Moment: Maintaining Closed Frequent Itemsets over a Stream Sliding Window

Yun Chi*, Haixun Wang†, Philip S. Yu†, Richard R. Muntz*
*Department of Computer Science, University of California, Los Angeles, CA 90095
†IBM Thomas J. Watson Research Center, Hawthorne, NY 10532

Abstract

This paper considers the problem of mining closed frequent itemsets over a sliding window using limited memory space. We design a synopsis data structure to monitor transactions in the sliding window so that we can output the current closed frequent itemsets at any time. Due to time and memory constraints, the synopsis data structure cannot monitor all possible itemsets. However, monitoring only frequent itemsets will make it impossible to detect new itemsets when they become frequent. In this paper, we introduce a compact data structure, the closed enumeration tree (CET), to maintain a dynamically selected set of itemsets over a sliding-window. The selected itemsets consists of a boundary between closed frequent itemsets and the rest of the itemsets. Conceptual drifts in a data stream are reflected by boundary movements in the CET. In other words, a status change of any itemset (e.g., from non-frequent to frequent) must occur through the boundary. Because the boundary is relatively stable, the cost of mining closed frequent itemsets over a sliding window is dramatically reduced to that of mining transactions that can possibly cause boundary movements in the CET. Our experiments show that our algorithm performs much better than previous approaches.

1 Introduction

Data streams arise with the introduction of new application areas, including ubiquitous computing and electronic commerce. Mining data streams for knowledge discovery is important to many applications, such as fraud detection, intrusion detection, trend learning, etc. In this paper, we consider the problem of mining closed frequent itemsets on data streams.

Mining frequent itemset on static datasets has been studied extensively. However, data streams have posed new challenges. First, data streams are continuous, high-speed, and unbounded. Archiving everything from streams is impossible, not to mention mining association rules from them using algorithms that require multiple scans. Second, the data distribution in streams are usually changing with time, and very often people are interested in the most recent patterns.

*The work of these two authors was partly supported by NSF under Grant Nos. 0086116, 0085773, and 9817773.

It is thus of great interest to mine itemsets that are *currently* frequent. One approach is to always focus on frequent itemsets in the most recent window. A similar effect can be achieved by exponentially discounting old itemsets. For the window-based approach, we can immediately come up with two naive methods:

1. Regenerate frequent itemsets from the entire window whenever a new transaction comes into or an old transaction leaves the window.
2. Store every itemset, frequent or not, in a traditional data structure such as the prefix tree, and update its support whenever a new transaction comes into or an old transaction leaves the window.

Clearly, method 1 is not efficient. In fact, as long as the window size is reasonable, and the conceptual drifts in the stream is not too dramatic, most itemsets do not change their status (from frequent to non-frequent or from non-frequent to frequent) often. Thus, instead of regenerating all frequent itemsets every time from the entire window, we shall adopt an *incremental* approach.

Method 2 is incremental. However, its space requirement makes it infeasible in practice. The prefix tree [1] is often used for mining association rules on static data sets. In a prefix tree, each node n_I represents an itemset I and each child node of n_I represents an itemset obtained by adding a new item to I. The total number of possible nodes is exponential. Due to memory constraints, we cannot keep a prefix tree in memory, and disk-based structures will make real time update costly.

In view of these challenges, we focus on a *dynamically selected* set of itemsets that are i) informative enough to answer at any time queries such as "what are the (closed) frequent itemsets in the current window", and at the same time, ii) small enough so that they can be easily maintained in memory and updated in real time.

The problem is, of course, what itemsets shall we select for this purpose? To reduce memory usage, we are tempted to select, for example, nothing but frequent (or even closed frequent) itemsets. However, if the frequency counts of a non-frequent itemset is not monitored, we will never know when it becomes frequent. A naive approach is to monitor all itemsets whose support is above a reduced threshold $minsup-\epsilon$, so that we will not miss itemsets whose current support is within ϵ of $minsup$ when they become frequent. This approach is apparently not general enough.

In this paper, we design a synopsis data structure to keep track of the boundary between closed frequent itemsets and

the rest of the itemsets. Conceptual drifts in a data stream are reflected by boundary movements in the data structure. In other words, a status change of any itemset (e.g., from non-frequent to frequent) must occur through the boundary. The problem of mining an infinite amount of data is thus converted to mine data that can potentially change the boundary in the current model. Because most of the itemsets do not often change status, which means the boundary is stable, and even if some does, the boundary movement is local, the cost of mining closed frequent itemsets is dramatically reduced.

Our Contribution This paper makes the following contributions: (1) We introduce a novel algorithm, Moment[1], to mine closed frequent itemsets over data stream sliding windows. To the best of our knowledge, our algorithm is the first one for mining *closed* frequent itemsets in data streams. (2) We present an in-memory data structure, the *closed enumeration tree* (CET), which monitors closed frequent itemsets as well as itemsets that form the boundary between the closed frequent itemsets and the rest of the itemsets. We show that i) a status change of any itemset (e.g., from non-frequent to frequent) must come through the boundary itemsets, which means we do not have to monitor itemsets beyond the boundary, and ii) the boundary is relatively stable, which means the update cost is minimum. (3) We introduce a novel algorithm to maintain the CET in an efficient way. Experiments show Moment has significant performance advantage over state-of-the-art approaches for mining frequent itemsets in data streams.

Related Work Mining frequent itemsets from data streams has been investigated by many researchers. Manku et al [13] proposed an approximate algorithm that for a given time t, mines frequent itemsets over the *entire* data streams up to t. Charikar et al [6] presented a 1-pass algorithm that returns most frequent *items* whose frequencies satisfy a threshold with high probabilities. Chang et al [5] presented an algorithm, *estDec*, that mines recent frequent itemsets where the frequency is defined by an aging function. Giannella et al [9] proposed an approximate algorithm for mining frequent itemsets in data streams during arbitrary time intervals. An in-memory data structure, *FP-stream*, is used to store and update historic information about frequent itemsets and their frequency over time and an aging function is used to update the entries so that more recent entries are weighted more. Asai et al [3] presented an online algorithm, *StreamT*, for mining frequent rooted ordered trees. To reduce the number of subtrees to be maintained, an update policy that is similar to that in online association rule mining [11] was used and therefore the results are inexact. In all these studies, approximate algorithms were adopted. In contrast, our algorithm is an exact one because we assume that the approximation step has been implemented through the sampling scheme and our algorithm works on a sliding window containing the random samples (which are a synopsis of the data stream).

In addition, closely related to our work, Cheung et al [7, 8] and Lee et al [12] proposed algorithms to maintain discovered frequent itemsets through incremental updates. Although these algorithms are exact, they focused on mining *all* frequent itemsets (as do the above approximate algorithms). The large number of frequent itemsets makes it impractical to maintain information about all frequent itemsets using in-memory data structures. In contrast, our algorithm maintains only closed frequent itemsets. As demonstrated by extensive experimental studies, e.g., [16], there are usually much fewer closed frequent itemsets compared to the total number of frequent itemsets.

[1] Maintaining Closed Frequent Itemsets by Incremental Updates

2 Problem Statement

Preliminaries Given a set of items Σ, a database $\mathcal{D}$ wherein each transaction is a subset of Σ, and a threshold s called the *minimum support* (*minsup*), $0 < s \leq 1$, the frequent itemset mining problem is to find all itemsets that occur in at least $s|\mathcal{D}|$ transactions.

We assume that there is a lexicographical order among the items in Σ and we use $X \prec Y$ to denote that item X is lexicographically smaller than item Y. Furthermore, an itemset can be represented by a sequence, wherein items are lexicographically ordered. For instance, $\{A, B, C\}$ is represented by ABC, given $A \prec B \prec C$. We also abuse notation by using $\prec$ to denote the lexicographical order between two itemsets. For instance, $AB \prec ABC \prec CD$.

As an example, let $\Sigma = \{A, B, C, D\}$, $\mathcal{D} = \{CD, AB, ABC, ABC\}$, and $s = \frac{1}{2}$, then the frequent itemsets are

$$\mathcal{F} = \{(A, 3), (B, 3), (C, 3), (AB, 3), (AC, 2), (BC, 2), (ABC, 2)\}$$

In $\mathcal{F}$, each frequent itemset is associated with its support in database $\mathcal{D}$.

Combinatorial Explosion According to the *a priori* property, any subset of a frequent itemset is also frequent. Thus, algorithms that mine *all* frequent itemsets often suffer from the problem of combinatorial explosion.

Two solutions have been proposed to alleviate this problem. In the first solution (e.g., [4], [10]), only *maximal* frequent itemsets are discovered. A frequent itemset is *maximal* if none of its proper supersets is frequent. The total number of maximal frequent itemsets $\mathcal{M}$ is much smaller than that of frequent itemsets $\mathcal{F}$, and we can derive each frequent itemset from $\mathcal{M}$. However, $\mathcal{M}$ does not contain information of the support of each frequent itemset unless it is a maximal frequent itemset. Thus, mining only maximal frequent itemsets loses information.

In the second solution (e.g., [14], [15]), only *closed* frequent itemsets are discovered. An itemset is closed if none of its proper supersets has the same support as it has. The total number of closed frequent itemsets $\mathcal{C}$ is still much smaller than that of frequent itemsets $\mathcal{F}$. Furthermore, we can derive $\mathcal{F}$ from $\mathcal{C}$, because a frequent itemset I must be a subset of one (or more) closed frequent itemset, and I's support is equal to the maximal support of those closed itemsets that contain I.

In summary, the relation among $\mathcal{F}$, $\mathcal{C}$, and $\mathcal{M}$ is $\mathcal{M} \subseteq \mathcal{C} \subseteq \mathcal{F}$. The closed and maximal frequent itemsets for the above examples are

$$\mathcal{C} = \{(C, 3), (AB, 3), (ABC, 2)\}$$
$$\mathcal{M} = \{(ABC, 2)\}$$

Since $\mathcal{C}$ is smaller than $\mathcal{F}$, and $\mathcal{C}$ does not lose information about any frequent itemsets, in this paper,set we focus on mining the closed frequent itemsets because they maintain sufficient information to determine all the frequent itemsets as well as their support.

Problem Statement The problem is to mine (closed) frequent itemsets in the most recent N transactions in a data stream. Each transaction has a time stamp, which is used as the *tid* (transaction id) of the transaction. Figure 1 is an example with $\Sigma = \{A, B, C, D\}$ and window size $N = 4$. We use this example throughout the paper with minimum support $s = \frac{1}{2}$.

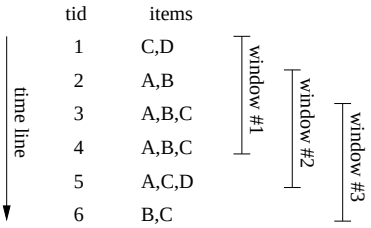


Figure 1: A Running Example

To find frequent itemsets on a data stream, we maintain a data structure that models the current frequent itemsets. We update the data structure incrementally. The combinatorial explosion problem of mining frequent itemsets becomes even more serious in the streaming environment. As a result, on the one hand, we cannot afford keeping track of all itemsets or even frequent itemsets, because of time and space constraints. On the other hand, any omission (for instance, maintaining only $\mathcal{M}$, $\mathcal{C}$, or $\mathcal{F}$ instead of all itemsets) may prevent us from discovering future frequent itemsets. Thus, the challenge lies in designing a compact data structure which does not lose information of any frequent itemset over a sliding window.

3 The Moment Algorithm

We propose the Moment algorithm and an in-memory data structure, the *closed enumeration tree*, to monitor a dynamically selected small set of itemsets that enable us to answer the query "what are the current closed frequent itemsets?" at any time.

3.1 The Closed Enumeration Tree

Similar to a prefix tree, each node n_I in a *closed enumeration tree* (CET) represents an itemset I. A child node, n_J, is obtained by adding a new item to I such that $I \prec J$. However, unlike a prefix tree, which maintains *all* itemsets, a CET only maintains a *dynamically selected* set of itemsets, which include i) closed frequent itemsets, and ii) itemsets that form a *boundary* between closed frequent itemsets and the rest of the itemsets.

As long as the window size is reasonably large, and the conceptual drifts in the stream is not too dramatic, most itemsets do not change their status (from frequent to non-frequent or from non-frequent to frequent). In other words, the effects of transactions moving in and out of a window offset each other and usually do not cause change of status of many involved nodes.

If an itemset dos not change its status, nothing needs to be done except for increasing or decreasing the counts of the involved itemsets. If it does change its status, then, as we will show, the change must come through the boundary nodes, which means the changes to the entire tree structure is still limited.

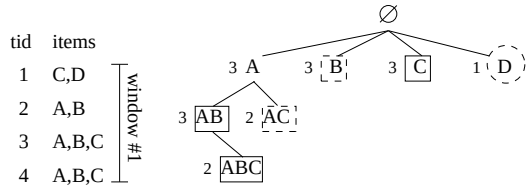


Figure 2: The Closed Enumeration Tree Corresponding to Window #1 (each node is labeled with its $support$)

We further divide itemsets on the boundary into two categories, which correspond to the boundary between frequent and non-frequent itemsets, and the boundary between closed and non-closed itemsets, respectively. Itemsets within the boundary also have two categories, namely the closed nodes, and other intermediary nodes that have closed nodes as descendants. For each category, we define specific actions to be taken in order to maintain a shifting boundary when there are concept drifts in data streams (Section 3.3). The four types of itemsets are listed below.

infrequent gateway nodes A node n_I is an infrequent gateway node if i) I is an infrequent itemset, and ii) n_I's parent or the siblings of n_I's parent (if there are any) are frequent. In Figure 2, D is an infrequent gateway node (represented by dashed circle). In contrast, AD is not an infrequent gateway node (hence it does not appear in the CET), because D is infrequent.

unpromising gateway nodes A node n_I is an unpromising gateway node if i) I is a frequent itemset, and ii) there exists a closed frequent itemset J such that $J \prec I$, $J \supset I$, and J has the same support as I does. In Figure 2, B is an unpromising gateway node because AB has the same support as it does. So is AC because of ABC. In Figure 2, unpromising gateway nodes are represented by dashed rectangles. For convenience of discussion, when a node in the CET is neither an infrequent gateway node nor an unpromising gateway node, we call it a *promising* node.

intermediate nodes A node n_I is an intermediate node if i) I is a frequent itemset, ii) n_I has a child node n_J such that J has the same support as I does, and iii) n_I is not an unpromising gateway node. In Figure 2, A is an intermediate node because its child AB has the same support as A does.

closed nodes These nodes represent closed frequent itemsets in the current sliding-window. A closed node can be an internal node or a leaf node. In Figure 2, C, AB, and ABC are closed nodes, which are represented by solid rectangles.

3.2 Node Properties

We prove some properties for the nodes in the CET. Properties 1 and 2 enable us to prune a large amount of itemsets from the CET, while Property 3 makes sure certain itemsets are not pruned. Together, they enable us to mine closed frequent itemsets over a sliding window using an efficient and compact synopsis data structure.

Property 1. *If n_I is an infrequent gateway node, then any node n_J where $J \supset I$ represents an infrequent itemset.*

Proof. Property 1 is derived from the *a priori* property. □

A CET achieves its compactness by pruning a large amount of the itemsets. It prunes the descendants of n_I and the descendants of n_I's siblings nodes that subsume I. However, it 'remembers' the boundary where such pruning occurs, so that it knows where to start exploring when n_I is no longer an infrequent gateway node. An infrequent gateway node marks such a boundary. In particular, infrequent gateway nodes are leaf nodes in a CET. For example, in Figure 2, after knowing that D is infrequent, we do not explore the subtree under D. Furthermore, we do not join A with D to generate A's child nodes. As a result, a large amount of the itemsets are pruned.

Property 2. *If n_I is an unpromising gateway node, then n_I is not closed, and none of n_I's descendents is closed.*

Proof. Based on the definition of unpromising gateway nodes, there exists an itemset J such that i) $J \prec I$, and ii) $J \supset I$ and $support(J) = support(I)$. From ii), we know n_I is not closed. Let i_{max} be the lexicographically largest item in I. Since $J \prec I$ and $J \supset I$, there must exist an item $j \in J \backslash I$ such that $j \prec i_{max}$. Thus, for any descendant $n_{I'}$ of n_I, we have $j \notin I'$. Furthermore, because $support(J) = support(I)$, itemset $J \backslash I$ must appear in every transaction I appears, which means $support(n_{I'}) = support(n_{\{j\} \cup I'})$, so I' is not closed. □

Descendants of an unpromising gateway node are pruned because no closed nodes can be found there, and it 'remembers' the boundary where such pruning occurs.

Property 3. *If n_I is an intermediate node, then n_I is not closed and n_I has closed descendants.*

Proof. Based on the definition of intermediate nodes, n_I is not closed. Thus, there must exists a closed node n_J such that $J \supset I$ and $support(J) = support(I)$. If $I \prec J$, then n_J is n_I's descendant since $J \supset I$. If $J \prec I$, then n_I is an unpromising gateway node, which means n_I is not an intermediate node. □

Property 3 shows that we cannot prune intermediate nodes.

3.3 Building the Closed Enumeration Tree

In a CET, we store the following information for each node n_I: i) the itemset I itself, ii) the node type of n_I, iii)$support$: the number of transactions in which I occurs, and iv) tid_sum: the *sum* of the tids of the transactions in which I occurs. The purpose of having tid_sum is because we use a hash table to maintain closed itemsets.

The Hash Table We frequently check whether or not a certain node is an unpromising gateway node, which means we need to know whether there is a closed frequent node that has the same support as the current node.

We use a hash table to store all the closed frequent itemsets. To check if n_I is an unpromising gateway node, by definition, we check if there is a closed frequent itemset J such that $J \prec I$, $J \supset I$, and $support(J) = support(I)$.

We can thus use $support$ as the key to the hash table. However, it may create frequent hash collisions. We know if $support(I) = support(J)$ and $I \subset J$, then I and J must occur in the same set of transactions. Thus, a better choice is the set of tids. However, the set of tids take too much space, so we instead use $(support, tid\_sum)$ as the key. Note that tid_sum of an itemset can be incrementally updated. To check if n_I is an unpromising gateway node, we hash on the $(support, tid\_sum)$ of n_I, fetch the list of closed frequent itemsets in the corresponding entry of the hash table, and check if there is a J in the list such that $J \prec I$, $J \supset I$, and $support(J) = support(I)$.

Tree Construction To build a CET, first we create a root node $n_\emptyset$. Second, we create $|\Sigma|$ child nodes for $n_\emptyset$ (i.e., each $i \in \Sigma$ corresponds to a child node $n_{\{i\}}$), and then we call *Explore* on each child node $n_{\{i\}}$. Pseudo code for the *Explore* algorithm is given in Figure 3.

Explore (n_I, $\mathcal{D}$, $minsup$)

```
1:  if support(n_I) < minsup · |D| then
2:      mark n_I an infrequent gateway node;
3:  else if leftcheck(n_I) = true then
4:      mark n_I an unpromising gateway node;
5:  else
6:      foreach frequent right sibling n_K of n_I do
7:          create a new child n_{I∪K} for n_I;
8:          compute support and tid_sum for n_{I∪K};
9:      foreach child n_{I'} of n_I do
10:         Explore(n_{I'}, D, minsup);
11:     if ∃ a child n_{I'} of n_I such that
            support(n_{I'}) = support(n_I) then
12:         mark n_I an intermediate node;
13:     else
14:         mark n_I a closed node;
15:         insert n_I into the hash table;
```

Figure 3: The *Explore* Algorithm

Explore is a depth-first procedure that visits itemsets in lexicographical order. In lines 1-2 of Figure 3, if a node is found to be infrequent, then it is marked as an infrequent gateway node, and we do not explore it further (Property 1). However, the $support$ and tid_sum of an infrequent gateway node have to be stored because they will provide important information during a CET update when an infrequent itemset can potentially become frequent.

In lines 3-4, when an itemset I is found to be non-closed because of another lexicographically smaller itemset, then n_I is an unpromising gateway node. Based on Property 2, we do not explore n_I's descendants, which does not contain any closed frequent itemsets. However, n_I's $support$ and

tid_sum must be stored, because during a CET update, n_I may become promising.

In *Explore*, *leftcheck*(n_I) checks if n_I is an unpromising gateway node. It looks up the hash table to see if there exists a previously discovered closed itemset that has the same support as n_I and which also subsumes I, and if so, it returns *true* (in this case n_I is an unpromising gateway node); otherwise, it returns *false* (in this case n_I is a promising node).

If a node n_I is found to be neither infrequent nor unpromising, then we explore its descendants (lines 6-10). After that, we can determine if n_I is an intermediate node or a closed node (lines 11-15) according to Property 3.

Complexity The time complexity of the *Explore* algorithm depends on the size of the sliding-window N, the minimum support, and the number of nodes in the CET. However, because *Explore* only visits those nodes that are necessary for discovering closed frequent itemsets, so *Explore* should have the same asymptotic time complexity as any closed frequent itemset mining algorithm that are based on traversing the enumeration tree.

3.4 Updating the CET

New transactions are inserted into the window, as old transactions are deleted from the window. We discuss the maintenance of the CET for the two operations: addition and deletion.

Adding a Transaction

In Figure 4, a new transaction T (tid 5) is added to the sliding-window. We traverse the parts of the CET that are related to transaction T. For each related node n_I, we update its $support$, tid_sum, and possibly its node type.

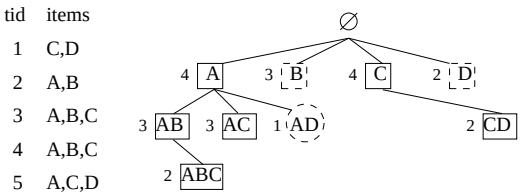


Figure 4: Adding a Transaction

Most likely, n_I's node type will not change, in which case, we simply update n_I's *support* and tid_sum, and the cost is minimum. In the following, we discuss cases where the new transaction T causes n_I to change its node type.

n_I was an infrequent gateway node. If n_I becomes frequent (e.g., from node D in Figure 2 to node D in Figure 4), two types of updates must be made. First, for each of n_I's left siblings it must be checked if new children should be created. Second, the originally pruned branch (under n_I) must be re-explored by calling *Explore*.

For example, in Figure 4, after D changes from an infrequent gateway node to a frequent node, node A and C must be updated by adding new children (AD and CD, respectively). Some of these new children will become new infrequent gateway nodes (e.g., node AD), and others may become other types of nodes (e.g., node CD becomes a closed node). In addition, this update may propagate down more than one level.

n_I was an unpromising gateway node. Node n_I may become promising (e.g., from node AC in Figure 2 to node AC in Figure 4) for the following reason. Originally, $\exists(j \prec i_{max}\ and\ j \notin I)$ s.t. j occurs in each transaction that I occurs. However, if T contains I but not any of such j's, then the above condition does not hold anymore. If this happens, the originally pruned branch (under n_I) must be explored by calling *Explore*.

n_I was a closed node. Based on the following property, n_I will remain a closed node.

Property 4. *Adding a new transaction will not change a node from closed to non-closed, and it will not decrease the number of closed itemsets in the sliding-window.*

Proof. Originally, $\forall J \supset I, support(J) < support(I)$; after adding the new transaction T, $\forall J \supset I$, if $J \subset T$ then $I \subset T$. Therefore if J's support is increased by one because of T, so is I's support. As a result, $\forall J \supset I, support(J) < support(I)$ still holds after adding the new transaction T. However, if a closed node n_I is visited during an addition, its entry in the hash table will be updated. Its $support$ is increased by 1 and its tid_sum is increased by adding the tid of the new transaction. □

n_I was an intermediate node. An intermediate node, such as node A in Figure 2, can possibly become a closed node after adding a new transaction T. Originally, n_I was an intermediate node because one of n_I's children has the same support as n_I does; if T contains I but none of n_I's children who have the same support as n_I had before the addition, then n_I becomes a closed node because its new support is higher than the support of any of its children. However, n_I cannot change to an infrequent gateway node or an unpromising gateway node. First, n_I's support will not decrease because of adding T, so it cannot become infrequent. Second, if before adding T, $leftcheck(n_I) = false$, then $\nexists(j \prec i_{max}\ and\ j \notin I)$ s.t. j occurs in each transaction that I occurs; this statement will not change after we add T. Therefore, $leftcheck(n_I) = false$ after the addition.

Figure 5 gives a high-level description of the addition operation. Adding a new transaction to the sliding-window will trigger a call of *Addition* on $n_\emptyset$, the root of the CET.

Deleting a Transaction

In Figure 6, an old transaction T (tid 1) is deleted from the sliding-window. To delete a transaction, we also traverse the parts of the CET that is related to the deleted transaction. Most likely, n_I's node type will not change, in which case, we simply update n_I's *support* and tid_sum, and the cost is minimum. In the following, we discuss the impact of deletion in detail.

If n_I was an infrequent gateway node, obviously deletion does not change n_I's node type. If n_I was an unpromising gateway node, deletion may change n_I to infrequent but will not change n_I to promising, for the following reason. For an unpromising gateway node n_I, if before deletion,

Addition (n_I, I_{new}, $\mathcal{D}$, $minsup$)

1: **if** n_I is not relevant to the addition **then return**;
2: **foreach** child node $n_{I'}$ of n_I **do**
3: update $support$ and tid_sum of $n_{I'}$;
4: $\mathcal{F} \leftarrow \{n_{I'} | n_{I'}$ is newly frequent$\}$;
5: **foreach** child node $n_{I'}$ of n_I **do**
6: **if** $n_{I'}$ is infrequent **then**
7: (re)mark $n_{I'}$ an infrequent gateway node;
8: **else if** $leftcheck(n_{I'}) = true$ **then**
9: (re)mark $n_{I'}$ an unpromising gateway node;
10: **else if** $n_{I'}$ is a newly frequent node or $n_{I'}$ is a newly promising node **then**
11: Explore($n_{I'}$, $\mathcal{D}$, $minsup$);
12: **else**
13: **foreach** $n_K \in \mathcal{F}$ $s.t.$ $I' \prec K$ **do**
14: add $n_{I' \cup K}$ as a new child of $n_{I'}$;
15: Addition($n_{I'}$, I_{new}, $\mathcal{D}$, $minsup$);
16: **if** $n_{I'}$ was a closed node **then**
17: update $n_{I'}$'s entry in the hash table;
18: **else if** $\nexists$ a child node $n_{I''}$ of $n_{I'}$ $s.t.$ $support(n_{I''}) = support(n_{I'})$ **then**
19: mark $n_{I'}$ a closed node;
20: insert $n_{I'}$ into the hash table;
21: **return**;

Figure 5: The *Addition* Algorithm

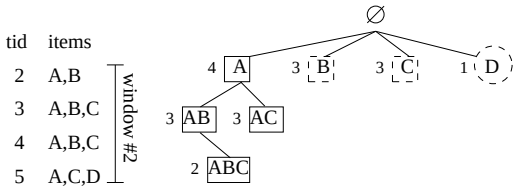


Figure 6: Deleting a Transaction

$leftcheck(n_I) = true$, then $\exists(j \prec i_{max}$ and $j \notin I)$ s.t. j occurs in each transaction that I occurs; this statement remains true when we delete a transaction.

If n_I was a frequent node, it may become infrequent because of a decrement of its support, in which case, all n_I's descendants are pruned and n_I becomes an infrequent gateway node. In addition, all of n_I's left siblings are updated by removing children obtained from joining with n_I. For example in Figure 6, when transaction T (tid 1) is removed from the window, node D becomes infrequent. We prune all descendants of node D, as well as AD and CD, which were obtained by joining A and C with D, respectively.

If n_I was a promising node, it may become unpromising because of the deletion, as happens to node C in Figure 6. Therefore, if originally n_I was neither infrequent nor unpromising, then we have to do the *leftcheck* on n_I. For a node n_I to change to unpromising because of a deletion, n_I must be contained in the deleted transaction. Therefore n_I will be visited by the traversal and we will not miss it.

If n_I was a closed node, it may become non-closed. To demonstrate this, we delete another transaction T (tid 2) from the sliding-window. Figure 7 shows this example where previously closed node n_I (e.g. A and AB) become non-closed because of the deletion. This can be determined by looking at the supports of the children of n_I after visiting them. If a previously closed node that is included in the deleted transaction remains closed after the deletion, we still need to update its entry in the hash table: its $support$ is decreased by 1 and its tid_sum is decreased by subtracting the tid of the deleted transaction.

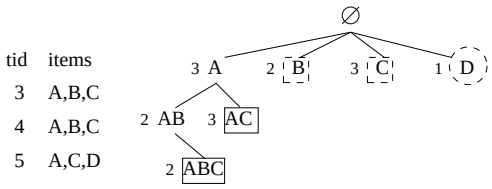


Figure 7: Another Deletion

From the above discussion we derive the following property for the deletion operation on a CET.

Property 5. *Deleting an old transaction will not change a node in the CET from non-closed to closed, and therefore it will not increase the number of closed itemsets in the sliding-window.*

Proof. If an itemset I was originally non-closed, then before the deletion, $\exists j \notin I$ $s.t.$ j occurs in each transaction that I occurs. Obviously, this fact will not be changed due to deleting a transaction. So I will still be non-closed after the deletion. □

Figure 8 gives a high-level description of the deletion operation. Some details are skipped in the description. For example, when pruning a branch from the CET, all the closed frequent itemsets in the branch should be removed from the hash table.

Deletion (n_I, I_{old}, $minsup$)

1: **if** n_I is not relevant to the deletion **then return**;
2: **foreach** child node $n_{I'}$ of n_I **do**
3: update $support$ and tid_sum of $n_{I'}$;
4: $\mathcal{F} \leftarrow \{n_{I'} | n_{I'}$ is newly infrequent$\}$;
5: **foreach** child node $n_{I'}$ of n_I **do**
6: **if** $n_{I'}$ was infrequent or unpromising **then**
7: **continue**;
8: **else if** $n_{I'}$ is newly infrequent **then**
9: prune $n_{I'}$'s descendants from CET;
10: mark $n_{I'}$ an infrequent gateway node;
11: **else if** $leftcheck(n_{I'}) = true$ **then**
12: prune $n_{I'}$'s descendants from CET;
13: mark $n_{I'}$ an unpromising gateway node;
14: **else**
15: **foreach** $n_K \in \mathcal{F}$ $s.t.$ $I' \prec K$ **do**
16: prune $n_{I' \cup K}$ from the children of $n_{I'}$;
17: Deletion($n_{I'}$, I_{old}, $minsup$);
18: **if** $n_{I'}$ was closed and $\exists$ a child $n_{I''}$ of $n_{I'}$ $s.t.$ $support(n_{I''}) = support(n_{I'})$ **then**
19: mark $n_{I'}$ an intermediate node;
20: remove $n_{I'}$ from the hash table;
21: **else if** $n_{I'}$ was a closed node **then**
22: update $n_{I'}$'s entry in the hash table;
23: **return**;

Figure 8: The *Deletion* Algorithm

Discussion

For addition, *Explore* is the most time consuming operation, because it scans the transactions in the sliding-window. However, as will be demonstrated in the experiments, the number of such invocations is very small, as most insertions will not change node types. In addition, the new branches grown by calling *Explore* are usually very small subsets of the whole CET, therefore such incremental growing takes much less time than regenerating the whole CET. On the other hand, deletion only involves related nodes in the CET, and does not scan transactions in the sliding-window. Therefore, its time complexity is at most linear to the number of nodes. Usually it is faster to perform a deletion than an addition.

It is easy to show that if a node n_I changes node type (frequent/infrequent and promising/unpromising), then I is in the added or deleted transaction and therefore n_I is guaranteed to be visited during the update. Consequently, our algorithm will correctly maintain the current close frequent itemsets after any of the two operations. Furthermore, if n_I remains closed after an addition or a deletion and I is contained in the added/deleted transaction, then its position in the hash table is changed because its $support$ and tid_sum are changed. To make the update, we delete the itemset from the hash table and re-insert it back to the hash table based on the new key value. However, such an update has constant time complexity.

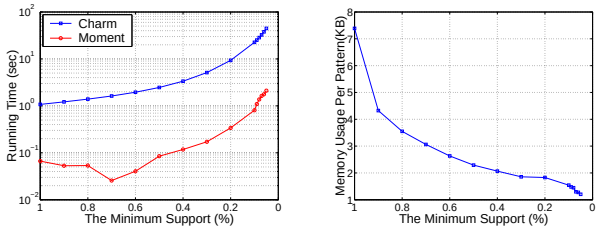
In our discussion so far, we used sliding-windows of fixed size. However, the two operations–*addition* and *deletion*–are independent of each other. Therefore, if needed, the size for the sliding-window can grow or shrink without affecting the correctness of our algorithm. In addition, our algorithm does not restrict a deletion to happen at the end of the window: at a given time, any transaction in the sliding-window can be removed. For example, if when removing a transaction, the transaction to be removed is picked following a random scheme: e.g., the newer transactions have lower probability of being removed than the older ones, then our algorithm can implement a sliding-window with *soft* boundary, i.e., the more recent the transaction, the higher chance it will remain in the sliding-window.

In addition, so far our algorithm only handles one transaction in one update. In reality, there are situations in which data are bursty and multiple transactions need to be added and deleted during one update. However, it is not difficult to adapt our algorithm to handle multiple transactions in one update. Originally, for an addition or a deletion, we traverse the CET with the single added or deleted transaction; if an update contains a batch of transactions, we can still traverse the CET in the same fashion using the batch of transactions and project out unrelated transactions along the traversal.

4 Experimental Results

We performed extensive experiments to evaluate the performance of Moment. We use Charm, a state-of-the-art algorithm proposed by Zaki et al [16], as the baseline algorithm to generate closed frequent itemsets without using incremental updates. All experiments were done on a 2GHz Intel Pentium IV PC with 2GB main memory, running RedHat Linux 7.3 operating system. All algorithms are implemented in C++ and compiled using the g++ 2.96 compiler.

T20I4D100K The first dataset is generated using the synthetic data generator described by Agrawal et al in [2]. Data from this generator mimics transactions from retail stores. We have adopted the commonly used parameters: the size of the sliding window N is 100,000, the average size of transactions T is 20, the average size of the maximal potentially frequent itemsets I is 4. We call this dataset $T20I4D100K$. We generated 100,100 transactions and we report the average performance over 100 consecutive sliding windows (each with size 100,000).



(a) *(b)*

Figure 9: Running Time and Memory Usage for $T20I4D100K$

Figure 9 gives the result on $T20I4D100K$. Figure 9(a) shows the average running time for Moment and for Charm over the 100 sliding windows under different minimum supports. As can be seen from the figure, as minimum support decreases, because the number of closed frequent itemsets increases, the running time for both algorithms grows. However, the response time of Moment is faster than that of Charm by more than an order of magnitude under all the minimum supports.

$minsup$ (in %)	closed itemset #	CET node #	CET node # per closed	changed node #	new node #
1.0	4097	148450	36.2	0.14	6.28
0.9	5341	168834	31.6	0.06	0.92
0.8	6581	187076	28.4	0.13	0.64
0.7	8220	212774	25.9	0.09	0.35
0.6	10270	249549	24.3	0.08	1.30
0.5	12655	309575	24.5	0.10	2.58
0.4	16683	433595	26.0	0.18	4.20
0.3	24907	722645	29.0	0.26	9.52
0.2	45353	1614726	35.6	0.67	27.23
0.1	172396	5955425	34.5	3.41	88.69
0.05	722261	19691999	27.3	14.75	286.34
0.03	1704558	45246906	26.5	41.07	646.84

Table 1: Data Characteristics for $T20I4D100K$

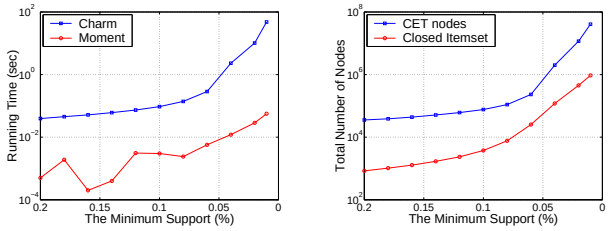
Table 1 shows the characteristics of the data and the mining results. All reported data are average values taken over the 100 sliding windows. The first three columns show the minimum support, the number of closed itemsets, and the number of nodes in the CET. From the table we can see that as the minimum support decreases, the number of closed itemsets grows rapidly. So does the number of nodes in the CET. However, the ratio between the number of nodes in the CET and the number of closed itemsets (which is reported in column 4) remains approximately the same. This implies that the sizes of the CET is linear in the number of closed frequent itemsets.

Because an addition may trigger a call for *Explore()* which is expensive, we study how many nodes change their status from infrequent/unpromising to frequent/promising (column 5) and how many new nodes are created due to the addition (column 6). From the data we can see that during an addition, the average number of nodes that change

from infrequent to frequent or from unpromising to promising in the CET is very small relative to the total number of nodes in the CET. Similarly, the number of new nodes created due to an addition is also very small. These results verify the postulation behind our algorithm: that an update usually only affects the status of a very small portion of the CET and the new branches grown because of an update is usually a very small subset of the CET.

We also have studied the memory usage for Moment. As shown in Figure 9(b), the average memory usage per closed frequent itemset actually decreases as the minimum support decreases. This suggests that as the CET becomes larger, it becomes more memory-efficient in terms of memory usage per closed frequent itemset.

BMS-WebView-1 The second dataset we have used is BMS-WebView-1, which is a real dataset that contains a few months of clickstream data from an e-commerce web sites. This dataset was used in KDDCUP 2000 [17]. There are 59,602 transactions in the dataset. We set the sliding window size N to be 50,000 and do experiment on 100 consecutive sliding windows. Other parameters are: the number of distinct items is 497, the maximal transaction size is 267, and the average transaction size is 2.5.



(a) *(b)*

Figure 10: Performance for BMS-WebView-1

Figure 10(a) shows the running time for Moment and Charm. From the figure we can see that because the average transaction size of this data set (2.5) is smaller than that of the previous synthetic dataset (20), the relative performance of Moment is even better–it outperforms Charm by 1 to 2 orders of magnitudes. Figure 10(b) shows the total number of nodes in the CET and the total number of closed itemsets. As can be seen, although both grow exponentially as the minimum support decreases, the relative ratio between the two remains approximately the same, which suggests that for real data, the CET size is also linear in the number of closed frequent itemsets.

5 Conclusion

In this paper we propose a novel algorithm, Moment, to discover and maintain all closed frequent itemsets in a sliding window that contains the most recent samples in a data stream. In the Moment algorithm, an efficient in-memory data structure, the closed enumeration tree (CET), is used to record all closed frequent itemsets in the current sliding window. In addition, CET also monitors the itemsets that form the boundary between closed frequent itemsets and the rest of the itemsets. We have also developed efficient algorithms to incrementally update the CET when newly-arrived transactions change the content of the sliding window. Experimental studies show that the Moment algorithm outperforms a state-of-the-art algorithm that mines closed frequent itemsets without using incremental updates. In addition, the memory usage of the Moment algorithm is shown to be linear in the number of closed frequent itemsets in the sliding window.

Acknowledgement Thanks to Professor Mohammed J. Zaki at the Rensselaer Polytechnic Institute for providing us the Charm source code.

References

[1] R. C. Agarwal, C. C. Aggarwal, and V. V. V. Prasad. A tree projection algorithm for generation of frequent item sets. *Journal of Parallel and Distributed Computing*, 61(3):350–371, 2001.

[2] R. Agrawal and R. Srikant. Fast algorithms for mining association rules. In *Proc. of the 20th Intl. Conf. on Very Large Databases (VLDB'94)*, 1994.

[3] T. Asai, H. Arimura, K. Abe, S. Kawasoe, and S. Arikawa. Online algorithms for mining semi-structured data stream. In *Proc. 2002 Int. Conf. on Data Mining (ICDM'02)*, 2002.

[4] R. J. Bayardo, Jr. Efficiently mining long patterns from databases. In *Proceedings of the ACM SIGMOD*, 1998.

[5] J. H. Chang and W. S. Lee. Finding recent frequent itemsets adaptively over online data streams. In *Proc. of the 2003 Int. Conf. Knowledge Discovery and Data Mining (SIGKDD'03)*, 2003.

[6] M. Charikar, K. Chen, and M. Farach-Colton. Finding frequent items in data streams. In *Proc. of the 29th Int'l Colloquium on Automata, Languages and Programming*, 2002.

[7] D. W. Cheung, J. Han, V. Ng, and C. Y. Wong. Maintenance of discovered association rules in large databases: An incremental updating technique. In *Proceedings of the Twelfth International Conference on Data Engineering*, 1996.

[8] D. W. Cheung, S. D. Lee, and B. Kao. A general incremental technique for maintaining discovered association rules. In *Proceedings of the Fifth International Conference on Database Systems for Advanced Applications (DASFAA)*, 1997.

[9] C. Giannella, J. Han, E. Robertson, and C. Liu. Mining frequent itemsets over arbitrary time intervals in data streams. Technical Report tr587, Indiana University, 2003.

[10] K. Gouda and M. J. Zaki. Efficiently mining maximal frequent itemsets. In *Proceedings of the 2001 IEEE Int'l Conf. on Data Mining*, 2001.

[11] C. Hidber. Online association rule mining. In *Proc. of the ACM SIGMOD int'l conf. on Management of data*, 1999.

[12] C. Lee, C. Lin, and M. Chen. Sliding-window filtering: an efficient algorithm for incremental mining. In *Proc. of the int'l conf. on Info. and knowledge management*, 2001.

[13] G. Manku and R. Motwani. Approximate frequency counts over data streams. In *Proceedings of the 28th International Conference on Very Large Data Bases*, 2002.

[14] J. Wang, J. Han, and J. Pei. Closet+: searching for the best strategies for mining frequent closed itemsets. In *Proc. of the 2003 Int. Conf. Knowledge Discovery and Data Mining (SIGKDD'03)*, 2003.

[15] M. J. Zaki. Fast vertical mining using diffsets. In *Proc. of the 2003 Int. Conf. Knowledge Discovery and Data Mining (SIGKDD'03)*, 2003.

[16] M. J. Zaki and C. Hsiao. Charm: An efficient algorithm for closed itemset mining. In *2nd SIAM Int'l Conf. on Data Mining*, 2002.

[17] Z. Zheng, R. Kohavi, and L. Mason. Real world performance of association rule algorithms. In *Proc. of the 2001 Int. Conf. Knowledge Discovery and Data Mining (SIGKDD'01)*, 2001.