RC23312 (W0408-155) August 30, 2004
Computer Science

IBM Research Report

Catch the Moment: Maintaining Closed Frequent Itemsets over a Data Stream Sliding Window

Yun Chi*, Haixun Wang, Philip S. Yu, Richard R. Muntz*
IBM Research Division
Thomas J. Watson Research Center
P.O. Box 704
Yorktown Heights, NY 10598

*Department of Computer Science
University of California at Los Angeles
Los Angeles, CA 90095

Catch the Moment: Maintaining Closed Frequent Itemsets over a Data Stream Sliding Window

Yun Chi*
Department of Computer Science
University of California, Los Angeles
Los Angeles, CA 90095
ychi@cs.ucla.edu

Haixun Wang
IBM Thomas J. Watson Research Center
Hawthorne, NY 10532
haixun@us.ibm.com

Philip S. Yu
IBM Thomas J. Watson Research Center
Hawthorne, NY 10532
psyu@us.ibm.com

Richard R. Muntz*
Department of Computer Science
University of California, Los Angeles
Los Angeles, CA 90095
muntz@cs.ucla.edu

Abstract

This paper considers the problem of mining closed frequent itemsets over a data stream 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 contain a boundary between closed frequent itemsets and the rest of the itemsets. Concept 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 a representative algorithm for the sate-of-the-art approaches.

keywords: data streams, sliding window, closed frequent itemset, incremental learning

*The work of these two authors was partly supported by NSF under Grant Nos. 0086116, 0085773, and 9817773. In addition, the work of the first author was partly supported by an IBM summer internship.

1 Introduction

1.1 Motivation

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 data sets 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.

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 concept 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 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. Concept 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 relatively stable, and even if some does, the boundary movement is local, the cost of mining closed frequent itemsets is dramatically reduced.

1.2 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. (4) We have done extensive experimental studies to evaluate the performance of the proposed new algorithm. Experiments show that for mining closed frequent itemsets in data streams, Moment has significant performance advantage over a representative algorithm for the state-of-the-art approaches.

1.3 Related Work

Mining frequent itemsets from data streams has been investigated by many researchers. Manku et al [14] 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. Teng et al [15] presented an algorithms, FTP-DS, that mines frequent temporal patterns from data streams of itemsets. 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

[1] Maintaining Closed Frequent Itemsets by Incremental Updates

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 [12] was used and therefore the results are inexact. In all these studies, approximate algorithms were adopted. In contrast, our algorithm is an exact one. On the other hand, we can also assume that an approximation step has been implemented through the sampling scheme and our exact 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 [13] 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., [17], there are usually much fewer closed frequent itemsets compared to the total number of frequent itemsets.

The rest of the paper is organized as follows. In section 2, we give necessary background in frequent itemset mining. In section 3, we describe in detail our Moment algorithm. In section 4, we give experimental results. We give conclusion in section 5.

2 Problem Statement

2.1 Preliminaries

Given a set of items Σ, a database $\mathcal{D}$ wherein each transaction is a subset of Σ, and a threshold f called the *minimum frequency*, $0 < f \leq 1$, the frequent itemset mining problem is to find all itemsets that occur in at least $f|\mathcal{D}|$ transactions. For an itemset I, we call the number of transactions in which I occurs the *support* of I. In addition, we define the *minimum support* (*minsup*) s as $s = f|\mathcal{D}|$.

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 $s = 2$, 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 $\mathcal{D}$.

2.2 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 usually 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 the itemset is a maximal frequent itemset. Thus, mining only maximal frequent itemsets loses information.

In the second solution (e.g., [16], [17]), only *closed* frequent itemsets are discovered. An itemset is closed if none of its proper supersets has the same support as it has. Usually, 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

$$\begin{aligned}\mathcal{C} &= \{(C,3),(AB,3),(ABC,2)\}\\ \mathcal{M} &= \{(ABC,2)\}\end{aligned}$$

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

2.3 Problem Statement

The problem is to mine closed frequent itemsets in the most recent N transactions (or the most recent N *samples*) 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 $s = 2$.

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 all 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.

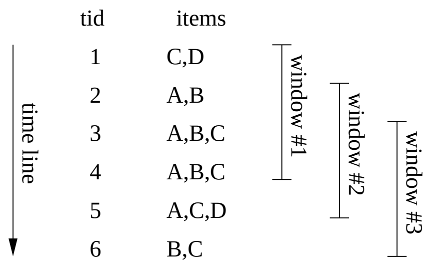


Figure 1: A Running Example

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 concept drifts in the stream are 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 does 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.

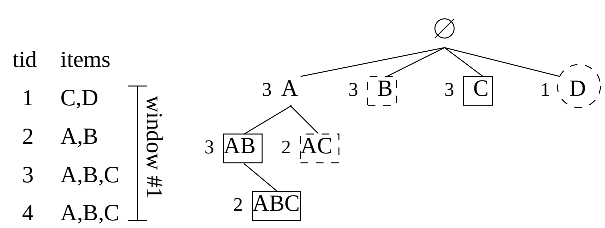


Figure 2: The Closed Enumeration Tree Corresponding to Window #1 (each node is labeled with its *support*)

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, ii) n_I's parent, n_J, is frequent, and iii) I is the result of joining I's parent, J, with one of J's frequent siblings. In addition, we define all nodes at the first level of the CET tree that correspond to infrequent items as infrequent gateway nodes. 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 B 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 the following 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, the CET '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 the CET 'remembers' the boundary where such pruning occurs by recording the unpromising gateway nodes.

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 $J \prec I$, then n_I is an unpromising gateway node, which means n_I cannot be an intermediate node. So we have $I \prec J$. However, if $I \prec J$, then n_J must be n_I's descendant because $J \supset I$. □

Property 3 shows that we cannot prune intermediate nodes in a CET.

3.3 Building the Closed Enumeration Tree

For each node n_I in a CET, we store the following information: i) the itemset I itself,[2] 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.

[2]In our implementation, we do not actually store the whole itemset I in node n_I—instead, we only store the last item in I. Because we always visit a node following a root-path of the CET, we can derive the itemset I by concatenating the items stored in the nodes along the root-path.

3.3.1 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 *tid*s. However, the set of *tid*s 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 $support(J) = support(I)$.

To save space, in the hash table entries, instead of the itemsets themselves, we store the pointers pointing to the corresponding nodes in the CET.

3.3.2 FP-Tree for Transactions

We store the transactions in the sliding window in an FP-tree, in order to reduce the memory footprint and to speed up exploration of the transactions. FP-tree was first introduced by Han et al for mining frequent itemsets without candidate generation [11]. In an FP-tree, each transaction is stored along a root-path; when transactions have a common prefix, the common part only needs to be stored once; a counter is used to record the number of times the common part is repeated. As demonstrated by Han et al, an FP-tree is a compact data structure that stores all necessary information for frequent itemsets mining and it is usually much smaller than the database itself. Figure 3 shows the FP-tree for the first sliding window. Note that the items are stored in an inverse lexicographical order among the root-path. This arrangement makes it easy to explore the FP-tree.

Our FP-tree is slightly different from the one we described above. First, we use the FP-tree to store all the transactions in the sliding window, so we do not prune infrequent items. Second, in addition to the head table in traditional FP-trees (which is used to record the starting pointers to each item), we also maintain another table, the tid table. In the tid table, for each tid (transaction id), there is a pointer pointing to a node in the FP-tree, which we call the node the *tail* of the transaction; the path from the tail to the root of the FP-tree gives us the itemset corresponding to the given tid. By using the FP-tree with the tid table, we do not need the transactions anymore.

To add a transaction to the sliding window, we store the corresponding itemset in the FP-tree and insert a new entry at the end of the tid table, where the pointer in the new entry points to the tail of the new transaction in the FP-tree; to delete a transaction from the sliding window, we pop an entry from the front of the tid table, and use the pointer to locate in the

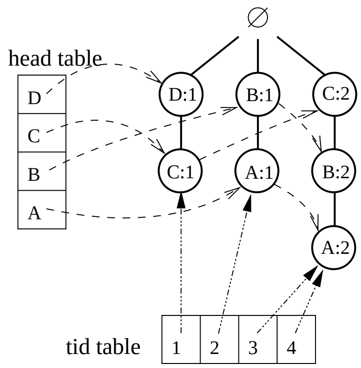


Figure 3: The FP-Tree for Transactions in the Sliding Window

FP-tree the tail of the transaction to be deleted. We then follow the path from the tail to the root of the FP-tree, and update the counters along the path. Notice that although the size of the tid table is the same as that of the sliding window (N), if we follow a first-in-first-out rule for updating the sliding window, most part of the tid table can be stored in disk, because we only update the front and the end of the tid table.

3.3.3 CET 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 4.

Explore is a depth-first procedure that visits itemsets in lexicographical order. For an itemset I, *Explore* consults the FP-tree to determine the *support* and *tid_sum* of I. In lines 1-2 of Figure 4, 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.

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

if $support(n_I) < minsup$ **then**
    mark n_I an infrequent gateway node;
else if $leftcheck(n_I) = true$ **then**
    mark n_I an unpromising gateway node;
else
    foreach frequent right sibling n_K of n_I **do**
        create a new child $n_{I \cup K}$ for n_I;
        compute *support* and *tid_sum* for $n_{I \cup K}$;
    foreach child $n_{I'}$ of n_I **do**
        Explore($n_{I'}$, $\mathcal{D}$, *minsup*);
    if $\exists$ a child $n_{I'}$ of n_I such that
        $support(n_{I'}) = support(n_I)$ **then**
        mark n_I an intermediate node;
    else
        mark n_I a closed node;
        insert n_I into the hash table;

Figure 4: The *Explore* Algorithm

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 is 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.[3]

3.4.1 Adding a Transaction

In Figure 5, 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.

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

[3]At the time that a new transaction is added to the sliding window, the window size is temporarily increased to $N + 1$; after that, deleting a transaction from the sliding window will change the window size back to N. Therefore in our algorithm, we assume that the minimum support (*minsup*) remained unchanged during the addition and the deletion.

tid	items
1	C,D
2	A,B
3	A,B,C
4	A,B,C
5	A,C,D

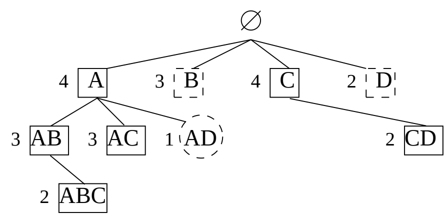


Figure 5: Adding a Transaction

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 5), 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 5, 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 5) 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 therefore 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.

```
Addition (n_I, I_new, D, minsup)
if n_I is not relevant to the addition then return;
foreach child node n_I' of n_I do
    update support and tid_sum of n_I';
F ← {n_I' | n_I' is newly frequent};
foreach child node n_I' of n_I do
    if n_I' is infrequent then
        (re)mark n_I' an infrequent gateway node;
    else if leftcheck(n_I') = true then
        (re)mark n_I' an unpromising gateway node;
    else if n_I' is a newly frequent node or
         n_I' is a newly promising node then
        Explore(n_I', D, minsup);
    else
        foreach n_K ∈ F s.t. I' ≺ K do
            add n_I'∪K as a new child of n_I';
        Addition(n_I', I_new, D, minsup);
        if n_I' was a closed node then
            update n_I''s entry in the hash table;
        else if ∄ a child node n_I'' of n_I' s.t.
            support(n_I'') = support(n_I') then
            mark n_I' a closed node;
            insert n_I' into the hash table;
return;
```

Figure 6: The *Addition* Algorithm

Figure 6 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.

From the above discussion and from the *Addition* algorithm shown in Figure 6, we can easily derive the following property of *Addition*:

Property 5. *The Addition algorithm will not decrease the number of nodes in a CET.*

3.4.2 Deleting a Transaction

In Figure 7, 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.

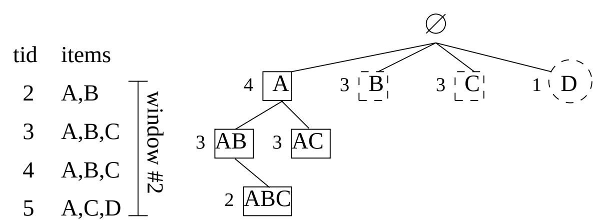


Figure 7: Deleting a Transaction

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, $leftcheck(n_I) = true$, 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 7, 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, for the following reason. If before the deletion, $\exists(j \prec i_{max} \; and \; j \notin I)$ s.t. j occurs in each transaction that I occurs, except only for the transaction to be deleted, then after deleting the transaction, I becomes unpromising. This happens to node C in Figure 7. Therefore, if originally n_I was neither infrequent nor unpromising, then we have to do the *leftcheck* on n_I. From the above discussion we can also see that 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 8 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.

tid	items
3	A,B,C
4	A,B,C
5	A,C,D

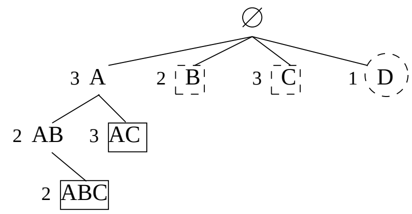


Figure 8: Another Deletion

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

Property 6. *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 9 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.

From the above discussion and from the *Deletion* algorithm shown in Figure 9, we can easily derive the following property of *Deletion*:

Property 7. *The Deletion algorithm will not increase the number of nodes in a CET.*

3.5 Discussion

In the addition algorithm, *Explore* is the most time consuming operation, because it scans the transactions stored in the FP-tree. 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 stored in the FP-tree. 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

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

if n_I is not relevant to the deletion **then return**;
foreach child node $n_{I'}$ of n_I **do**
    update *support* and *tid_sum* of $n_{I'}$;
$\mathcal{F} \leftarrow \{n_{I'} | n_{I'}$ is newly infrequent$\}$;
foreach child node $n_{I'}$ of n_I **do**
    if $n_{I'}$ was infrequent or unpromising **then**
        continue;
    else if $n_{I'}$ is newly infrequent **then**
        prune $n_{I'}$'s descendants from CET;
        mark $n_{I'}$ an infrequent gateway node;
    else if $leftcheck(n_{I'}) = true$ **then**
        prune $n_{I'}$'s descendants from CET;
        mark $n_{I'}$ an unpromising gateway node;
    else
        foreach $n_K \in \mathcal{F}$ $s.t.$ $I' \prec K$ **do**
            prune $n_{I' \cup K}$ from the children of $n_{I'}$;
        Deletion($n_{I'}$, I_{old}, $minsup$);
        if $n_{I'}$ was closed and $\exists$ a child $n_{I''}$ of $n_{I'}$
            $s.t.$ $support(n_{I''}) = support(n_{I'})$ **then**
            mark $n_{I'}$ an intermediate node;
            remove $n_{I'}$ from the hash table;
        else if $n_{I'}$ was a closed node **then**
            update $n_{I'}$'s entry in the hash table;
return;

Figure 9: The *Deletion* Algorithm

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 amortized 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 [17], as the baseline algorithm to generate closed frequent itemsets without using incremental updates. We have used the latest version of Charm. As demonstrated in many studies (e.g., [17, 18]), among the algorithms that mine closed frequent itemsets, Charm has best performance for various data sets. All our experiments were done on a 2GHz Intel Pentium IV PC with 2GB main memory, running RedHat Linux 7.3 operating system. Both Charm and Moment are implemented in C++ and compiled using the g++ 2.96 compiler with -O3 optimization level.

For the performance study, we have used 3 synthetic data sets and 4 real-world data sets. The data characteristics for all the data sets are summarized in Table 1. We will describe each data set in detail in the following sections.

Database	# Items	Avg. Length	Max Length	# Records	Window Size
T20I4N10K-100K	1,000	20	44	100,000	10K-100K
T40I10N10K-100K	1,000	40	80	100,000	10K-100K
T20I10N100K	1,000	20	49	100,000	100K
BMS-WebView-1	497	2.5	267	59,602	50K
BMS-WebView-2	3,340	4.6	161	77,512	50K
BMS-POS	1,657	6.5	164	515,597	500K
Mushroom	120	23	23	8,124	8K

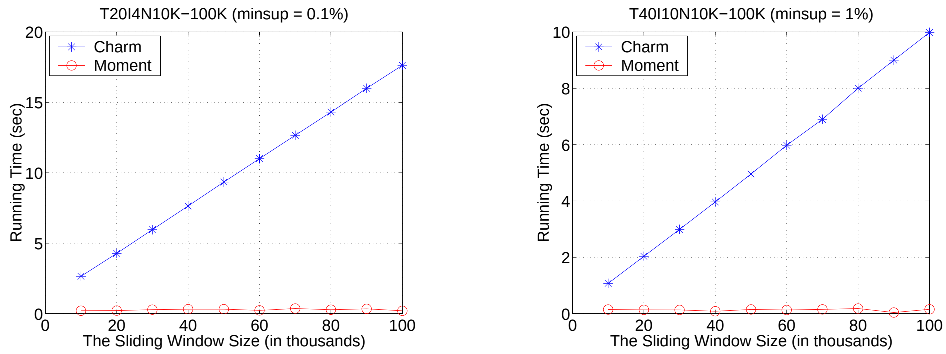
Table 1: Data Characteristics

4.1 Synthetic Data Sets

The synthetic data sets are generated using the synthetic data generator developed by Agrawal et al [2]. Data from this generator mimics transactions from retail stores. Here are some of the parameters that we have controlled: the size of the sliding window N, the average size of transactions T, the average size of the maximal potentially frequent itemsets I.

Performance under Different Sliding Window Sizes

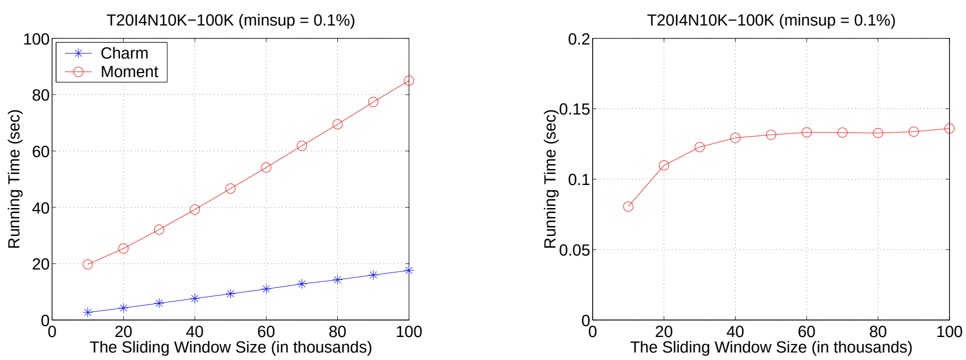
In the first experiment, we compare Moment and Charm under different sliding window sizes. For this study, we generated two data sets: in the first one, T20I4N10K-100K, we have set the parameters as $T = 20$, $I = 4$; in the second one, T40I10N10K-100K, we have set the parameters as $T = 40$, $I = 10$. In both data sets, we let the sliding window size N grow from 10K to 100K. For both algorithms, we report the average running time over 100 sliding windows.



(a) (b)

Figure 10: Running Time vs. Sliding Window Size

As shown in Figure 10, as the sliding window size increases, the time to generate all closed frequent itemsets for Charm grows in a linear fashion. In contrast, the running time of Moment does not change too much with the sliding window size. This result demonstrates an advantage of the Moment algorithm: because of its *incremental* updating fashion, it is not sensitive to the sliding window size.



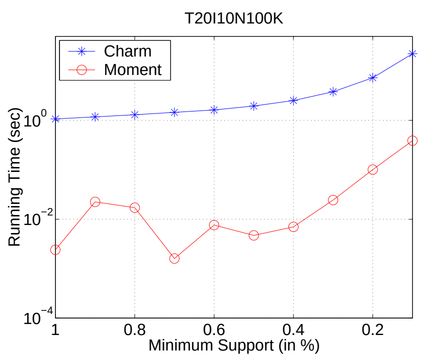
(a) (b)

Figure 11: Bulk Loading vs. Incremental Loading

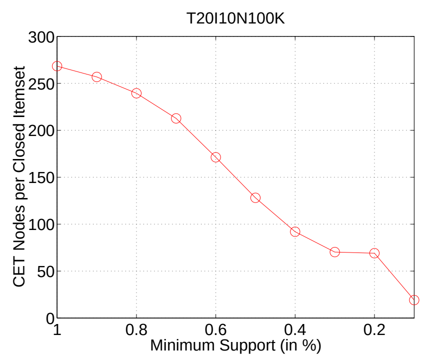
In Figure 11(a) we compare the time for Moment to bulk-load the first sliding window (by calling *Explore()*) and the time for Charm to mine the closed frequent itemsets in the first sliding window. As can be seen from the figure, for getting the results in the first sliding window, Charm is faster by 5 to 10 times. There are several reasons for this result: first, we have used the latest version of Charm, which is heavily optimized for large set operations (e.g., by using the *diffset* techniques); second, Moment has extra data structures to maintain (e.g., creating the CET nodes, update their support and tid_sum, etc.). However, we argue that this comparison is not fair–Moment is an incremental algorithm and the bulk-loading should not be used at all. To show this point, we have done the following experiment: originally, the sliding window is empty, then transactions are added one by one until the sliding window is full. We have done this experiment under different sliding window sizes (10K to 100K), and in Figure 11(b) we report the average time for adding each new transaction, where the time includes the time for updating the FP-tree and that for updating the CET. As we can see from the figure, the average time per transaction is very small and it is not very sensitive to the window size.

Performance under Different Minimum Support

In the second experiment, we compare the performance of Moment and Charm under different minimum supports. The data set we have used, $T20I10N100K$, has the following parameters: $T = 20$, $I = 10$, N=100K. We let the minimum support decrease from 1% to 0.1%.



(a)



(b)

Figure 12: Performance on T20I10N100K

Figure 12(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.

Table 2 shows the number of closed itemsets under different minimum supports. In addition, in the table we show some static and dynamic statistics about the CET data structure. 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

minsup (in %)	closed itemset #	CET node #	changed node #	added node #	deleted node #
1.0	721	193526	0.01	6.20	12.39
0.9	821	210673	0.01	6.46	0.00
0.8	967	231376	0.04	13.94	6.76
0.7	1211	257498	0.02	0.26	0.00
0.6	1649	282330	0.04	0.67	0.27
0.5	2544	325834	0.05	1.62	0.16
0.4	4468	410629	0.07	6.40	3.17
0.3	9176	644622	0.16	5.53	6.67
0.2	22446	1549740	0.59	30.06	48.64
0.1	386075	7394420	38.70	147.67	115.64

Table 2: Data Characteristics for $T20I10N100K$

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 shown in Figure 12(b)) actually decreases as the minimum support decreases. This implies that as the sizes of the CET grows larger, it becomes more efficient and the size of the CET is bounded by the number of closed itemsets times a constant number.

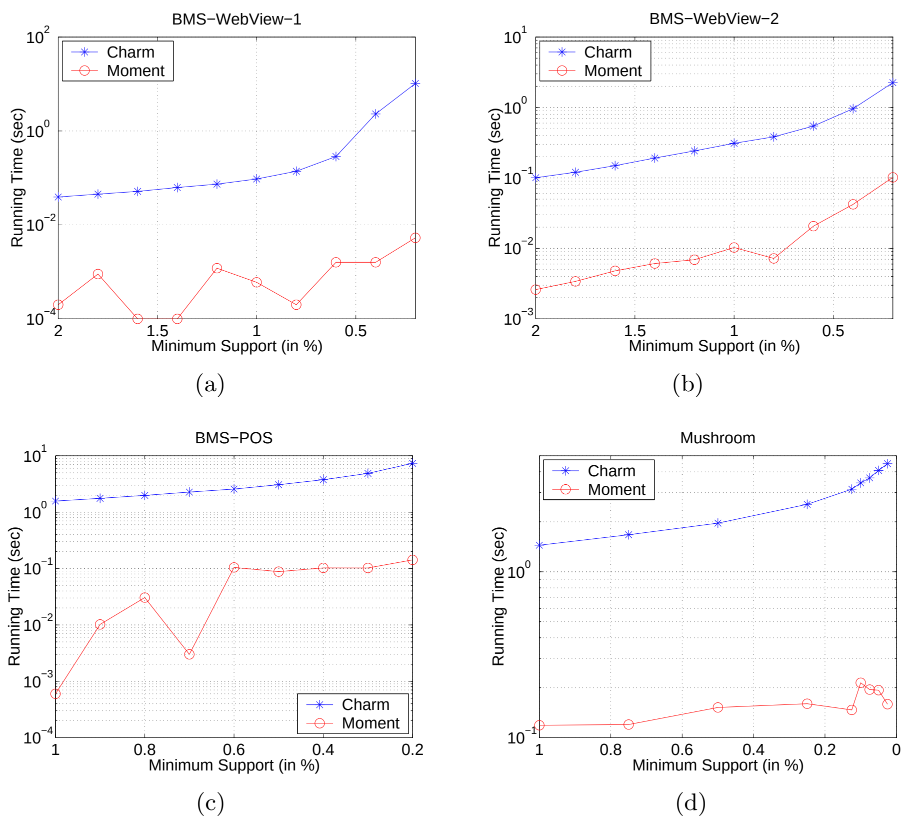
Because an addition may trigger a call for *Explore()* which is expensive, we study how many nodes change their status from infrequent/unpromising to frequent/promising (column 4) and how many new nodes are created due to the addition (column 5). From the data we can see that during an addition, the 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. In addition, we have reported the average number of CET nodes deleted due to a deletion (column 6). It can be seen that this number is in about the same scale as that of added nodes. However, because a deletion does not query the FP-tree and does not grow the CET, it is a relatively inexpensive operation and therefore will not affect the performance too much.

4.2 Read-World Data Sets

We have used 4 real-world data sets to study the performance of Moment. The first 3 data sets were used for KDDCUP 2000 [18]. Among these 3 data sets, the first two, BMS-WebView-1 and BMS-WebView-2, record several months of clickstream data from two e-commerce web sites; the third one, BMS-POS, contains several years of point-of-sale data from a large electronics retailer [18]. Our forth real-world data set is the Mushroom data set used by Zaki et al [17] and it belongs to the family of "dense" data, where there exists strong correlation among transactions. The data characteristics for the 4 data sets are summarized in Table 1.

Running Time Performance

Figures 13(a), 13(b), and 13(c) show the average running time of Moment and Charm for the BMS-WebView-1, the BMS-WebView-2, and the BMS-POS data sets, under different minimum supports. From the figure we can see that Moment outperforms Charm by one or two orders of magnitude for all the three data sets. Similar results are obtained for the Mushroom data set, as given in Figure 13(d). These results show that Moment also has good performance in real-world data sets of various characteristics (sparse or dense).



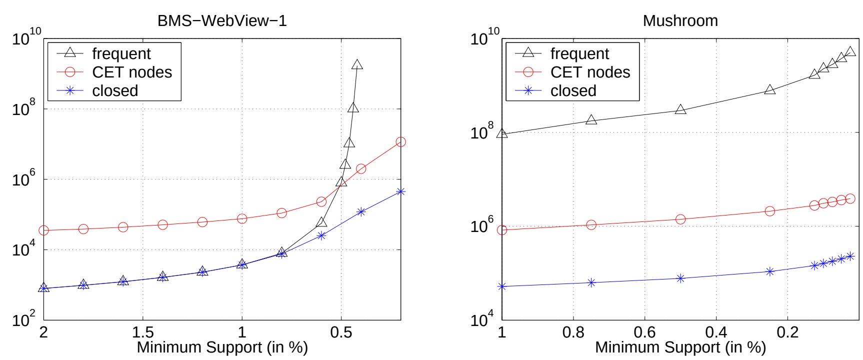
(a) (b) (c) (d)

Figure 13: Performance for Real-World Data Sets

The Number of CET Nodes

One design consideration for Moment is to maintain in CET only information related to *closed* frequent itemsets, instead of *all* frequent itemsets. In this section, we use real data sets to justify this consideration.

We show the total number of frequent itemsets, the number of closed frequent itemsets, and the number of CET nodes for two data sets. Figure 14(a) shows these numbers for the BMS-WebView-1 data set and Figure 14(b) shows these numbers for the Mushroom data set under different minimum supports. As can be seen from Figure 14(a), because BMS-WebView-1 is a relatively sparse data set, under high minimum supports, the number of closed frequent itemsets and that of all frequent itemsets do not have much difference; however, when the minimum support decreases further, as some large itemsets become frequent, the total number of frequent itemsets blows up dramatically; in contrast, the number of CET nodes still keeps a constant ratio relative to the number of closed frequent itemsets. The Mushroom data set, in comparison, is relatively dense, and therefore, even at high minimum support, there are much more frequent itemsets than closed frequent itemsets. As shown in Figure 14(b), although the number of CET nodes is about one order of magnitude more than the number of closed frequent itemsets, it is at least two orders of magnitude fewer than the total number of frequent itemsets.



(a) (b)

Figure 14: The Number of Closed Itemsets, CET Nodes, and 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 running time of the Moment algorithm is not sensitive to the sliding window size and Moment outperforms a state-of-the-art algorithm that mines closed frequent itemsets without using incremental updates. In addition, the number of CET nodes is shown to be proportional to that of closed frequent itemsets. Under low minimum supports or when applied to dense data sets, CET has much fewer number of nodes than the total number of frequent itemsets.

Acknowledgement We thank 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] J. Han, J. Pei, and Y. Yin. Mining frequent patterns without candidate generation. In *2000 ACM SIGMOD Intl. Conference on Management of Data*, May 2000.

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

[13] 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.

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

[15] W-G Teng, M-S Chen, and P. S. Yu. A regression-based temporal pattern mining scheme for data streams. In *Proceedings of 29th International Conference on Very Large Data Bases (VLDB'03)*, 2003.

[16] 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.

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

[18] 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.