HoppingSketch: More Accurate Temporal Membership Query and Frequency Query

Zhuochen Fan, Yubo Zhang, Siyuan Dong, Yi Zhou, Fangyi Liu, Tong Yang, *Member, IEEE*
Steve Uhlig, and Bin Cui, *Senior Member, IEEE*

Abstract—Nowadays, research on temporal membership queries is indispensable. Generally, temporal membership queries exist in two modalities: fixed windows and sliding windows, the latter having obvious advantages. The first sketch that implements temporal membership queries is the persistent Bloom filter (PBF). PBF has two shortcomings: it does not support sliding windows nor frequency queries. Here, we propose HoppingSketch to promote the original PBF. It is the first sketch that implements temporal membership queries for sliding windows. HoppingSketch is a general and efficient data stream processing framework, able to implement different tasks thanks to different atomic sketches. When the atomic sketches are Bloom filters and we apply them to PBF, HoppingSketch can achieve significantly higher temporal membership query accuracy than the original PBF. When the atomic sketches are sketches of Count- Min, Conservative Update, and Count, HoppingSketch can achieve more accurate frequency query than by applying PBF on the corresponding sketches. Our experimental results demonstrate the advantages of HoppingSketch compared with the state-of-the-art.

Index Terms—Temporal membership query, Sliding windows, Frequency query, Accuracy, Atomic sketch, Bloom filter, CM sketch, CU sketch, Count sketch

1 Introduction

1.1 Background and Motivation

Membership queries refer to querying whether an item occurs in a set. Among membership queries, temporal membership queries play an increasingly important role. We first introduce the definition of temporal membership query. Given an item e and a time range $[t_i, t_j] \subseteq [1, T]$, a temporal membership query refers to querying whether an item e appears within time range $[t_i, t_j]$, where T is the upper bound on the time dimension. For example, the system administrator of a website wants to know whether the IP address of interest has visited the website within an specific time range [1].

Temporal membership queries exist in two modalities: fixed windows and sliding windows. Fixed windows refer to dividing the data stream into a series of windows of the same size in terms of time or number of items, and each window has independent statistics. Sliding windows refer to the statistics of the most recent time window or a certain number of most recent items [2]. In practice, sliding windows often have clear advantages over fixed windows. For example, heavy hitters detection [3] is an important task, $i.e.$, finding the items whose frequency is larger than a predefined threshold. With fixed windows, heavy hitters will easily be unreported. Sliding windows can handle such situations well because they are much more flexible than the fixed windows. Further, they can better describe and process temporal membership queries. However, it is more challenging to apply sliding windows to temporal membership queries, because the oldest items need to be found and cleared in time, which requires higher execution time and space costs.

- *Zhuochen Fan, Yubo Zhang, Siyuan Dong, Yi Zhou, Tong Yang and Bin Cui are with the School of Computer Science, Peking University, Beijing 100871, China. E-mail: {fanzc, zhangyubo18, dongsiyuan, chouti, bin.cui}@pku.edu.cn, yangtongemail@gmail.com*
- *Fangyi Liu is with the School of Computer Science, Beijing University of Posts and Telecommunications, Beijing 100876, China. Email: liufy@bupt.edu.cn*
- *Steve Uhlig is with the School of Electronic Engineering and Computer Science, Queen Mary University of London, London E1 4NS, U.K. Email: steve@eecs.qmul.ac.uk*
  Corresponding authors: Tong Yang. Co-primary authors: Zhuochen Fan and Yubo Zhang.

1.2 Prior Art and Limitations

The recent seminal work, persistent Bloom filter (PBF) [1], defines temporal membership and proposes a sketch (a kind of probabilistic data structure) composed of many Bloom filters. Therefore, Bloom filter is a constituent unit of PBF. In this paper, each constituent unit of a larger data structure like this is called an *atomic* sketch. Further, PBF uses a tree structure: each node in the tree represents a period of time, and a Bloom filter is used to store all items corresponding to this period of time. Its key idea is to decompose the timestamp in binary and store it in Bloom filters from leaf to root along the tree. When querying the item, PBF queries from the root to the leaves. In this way, PBF keeps a low false positive rate and low memory consumption while implementing temporal membership queries. Unfortunately, PBF has two shortcomings: First, it does not support sliding windows. Second, it does not support frequency queries, $i.e.$, reporting the number of occurrences of the given items.

1.3 Our Solution and Contributions

The main contributions of this paper are summarized as follows: We propose **HoppingSketch** to promote the original PBF. It is a novel and efficient sketch framework. By proposing the corresponding sketch-based sliding window algorithms on PBF and further converting atomic sketches, it realizes more accurate temporal membership queries and frequency queries, respectively.

Specifically, HoppingSketch overcomes two shortcomings of PBF: (1) Since PBF does not support any sliding window at present, we naturally adapt sliding window[1] to temporal membership queries for the first time through HoppingSketch, which overcomes load imbalance and significantly improves accuracy; (2) Since PBF does not support frequency queries, we design HoppingSketch as a general framework, which can flexibly implement different tasks by transforming different atomic sketches. Therefore, we only need to replace its atomic sketches with sketches of Count-Min (CM) [4], Conservative Update (CU) [5], Count (C) [6], *etc.* that support frequency queries, and we can flexibly implement frequency queries.

HoppingSketch consists of m Bloom filters and maintains k windows in each Bloom filter. The main challenge is that new items are easily added, but removing the oldest items in time is challenging under high-speed and limited memory conditions. To address the above problems, a simple idea is to clear the information of the oldest window among the m Bloom filters before allocating freed memory for the latest window w. This method needs to allocate many Bloom filters and may suffer load imbalance, and we introduce memory-sharing technique to avoid this issue, see Section 3 for more details. Based on the above methodology, HoppingSketch is the first solution that implements the temporal membership query for the sliding window, which improves the performance of PBF well. Further, we have expanded the functionality of HoppingSketch to support frequency queries. More details are provided in Section 4. Finally, we conducted extensive experiments, see Section 5 for details. The experimental results show that when HoppingSketch is applied to the PBF (PBF-H), the false positive rate (FPR) of PBF-H is reduced by between 38.7% and 47.2% than the original PBF on average. When using CM [4], CU [5] and C [6] as the atomic sketches, HoppingSketch can achieve up to 7.2, 5.6 and 2.6 times lower ARE than the PCM, PCU, and PC[2] for frequency queries, respectively. We have open-sourced all code of HoppingSketch at GitHub [7].

2 Related Work

2.1 Persistent Bloom Filter (PBF)

PBF [1] is used for time membership queries in a compact space, including two versions, PBF-1 and PBF-2. PBF-1 performs binary decomposition on the time query range, and constructs a Bloom filter (BF) for the corresponding items of each time range generated by the binary decomposition. The insertion and query processing is similar to the classic segment tree operation. PBF-2 improves on PBF-1, using only one BF in the entire decomposition level instead of using one BF in each time interval of each level. This reduces the space overhead of PBF-1, but the drawback is time instability when performing temporal membership queries.

1. Actually, we use the *hopping window* to approximate the sliding window. The difference between sliding windows and hopping windows is: sliding windows always query the past k items, where k is the window size; while hopping windows divide k items into a whole window, and insert, query, or delete in units of windows rather than items.

2. Here, PCM, PCU and PC refer to the new algorithms that we directly apply PBF to CM, CU and C for comparison in Section 5.2.

2.2 Classic sketches for Data Streams

They mainly include the Bloom filter (BF) [8], the CM sketch (CM) [4], the CU sketch (CU) [5] and the Count sketch (C) [6], *etc.* Among them, BF is designed for membership queries, while the others for frequency queries. A standard BF consists of u bits array along with v hash functions. Each bit is set to 0 at the beginning. For each incoming item, its v mapped bits are set to 1. For membership queries, BF checks its v mapped bits to see if all of them have been set to 1. CM and CU consist of Λ arrays, each array A_z $(1 \leq z \leq \Lambda)$ has Φ counters, and is associated with a hash function $h_z(.)$. When inserting an item e, CM increments the mapped counters $A_z[h_z(e)]$ by 1. CU is very similar to CM. The difference is that when inserting an item e, CU only increments the mapped counter with the minimum value by 1. Also, it does not support delete operations. C is similar to CM and CU except that each array is associated with two hash functions $h_z(.)$ and $g_z(.)$, and $g_z(.)$ maps each item to -1 or +1 with the same probability. When inserting an item e, C calculates all hash functions and adds $g_z(e)$ to the counters $A_z[h_z(e)]$ for each z. When querying an item e, C only reports the median of $A_1[h_1(e)] \times g_1(e)$, $A_2[h_2(e)] \times g_2(e)$, $\cdots$, $A_\Lambda[h_\Lambda(e)] \times g_\Lambda(e)$. Therefore, C has double-sided errors, and CM and CU have one-sided errors.

3 Basic HoppingSketch

In this section, we describe the basic HoppingSketch, which uses the Bloom filters (BFs) as the atomic sketches.

3.1 Problem Statement

Given the length K of a single window, a data stream $\mathcal{S}$ is defined as $\mathcal{S} = \{(e_1, t_1), (e_2, t_2), \ldots, (e_i, t_i), \ldots\}$, where e_i is an item belonging to the set $U = \{1, 2, \ldots, N\}$, and $t_i \in \mathbb{Z}_+$ is a monotonically increasing timestamp indicating the time item e_i occurs. The items are partitioned into windows according to their timestamps. Window $i \in \mathbb{Z}_+$ contains items at time $((i-1)K, iK]$. It is not possible to store an infinite data stream with limited memory, so all queries are about the M latest windows.

Temporal Membership Query. Given an item e from the w-th window, and $l \leq r$, we want to know whether item e appears in windows $l, l+1, \ldots, r$.

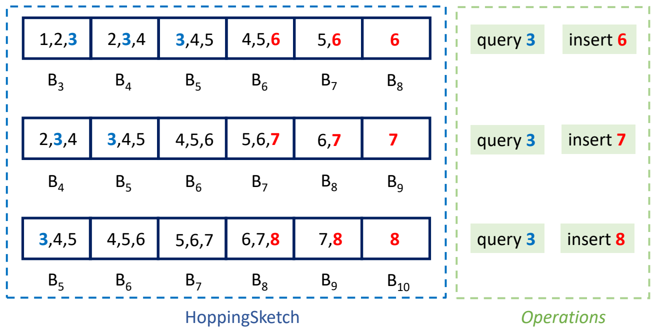


Fig. 1: The basic HoppingSketch, where the atomic sketch is Bloom filter and $m = 6, k = 3$

3.2 Data Structure

As shown in Figure 1, the data structure of basic HoppingSketch consists of m ($M \leq m \leq M + k - 1$) Bloom filters. We assume $m = M$ by default. We use B_i to denote the i-th Bloom filter created by HoppingSketch.

Instead of inserting to one Bloom filter with k hash functions, we insert items to k Bloom filters with one hash function. Each Bloom filter has a size of n bit, uses one hash function and stores items from at most k windows.

When items from a new window (call it window w) arrives, the oldest Bloom filter among m Bloom filters is removed and the freed memory is used to create a new Bloom filter B_{w+k-1}. Then, items from window w is inserted to Bloom filter $B_w, B_{w+1}, \ldots, B_{w+k-1}$.

Analysis. By inserting items to k Bloom filters, the memories are shared among k Bloom filters. Suppose the i-th window contains n_i items. The false positive rate of inserting one Bloom filter with k hash functions is about $1 - e^{-\frac{1}{m}kn_i}$, while the false positive rate of inserting k Bloom filter with one hash function is about $1-e^{-\frac{1}{m}\sum_{j=0}^{k-1} n_{i-k}}$ (m denotes the number of bits in the Bloom filter). If some n_i is significantly larger than others, our algorithm will have better false positive rate for window i queries.

3.3 Operations

3.3.1 Implementation:

The pseudocode of the new window allocation operation is shown in Algorithm 1. We use an array R of size m to store all the Bloom filters and use a variable lat to record the ID of the latest window, which is initialized to 0. The m newest Bloom filters $B_{lat+k-m}, B_{lat+k-(m-1)}, \ldots, B_{lat+k-1}$ are stored in R (assume $lat \geq m$). The Bloom filter B_k is stored at $R[k \bmod m]$.

Algorithm 1: New window allocation procedure for basic HoppingSketch

```
1 $lat \leftarrow lat + 1$
2 if $lat + k - 1 > m$ then
3     delete the oldest Bloom filter $B_{lat+k-1-m}$ stored
        at $R[(lat + k - 1) \bmod m]$
4 create Bloom filter $B_{lat+k-1}$ at
    $R[(lat + k - 1) \bmod m]$
```

3.3.2 Insertion:

The pseudocode of the insertion operation is shown in Algorithm 2. For item e from the w-th window, we insert (e, w) into $B_w, B_{w+1}, \ldots, B_{w+k-1}$. If some of them has already been removed, we just skip it. When inserting (e, w) into the i-th Bloom filter, we use the unique hash function $h_i(.)$ in the i-th Bloom filter to hash e, *i.e.*, set the bit $B_i[(h_i(e) + w) \bmod n]$ to 1, where n is the bit size of Bloom filter.

Here k insertions to Bloom filters are performed in total. So the time complexity is $O(k)$ per insertion.

Algorithm 2: Insertion procedure for basic HoppingSketch

```
  Input: An item $e$ from window $w (w \leq lat)$
1 for $i \leftarrow 0$ to $k - 1$ do
2     if $w + i \geq lat + k - m$ then
3         add $(e, w)$ to $R[(w + i) \bmod m]$
```

3.3.3 Query:

The pseudocode of the query operation is shown in Algorithm 3. For item e from the w-th window, we query the existence of (e, w) in $B_w, B_{w+1}, \ldots, B_{w+k-1}$. Report "not present" if and only if one of the queried Bloom filters reports "not present", *i.e.*, $\exists w \leq i \leq w + k - 1$, B_i has not been removed and $B_i[(h_i(e) + w) \bmod n] = 0$.

Here at most k queries to Bloom filters are performed in total. So the time complexity is $O(k)$ per query.

Algorithm 3: Query procedure for basic HoppingSketch

```
  Input: An item $e$ from window $w (w \leq lat)$
1 for $i \leftarrow 0$ to $k - 1$ do
2     if $w + i \geq lat + k - m$ then
3         query $(e, w)$ in $R[(w + i) \bmod m]$
4         if $R[(w + i) \bmod m]$ reports "not present" then
5             return "not present"
6 return "present"
```

3.3.4 Example:

As shown in Figure 1, the structure consists of $m = 6$ Bloom filters. For each Bloom filter, we label it as $B_1, B_2, \cdots, B_6$. Also, we set $k = 3$ and use one hash function in each Bloom filter.

Insertion: For item e from 8-th window, as shown, we insert $(e, 8)$ into $B_w, B_{w+1}, \ldots, B_{w+k-1}$, that is B_8, B_9, B_{10}. Then, we use the unique hash function in the Bloom filter to hash item e. For example, in Figure 1 phase 3, when we insert $(e, 8)$ into B_9, we set the bit $B_9[(h_9(e) + 8) \bmod n]$ to 1.

Query: In Figure 1 phase 3, for item e from 3-th window, we query if item $(e, 3)$ is in the window $w = 3$. Then, we query for presence in $B_w, B_{w+1}, \ldots, B_{w+k-1}$, *i.e.*, B_3, B_4, B_5. Since B_3, B_4 have been emptied (or saved to external storage) in phase 3, we just skip them and query B_5. The item $(e, 3)$ exists if the Bloom filter reports "present", *i.e.*, $B_5[(h_5(e) + 3) \bmod n] = 1$.

4 Applications

In this section, we show how to apply the HoppingSketch to existing sketches. We use the persistent Bloom filter (PBF) [1], CM [4], CU [5] and C [6] as case studies.

4.1 Temporal Membership Query

We apply HoppingSketch, described in Section 3, to the PBF-1 of PBF [1].

Data Structure: As shown in Figure 2, the data structure consists of L levels. For level $\ell \in [0, L - 1]$, we maintain

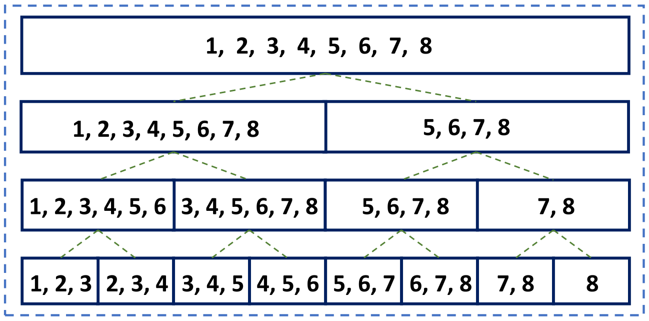


Fig. 2: HoppingSketch's application on PBF (PBF-H) and its binary tree version (connected by green dotted line).

a HoppingSketch that can store at most $M_\ell = \lfloor \frac{M-2}{K_\ell} \rfloor + 2$ windows of length $K_\ell = 2^{L-1-\ell}$. We use $W_{\ell,w}$ to denote the w-th window of level ℓ. Window $W_{\ell,w}$ is consist of K_ℓ windows from level $L-1$, i.e. window $(w-1)K_\ell+1, (w-1)K_\ell+2, \ldots, wK_\ell$.

By the definition of $W_{\ell,w}$, $W_{\ell,w} = W_{\ell+1,2w-1} \cup W_{\ell+1,2w}$ for $l = 0, 1, \ldots, L-2$. This structure can be represented by a binary tree. Each node represents a window $W_{\ell,w}$. For convenience, we call 'the node representing window $W_{\ell,w}$' by node $W_{\ell,w}$. Each node $W_{\ell,w} (\ell < L-1)$ has two children, node $W_{\ell+1,2w-1}$ and node $W_{\ell+1,2w}$. The binary tree in Figure 2 shows the structure of the PBF-H. This illustrates the connection between our algorithm and the PBF[3].

The binary tree structure is similar to a well known data structure called segment tree. In this structure, each node $W_{\ell,w}$ represents a time interval $[(w-1)K_\ell+1, wK_\ell]$ and every query interval can be decomposed into several sub-intervals which can be found in the binary tree.

Theorem 4.1. *For every query interval* $[a, b]$ $(a < b < a + M)$, *let the query result be* $W_q = \bigcup_{i \in [a,b]} W_{L-1,i}$, *there exists a set* $R_0 = \{(\ell_i, w_i)\}$ *satisfying* $W_q = \bigcup_{(\ell,w) \in R_0} W_{\ell,w}$ *and* $|R_0| \leq 2L + 4$.

Proof. Let $R_{L-1} = \{(L-1, k)\}_{k=a}^{b}$ be the initial set. R_0 is the desired set.

For $l = L-1, L-2, \ldots, 1$, let $T_l = \{w | (l, 2w-1), (l, 2w) \in R_l\}$, then $R_{l-1} = \{(l-1, w) | w \in T_l\} \cup \{(i, w) | i \neq l \vee w \notin T_l\}$.

We can prove by induction that T_l contains all items in $[\min T_l, \max T_l]$. By definition, T_{L-1} contains all items in $[\lfloor (a+1)/2 \rfloor, \lfloor b/2 \rfloor]$. If T_l contains all items in $[\min T_l, \max T_l]$, then T_{l-1} contains all items in $[\lfloor (\min T_l + 1)/2 \rfloor, \lfloor \max T_l / 2 \rfloor]$.

Hence, $\forall 1 \leq l < L, |\{(l, w) \in R_0\}| \leq 2$. Only $(l, \min T_{l+1}), (l, \max T_{l+1})$ (let $T_L = \{i | a \leq i \leq b\}$ for convenience) may be items of R_0. There are at most 4 windows in level 0, so we have $|R_0| \leq 2L + 4$. □

Denote $B_{l,w}$ as the w-th Bloom filter of level l.

When data from a new window w arrives, for those levels satisfying $w \bmod K_\ell = 1$, a new window must be allocated to them (see the previous section).

Insertion: For item e from window w, for every level $\ell = 0, 1, \ldots, L-1$, we insert a new item e to Bloom filter $B_{\ell, \lfloor (w-1)/K_\ell \rfloor + 1}$. The insertion of HoppingSketch is described in the previous section. Here L insertions to HoppingSketch are performed in total, so the time complexity is $O(Lk) = O(k \log m)$ per insertion.

Query: From Theorem 4.1, we can decompose the query interval $[x, y]$ into $O(\log M)$ sub-intervals on the binary tree. The decomposition procedure is shown below (Algorithm 4). Here L queries to HoppingSketch are performed in total, so the time complexity is $O(Lk) = O(k \log m)$ per query.

Algorithm 4: Decomposition procedure for HoppingSketch.

```
Input: Time interval $X, Y$
Output: Intervals $\{(x_i, y_i)\}_{i=1}^{k}$ s.t.
        $[X, Y] = \bigcup_{i=1}^{k} [x_i, y_i]$
1  Function decomp($x, y, X, Y$):
2      if $X = Y$ then
3          return $\{(X, X)\}$
4      $mid := \lfloor \frac{X+Y}{2} \rfloor$
5      if $Y \leq mid$ then
6          return $decomp(x, mid, X, Y)$
7      else if $X > mid$ then
8          return $decomp(mid+1, y, X, Y)$
9      else
10         return $decomp(x, mid, X, Y) \cup$
             $decomp(mid+1, y, X, Y)$
11 return s
```

4.2 Frequency Query

HoppingSketch can be used to implement frequency queries by using CM, CU and C as the atomic sketches, respectively. It can query the number of occurrences of an item in a certain time period.

Allocation & Insertion. As long as the atomic sketch supports creation (create a new sketch), deletion (delete an existing sketch) and insertion (insert an element pair (e, w) into the sketch), we can always generalize Algorithm 1 and Algorithm 2 to the new atomic sketch.

Query. The query procedure should be crafted to the task at hand. The key principle is to imitate the query procedure of atomic sketch. Most of the atomic sketches first maps e to k indices $i_1, i_2, \ldots, i_k$ by k hash functions, then obtain the query results by the k values on indices $i_1, i_2, \ldots, i_k$. For HoppingSketch, we first maps (e, w) to k indices in k atomic sketches, then obtain the query results by the k query results from atomic sketch in the same way. We show the query procedure of HoppingSketch applied to CM Sketch as an example (Algorithm 5).

5 Experimental Results

In this section, we conduct extensive experiments to evaluate the performance of HoppingSketch.

5.1 Experimental Setup

Our experimental setup includes the algorithms we compare against, the datasets used, the evaluation metrics, and default settings.

3. In short, compared with the original PBF-1 structure, each node of PBF-H maintains HoppingSketch instead of the standard Bloom filter.

Algorithm 5: Query procedure for HoppingSketch applied to CM sketch.

```
Input: An item $e$ from window $w (w \leq lat)$
1 $s := \infty$
2 for $i \leftarrow 0$ to $k-1$ do
3     if $w + i > lat - m$ then
4         $s := \min(s, R[(w+i) \bmod m].query(e, w))$
5 return s
```

5.1.1 Datasets

The following two real-world datasets and one Synthetic Dataset are used in our experiments.

- **IP Trace Dataset.** The IP Trace Dataset is a public dataset that includes anonymized IP traces from high-speed Internet Backbone links collected by CAIDA [9]. Each item contains a source IP address (4 bytes) and a destination IP address (4 bytes), 8 bytes in total.
- **WebDocs Dataset.** The WebDocs is a collection of web HTML documents built by a number of web pages [10]. Each item in the Web page dataset is 8 bytes.
- **Synthetic Datasets.** We generate the Synthetic Dataset that follows the Zipf [11] distribution using Web Polygraph [12], an opensource performance testing tool. The length of each item ID is 4 bytes.

5.1.2 Implementation

All the algorithms are implemented in C++. The hash function we adopted in these algorithms is the Bob hash [13]. All programs are run on a server with 128 GB system memory and a 18-core CPU (36 threads, Intel(R) Core(R) CPU i9-10980XE @4.00GHz). We set the number of hash functions to 3 for all algorithms, which usually gives nearly optimal performance. For every dataset, we set a memory limit for temporal membership query and frequency query. The size of atomic sketches are adjusted base on the memory limits.

Since few algorithms can perform both membership query and frequency estimation like HoppingSketch, we need to separately select the algorithms that implement membership query and frequency query together for comparison.

Temporal Membership Query: The first is the original persistent Bloom filter [1] (PBF), and we use PBF-1 version for comparison. The second is HoppingSketch applied to PBF-1 described in Section 4 (PBF-H). Therefore, for the temporal membership query task: PBF **v.s.** PBF-H.

Frequency Query: We use the following schemes for comparison: 1) PBF applied to CM [4] (PCM) **v.s.** HoppingSketch applied to PCM (PCM-H); 2) PBF applied to CU [5] (PCU) **v.s.** HoppingSketch applied to PCU (PCU-H); 3) PBF applied to C [6] (PC) **v.s.** HoppingSketch applied to PC (PC-H). Note that we only need to change the atomic sketch of PBF from BF to CM, CU, and C to efficiently implement PCM, PCU, and PC.

5.1.3 Evaluation Metrics

We choose three typical metrics, including FPR, ARE and efficiency (insertion time and query time), to measure the above tasks. In the following, we regard $\mathcal{S}$ as the data stream, and $(e, w) \in \mathcal{S}$ as item e arrived in the w-th time window at least once.

- **FPR (False Positive Rate, temporal membership query):** For every time window w, we define the false positive rate as $\mathrm{FPR}_w = \frac{1}{|E_w|}\sum_{e \in E_w} \hat{f}_{(e,w)}$. Where $E_w = \{e | (e,w) \notin \mathcal{S} \wedge \exists w', (e, w') \in \mathcal{S}\}$ and $\hat{f}_{(e,w)}$ represents the membership query result (0 or 1) of item i in the w-th time window.
- **Efficiency (Insertion Time and Query Time, temporal membership query):** We sample 100,000 random items from IP Trace Dataset to investigate the insertion and query cost by examining the amortized cost of inserting and querying one item when we vary the value of query length $|q|$.
- **ARE (Average Relative Error, frequency query):** For every window w, we define the average relative error as $\mathrm{ARE}_w = \frac{1}{|W_w|}\sum_{e \in W_w} \frac{|f_{(e,w)} - \hat{f}_{(e,w)}|}{f_{(e,w)}}$. Here, $W_w = \{e | (e,w) \in \mathcal{S}\}$, and $f_{(e,w)}$ and $\hat{f}_{(e,w)}$ represent the actual and estimated frequency of item e in the w-th time window respectively.

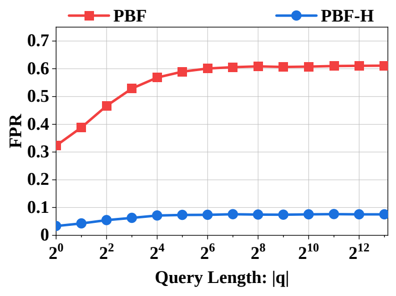
5.2 Comparison with Prior Art

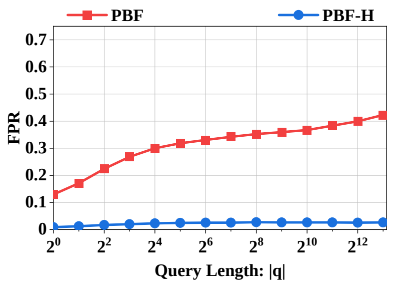
FPR *vs.* query length $|q|$ (Figure 3(a)-3(c)): *This experiment shows that the FPR of PBF-H is much lower than the original PBF. Here $|q|$ denotes the length of each query interval.* Therefore, compared with original PBF, PBF-H has obvious advantages in handling queries of longer query intervals. We find that, on the Synthetic Dataset, the FPR of PBF-H is about 47.2% lower than the original PBF. When compared in multiples, the FPR of PBF-H is about 9.6 times lower than the original PBF. On the two real-world datasets, the FPR of PBF-H is about 38.7% lower than the original PBF. When compared in multiples, the FPR of PBF-H is about 7.1 times lower than the original PBF. Specifically, the FPR of PBF-H and PBF both increase with the increase of query length $|q|$, but PBF-H grows only slightly as $|q|$ increases exponentially and stays as low as less than 7.5% even when $|q|$ is over 2^{10} on all datasets. The initial FPR of the original PBF is much larger than PBF-H, and it increases rapidly as $|q|$ increases.

Insertion time and query time *vs.* query length $|q|$ (Figure 5(a)-5(b)): *This experiment shows that the insertion time and query time of PBF-H are slightly slower than the original PBF.* Specifically, the insertion time of PBF-H is about $0.43ms$ longer than PBF, and the query time of PBF-H is around $0.3ms$ longer than PBF. Although PBF-H has no obvious advantages over the original PBF in terms of insertion time and query time, its efficiency is still acceptable.

ARE *vs.* query length $|q|$ (Figure 4(a)-4(c)): *This experiment shows that the ARE of PCM-H, PCU-H and PC-H is much lower than the corresponding PCM, PCU and PC.* The details are as follows.

1) **PCM v.s. PCM-H**. We find that, on the Synthetic Dataset, the ARE of the PCM-H is around 7.2 times lower than PCM. On the two real-world datasets, the ARE of PCM-H is about 4.5 times lower than PCM as the query length $|q|$ increases.
2) **PCU v.s. PCU-H**. We find that, on the Synthetic Dataset, the ARE of the PCU-H is around 5.6 times lower than PCU. On the two real-world datasets, the ARE of PCU-H is about 3.5 times lower than PCU as the query length $|q|$ increases.


(a) IP Trace


(b) WebDocs

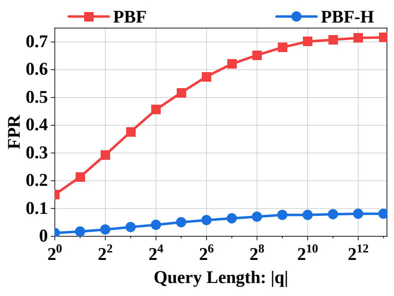
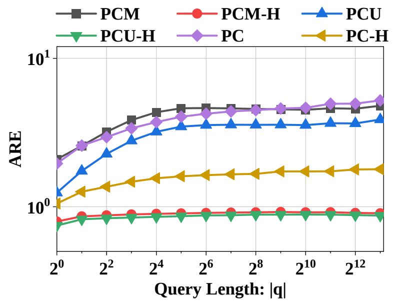
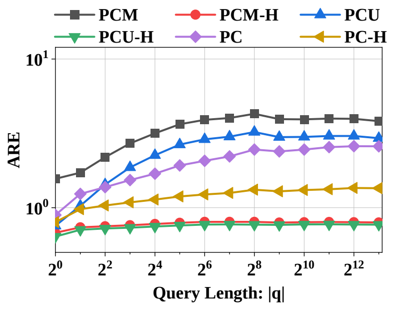

(c) Synthetic

Fig. 3: FPR **v.s.** query length $|q|$.


(a) IP Trace


(b) WebDocs

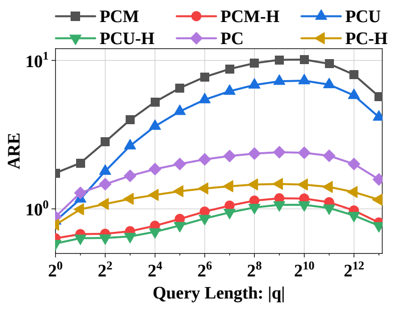
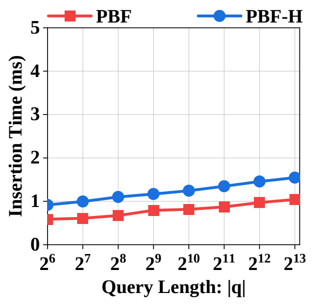

(c) Synthetic

Fig. 4: ARE **v.s.** query length $|q|$.


(a) Insertion time

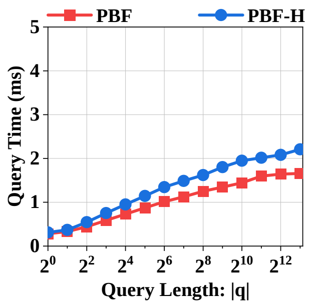

(b) Query time

Fig. 5: Efficiency v.s. query length $|q|$.

3) **PC v.s. PC-H**. We find that, on the Synthetic Dataset, the ARE of the PC-H is around 1.5 times lower than PC. On the two real-world datasets, the ARE of PC-H is about 2.2 times lower than PC as the query length $|q|$ increases.

6 Conclusion

This paper proposes a novel sketch framework HoppingSketch, which is the first sketch to implement temporal membership queries on the sliding window, and can flexibly implement different tasks by transforming different atomic sketches. When the atomic sketch is Bloom filter, it achieves significantly higher accuracy of temporal membership query than the original PBF. Further, when the atomic sketch is CM, CU or C, it achieves more accurate frequency query than the PBF's application on CM, CU and C. Our experimental results show that HoppingSketch achieves higher accuracy: the FPR of PBF-H is between 38.7% to 47.2% lower than the original PBF, and the ARE of PCM-H, PCU-H and PC-H achieves up to 7.2, 5.6 and 2.2 times lower ARE than PCM, PCU and PC as the query length $|q|$ increases, respectively.

Acknowledgment

This work is supported by Key-Area Research and Development Program of Guangdong Province 2020B0101390001, National Natural Science Foundation of China (NSFC) (No. U20A20179, 61832001).

References

[1] Y. Peng, J. Guo, F. Li, W. Qian, and A. Zhou, "Persistent bloom filter: Membership testing for the entire history," in *Proc. SIGMOD*, 2018, pp. 1037–1052.
[2] Datar, Mayur, Gionis, Aristides, Indyk, Piotr, Motwani, and Rajeev, "Maintaining stream statistics over sliding windows," *SIAM J. Comput.*, vol. 31, no. 6, pp. 1794–1813, 2002.
[3] Y. Zhang, X. Lin, Y. Yuan, M. Kitsuregawa, X. Zhou, and J. W. Yu, "Duplicate-insensitive order statistics computation over data streams," *IEEE Trans. Knowl. Data Eng.*, vol. 22, no. 4, pp. 493–507, 2010.
[4] G. Cormode and S. Muthukrishnan, "An improved data stream summary: the count-min sketch and its applications," *J. Algorithms*, vol. 55, no. 1, pp. 58–75, 2005.
[5] C. Estan and G. Varghese, "New directions in traffic measurement and accounting: Focusing on the elephants, ignoring the mice," *ACM Trans. Comput. Syst.*, vol. 21, no. 3, pp. 270–313, 2003.
[6] M. Charikar, K. Chen, and M. Farach-Colton, "Finding frequent items in data streams," in *Proc. ICALP*, 2002, pp. 693–703.
[7] "Source code related to HoppingSketch." [Online]. Available: https://github.com/pkufzc/HoppingSketch
[8] B. H. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Commun. ACM*, vol. 13, no. 7, pp. 422–426, 1970.
[9] "The CAIDA Anonymized Internet Traces." [Online]. Available: http://www.caida.org/data/overview/
[10] "Real-life transactional dataset." [Online]. Available: http://fimi.ua.ac.be/data/
[11] D. M. W. Powers, "Applications and explanations of zipf's law," *Proc. NeMLaP/CoNLL*, vol. 5, no. 4, pp. 595–599, 1998.
[12] A. Rousskov and D. Wessels, "High-performance benchmarking with web polygraph," *Software Pract. Exper.*, vol. 34, no. 2, pp. 187–211, 2004.
[13] "Hash website." [Online]. Available: http://burtleburtle.net/bob/hash/evahash.html