HoppingTimer: A Near-optimal Framework for Basic Estimation of Data Streams in Hopping Windows

Kaicheng Yang, Jianyu Wu, Tong Yang
School of Computer Science, Peking University, China

Abstract—**In high-speed data streams, recent items are often much more significant than outdated ones. Therefore, basic estimation of data streams in hopping windows is an important topic. Basic estimation tasks include cardinality estimation and membership query. There are three classic algorithms for basic tasks in fixed windows. The design goal of this paper is to devise a generic and near-optimal framework to adapt them to hopping windows. In this paper, we propose the HoppingTimer, a generic and near-optimal framework which can adapt fixed-window algorithms to time-based and count-based hopping windows for basic tasks. The key idea of HoppingTimer is to use hopping timestamps and local cleaning to clean outdated items. We apply HoppingTimer to three algorithms for basic tasks. Experimental results show that HoppingTimer is near-optimal in hopping windows, and achieves false positive rate about 1000 times lower than the state-of-the-art when using metrics of sliding-window model.**[1]

I. Introduction

A. Background and Motivation

Approximate data stream processing has been extensively applied in various fields, including anomaly/intrusion [1], [2], quality of service [3], [4] and financial data trackers [5], [6]. Among various tasks in data streams, basic estimation tasks, including cardinality estimation [7]–[10], and membership query [11], [12], have obtained the widest studies and applications. *Cardinality* refers to the number of distinct items appearing in the data stream; *Membership query* refers to whether an item appears in the data stream. In many applications, such as caching strategy [13], KV-store [14], DDos detection [15] and load balancing [16], data and queries are time-sensitive, *i.e.*, recent items are often more significant than outdated ones. This requires the basic estimation tasks to be time-sensitive.

Approximate estimation in hopping windows is time-sensitive, where hopping windows refer to consecutive windows which hop forward one step in time by a constant period. As pointed by Panes [17], when the step size gradually approximates 0, the hopping-window model becomes the sliding-window model. Therefore, we mainly focus on the hopping-window model in this paper. Estimation in hopping windows is more challenging than in fixed windows, as it needs to clean the information of outdated items out of the hopping window, which is time-consuming or memory-consuming. In this paper, we claim an algorithm in hopping windows is *optimal* when this algorithm can completely clean all outdated information of outdated items at every hop.

There are two typical models for hopping windows: 1) count-based window model (contains a certain number of items); 2) time-based window model (contains items arriving in a certain amount of time). Supporting time-based window model is more useful, important and challenging than supporting count-based window model. Count-based window model is just a special case of time-based window model, as it assumes that all items arrive at a constant interval, which is often not true in practice. Therefore, it's both desirable and challenging to support both window models with high accuracy. The *design goal* of this paper is to devise a generic and near-optimal framework which can be applied to both time-based and count-based window models, providing basic estimation of data streams in hopping windows.

B. Prior Art and Limitations

While a great many works focus on solutions for basic tasks in fixed windows, time-sensitive solutions in hopping windows are much fewer and much more challenging. Existing time-sensitive solutions can be divided into two kinds. The first kind can only address one specific task, including Timing Bloom Filter [18], Counter Vector Sketch [19], Sliding Hyperloglog [20], *etc.* The second kind can address multiple tasks. SWAMP [21]–[24] is a framework which can handle cardinality estimation, membership query and frequency estimation. However, SWAMP cannot be applied to time-based window model and is also space-consuming. In summary, no existing work can achieve the above design goal of this paper.

C. Our Solution

In this paper, we propose the *HoppingTimer*, a generic and near-optimal framework which can adapt fixed-window algorithms to time-based and count-based hopping windows for basic estimation tasks. We have two key techniques namely *hopping timestamp* and *local cleaning*.

We first explain the rationale of our key techniques. Since our goal is to devise a framework for both count-based and time-based window models, we choose to record timestamps of items in data streams. However, 64-bit timestamps take

[1] Tong Yang (yangtong@pku.edu.cn) is the corresponding author.

979-8-3503-6900-7/24/$31.00 ©2024 IEEE

up too much memory. To address this issue, we propose the *hopping timestamp*, which is essentially compressed timestamp (*e.g.*, 4-bit or 8-bit). To achieve optimal compression of timestamps, we need to guarantee that every time the hopping window hops forward one step, the hopping timestamp increments by one. Compressing timestamps could mistake outdated timestamp for correct timestamp, which will incur error, and we call this error *timestamp error* in this paper. To eliminate timestamp error, we propose two solutions namely global cleaning and *local cleaning*. Global cleaning globally traverses the data structure and cleans all items with outdated hopping timestamps every time the hopping window hops forward one step. However, the traversing process is time-consuming, so it can hardly process high-speed data streams. To address this issue, we recommend using the local cleaning. The data structure consists of lots of cells. We group these cells into many equal-sized groups. When inserting an item, existing algorithms will hash the item e into 1 or k cells, which are called *hashed cells* and the corresponding groups are called *hashed groups*. Then we perform the local cleaning: we locally traverse all cells in the hashed groups and clean outdated ones. There is no additional memory access in local cleaning because each group is set small enough to be read in a single memory access. Local cleaning could not eliminate timestamp error with a low probability, and we add one bit to the hopping timestamp to further reduce the probability of timestamp error. We theoretically prove that both the upper bound of the probability of timestamp error and the additional false positive rate caused by timestamp error in membership query are $O\left(e^{-\alpha f}\right)$, where f is the number of cells in each group, α is a positive parameter independent of f. Therefore, both timestamp error and its influence are almost negligible when f is set reasonably.

To verify the generality of our framework, we apply HoppingTimer to three algorithms for two basic estimation tasks: Bitmap [25], Bloom filter [26], [27] and Bloom filter with *hopping counters*, where hopping counter is also a technique proposed in this paper. First, the accuracy of our framework is near-optimal: our experimental results show that the average accuracy difference between our framework and the optimal accuracy is less than 0.1%. Second, as the state-of-the-art algorithms focus on the sliding-window model, we compare our framework with them using the metrics of sliding-window model. 1) For cardinality estimation, HoppingTimer achieves relative error about 1000 times lower than SWAMP; 2) For membership query, HoppingTimer achieves false positive rate about 1000 times lower than SWAMP.

II. Related Work

For measurement tasks in approximate data stream processing, there are mainly fixed-window algorithms and sliding-window algorithms. Fixed-window algorithms estimate aggregate information of consecutive disjoint windows while sliding-window algorithms care about the most recent information. Sliding window is always containing the latest items and continuously changing, so measurement on sliding-window model is much harder and more important than that on fixed-window model. In this section, we introduce prior work for fixed windows and sliding windows respectively.

A. Prior Art for Fixed Windows

Fixed-window algorithms can be classified to two kinds based on the smallest memory unit used in their data structures. In this section, we introduce both two kinds fixed-window algorithms.

Bit-based algorithms: HoppingTimer can adapt all fixed-window algorithms of this kind to hopping windows. This kind of algorithms uses bits as their smallest memory units. For example, for the membership query, the Bloom filter [26], [27] maintains a large bit array. Each arriving item e is mapped to k bits by k pairwise-independent hash functions and these k bits are set to 1. When querying an item, we check whether all the k bits associated with it have been set to 1. For each incoming item, the Bitmap [25] only maps it to 1 bit and set it to 1. When a window expires, the Bitmap counts u, the number of 0 in the bit array, using Maximum Likelihood Estimator (MLE) to estimate the cardinality. Besides, many other bit-based algorithms have been proposed, such as Compact Spreader Estimator (CSE) [28], which provides SuperSpreader estimation in network streams.

Counter-based algorithms: This kind of algorithms maintains counter arrays. For example, HyperLogLog [29] counts the number of leading 0 bits of the hash value of each item and records the maximum in counters. Then HyperLogLog uses the maximum to estimate cardinality. Counting Bloom Filter [30], which is a variant of Bloom filter, changes the bit array to a counter array, enabling it to support deletion.

B. Prior Art for Sliding Windows

SWAMP [21] is the state-of-the-art generic algorithm, supporting many sliding window tasks. SWAMP records the fingerprints of all the items in the sliding window, so its memory usage is proportional to the number of items in the sliding window. In time-based window model, the number of items in the sliding window varies with time. Since data streams can be bursty (*e.g.* network flows), SWAMP with its pre-allocated memory may not be able to record the fingerprints of all the items in the sliding window. Timing Bloom Filter (TBF) [18] supports membership queries. In order to clean outdated information, TBF changes each bit in the Bloom filter to a timestamp recording the relative time when the bit is set in the window. Sliding HyperLogLog (SHLL) [20] adopts the method of HyperLogLog to provide cardinality estimation. It uses a monotone priority queue to evict outdated information. Counter Vector Sketch (CVS) [19] changes each bit in Bitmap to a counter. The counter is set to C (a predefined constant) each time it is mapped. At set intervals, the algorithm randomly chooses a few counters and reduce them by one. In this way, CVS approximates the cardinality in the sliding window. Timestamp-vector algorithm (TSV) [31] changes the bit array in Bitmap to a 64-bit double typed timestamp vector. For each incoming item, it is mapped to a timestamp, which is set to the current time. By comparing each timestamp value with the current time, TSV can estimate the cardinality in the latest window.

All the above representative sliding window algorithms have their shortcomings. SWAMP is not applicable to time-based sliding window, which is more useful and practical. TBF, CVS, SHLL and TSV can be applied to time-based sliding window, but they are not generic, *i.e.*, they can only address one measurement task.

III. The HoppingTimer Framework

In this section, we first introduce our framework for hopping-window measurements, HoppingTimer, with two key techniques namely hopping timestamp and local cleaning. Second, we apply our framework to three typical algorithms: Bitmap, Bloom filter and Bloom filter with a counter. Third, we analyze the potential errors in HoppingTimer and how to control these errors. Fourth, we discuss that, with a proper approximation, HoppingTimer can also provide sliding-window measurements. Important notations and their meanings are demonstrated in Table I.

TABLE I: Notations used in the paper.

Notation	Meaning
$\mathcal{S}$	a data stream
e	an item in the data stream
t	arrival time of an item
W	size of hopping window
s	hopping step size
d	bit width of hopping timestamp
Ts	a hopping timestamp
m	number of cells
$C[i]$	i^{th} cell
f	number of cells per group
k	number of hash functions
$h(.)$	hash functions from items to cells
$\delta T()$	timestamp difference between two hopping timestamps

A. Preliminaries

Data streams: A data stream $\mathcal{S}$ is a sequence of items, *i.e.*, $S = \{e_1, e_2, ...\}$. Each item in $\mathcal{S}$ could occur once or more than once.

Hopping-window model (Fig. 1): Hopping windows are a series of consecutive windows which hop forward one step in time by a constant period. The disjoint time interval between two adjacent hopping window is called the **hopping step**.

Data structure model: The data structure consists of an array of cells, and is associated with one or several hash functions. Typically each cell stores a bit. Our framework can be applied to any algorithm in fixed windows conforming to this model by extending the 1-bit cells to proper bit width.

B. The HoppingTimer Framework

In this part, we introduce the HoppingTimer with its two key techniques namely **hopping timestamp** and **local cleaning**. Hopping timestamp is used to record coarse-grained temporal information of incoming items. The key idea of hopping timestamp is to compress timestamps so as to balance the memory usage and accuracy. For measurements in hopping windows, we need to clean the items out of the hopping window. To address this issue, we propose two cleaning methods: **global cleaning** and local cleaning. Global cleaning can clean all outdated information without error, but is time-consuming. Local cleaning is much faster, and can clean outdated information with small error. In HoppingTimer, we adapt local cleaning to clean outdated information.

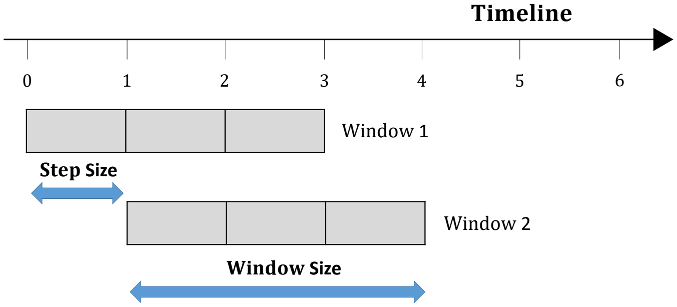


Fig. 1: Hopping windows.

Hopping Timestamp: Hopping timestamp is the compression of the 64-bit precise timestamp. It is coarse-grained but much more memory-efficient. For example, 8-bit integer is a typical instance for a hopping timestamp, which costs $1/8$ memory of the precise timestamp. A d-bit hopping timestamp ranges from 1 to $2^d - 1$. Every time the hopping window hops forward one step, the hopping timestamp increments by 1, and comes back to 1 when exceeding to the upper bound, *i.e.*, $2^d - 1$. We reserve 0 to indicate that there is no item in a cell. Therefore, when the window size is fixed, smaller hopping steps require longer hopping timestamps, *i.e.*, larger memory size. Hopping timestamps are stored in cells. If a cell is hashed to by an incoming item, the hopping timestamp stored in the cell will be updated to the real-time hopping timestamp; If a cell has not been accessed/updated for a hopping window, the hopping timestamp stored in the cell is expected to be cleaned, *i.e.*, set to 0 by the cleaning method.

Global Cleaning: The compression in hopping timestamps may mistake an outdated hopping timestamp as if it was accessed within the current hopping window, which incurs *timestamp error*. Global cleaning can eliminate the timestamp error. Specifically, all the cells are traversed and all the outdated hopping timestamps are cleaned instantly every time the hopping window hops a step. This scheme totally eliminates the timestamp error since the hopping timestamp is cleaned once it is outdated. Further, due to the instant cleaning, the range of the hopping timestamp is fully utilized, which leads to the **optimal** fine-grained timescale. Specifically, when the hopping window hops and the hopping timestamp updates, only the hopping timestamps that are the same as the new hopping timestamp are actually outdated and need to be cleaned, because older hopping timestamps have been certainly cleaned before. With global cleaning, all hopping timestamps are valid within the current hopping window. Therefore, if an algorithm uses both hopping timestamp and global cleaning, we consider it as optimal in hopping windows. However, the traversing process in global cleaning needs to visit the entire data structure, resulting in thousands or more memory accesses. Such many memory accesses occurring every time the hopping window hops forward one step are time-consuming and may incur interruptions of the stream processing.

Local Cleaning: Local cleaning approximately cleans outdated information with small error. It does not need the traversing access, and thus is much faster than global cleaning. Every time a cell is accessed/updated, a small number of adjacent cells are traversed for cleaning. The arrival rate of items is fast, indicating that a great many cells will be accessed/updated per second. Therefore, as long as the arrival rate of items is fast enough, outdated information will be totally cleaned in time. The details of local cleaning are illustrated as follows. The cells are divided consecutively into equal-sized groups. The memory size of each group is small enough to be visited within a single memory access. For example, if the bit width of a cell is typically set to 8, we can group 8 cells as a 64-bit integer, which can be visited in one memory access in most modern operating system and hardware device. When an item is inserted into a cell, all the outdated cells in the same group are cleaned simultaneously. Due to the theoretical proof presented in Section IV-A, it is nearly impossible that timestamp error occurs in any cell. The hopping step in local cleaning is inappropriate to set to the same size as that in global cleaning, since we have to leave enough time for the outdated cells to be accessed/updated and cleaned locally. If we want to maintain the same hopping step size as global cleaning, an extra bit should be used.

C. HoppingTimer for Bloom filter

The data structure of Bloom filter can provide membership query in fixed windows. We make an extension to this data structure to support membership query in hopping windows.

Data structure: The data structure extends the m-bit array in the Bloom filter to an array of m cells, associated with k hash functions, $h_1, h_2, ..., h_k$, each of which is used to map incoming items to one of the cells. Each cell stores a d-bit hopping timestamp. The i^{th} cell is denoted by $C[i]$ and the hopping timestamp in the i^{th} cell is denoted by $C[i].Ts$.

Initialization: All cells in the data structure are initialized to 0. The cells are divided into g groups equally and consecutively. With the size of hopping windows W and the hopping step size s specified by the user, the bit width of hopping timestamp d is set to $\log_2(\frac{W}{s}) + 1$, since we have to leave enough time for the outdated cells to be updated and cleaned at the cost of 1 bit. For easier understanding, in the formulas of the rest of this paper, we use d instead of s.

Insertion: To insert an item e_i with arrival time t_i, we first calculate its corresponding hopping timestamp Ts_i by the following formula:

$$Ts = \left\lfloor \frac{t}{\frac{W}{2^{d-1}}} \right\rfloor \bmod (2^d - 1) + 1 \tag{1}$$

Then, for each hash function h_j, we determine its hashed cell $C[L]$, where L is the result of the hash function, $h_j(e_i)$. Then we update $C[L].Ts$ to Ts_i.

Cleaning: We adopt our proposed key technique, local cleaning, to clean the outdated items. We first determine the hashed groups to which the hashed cells belong. Then we locally traverse all cells in the hashed groups and set the outdated ones to 0. To judge whether a cell is outdated, We perform the following two steps. First, for cell $C[L]$, if $C[L].Ts = 0$, then $C[L]$ is not outdated because no item has visited it. Second, if $C[L].Ts \neq 0$, we estimate the timestamp difference, $\delta T(Ts_i, C[L].Ts)$, according to the following formula:

$$\delta T(Ts_1, Ts_2) = (Ts_1 - Ts_2) \bmod (2^d - 1) \tag{2}$$

If $\delta T(Ts_i, C[L].Ts) \geq 2^{d-1}$, we consider that no item has been inserted into the cell for at least 2^{d-1} steps, *i.e.*, a hopping window, then $C[L]$ is considered as outdated, and we set $C[L].Ts$ to 0. Otherwise, $C[L]$ is considered as not outdated and remains unchanged.

Query: To query the membership of item e_i, we first calculate the hopping timestamp of current time, denoted by Ts_N. Then, for each hashed cell $C[L]$, we check whether $C[L].Ts = 0$ and $\delta T(Ts_N, C[L].Ts) \geq 2^{d-1}$. If no hopping timestamp in the hashed cells meets the condition, we answer that item e_i is in the current hopping window. Otherwise, we answer that item e_i is not in the current hopping window.

D. HoppingTimer for BitMap

The data structure of Bitmap can provide cardinality estimation in fixed windows. We make an extension to this data structure to support cardinality estimation in hopping windows.

Data structure: HoppingTimer for Bitmap has a similar data structure to that for Bloom filter. The only difference is that there is only one hash functions, h_1.

Initialization: The initialization is the same as that for Bloom filter.

Insertion: The insertion is similar to that for Bloom filter. The difference is that we only update one hashed cell with one hash function.

Cleaning: The cleaning is similar to that for Bloom filter. The difference is that we only clean one hashed group to which the hashed cell belongs.

Query: The number of no hashed bits u is initialized to 0. To query the cardinality in the hopping window, we first calculate the hopping timestamp of current time, denoted by Ts_N. Then, for each cell $C[L]$, we check whether $C[L].Ts = 0$ and $\delta T(Ts_N, C[L].Ts) \geq 2^{d-1}$. If so, we increment u by 1. After traversing all the cells in the data structure, the estimated cardinaility is calculated as $-mln\frac{u}{m}$, where m is the number of cells in the data structure. Finally, we return the estimated cardinaility as the result.

E. HoppingTimer for Bloom filter+Hopping Counters

The data structure consisting of Bloom filter and a counter can provide cardinality estimation in fixed windows. Each

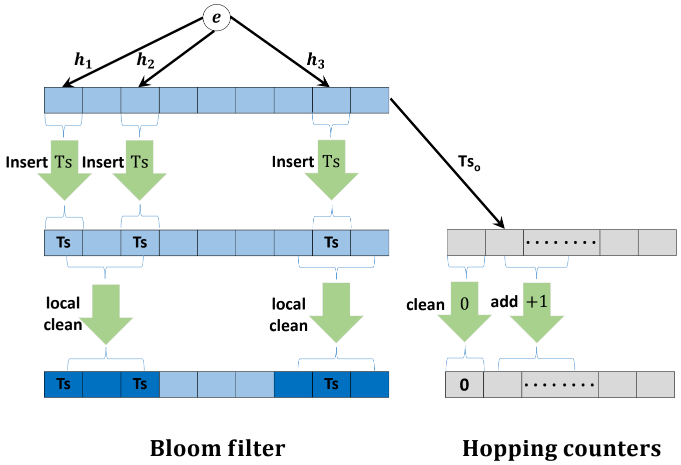


Fig. 2: HoppingTimer for Bloom filter and Hopping Counters. The Bloom filter is divided into 3 groups and each group has 3 cells. To insert item e with hopping timestamp Ts, we calculate the hash functions to get k hashed cells, pick out the oldest hopping timestamp Ts_o from the hashed cells. For the Bloom filter, we update the hopping timestamps in the hashed cells to Ts and locally clean the hashed groups. For the hopping counters, we insert Ts_o to them by incrementing a series of hopping counters by 1 and set the previous hopping counter to 0.

time when the Bloom filter identifies a new inserted item, the counter increments by 1. We make an extension to this data structure to support cardinality estimation in hopping windows.

Data Structure (Fig. 2): Compared to the data structure for Bloom filter, there is an additional circular array consisting of $2^d - 1$ counters. We call counters in the circular array hopping counters. We randomly choose a counter from the array as the first hopping counter. The j^{th} hopping counter, denoted by $HC[j]$, is used to record the cardinality in the hopping window which ends at hopping timestamp j.

Initialization: The initialization is the same as that for Bloom filter, except that all the hopping counters are set to 0.

Insertion: The insertion consists of two parts, the insertion in Bloom filter and the insertion in hopping counters. The first part is for membership query, which is the same as HoppingTimer for Bloom filter, and the second part is for cardinality estimation. Suppose item e_i arrives at t_i, of which the hopping timestamp is Ts_i. Before the hashed cells in the Bloom filter are cleaned, we query e_i in the Bloom filter and temporally maintain the query result, *i.e.*, the hopping timestamp for the last arrival time of e_i, which is denoted by Ts_o. We first judge whether Ts_o is within the current hopping window by formula 2. If Ts_o is out of the current hopping window, meaning that e_i is a new item for the current hopping window and its next $2^{d-1} - 1$ hopping windows, then we increment counter $HC[Ts_i]$ and its next $2^{d-1} - 1$ counters by 1. Otherwise, we consider that e_i is not outdated for current hopping window. Further, the cardinality of the next $2^{d-1} - 1 - \delta T(Ts_i, Ts_o)$ hopping windows has already counted e_i in. Therefore, we only increment counter $HC[Ts_i + 2^{d-1} - \delta T(Ts_i, Ts_o)]$ and its next $\delta T(Ts_i, Ts_o) - 1$ counters by 1.

Cleaning: Except the local cleaning, we set counter $HC[Ts_i - 1]$ to 0 in addition.

Query: The membership query is the same as that in Bloom filter. To estimate the cardinality in the hopping window, we first calculate the hopping timestamp of current time, denoted by Ts_N. Then we return the value of counter $HC[Ts_N]$ as the cardinality.

Extension: The HoppingTimer can also implement the function of estimating the number of items in the hopping windows after slight adjustment. The only difference is the insertion operation. To insert an item e_i arriving at hopping timestamp Ts_i, instead of the former update scheme of hopping counter, we increment $HC[Ts_i]$ and its next $2^{d-1} - 1$ counters by 1 in any case.

F. Discussion

In this section, we discuss two kinds of errors in hopping windows. Further, we propose how to approximate the sliding windows.

Fixed-window Error: Fixed-window error is the error caused by the mechanism of fixed-window algorithms (*e.g.* Bitmap or Bloom filter). Multiple different items mapped into a same cell incurs hash collisions, which lead to a typical fixed-window error. For example, when querying the membership of an item in the Bloom filter, if all its hashed cells have been unfortunately mapped into by other items, the queried item will be considered in the hopping window though it has never appeared. An ideal and optimal hopping-window algorithm may eliminate all the errors caused by the window hopping, but is incapable to correct fixed-window errors.

Timestamp Error: Timestamp error is caused by hopping timestamps. Local cleaning might not clean outdated hopping timestamps completely. For example, several groups may have not been cleaned for a long time because of local cleaning. As a result, all items remain in these group are outdated. Also, items long apart in time may have the same hopping timestamp, which result in that part of the outdated cells will not be cleaned when these groups are finally locally cleaned. In Section IV, we theoretically prove that the upper bound of the probability that timestamp error occurs is $O(e^{-\alpha f})$, where f is the number of cells in each group and α is an positive parameter independent of f. Therefore, there is almost only fixed-window error when applied to hopping-window model if f is reasonably set.

HoppingTimer for Sliding Windows: When the hopping step is set to 1 and the hopping timestamp is 64-bit width, the hopping-window model is equivalent to the sliding-window model. Therefore, we can use hopping windows to approximate sliding windows by gradually increasing the length of hopping timestamps and reducing the size of the hopping steps. Further, to avoid the potential false negative error, we increment the threshold of timestamp difference, which is used to determine whether outdated, by 1.

IV. Mathematical Analysis

In this section, we first provide a theoretical analysis for the probability of timestamp error caused by local cleaning in Section IV-A. Then, we provide a theoretical analysis for the additional false positive rate caused by the timestamp error in Section IV-B. We prove that the timestamp error and the additional false positive rate for membership query are both limited by the bound that is inverse of an exponential function of the number of cells in each group.

Since in time-based hopping window model, the distribution of the arrival time of the items in the data stream is completely random, we analyze the data stream in the count-based window model. Let W be the size of hopping windows, d be the length of hopping timestamp, m be the number of cells in the data structure, k be the number of hash functions and f be the number of cells in each group.

A. Probability of Timestamp Error

In this section, we provide a theoretical analysis for the probability of timestamp error caused by local cleaning.

Theorem 1: The probability of timestamp error is $O\left(e^{-\alpha f}\right)$, where $\alpha = \frac{(2^{d-1}-1)k}{2^{d-1}(2^d-1)m}W$, and satisfies that

$$Pr\{error\} \leq \frac{m}{e^{\frac{(2^{d-1}-1)k(f+1)}{2^{d-1}(2^d-1)m}W} - 1} \tag{3}$$

Proof: The timestamp error will occur if and only if there is outdated hopping timestamp remaining in the data structure. We split the time into pieces to analyze timestamp error. Let t_c be the current time, $t_c = \frac{xW}{2^{d-1}}$, where $x \geq 2^{d-1}$ and x is an integer. Then we try to calculate the probability that there is an outdated hopping timestamp in a fixed cell and its arrival time is in $(t_c - \frac{(i+1)W}{2^{d-1}}, t_c - \frac{iW}{2^{d-1}}]$, where $i \geq 2^{d-1}$ and i is an integer.

We use A_i to denote the event that an outdated hopping timestamp with arrival time in $(t_c - \frac{(i+1)W}{2^{d-1}}, t_c - \frac{iW}{2^{d-1}}]$ remains in the fixed cell, X_i to denote the event that at least one item is inserted to the fixed cell in $(t_c - \frac{(i+1)W}{2^{d-1}}, t_c - \frac{iW}{2^{d-1}}]$, Y_i to denote the event that no local cleaning in its group considers the cell as outdated and cleans it and no item is inserted to the cell after $t_c - \frac{iW}{2^{d-1}}$. As discussed above, A_i happens if and only if both X_i and Y_i happen. Because X_i and Y_i are obviously independent, it follows directly that

$$Pr\{A_i\} = Pr\{X_i\}Pr\{Y_i\} \tag{4}$$

First, we calculate the probability that an hopping timestamp is inserted to the fixed cell in $(t_c - \frac{(i+1)W}{2^{d-1}}, t_c - \frac{iW}{2^{d-1}}]$. For each incoming item, the probability that it is not hashed to the fixed cell is

$$Pr\{not\ hit_1\} = (1 - \frac{1}{m})^k \tag{5}$$

As the number of items in the time interval is $\frac{W}{2^{d-1}}$, we have

$$\begin{aligned} Pr\{X_i\} &= 1 - Pr\{not\ hit_1\}^{\frac{W}{2^{d-1}}} \\ &= 1 - (1 - \frac{1}{m})^{\frac{kW}{2^{d-1}}} \end{aligned} \tag{6}$$

Second, we calculate the probability that no local cleaning cleans it and no item is inserted into this cell after $t_c - \frac{iW}{2^{d-1}}$. We use Ts_a to denote the hopping timestamp of the arrival time of an incoming item and Ts_c to denote the hopping timestamp of the item in the cell. According to the mechanism of our local cleaning, We divided the incoming items after $t_c - \frac{iW}{2^{d-1}}$ to two kinds. The first kind of items satisfy $\delta T(Ts_a, Ts_c) \geq W$, while the second kind of items satisfy $\delta T(Ts_a, Ts_c) < W$. For the first kind of items, the fixed cell will not be cleaned or updated if and only if the incoming item is inserted into other groups. We have

$$Pr\{not\ clean\ or\ hit\} = (1 - \frac{f}{m})^k \tag{7}$$

For the second kind of items, the fixed cell will not be cleaned or updated if and only if the incoming item is not inserted into itself. We have

$$Pr\{not\ hit_2\} = (1 - \frac{1}{m})^k \tag{8}$$

Then we need to count the number of the first kind of items and the number of the second kind of items to calculate the probability of Y_i. We denote them by $N(K_1)$ and $N(K_2)$, and denote the total number of items after $t_c - \frac{iW}{2^{d-1}}$ by $N(V)$, which is $\frac{iW}{2^{d-1}}$. For every hopping timestamp, in each whole window of size $\frac{2^d-1}{2^{d-1}}W$, there are $\frac{2^{d-1}-1}{2^{d-1}}W$ items of the first kind and W items of the second kind. As the items just after $t_c - \frac{iW}{2^{d-1}}$ are of the second kind, we can conclude that

$$N(K_1) \leq N(K_2) \leq \frac{2^{d-1}}{2^{d-1}-1}N(K_1) + \frac{2^{d-1}-1}{2^{d-1}}W \tag{9}$$

It is obvious that

$$N(K_1) + N(K_2) = N(V) = \frac{iW}{2^{d-1}} \tag{10}$$

From (9) and (10), We get that

$$N(K_1) \geq \frac{(2^{d-1}-1)(i-2^{d-1}+1)}{2^{d-1}(2^d-1)}W \tag{11}$$

Then, we can calculate the upper bound of the probability that Y_i happens. We have

$$\begin{aligned} Pr\{Y_i\} &= Pr\{not\ clean\ or\ hit\}^{N(K_1)}Pr\{not\ hit_2\}^{N(K_2)} \\ &= (1 - \frac{f}{m})^{kN(K_1)}(1 - \frac{1}{m})^{kN(K_2)} \\ &\leq (1 - \frac{f}{m})^{kN(K_1)}(1 - \frac{1}{m})^{kN(K_1)} \\ &\leq e^{-\frac{k(f+1)N(K_1)}{m}} \\ &\leq e^{-\frac{(2^{d-1}-1)(i-2^{d-1}+1)k(f+1)}{2^{d-1}(2^d-1)m}W} \end{aligned} \tag{12}$$

Then we calculate the probability that an outdated hopping timestamp with arrival time in $(t_c - \frac{(i+1)W}{2^{d-1}}, t_c - \frac{iW}{2^{d-1}}]$ remains in the fixed cell. We have

$$\begin{aligned} Pr\{A_i\} &= Pr\{X_i\}Pr\{Y_i\} \\ &\le (1-(1-\frac{1}{m})^{\frac{kW}{2^{d-1}}})e^{-\frac{(2^{d-1}-1)(i-2^{d-1}+1)k(f+1)}{2^{d-1}(2^d-1)m}W} \\ &\le e^{-\frac{(2^{d-1}-1)(i-2^{d-1}+1)k(f+1)}{2^{d-1}(2^d-1)m}W} \end{aligned} \tag{13}$$

Using the union bound for i, we conclude that the upper bound of the probability that an outdated hopping timestamp occurs in a fixed cell. We have

$$\begin{aligned} Pr\{error\ in\ a\ cell\} &\le Pr\{\exists i \ge 2^{d-1}, A_i\} \\ &\le \sum_{i=2^{d-1}}^{+\infty} Pr\{A_i\} \\ &\le \sum_{i=2^{d-1}}^{+\infty} e^{-\frac{(2^{d-1}-1)(i-2^{d-1}+1)k(f+1)}{2^{d-1}(2^d-1)m}W} \\ &= \frac{1}{e^{\frac{(2^{d-1}-1)k(f+1)}{2^{d-1}(2^d-1)m}W}-1} \end{aligned} \tag{14}$$

Again, using the union bound for all cells, we can get the upper bound of the probability that an outdated hopping timestamp occurs in the array. We have

$$\begin{aligned} Pr\{error\} &\le mPr\{error\ in\ a\ cell\} \\ &= \frac{m}{e^{\frac{(2^{d-1}-1)k(f+1)}{2^{d-1}(2^d-1)m}W}-1} \end{aligned} \tag{15}$$

Obviously the probability that timestamp error occurs, *i.e.*, $Pr\{error\}$ is $O\left(e^{-\alpha f}\right)$, where $\alpha = \frac{(2^{d-1}-1)k}{2^{d-1}(2^d-1)m}W$.

B. Additional False Positive Rate Caused by Timestamp Error

In this section, we provide a theoretical analysis for the additional false positive rate caused by the timestamp error.

Theorem 2: The additional FPR caused by the timestamp error is $O\left(e^{-\alpha f}\right)$, where $\alpha = \frac{(2^{d-1}-1)k}{2^{d-1}(2^d-1)m}W$.

Proof: The false positive error happens if an item that never appears in the window is considered as appeared. To calculate the additional false positive rate (short for FPR) caused by the timestamp error, we calculate the FPR of the HoppingTimer in fixed windows and in hopping windows, and then get the difference.

We first calculate the FPR of the HoppingTimer without timestamp error, *i.e.*, the FPR in fixed windows. An item will be considered as appearing in the window if and only if each of its hashed cells records a hopping timestamp. For an arbitrary cell, we can calculate the probability that the cell records a hopping timestamp. We have

$$Pr\{hit_1\} = 1-(1-\frac{1}{m})^{kW} \tag{16}$$

Then, for an item that never appears in the window, we can calculate the probability that all its hashed cells record timestamps, *i.e.*, the FPR. We have

$$\begin{aligned} FPR\{fixed\} &= Pr\{hit_1\}^k \\ &= (1-(1-\frac{1}{m})^{kW})^k \end{aligned} \tag{17}$$

Second, we calculate the FPR of the HoppingTimer in hopping windows. If a hopping timestamp is recorded in a cell, either an item is hashed to the cell in the latest window, or a timestamp error occurs in the cell. Using the union bound, we can conclude the probability that an arbitrary cell records a timestamp is less than or equal to the sum of the probability that timestamp error occurs and the probability that the cell is hashed to by an item in the window. We have

$$Pr\{hit_2\} \le Pr\{hit_1\} + Pr\{error\ in\ a\ cell\} \tag{18}$$

Then, we can get the FPR with timestamp error. We have

$$\begin{aligned} FPR\{hopping\} &= Pr\{hit_2\}^k \\ &\le (Pr\{hit_1\} + Pr\{error\ in\ a\ cell\})^k \end{aligned} \tag{19}$$

Finally, from (17) and (19), we can calculate the additional FPR. Let $Pr\{hit_1\}$ be a, $Pr\{error\ in\ a\ cell\}$ be b. We have

$$\begin{aligned} FPR\{add\} &= FPR\{hopping\} - FPR\{fixed\} \\ &\le (a+b)^k - a^k \\ &= \sum_{i=1}^{k} \binom{k}{i} a^{k-i} b^i \\ &= \sum_{i=1}^{k} \binom{k}{i} Pr\{hit_1\}^{k-i} Pr\{error\ in\ a\ cell\}^i \end{aligned} \tag{20}$$

Since $Pr\{hit_1\} < 1$ and $Pr\{error\ in\ a\ cell\}$ is $O\left(e^{-\alpha f}\right)$, we can conclude that the additional false positive rate is also $O\left(e^{-\alpha f}\right)$, where $\alpha = \frac{(2^{d-1}-1)k}{2^{d-1}(2^d-1)m}W$.

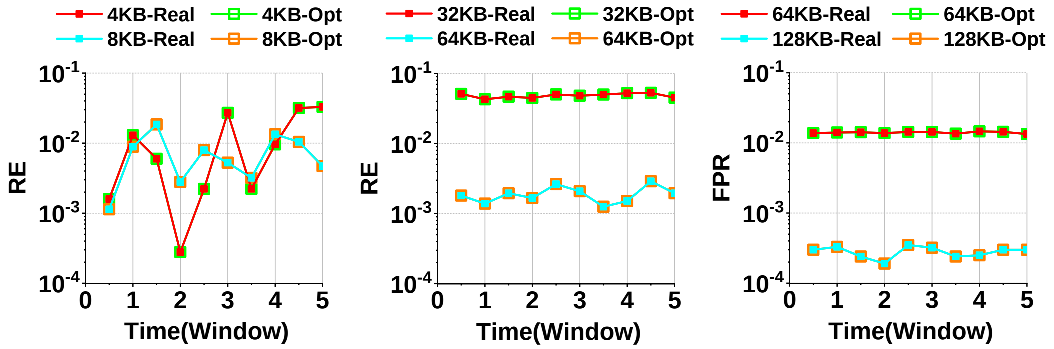
V. EXPERIMENTAL RESULTS

In this section, we call the HoppingTimer **HT** for short. We first show the experimental setup. Then we show the performance of algorithms of HT compared to the optimal conditions in hopping windows when parameters vary, where optimal conditions mean that no outdated hopping timestamp is recorded by HT. Finally, we compare algorithms of HT to the prior work, using metrics of sliding-window model.

A. Experimental Setup

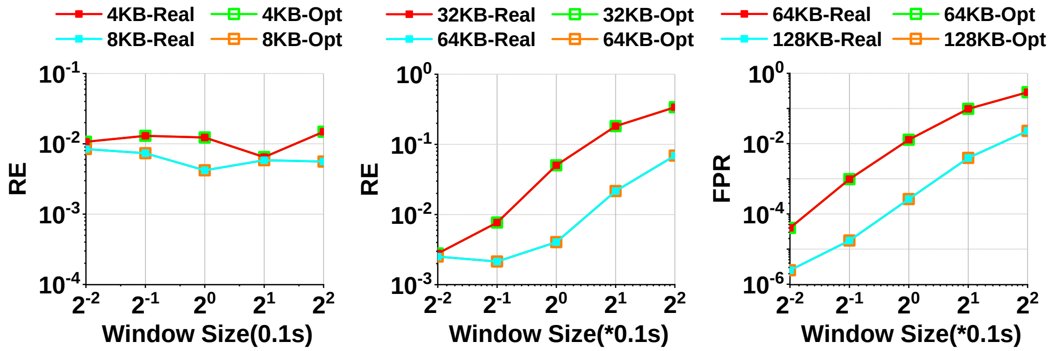
1) Datasets:

- **CAIDA:** As many other papers do, we use the public traffic dataset which is released by CAIDA [32] to test all algorithms in HT and other algorithms.
- **Webpage:** Webpage [33] is collected from websites by crawling a number of webpages.
- **Campus:** Campus consists of IP traces which are collected from the main gateway in our campus.



(a) HT-BM (b) HT-HC (c) HT-BF

Fig. 3: Stability of HT as the time goes by.



(a) HT-BM (b) HT-HC (c) HT-BF

Fig. 4: Adaptability to different window sizes.

2) Evaluation Metrics :

- **FPR (False Positive Rate):** $\delta = \frac{n}{m}$, where m denotes the number of queried items that do not appear in the latest hopping window or sliding window, n denotes the number of items that are mistaken for the items in the latest window. Our algorithm does not have false-negative, so we only use FPR to evaluate the accuracy of the membership query in HT.
- **RE (Relative Error of Cardinality):** $\frac{|D-\hat{D}|}{D}$, where D denotes the number of distinct items in the latest window and $\hat{D}$ denotes the estimated value of D. We use RE to evaluate the accuracy of our algorithm in cardinality estimation.
- **Throughput:** We use MIPS (million insertions per second) to evaluate the throughput of insertion for each algorithm.

3) Default Settings :

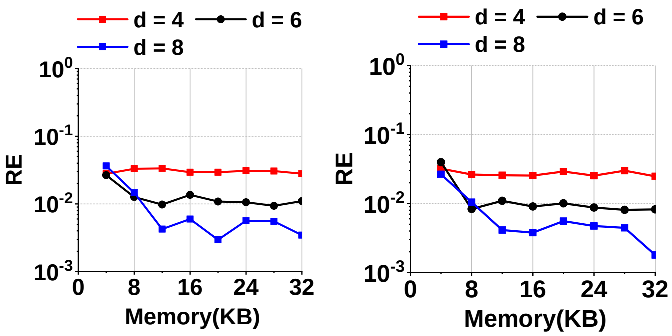
We test our algorithms on CPU platform and implement them in C++. The default parameters are set as follows. The number of cells in each group, denoted by f, is 8. The length of hopping timestamp, denoted by d, is 8-bit. The size of count-based window is 65536 and the size of time-based window is 0.1 second. The number of hash functions, denoted by k, is 8. In all experiments, We use BOB Hash [34] to calculate 32-bit hash values of items. More detailed settings for other algorithms are listed below.

- **HT-BM and HT-HC:** For cardinality estimation in sliding windows, we compare **HT-BM** (short for HoppingTimer for Bitmap) and **HT-HC** (short for HoppingTimer for Bloom filter+Hopping counters) to SWAMP [21], CVS [19] and TSV [31]. For SWAMP, we set the length of its fingerprint to 16-bit. For CVS, we set the size of its counter to 4-bit. For TSV, we set the length of its timestamp to 64-bit.
- **HT-BF:** For membership query in sliding windows, we compare our **HT-BF** (short for HoppingTimer for Bloom filter) to SWAMP and TBF [18]. For SWAMP, we also set the length of its fingerprint to 16-bit. For TBF, we set the length of its timestamp to 18-bit and number of hash functions to 8. Though HT-HC can also provide membership query, we don't compare it with other algorithms because it is strictly inferior than HT-BF in membership query.

B. Impact of Parameters in Hopping Windows

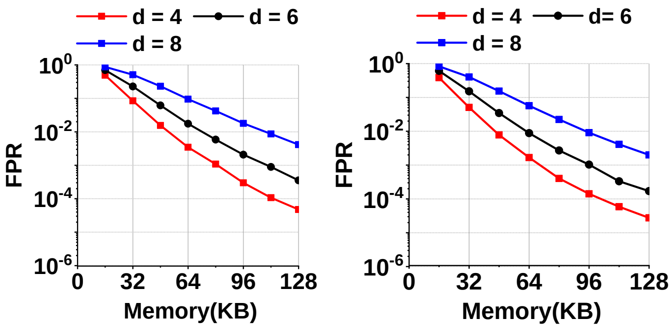
Performance vs. time (Fig. 3): *The experimental results show that HT is near-optimal and stable as the time goes by.* We test algorithms of HT every half window with two different sizes of memory on CAIDA, using time-based window model, and compare them to the optimal conditions. The performance of HT-HC and HT-BF is stable and quite close to the optimal conditions when given enough memory. HT-BM is not so stable, but it is near-optimal and keeps low RE.

Performance vs. window size (Fig. 4): *The experimental results show that HT is near-optimal and adaptable as the window size varies.* We test our algorithms with two different sizes of memory on CAIDA, using time-based window model, and compare them to the optimal conditions. The performance of all algorithms of HT is adaptable and quite close to the optimal conditions when the window size varies.



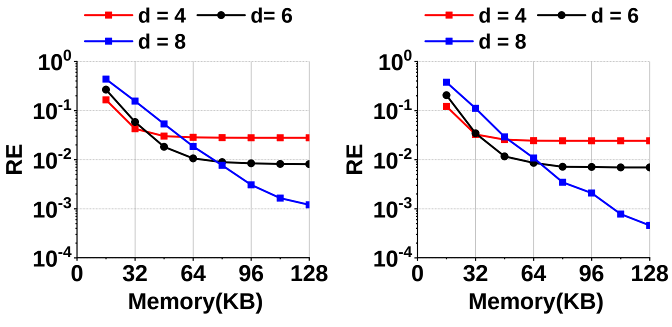
(a) CAIDA (b) Webpage

Fig. 5: Performance vs. d on HT-BM.



(a) CAIDA (b) Webpage

Fig. 6: Performance vs. d on HT-BF.

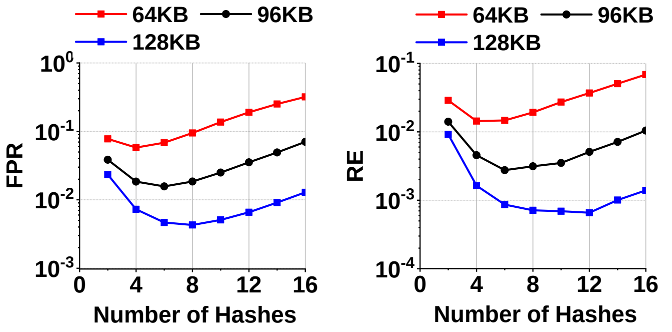


(a) CAIDA (b) Webpage

Fig. 7: Performance vs. d on HT-HC.

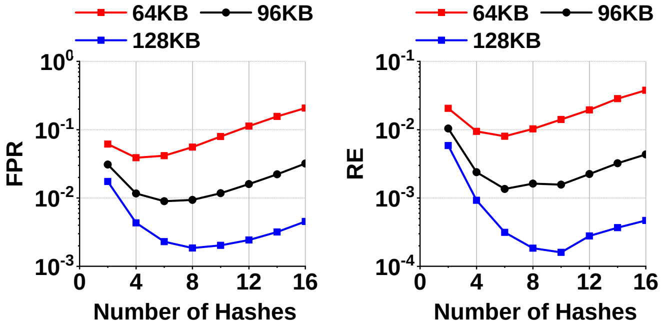
Performance vs. d on HT-BM (Fig. 5): *The experimental results show that within a certain range of memory, longer hopping timestamp can achieve lower RE.* We test the RE of HT-BM with three different lengths of hopping timestamp on CAIDA and Webpage rspectively, varying the memory size from 4KB to 32KB. The performance of 8-bit hopping timestamp is always better than that of 4-bit and 6-bit.

Performance vs. d on HT-BF (Fig. 6): *The experimental results show that within a certain range of memory, shorter hopping timestamp can achieve lower FPR.* We test the RE of HT-BM with three different lengths of hopping timestamp on CAIDA and Webpage rspectively, varying the memory size from 16KB to 128KB. The performance of 4-bit hopping timestamp is always better than that of 6-bit and 8-bit.



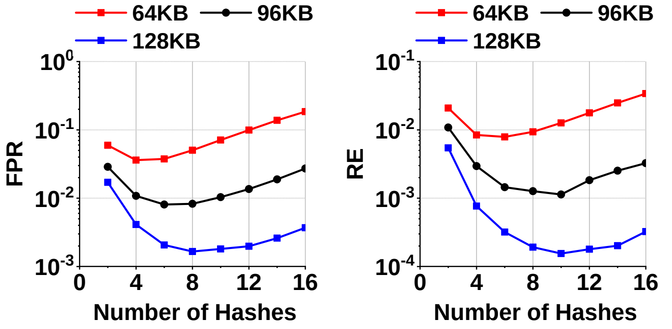
(a) HT-BF (b) HT-HC

Fig. 8: Performance vs. k on CAIDA.



(a) HT-BF (b) HT-HC

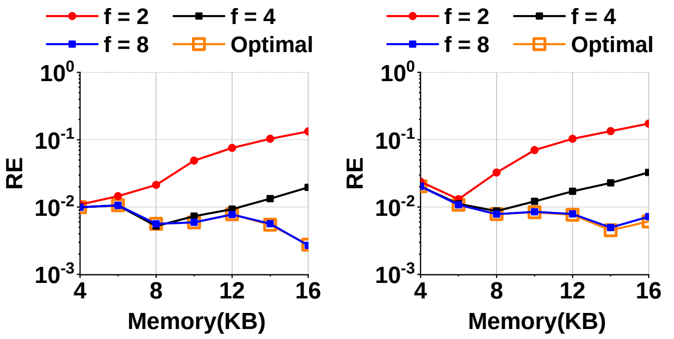
Fig. 9: Performance vs. k on Webpage.



(a) HT-BF (b) HT-HC

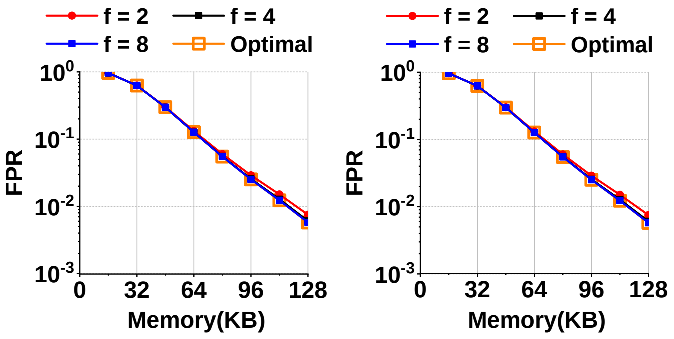
Fig. 10: Performance vs. k on Campus.

Performance vs. d on HT-HC (Fig. 7): *The experimental results show that when the memory size is small, shorter hopping timestamp can achieve lower RE; when the the memory* *size is large, longer hopping timestamp can achieve lower RE.* We test the RE of HT-HC with three different lengths of hopping timestamp on CAIDA and Webpage respectively, varying the memory size from 16KB to 128KB. When the memory is less than 32KB, 4-bit hopping timestamp achieves the lowest RE. When the memory is larger than 32KB and less than 64KB, 6-bit hopping timestamp achieves the lowest RE. When the memory is larger than 64KB, 8-bit hopping timestamp achieves the lowest RE.



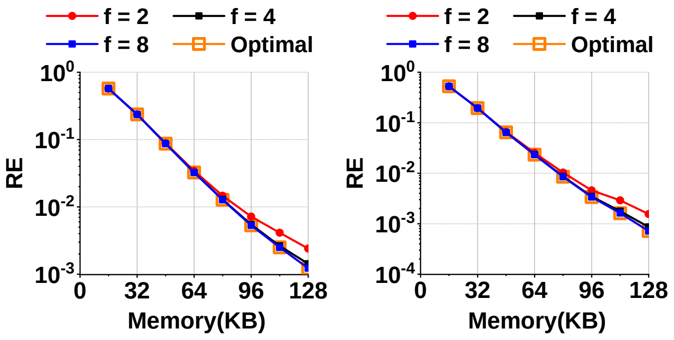
(a) CAIDA (b) Webpage

Fig. 11: Performance vs. f on HT-BM.



(a) CAIDA (b) Webpage

Fig. 12: Performance vs. f on HT-BF.



(a) CAIDA (b) Webpage

Fig. 13: Performance vs. f on HT-HC.

Summary and Analysis of Parameter d: On HT-BM, larger d performs better. That is because within the memory range that we set, the error caused by hash collision is less than the error caused by coarse-grained hopping timestamp. Therefore the more precise timestamp can perform better. On HT-BF, smaller d performs better. That is because within the memory range that we set, the error caused by hash collision is more than the error caused by coarse-grained hopping timestamp. Therefore the shorter timestamp can perform better. On HT-HC, when the memory size is small, shorter hopping timestamp can achieve lower RE; when the the memory size is large, longer hopping timestamp can achieve lower RE. That is because when the memory is large enough, the error caused by hash collision is negligible while the error caused by coarse-grained hopping timestamp can be persistently improved. Therefore the longer timestamp can perform better with large memory.

Performance vs. k (Fig. 8, Fig. 9, Fig. 10): *The experimental results show that there is always an optimal number of hash functions when the other parameters are fixed.* We test our algorithms HT-BF and HT-HC with three different sizes of memory on CAIDA, Webpage and Campus respectively. As k increases, the performance of all algorithms with all kinds of memory sizes gets better and then gets worse. We find that under the experimental conditions, $k = 8$ performs nearly the best. Therefore we choose $k = 8$ as our default setting.

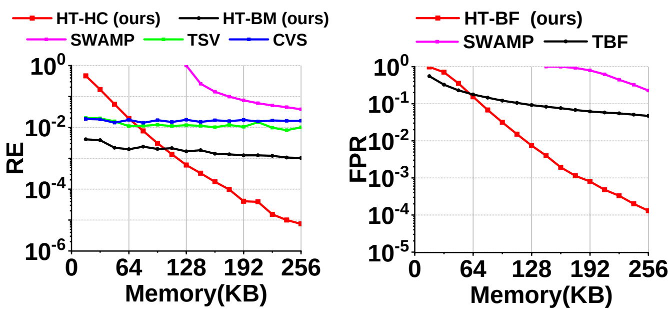
Summary and Analysis of Parameter k: As k increases, the performance of all algorithms with all kinds of memory sizes gets better and then gets worse. That is because when k is small, a large amount of memory is not utilized; when k is large, hash collisions seriously hurt the performance.

Performance vs. f on HT-BM (Fig. 11): *The experimental results show that the performance of HT-BM becomes better when f becomes larger.* We test HT-BM with three different sizes of cell group and compare it to the optimal condition by varying the memory size from 4KB to 16KB on CAIDA and Webpage respectively. We find when each group contains 8 cells, the performance is almost optimal when the memory is less than 16KB.

Performance vs. f on HT-BF (Fig. 12): *The experimental results show that the performance of HT-BF becomes better when f becomes larger.* We test HT-BF with three different sizes of cell group and compare it to the optimal condition by varying the memory size from 16KB to 128KB on CAIDA and Webpage respectively. We find when each group contains 8 cells, the performance is almost optimal when the memory is less than 128KB.

Performance vs. f on HT-HC (Fig. 13): *The experimental results show that the performance of HT-HC becomes better when f becomes larger.* We test HT-HC with three different sizes of cell group and compare it to the optimal condition by varying the memory size from 16KB to 128KB on CAIDA and Webpage respectively. We find when each group contains 8 cells, the performance is almost optimal when the memory is less than 128KB.

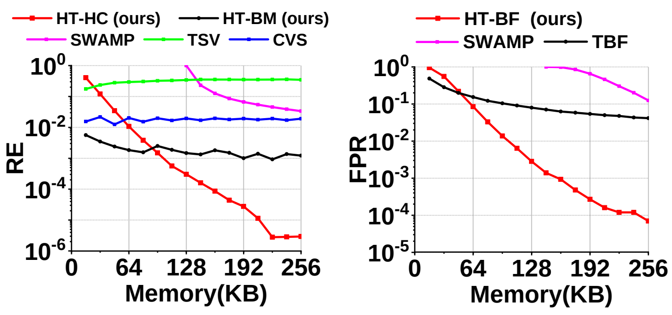
Summary and Analysis of Parameter f: As f increases, the performance of all algorithms with all kinds of memory sizes gets better. That is because when f increases, more outdated hopping timestamps are cleaned.



(a) Cardinality

(b) Membership

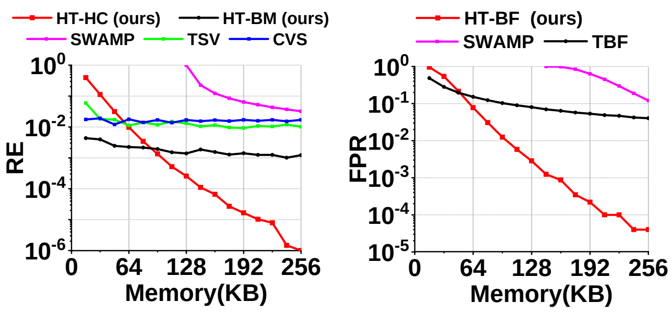
Fig. 14: Accuracy comparison on CAIDA.



(a) Cardinality

(b) Membership

Fig. 15: Accuracy comparison on Webpage.



(a) Cardinality

(b) Membership

Fig. 16: Accuracy comparison on Campus.

C. Accuracy Performance Comparison in Sliding Windows

We compare the accuracy of HT to the state-of-the-art algorithms. As not all algorithms can support time-based sliding window, we test them based on count-based windows.

HT-BM and HT-HC vs. Others (Fig. 14a, Fig. 15a, Fig. 16a): *The experimental results show when estimating the cardinality, our HT-BM and HT-HC work better than other algorithms. We can choose an appropriate algorithm from either of the two depending on the amount of memory available.* We test HT-BM and HT-HC and compare them to other algorithms by varying the memory size from 16KB to 256KB on CAIDA, Webpage and Campus respectively. We find when the memory is limited, HT-BM can achieve the lowest RE among all algorithms. When the memory is sufficient, HT-HC can further reduce the RE.

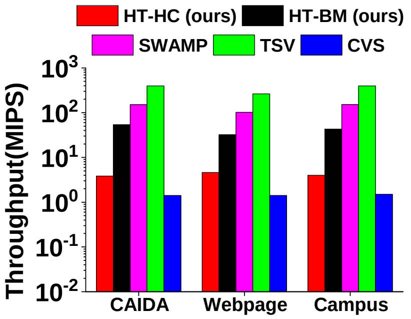


Fig. 17: Throughput Comparison of HT-BM and HT-HC.

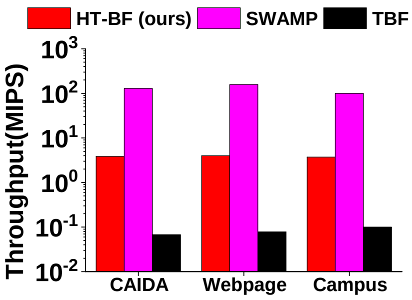


Fig. 18: Throughput Comparison of HT-BF.

HT-BF vs. Others (Fig. 14b, Fig. 15b, Fig. 16b): *The experimental results show when querying the membership, our HT-BF achieves better performance than other algorithms.* We test HT-BF and compare it to other algoritgms by varying the memory size from 16KB to 256KB on CAIDA, Webpage and Campus respectively. We find when the memory is 256KB, HT-BF achieves about 1000 times lower FPR than TBF and SWAMP.

D. Throughput Comparison

In this section, we compare the throughput of HT to the state-of-the-art algorithms.

HT-BM and HT-HC vs. Others (Fig. 17): *The experimental results show that the throughput of HT-BM and HT-HC is slower than TSV and SWAMP, and faster than CVS* We test HT-BM and HT-HC and compare them to other algorithms on CAIDA, Webpage and Campus respectively. Though our algorithms are slower than part of other algorithms, our accuracy performance is always better than them.

HT-BF vs. Others (Fig. 18): *The experimental results show that the throughput of HT-BF is slower than SWAMP, and faster than TBF.* We test HT-BF and compare it to other algorithms on CAIDA, Webpage and Campus respectively. Though our algorithm is slower than part of other algorithms, our accuracy performance is always better than them.

VI. Conclusion

In this paper, we propose the HoppingTimer for basic estimation of high-speed data streams in hopping windows. It is a near-optimal and generic framework which can adapt three fixed-window algorithms to hopping windows for basic estimation tasks. With key techniques namely hopping timestamp and local cleaning, HoppingTimer is applicable for both time-based hopping windows and count-based hopping windows. We theoretically analyze the probability that timestamp error occurs, finding that almost no outdated hopping timestamps will remain if the number of cells in each group is set reasonably. HoppingTimer can also approximate sliding windows by increasing the length of hopping timestamps. Experimental results show that, HoppingTimer is near-optimal in hopping windows, and achieves a false positive rate about 1000 times lower than the state-of-the-art.

ACKNOWLEDGEMENT

We would like to thank the anonymous reviewers for their valuable suggestions that helped improve the paper. This work is supported by National Key R&D Program of China (No. 2022YFB2901504), and National Natural Science Foundation of China (NSFC) (No. U20A20179, 62372009).

REFERENCES

[1] 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 Workshop on Intelligence and Security Informatics*, pages 96–111. Springer, 2012.

[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 *International Conference on Information Security*, pages 415–426. Springer, 2006.

[3] Andrew R Curtis, Jeffrey C Mogul, Jean Tourrilhes, Praveen Yalagandula, Puneet Sharma, and Sujata Banerjee. Devoflow: Scaling flow management for high-performance networks. In *Proceedings of the ACM SIGCOMM 2011 conference*, pages 254–265, 2011.

[4] Abdul Kabbani, Mohammad Alizadeh, Masato Yasuda, Rong Pan, and Balaji Prabhakar. Af-qcn: Approximate fairness with quantized congestion notification for multi-tenanted data centers. In *2010 18th ieee symposium on high performance interconnects*, pages 58–65. IEEE, 2010.

[5] Bryan Ball, Mark Flood, Hosagrahar Visvesvaraya Jagadish, Joe Langsam, Louiqa Raschid, and Peratham Wiriyathammabhum. A flexible and extensible contract aggregation framework (caf) for financial data stream analytics. In *Proceedings of the International Workshop on Data Science for Macro-Modeling*, pages 1–6, 2014.

[6] Lajos Gergely Gyurkó, Terry Lyons, Mark Kontkowski, and Jonathan Field. Extracting information from the signature of a financial data stream. *arXiv preprint arXiv:1307.7244*, 2013.

[7] Aiyou Chen, Li Erran Li, and Jin Cao. Tracking cardinality distributions in network traffic. In *IEEE INFOCOM 2009*, pages 819–827. IEEE, 2009.

[8] Chen Qian, Hoilun Ngan, Yunhao Liu, and Lionel M Ni. Cardinality estimation for large-scale rfid systems. *IEEE transactions on parallel and distributed systems*, 22(9):1441–1454, 2011.

[9] Philippe Flajolet and G Nigel Martin. Probabilistic counting algorithms for data base applications. *Journal of computer and system sciences*, 31(2):182–209, 1985.

[10] Cristian Estan, George Varghese, and Mike Fisk. Bitmap algorithms for counting active flows on high speed links. In *Proceedings of the 3rd ACM SIGCOMM conference on Internet measurement*, pages 153–166. ACM, 2003.

[11] Fang Hao, Murali Kodialam, TV Lakshman, and Haoyu Song. Fast multiset membership testing using combinatorial bloom filters. In *IEEE INFOCOM 2009*, pages 513–521. IEEE, 2009.

[12] Yu Hua, Bin Xiao, Bharadwaj Veeravalli, and Dan Feng. Locality-sensitive bloom filter for approximate membership query. *IEEE Transactions on Computers*, 61(6):817–830, 2011.

[13] Gil Einziger, Roy Friedman, and Ben Manes. Tinylfu: A highly efficient cache admission policy. *ACM Transactions on Storage (ToS)*, 13(4):1–31, 2017.

[14] Haibo Chen, Heng Zhang, Mingkai Dong, Zhaoguo Wang, Yubin Xia, Haibing Guan, and Binyu Zang. Efficient and available in-memory kv-store with hybrid erasure coding and replication. *ACM Transactions on Storage (TOS)*, 13(3):1–30, 2017.

[15] Jelena Mirkovic and Peter Reiher. A taxonomy of ddos attack and ddos defense mechanisms. *ACM SIGCOMM Computer Communication Review*, 34(2):39–53, 2004.

[16] Sugam Agarwal, Murali Kodialam, and TV Lakshman. Traffic engineering in software defined networks. In *2013 Proceedings IEEE INFOCOM*, pages 2211–2219. IEEE, 2013.

[17] Jin Li, David Maier, Kristin Tufte, Vassilis Papadimos, and Peter A Tucker. No pane, no gain: efficient evaluation of sliding-window aggregates over data streams. *Acm Sigmod Record*, 34(1):39–44, 2005.

[18] Linfeng Zhang and Yong Guan. Detecting click fraud in pay-per-click streams of online advertising networks. In *2008 The 28th International Conference on Distributed Computing Systems*, pages 77–84. IEEE, 2008.

[19] Jingsong Shan, Jianxin Luo, Guiqiang Ni, Zhaofeng Wu, and Weiwei Duan. Cvs: fast cardinality estimation for large-scale data streams over sliding windows. *Neurocomputing*, 194:107–116, 2016.

[20] Yousra Chabchoub and Georges Hébrail. Sliding hyperloglog: Estimating cardinality in a data stream over a sliding window. In *2010 IEEE International Conference on Data Mining Workshops*, pages 1297–1303. IEEE, 2010.

[21] Eran Assaf, Ran Ben Basat, Gil Einziger, and Roy Friedman. Pay for a sliding bloom filter and get counting, distinct elements, and entropy for free. In *IEEE INFOCOM 2018-IEEE Conference on Computer Communications*, pages 2204–2212. IEEE, 2018.

[22] Laura Feinstein, Dan Schnackenberg, Ravindra Balupari, and Darrell Kindred. Statistical approaches to ddos attack detection and response. In *Proceedings DARPA information survivability conference and exposition*, volume 1, pages 303–314. IEEE, 2003.

[23] George Nychis, Vyas Sekar, David G Andersen, Hyong Kim, and Hui Zhang. An empirical evaluation of entropy-based traffic anomaly detection. In *Proceedings of the 8th ACM SIGCOMM conference on Internet measurement*, pages 151–156. ACM, 2008.

[24] Arno Wagner and Bernhard Plattner. Entropy based worm and anomaly detection in fast ip networks. In *14th IEEE International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprise (WETICE'05)*, pages 172–177. IEEE, 2005.

[25] Kyu-Young Whang, Brad T Vander-Zanden, and Howard M Taylor. A linear-time probabilistic counting algorithm for database applications. *ACM Transactions on Database Systems (TODS)*, 15(2):208–229, 1990.

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

[27] Haipeng Dai, Yuankun Zhong, Alex X Liu, Wei Wang, and Meng Li. Noisy bloom filters for multi-set membership testing. In *ACM SIGMETRICS Performance Evaluation Review*, volume 44, pages 139–151. ACM, 2016.

[28] MyungKeun Yoon, Tao Li, Shigang Chen, and J-K Peir. Fit a spread estimator in small memory. In *IEEE INFOCOM 2009*, pages 504–512. IEEE, 2009.

[29] Philippe Flajolet, Éric Fusy, Olivier Gandouet, and Frédéric Meunier. Hyperloglog: the analysis of a near-optimal cardinality estimation algorithm. In *Discrete Mathematics and Theoretical Computer Science*, pages 137–156. Discrete Mathematics and Theoretical Computer Science, 2007.

[30] Sasu Tarkoma, Christian Esteve Rothenberg, and Eemil Lagerspetz. Theory and practice of bloom filters for distributed systems. *IEEE Communications Surveys & Tutorials*, 14(1):131–155, 2011.

[31] Hyang-Ah Kim and David R O'Hallaron. Counting network flows in real time. In *GLOBECOM'03. IEEE Global Telecommunications Conference (IEEE Cat. No. 03CH37489)*, volume 7, pages 3888–3893. IEEE, 2003.

[32] The caida anonymized internet traces, 2020. http://www.caida.org/data/overview/.

[33] Frequent itemset mining dataset repository. http://fimi.ua.ac.be/data, 2004.

[34] Bob jenkins' hash function web page, paper published in dr dobb's journal, 2008. http://burtleburtle.net/bob/hash/doobs.html.