DELAY-CFIM: A Sliding Window Based Method on Mining Closed Frequent Itemsets over High-Speed Data Streams

Chunkai Zhang, Yulong Hu, Lei Zhang
School of Shenzhen Graduate, Harbin Institute of Technology, Shenzhen, China

Abstract—Closed frequent itemset mining plays an essential role in data stream mining. It could be used in business decisions, basket analysis, etc. Most methods for mining closed frequent itemsets store the streamlined information in compact data structure when data is generated. Whenever a query is submitted, it outputs all closed frequent itemsets. However, the online processing of existing approaches is so slow that those methods cannot deal with data streams generated at a high speed. In this paper, a novel method DELAY-CFIM for mining closed frequent itemsets is proposed to solve the problem of slow online processing. It divides the closed frequent itemset mining process over data streams into two steps. Firstly, when transactions are generated, it stores the frequency information of itemsets in a summary data structure. Then it mines closed frequent itemsets until a query is submitted. The method can improve the speed of online processing.

Keywords: closed frequent itemset, data stream, sliding window.

I. INTRODUCTION

Recently, data stream mining has been a hot topic in data mining. With the development of information technology, large amount of data streams are generated every day[1]. Different from the traditional static dataset, a data stream is a massive open-ended sequence of data elements continuously generated at a rapid rate. In order to play its role, the data streams need to be converted into useful information so that they could be applied in different applications. Frequent itemset mining is one of the most important types in data stream mining. It could be applied in many different domains, including network monitoring, market basket analysis, catalog design and cross-marketing, and customer shopping behavior analysis, etc[2].

The approaches for mining closed frequent itemsets over data streams are mostly based on the methods for mining traditional static dataset. To solve the problems caused by fast data streams and massive data, most of the algorithms maintain a summary data structure in memory. Due to time and memory constraints, it is impossible to monitor all the information of data streams in the summary data structure. Hence, window mechanisms are involved to deal with the data streams. According to the stream processing model[3], algorithms of frequent itemset mining over data streams could be divided into three categories: sliding window, landmark window and damped window. The algorithms based on sliding window try to mine the most recent frequent itemsets over data streams. A users’ specified threshold $\text{windowSize}$ is involved to limit the number of transactions in the sliding window. Algorithms based on landmark window not only concern about the information in current window, but also consider the history data. Because of the large amounts of data, it is impossible to store all the history data in the summary data structure. Hence these algorithms usually provide each itemset with an estimated frequency and ignore the itemsets whose estimated frequency is lower than the specified threshold. The algorithms based on damped window do not ignore the history data totally. Data is stored from the landmark time point. However, damped window algorithms give a weight to the obsolete data for decreasing the importance of them[4]. Therefore, damped window mechanism combines ideas of sliding window and landmark window and considers the contribution of recent windows is more than that of older ones.

Most methods for mining closed frequent itemsets are based on sliding window. In [5], Chi et al proposed the first one-pass algorithm-MOMENT Algorithm for mining closed frequent itemsets over data streams. MOMENT Algorithm maintains all closed frequent itemsets and several boundary itemsets in main memory based on a user specified threshold. But it wastes a mass of memory to maintain the boundary nodes and only outputs closed frequent itemsets whose supports are higher than the user specified threshold. In [6], Jiang et al presented an improved MOMENT Algorithm, CFI-Stream, which maintains all closed itemsets in a summary data structure and output closed itemsets with arbitrary value of support. However, due to the nature of the algorithm, it performs well when minimum support is low but much worse when minimum support turns higher. In [7], Ren et al proposed HCFI Algorithm. Different from MOMENT and CFI-Stream, it uses a vertical representation of itemsets and involves hash table to reduce the time overhead of closure detection. But it is only suitable for the data streams with a few items and applications with small window size. In [8], Yen et al shown CloStream Algorithm with list data structure to maintain closed itemsets. It performs well when the total number of items is not large.

However, these algorithms make closure detection for each frequent subset of a transaction, which leads to an exponential complexity of online processing. This paper focuses on the problem of mining closed frequent itemsets over data streams and proposes a method with a linear complexity of online processing. In the paper, a summary data structure ($\text{OTT}$) is designed to store the compact information of data streams. And a novel algorithm DELAY-CFIM is proposed to discover the closed frequent itemsets from $\text{OTT}$. DELAY-CFIM scans $\text{OTT}$...
once and generates the frequent itemsets by reinserting the suffix itemsets into OTT. Then it checks the closure feature for each frequent itemset on the closed frequent itemset tree (CFIT). Several efficient pruning strategies are proposed to reduce the time and space overhead of DELAY_CFIM. The algorithm is proposed based on sliding window for capturing changes of data streams in time. Different from the previous approaches for closed frequent itemset mining, DELAY_CFIM delays the mining procedure until a query is submitted. Hence, the online processing of DELAY_CFIM is much faster than the previous methods.

II. PRELIMINARY

Closed frequent itemsets record complete and condensed information of frequent itemsets. And sliding window records the most recent complete information of data streams. Hence mining closed frequent itemsets over sliding window adapts rapidly to the change in data streams.

A. Closed Frequent Itemsets

Define a threshold \( s \) called minimum support (\( \text{min}_{\text{sup}} \)), \( 0 < s \leq 1 \). Frequent itemset (FI) is an itemset whose \( \text{support} \) is not less than \( s \).

An itemset \( A \) is called closed itemset only if there does not exist any superset \( B \) of \( A \) with the same support of \( A \).

According to the definitions above, if an itemset is both frequent and closed in \( D \), Which is defined as a database of transactions, it is closed frequent itemset which is abbreviated as CFI in this paper.

B. Sliding Window

In Fig.1, a sliding window model is shown with \( \text{windowSize} = 4 \). Firstly, four transactions \( T_1, T_2, T_3 \), and \( T_4 \) are included in current window \( w_1 \). As transaction \( T_5 \) arrives, \( T_1 \) leaves the sliding window, window \( w_2 \) becomes the current window with transactions \( T_2, T_3, T_4 \), and \( T_5 \). According to the sliding window mechanism, when a query is submitted, only the transactions in current window need to be mined. Such as it is shown in the Fig.1, if there is a query submitted after \( T_4 \)'s arrival, only the transactions in \( w_1 \) are mined, the mining result of CFIS is: \( \{b, c\}, \{b\} \). However, if the query is submitted after \( T_5 \)'s coming, the current window changes to be \( w_2 \), the mining result is: \( \{b\}, \{c\}, \{d\} \).

III. ORDERLY TRANSACTION TREE AND CLOSED FREQUENT ITEMSET TREE

A. The Summary Data Structure-OTT

Definition1. An Orderly Transaction Tree (OTT) is a transaction-ordered and tree based data structure defined as follows:

1) OTT is composed of the transactions in the current sliding window. It maintains almost all the information in the current sliding window except the generated order of transactions. Each node \( N \) on OTT represents an itemset \( I \) including the items on the path from root to the node, and each child node of \( N \) represents an itemset which is obtained by adding a new item to \( I \).

2) Each node on OTT consists of four data fields: \( \text{item\_id}, \text{item\_count}, \text{temp\_count} \) and \( \text{fromTree\_id} \).
   a) \( \text{item\_id} \) identifies a unique id in \( I \).
   b) \( \text{item\_count} \) registers the number of the transactions, which have their items sorted in ascending order, with the same prefix item sequence as the itemset represented by current node.
   c) \( \text{temp\_count} \) records the number of the temporary inserted itemsets in mining process. During construction and maintenance process of OTT, the value of \( \text{temp\_count} \) assigned to a new node is 0.
   d) \( \text{fromTree\_id} \) records where the reinserted sub-tree comes from during mining process.

Fields \( \text{item\_id} \) and \( \text{item\_count} \) are maintained in the whole process including construction and maintenance of OTT and the mining process. Fields \( \text{temp\_count} \) and \( \text{fromTree\_id} \) only take effect in the mining operation.

3) Child nodes for each node on OTT are linked up in ascending order according to the field \( \text{item\_id} \). For example, if a node \( N \) includes four child nodes whose \( \text{item\_id} \) are separately \( d, b, c \) and \( a \), according to the definition, these nodes must be linked up in the order \( a, b, c, d \). The ordered structure plays a key role in mining process.

B. Construction and maintenance of OTT

1) Construction of OTT

OTT is constructed from an empty tree. The construction scenario of OTT is described as follows:

a) When a transaction \( T \) arrives, sort the items of \( T \) in an ascending order. The ordered transaction is called \( OT \) in this thesis.

b) Insert \( OT \) into OTT. If the path covering \( OT \) exists, update the field \( \text{item\_count} \) of each node on the path. Otherwise, construct a new path for \( OT \) and keep the ordered structure for OTT at the meantime.

2) Maintenance of OTT

As the transaction are generated, the total number of transactions will exceeds \( \text{windowSize} \). Hence the outdated
transactions must be removed from OTT. The maintaining process is described as follows:

Whenever a new transaction is generated, insert it into OTT as described in OTT construction process. Then delete the outdated transaction from OTT which have left the sliding window. During the process, if there are nodes whose item_count fields have been reduced to zero, delete the nodes.

Fig. 2 shows an example for maintenance of OTT. Fig. 2 a) outlines a transaction sequence in data stream and defines windowSize as 4. Fig. 2 b) ~ d) presents the changes of OTT with the sliding window moving on. In Fig. 2 c), transaction Tc is generated and transaction Td becomes outdated. Hence, path \{a,c,d\} is constructed for Tc, and nodes b and c are deleted from path \{a,b,c\} as their item_count have decreased to zero.

![Fig.2 Maintain of OTT](image)

### C. Closed Frequent Itemset Tree-CFIT

**Definition 2.** A Closed Frequent Itemset Tree (CFIT) is an ordered structure for maintaining closed frequent itemsets, it is defined as follows:

1) A Closed Frequent Itemset Tree (CFIT) is composed of an Itemset Tree and an Item List. Itemset Tree maintains the closed itemsets generated so far. And Item List records the items appearing in Itemset Tree. Several Special Link-List structures (SLL) which begin with an Item List node followed by several Itemset Tree nodes are constructed among Itemset Tree and Item List nodes.

2) Each node on Itemset Tree consists of four fields: item_id, listItem_count, node_height and treeNextNode_pointer.

   a) item_id identifies a unique id in I.
   b) listItem_count records frequency of the itemset represented by the node. Each node, whose value of listItem_count is unequal to its child nodes, represents a closed frequent itemset including the items from tree root to current node.
   c) node_height registers height of the node on OTT.
   d) treeNextNode_pointer links up the nodes with same item_id on Itemset Tree. The nodes which are linked up by treeNextNode_pointer compose the tail part of Special Link-List structure (SLL). All the nodes linked up together in the same SLL have an equal value of item_id and are sorted in descending order by the field treeItem_count.

3) Each node in Item List consists of three fields: item_id, listItem_count and listItem_pointer.

   a) item_id identifies a unique id in I.
   b) listItem_count records the largest value of treeItem_count of OTT nodes which are with the same item_id as current Item List node.
   c) listItem_pointer links up the nodes with the same item_id on OTT and makes the node be head of a SLL.

4) The child nodes of each node on Itemset Tree are sorted in ascending order according to item_id.

5) The nodes in Item List are linked up in ascending order by item_id.

### D. Construction of CFIT

According to the definition of CFIT, the construction process of CFIT is described as follows:

1) When a new closed itemset CI is generated, firstly update the Item List.

   a) If all items of CI are in the Item List and frequency of CI is larger than listItem_count of the corresponding nodes, update listItem_count with the frequency of CI.
   b) Otherwise, if any item in CI does not exist in the Item List, create a new node for it and keep the correct order of the nodes in Item List in the meantime.

2) Then update the Itemset Tree.

   a) If the path on Itemset Tree covering CI exists and the treeItem_count value of the corresponding node is smaller than the frequency of CI, update the field treeItem_count with the frequency of CI.
   b) Otherwise, if the path does not exist, construct the path and assigned CI frequency to the field treeItem_count of each new node. In the insertion operation, the special structure of Itemset Tree and Special Link-List must be maintained.

### IV. ALGORITHM DELAY-CFIM

DELAY-Closed Frequent Itemset Mining (DELAY-CFIM), is introduced in this section. It is composed of two steps: frequent itemset generation and closure detection. Whenever a query is submitted, the algorithm generates all the frequent itemsets in the current window from OTT. Then closure detection is done on CFIT for each frequent itemset generated. The correctness of DELAY-CFIM is proved in this section and several effective pruning strategies are introduced at last.

**A. Frequent Itemset Generation**

Based on the definition of OTT, all condensed frequency information of itemsets is maintained on it. Since the information has been aggregated and compressed, if it is made
full use of, much time and memory space would be reduced. Algorithm 1 presents the pseudo code for generating the frequent itemsets from OTT.

**Algorithm 1.** Frequent itemset generation  
**Input:** Root of OTT in current window (T)  
- Current record path (p) // initialized by ∅  
- Minimum support s  
**Output:** A set of frequent itemsets  

1. if (T ➔ children = NULL ) then  
2. for each child node Tc of T  
3. Reinsert( Tc , T )  
4. if (Tc ➔ item_count + Tc ➔ temp_count ≥ s × windowSize ) then  
5. p' = p ∪ Tc  
6. FIGeneration ( Tc , p' , s )  
7. restore Tc  
8. end if  
9. end for  
10. end if  
11. if ( p ≠ ∅ )  
12. output p  
13. end if  

**Algorithm 2.** Reinsert  
**Input:** Root of source tree Ts  
Root of destination tree Td  
**Output:** The result tree  

1. if (Ts ➔ children = NULL ) then  
2. for each child node Ts of Ts  
3. if (Ts ➔ item_count = item_count of one child Ts of Td ) then  
4. Temp_count + = Ts ➔ item_count + Ts ➔ temp_count  
5. else  
6. create Ts ➔ item_count=0 and  
7. Temp_count + = Ts ➔ item_count + Ts ➔ temp_count  
8. end if  
9. Reinsert( Ts , Ts )  
10. end for  
11. end if

Algorithm 1 is a depth-first procedure visiting OTT in post order. Before visiting a subtree TC on OTT, it first reinserts the subtrees of TC into TC’s parent node T (line 3). Then it compares TC’s support with min_sup. If TC’s support is larger than s, it adds TC to the current path and recursively visits TC. At last it restores whole structure of TC after visiting it (lines 4-7).

Algorithm 2 shows the pseudo code of reinsertion operation. If the destiny child node TDC to be inserted exists, it would update the temp_count field of TDC (lines 3-4). Otherwise, a new tree node would be created with item_count set to be zero (lines 5-7).

B. Correctness Proof for Frequent Itemset Generation

**Lemma 1.** Algorithm 1 generates all frequent itemsets for a given OTT.

**Proof:** We prove Lemma 1 in two steps:

1. Firstly we prove that Algorithm 1 generates all items for OTT if without support condition (Line 4 in Algorithm 1).
2. Then we prove the infrequent itemsets have been pruned in the check of support condition.

For a given set of items I = {i1 , i2, ..., in} , we sort the items in each subsets of I in ascending order. Hence the subsets of I are divided into n parts: subsets beginning with i1, subsets beginning with i2, ..., subsets beginning with in. The generated sequence of frequent itemsets in Algorithm 1 is just following it, which means frequent itemsets beginning with i1 are generated at first and frequent itemsets beginning with in are generated at last. It is because OTT is an orderly tree, and algorithm 1 travels it in post order. According to the definition of OTT, the subtrees on OTT with root height equal to 2 are rooted at i1,i2,...,in. Now, we will prove before mining frequent itemsets on a subtree of OTT which is rooted at in, all of the itemsets beginning with in in current sliding window have been inserted into the subtree. An inductive method is used in the proof.

Firstly, considering of the case of m = 1, we check the feature of the subtree rooted at i1. Since the items in each transaction have been sorted before inserting it into OTT, and all the itemsets including i1 are beginning with i1. Hence all the itemsets beginning with i1 have been inserted into the subtree rooted at i1 during OTT construction.

Then suppose the feature is matched when m < k . Consider the case: m = k. We divide the itemsets beginning with ik into two parts and check them separately.

1. Items included by ordered transactions beginning with ik. It is easy to prove that these itemsets have been inserted into the subtree rooted at ik during OTT construction.

2. Items included by ordered transactions not beginning with ik. Since items in each transaction have been sorted in ascending order, these transactions must begin with one of items i1 , i2, ..., ik-1. And according to the assumption, before mining subtree rooted at ik, the algorithm has traveled and reinserted those subtrees rooted at i1 , i2, ..., ik-1. Hence, all subsets beginning with ik in these transactions have been reinserted into subtree rooted at ik.
At last, according to the inductive method, all itemsets beginning with $i_n$ have been inserted into the subtree rooted at $i_n$ before it is visited.

Since Algorithm 1 is a recursive method, the feature introduced above is satisfied by any depth of the recursive process. Hence, when a path is outputted, all the frequency information of the itemsets in the path has been aggregated on it.

So far, proof of the first step has been finished. Then we prove that each itemset which Algorithm 1 outputs is frequent. In Algorithm 1, the condition shown in the Equation 2 is used to prune the subtrees whose supports are lower than $s$. According to the definition of $OTT$, the itemset represented by $T_c$ is a subset of that represented by $T_c$’s child nodes. Hence the supports of nodes on the subtree rooted at $T_c$ must be not higher than support of $T_c$, which means all the itemsets pruned in Algorithm 1 are infrequent. And it is obvious that itemsets output by Algorithm 1 are frequent.

$T_c \rightarrow \text{item\_count} + T_c \rightarrow \text{temp\_count} \geq s \times \text{windowSize}$ (2)

In conclusion, Algorithm 1 generates all the frequent itemsets for a given $OTT$.

C. Closure Detection

According to the definition of $CFIT$, closed frequent itemsets could be maintained on it. Whenever a new frequent itemset $FI$ is generated, one scan on $CFIT$ is processed to check whether there exists a superset of $FI$ with support equal to $FI$’s support. If it is, $FI$ is not closed. Otherwise, it would be inserted into $CFIT$. Algorithm 3 presents the pseudo code for closure detection on $CFIT$.

Algorithm 3. Closure Detection

Input: A frequent itemset ($FI$)
1. The frequency of $FI$ ($CFI$)
2. The Itemset Tree in $CFIT$ ($IT$)
3. The Itemset List in $CFIT$ ($IL$)
Output: A Boolean value (closed)
1. $s$ = true
2. for each item $i$ in $FI$ do
3. if ($i \notin IL$ || $\exists \exists$ item $j$ with the same item_id as $i$)
4. then $FI$ into $CFIT$
5. return TRUE
6. else
7. $FI$ into Item List
8. Record lastN as the IL node with the same item_id as the last item of $FI$
9. break
10. for each $IT$ node ($IT$Node) in the SSL beginning with lastN do
11. if ($FI = IT$Node $\rightarrow$ trueItem_count) then
12. if the path from root node of $IT$ to $IT$Node covers $nFI$ then
13. return FALSE
14. $FI$ into $CFIT$
15. return TRUE

Algorithm 3 does closure detection in two steps: Step1. Check the items of $FI$ in the Item List (lines 1-8), if any item of $FI$ is not included in Item List, $FI$ is closed, Algorithm 3 inserts it into $CFIT$ and terminates itself.

Step2. Then check the path on Itemset Tree which might cover $FI$ (lines 9-14). If all of the possible paths do not cover it, $FI$ is closed, Algorithm 3 inserts it into $CFIT$ and terminates itself.

D. Correctness Proof for Closure Detection

According to the description above, Algorithm 3 only checks supersets for a given frequent itemset on $CFIT$. That means Algorithm 3 is based on the hypothesis that the frequent itemsets generated after the current itemset ($FI$) cannot be the superset of $FI$. In this section, proof of the hypothesis is presented.

Lemma 2. The frequent itemsets generated after $FI$ in Algorithm 1 cannot be the superset of $FI$.

Proof. We divide the supersets of $FI$ into two parts. Suppose:

$FI = \{i_1, i_2, ..., i_k\}$

Supersets in the first part begin with prefix-set $FI$:

$FI_{sup1} = \{i_1, i_2, ..., i_k, i_{l1}, i_{l2}, ..., i_{lw}\}$

$i_{l1} < i_1 < i_2 < ... < i_k < i_{l1} < i_{l2} < ... < i_{lw}$ (4)

The other supersets compose the second part. Take one of them for example:

$FI_{sup2} = \{i_1, i_2, ..., i_f(j,j-1), i_{l1}, i_{l2}, ..., i_{lw}\}$

$i_f(j,j-1) < i_{l1} < i_{l2} < ... < i_{lw}$ (5)

As $OTT$ has a special structure that the child nodes of each node have been sorted in ascending order, we can conclude:

1) The first part of the supersets must be generated when Algorithm 1 travels subtree with the pre-path $FI$. Since Algorithm 1 travels $OTT$ in post order, all of the supersets in the first part are generated before $FI$.

2) According to the special structure of $OTT$, the second part of the supersets must be generated before visiting subtree with the pre-path $FI$. We take $FI_{sup2}$ for example, the subtree with the pre-path $i_1, i_2, ..., i_f(j,j-1), i_{l1}$ must be visited before the subtree with the pre-path $i_f(j,j-1), i_{l1}, i_{l2}, ..., i_{lw}$. Hence, $FI_{sup2}$ must be generated before $FI$.

Hence, all the supersets of $FI$ are generated before itself.

In Step2 of Algorithm 3, the SSL begins with the Item List node lastN has been scanned. And Algorithm 3 only checks the closed itemsets matching the conditions as follows:

1) Including the last item of $FI$.
2) With a frequency equal to $FI$.

According to the definition of the closed itemsets, it is easy to prove that the itemsets not including the last item of $FI$ cannot be the superset of $FI$. Lemma 3 proves the correctness of the second condition.

Lemma 3. The frequency of $FI$'s supersets on $CFIT$ must equal to $FI$'s frequency.

Proof. Suppose there is a closed itemset $\overline{FI}$ which is the superset of $FI$ and has a frequency $f_\overline{FI}$ which is larger than
frequency of \( FI \) (\( f_{FI} \)). Since \( \overline{FI} \) is the superset of \( FI \), the transactions in \( D \) including \( \overline{FI} \) must include \( FI \). According to the definition of frequency, there are \( f_{\overline{FI}} \) transactions in \( D \) including \( \overline{FI} \) and they must include \( FI \) at the meantime. Hence frequency of \( FI \) is not less than \( f_{\overline{FI}} \) which is in contradiction with the assumption. Hence only the closed itemsets with frequency not larger than that of \( FI \) could be the superset of \( FI \).

According to the definition of closed itemsets, closed itemset with a lower frequency than \( FI \) does not need to be checked. Hence, Algorithm 3 checks the closed itemsets with the same frequency as \( FI \) only.

In conclusion, Algorithm 3 checks the closure feature correctly for each frequent itemset generated in Algorithm 1.

E. Pruning

All the frequent itemsets in the current sliding window are generated in Algorithm 1. But some of these itemsets can be determined to be not closed without detection.

The rules for maintaining \( \text{fromTree} \_id \) are introduced as below:

1) In the process of \( OTT \) construction and maintenance, each node on \( OTT \) is created with \( \text{fromTree} \_id = -1 \).

2) In the reinsertion operation, four cases are considered:
   a) The field \( \text{fromTree} \_id \) of destiny subtree equals to \(-1\). It remains \(-1\) after reinsertion.
   b) If the source subtree and the destiny subtree are with the same value of \( \text{fromTree} \_id \), it would not be changed after reinsertion.
   c) If the source subtree and the destiny subtree are with the different values of \( \text{fromTree} \_id \), after reinsertion, \( \text{fromTree} \_id \) of destiny subtree would be assigned to \(-1\).
   d) If the destiny subtree does not exist, a new subtree should be created with the field \( \text{fromTree} \_id \) being the parent node of the source node.

According to the analysis above, a pruning rule is proposed: In the process of Frequent Itemset Generation, only the subtrees with \( \text{fromTree} \_id \) value equal to \(-1\) are mined.

V. EXPERIMENTAL RESULTS

In this Section, DELAY-CFIM is compared with CFI-Stream, a classic algorithm on closed frequent itemset mining over data streams.

A. Datasets Used in Experiments

1) IBM quest market-basket synthetic data

The synthetic data sets used in this thesis are generated by IBM synthetic data generator[5]. It simulates the transactions in the retailing environment. Each item in the dataset represents a commodity in the retail stores maybe a super market. Each transaction in the dataset represents a sale record of customers. Then a frequent itemsets represents the commodities which customers usually buy together.

The parameters of the data set are described as below:

a) \( \overline{T} \): Average transaction size.

b) \( I \) : Average size of maximal potentially frequent itemsets.

c) \( N \) : Number of items.

d) \( D \) : Number of transactions.

According to the definitions above, a dataset named T10.I5.N10k.D100k includes 100k transactions, the average number of items in each transaction is 10, the average size of maximal potentially frequent itemsets is 5, and the total number of items is 10k.

2) Real dataset BMS-WebView-2

BMS-WebView-2 is a real dataset containing several months’ click stream data from an e-commerce web site. Each transaction in the dataset is a set of product detail pages which are clicked in a web session. This dataset has been used in KDDCUP 2000[9]. There are totally 77,512 transactions and 3,340 distinct items in BMS-WebView-2. Average transaction size of it is 5.

B. Experiments on Sliding Window

1) Experiments on different \( \text{windowSize} \)

Fig. 3 a) shows the average online running time for each transaction in CFI-Stream and DELAY-CFIM for the dataset T5.I4.N1k.D100k. In the experiments, threshold \( \text{min} \_\text{sup} \) has been set to 0.1%. In the figure, DELAY-CFIM consumes much less online processing time than CFI-Stream, because CFI-Stream needs to do closure detection for all subsets of each transaction, in this process it must scan the summary data structure for many times. However, DELAY-CFIM only inserts the transaction to the summary data structure \( OTT \), which only scans \( OTT \) once. In the figure, the online running time of CFI-Stream is more sensitive for \( \text{windowSize} \) than that of DELAY-CFIM, as it needs to scan the current window during CFI maintaining, larger \( \text{windowSize} \) leads to a longer scanning time.

Fig. 3 b) shows the average processing time of a query for the dataset T5.I4.N1k.D100k. Threshold \( \text{min} \_\text{sup} \) is set to 0.1%. In the figure, as the \( \text{windowSize} \) increases, online processing time of both algorithms increase. However, DELAY-CFIM consumes much more offline processing time than CFI-Stream. That is the price of the less online processing time.

Fig. 4 shows the number of nodes generated during the algorithms running. In the figure, more \( OTT \) nodes are created.
than CFI nodes, however, considering the node structure of them, each CFI node stores a whole itemset but each OTT node only records an item. At fact, they almost consume same amount of memory space.

According to the analysis above, both DELAY-CFIM and CFI-Stream takes more time and space overhead as window size increases. But the online processing time of CFI-Stream is sensitive than DELAY-CFIM. As DELAY-CFIM delays the mining process until a query is submitted, it consumes less online running time but more offline processing time.

2) Experiments on different query frequencies

Fig. 5 shows the average running time over 10k sliding windows with different query frequencies for real dataset BMS-WebView-2. Thresholds \( \min_{sup} \) and \( \text{windowSize} \) are separately set to be 0.1% and 50k.

In the figure, DELAY-CFIM performs much better than CFI-Stream when query frequency is low. And DELAY-CFIM is much more sensitive of query frequency than CFI-Stream. This is because CFI-Stream keeps all closed frequent itemsets in memory all the time, the offline processing time complexity of CFI-Stream is linear. Hence the offline processing time of DELAY-CFIM is larger than that of CFI-Stream. Therefore frequent queries lead to amount of total running time.

VI. CONCLUSION

In this paper a novel algorithm, DELAY-CFIM, is proposed to maintain the compact information in the current data stream sliding window and output the closed frequent itemsets whenever a query is submitted. The algorithm offers a method to reduce the online processing time and delay mining closed frequent itemsets until a query is submitted. Experimental results show that DELAY-CFIM outperforms the representation algorithm CFI-Stream in time overhead, especially when query frequency is low.

In addition, the method proposed in this thesis is based on sliding window, which limits its applied fields. If the summary data structure OTT is modified to maintain information in landmark window or damped window, the algorithm could be applied in more occasions. Although this paper use the delay strategy to reduce the online processing time overhead, it consumes more time when queries are proposed, especially when the minimum support is low. Hence, more researches are required to speed up the mining procedure.

REFERENCES