

A Fast Algorithm for Mining Frequent Closed Itemsets over Stream Sliding Window

Show-Jane Yen

Dept. of Computer
Science & Information
Engineering
Ming Chuan
University
Taipei, Taiwan
sjyen@mail.mcu.edu.tw

Cheng-Wei Wu

Dept. of Computer
Science & Information
Engineering
National Cheng Kung
University
Taipei, Taiwan
silvemoonfox@hotmail.com

Yue-Shi Lee

Dept. of Computer
Science & Information
Engineering
Ming Chuan
University
Taipei, Taiwan
leeys@mail.mcu.edu.tw

Vincent S. Tseng

Dept. of Computer
Science & Information
Engineering
National Cheng Kung
University
Taipei, Taiwan
tsengsm@mail.ncku.edu.tw

Chaur-Heh Hsieh

Dept. of Computer &
Communication
Engineering
Ming Chuan
University
Taipei, Taiwan
hsiehch@mail.mcu.edu.tw

Abstract—Mining frequent patterns refers to the discovery of the sets of items that frequently appear in a transaction database. Many approaches have been proposed for mining frequent itemsets from a large database, but a large number of frequent itemsets may be discovered. In order to present users fewer but more important patterns, researchers are interested in discovering frequent closed itemsets which is a well-known complete and condensed representation of frequent itemsets. In this paper, we propose an efficient algorithm for discovering frequent closed itemsets over a data stream. The previous approaches need to do a large number of searching operations and computations to maintain the closed itemsets when a transaction is added or deleted. Our approach only performs few intersection operations on the transaction and the closed itemsets related to the transaction without doing any searching operation on the previous closed itemsets. The experimental results show that our approach significantly outperforms the previous approaches.

Keywords—frequent closed itemsets; data stream; sliding window; data mining;

I. INTRODUCTION

Mining frequent itemsets [1, 4, 8] from a transaction database is a fundamental task to several data mining applications. The problem of discovering frequent itemsets in database is stated as follows: Given a finite set of items $I = \{a_1, a_2, \dots, a_m\}$, a transaction t is a set of items which were purchased by a customer at the same time. A transaction database D contains a set of transactions and each transaction is given a unique identifier (TID). An itemset $X = \{i_1, i_2, \dots, i_k\}$ ($i_j \in I, 1 \leq j \leq k$) is a set of k distinct items, where k is the length of X . The support count of X is denoted as $SC(X)$ and defined as the number of transactions which contain X . The support of X is defined as the ratio of $SC(X)$ to the total number of transactions in D . An itemset is said to be frequent if its support is no less than a user specified threshold min_sup which is called minimum support. The problem of mining frequent itemsets from a database D is to discover all the itemsets whose supports are no less than min_sup . For example, suppose the min_sup is set to be 40% for the transaction database in Table I, that is, the minimum support count is $40\% \times 5 = 2$.

TABLE I. TRANSACTION DATABASE

TID	Item Bought
t_1	C,D
t_2	A,B
t_3	A,B,C
t_4	A,B,C
t_5	A,C,D

Therefore, the frequent itemsets and their support counts are $\{A\}:4$, $\{B\}:3$, $\{C\}:4$, $\{D\}:2$, $\{AB\}:3$, $\{AC\}:3$, $\{BC\}:2$, $\{CD\}:2$ and $\{ABC\}:2$.

Several algorithms [1, 4, 9] have been proposed for mining frequent itemsets from transaction databases. But they mainly focused on static database and did not consider applications that involve data streams. A transaction data stream is an ordered sequence of transactions that comes in a timely order. Mining frequent itemsets over data streams is an important technique which is essential to a wide range of applications such as network traffic analysis, web click stream mining, on-line transaction analysis and many other important tasks. However, mining frequent patterns from data streams poses many challenges because data comes continuously, unbounded and usually with high speed. Furthermore, data distribution in a stream usually changes with time such that the status of itemsets may be changed (from frequent to infrequent or from infrequent to frequent). Hence, how to efficiently capture all the frequent itemsets over a data stream without information loss is a big challenge.

There are lots of works [2] in developing efficient algorithms for mining frequent itemsets from a data stream. But they often discover a large number of frequent itemsets and present too many redundant patterns to users. It is widely recognized that the larger the set of frequent itemsets is, the more the processing cost is required. In other words, the performance of the algorithms may be degraded when a database contains lots of frequent itemsets or a low threshold is used. In order to present users fewer but more important patterns, one of the solutions is to mine only the frequent closed itemsets [6, 8, 10]. Closed itemsets are the itemsets that have no proper supersets with the same support. Non-closed itemsets are regarded as redundant [6, 8, 10]. A closed itemset

is said to be frequent if its support is no less than min_sup . For example, the frequent closed itemsets in Table I are $\{A\}:4$, $\{C\}:4$, $\{AB\}:3$, $\{AC\}:3$, $\{CD\}:2$ and $\{ABC\}:2$. In this example, itemset $\{B\}$ is non-closed since its support is the same as $\{AB\}$. This implicitly indicates that $\{B\}$ will not appear in a transaction without $\{A\}$ and hence the whole information about $\{B\}$ is included in $\{AB\}$.

It is well known that frequent closed itemset provides users not only condensed but also complete information about frequent itemsets. In general, the number of frequent closed itemsets is much less than the number of frequent itemsets, that is, the set of frequent closed itemsets is a small subset of the set of frequent itemsets. Besides, a complete set of frequent closed itemsets can be used to uniquely derive all frequent itemsets and supports without losing any information [8].

Recently, some methods for mining frequent closed itemsets over a data stream were presented. Chi et al. proposed the *Moment* algorithm [3] to incrementally update the frequent closed itemsets over a data stream. They use a tree like structure, called *CET Tree* (*Closed Enumerate Tree*), to maintain a dynamically selected set of itemsets. Each node in the CET Tree represents an itemset with different node type. Whenever a transaction arrives or leaves, the nodes in the CET Tree are inserted, deleted or updated according to their node types. However, exploration of frequent closed itemsets and node type checking are time-consuming, especially when the minimum support is low. Furthermore, in order to maintain frequent closed itemsets, Moment needs to maintain a large number of nodes representing non-closed itemsets. A huge number of such non-closed nodes may require a lot of memory space when the transactions in a stream contain many frequent closed itemsets.

NewMoment [7] was recently proposed by Li et al.. It uses a tree structure, called *NewCET Tree*, to maintain the information of frequent closed itemsets. Each node in *NewCET Tree* consists of a frequent closed itemset and its support count. The search space and the memory consumption of NewMoment are much smaller than that of Moment, since there is no non-closed node in NewCET Tree. Besides, it adopts a bit-sequence technique to improve time efficiency for exploration of frequent closed itemsets. Both Moment and NewMoment relies on a depth-first, Apriori-like technique to construct their structures. When minimum support is low, the two algorithms may become inefficient, since they must enumerate a large number of candidate itemsets, which is very time-consuming.

Although Moment and NewMoment show good performance in [3, 7], they generally use a fixed support threshold to discover closed itemsets in data stream. In other words, the minimum support threshold needs to be pre-defined before mining and it cannot be changed during the mining process. However, in a data stream, data distribution of frequent closed itemsets usually changes with time such that users often need to alter support thresholds to obtain desired mining results.

To interactively present frequent closed itemsets to users online based on any minimum support threshold, Jing et al. proposed an algorithm *CFI-Stream* [5] to maintain the complete set of closed itemsets over data stream. The

experiments in [5] show that CFI-Stream achieves a better performance than Moment in terms of time and space, particularly at lower minimum support. CFI-Stream uses a lexicographical ordered tree, called *DIU Tree* (*Direct Update Tree*), to maintain all the closed itemsets and their supports such that any frequent closed itemset can be efficiently obtained based on any minimum support threshold. Comparing with Moment-based algorithm [3, 7], CFI-Stream is much more efficient and flexible when the minimum support is changed. However, the computational cost of CFI-Stream is very expensive since CFI-Stream uses a *candidate generate-and-test mechanism* to update closed itemsets. The mechanism is described as follows. When a transaction which contains k distinct items is added to (or deleted from) the database, CFI-Stream needs to *generate* $(2^k - 1)$ subsets (i.e., candidate closed itemsets) for the transaction. To *test* whether a candidate is closed or not, in the worst case, CFI-Stream needs to search all its supersets by traversing parts of DIU Tree. In other words, CFI-Stream requires to search from the DIU Tree $(2^k - 1)$ times for a transaction with k distinct items, which seriously degrades the performance of CFI-Stream.

II. PRELIMINARIES

In this section, we introduce some preliminaries for mining frequent closed itemsets over a data stream. A *data stream DS* = $\langle t_1, t_2, \dots, t_r \rangle$ is a continuous sequence of transactions in a timely order. A transaction-sensitive sliding window [2] is denoted as W , which captures the latest w transactions from a data stream. When a new transaction arrives to W , it would be added into W if the current number of transactions in W is less than w . Otherwise, if the current number of transactions in W is equal to w , then the oldest transaction would be expired from W and the new transaction would be added to W . The *support count* of an itemset X in W is denoted as $SC_W(X)$, and defined as the number of transactions which contain X in W . The *support* of X in W is defined as the ratio of $SC_W(X)$ to the total number of transactions in W . An itemset X in W is said to be *frequent* if $SC_W(X)$ is no less than $min_sup \times |W|$. The function f takes a set of transactions T in W as an input and returns an itemset which is the result of the intersection of all the transactions in T . The function g takes an itemset Y as an input and returns a set of transactions in W , which contain Y . The function $C_W(X)$ takes an itemset X as an input and returns an itemset which is the intersection of all the transactions in W belong to $g(X)$, that is, $f(g(X))$. For any itemset X , $C_W(X)$ is called the *closure* of X in W . The support of X in W is the same as the support of $C_W(X)$ since X and $C_W(X)$ in W appear in the same transactions.

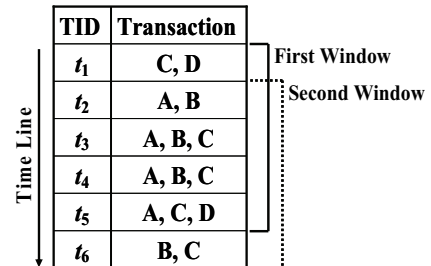


Figure 1. Transaction-sensitive sliding window

Definition 1. An itemset X is called a *closed itemset* in W if and only if $C_W(X) = X$.

$C_W(X) = X$ means that there is no proper superset of X with the same support as X . Therefore, X is a closed itemset. If $C_W(X) \neq X$, then the support of X is same as the support of its superset $C_W(X)$, that is, X is not a closed itemset. For example, Figure 1 shows a transaction-sensitive sliding window when the window size is set to 5. The first window W_1 consists of the transactions from t_1 to t_5 . When a new transaction t_6 arrives to the window, the oldest transaction t_1 is expired from the window. The second window W_2 consists of the transactions from t_2 to t_6 . In the first window W_1 , $g(\{B\}) = \{t_2, t_3, t_4\}$ is a set of all the transactions which contain $\{B\}$. $C_{W_1}(\{B\}) = f(g(\{B\})) = f(\{t_2, t_3, t_4\}) = t_2 \cap t_3 \cap t_4 = \{AB\} \cap \{ABC\} \cap \{ABC\} = \{AB\}$ is the closure of $\{B\}$. In the first window, $\{B\}$ is not a closed itemset, since $C_{W_1}(\{B\}) = \{AB\} \neq \{B\}$.

Definition 2. An itemset X is a *frequent closed itemset* in W if and only if $X = C_W(X)$ and $SC_W(X)$ is no less than $\min_sup$.

Problem Statement. Given a transaction-sensitive sliding window size w , the problem is to efficiently find the complete set of frequent closed itemsets from the most recent w transactions over a data stream based on any user-specified $\min_sup$.

In the following, we list some important properties about closed itemsets, which are intuitive and widely used in the previous literature [6, 8, 10].

Property 1. If $C_W(X) = Y$, then $SC_W(X) = SC_W(Y)$.

Rationale: If $C_W(X) = Y$, then $g(X) = g(Y)$. Therefore, $SC_W(X) = SC_W(Y)$.

Property 2. If $C_W(X) = Y$, then $C_W(Y) = Y$.

Rationale: Since $C_W(X) = Y$, $g(X) = g(Y)$. $C_W(Y) = f(g(Y)) = f(g(X)) = C_W(X) = Y$.

Property 3. X is closed if and only if $SC_W(X) > SC_W(Z)$, for all itemset Z , $Z \supset X$.

Rationale: If X is closed in W , then $C_W(X) = X$, that is, $C_W(X) \neq Z$ for all $Z \supset X$. Therefore, some transactions which contain X do not contain Z , that is, $SC_W(X) > SC_W(Z)$.

Property 4. If $X \subseteq Y$, then $g(Y) \subseteq g(X)$ and $C_W(X) \subseteq C_W(Y)$.

Property 5. $f(g(X)) \cap f(g(Y)) = f(g(X) \cup g(Y))$.

Theorem 1. For any two closed itemset X and Y , if $(X \cap Y) = S$ and $S \neq \emptyset$, then S is a closed itemset.

Rationale: To prove S is closed, we need to show that $C_W(S) = S$, that is, $C_W(S) \subseteq S$ and $S \subseteq C_W(S)$. If $X \cap Y = S$, then $g(X) \subseteq g(S)$ and $g(Y) \subseteq g(S)$ (Property 4). Therefore, $(g(X) \cup g(Y)) \subseteq g(S)$. According to Property 4 and Property 5, $C_W(S) = f(g(S)) \subseteq f(g(X) \cup g(Y)) = (f(g(X)) \cap f(g(Y))) = (C_W(X) \cap C_W(Y)) = (X \cap Y) = S$. That is, $C_W(S) \subseteq S$. According to Property 2, we have $S \subseteq C_W(S)$.

III. CLOSTREAM ALGORITHM

In this section, we first introduce our structures and then describe the CloStream algorithm in detail. The algorithm consists of two sub-procedures: CloStream+ and CloStream—. CloStream+ is used to update closed itemsets when a transaction arrives to the window, while deletion operation is used to update closed itemsets when a transaction leaves from the window. We use two data structures, called Closed Table and Cid List, to maintain the information of closed itemsets over a data stream. The details of the structures are described as follows.

Closed Table consists of three fields: *Cid*, *CItemset* and *SC*. Each closed itemset (CItemset) X is assigned a unique closed itemset identifier (Cid) which is a non-zero positive integer, and its support count (SC) is also recorded in a Closed Table. Each record in a Closed Table is denoted as $(c, X, SC(X))$, where c , X and $SC(X)$ are the values of Cid, CItemset and SC. Given a Cid c of an itemset, the itemset and its support count can be directly retrieved from the Closed Table. Initially, the first record in a Closed Table is set to $(0, \text{null}, 0)$.

Cid List consists of two fields: *Item* and *Cidset* field. The Cidset for an item i_p is denoted as $\text{cidset}(i_p)$ which is a set of Cids of closed itemsets containing item i_p . Table II and Table III show the Closed Table and Cid List after the five transactions in Figure 1 arrive to the first window. In Table III, $\text{cidset}(B) = \{2, 3\}$, since item B is contained in the two closed itemsets $\{AB\}$ and $\{ABC\}$, and their Cids are 2 and 3, respectively.

TABLE II. ORIGINAL CLOSED TABLE

Cid	CItemset	SC
0	null	0
1	{CD}	2
2	{AB}	3
3	{ABC}	2
4	{C}	4
5	{ACD}	1
6	{A}	4
7	{AC}	3

TABLE III. THE ORIGINAL CIDLIST

Item	Cidset
A	{2, 3, 5, 6, 7}
B	{2, 3}
C	{1, 3, 4, 5, 7}
D	{1, 5}

A. CloStream+

In this subsection, we describe the procedure *CloStream+*. Let t_A be a new transaction, D_B be the set of the transactions in the current window, and $D_U = D_B \cup \{t_A\}$ be the set of the transactions in the next window after adding t_A . $SC_{D_B}(X)$ and $SC_{D_U}(X)$ are the support counts of the itemset X in D_B and D_U , respectively. $C_{D_B}(X)$ and $C_{D_U}(X)$ are the closures of X in D_B and D_U , respectively. The sets of all the closed itemsets in D_B and D_U are denoted as C_{D_B} and C_{D_U} , respectively.

When a transaction $t_A = \{i_1, i_2, \dots, i_k\}$ ($i_j \in I$, $1 \leq j \leq k$) arrives to D_B , we observe that the support count of t_A is increased by one

but the support count of t_A 's supersets would not be increased. According to Property 3, t_A must be a closed itemset in D_U . Besides, only the supports of the subsets of t_A would be increased and thus only the subsets of t_A would be changed from non-closed to closed.

According to Theorem 1, the closed subsets of t_A in D_U can be obtained by performing intersection operations on t_A and all the closed itemsets in D_B . However, this process is very time-consuming, since the number of closed itemsets in D_B may be very large and there may be many closed itemsets without common items with t_A . In order to avoid performing a large number of invalid intersections, CloStream⁺ uses a Cid List and a SET function to find those closed itemsets which at least have a common item with t_A , that is, the intersections of the closed itemsets and t_A are not empty.

The SET function is defined as follows: $SET(\{t_A\}) = \text{cidset}(i_1) \cup \dots \cup \text{cidset}(i_k)$, where $\text{cidset}(i_j)$ ($1 \leq j \leq k$) is the cidset for item i_j in Cid List. Each Cid in $SET(\{t_A\})$ represents a closed itemset which has at least a common item with t_A . Therefore, a lot of intersection operations can be reduced, since CloStream⁺ only performs intersection operations on t_A and those closed itemsets whose Cids are in $SET(\{t_A\})$.

By performing intersection operations on t_A and every closed itemsets whose Cids are in $SET(\{t_A\})$, we can obtain the closed subsets of t_A in D_U . In order to update the supports of the closed subsets of t_A in D_U , we can find their closures in D_B for each of them. The support counts of them in D_U are equal to the support counts of their closures in D_B increasing by one. CloStream⁺ uses a temp table, which is denoted as $Temp_A$, to store the closed subsets of t_A and their closures in D_B when a transaction t_A is added. $Temp_A$ is a hash structure and consists of two fields: *UItemset* and *Closure_Id*. Each record in $Temp_A$ is denoted as (S, t) , where S is a closed itemset which needs to be updated and t is the identifier of the closure of S . In the following, we describe the algorithm for CloStream⁺ in details.

As t_A is added to the current window, t_A is closed in D_U and then we put t_A into $Temp_A$. Initially, the *Closure_Id* of t_A is set to 0. To obtain the closed itemsets which need to be updated, CloStream⁺ performs intersection operations on t_A and each closed itemset whose Cid i is in $SET(\{t_A\})$. Let an itemset whose Cid is i be denoted as $CT[i]$. For an itemset $S = t_A \cap CT[i]$ ($i \in SET(\{t_A\})$), S is closed in D_U according to Theorem 1. If S is not in $Temp_A$, then S is put into $Temp_A$ and its current *Closure_Id* is i , since $CT[i]$ is a superset of S and may be the closure of S . Otherwise, the current *Closure_Id* of S , say, t , is replaced with i if $SC_{DB}(CT[i])$ is greater than $SC_{DB}(CT[t])$. This is because $CT[i]$ is not the closure of S and $CT[i]$ may be the closure of S .

After performing intersection operations on t_A and each closed itemset whose Cid is in $SET(\{t_A\})$, all closed itemsets which need to be updated and the Cids of their closures in D_B are stored in $Temp_A$. CloStream⁺ uses the information in $Temp_A$ to update the Closed Table and Cid List. For each record (u, c) in $Temp_A$, if u equals to $CT[c]$, then $SC_{DU}(CT[c])$ is increased by 1 since u is originally closed in D_B . Otherwise, u is a new closed itemset and assigned a unique Cid n .

TABLE IV. THE CLOSED TABLE AFTER ADDING t_6

Cid	CItemset	SC
0	null	0
1	{CD}	2
2	{AB}	3
3	{ABC}	2
4	{C}	5
5	{ACD}	1
6	{A}	4
7	{AC}	3
8	{BC}	3
9	{B}	4

TABLE V. THE CIDLIST AFTER ADDING t_6

Item	Cidset
A	{2, 3, 5, 6, 7}
B	{2, 3, 8, 9}
C	{1, 3, 4, 5, 7, 8}
D	{1, 5}

CloStream⁺ adds a new record with Cid = n , CItemset = S , and $SC_{DU} = (SC_{DB}(CT[c]) + 1)$ to the Closed Table. Besides, identifier n is also added into the cidsets for the items that contained in u in Cid List.

After dealing with all records in $Temp_A$, the information of all the closed itemsets in D_U are maintained in the Closed Table and Cid List, and the allocated memory space for $Temp_A$ can be released. All frequent closed itemsets can be obtained by scanning Closed Table once.

B. A Running Example for CloStream⁺

After adding t_1, t_2, t_3, t_4 and t_5 into the first window in Figure 1, the Closed Table and Cid List are shown in Table II and III, respectively. As $t_6 = \{BC\}$ arrives, $(\{BC\}, 0)$ is put into $Temp_A$. Then CloStream⁺ performs intersection operations on each closed itemset whose Cid is in $SET(\{BC\}) = \{1, 2, 3, 4, 5, 7\}$. The intersection of t_6 and $CT[1] = \{CD\}$ is $\{C\}$. Because $\{C\}$ does not exist in $Temp_A$, the record $(\{C\}, 1)$ is put into $Temp_A$. Next, the intersection of t_6 and $CT[2] = \{AB\}$ is $\{B\}$. Because $\{BC\}$ does not exist in $Temp_A$, the record $(\{BC\}, 2)$ is also put into $Temp_A$. Next, the intersection of t_6 and $CT[3] = \{ABC\}$ is $\{BC\}$. Since $\{BC\}$ exists in $Temp_A$ along with its current *Closure_Id* 0, we compare $SC_{DB}(CT[3])$ with $SC_{DB}(CT[0])$. Because $SC_{DB}(CT[3])$ is greater than $SC_{DB}(CT[0])$, the *Closure_Id* of $\{BC\}$ is replaced with 3. CloStream⁺ uses the same way to deal with the other Cids which belong to $SET(\{BC\})$. Finally, the content of $Temp_A$ is shown in Table VI, which includes all closed itemsets in t_6 after adding t_6 and the identifiers of their closures before adding t_6 .

CloStream⁺ uses $Temp_A$ to update Closed Table and Cid List. In $Temp_A$, the first record is $(\{BC\}, 3)$, since $\{BC\}$ is not equal to $CT[3]$, CloStream⁺ assigns 8 as a Cid for $\{BC\}$ and $SC_{DU}(\{BC\}) = SC_{DB}(CT[3]) + 1$. Cid 8 is then added into $\text{cidset}(B)$ and $\text{cidset}(C)$ respectively. The next record in $Temp_A$ is $(\{C\}, 4)$. Because $\{C\}$ equals to $CT[4]$, which means that $\{C\}$ is closed in the first window, the support count of $CT[4] = \{C\}$ is increased by 1. For the last record $(\{B\}, 2)$, since $\{B\}$ is not equal to $CT[2]$, which means that $\{B\}$ is a new closed itemset, CloStream⁺ assigns 9 as a Cid for $\{B\}$ and $SC_{DU}(\{B\}) = SC_{DB}(CT[2]) + 1$.

TABLE VI. TEMP_A

UItemset	Closure_Id
{BC}	3
{C}	4
{B}	2

Besides, Cid 9 is added into the cidset(B). Finally, the updated Closed Table and Cid List are shown in Table IV and Table V, respectively.

C. CloStream⁻

In this subsection, we describe the procedure *CloStream⁻*. Let t_D be the oldest transaction in the current window W , D_B be the original set of transactions, and $D_D = D_B - \{t_D\}$ be the set of the transactions after deleting t_A from D_B . $SC_{DB}(X)$ and $SC_{DD}(X)$ are the support counts for the itemset X in D_B and D_D , respectively. The closures of X in D_B and D_D are denoted as $C_{DB}(X)$ and $C_{DD}(X)$, respectively. The sets of closed itemsets in D_B and D_D are denoted as C_{DB} and C_{DD} , respectively.

Whenever a transaction $t_D = \{i_1, i_2, \dots, i_k\} (i_j \in I, 1 \leq j \leq k)$ is deleted from the database D_B , only the support counts of the subsets of t_D will be decreased by one, that is, only the subsets of t_D may be changed from closed to non-closed. Therefore, *CloStream⁻* first find out the closed subsets of t_D in D_B and then determine which closed subsets of t_D will turn out to be non-closed. The closed subsets of t_D in D_B can be obtained by performing intersection operations on t_D and all the closed itemsets in C_{DB} according to Theorem 1.

However, it is very time-consuming to perform a large number of intersections, since there is a large number of closed itemsets in D_B . In order to avoid performing a lot of invalid intersections, *CloStream⁻* also uses Cid List and computes the SET function to obtain the closed itemsets which have at least a common item with t_D .

CloStream⁻ uses a temp table, called *Temp_D*, to store the information about closed subsets of t_D in D_B , which consists of a triple: *DItemset*, *Closure_Id* and *HS*. Suppose that each record in *Temp_D* is denoted as (S, c, h) , in which S is a closed subsets of t_D , c is an identifier of the closure of S in D_B , and h is the identifier of a closed superset of S , which is not contained in t_D and whose support is the largest among the supports of all the supersets of S . *HS* is used to check if S is closed in D_D . If $SC_{DD}(S) = SC_{DD}(CT[h])$, then S is non-closed in D_D , since there is a superset $CT[h]$ with the same support as S . For a closed superset Z of S ($Z \subset t_D$), $SC_{DD}(Z)$ must not be equal to $SC_{DD}(S)$, since the support counts of S and Z are decreased by one simultaneously after deleting t_D . In the following, we describe the algorithm for *CloStream⁻* in details.

As t_D is deleted from the database, *CloStream⁻* puts t_D into *Temp_D* since t_D is a closed itemset in D_B . The *Closure_Id* and *HS* of t_D are set to zeros. To obtain the closed subsets of t_D in D_B , *CloStream⁻* performs intersection operations on t_D and each closed itemset $CT[i]$ whose cid i is in $SET(\{t_D\})$. Suppose that $S = t_D \cap CT[i]$, then S is a closed subset in D_B according to Theorem 1. If S is not in *Temp_D*, then $(S, i, 0)$ is put into *Temp_D*, since $CT[i]$ may be the closure of S . Otherwise, if S exists in *Temp_D* with *Closure_Id* = c and *HS* = h , there are two cases need to be considered. Case 1: if $SC_{DB}(CT[i])$ is greater than

$SC_{DB}(CT[c])$, then $CT[i]$ may be the closure of S but $CT[c]$ is not (Property 3). In this case, (S, c, h) is replaced with (S, i, c) . Case 2: if $SC_{DB}(CT[i])$ is not greater than $SC_{DB}(CT[c])$ but greater than $SC_{DB}(CT[h])$, then (S, c, h) is replaced with (S, c, i) . Otherwise, the record (S, c, i) remains unchanged.

After performing intersection operations on t_D and each closed itemset whose Cid is in $SET(\{t_D\})$, each closed subset S of t_D which need to be updated and its closure c in D_B are stored in *Temp_D*. According to Definition 1, $CT[c] = S$. *CloStream⁻* uses the identifier c to find S in Closed Table and decreases the support count of S by 1. After finding all the closed itemsets which need to be updated and updating their support counts, *CloStream⁻* then determines which closed itemsets become non-closed.

For each record (S, c, h) in *Temp_D*, if $SC_{DD}(CT[c]) = SC_{DD}(CT[h])$, then S becomes non-closed in D_D since there is a superset $CT[h]$ with the same support as S . *CloStream⁻* removes the information of $CT[c]$ from the Closed Table and Cid List. Notice that if $SC_{DD}(CT[c]) = 0$, the information of $CT[c]$ is also be removed. After processing all the records in *Temp_D*, the information of the closed itemsets after deleting t_D is maintained in the updated Closed Table and Cid List, and the allocated memory space for *Temp_D* can be released. By scanning Closed Table once, frequent closed itemsets and their supports in D_D can be obtained.

TABLE VII. THE CLOSED TABLE AFTER AFTER DELETING t_1

Cid	CItemset	SC
0	null	0
2	{AB}	3
3	{ABC}	2
4	{C}	4
5	{ACD}	1
6	{A}	4
7	{AC}	3
8	{BC}	3
9	{B}	4

TABLE VIII. THE CIDLIST AFTER DELETING t_1

Item	Cidset
A	{2, 3, 5, 6, 7}
B	{2, 3, 8, 9}
C	{3, 4, 5, 7, 8}
D	{5}

D. A Running Example for CloStream⁻

Suppose D_B is the set of the transactions after adding t_6 into Table I. The Closed Table and Cid List are shown in Table IV and Table V, respectively. When t_1 is deleted from D_B , t_1 is put into *DItemset* of *Temp_D* since t_1 is a closed itemset in D_B . The values of *Closure_Id* and *HS* are set to zeros. In order to efficiently obtain all the closed subsets of t_1 , *CloStream⁻* performs intersection operations on $t_1 = \{CD\}$ and each closed itemset whose cid is in $SET(\{CD\}) = \{1, 3, 4, 5, 7, 9\}$.

The intersection of t_1 and $CT[1] = \{CD\}$ is $\{CD\}$. We find that $\{CD\}$ exists in *Temp_D* with *Closure_Id* = 0 and *HS* = 0. The record $(\{CD\}, 0, 0)$ is replaced with $(\{CD\}, 1, 0)$ since $SC_{DB}(CT[1])$ is greater than $SC_{DB}(CT[0])$. Next, the intersection of t_1 and $CT[3] = \{ABC\}$ is $\{C\}$. Since $\{C\}$ does not exist in *Temp_D*, the record $(\{C\}, 3, 0)$ is put into *Temp_D*.

The intersection of t_1 and $CT[4] = \{C\}$ is $\{C\}$. We find that $\{C\}$ exists in $Temp_D$ with $Closure_Id = 3$ and $HS = 0$. Because $SC_{DB}(CT[4])$ is greater than $SC_{DB}(CT[3])$, the $Closure_Id$ of $\{C\}$ is replaced with 4. Besides, the HS of $\{C\}$ is replaced with 3, since $C_{DB}(CT[3])$ is greater than $SC_{DB}(CT[0])$. Therefore, the current $Closure_Id$ and HS of $\{C\}$ are 4 and 3, respectively.

The next Cid in $SET(\{CD\})$ is 5. The intersection of t_1 and $CT[5] = \{ACD\}$ is $\{CD\}$, and $\{CD\}$ exists in $Temp_D$ with $Closure_Id = 1$ and $HS = 0$. Because $SC_{DB}(CT[5])$ is not greater than $SC_{DB}(CT[1])$ but greater than $SC_{DB}(CT[0])$, only the value of HS is updated as 5.

After performing intersection operations on t_1 and all the closed itemsets whose $Cids$ are in $SET(\{CD\})$, the content of $Temp_D$ is shown in Table IX. CloStream then decreases support counts of $\{CD\}$ and $\{C\}$ by one from Closed Table.

In Table IX, the first record is $(\{CD\}, 1, 5)$. Since $SC_{DD}(CT[1]) = SC_{DD}(CT[5])$, $CT[1] = \{CD\}$ turns out to be non-closed after deleting t_1 . For the second record $(\{C\}, 4, 7)$, $CT[4] = \{C\}$ is still closed in D_D , since $SC_{DD}(CT[4]) \neq SC_{DD}(CT[7])$. After processing all the records in $Temp_D$, the updated Closed Table and Cid List are shown in Table VII and Table VIII. By scanning Closed Table once, all the frequent closed itemsets in D_D and their support counts can be obtained.

TABLE IX. $Temp_D$

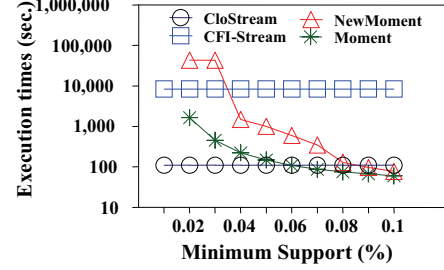
DItemset	Closure_Id	HS
$\{CD\}$	1	5
$\{C\}$	4	7

IV. EXPERIMENTAL RESULTS

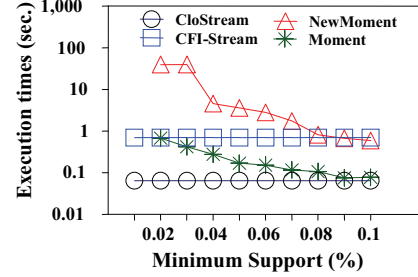
In this section, we evaluate the performance of our CloStream algorithm and compare it with the three famous algorithms CFI-Stream, Moment and NewMoment. The experiments are performed on a 1.83 GHz Intel Core 2 Duo Processor with 2 Gigabyte memory, and running on Windows Vista. CloStream and CFI-Stream were coded in Java language. Moment [3] and NewMoment [7] are provided by their authors and coded in C++ language. To evaluate the performance of these algorithms, we generate a synthetic dataset T10I4D10100 from IBM data generator [1]. The parameters are described as follows: D is the total number of transactions; T is average transaction size; I is the average size of maximal potential frequent itemsets; N ($=2000$) is the number of distinct items. Besides, a real-world dataset BMS-Web-View-1 was downloaded from FIMI Repository [11], where $I = 497$, $T = 2.5$ and $D = 59,602$. The maximum length of the transactions in BMS-Web-View-1 is 267.

Fig. 2(a) shows the loading time of the first window for the four algorithms on the dataset T10I4D10100 under different minimum supports as the size of sliding window is set to 10K. From Fig. 2(a), we can see that the execution times for Moment and NewMoment increase as the minimum support decreases, since the lower the minimum support is, the larger the search space is for Moment and NewMoment. On the other hand, the execution times for CFI-Stream and CloStream are not affected by different minimum supports, since CFI-Stream and CloStream maintain all the closed itemsets in their structures. When min_sup is less than 0.06%, CloStream outperforms

Moment and NewMoment, and the performance gap significantly increases as the minimum support decreases.



(a) Times for loading the first window



(b) Times for sliding 100 consecutive windows

Figure 2. Execution times on dataset T10I4D10100 for the four algorithms

When min_sup is high, Moment and NewMoment run slightly faster than CloStream since they only generate frequent closed itemsets, but CloStream generates all the closed itemsets. In Fig. 2(a), CFI-Stream runs slower than CloStream about 100 times, since CFI-Stream needs to generate all the subsets (i. e., candidate closed itemsets) for each transaction and search all the supersets for each candidate from DIU Tree to check if the candidate is closed. In other words, a transaction of length k will lead CFI-Stream to generate $(2^k - 1)$ candidates and search from DIU-Tree $(2^k - 1)$ times. Therefore, CFI-Stream is unfavorable to deal with long transactions.

Fig. 2(b) shows the average execution times for the four algorithms after sliding 100 consecutive windows on the dataset T10I4D10100 under different minimum supports. From Fig. 2(b), we can see that CloStream runs faster than Moment and CFI-Stream about 10 times when $min_sup = 0.02\%$, and outperforms NewMoment about 100 times when min_sup is less than 0.03%, since Moment, NewMoment and CFI-Stream need to search their structure to determine which itemsets are closed. However, whatever the transactions are added or deleted, our CloStream algorithm only needs to do some intersection operations without doing any searching operations on the previous closed itemsets.

Fig. 3(a) shows the loading time of the first window for the algorithms Moment, NewMoment and CloStream on the real dataset BMS-Web-View-1 under different minimum supports as the size of sliding window is set to 10K. Since the maximum length of the transactions in this dataset is 267, CFI-Stream needs to enumerate $(2^{267} - 1)$ candidates and search from DIU Tree over 100 billion times. The performance for CFI-Stream is much worse than the three algorithms. Therefore, we only

compare the performance of our CloStream with Moment and NewMoment on this dataset. From Fig. 3(a), we can see that CloStream runs faster than Moment and NewMoment over 100 times when min_sup is less than 0.03% and 0.06%, respectively. Particularly, when min_sup is less than 0.03%, Moment runs out of memory, since it not only captures lots of non-closed nodes in CET-Tree but also maintains all frequent closed itemsets in CET-Tree and hash table. Although NewMoment is able to run when min_sup is less than 0.05%, it requires more than 2.5 hours.

Fig. 3(b) shows the average execution times for the three algorithms over 100 consecutive windows on dataset BMS-Web-View-1 under different minimum supports. From Fig. 3(b), we can see that CloStream significantly outperforms NewMoment and runs faster than Moment when min_sup is less than 0.04%. Although NewMoment uses less memory space than Moment, it requires a long execution time. Particularly, when $min_sup = 0.05\%$, the total execution time of NewMoment for sliding 100 consecutive windows is over $(100 \times 1,000) = 100,000$ seconds, while CloStream only takes about $(0.1 \times 100) = 10$ seconds under the same threshold.

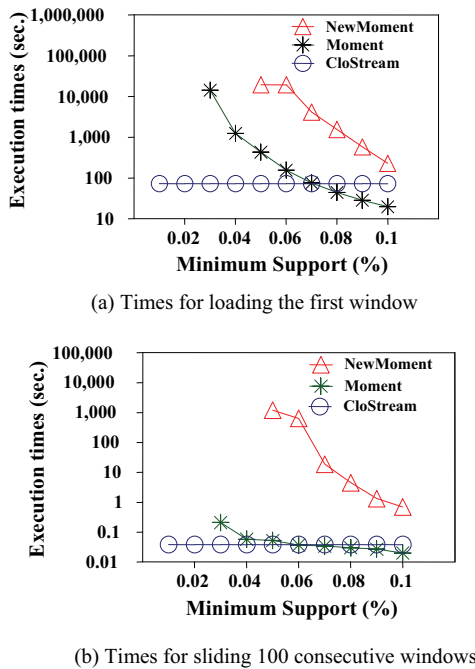


Figure 3. Execution times on the real-world dataset BMS-Web-View-1 for the three algorithms

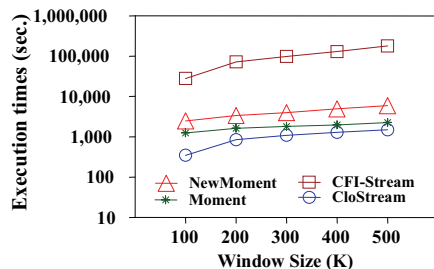


Figure 4. Scalability for the four algorithms under various sliding window sizes

Fig. 4 shows the execution times of the four algorithms for loading the first window under different sliding window sizes on dataset T514. In this experiment, the size of sliding window is increased from 100K to 500K and the minimum supports for Moment and NewMoment are set to 0.005%. From Fig 4 we can see that CloStream outperforms the other three algorithms and has a good scalability for large dataset.

V. CONCLUSIONS

In this paper, we propose a novel algorithm CloStream to discover frequent closed itemsets over stream sliding windows. We use Cid List and the SET function to find the closed itemsets which have the common items with the added or deleted transaction. Different from the previous approaches which need to take a lot of time to search from a tree structure, CloStream only needs to do the intersections on the closed itemsets which return from the SET function and the added or deleted transaction without searching from our proposed structures. Besides, all the frequent closed itemsets can be obtained online based on any user-specified thresholds. The experimental results also show that our CloStream algorithm outperforms the three famous algorithms CFI-Stream, Moment and NewMoment on both synthetic datasets and real-world dataset.

REFERENCES

- [1] R. Agrawal, R. Srikant, "Fast Algorithm for Mining Association Rules," in Proc. of Int'l Conf. on Very Large Data Bases, pp. 487-499, 1994.
- [2] J. Chang, Y. Ke, W. Ng, "Research Issues in Data Stream Association Rule Mining," Knowledge and Information Systems (KAIS), vol. 16, issue 1, pp. 1-27, 2008.
- [3] Y. Chi, H. Wang, P. Yu and R. Muntz, "Catch the moment: maintaining closed frequent itemsets over a data stream sliding window," Knowledge and Information Systems (KAIS), vol. 10, issue 3, pp. 265-294, 2006.
- [4] J. Han, J. Pei, Y. Yin and R. Mao, "Mining frequent patterns without candidate generation: a frequent-pattern tree approach," Data Mining and Knowledge Discovery (DMKD), vol. 8, issue 1, pp. 53-87, 2004.
- [5] N. Jiang and L. Gruenwald, "CFI-Stream: mining closed frequent itemsets in data streams," in Proc. of ACM SIGKDD Int'l Conf. on Knowledge Discovering and Data Mining, pp. 592-597, 2006.
- [6] C. Lucchese, S. Orlando and R. Perego, "Fast and Memory Efficient Mining of Frequent Closed Itemsets," IEEE Transactions on Knowledge and Data Engineering (TKDE), vol. 18, issue 1, pp. 21-36, 2006.
- [7] H. F. Lee, C. C. Ho and S. Y. Lee, "Incremental updates of closed frequent itemsets over continuous data streams," Expert System with Applications (ESWA), vol. 36, issue 2, pp. 2451-2458, 2009.
- [8] N. Pasquier, Y. Bastide, R. Taouil and L. Lakhal, "Generating a Condensed representation for Association Rules," Journal of Intelligent Information Systems (JIIS), vol. 24, issue 1, pp. 29-60, 2005.
- [9] S. J. Yen, Y. S. Lee, C. K. Wang, J. W. Wu and L. Y. Ouyang, "The Studies of Mining Frequent Patterns Based on Frequent Pattern Tree," in Proc. of Int'l Conf. on Pacific-Asia Conference on Knowledge Discovery and Data Mining (PAKDD), pp. 398-416, 2009.
- [10] M. J. Zaki and C. J. Hsiao, "Efficient Algorithms for Mining Closed Itemsets and Their Lattice Structure," IEEE Transactions on Knowledge and Data Engineering (TKDE), vol. 17, issue 4, pp. 462-478, 2005.
- [11] Frequent Itemset Implantation Repository (FIMI), <http://fimi.cs.helsinki.fi/>