ADMiner: An Incremental Data Mining Approach Using a Compressed FP-tree

Chien-Min Lin, Yu-Lung Hsieh, Kuo-Cheng Yin
Dept. of Information Engineering and Computer Science, Feng Chia University, Taichung, Taiwan
Email: {cm1098, yuhlong.hsieh, inn0206}@gmail.com

Ming-Chuan Hung
Dept. of Industrial Engineering and Systems Management, Feng Chia University, Taichung, Taiwan
Email: mchong@fcu.edu.tw

Don-Lin Yang*
Dept. of Information Engineering and Computer Science, Feng Chia University, Taichung, Taiwan
Email: dlyang.tw@gmail.com
*corresponding author

Abstract—**In real world applications, most transaction databases are often large and constantly updated. Current data mining algorithms face the problem of processing a large number of transactions in dynamic environments. Since memory space is limited, it is critical to be able to use available storage efficiently and to process more transactions. In this paper, we propose an improved data structure of a compressed FP-tree to mine frequent itemsets with greater efficiency. Use of our method can minimize the I/O overhead, and, more importantly, it can also perform incremental mining without rescanning the original database. Our experimental results show that the method we propose not only requires less memory, but also performs incremental mining more efficiently.**

Index Terms—**Association rule, Frequent pattern, Compressed FP-tree, Incremental data mining**

I. Introduction

In data mining research, transaction databases are often large and constantly updated. This causes problems for many data mining algorithms [1-2] since the available memory space is limited. Apart from the storage problem, processing the transaction database incrementally in an efficient manner is also a critical issue. Traditional data mining approaches rescan the whole database after updates are made, but it is not efficient to repeat the same process more than once. Therefore, it is necessary to develop data mining algorithms that can both process dynamic databases [3-6] using a better method and utilize memory space more efficiently.

Traditional data mining algorithms generate many candidates to find frequent itemsets like the Apriori algorithm [1]. In addition to the problem of the large number of candidates, this algorithm also demands an efficient data structure to store frequent itemsets for further processing. Without needing to generate all candidates, *FP-growth* [2], fully utilizes the common path of the *FP-tree* structure to store potential frequent itemsets. However, this is not optimal for the incremental mining of dynamic databases. *FP-growth* scans the database and sorts the frequent items by their frequencies. Since the updated database likely causes the frequent item sequences to change, *FP-tree* must be rebuilt such that frequent itemsets can be identified accordingly.

Our challenge is to improve the storage of potential frequent itemsets for incremental mining while retaining the advantage of sharing the common path of *FP-tree*. In this paper, we propose a compact structure for storing potential frequent itemsets based on *FP-tree* to process item transactions in main memory.

In addition, transaction databases are updated regularly in real world situations. In most cases, *FP-growth* must rescan the updated database and rebuild *FP-tree*, due to the change in the support count of frequent 1-itemsets. To avoid the cost of repeatedly scanning the original database, we simply adjust, rather than completely rebuild, the tree after the updating process. Since we can compress the *FP-tree* structure to save storage space, we can build a compact version of *FP-tree* without specifying the minimum support. Therefore, our approach allows incremental mining of dynamic databases for any support threshold.

II. Related Work

Association rule mining [7-10] is one of the most popular data mining techniques. It finds interesting information among a large set of data items which appear together. For example, the rules found from a sales database are useful for the marketing manager's decision making. The application of these association rules is in market basket analysis. This mining helps the decision maker analyze customers' purchasing habits by discovering association among items.

Han et al. proposed the *FP-growth* [2] algorithm to mine frequent itemsets without generating candidate itemsets. *FP-growth* uses a tree structure, *FP-tree*, to solve the problem of Apriori for efficiently mining association rules. *FP-tree* captures the content of the transaction database and compresses all the transactions. It successfully avoids scanning the database many times.

In addition to *FP-tree*, a header table is used to traverse the tree and find frequent itemsets quickly. The header records frequent 1-itemsets in decreasing order of their frequencies. Each 1-itemset in the header table points to the corresponding node of *FP-tree*. If two nodes of *FP-tree* have the same item, a link will be generated between them. During construction an *FP-tree* needs to scan the transaction database twice.

The next part of the *FP-growth* algorithm is to mine frequent patterns using the constructed *FP-tree*. It traverses the nodes of frequent itemsets from the least frequent item to the root of the *FP-tree* by using the header table. Paths with the same prefix item in the *FP-tree* are used to construct the conditional *FP-tree*. Using the conditional *FP-tree*, the algorithm can generate frequent itemsets with the same prefix.

FIUT [11] was proposed to use a special frequent items ultrametric tree, called *FIU-tree*. The algorithm puts frequent k-itemsets into *FIU-tree*. These items are stored in the tree by lexicographical order to compress the items for more efficient use of memory space. This algorithm only scans the database twice. It can be divided into two phases. In the first phase, it scans the database to generate all frequent items and prune infrequent items. Next, frequent items are stored in lexicographical order. In the second phase, the algorithm constructs a frequent items ultrametric tree, i.e., *FIU-tree*, for mining frequent itemsets.

[12-14] use a compressed *FP-tree* approach to reduce the number of nodes. However, they cannot deal with transaction updates or the change in the support count threshold.

III. Proposed Method

In this section, we describe our proposed approach and introduce how it works. Simple examples will then be presented in the next section.

A. Basic Concept of Our Approach

Although traditional *FP-growth* employs the compact structure of *FP-tree* by taking advantage of the common prefix, it still cannot work for very large databases. To deal with the space problem, we try to further compress the nodes in the *FP-tree*. Therefore, we propose a compressed version of the *FP-tree*, called *CFP-tree*. Items with the same count value will be put into the same node.

In addition, *FP-growth* is not suitable for incremental data mining. Because an *FP-tree* is based on the set of frequent items, it needs to be re-built when the database updates change them. A more serious problem is the dynamic of the data mining application where the support thresholds vary constantly. Since in most cases a different support threshold would result a different set of frequent items, *FP-Tree* needs to re-scan the database to build a new *FP-tree*. Since FIUT suffers similar problems, no further discussion of this will be made. To enable incremental data mining with varying support thresholds, we build *FP-tree* with all the items in the database.

Thus, we propose a compressed version of the *FP-tree*, called *CFP-tree*, to improve the usage of memory space. Instead of re-building *FP-tree* after the database updates and/or support threshold changes, we dynamically adjust the *FP-tree* which is built with all the items in the database.

B. Our Proposed Algorithm

We propose an algorithm called ADMiner that uses an assemble-and-detach approach for mining incremental datasets. We define some terms and symbols as below:

Definition 1. Let $I=\{i_1,i_2,\ldots,i_m\}$ be a set of distinct items and $T=\{t_1,t_2,\ldots,t_n\}$ be a set of transaction identifiers. A database DB contains a set of transactions in the form of $<tid,itemset>$ where $tid\in T$ and $itemset\subseteq I$.

Definition 2. Let X be an itemset where $X\subseteq I$. We say that X is a frequent itemset (pattern) if the frequency of X appearing in the transactions of a DB is greater than or equal to a user specified threshold (min-sup).

Definition 3. Let i, j be two items with frequencies f_i and f_j, respectively. We say that $i < j$, if (1) $f_i < f_j$, or (2) $f_i = f_j$ and $i < j$ in alphabetical order.

We now introduce two major data structures: the Item-Frequency List (*IF-list*) and the Compressed FP-tree (*CFP-tree*) in ADMiner. The data structure *IF-list* consists of a pair of item and its frequency (*IF-pair*). The *IF-pairs* form an *IF-list* and are ordered as stated in Definition 3. The sequence of *IF-pairs* in an *IF-list* is unique when a path in the *CFP-tree* is constructed. The *CFP-tree* is used to store the items of every transaction in a database. The *CFP-tree* consists of Compressed FP-Nodes (*CFP-node*) that store all the items and their subsequence of the same support count. The formal definition is given below:

Definition 4. Let S be an *IF-list* in which every element is an *IF-pair* of the form $< i, f_i >$, where item $i \in I$ and frequency $f_i \geq 0$.

Definition 5. A *CFP-tree* consists of one or more *CFP-nodes*, and each node is of the form X: f_x where itemset $X \subseteq I$ and f_x is the frequency of X. The root of a *CFP-tree* is a null itemset with a frequency value of 0.

The examples of *IF-list* and *CFP-tree* will be shown later in TABLE and Figure 1 (c), respectively.

The ADMiner algorithm has two phases. The first phase is called the construction phase. In this phase, it constructs the *CFP-tree* by scanning the database twice. In the first database scan, the ADMiner creates an *IF-list* to record the information of items and their frequencies. In the second database scan, a *CFP-tree* is constructed by adding a new path or merging an existing path for every transaction in the database.

The second phase decomposes the *CFP-tree* as needed for adjustment. The order of item sequence will be changed, as well as the frequency of the items stored in the same node. In this phase, a node may split into two or more new nodes and the order of *IF-list* is likely to be changed. Necessary steps are required to make adjustments and obtain a new *CFP-tree*.

These two phases are described as follows:

Step1: Scan the original database DB to calculate the frequency of transactions containing the items $i_j \in I$, where $j = 1, 2, \ldots, m$.

After processing the last transaction, we have constructed the data structure *IF-list* which contains all of the items and their frequencies. The elements of *IF-list* are sorted by frequency in descending order, as described in *Definition 3*. Next, we start the second scan of DB.

Step2: Read a transaction at a time from DB and insert each item into the *CFP-tree*, working down to the end of the original database DB.

Substep2.1: Read a transaction to get the current sequence of items.

Substep2.2: Sort the items in the same order of the *IF-list*.

Substep2.3: Add the sorted items to the *CFP-tree*.

To add the sorted Items Sequence of the current Transaction (*IST*) to the *CFP-tree*, we check the Item Sequence of the Node (*ISN*) starting from the first leftmost child of the root node. The item sequence of the first transaction, *IST*(*tid* 1), will be simply inserted into the first child node of the root. There are five cases in the construction of the rest of the *CFP-tree* as follows:

Case1: *ISN*(*node i*) = *IST*(*tid j*), where *i* and *j* are indices for a node and a transaction, respectively.

(1). Increase the support count of node *i* by 1.

Case2: *ISN* (*node i*)$\subset$*IST* (*tid j*), where the first item of *ISN* = the first item of *IST*.

(1). Increase the support count of node *i* by 1.

(2). If *node i* is a leaf node, add a new child node containing the items of *IST* (*tid j*) – *ISN* (*node i*) and set the support count of the new node to 1.

(3). If *node i* is not a leaf node, set *IST* (*tid j*) = *IST* (*tid j*) – *ISN* (*node i*) and continue the construction process by checking with the children of *node i*.

Case3: *IST* (*tid j*) $\subset$ *ISN* (*node i*), where the first item of *ISN* = the first item of *IST*.

(1). Set *ISN* (*node i*) = *ISN* (*node i*) – *IST* (*tid j*) and increase the support count of *node i* by 1.

(2). If *node i* is a leaf node, add a new child node containing the items of *ISN* (*node i*) – *IST* (*tid j*) and set the same support count to the new node.

(3). If *node i* is not a leaf node, add a new child node containing the items of *ISN* (*node i*) – *IST* (*tid j*), set the same support count to the new node, and move all the child nodes of *node i* to become the child nodes of this new node.

Case4: *Node i* and *IST* (*tid j*) have at least the first *c* item(s) in common.

(1). Set *ISN* (*node i*) = *ISN* (*node i*) - all the common items.

(2). Add a new node containing all the common items as the parent node of *node i* and set the support count to be the support count of *node i* plus 1.

(3). Set *IST* (*tid j*) = *IST* (*tid j*) - all the common items.

(4). Add a new node containing the items of *IST* (*tid j*) as the child node of *node i* and set the support count of the new node to 1.

Case5: Check all the child nodes of the root and do not find a match for the above four cases.

(1). Under the root, add a new child node containing the items of *IST* (*tid j*) and set the support count of the new node to 1.

In the second phase, we process the transaction database that has been updated. Since the frequencies of some items have been changed, their corresponding *CFP-nodes* of the *CFP-tree* need to be rearranged. To adjust the *CFP-tree*, we have the following five steps:

Step1: For an update database DB+ containing new transactions, calculate the frequency of items presented and obtain a new *IF-list*.

Step2: Adjust the current tree to match the item order of the new *IF-list*. Here we move one branch (or path) at a time from the current *CFP-tree* to a temporary *CFP-tree* until no branch is found.

Substep2.1: Remove a branch from the current *CFP-tree*.

Substep2.2: Sort it by the item order of new *IF-list*.

Substep2.3: Add the sorted item sequence into the temporary *CFP- tree*.

Substep2.3.1: Treat the sorted item sequence as a transaction (*IST*) to be compared with the item sequence of the temporary tree node (*ISN*).

Substep2.3.2: Use the same five cases in *Step2* of the first phase to insert the branch into the temporary *CFP-tree*.

Step3: Process a transaction at a time in the update database DB+, updating the temporary *CFP-tree* until the last transaction is done.

Step4: Move the temporary *CFP-tree* to become the current *CFP-tree*.

Step5: Mine the frequent patterns from the current *CFP-tree* using the specified min-sup.

IV. Examples of The Proposed Approach

Examples are presented in three scenarios. First, we show how to construct a *CFP-tree* for mining frequent itemsets. Second, an incremental mining example for adding a new item sequence to the *CFP-tree* is provided after a database update. The last example for adjusting *CFP-tree* after adding a new transaction and deleting the oldest transaction is presented with stream data mining.

TABLE I.

Original Transaction Database (DB_1)

tid	Items bought
100	A, C, F, M, P
200	A, B, C, F, M
300	B, F
400	B, C, P
500	A, C, F, M, P

A. *CFP-tree Construction*

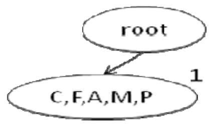
In the tree construction process, we use the transaction database DB_1, as shown in TABLE . Each row represents a transaction. The first column is the transaction identifier TID and the second column is the items being bought.

TABLE II.

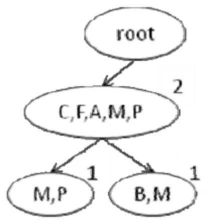
THE IF-LIST OF DB_1

C	F	A	B	M	P
4	4	3	3	3	3

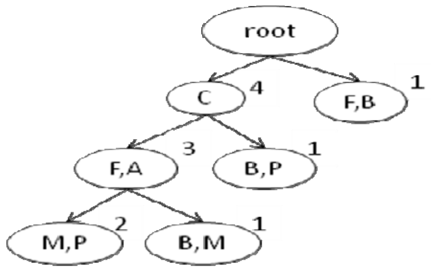
At the beginning, we need to obtain the *IF-list* by scanning the database and counting the frequency of each item. The items in the *IF-list* are sorted in descending order, as defined in Definition 3, which guarantees the unique sequence for the *CFP-tree* to be constructed. In this case, the ordering sequence is C, F, A, B, M, and P, as shown in TABLE II.



(a). After inserting *tid* 100



(b). After inserting *tid* 200



(c). After inserting *tid* 500

Figure 1. Building a CFP-tree for the original transaction database

After obtaining the *IF-list*, ADMiner will construct *CFT-tree* in this step. Before the items of transaction are added to the *CFT-tree*, their item sequences are sorted in the *IF-list*. The item sequence of first transaction is C, F, A, M, and P. These items are added to the *CFP-tree*. According to our rules, items with the same support count will be put into the same node, as shown in Figure 1(a). Secondly, the item sequence of *tid* 200 becomes C, F, A, B, and M. Since there exists a common path, C, F, and A, the first node will be split into two parts, (C, F, A) and (M, P) such that (M, P) becomes the first child of (C, F, A). After adjusting the tree, the node (B, M) is inserted as the next child node and their support counts are added, as shown in Figure 1(b). The above steps are repeated until all the transactions are processed. The final results are displayed in Figure 1(c).

B. *Incremental Data Mining*

In the real world, new transactions are added into databases all the time. To mine the database incrementally, ADMiner processes the new transactions efficiently without rescanning the original database. For simplicity, we use the first three transactions in TABLE as the original increment database (DB_2), and the remaining two transactions are regarded as the addition transaction database (DB_3).

TABLE III.

THE IF-LIST OF DB_2

F	A	B	C	M	P
3	2	2	2	2	1

TABLE IV.

THE IF-LIST TABLE OF DB_2 AND DB_3

C	F	A	B	M	P
4	4	3	3	3	3

First, we use DB_2 to build a *CFP-tree*. Then we apply the transactions in the addition database DB_3 to update DB_2 and its *CFP-tree*. The incremental mining process is summarized in the following steps.

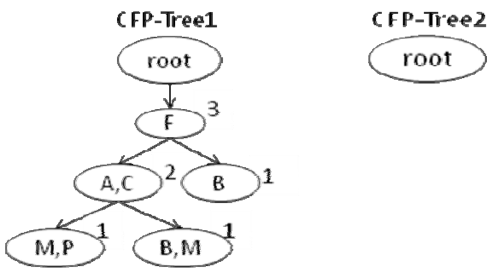
Step1. Adjust the *CFP-tree*

Before performing the database update, a *CFP-tree* is constructed in the *CFP-tree1* of Figure 2(a) and the order of item sequencing is F, A, B, C, M, and P, as shown in TABLE . We then scan the addition database DB_3 to update the *IF-list* and the results are shown in TABLE . The order of item sequencing has been changed to C, F, A, B, M, and P. Therefore, our *CFP-tree* must be adjusted.

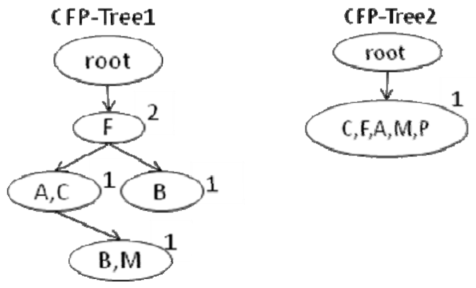
Step2. Update the *CFP-tree* using the addition transactions

In Figure 2 the original *CFP-tree* is constructed under *CFP-tree1*, and *CFP-tree2* is used to hold the new *CFP-tree* after the adjustment.

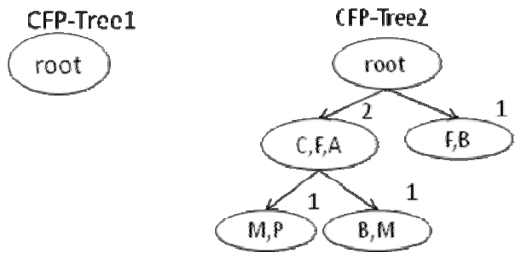
First, we remove the path (F, A, C, M, P) from Figure 2(a), resort it to become (C, F, A, M, P), as based on the order in TABLE , and add the results to *CFP-Tree2* as shown in Figure 2(b). Similarly, we remove the path (F, A, C, B, M), resort it to become (C, F, A, B, M), and add it to *CFP-Tree2*. The above steps are repeated until no path is left in *CFP-Tree1*. The results of *CFP-Tree2* are shown in Figure 2(c). Then, transactions *tid* 400 and *tid* 500 are added into *CFP-Tree2*, as shown in Figure 2(d).



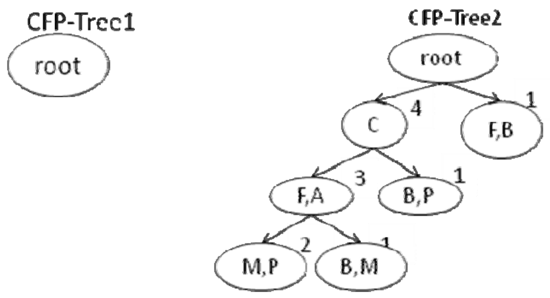
(a). The original CFP-tree is constructed under *CFP-Tree1*



(b). Remove the path (F, A, C, M, P) from *CFP-Tree1* of Figure (a) and add it to *CFP-Tree2*



(c). Remove the path (F, A, C, B, M) and (F, B) from *CFP-Tree1* of Figure (b) and add it to *CFP-Tree2*



(d). Insert *tid*s 400 and 500 (DB_3) into *CFP-Tree2*

Figure 2. *CFP-tree* construction and incremental update

C. Stream Data Mining

Since the deletion transaction simply causes a reverse effect on the database and the *CFP-tree*, ADMiner can take care of deletion as long as the deleted items (and itemsets) exist in the previous transactions.

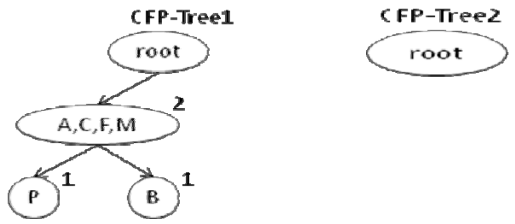
To demonstrate this capability, we use stream data mining as an extension of ADMiner. To mine stream data in a fixed size window, newly arriving transactions are added into the window while the old ones are removed. This works with the above assumption that our approach can delete existing items and itemsets. The rest of this section will give a simple example which shows how to perform stream data mining using ADMiner.

TABLE V.

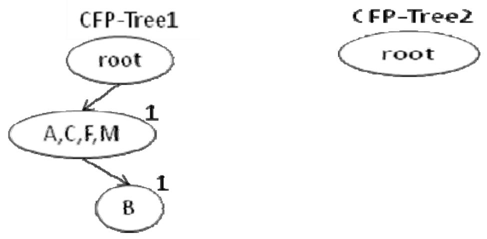
IF-LIST OF *tid* 200 AND *tid* 300

B	F	A	C	M	P
2	2	1	1	1	0

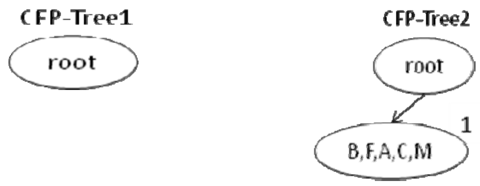
As before, we use the transaction database DB_2. In this example, the window size is 2. That means we only keep two transactions in our *CFP-tree* at a time.



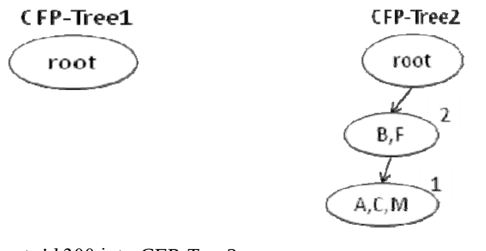
(a). Insert *tid* 100 and *tid* 200 into *CFP-Tree1*



(b). Delete *tid* 100 (A, C, F, M, P) from *CFP-Tree1* of Figure (a)



(c). Remove the path (A, C, F, M, B) from *CFP-Tree1* of Figure (b) and add it to *CFP-Tree2*



(d). Insert *tid* 300 into *CFP-Tree2*

Figure 3. *CFP-tree* construction for window-based stream data mining

To begin, we construct the *CFP-tree* using the same construction process. The first window has the first two transactions of DB_2, *tid* 100 and *tid* 200. They are added into the *FP-Tree1*, as shown in Figure 3 (a).

Next, *tid* 200 and *tid* 300 are in the second window. Hence, we delete transaction *tid* 100 and insert transaction *tid* 300. The *IF-list* is updated, as shown in TABLE . We get the item sequence B, F, A, C, and M.

The next step is to adjust the tree. Since transaction *tid* 100 (A, C, F, M, P) is deleted, we remove its items from the tree, as shown in Figure 3 (b). Then, as described in the Incremental Data Mining section, we use the assemble-and-detach approach to adjust the tree based on the *IF-list*. The path A, C, F, M, and B is removed from *CFP-Tree1* and inserted into *CFP-Tree2* in a new order of B, F, A, C, and M, as shown in Figure 3 (c). Then transaction *tid* 300 (B, F) is inserted into the new *CFP-Tree2*, as shown in Figure 3 (d). The final step is to replace *CFP-Tree1* with *CFP-Tree2* in preparation for the next run of database updating.

V. Experiments

A. *Experimental Environment*

We implemented ADMiner in Java with JDK 1.6. The experiments are all performed on a 2.66GHz Intel Core 2 Duo CPU with 4GB DDR2 memory and running Microsoft Windows XP SP2.

Some of the datasets used for the experiments are produced by the IBM dataset generator [15]. The length of each itemset follows a Poisson distribution whose mean is equal to L. The parameters for generating these datasets are shown in TABLE .

TABLE VI.

Parameters used for IBM dataset generator

D	Number of transactions
T	Average size of transactions
I	Average size of maximal potentially-large itemsets
L	Number of potentially-large itemsets
N	Number of items

In order to fairly evaluate ADMiner, a real dataset called BMS-POS [16] and several other datasets are used in our experiments. The BMS-POS dataset records several years of sales data from electronics retailers. Each transaction of the dataset represents the items a customer bought at one time. There are 515,597 transactions and 1,657 items in the dataset.

TABLE VII.

The number of nodes with BMS-POS

Dataset	Traditional *FP-tree* nodes	*CFP-Tree* nodes
BMS-POS	1,622,826	377,113

B. *Experimental Results*

We use the BMS-POS dataset to perform experiments on the traditional *FP-tree* algorithm and ADMiner. The results are shown in 错误!未找到引用源。. It can be observed that the total node numbers of the traditional *FP-tree* are four times greater than ours. This shows that the traditional *FP-tree* uses more memory space than ADMiner. Thus, our approach can process more transactions and items with limited memory.

In order to show the scalability of ADMiner in terms of the *CFP-tree* size, we divided the BMS-POS dataset into five partitions. Each partition has 100k transactions. The experimental results of memory usage are shown in Figure 4. It is obvious that the traditional *FP-growth* algorithm has a rapid increase in number of nodes when the number of transactions ranges from 100K to 500K. ADMiner has better scalability performance and uses less memory than the traditional *FP-growth* algorithm.

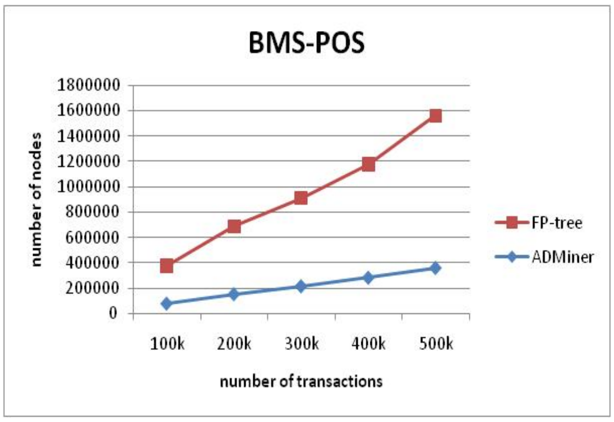


Figure 4. The comparison of the number of nodes with BMS-POS dataset.

In Figure 5, we can see that the execution performance of ADMiner is better than that of the traditional *FP-growth* algorithm. Because the tree size of ADMiner is smaller than the traditional *FP-tree*, traversing the whole tree requires less time.

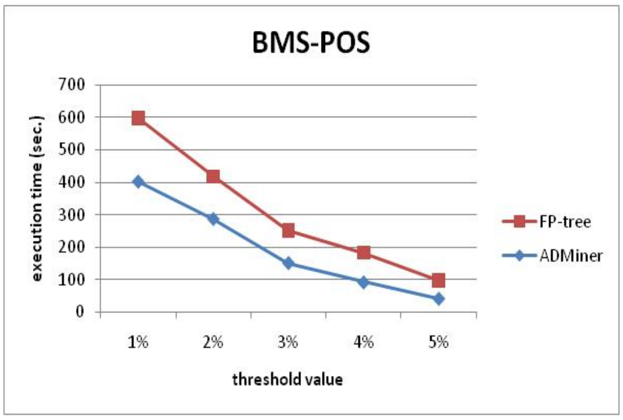


Figure 5. Execution time for different threshold values with BMS-POS dataset.

We also use the T10I4D100K dataset to measure ADMiner. The T10I4D100K dataset was generated by the IBM generator. This dataset has a total of 100,000 transactions and 999 distinct items. The results are shown in TABLE . In this experiment, it can be observed that the total number of the traditional FP-tree nodes is 714730 and the number of our *CFP-tree* nodes is 113511. The number of nodes in the traditional *FP-tree* is six times that of our *CFP-tree*.

We also divided the T10I4D100K dataset into five partitions to show the scalability of ADMiner. Each partition has 20k transactions. The experimental results of memory usage are shown in Figure 6. We can see that the traditional *FP-growth* algorithm has a rapid increase in node number as the number of transactions increases from 20k to 100k. The comparison of execution times is shown in Figure 7, where the performance of ADMiner is better than that of the traditional algorithm. When the minimum support is greater than 2.5%, the number of frequent itemsets decreases and the advantages of ADMiner also go down.

TABLE VIII.

THE NUMBER OF NODES WITH T10I4D100K

Dataset	Traditional *FP-tree* nodes	*CFP-Tree* nodes
T10I4D100K	714730	113511

For dense datasets, we use a real world chess dataset [16] for our experiments. This dataset has a total of 3190 transactions and 75 distinct items. The results are shown in TABLE . The number of nodes in the traditional *FP-tree* is six times that of our *CFP-tree*.

TABLE IX.

THE NUMBER OF NODES WITH CHESS DATASET

Dataset	Traditional *FP-tree* nodes	*CFP-tree* nodes
chess	38609	5800

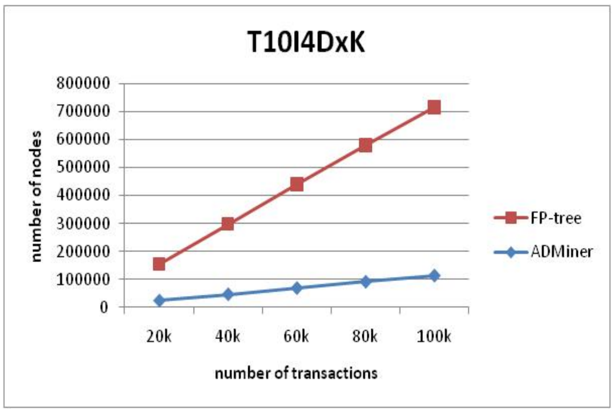


Figure 6. The comparison of the number of nodes with T10I4D100K dataset.

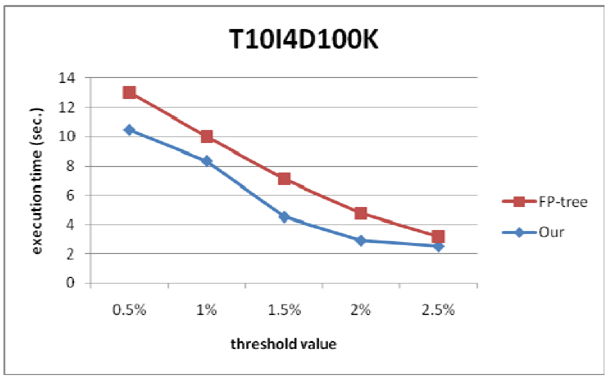


Figure 7. Execution time for different threshold values with T10I4D100K dataset.

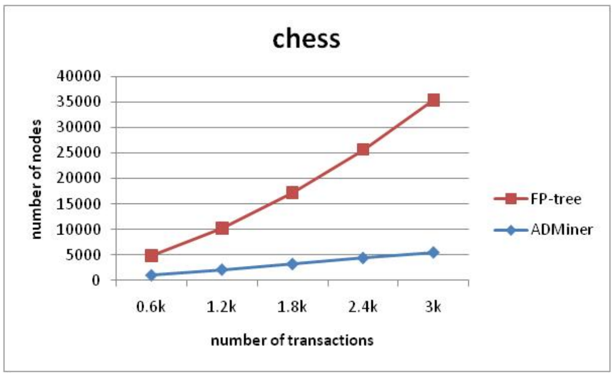


Figure 8. The comparison of the number of nodes with chess dataset.

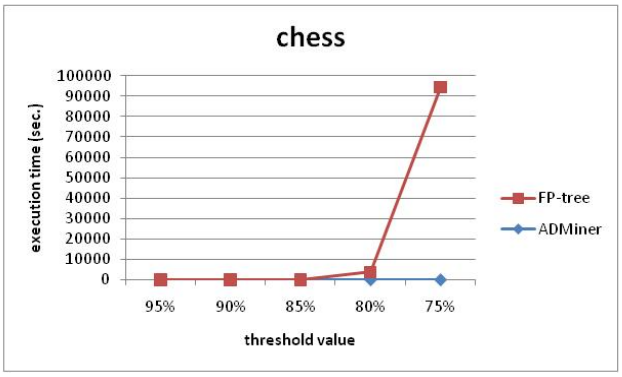


Figure 9. Execution time for different threshold values with chess dataset.

We also divided the chess dataset into five partitions to show the incremental update application (addition) of ADMiner vs. that of the traditional *FP-tree* algorithm. Each partition has 0.6k transactions. Experimental results of memory usage are shown in Figure 8. From these results, we can see that the traditional *FP-tree* algorithm has a rapid increase in node number when the number of transactions ranges from 0.6k to 3k. The comparison of execution times is shown in Figure 9. We can see the performance of ADMiner is better than that of the traditional algorithm when the minimum threshold is less than 80%.

From the above experiments, we know that ADMiner uses less memory than the traditional *FP-tree*. On average, the node number in the traditional *FP-tree* is five times that of our *CFP-tree*. The same is true for the execution time since our *CFP-tree* is much smaller than the traditional *FP-tree*.

C. Discussion

First, we discuss some situations which may influence the performance of ADMiner. If the transaction database DB is a dense dataset, there will be many common paths. This means that many common branches can be shared in a tree. Thus, our *CFP-tree* is much smaller than a traditional *FP-tree* and has many advantages. ADMiner can process a larger number of transactions and items with limited memory, and also traverse the tree quickly.

On the other hand, if the transaction database is a sparse dataset, our *CFP-tree* is still smaller than a traditional *FP-tree*. Although our tree will be wider than that of a dense dataset, it can still save memory space. However, because there are less common paths, it needs to search more nodes than a dense dataset does. Hence, it takes more time during node searching, but the memory usage is still very efficient.

VI. Conclusions and Future Work

In real world applications, data mining algorithms often face the problems of huge databases and routine updates. Since memory space is limited, processing the updated database without rescanning the original one is a critical issue. We propose an improved data structure based on *FP-tree* to process more transactions using limited memory. In addition to saving memory, ADMiner deals with the incremental mining of the dynamic database by using the results of previous mining processes.

In future research, we will make further improvements and adjustments to the *CFP-tree* and apply our approach when mining closed itemsets [17], sequential patterns [18] and global-local frequent patterns [19].

Acknowledgment

This research was supported by the National Science Council, Taiwan, under grants NSC 98-2221-E-035-059-MY2 and NSC 101-2221-E-035-082.

References

[1] R. Agrawal, T. Imielinski, and A. Swami, “Mining association rules between sets of items in large database,” *ACM SIGMOD Int’l Conf. on Management of Data*, 1993, pp. 207-216.

[2] J. Han, J. Pei, and Y. Yin, “Mining frequent patterns without candidate generation,” *ACM SIGMOD Int’l Conf. on Management of Data*, 2000, pp. 1-12.

[3] C.W. Lin, T.P. Hong, and W.H. Lu, “The Pre-FUFP algorithm for incremental mining,” *Expert Systems with Applications*, vol. 35. no. 5, 2009, pp. 9498-9505.

[4] C.I. Ezeife, and Y. Su, “Mining incremental association rules with generalized FP-tree,” *Proc. of the 15th Conf. of the Canadian Society for Computational Studies of Intelligence on Advances in Artificial Intelligence*, 2002, pp. 147–160.

[5] J.L. Koh and S.F. Shieh, “An efficient approach for maintaining association rules based on adjusting FP-tree structures,” *The Ninth Int’l Conf. on Database Systems for Advanced Applications*, Vol. 2973, 2004, pp. 417-424.

[6] W. Cheung and O.R. Zaïane, “Incremental mining of frequent patterns without candidate generation or support constraint,” *Proc. of the Seventh Int’l Database Engineering and Applications Symposium (IDEAS’03)*, 2003, pp. 111-116.

[7] A. Savasere, E. Omiecinski, and S. Navathe, “An efficient algorithm for mining association rules in large databases,” *Proc. of the 21st Int’l Conf. on VLDB*, 1995, pp. 432-444.

[8] S. Orlando, P. Palmerini, and R. Perego, “Enhancing the Apriori algorithm for frequent set counting,” *Proc. of the Third Int’l Conf. on Data Warehousing and Knowledge Discovery*, 2001, pp. 71-82.

[9] H. Huang, X. Wu, X. and R. Relue, “Association analysis with one scan of databases,” *Proc. of the IEEE Int’l Conf. on Data Mining (ICDM’02)*, 2002, pp. 629-632.

[10] K. Wang, L. Tang, J. Han, and J. Liu, ”Top down FP-Growth for association rules mining,” *Proc. of the Sixth Pacific-Asia Conf. on Advances in Knowledge Discovery and Data Mining (PAKDD’02)*, 2002, pp. 334-340.

[11] Y.J. Tsay, T.J. Hsu, and J.R. Yu, ”FIUT: A new method for mining frequent itemsets,” *Information Sciences*, vol. 179, no. 11, 2009, pp. 1724-1737.

[12] R.P. Gopalan, and Y.G. Sucahyo, “High performance frequent patterns extraction using compressed FP-Tree,” *Proc. of the SIAM Int’l Workshop on High Performance and Distributed Mining*, April 2004.

[13] F. Chen, L. Shang, M. Li, Z.G. Chen, and S.F. Chen, “Mining frequent patterns based on compressed FP-tree without conditional FP-tree generation,” IEEE Int’l Conf. on Granular Computing, 2006, pp. 478- 481.

[14] Y.G. Sucahyo, and R.P. Gopalan, “CT-PRO: A bottom-up non recursive frequent itemset mining algorithm using compressed FP-tree data structure,” *Proc. of the IEEE ICDM Workshop on Frequent Itemset Mining Implementations (FIMI)*, Brighton, UK, 2004.

[15] IBM Almaden Research Center, 2006. Synthetic data generation code for associations and sequential patterns, *URL: http://www.almaden.ibm.com/ software/quest/*

[16] KDDCUP2000, *http://www.ecn.purdue.edu/KDDCUP*.

[17] C.H. Lee, K.C. Yin, D.L. Yang, and J. Wu, “Efficient mining of frequent closed itemsets without closure checking,” *I. J. Comp. Inf. Sys. & Industrial Management Appl. (IJCISIM)*, vol. 1, no. 2, 2009, pp. 58-67.

[18] J.R. Lin, C.Y. Hsieh, D.L. Yang, J. Wu, and M.C. Hung, “A flexible and efficient sequential pattern mining algorithm,” *Int’l. J. of Intelligent Information and Database Systems*, vol. 3, no. 3, 2009, pp. 291-310.

[19] K.C. Yin, Y.L. Hsieh, and D.L. Yang, “GLFMiner: global and local frequent pattern mining with temporal intervals,” *Proc. of the 5th IEEE Conf. on Industrial Electronics and Applications (ICIEA2010)*, 2010, pp. 2248-2253.

Chien-Min Lin received the B.E. degree and M.S. degree from the Department of Information Engineering and Computer Science at Feng Chia University, Taiwan, in 2004 and 2008 respectively. His research interests include data mining and distributed system.

Yu-Lung Hsieh received the M.S. degree from the Department of Information Engineering and Computer Science at Feng Chia University, Taiwan, in 2005. He is now a Ph. D. candidate in the Department of Information Engineering and Computer Science at Feng Chia University. His research interests include data mining and genetic algorithm.

Kuo-Cheng Yin received the B.E. degree and M.S. degree from the Department of Information Engineering and Computer Science at Feng Chia University, Taiwan, in 1990 and 1992 respectively. He is now a Ph. D. candidate in the Department of Information Engineering and Computer Science at Feng Chia University. His research interests include data mining and software engineering.

Ming-Chuan Hung received the B.E. degree in Industrial Engineering and the M.S. degree in Automatic Control Engineering from Feng Chia University, Taiwan, in 1979 and 1985 respectively, and the Ph.D. degree from the Department of Information Engineering and Computer Science at Feng Chia University in 2006. From 1985 to 1987, he was an instructor in the Mechanics Engineering Department at National Chin-Yi Institute of Technology. Since 1987, he has been an instructor in the Industrial Engineering Department at Feng Chia University and served as a secretary in the College of Engineering from 1991 to 1996. Dr. Hung is currently an Associate Professor. His research interests include data mining, CIM, and e-commerce applications. He is a member of the CIIE.

Don-Lin Yang received the B.E. degree in Computer Science from Feng Chia University, Taiwan in 1973, the M.S. degree in Applied Science from the College of William and Mary in 1979, and the Ph.D. degree in Computer Science from the University of Virginia in 1985. He was a staff programmer at IBM Santa Teresa Laboratory from 1985 to 1987 and a member of the technical staff at AT&T Bell Laboratories from 1987 to 1991. Since then, he joined the faculty of Feng Chia University, where he was in charge of the University Computer Center from 1993 to 1997 and served as the Chairperson of the Department of Information Engineering and Computer Science from 2001 to 2003. Dr. Yang is currently a professor at Feng Chia University. His research interests include software engineering, database systems and data mining. He is a member of the IEEE computer society and the ACM.