the memory is set to 20KB. When we increasing the memory size, a larger d can provide a higher recall rate.

`Impact of the number of fields in each buckets on ARE (Figure 22(c)):` We observe that the best parameter setting of the number of fields in each buckets is $d = 4$ when the memory is set to 20KB. When we



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

Fig. 17. ARE of Heavy Hitter Query.



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

Fig. 18. Insertion Speed of Heavy Hitter Query.

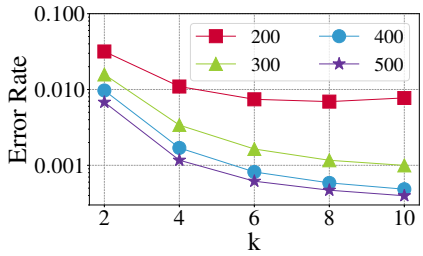


Fig. 19. Varying the number of hash functions in SI-BF .

increasing the memory size, a larger d can provide a lower ARE.

`Impact of the number of fields in each buckets on Insertion Speed (Figure 22(d))`: We observe that the insertion speed will decrease when we increasing d. For different memory size, the insertion speed is very close.

`Summary`: For our experimental parameter setting (the length of the sliding window $N = 1$M), the number of fields in each buckets is recommended to $d = 4$, and the memory size is about 100KB.

X. Conclusion

Data stream processing in sliding windows is an important and challenging work. While existing works use 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 Gyurk, 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.

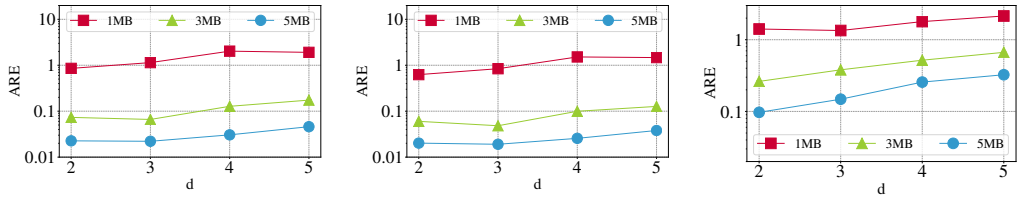
[9] Burton H Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, 1970.

[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.

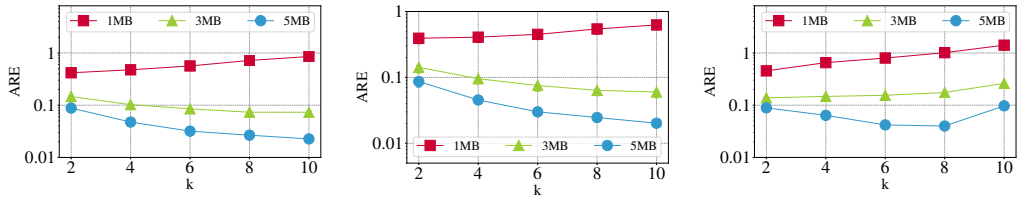
[12] Mayur Datar, Aristides Gionis, Piotr Indyk, and Rajeev Motwani. Maintaining stream statistics over sliding windows. *Siam Journal on Computing*, 31(6):1794–1813, 2002.

[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.



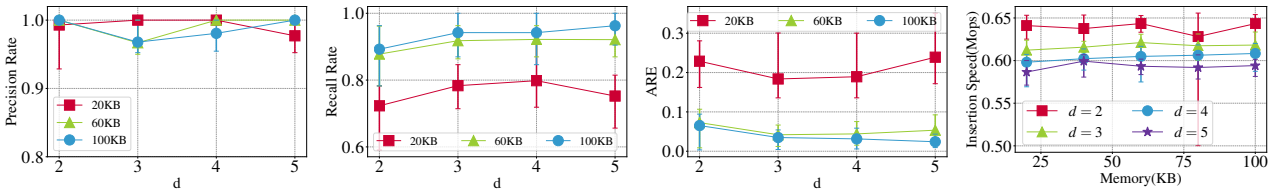
(a) Sl-CM (b) Sl-CU (c) Sl-Count

Fig. 20. Varying the number of fields in each buckets of Sliding sketch.



(a) Sl-CM (b) Sl-CU (c) Sl-Count

Fig. 21. Varying the number of hash functions in Sliding sketch.



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

Fig. 22. Varying the number of the field in each buckets of HeavyKeeper.

[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] Michael Mitzenmacher. Network applications of bloom filters: A survey. *Internet Mathematics*, 1(4):485–509, 2004.
[38] Wu Chang Feng. Blue : A new class of active queue management algorithms. *Tech Rep*, 18(3):298–312, 1999.
[39] The caida anonymized 2016 internet traces. http://www.caida.org/data/overview/.
[40] Real-life transactional dataset. http://fimi.ua.ac.be/data/.
[41] The Network dataset Internet Traces. http://snap.stanford.edu/data/.
[42] Alex Rousskov and Duane Wessels. High-performance benchmarking with web polygraph. *Software: Practice and Experience*, 34(2):187–211, 2004.
[43] David MW Powers. Applications and explanations of Zipf's law. In *Proc. EMNLP-CoNLL*. Association for Computational Linguistics, 1998.
[44] Hash website. http://burtleburtle.net/bob/hash/evahash.html.
[45] 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.