Improved Decaying Bloom Filter for Duplicate Detection in Data Streams Over Sliding Windows

Xiajun Wang1, Hong Shen1;2 (Corresponding author.)
1Department of Computer Science, University of Science and Technology of China, China
2School of Computer Science, University of Adelaide, Australia
hongshen@ustc.edu.cn

Abstract—Approximate duplicate detection based on the Decaying Bloom Filter (DBF) for data streams over sliding windows (DDMDBF) [1] is an effective technique, but may have a large false positive rate. Because it simply takes a querying element to be duplicated when the counters that this element is hashed to are non-zero, while neglects the actual values of the counters. In this paper, we propose a new data structure, Flag Decaying Bloom Filter (FDBF), which can maintain duplicate information more accurately by extending DBF with one additional flag bit for each integer counter. Then we propose an efficient approximate duplicate detection method (DDMFDBF) based on FDBF that reduces the false positive rate (FPR) \( p \) (0 < \( p \) < 1) of DDMDBF by a factor of \( p^\frac{1}{1-2} \) for approximately same bit space. Experimental results on synthetic data validate the analytical results on the efficiency and accuracy of our method.

Keywords—Counting Bloom Filter, Decay Bloom Filter, Flag Decaying Bloom Filter, False Positive, Duplicate Detection

I. INTRODUCTION

With the boost of the Internet, online monitoring of data streams has become an important issue in data stream management. This issue has applications in many domains: databases management, network monitoring, data mining, probability and statistics. Detecting duplicates over data streams is a very interesting problem and has many applications. For example, in URL Crawling [2][3], search engines periodically crawl the web to enlarge their web page database. For the URL of a web page, which is usually extracted from a crawled web page, a search engine must decide whether to fetch the URL from remote web site or not. Given that the set of URLs perform large volume of searches within limited time, so it must probe its URL archive to find whether the URL is in or not by an efficient online query method. In another example [4][5], duplicate detection can also be applied to select or query distinct IP addresses on the network within the past several hours which is useful to analyse network user behavior and network traffic.

According to the way the data stream is handled [6], there are generally two basic classes of duplicate detection over data streams: Duplicate Detection over a Landmark Window and Duplicate Detection over a Sliding Window. The former one focuses on duplicate elements that have occurred since the occurrence of a specific landmark usually defined in terms of time units (e.g. in one day or one week basis) or in terms of the number of element that has been observed since the former landmark. The later one focuses on the duplicate elements that occurred in the last \( W \) elements.

In a data stream, user may be more interested in new data elements than old ones, so detecting duplicates over a sliding window which focuses on the most recent \( W \) elements is usually more reasonable than over a landmark window. Several data structures and related algorithms have been proposed recently to detect duplicate over a sliding window. Among them, Counting Bloom Filter [6] is used most commonly. The CBF uses an array of \( n \) integer counters to substitute the original bit vector in traditional Bloom Filter (BF) [7]. On the CBF basis, Shen and Zhang proposed DBF and an efficient algorithm for approximately detecting duplicates over sliding windows (DDMDBF) [1]. Unlike the duplicate detection method based on CBF, DDMDBF doesn’t need additional space to keep the arrival information of elements, so it can build more integer counters than CBF (given that the total memory space is limited) and has lower false positive rate than the duplicate detection method based CBF. When a newly incoming element is hashed to \( k \) counters, DDMDBF sets the \( k \) counters to \( W \). When the sliding window is full, DDMDBF decreases all non-zero counters by one. So DDMDBF will not have the counter overflow problem.

DDMDBF may have a large false positive rate, because it simply take an element \( e \) to be duplicated when all \( k \) counters \( \{C_i, \cdots, C_k\} \) what the element \( e \) is hashed to are non-zero \( \{C_i \neq 0, \cdots, C_k \neq 0\} \), which neglects the actual values of the counters. When the \( k \) counters are non-zero and not equal, this is unreasonable and may produces many false positive errors. In detail, if the \( k \) counters \( \{C_i \neq 0, \cdots, C_k \neq 0\} \) are not equal and \( e \) has duplicate (assumed \( e \in \varnothing \) in the latest \( W \) elements, a counter \( C_i \)\( \text{min}(C_i, \cdots, C_k) \), \( C_i \in \{C_i, \cdots, C_k\} \) is
definitely hashed by at least two of the latest \( W \) elements (including \( e' \) and some elements which come after \( e' \)). In other words, if the condition is true that the \( k \) counters \( \{ C_i \neq 0, \cdots, C_k \neq 0 \} \) are not equal and \( e \) has duplicates in the latest \( W \) elements, the conclusion is true that \( C_j \min \{ C_i, \cdots, C_k \} (C_i \in \{ C_i, \cdots, C_k \}) \) is set by at least two of the latest \( W \) elements. The conclusion will be false if the condition is false. Thus \( e \) can be considered distinct when the \( k \) counters \( \{ C_i \neq 0, \cdots, C_k \neq 0 \} \) are not equal and \( \{ C_i \in \{ C_i, \cdots, C_k \} \} \) is set once

Based on the above consideration, we propose a new data structure DDMDBF as an extension of DBF and a duplicate detection method DDMFDBF based on FDBF. FDBF has the same main framework like DBF and CBF, but it introduces one additional flag bit \( I \) for each counter \( C_i \) and use \( I \) to indicate whether \( C_i \) has been hashed by elements from \( bn \geq 2 \) blocks or not. By this DDMFDBF take a element \( e \) to be duplicated or not based on the integer counters \( \{ C_i, \cdots, C_k \} \) that \( e \) is hashed to and the according flag bits \( \{ I_i, \cdots, I_k \} \). Thus, DDMFDBF stores duplicate information more accurately than DBF and eliminates many false positive (FP) errors when the \( k \) hashed counters are non-zero and not equal. DDMFDBF reduces much less false positive rates than DDMDBF [1]. By analytical and experimental results, we prove that: compared with DDMDBF [1], given block number \( B (B \gg 2) \) and window size \( W \), DDMFDBF reduces the false positive rate (FPR) \( p (0 < p < 1) \) of DDMDBF by a factor of \( p^1/\sqrt{2} \) for approximately same bit space \((G + G \log(B) \approx G)\)

This paper is organized as follows. In Section 2, we will give FDBF definition. In Section 3, we will present a duplicate detection method DDMFDBF over sliding windows based on FDBF and analytical results on the false positive rate (FP) of DDMFDBF. We report the experimental results of our method over synthetic data in Section 4, followed by the conclusions in section 5.

II. FLAG DECAYING BLOOM FILTER

A traditional Bloom Filter [7] uses an \( n \)-bit vector and \( k \) hash functions: \( h_1, h_2, \cdots, h_k, h(x) \in \{ 1 \cdots n \} \). For each element of a data set \( (e \in S, |S| = w) \), it sets the bits at positions \( h_1 (e), h_2 (e), \cdots, h_k (e) \) to 1. After storing each element in \( S \), for a newly incoming element \( e \) we can probe the \( k \) bits at positions \( h_1 (e'), h_2 (e'), \cdots, h_k (e') \) to answer if \( e \) has occurred in \( S \) or not with only a false positive rate \((1-1/n)^k \approx (1-e^{-\delta/n}) \). Both CBF [6] and DBF [1] substitute the \( n \)-bit vector in BF with \( n \) integer counters. The difference between them is that, CBF increases the counters \( y \) Note that through this paper we assume \( \log \) to be \( \log_2 \) and \( l \) to be \( \log \) while DBF decreases them by one. When using a DBF to detect duplicates over a sliding window, DDMFDBF [1] applies three steps. Considering that DDMFDBF adopts one additional flag bit for each integer counter, we modify the three steps in DBF as follows. Given the sliding window size \( W \) and a newly incoming element \( e \), the modified three steps are: (the initial values of all counters and flag bits are set to zero.)

Step 1: \( e \) is hashed to the counters \( C[h_1(e)], C[h_2(e)], \cdots, C[h_k(e)] \) at positions \( h_1(e), h_2(e), \cdots, h_k(e) \) with their according indicating bits: \( I[h_1(e)], I[h_2(e)], \cdots, I[h_k(e)] \), if one of the counters is zero, \( e \) is reported as distinct; if all of the counters are non-zero, we find the minimum value \( C_{\min} = \min \{ C[h_1(e)], \cdots, C[h_k(e)] \} \), then we investigate that whether the indicating bit \( I[h(e)] \) of the counter \( C[h(e)] \) \( C_{\min} \), \( j \in \{ 1, k \} \) is 1 and report \( e \) as duplicate if so and distinct otherwise.

Step 2: all non-zero counters in FDBF are decreased by one, then if the value of a counter is one we clear its according indicating bit. (those counters with value one are set by the last element in current sliding window so they can not be set by at least two elements in the current sliding window)

Step 3: set \( I[h(e)] \) to 1 \((J = 1 \cdots k)\) if \( C[h(e)] < W \), then set all \( I[h_j(e)] \), \((J = 1 \cdots k)\) to \( W \) (if the counters before probing \( e \) is already non-zero, then it is set twice now)

We call the first step duplicate detection process and the last two update process. It is easy to see that FDBF will represent the latest \( W \) elements in the current sliding window by the decaying property just like in DBF. Fig.1 gives an example of how a FDBF is used for keeping the latest \( W \) elements in the current sliding window. For the above example in Fig.1, \( e_i \) is hashed to two counters 10 and 40, both with flag bits zero, so \( e_i \) is reported as distinct according to step 1. The reason is that by the decaying property the latest element in the current sliding window is hashed to
counters with value $W$, the next latest element in the current sliding window is hashed to counters with value $W \land 1$, and so on. It is obvious that the larger the counters are the more recent the elements they represent and only the later incoming elements in the sliding window can change the counters that are set by the former elements. So in Fig.1, if $e_i$ truly has a duplicate in the sliding window, its duplicate must occur in or before the tenth element in the current sliding window and counter with value 20 is changed by the later incoming elements which is not equal to $e_i$ (if it is equal to $e_i$, it must also changes the counter with value 10). In other words, if the counter with value 20 is only set once in the incoming elements can change the counters used by the hashed to counters with value $W_i$ and counter with value 20 is changed by the later incoming elements which is not equal to $e_i$.

**Corollary 1:** Given a sliding window size $W$, a FDBF and a querying element $e_i$, $e_i$ is hashed to $k$ counters: $(C_1 \neq 0, \cdots, C_k \neq 0)$ with their according flag bits: $I_1, \cdots, I_k$, if the $k$ counters are not equal and a counter $e_i \in \{C_1, \cdots, C_k\}$ has $I_i = 0$, $e_i$ is taken to be distinct.

**Proof:** By step 2 and step 3, it is easy to see that the later incoming elements can change the counters used by the former elements. The latest element in the current sliding window is hashed to counters with value $W$, the next latest is hashed to counters with value $W \land 1$, and so on. If $e_i$‘s $k$ hashed counters, $(C_1 \neq 0, \cdots, C_k \neq 0)$, are not equal and $e_i$ has duplicate $e_o$ in the current sliding window, a counter $C_j \min \{C_1, \cdots, C_k\}$, $C_j \in \{C_1, \cdots, C_k\}$, is definitely hashed by the later incoming elements (which comes later than $e_i$ in the current sliding window). So $C_j$ is used by at least the latest $W$ elements and $I_j$ is set to 1. When the $k$ counters $(C_1 \neq 0, \cdots, C_k \neq 0)$ are not equal, the condition that $e_i$ has duplicate $e_o$ in the latest $W$ elements is sufficient for the conclusion that $C_j \min \{C_1, \cdots, C_k\}$: $C_j \in \{C_1, \cdots, C_k\}$ has $I_j = 1$. If the conclusion is false, the condition is false. So when the $e_i$‘s $k$ counters are not equal and a counter $C_j \min \{C_1, \cdots, C_k\}$: $C_j \in \{C_1, \cdots, C_k\}$ has $I_j = 0$, we take $e_i$ to be distinct.

**Corollary 2:** Given a sliding window size $W$ and a FDBF, there is no false negative error in the duplicate detection process based on FDBF.

**Proof:** This is straightforward by the above three steps in FDBF; It is easy to see that FDBF uses $n \ast \log W + n$ bits space, while DBF uses $n \ast \log W$ bits. In the next section, we will discuss the false positive rate of FDBF when applied to process to data stream in a block-wise manner.

III. DUPLICATE DETECTION BASED ON FDBF

In this section, we will give a duplicate detection method based on FDBF. The basic idea is based on processing a data stream in a block-wise manner which is similar to DDMDBF. Like in DDMDBF, we divide a data stream into blocks with size $W=(B \land 1)$ (each block has size: $W=(B \land 1)$), so for a sliding window with size $W$, there will be $B$ or $B/1$ blocks covered by the current sliding window. Elements belonged to the latest block in the current window are hashed to counters with value $B$, the elements belonged to the next latest block are hashed to counters with value $B \land 1$ and so on. We modify step 2 and step 3 in section 2 to a block-wise manner. As to step 2, when a block is full, we decrease all non-zero counters and clear those flag bits if its according counter is one after decreasing. As to step 3, we set $I[h_j(e_i)], J = 1 \cdots k$ to 1 if $C[h(e_i)] < B$ which means $C[h(e_i)]$ is used by elements from two blocks. (It should be noted that $I[\text{index}], \text{index} = 1 \cdots n$ now indicate whether its according counter $C[\text{index}], \text{index} = 1 \cdots n$ is used by elements from at least two blocks in the current window).

DDMDBF is given in Algorithm 1. (We use a variable Iteration to denote the current size of a data stream, if Iteration mod $W=(B \land 1) = 0$, we are sure that a block is full and do the update step.) We now study the properties of DDMDBF. For a sliding window size $W$ and block number $B$, by the update steps in DDMDBF, it is easy to see that there are $W < B < W+W=(B \land 1)$ number of elements represented by the counters and flag bits in FDBF. Given $W'$ the actual number of element represented by FDBF and $n$ a

Algorithm 1: Approximately Detect Duplicates over Sliding Windows Using FDBF

**Input:** sliding window size $W$, block number $B$ and a stream: $S = e_1, e_2, \cdots, e_S$  " 

**Output:** a sequence of yes/no corresponding to each element in $S$

for each $e_i \in S$ do

1. Probe $k$ counters $C[h_j(e_i)], J = 1 \cdots k$;

2. Find the minimum $C_{\ast} \in k$ counters $C[h_j(e_i)], J = 1 \cdots k$;

3. $\text{IsDuplicate} = \text{yes}$; // $e_i$ is assumed to be duplicated initially

4. if $C_{\ast} = 0$ then

   $\text{IsDuplicate} = \text{no}$

else// all $C[h_j(e_i)], J = 1 \cdots k$ are non-zero

   for each $C[h_j(e_i)], J = 1 \cdots k$ do

   if $C[h_j(e_i)] \geq 0$ and $I[h_j(e_i)] = 0$ then

   $\text{IsDuplicate} = \text{no}$;
2. Break ; // go to Step 5 
end 
end  
5. if Iteration mod $W=(B-1)=0$ then 
for each 
non-zero 
counter $C[index]$, index $\in 1 \ldots n$ 
do 
1. $C[index] = C[index] - 1$; 
2. if $C[index] = 1$ then $I[index] = 0$ 
end 
6. for each counter $C[h_j(e_i)]$, $j = 1 \ldots k$ do 
1. if $C[h_j(e_i)] < B$ then $I[h_j(e_i)] = 1$ 
2. $C[h_j(e_i)] = B$ 
end 
7. Iteration=Iteration+1; 
8. Output IsDuplicate 
end 

newly incoming distinct element $e_i$ the false positive error is incurred by the $W$ elements in the current sliding window and $W' - W$ elements which are outdated but still kept in the FDBF. 

There are two situations that $e_i$ is considered to be distinct in the duplicate detection problem over a sliding window with size $W$: firstly, $e_i$ has no duplicate in the latest $W'$ elements represented by the FDBF; secondly, $e_i$ has duplicates in the outdated $W' - W$ elements (they are still kept in FDBF, like elements in block 1 depicted in Fig.2) and don’t have duplicates in the latest $W$ elements. The distance between $e_i$ and its nearest duplicate in the past elements is large than $W$ in both cases. For the first situation, [1] call it Active error: if $e_i$ is taken to be duplicated; for the second, [1] call it UD error: It is straightforward to see that both UD error and Active error reach their maximum when $W' = W+W=(B-1)$ (a FDBF keeps at most $W+W=(B-1)$ elements with $W=(B-1)$ outdated). We depict this situation in Fig.2. So in the following, 

we only study the UD error: $P(UD\text{\_error})$ and Active\_error: $P(Active\_error)$ in the situation depicted in Fig.2. It is easy to see $P(FP) = P(UD\_error) + P(Active\_error)$.

Given a set of all possible outcomes: $AS = \{e_1, \ldots, e_{num}\}$ and a data stream $S = e_1, e_2, \ldots, e_j, \ldots (e_i \in AS)$, we assume that the stream $S$ is an outcome of a series of independent random variables $x_1, x_2, \ldots, x_i, \ldots$ which take value in $AS$ and is subjected to probability distribution $P$: $x_i$ has probability $P(x_i = e_i)$ to be $e_i, (e_i \in AS)$. In practice, $AS$ is usually a very large set which describes all possible elements that may occur in the data stream, for example in a click stream, $AS$ is the set of all possible click IDs (each ID is 32 bits long) that may occur in the stream. In this paper, we assume that the distribution of $x_{j}, j=1 \ldots N$ which determines the $i$-th element in the data stream are not identical while these underlying random variables are assumed to be independent and identically distributed in the SBF and DBF. For simplicity of illustration, we denote $x_{i}, i=1 \ldots x_{W'}$ as the underlying random variables which determine the elements in the $B$ continuously blocks in Fig.2 ($x_j, j=1 \ldots x_{W'}$ is subjected to distribution: $P_j$). It is easy to see that $x_{i}\ldots x_{W'}$ are outdated elements in block 1 but still kept in the FDBF when algorithm 1 probes $e$ in Fig.2, while $x_{W'+1}\ldots x_{W'}$ are elements in the current sliding window. For $e_i \in AS = \{e_1, \ldots, e_{num}\}$, we defines:

<table>
<thead>
<tr>
<th>$i$</th>
<th>$\text{TABLE I. THE PROBABILITY DISTRIBUTION OF C[i] AND I[i]}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P[C[i]=0, I[i]=0] = p_{00}, P[C[i]=0, I[i]=1] = 1-p_{00}$</td>
<td>$P[C[i]=1, I[i]=0] = (1-p_{00})^{(1-p_{00})+p_{00}} \cdot B, b = 3 \ldots B$</td>
</tr>
<tr>
<td>$P[C[i]=1, I[i]=1] = (1-p_{00})^{(1-p_{00})+p_{00}} \cdot B, b = 3 \ldots B$</td>
<td>$P[C[i]=0, I[i]=0] = (1-p_{00})^{(1-p_{00})+p_{00}} \cdot B, b = 3 \ldots B$</td>
</tr>
</tbody>
</table>

by elements in a block with $w$ distinct elements. With the decaying steps in algorithm 1, it is easy to obtain the above distribution table.

Corollary 4: For a sliding window size $W$, $B$ continuous blocks and $k = \ln(2)^m(Bow)$ hash functions, we have $P(Active\_error) \approx (1-2)^w$ when probing a distinct element $e_i$ in Fig.2

Proof: The $k$ counters hashed by $e_i$ are $C[h_1], \ldots, C[h_k]$.
$C[h_i] \geq h_1, \ldots, h_k \leq n, \ min$ denotes the position of the minimum value in $C[h_1], \ldots C[h_k]$. Given that $e_i$ has no duplicates in the latest $W'$ elements kept by FDBF in Fig.2, algorithm 1 take it to be duplicated when the following events happen:

$\{h_k' \in \{h_1, h_2, \ldots h_k\} \}$

Events: $C[min] = b$ and $C[h_k'] > C[min]$ has $C[h_k'] = 1, b=1,B-1$

Event #2: $C[min] = B$

$P(Events_2) = \sum_{i=0}^{B-1} \sum_{k=1}^{B-1} \sum_{i=1}^{B-1} C[i] \cdot P(C[i] = b) \cdot P(I[i]=1) = B \cdot \sum_{i=1}^{B} P(C[i]=b, I[i]=1)$

$C[i] \cdot P(C[i]=b, I[i]=1)$ is combination number

Figure 2. The situation when $W'$ the actual number of element represented by FDBF is maximized to $W+W=(B-1)$

351
we simplify \( P(\text{Active_error}) \) in the following:
\[
P(\text{Active_error}) = \sum_{i=1}^{k} P(\text{Event}_i) \quad \text{where} \quad \text{Event}_1, \ldots, \text{Event}_k \text{ are disjoint}
\]
\[
P(\text{Active_error}) \approx (1/4)^k
\]
\( P_0 \approx (1/2)^{1/2} \) (i.e., \( \ln(2) \approx 1/B \) which minimize \( (1 - P_0^B)^k \))
\[
p_0 = (1 - 1/2)^{B/k} = \sum_{i} C_{i} r / (-1/2)^k
\]
Thus we have: \( P(\text{Active_error}) \approx (1/2)^k \).

**Corollary 5.** For a sliding window size \( W \), \( B \) continuous blocks with \( k = \ln(2)n/\ln(W) \) hash functions, \( P(UD\text{ _error}) \leq 1/B \) in coloary 3 and \( P(Active\text _error) \approx \)

DDMDBF [1] with \( 2G \) bits space and \( W \) window size have
\[
P(FP) \approx 2(1/2) \sqrt{2\ln(2)G/W} \quad \text{which is same as}
\]
DDMFDBF with \( G + G/\log(B) \).

**IV. EXPERIMENT**

Our platform is Pentium IV processor with 1G memory and the OS is Windows XP. DDMFDBF is implemented in Matlab 7.0. We evaluate the proposed method on synthetic sets of 10: unique integer values, so each reports of duplicate is a false positive error. The two methods DDMDBF and DDMFDBF were compared over the 2-width sliding windows. These two methods use the same hash functions of modulo/multiply type form [9]. For a element \( \epsilon \) in the stream, its hash value \( H(\epsilon) = [m(a*\epsilon \mod 1)] \), where \( a \) is taken uniformly at random from [0, 1]. From Fig.3 we can see the FPR of DDMFDBF is much lower than that of DDMDBF. Fig.4 proves that the FPR of DDMFDBF using \( G + G/\log(B) \approx G \) bits is lower than the FPR of DDMDBF using \( 2G \) bits. It also shows that the FPR of DDMDBF approaches the FPR of DDMFDBF as the ratio between the size of space and the size of sliding window increases.

**V. CONCLUSION**

In this paper, we propose Flag Decaying Bloom Filter (FDBF), which is an extension of Decaying Bloom Filter (DBF), and a simple and efficient duplicate detection method (DDMFDBF) based on DBF. For each integer counter in DBF, FDBF introduces a flag bit which stands for whether the counter has been hashed by elements from \( ln \), \( 2 \) blocks or not. Thus FDBF keeps the information of elements in the current sliding window more accurate than DBF and significantly eliminates false positive errors when the \( k \) counters hashed by a distinct querying element are non-zero. Given bit space \( G + G/\log(B) \approx G \), block number \( B \) (\( B \) is usually very large) and sliding window size \( W \), DDMFDBF reduces false positive rate significantly to
\[
(1/2) \sqrt{2\ln(2)G/W} - 1 \quad \text{which is equal to DDMDBF using 2G bit space. Both analytical and experimental results support the efficiency and accuracy of DDMFDBF.}
\]

**REFERENCES**


