Shen H, Zhang Y. Improved approximate detection of duplicates for data streams over sliding windows. JOURNAL OF COMPUTER SCIENCE AND TECHNOLOGY 23(6): 973–987 Nov. 2008

Improved Approximate Detection of Duplicates for Data Streams Over Sliding Windows

Hong Shen[1,2] (沈 鸿) and Yu Zhang[1] (张 育)

[1] *Department of Computer Science and Technology, University of Science and Technology of China, Hefei 230027, China*

[2] *School of Computer Science, University of Adelaide, SA 5005, Australia*

E-mail: hongshen@ustc.edu.cn; zhangyu110@mail.ustc.edu.cn

Received January 14, 2008; revised July 22, 2008.

Abstract Detecting duplicates in data streams is an important problem that has a wide range of applications. In general, precisely detecting duplicates in an unbounded data stream is not feasible in most streaming scenarios, and, on the other hand, the elements in data streams are always time sensitive. These make it particular significant approximately detecting duplicates among newly arrived elements of a data stream within a fixed time frame. In this paper, we present a novel data structure, Decaying Bloom Filter (DBF), as an extension of the Counting Bloom Filter, that effectively removes stale elements as new elements continuously arrive over sliding windows. On the DBF basis we present an efficient algorithm to approximately detect duplicates over sliding windows. Our algorithm may produce false positive errors, but not false negative errors as in many previous results. We analyze the time complexity and detection accuracy, and give a tight upper bound of false positive rate. For a given space G bits and sliding window size W, our algorithm has an amortized time complexity of $O(\sqrt{G/W})$. Both analytical and experimental results on synthetic data demonstrate that our algorithm is superior in both execution time and detection accuracy to the previous results.

Keywords data stream, duplicate detection, bloom filter, approximate query, sliding window

1 Introduction

Recently, online monitoring of data streams has emerged as an important problem in Data Stream Management Systems (DSMS)[1,2]. One interesting problem in data monitoring is to find duplicate elements, which was studied by several researchers[3,4]. Duplicate detection over data streams has applications in a wide range of areas. For example, recently, Metally *et al.*[3] proposed duplicate detection in click streams. In a web advertising scenario, advertisers pay web site publishers for clicks on their advertisements. However, a phenomenon called click inflation[5] shows that there is an incentive for publishers to falsely increase the number of clicks generated from their sites. Hence a third party, the advertising commissioner, has to detect those false clicks by monitoring duplicate click IDs. Each click ID is represented by a pair of a customer ID and an advertisement ID. When calculating the publishers' commissions, traditionally the advertising commissioner runs queries to capture duplicate clicks within a short period of time, a day for example. Another network monitoring application is URL Crawling[6,7], where search engines regularly crawl the web to enlarge their collections of web pages. Given the URL of a web page, which is often extracted from the content of a crawled page, a search engine must probe its archive to find out if the URL is in the collection and if fetching the URL from a remote site can be avoided. In addition, duplicate detection can also be used to query distinct IP addresses. In network monitoring and accounting, it is often important to understand the traffic and to identify the customers on the network[4,8]. For example, these queries may be interesting to network monitors: who are the customers on the network within the past 12 hours? Where they go? The results could be helpful in analyzing customer profiles, interests and network traffic.

According to the way the stream is handled, there are two basic variants of duplicate detection over data streams[3]: *Duplicate Detection over a Landmark Window* asks for duplicate elements that have occurred since the occurrence of a specific landmark which can

Regular Paper

This work is supported by the "Hundred Talents Program" of CAS and the National Natural Science Foundation of China under Grant No. 60772034.

be defined in terms of either the number of elements or time units; *Duplicate Detection over a Sliding Window* asks for duplicate elements that have occurred in the last W elements.

Since the elements in a data stream are always time sensitive and the available memory space to buffer elements is limited, detecting duplicates over a sliding window covering the latest W elements of the stream makes more sense than over a landmark window. Certainly, the window size can be expressed either in terms of the number of elements or time units. To find duplicates over sliding windows in a data stream, a straightforward way is to store all elements over a sliding window in a buffer, and when observing a new element, we just compare it to all the elements in the buffer. The new element is identified as distinct if it is not found, and duplicate otherwise. When the buffer is full, a newly arrived element may evict another element out of the buffer before it is stored. The main disadvantage of this solution to processing a window of size W is that it requires $O(W)$ comparisons to tell whether a newly observed element is a duplicate, or not.

Although building an index on the elements can reduce the search cost to $O(\log W)$ per element, it would, on the other hand, increase the element insertion cost to $O(\log W)$ instead of just appending an element in constant time. In addition, take click streams for an example, each Click ID is represented by 64 characters. Thus, keeping all IDs in a sliding window of size W entails keeping $64 \times W$ bytes, which could be infeasible for a large window.

The solution described above solves the problem exactly, but suffers from slow processing of the data stream. On the other hand, exact duplicate detection requires to store the entire current window, so if the size of sliding windows is too large, it may be infeasible to store the entire window with the limited main memory. To cope with high stream rates, providing fast answers in an on-line fashion in a reduced storage space and an acceptable error rate is clearly more desirable than a precise one that is slow. Therefore, we only consider the problem of approximate detection of duplicates over sliding windows.

The contributions of this paper are:

1) an efficient solution to detecting duplicate element over sliding windows without false-negative error;
2) analysis of time requirements of our proposed solution as well as the false positive rate;
3) a set of experiments to evaluate the performance of our proposed solution as well as its comparison with other methods.

The rest of this paper is organized as follows. Section 2 reviews previous work on duplicate detection over data streams. In Section 3, we give the problem statement and some background knowledge on Bloom Filters. In Section 4, we introduce a novel data structure that is the foundation for solving our problem. In Section 5, we present our duplicate detection algorithm. We report the results of our experiments in Section 6, followed by the conclusions in Section 7.

2 Related Work

A straightforward exact solution to detecting duplicates in a streaming environment is the buffering or caching method, which has been studied in many fields such as database systems, operating systems and more recently URL caching in Web crawling[7]. The problem of exact duplicate elimination is well studied, and many efficient algorithms have been developed (e.g., see [9] for details). Another branch of duplicate detection techniques focuses on fuzzy duplicates detection[10−13], which targets at identifying multiple representations of real-world objects stored in a data source, and is a critical task in data cleaning, data mining, and data integration.

For the problem of approximately detecting duplicates in a streaming environment, Metwally *et al.*[3] considered different window models and respectively proposed different solutions based on Bloom Filters (BF)[14]: landmark windows, jumping windows and sliding windows.

For the landmark window model, they applied the original Bloom filters to duplicates detection, and thus did not consider the situation that the BF becomes "full" (which will be discussed in the next section). Recently, Deng and Rafiei[4] proposed Stable Bloom Filter, a data structure that continuously evicts elements in the landmark window with a probability that is related to their arriving order to make room for those newly arrived elements.

For the jumping window model, [3] split a large jumping window into multiple sub-windows, and presented both the jumping window and the sub-windows with Bloom Filters of the same size. Thus, a jumping window can "jump" forward by updating (adding new and removing old) sub-window BFs.

For the sliding window model, which is the scenario we consider, [3] used Counting Bloom filters (CBF)[15] to allow removing stale (old) elements out of the filter. However, this can be done only when the element to be removed is known, which is not possible in many streaming cases. For example, if the oldest element

needs to be removed, one has to know which counters are touched by the oldest element, but this information cannot be found in CBF. Moreover, the CBF has two other major drawbacks in detecting duplicates over sliding windows when the input stream is skewed as will be discussed in the next section.

In addition, for the problem of approximate membership query in a non-streaming environment, several improvements[15,16] have been proposed over the original Bloom Filter. Cohen *et al.*[17] extended the Bloom Filter to answering multiplicity queries. In [18] an extension of Bloom Filter was used to count distinct elements. Cheng *et al.*[19] studied the problem of time-decaying counters and proposed the time-decaying Bloom Filter as an extension of CBF.

In this paper, we propose a more efficient solution to approximately detecting duplicates over sliding windows. On the CBF basis, we introduce first a data structure called Decaying Bloom Filter (DBF), and then an efficient algorithm for approximately detecting duplicates and dynamically updating DBF to represent the newly arrived elements in the current window. Unlike Metwally's solution[3], which uses CBF to remember multiplicities of distinct elements in sliding windows and extra space to keep the arrival information of these elements in order to remove them from CBF when they slide out of sliding windows, our data structure, DBF, can dynamically represent this information without needing extra space. Then, we give the Decaying property of DBF. This property shows that it is unnecessary to reflect the multiplicities of elements which are required by CBF in solving our problem.

Although our data structure uses a similar name to that in [19], there are several essential differences between them: the counters in the Time-Decaying Bloom Filter (TDBF)[19] are used for maintaining the time-decaying counters over a landmark window. When a new element arrives, TDBF adds its mapped counters by 1, and then decreases the values of all counters by a user-specified decaying function every fixed time unit. In contrast, the counters in our DBF are used for maintaining the arrival information of elements in a sliding window. When a new element arrives, we set its mapped counters to W (the size of sliding window), and decrease all positive-value counters in the DBF by 1. Clearly, the value of an element's mapped counters in DBF gives an estimate on the element's expiration time (expiry, i.e., remaining life time) in the current window, whereas that in TDBF stands for an estimate on the multiplicity (time-decaying counter) of this element. Because of this essential difference, TDBF does not support dynamically maintaining membership of elements over sliding windows as DBF does. Therefore, they are oriented forwards different data stream applications. While DBF is concerned with the problem of online analysis over data stream — detection of duplicates — which has a strict limitation on the amortized time complexity for processing each element and memory space, TDBF targets at the problem of maintaining time-decaying counters in data streams. Furthermore, since in DBF blocking and delaying techniques are applied in the update process, DBF requires much less time for the update than TDBF.

Our algorithm operates on the DBF at the block level by dividing a data stream into non-overlapping blocks and processing a block instead of a single data element. It also incorporates a delaying technique to postpone the update to DBF as needed. Thus, it updates a smaller number of DBF counters than that in Metwally's result to CBF[3] when the current window slides, so runs more efficiently.

On the DBF basis, for an approximate membership query the false-positive rate of the answer from our solution can be controlled at a lower level than that in Metwally's result[3] based on CBF.

3 Problem Statement and Background to Bloom Filters

3.1 Problem Statement

There are two kinds of sliding windows: time-sliding window and element-based sliding window. We only discuss the latter and the situation of time-sliding windows can be easily deduced from that.

We define a data stream as a sequence of numbers: $S_N = e_1, \ldots, e_i, \ldots, e_N$, where N is the size of the stream. Without loss of generality, the value of N can be infinite. Generally, a stream can be a sequence of records, and it is not hard to transform each record to a number (e.g., through hashing or fingerprinting).

We formulate the problem as follows: given a data stream S_N, memory space of G bits and sliding window size of W bits, we want to find out whether each current element e_i in S_N has duplicates in $e_{\max(i-W,1)}, \ldots, e_{i-1}$ or not. For S_N, there are two constraints: answers must be provided in an on-line fashion and G is not large enough to store all distinct elements in $e_{\max(i-W,1)}, \ldots, e_{i-1}$. Therefore, it is difficult to solve the problem precisely. Our goal is to develop a fast approximate solution that has a small error rate.

To address this problem, we first review the techniques of Bloom Filter and its variants that have been previously used in different situations for detecting

duplicates over data streams. We discuss their disadvantages for approximately detecting duplicates over sliding windows.

3.2 Bloom Filter and Its Variants

Bloom Filter (BF) was proposed by Burton Bloom in 1970[14] and used for detecting approximate membership of elements. A BF uses k hash functions $h_1, h_2, \ldots, h_k$ to hash each element of set S into a bit array of size m, where S comes from a universe U. All bits in BF are initially set to 0. For each element $e \in S$, the bits at positions $h_1(e), h_2(e), \ldots, h_k(e)$ in the filter are set to 1. BF only allows for membership queries. For example, given a newly arrived element $u \in U$, whether u is a duplicate in S can be determined by the bits at positions $h_1(u), h_2(u), \ldots, h_k(u)$. If any of these bits is zero, we know u is a distinct element. Otherwise, it is regarded as a duplicate with a certain probability of error. This method has a small probability of producing a false-positive error, i.e., a distinct element is wrongly reported as duplicate.

If the hash functions are perfectly random, the probability of a false positive (false positive rate) $\Pr(FP) = (1-P_0)^k = (1-(1-1/m)^{kn})^k \approx (1-e^{-kn/m})^k$, where P_0 is the probability that a specific bit is still 0 after inserting n distinct elements. The expression on the right is minimized when $k = m \ln 2/n$, in which case the error rate is $\Pr(FP)_{\min} = (1-1/2)^k = (0.6185)^{m/n}$, where m is the number of bits in BF.

Although BF is simple and space efficient, it does not allow deletions. Deleting elements from a BF cannot be done simply by changing them back to zeros, as a single bit may correspond to multiple elements. Therefore, in the data stream environment, if we apply BF in detecting duplicates, when more and more new elements arrive, the fraction of zeros in BF will decrease continuously, and the false positive rate will increase accordingly, and finally reach the limit one. At this time every distinct element will be reported as duplicate, indicating that BF fails completely. We call such a state of BF "full".

For the purpose allowing deletion of stale elements, Counting Bloom Filter (CBF) was proposed[15]. A CBF uses an array of m integer counters $(C_1, C_2, \ldots, C_m)$ instead of bits; the counters track the number of elements currently hashed to that location. If a CBF represents a total of M data elements, including repeated values, the sum of the values of all counters equals $\sum_{i=1}^{m} C_i = k \times M$. Deletion can now be safely done by decreasing the values of relevant counters. A BF can be derived from a CBF by setting all non-zero counters to one. Since the usual data structure representing the CBF consists of a static data set where counters have a fixed size over time, counters must be chosen large enough to avoid overflow. It was shown that four bits sufficed for most non-streaming applications[15].

Though CBF was used for detecting duplicates for data streams over sliding windows in [3], it has three major drawbacks: 1) false negative errors — when the input stream is highly skewed, an insertion of a new element may result in a counter overflow, and then deletion operations to elements that are hashed to these corresponding counters can no longer be reflected; 2) memory waste when the input stream is skewed — in CBF all counters have the same number of bits deciding the maximum number of elements hashed to them; 3) extra space required to maintain the oldest element which is needed for the deletion (counter) operation.

Recently Stable Bloom Filter (SBF)[4] was proposed based on CBF[15] for approximately detecting duplicates over landmark windows with limited space. Since recent data are more important, SBF continuously evicts elements in the current window with a probability that is related to this importance so that SBF has room for those newly arrived elements. Because this importance decays over time, the probability of deletion of a new element is smaller than that of an old element. This is useful in many streaming scenarios over landmark windows. However, SBF is not feasible to be applied over sliding windows because of its high probability of false negative errors.

4 Decaying Bloom Filter Architecture

In this section, we first introduce a data structure, Decaying Bloom Filter (DBF), which is an extension of Counting Bloom Filter (CBF), and a series of operations over it. We then show the decaying property of DBF and point out that DBF can represent elements over sliding windows with no additional space needed to keep the arrival information of elements and with no risk of counter overflow. Finally, we analyze the performances of DBF when it is used for detecting duplicates over sliding windows.

4.1 Decaying Bloom Filter Design

Definition 1 (Decaying Bloom Filter (DBF)). *Given memory space G bits, a DBF consists of an array of counters $DBF[1]$, ..., $DBF[m]$, each being allocated to d $(= \lceil \log(W+1) \rceil)$[①] bits with a minimum value*

[①]Note that throughout this paper we assume log to be $\log_2$.

zero and maximum value W, where W is the size of the sliding window and $G = m \times d$.

When using a DBF to detect duplicates over sliding windows, the initial values of all counters are set to zero. For each newly arrived element in the stream, we execute the following three operations in order:

1) as in Bloom Filter, the element is mapped to k counters by some uniform and independent hash functions. Whether the element has duplicates can be determined by probing whether all the k counters it is hashed to are non-zero;

2) for all counters with values greater than zero, decrease their values by one;

3) set (the values of) the same k counters as in the first operation to W.

Each execution of these three operations is called an iteration which contains the entire process for a newly arrived element. We call the first operation the duplicate detection process, and the next two operations the update process.

Concretely speaking, we use the values of counters in DBF to approximately represent the expiry of elements which are hashed to the counters. For every element that is inserted into a DBF, we set its mapped counters to W, and then we decrease all positive-value counters by one. A counter is decremented to 0 only if all the elements that are hashed to this counter slide out of the current window, which is equivalent to deleting all those obsolete elements from the DBF. Thus, it is possible to update the DBF as new elements are added to the sliding window, and as aged elements are deleted.

4.2 Decaying Property

A sliding window can be described by a decay function in many stream scenarios[20,21]. Given a window size W, the function is defined as $f(e) = 1$ if e is in the latest W elements and $f(e) = 0$ otherwise. On the DBF basis, we can find an important property which meets this sliding window decay function. We call it the decaying property.

Theorem 1 (Decaying Property). *For a continuous and unbounded data stream, if in each iteration the counters that new elements are hashed to are set to W, the DBF represents only the latest W elements at the present time.*

Proof. When the number of elements in the stream seen so far is smaller than W, because all counters mapped by the elements are impossible to decrease to 0 during l iterations ($l < W$) , no elements are removed from the DBF. It is trivial that DBF represents the latest W elements.

Next, we consider the situation that the number of elements in the stream seen so far is greater than or equal to W. Let us first consider the situation that two data sets are represented by two identical Bloom Filters. If the two sets are the same, then values of the corresponding bits in these Bloom Filters are the same. However, the converse may not be true. Suppose a data stream $S_{i+W-1} = \{e_1, e_2, \ldots, e_{i+W-1}\}$ ($i \geqslant 1$) and the set of the latest W elements over the data stream at the present time is S ($= \{e_i, e_{i+1}, \ldots, e_{i+W-1}\}$). Let us insert S_{i+W-1} into a DBF, and S into a Bloom Filter in which the size of array and the set of hash functions are the same to that in the DBF. Since all non-zero counters in the DBF were set at least once during iterations i and $i + W - 1$, the elements before e_i has no impact on the DBF at the present time. By taking all non-zero counters as one-value bits and all zero counters as zero-value bits, we can easily see that the DBF representing the set is equivalent to the original Bloom Filter. Therefore, the DBF only represents the latest W elements at the present time. □

Fig.1 shows an example of how a DBF is used for keeping a sliding window of size W with $k = 4$. DBF_{i+W-1} denotes the current state of the DBF after the latest element e_{i+W-1} is inserted, and its corresponding Bloom Filter represents the set $\{e_i, e_{i+1}, \ldots, e_{i+W-1}\}$ from data stream. When a new element e_{i+W} arrives, through updating DBF_{i+W-1} to DBF_{i+W}, we have another corresponding Bloom Filter which represents the set $\{e_{i+1}, e_{i+2}, \ldots, e_{i+W}\}$. In Fig.1, e_i has not been removed from DBF, because none of the k counters mapped by e_i is reduced to zero. In practice, e_i is removed from DBF since all of its mapped counters are set by other elements after e_i.

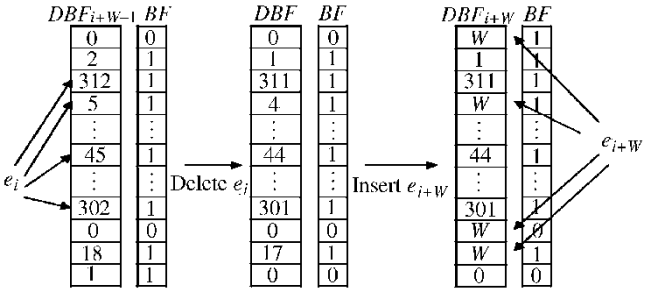


Fig.1. Insert and delete elements in a DBF.

The decaying property guarantees that the current window represented by DBF slides as new elements arrive. Thus, the DBF can keep a fixed-size sliding windows (we will introduce variable-size sliding windows in Section 5). Therefore, unlike CBF, we do not need additional space to keep the arrival information of elements

in the current window, and thus consequently save a lot of memory space. In addition, the number of bits assigned to each counter is only related to W and independent of the input stream. Therefore, in comparison with CBF, it is unnecessary to worry about counter overflow when a new element is inserted, and memory waste if we assign too many bits to each counter.

4.3 Performance Analysis

4.3.1 False Positive Rate

In the DBF, false positive is the only type of errors. The probability of false positive (FP) can be easily computed,

Corollary 1. *There could be only false positives in a DBF, and the probability of false positive (false positive rate (FPR)) is*

$$\Pr(FP) = (1-(1-1/m)^{kw})^k \approx (1-e^{-kw/m})^k \leqslant (1-e^{-kW/m})^k,$$

where k is the number of hash functions, m is the number of counters in the DBF, and w is the number of distinct elements in sliding widows.

Proof. According to Theorem 1 and because $w \leqslant W$, FPR of DBF can be easily derived from the deduction of FPR of the corresponding Bloom Filter. □

The upper bound of FPR can be reached when the stream elements in the current window are distinct to each other. In order to minimize the upper bound, we set $k = m \times \ln 2/W = G \times \ln 2/(W \times \log W)$. In this case the error rate is $\Pr(FP) \leqslant (1/2)^k = (1/2)^{G\times \ln 2/(W\times \log W)} = (0.6185)^{G/(W\times \log W)}$.

4.3.2 Time Complexity

Since our goal is to minimize the FPR given a fixed amount of space, we do not consider space complexity, but just focus on time complexity. There are three operations within each iteration in DBF for processing a newly arrived element: probing k cells to detect duplicates, decreasing all m counters by 1, setting the same k cells as probed in the first step to W. Suppose all these operations are time-equal, thus, the amortized time complexity of each iteration is $O(m+2k)$. Since k is negligible compared to m ($m \gg k$), the amortized time complexity of each iteration is $O(G/\log W)$.

DBF has good performances in saving space and avoiding counter overflow compared with CBF. However, it is still infeasible to directly use DBF to solve our problem. From the analysis of time complexity, the update process of a DBF is too slow and intolerable for real-time applications. In addition, the number of bits allocated to each counter is too large for many practical applications. For example, Given the fixed amount of memory space $G = 20 \times 2^{20}$ bits, an application which asks for the duplicate elements that occurred in the last 2^{20} elements needs 21 bits for every counter in a DBF and decreases about 2^{20} counters within each iteration.

5 Duplicate Detection Based on DBF

In this section, we present an efficient detection algorithm based on DBF mentioned above. The basic idea is to update DBF at the block level.

We divide the stream into non-overlapping blocks of the same size. In order to fix the size of each block, we introduce a threshold T, where T is an integer and $W \gg T$. As depicted in Fig.2, blocks are numbered by 0, 1, 2, ... and smaller number blocks contain older elements. We assume that both W and T are powers of 2 to simplify expression, so there are at least W/T blocks and at most $W/T+1$ blocks covered by a window when the number of elements seen in a data stream exceeds W. Similar to [22], at any given point of time, we assign each block to one of the following five states, depending on T, the number of elements seen so far and the current window size.

Expired. if all its elements are older than the last W elements.

Under Destruction (*UD*): if some of its elements are older than the last W elements, and all the remaining elements belong to the current window.

Active: if all its elements belong to the current window.

Under Construction (*UC*): if some of its elements belong to the current window, and all the remaining ones are yet to arrive.

Inactive: if none of its elements has arrived.

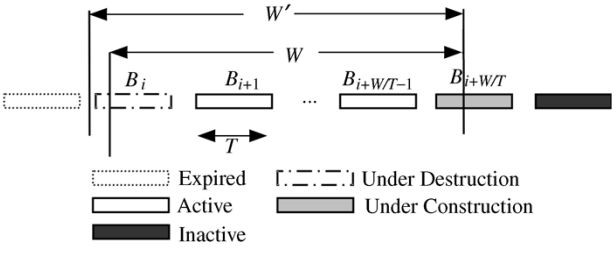


Fig.2. Blocks used in our algorithm.

Each block goes through the sequence of states *Inactive*, *UC*, *Active*, *UD*, and *Expired*. The process is as follows: for each newly arrived element, if there is no *UC* block, we create a *UC* block, and then we insert

the new element into the *UC* block; when the number of the elements inserted to the *UC* block reaches T, we change the state from *UC* to *Active*; and if there is a *UD* block, change the state from *UD* to *Expired*, i.e., remove it from the current window. At the same time, when an element belonging to an *Active* block slides out of the current window, we mark this block as a *UD* block. We call the time interval for a *UC* block from its set up changing into an *Active* block a *UA* interval. Note that each *UA* interval covers T elements.

Although the process of dealing with all blocks is complex, practically, we only need to deal with the blocks in the current window because the elements in *Expired* blocks and *Inactive* blocks have no impact on our query result. Therefore, we formalize the process as follows.

We give these elements belonging to the same block in the current window the same expiry between 1 and $W/T+1$ in order to group them. The expiry assigned to elements belonging to the latest block is $W/T+1$, to elements belonging to the next latest block is W/T, and so on. Since new blocks continuously arrive and stale blocks are removed at the same time, we need to assign a new expiry to elements belonging to existing blocks by decreasing the previously assigned expiry by one. If the expiry of an element is reduced to 0, the element is removed from the current window. This process is very similar to the update process of DBF if we take a block as an element.

We modify the update process of DBF to simulate the above process by decreasing all counter values after a new block arrives (i.e., a *UA* interval) instead of a new element. We can learn the arrival of a new block by T and the current data size of the stream. As shown in Algorithm 1, we keep a variable *Iteration* to denote the current data size of the stream. If *Iteration* mod $T=0$, we are sure that a *UC* block becomes an *Active* block, i.e., a new block arrives. When the current window slides to remove an *Expired* block, we only decrease all counters by 1 according to Theorem 1.

In order to distinguish the data structure used in our algorithm from DBF, we call the data structure block Decaying Bloom Filter (b_DBF). Since there are at most $W/T+1$ blocks in b_DBF covered by the sliding window instead of W elements in DBF, the maximum value of a counter is set to $W/T+1$ instead of W and the number of bits allocated to each counter is $\lceil \log(W/T+2) \rceil$ instead of $\lceil \log(W+1) \rceil$. The detail of our algorithm is given in Algorithm 1.

Algorithm 1. Approximately Detect Duplicates over Sliding Windows Using b_DBF

Input: the threshold T and a sequence of elements $S = e_1, e_2, \ldots, e_N, \ldots$

Output: a sequence of "Yes/No" corresponding to each input element.

```
Initialize DBF[1]...DBF[m] = 0 and Iteration = 1
for each e_i ∈ S do {
  Probe k counters DBF[h_1(e_i)], ..., DBF[h_k(e_i)]
  if none of the above k counters is 0 then
    IsDuplicate = "Yes"
  else
    IsDuplicate = "No"
  end if
  if (Iteration mod T) = 0 then
    for each counter DBF[j] ∈ {DBF[1], ..., DBF[m]}
    do
      if DBF[j] > 0 then
        DBF[j] = DBF[j] − 1
      end if
  end if
      for each counter ∈ {DBF[h_1(e_i)], ..., DBF[h_k(e_i)]}
      do
        DBF[h(e_i)] = W/T + 1
      Output IsDuplicate
      Iteration = Iteration + 1
}
```

5.1 Analysis

We now study the properties of b_DBF.

Let W' denote the number of elements represented by a b_DBF. It is not hard to find that if the number of elements in a stream seen so far is smaller than W, then $W' < W$; otherwise, $W \leqslant W' \leqslant W+T$, where W is the size of sliding windows. Since the stream has an unbounded length, we only discuss the situation that the number of elements in the stream seen so far is greater than W.

Theorem 2. *When the number of elements in a stream seen so far is greater than or equal to W in a b_DBF, if there is a UC block which contains N_{UC} elements in the current window, $W' = W+N_{UC} \leqslant W+T$, where $T \geqslant N_{UC} \geqslant 1$; otherwise, $W' = W$. These W' elements represented by the b_DBF are always the latest.*

Proof. When the number of elements in the stream seen so far is greater than W in a b_DBF, if a b_DBF has a *UC* block, there are $W/T-1$ *Active* blocks, a *UD* block and a *UC* block in the b_DBF, and consequently $W' = (W/T-1+1)T + N_{UC} = W + N_{UC} \leqslant W+T$. Otherwise, there are only W/T *Active* blocks in the b_DBF, and consequently $W' = (W/T) \times T = W$. In addition, because all blocks in the sliding window are always the latest, we can easily deduce that these W' elements represented by the b_DBF are always the latest. □

Corollary 2. *When the number of elements in the stream seen so far is greater than or equal to W (the size of sliding window), there are at least W/T blocks and at most $W/T+1$ blocks in a b_DBF, and the b_DBF keeps a variable-size sliding window where the window size is a variable.*

Proof. According to Theorem 2, a b_DBF has $W/T+1$ blocks if it contains a *UC* block and W/T blocks otherwise. In addition, the number of elements represented by the b_DBF is W', and the value of W' is variable and dependent on the number of elements in a *UC block*. From Theorem 2, $W \leqslant W' \leqslant W+T$. Therefore, the b_DBF keeps a variable-size sliding window. □

Corollary 3. *Given a b_DBF for detecting duplicates over W-width sliding windows, possible errors that may occur are only false positives.*

Proof. According to Theorem 2 and Corollary 2, a b_DBF keeps a variable-size sliding window to represent the newly arrived W' elements, $W' \geqslant W$. Therefore, for a duplicate element, i.e., when the number of elements between it and its nearest predecessor is smaller than or equal to W, it must be reported as a duplicate. However, for a distinct element e_i, i.e., when it has no predecessor or the number of elements between it and its nearest predecessor is greater than W, it is regarded as a duplicate with a certain probability of error. Therefore, there may be only false positives but no false negatives in b_DBF. □

In practice, by inheriting the decaying property of DBF, the b_DBF uses a variable-size sliding window to simulate a fixed-size sliding window. On one hand, in order to avoid false negatives, the b_DBF always represents the newly arrived W' ($\geqslant W$) elements. On the other hand, in order to limit the false-positive rate, we require that W' does not exceed $W+T$.

5.2 False Positive Rate

A false positive (FP) occurs in a b_DBF when a distinct element is wrongly reported as duplicate over the current window. We call the probability false positive rate.

Theorem 3. *Given a b_DBF with m counters, assume every element is inserted into the b_DBF by k hash functions. The false positive rate (FPR) for detecting duplicates over a W-width sliding window when the block size $2 \leqslant T < W$ is not greater than $T/(2\times W)+(1-e^{-k(W+T)/m})^k$.*

Proof. If an element e_i has a predecessor, we denote G_{e_i} as the number of elements between e_i and its nearest predecessor $e_{i-G_{e_i}}$ ($=e_i$). Let $A_{e_i,l}$ denote the event that $G_{e_i}=l$ and let $\bar{A}_{e_i}$ denote the event that e_i has no predecessor or $G_{e_i}>W+T$. The probability of the event that a false positive occurs, denoted by $\Pr(FP)$, is

$$\Pr(FP)=\sum_{l=1}^{\infty}[\Pr(FP|A_{e_i,l})\Pr(A_{e_i,l})] \tag{1}$$

$$=\sum_{l=1}^{W+T}[\Pr(FP|A_{e_i,l})\Pr(A_{e_i,l})]+\Pr(FP|\bar{A}_{e_i})\Pr(\bar{A}_{e_i}) \tag{2}$$

$$=\sum_{l=1}^{W}[\Pr(FP|A_{e_i,l})\Pr(A_{e_i,l})]+\sum_{l=W+1}^{W+T}[\Pr(FP|A_{e_i,l})\Pr(A_{e_i,l})]+\Pr(FP|\bar{A}_{e_i})\Pr(\bar{A}_{e_i}). \tag{3}$$

It is easy to see that

$$\sum_{l=1}^{W}[\Pr(FP|A_{e_i,l})\Pr(A_{e_i,l})]=0. \tag{4}$$

We have (4) because when $W \geqslant G_{e_i} \geqslant 1$, $e_{i-G_{e_i}}$ is in the current window, and hence, according to Corollary 3, we can easily get $\Pr(FP|A_{e_i,l})=0$ ($G_{e_i}=l$). When $W+T \geqslant G_{e_i} > W$, $e_{i-G_{e_i}}$ must be in a *UD* block or an *Expired* block depending on W, T and the position of e_i in the stream. If $G_{e_i} > W+(i \bmod T)$, then $e_{i-G_{e_i}}$ is in an *Expired* block; otherwise, $e_{i-G_{e_i}}$ is in a *UD* block. Let $A_{e_i,U}$ denote the event that $e_{i-G_{e_i}}$ is in a *UD* block and $A_{e_i,E}$ the event that $e_{i-G_{e_i}}$ is in an *Expired* block. Therefore, when $W+T \geqslant l=G_{e_i}>W$, we get

$$A_{e_i,l}=A_{e_i,U}+A_{e_i,E}$$
$$\Pr(A_{e_i,U}\cap A_{e_i,E})=0,$$

and

$$\Pr(FP|A_{e_i,U})=1. \tag{5}$$

So we have

$$\begin{aligned}&\sum_{l=W+1}^{W+T}[\Pr(FP|A_{e_i,l})\Pr(A_{e_i,l})]\\ =&\sum_{l=W+1}^{W+T}[\Pr(FP|A_{e_i,U})\Pr(A_{e_i,U})+\Pr(FP|A_{e_i,E})\Pr(A_{e_i,E})]\\ =&\sum_{l=W+1}^{W+T}\Pr(A_{e_i,U})+\sum_{l=W+1}^{W+T}[\Pr(FP|A_{e_i,E})\Pr(A_{e_i,E})].\end{aligned} \tag{6}$$

We call

$$\Pr(FP_{\text{Active}}) = \Pr(FP|\bar{A}_{e_i})\Pr(\bar{A}_{e_i}) + \sum_{l=W+1}^{W+T} [\Pr(FP|A_{e_i,E})\Pr(A_{e_i,E})] \quad (7)$$

Active error rate, and

$$\Pr(FP_{UD}) = \sum_{l=W+1}^{W+T} [\Pr(FP|A_{e_i,U})\Pr(A_{e_i,U})] = \sum_{l=W+1}^{W+T} [\Pr(A_{e_i,U})] \quad (8)$$

UD error rate.

By simple probabilistic calculation, we have

$$\begin{aligned}\Pr(FP_{\text{Active}}) &= \Pr(FP|\bar{A}_{e_i})\Pr(\bar{A}_{e_i}) + \sum_{l=W+1}^{W+T} [\Pr(FP|A_{e_i,E})\Pr(A_{e_i,E})] \\ &= (1-(1-1/m)^{kw})^k, \end{aligned} \quad (9)$$

where w is the number of distinct elements currently represented by the b_DBF, m is the number of counters, and k is the number of hash functions. Consequently,

$$\begin{aligned}\Pr(FP) &= \Pr(FP_{UD}) + \Pr(FP_{\text{Active}}) \\ &= \sum_{l=W+1}^{W+T} \Pr(A_{e_i,U}) + (1-(1-1/m)^{kw})^k. \end{aligned} \quad (10)$$

When $W > T$, (see the Appendix for detail)

$$\Pr(FP_{UD}) = \sum_{l=N+1}^{N+T} \Pr(A_{e_i,U}) < T/(2\times W). \quad (11)$$

In addition, since $(1-(1-1/m)^{kw})^k$ is monotonically increasing as w increases and $w \leqslant W+T$,

$$\begin{aligned}\Pr(FP) &\leqslant T/(2\times W) + (1-(1-1/m)^{k(W+T)})^k \\ &\approx T/(2\times W) + (1-e^{-k(W+T)/m})^k. \end{aligned} \quad (12)$$

This completes the proof. □

From (10), we can see that for a distinct element e_i (it has no predecessor or the number of elements between it and its nearest predecessor is greater than W), it may be reported as a duplicate for two reasons. First, it is likely that the counters $\{DBF[h_1(e_i)], \ldots, DBF[h_k(e_i)]\}$ are set by elements other than e_i. We call this kind of errors *Active error* and its probability *Active error rate*. Second, the nearest predecessor of e_i is out of sliding windows but still in a *UD* block. We call this kind of errors *UD error* and its probability *UD error rate*.

Furthermore, we can see that when $T = 1$, the *UD error rate* is 0 and $\Pr(FP) = (1-(1-1/m)^{kw})^k$, the FPR of a b_DBF is equal to that of a DBF under the same conditions. Therefore, DBF is only a special case of b_DBF when $T = 1$.

Corollary 4. *Given a b_DBF with m counters to detect duplicates over a W-width sliding windows, the Active error rate and the UD error rate are monotonically increasing as the block size T increases.*

Proof. When the number of counters in a b_DBF is fixed, the number of bits allocated to each counter is also fixed. If the block size T increases, the number of distinct elements represented by b_DBF can also increase, and from (9), we can conclude that $\Pr(FP_{\text{Active}})$ is increasing. In addition, according to (11), it is not hard to prove that the probability of event $A_{e_i,U}$ will also increase. Therefore, $\Pr(FP_{\text{Active}})$ and $\Pr(FP_{UD})$ are monotonically increasing as the value of T increases. □

In practice, given the size of sliding window W and the memory space G bits, the smaller the value of $d(= \lceil \log(W/T+2) \rceil)$ is, the larger the value of T is and consequently the larger the *UD error rate* is according to Corollary 4. Conversely, the larger the value of d is, the smaller the number of counters m $(= G/d)$ is and consequently the smaller the *Active error rate* is. The following theorem gives the proper value of d that minimizes the upper bound of FPR.

Theorem 4. *Given memory space G $(= m \times d)$ bits and the size of sliding window W, if the block size $2 \leqslant T < W$, the upper bound of Pr(FP)in b_DBF can be minimized to $(1/2)^{\sqrt{G\times\ln(2)/W}-1}$ when $d = k = \sqrt{G \times \ln(2)/W}$.*

Proof. Since $d = \lceil \log(W/T+2) \rceil \geqslant 2$, $W/(2^{d-1}-1) \geqslant T \geqslant W/(2^d-2)$. On one hand, the upper bound of *UD error rate* $\Pr(FP_{UD}) = T/(2\times W) \leqslant 1/(2^d-2) \approx (1/2)^d$. On the other hand, since the *UD error rate* is independent of k, we can minimize the *Active error rate* by setting $k = m \times \ln(2)/(W+T) = G \times \ln(2)/(W \times d + T \times d) \approx G \times \ln(2)/(W \times d)$, and the *Active error rate*

$$\begin{aligned}\Pr(FP_{\text{Active}}) &= (1-e^{-k(W+T)/m})^k \\ &= (1/2)^{G\times\ln(2)/(W\times d+T\times d)} \\ &\approx (1/2)^{G\times\ln(2)/(W\times d)}. \end{aligned}$$

Therefore, the upper bound of FPR

$$\begin{aligned}\Pr(FP) &= T/(2\times W) + (1-e^{-k(W+T)/m})^k \\ &\approx (1/2)^d + (1/2)^{G\times\ln(2)/(W\times d)}.\end{aligned}$$

From derivation on d, we can obtain that the upper bound of FPR will be minimized to $(1/2)^{\sqrt{G\times\ln(2)/W}-1}$ when $d=k=\sqrt{G\times\ln(2)/W}$. This proves the theorem. □

From Theorem 4, we can conclude that the upper bound of FPR is only dependent on the ratio between the amount of space and the size of sliding window. In order to confirm our conclusion, we use MATLAB 7.0 to calculate the upper bound of FPR for various d values when $G=2^{27}$ bits, $W=2^{20}$, and the number of hash functions is constrained to $k=G\times\ln(2)/(W\times d)$. The result is illustrated in Fig.3. Compared with DBF where $d=21$ and $\Pr(FP)\approx 5.3\%$, the upper bound of FPR of b_DBF is only about 0.39% when $d=9$. Let $\gamma=G/W$, note that in the optimal case, $d=\sqrt{\gamma\ln(2)}$.

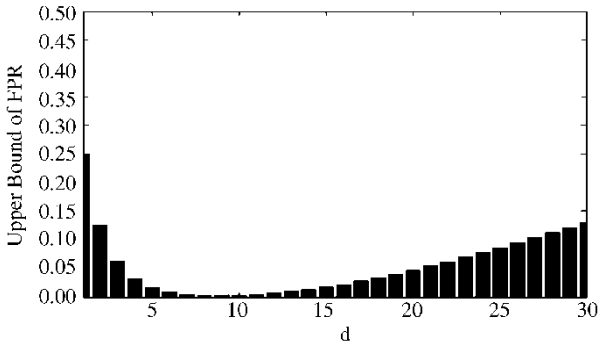


Fig.3. Upper bound of FPR of b_DBF for various d, with $G=2^{27}$, $W=2^{20}$, $k=G\times\ln(2)/(W\times d)$.

5.3 Setting of Parameters T and k

Up on the above discussion, to choose a proper value of T, we need to consider the performances of b_DBF. In terms of time complexity, T should be set as large as possible, since the larger the value of T is set, the smaller the number of decreases of all counters and the number of bits assigned to each counter will be; in addition, k should be set as small as possible, since the smaller the value of k is set, the smaller the number of hashing operations will be. However, for the false positive rate (FPR), since the upper bound of FPR is minimized when $d=k=\sqrt{\gamma\ln(2)}$ according to Theorem 4, the value of T should be set between $W/(2^{d-1}-1)$ and $W/(2^d-2)$. It seems that T and k should be properly set to balance the performances of our algorithm. Since our goal is to minimize the error rate, we first set $d=k=\sqrt{\gamma\ln(2)}$.

It is easy to find that a DBF can be derived from a b_DBF by setting $T=1$. Therefore, when $W/(2^d-2)\leqslant 1$, i.e., $G\geqslant W\times\log(W+2)/\ln(2)$, we can get $T=1$. Below we only discuss the special circumstances when $W/(2^d-2)>1$ and $\gamma<20$ (if higher time cost can be tolerated, larger values of γ can be applied). We set $k=d=\sqrt{\gamma\ln(2)}$ to guarantee the minimum upper bound of FPR according to Theorem 4, and T the minimum value that meets $d=\lceil\log(W/T+2)\rceil$, i.e., $T=W/(2^d-2)$, to reduce the practical FPR according to Corollary 4. Then, the minimum upper bound of FPR is $(1/2)^{\sqrt{G\times\ln(2)/W}-1}$ and the amortized time complexity for processing each element, denoted by τ, is

$$\begin{aligned}m/T+2k &= (m\times d)/(T\times d)+2d \\ &= G/(W\times d/(2^d-2))+2d \\ &= \gamma/(d/(2^d-2))+2d.\end{aligned}$$

Since $d=\sqrt{\gamma\ln(2)}$, $\tau=O(\sqrt{\gamma}\times 2^{\sqrt{\gamma\ln(2)}})$.

In comparison with DBF, b_DBF reduces τ from $O(G/\log W)$ to $O(\sqrt{\gamma}\times 2^{\sqrt{\gamma\ln(2)}})$, the number allocated to each counter from $O(\log W)$ bits to $O(\sqrt{\gamma\ln(2)})$ bits, and the upper bound of FPR from $(1/2)^{\gamma\times\ln(2)/\log W}$ to $(1/2)^{\sqrt{\gamma\times\ln(2)}-1}$.

However, in normal circumstances where $r\geqslant 20$ (containing the circumstance that $T=1$), this means that we use a larger space to minimize the upper bound of FPR against the size of sliding windows. Then, we cannot choose T like in special circumstances. Since it may be hard to tolerate a high amortized time cost, we will introduce an optimization algorithm in Subsection 5.5 which can further reduce the amortized time complexity while maintaining the minimum upper bound of FPR.

5.4 Resolving Time-Bottleneck by Averaging Counter Decreases

Algorithm 1 may have limitation in some stream scenarios, since the number of operations fluctuates largely in different iterations. If a *UC* block becomes an *Active* block within one iteration, we will decrease all counters of the array and consequently the iteration needs $m+2k$ operations; otherwise, the iteration only needs $2k$ operations. This may cause a time-bottleneck in some high-rate stream scenarios. Therefore, we improve Algorithm 1 by averaging counter decreases. Below is the detail. We divide all counters of the array into

non-overlapping parts and all the parts have the same number of counters except for the last part. As in Fig.4, we divide an m-size array into T (equal to the block size) parts $(P_1, P_2, \ldots, P_T)$. Let S_{P_i} $(T \geqslant i \geqslant 1)$ denote the number of counters in P_i, $S_{P_i} = \lceil m/T \rceil$ when $T > i \geqslant 1$ and $S_{P_T} = m - (T-1) \times \lceil m/T \rceil$. Therefore, $m = \sum_{i=1}^{T} S_{P_i}$.

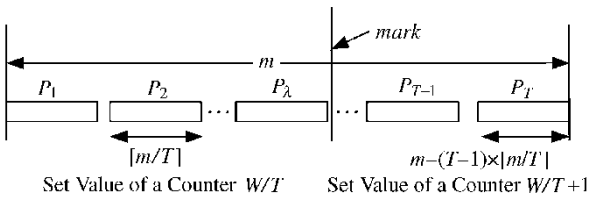


Fig.4. Averaging counter decreases by dividing all counters of a b_DBF into T parts.

When a new element e_i arrives, we only decrease all counters in P_λ, where $\lambda = (i+1) \bmod T$. Thus, by decreasing all counters in one of the parts within each iteration, we can amortize the time cost of decreasing all m counters, and each iteration needs no more than $\lceil m/T \rceil + 2k$ operations.

In order to distinguish the parts that are decreased during a UA interval, we need a mark to divide them. At the same time, we also need to change the operation of setting k counters. If a new element e is mapped to a counter DBF$[h_i(e)]$ $(k \geqslant i \geqslant 1)$ in P_j where $mark > j \geqslant 1$, we set DBF$[h_i(e)] = W/T$; otherwise, we set DBF$[h_i(e)] = W/T + 1$.

Although the optimization does not reduce the overall space complexity, time complexity and error rate, it can extend b_DBF to more rigorous stream applications that require processing each element in a very short time.

5.5 Reducing Time Complexity by Delaying Decreasing

Algorithm 1 cannot demonstrate its superiority when $\gamma \geqslant 20$ because of its high time cost. The time consumption of each iteration in b_DBF mainly depends on the process of decreasing all counter values of DBF. Although the decreasing process has been delayed by a block size in Algorithm 1, the process can be further delayed in order to avoid the slowness in updating. Hence, we propose such an optimization scheme that delays the decreasing process in b_DBF.

We introduce another threshold $Th = c \times W/T$ to denote the number of delay intervals, where each interval is equal to a UA interval and c is a constant that is very close to 1. In addition, we define a variable VT which is used for judging whether a new element is a duplicate and when to decrease all counter values.

In order to minimize the upper bound of FPR, we set $d = k = \sqrt{\gamma \ln(2)}$ $(= \lceil \log(W/T + Th + 2) \rceil)$ as in special circumstances. Unlike in special circumstances, we set $T = \lceil W/(2^{d-1} - 1) \rceil$ instead of $W/(2^d - 2)$ to make room in each counter to store the number of delay intervals. Furthermore, the minimum and maximum values of each counter are 0 and $W/T + Th + 1 (= 2^d - 1)$ respectively. Thus, $Th = c \times W/T = 2^{d-1} \approx W/T$, where $c = 2^{d-1}/(W/T) \approx 1$.

The details follow Algorithm 2. The initial values of the counters are all zero. For each newly arrived element in the stream, there are also three operations: first, a newly arrived element can check if it is duplicate by probing whether none of the k counters it is hashed to has a value smaller than VT; second, delay decreasing all counters until the value of VT reaches Th; third, set the same k counters as in the first operation to $W + VT + 1$.

Algorithm 2. Optimization by Delaying Decreasing
Input: the threshold T, Th and a sequence of elements $S = e_1, e_2, \ldots, e_N, \ldots$
Output: a sequence of "Yes/No" corresponding to each input element.

```
Initialize DBF[1], ..., DBF[m] = 0, Iteration = 0 and VT = 0
for each e_i ∈ S do {
    Probe k counters DBF[h_1(e_i)], ..., DBF[h_k(e_i)]
    if none of the above k counters has a value smaller than VT then
        IsDuplicate = "Yes"
    else
        IsDuplicate = "No"
    end if
    if (Iteration mod T) = 0 then
        VT = (VT + 1) mod Th
        if VT = 0 then
            for each counter DBF[j] ∈ {DBF[1], ..., DBF[m]} do
                if DBF[j] ⩽ Th then
                    DBF[j] = 0
                else
                    DBF[j] = DBF[j] − Th
                end if
        end if
    end if
    for each counter ∈ {DBF[h_1(e_i)], ..., DBF[h_k(e_i)]} do
        DBF[h(e_i)] = W/T + VT + 1
    Iteration = Iteration + 1
    Output IsDuplicate
}
```

According to Theorem 4, since $d = k = \sqrt{\gamma \ln(2)}$, this optimization algorithm, unlike Algorithm 1, maintains the upper bound of FPR at minimum. Most importantly, it has good performances in amortized time complexity.

Theorem 5. *Given the ratio between the size of memory space and the size of sliding windows, γ, when maintaining the minimum upper bound of the false positive rate, the amortized time complexity for processing each element is $\tau = O(\sqrt{\gamma})$.*

Proof. Since we decrease all counters every $Th \times T$ iterations, and $Th = c \times W/T$, the average number of operations in one iteration is the sum of the average number of operations of updating b_DBF and the average number of operations of detection process, i.e., $\tau = m/(T \times Th) + 2k = G/(W(c \times d)) + 2k = \gamma/(c \times d) + 2k$. Because $d = k = \sqrt{\gamma \ln(2)}$, $\tau = O(\sqrt{\gamma}) = O(\sqrt{G/W})$. □

Theorem 5 states that the amortized time complexity τ increases monotonically as G increases, which seems illogical since G is the size of memory space used. This is because the more memory is used, the more hash functions need to be employed for mapping elements, so the higher τ will be resulted. This is in fact true for all *BF* based schemes. We now explain this in detail. In the original Bloom Filter (BF), when given the size of bit array G (memory space), the number of hash functions k, and the number of distinct elements n, $\tau = O(k)$ and the false positive rate of BF is $\Pr(FP) = (1-(1-1/G)^{kn})^k \approx (1-e^{-kn/G})^k$. $\Pr(FP)$ is minimized to $\Pr(FP)_{\min} = (1/2)^k = (0.6185)^{G/n}$ when $k = G\ln(2)/n$[14]. Clearly, $\tau = O(G/n)$ when $\Pr(FP)$ is maintained at minimum, which is monotonically increasing as the memory space G increases. On the other hand, if set k to be a constant, in this case the false positive rate of BF is not minimized and τ becomes $O(1)$.

Our b_DBF shares the same property as BF — its amortized time complexity for processing each element is $\tau = O(m/(T \times Th) + 2k)$ and false positive rate is $\Pr(FP) = T/(2 \times W) + (1 - e^{-k(W+T)/m})^k$, where m is the number of counters in b_DBF, k is the number of hash functions, T is the size of block and Th is the number of delay intervals. The false positive rate is minimized to $\Pr(FP)_{\min} = (1/2)^{\sqrt{G \times \ln(2)/W} - 1}$ when the number of bits in each counter $d = k = \sqrt{G \times \ln(2)/W}$ and $T \times Th \approx W$. Since in this case $m = G/d = G/\sqrt{G \times \ln(2)/W} = \sqrt{G \times W/\ln(2)}$, $\tau = O(\sqrt{G/W})$. Therefore, when the false positive rate is maintained at minimum, this time complexity also monotonically increases as memory space G increases, though at a lower rate than BF. Otherwise, if k is set to be a constant, τ becomes $O(1)$ when $T \times Th \approx m$ and $O(m/(T \times Th))$ otherwise which also increases monotonically as G increases.

In most streaming scenarios, duplicate detection usually requires large sliding widows because the amount of data arriving in unit time is quite large from a data stream. Our work follows Metwally's work[3] to target at the problem of detecting duplicates over large windows using small memory space in data streams. When memory space G is very large and the size of sliding window is very small, e.g., $W = 1$, we can handle this situation simply by buffering or caching instead of applying our method (or any other BF-based methods) as stated in the introduction of the paper.

6 Experimental Results

In this section we discuss the results of our synthetic experiments on the proposed algorithm. The goals of the experiments of using b_DBF are to exploit the tradeoffs between space and time, on one hand, and between space and error rate, on the other hand. The larger memory space used to test for duplicates, the smaller the probability of producing false positive errors. However, the number of operations increases dramatically as the space usage of b_DBF increases when the size of sliding window is given. Throughout the experiments, we use the ratio between the size of the space and the size of sliding window as the independent axis in our graphs, and report the results accordingly. From them, the tradeoff between the space usage and the error rate can be seen. To evaluate our work, we have compared our b_DBF with CBF (Counting Bloom Filter) to detect duplicates over sliding windows.

In addition, the experiments validate our arguments in Subsection 5.2 and Subsection 5.3. As we mentioned above, we expect that the practical error rate for duplicate detection is close to the upper bound of the theoretical error rate calculated. The experiments support this proposition.

6.1 Data Sets

It is infeasible to test our approach over sliding windows with real data streams, since measuring the accuracy of our approach would require exact identification of all duplicates for every window. For example, if the stream size is 5×2^{20}, and the window size is fixed to 2^{20} elements, then the number of windows that cover the given stream is $5 \times 2^{20} - 2^{20} + 1 \approx 4 \times 2^{20}$. Thus, to check the error rate for our solution, we run the exact solution about 4×2^{20} times, and compare it to the results of the approximate solution. To overcome this

problem, we limit the experiments to only a small size of sliding windows and only run experiments on a synthetic data stream. This is in line with many previous studies[3,4].

There is another reason for using such a synthetic stream. The fraction of duplicates in the data stream is a dominant factor affecting the error rates, because FNs are only generated by duplicates and FPs by distincts. For example, a data stream full of duplicates will cause many FNs, while a stream consisting of all distincts causes no such errors at all. Since CBF and our algorithm could have FPs (false positives) but no FNs (false negatives), it is difficult and unnecessary to compare b_DBF with CBF over a large real data stream with duplicates. To fairly and effectively compare our method with CBF, we use a special synthetic stream of 10×2^{20} elements for our sliding window experiments. The elements are all distinct. Thus, all the duplicates output by the algorithm are erroneous. We use integers as data values.

6.2 Comparisons

Two sets of tests were conducted over the 2^{12}-width sliding windows. These tests were implemented using hash functions of modulo/multiply type from [17]: given a value v, its hash value $H(v)$, $0 \leqslant H(v) \leqslant m$, is computed by $H(v) = \lceil m(av \bmod 1) \rceil$, where a is taken uniformly at random from [0, 1].

In the first set of tests, we compare the FPR of our algorithm with that of CBF. In order to implement a fair comparison, we assign the same space to the two algorithms. Since CBF requires additional space to keep the arrival order information of elements represented by it, this space could be used, instead, to reduce the FPR within our algorithm. We suppose the additional space is equal to the space required by CBF. In addition, for simplicity, we assign the same number of bits as for the counter in b_DBF to each counter in CBF and do not consider the counter overflow. Fig.5 shows comparisons of the FPR obtained by using the additional memory to increase the size of the array of b_DBF within our algorithm versus that by using it to keep the order information within CBF while the ratio between the size of the space and the size of sliding window increases. From Fig.5, it is clear that the FPR of b_DBF is much lower than that of CBF and the practical FPR of b_DBF approaches the upper bound of theoretical FPR as the ratio between the size of space and the size of sliding window increases.

In the second set of tests, we observe the number of operations for each new element within CBF and b_DBF for various ratios between the size of space and the size of sliding window. From Fig.6, we can find the time complexity of b_DBF is much lower than that of CBF.

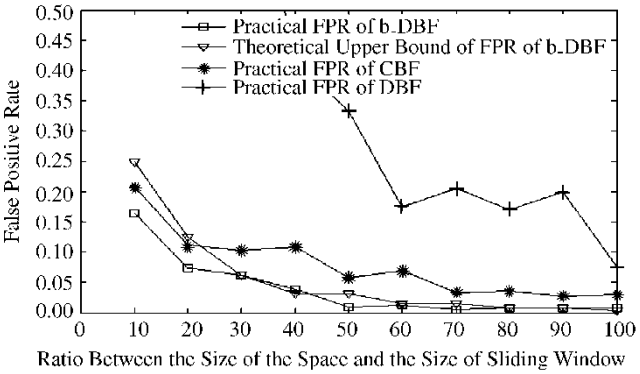


Fig.5. Comparisons of FPR between b_DBF and CBF for various ratios between the size of space and the size of sliding window using a synthetic data stream.

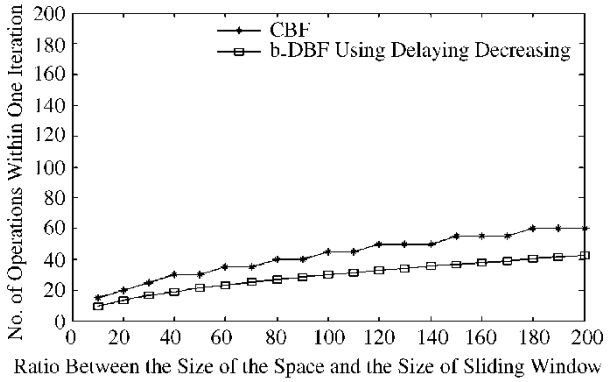


Fig.6. Comparisons of number of operations between b_DBF and CBF for various ratios between the size of space and the size of sliding window using a synthetic data stream.

7 Conclusions

In this paper, we studied the problem of approximately detecting duplicates over sliding windows. We introduced a data structure, namely, Decaying Bloom Filter (DBF), and a simple and efficient algorithm for duplicate detection. Our DBF avoids counter overflow and guarantees that all stale elements sliding out of the sliding window will be deleted. Thus, it avoids using additional space to keep the arrival information of elements in the current window as did in Counting Bloom Filter, resulting in a significant reduction in memory space. Based on DBF, our duplicate deletion algorithm works by dividing a data stream into non-overlapping blocks and processing a block instead of a single data

element. Experimental results have verified our theoretical analysis and showed that the proposed method achieves better performances in both deletion accuracy and execution time than previous results.

References

[1] Babcock B, Babu S, Datar M, Windom J. Model and issues in data stream systems. In *Proc. Principles of Database Systems* (*PODS*), Wisconsin, USA, June 2002, pp.1–16.

[2] Golab L, Ozsu M T. Issues in data stream management. *SIGMOD Record*, June 2003, 32(2): 5–14.

[3] Metwally A, Agrawal D, Abbadi A E. Duplicate detection in click streams. In *Proc. 14th Int. Conf. World Wide Web*, Chiba, Japan, May 2005, pp.12–21.

[4] Deng F, Rafiei D. Approximately detecting duplicates for streaming data using stable Bloom filters. In *Proc. ACM SIGMOD Conf.*, New York, USA, June 2006, pp.25–36.

[5] Anupam V, Mayer A, Nissim K, Pinkas B, Reiter M. On the security of pay-per-click and other web advertising schemes. In *Proc. 8th Int. Conf. World Wide Web*, Toronto, Canada, May 1999, pp.1091–1100.

[6] Heydon A, Najork M Mercator: A scalable, extensible web crawler. In *Proc. 8th Int. Conf. World Wide Web*, Toronto, Canada, May 1999, pp.219–229.

[7] Broder A Z, Najork M, Wiener J L. Efficient URL caching for world wide web crawling. In *Proc. 12th Int. Conf. World Wide Web*, Budapest, Hungary, May 2003, pp.680–689.

[8] Cisco network accounting services white paper. Cisco System Inc, 2002, pp.1–20, http://www.cisco.com /warp/public/cc/pd/iosw/prodlit/nwactwp.pdf.

[9] Garcia-Molina H, Ullman J D, Widom J. Database System Implementation. Upper Saddle River: Prentice Hall, Inc., NJ, USA, 1999.

[10] Bilenko M, Mooney R J. Adaptive duplicate detection using learnable string similarity measures. In *Proc. ACM SIGKDD Conf.*, Washington DC, USA, August 2003, pp.39–48.

[11] Weis M, Naumann F. Dogmatix tracks down duplicates in XML. In *Proc. ACM SIGMOD Conf.*, Baltimore, Maryland, USA, June 2005, pp.431–442.

[12] Ananthakrishna R, Chaudhuri S, Ganti V. Eliminating fuzzy duplicates in data warehouses. In *Proc. 28th Int. Conf. Very Large Data Bases*, Hong Kong, China, 2002, pp.586–597.

[13] Chaudhuri S, Ganti V, Motwani R. Robust identification of fuzzy duplicates. In *Proc. 21st Int. Conf. Data Engineering* (*ICDE*), Tokyo, Japan, 2005, pp.865–876.

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

[15] Fan L, Cao P, Almeida J, Broder A Z. Summary cache: A scalable wide-area Web cache sharing protocol. *IEEE/ACM Trans. Networking*, 2000, 8(3): 281–293.

[16] Mitzenmacher M. Compressed Bloom filters. In *Proc. 20th ACM Symposium on Principles of Distributed Computing* Netw. Rhode Island, 2001, pp.144–150.

[17] Cohen S, Matias Y. Spectral Bloom filters. In *Proc. ACM SIGMOD Conf.*, California, USA, June 2003, pp.241–252.

[18] Whang K, Zenden B T V, Taylor H M. A linear-time probabilistic counting algorithm for database applications. *ACM Trans. Database Syst.*, 1990, 15(2): 208–299.

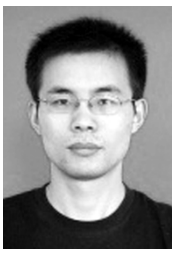
[19] Cheng K, Xiang L, Iwaihara M. Time-decaying Bloom filters for data streams with skewed distributions. In *Proc. 15th Workshop on Research Issues in Data Engineering: Stream Data Mining and Applications* (*RIDE-SDMA*), Tokyo, Japan, 2005, pp.63–69.

[20] Cohen E, Strauss M. Maintaining time-decaying stream aggregates. In *Proc. Principles of Database Systems* (*PODS*), San Diego, California, USA, June 2003, pp.223–233.

[21] Cormode G, Tirthapura S, Xu B. Time-decaying sketches for sensor data aggregation. In *Proc. Principles of Distributed Computing* (*PODC*), Portland, Oregon, USA, May 2007, pp.215–224.

[22] Arasu A, Manku G. Approximate counts and quanlites over sliding windows. In *Proc. Principles of Database Systems* (*PODS*), Paris, France, June 2004, pp.286–296.

Hong Shen is currently a specially appointed professor in University of Science & Technology of China, and holds a permanent position of professor (chair) of computer science in the University of Adelaide, Australia. He received the B.S. degree from Beijing University of Science & Technology, M.S. degree from University of Science & Technology of China, Ph.Lic. and Ph.D. degrees from Abo Akademi University, Finland, all in computer science. He was a professor and chair of the Computer Networks Laboratory in Japan Advanced Institute of Science and Technology (JAIST) during 2001∼2006, and professor (chair) of Compute Science at Griffith University, Australia, where he taught 9 years from 1992. With the main research interests in parallel and distributed computing, algorithms, data mining, high performance networks and multimedia systems, he has published more than 200 papers including over 100 papers in international journals such as a variety of IEEE/ACM transactions. Prof. Shen was the co-recipient of 1991 National Education Commission Science and Technology Progress Award and 1992 Chinese Academy of Sciences Natural Sciences Award. He serves on the editorial board of several journals including IEEE Transactions on Computers.



Yu Zhang is currently an M.S. candidate at Department of Computer Science, University of Science and Technology of China. He received the B.S. degree from Anhui University in 2005. His research interests include data stream management, query processing and optimization.

Appendix

Proof of Upper Bound of UD Error Rate

We start by assuming that there is a multiple set M, which comes from a universe U, and elements are orderly and uniformly distributed in this multiple set.

We divide the set M into B ($\geqslant 2$) blocks of the same size T and number them by 1, 2, $\ldots, B$. Clearly, $|M| = B \times T$. For any element $e \in U$, let A_i denote the event that e has a duplicate in block i ($1 \leqslant i \leqslant B$), and $\Pr(A_i)$ denote the probability of A_i. Since elements in M are uniformly distributed, $\Pr(A_i)$ is a constant and $\Pr(A_i) = \Pr(A_j)$ ($1 \leqslant i, j \leqslant B$).

Now let us consider the event A that e has a duplicate in block B but no duplicate in blocks whose numbers are smaller than B. Thus, the probability of this event is

$$\Pr(A) = (1 - \Pr(A_i))^{B-1} \times \Pr(A_i).$$

By derivation on $\Pr(A_i)$, we find that the $\Pr(A)$ can be maximized to $(1-1/B)^{B-1} \times (1/B)$ when $\Pr(A_i) = 1/B$. Therefore,

$$\Pr(A) \leqslant (1 - 1/B)^{B-1} \times (1/B).$$

Since $B \geqslant 2$, $(1 - 1/B)^{B-1}$ is no more than $1/2$ and monotonically decreases as B increases. We have

$$\Pr(A) \leqslant 1/(2 \times B) = T/(2 \times |M|).$$

If we take $|M|$ as the size of sliding windows, then we can conclude that $\Pr(FP_{UD})$ is no more than $T/(2 \times |M|)$.

When $B \geqslant 100$, since $(1 - 1/B)^{B-1} \approx e^{-1} = 0.37$, the *UD error rate* is no more than 0.37%. Fig.7 depicts the values of $\Pr(A)$ for different $\Pr(A_i)$ values with $B = 100$, which shows that $\Pr(A)$is maximized to 0.37% when $\Pr(A_i)$ is about 0.01.

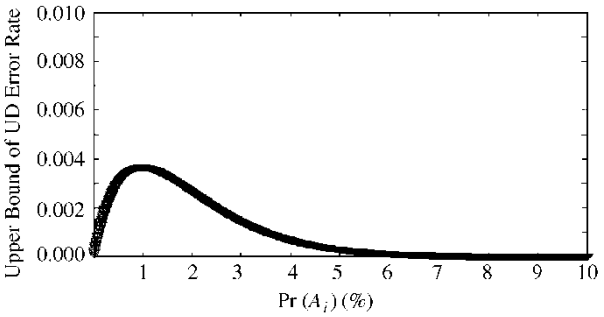


Fig.A1. Comparisons of the UD Error Rate for various $\Pr(A_i)$ with $B = 100$.