High Performance Frequent Patterns Extraction using Compressed FP-Tree

Raj P. Gopalan **Yudho Giri Sucahyo**

Department of Computing, Curtin University of Technology
Kent Street, Bentley, Western Australia 6102
{raj, sucahyoy}@cs.curtin.edu.au

Abstract

Many algorithms have been proposed to improve the performance of mining frequent patterns from transaction databases. Pattern growth algorithms like FP-Growth based on the *FP-tree* are more efficient than candidate generation and test algorithms. In this paper, we propose a new data structure named *Compressed FP-Tree* (*CFP-Tree*) and an algorithm named CT-PRO that performs better than the current algorithms including FP-Growth, OpportuneProject, and Apriori. The number of nodes in a *CFP-Tree* can be up to 50% less than in the corresponding *FP-Tree*. CT-PRO is empirically compared with FP-Growth, OpportuneProject, Apriori and CT-ITL using datasets that reveal the effective performance range of these algorithms. CT-PRO is also extended for mining very large databases and its scalability evaluated experimentally.

1 Introduction

Association Rule Mining (ARM) [1] has been the focus of research in many communities (e.g. data mining, artificial intelligence, machine learning) for a decade. Traditionally, ARM has been defined on market basket data. However, it has been used in many other application areas and also extended to data mining tasks of classification [2] and clustering [3]. Several ARM algorithms have been proposed based on both sequential and parallel paradigms. However, the existing algorithms rely on expensive computations using large amounts of memory or require many I/O scans over the database. By intelligently utilizing the limited memory and reducing the number of scans over the database, we can further improve the performance of ARM.

ARM algorithms typically divide the problem into two parts: find the frequent patterns and then use them to form the rules. The general performance of ARM is determined by the first part. Once frequent patterns are found, generating the association rules is straightforward. Constraints such as *support* and *confidence* are used to reduce the search space during mining.

The *Apriori property* (if a pattern is infrequent then its supersets can never be frequent) is the foundation for reducing the cost of all algorithms in ARM. The Apriori algorithm uses the *candidate generation and test* approach [4]. The main drawback of this approach is the many traversals over the database required to enumerate a significant part of the possible 2^n frequent patterns where n is the number of items. For sparse datasets where n is small, Apriori performs quite well, but its performance drops off rapidly when mining dense datasets with many long patterns.

The performance problem is partly overcome by algorithms using the *pattern growth* approach. It involves a divide and conquer strategy, whereby the database is divided into several projections and each projection is mined independently. The well-known FP-Growth algorithm implements this strategy to good effect [5].

Another factor contributing to the efficiency of FP-Growth is its compact representation of the database in memory using a variant of the prefix tree named *FP-Tree*. The use of prefix tree itself was introduced first in [6]. The performance gain from using variants of the prefix tree for representing transactions was previously demonstrated in [5] [7] [8] and [9]. In this paper, we propose a new data structure named *Compressed FP-Tree* (*CFP-Tree* for short) that is even more compact than *FP-Tree*.

One weakness of FP-Growth is the overhead of constructing many *conditional FP-Trees* during mining that reduces its performance as the patterns get longer and/or the support level gets lower. In this paper, we present a new algorithm named CT-PRO that divides the database into several projections and then mines each projection independently. The projections are also represented as *CFP-Trees.* Unlike FP-Growth, the mining process is performed by a non-recursive function and so we avoid the overhead of creating an extra data structure at each mining step.

The performance of CT-PRO is compared against other known efficient algorithms. To study the feasible performance range of the algorithm, we carried out extensive testing using a set of databases with varying number of both transactions and average number of items per transaction. For each dataset, we tested all the algorithms on supports 10 to 90. To relate our experimental results with previous work in the literature, we include performance results on the widely used Connect-4, Chess and Mushroom test datasets [16], which are also available at the FIMI (Fre-

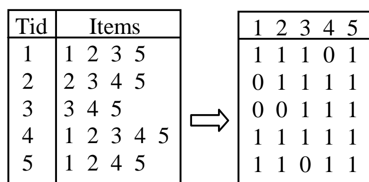
quent Itemset Mining Implementations) repository [18]. We also extended CT-PRO for very large databases and tested it on datasets ranging from 0.1 million to 11 million transactions. The results are presented later in this paper.

The rest of the paper is organized as follows: In Section 2, we define relevant terms used in frequent pattern mining. In Section 3, we describe the design and construction of *CFP-Tree*. Section 4 contains a description of the CT-PRO algorithm and the experimental results are reported in Section 5. Section 6 describes the extension of CT-PRO for mining very large databases along with experimental results. We conclude with a summary and pointers for further work in Section 7.

2 Definitions of Terms

Given a set of items $I=\{I_1, I_2, I_3, \ldots, I_n\}$ and a database *D* as a set of transactions *T*, each transaction is a subset of *I* ($T \subseteq I$) and is identified by a *TID*. An itemset *X* is a subset of items ($X \subseteq I$), and an itemset of length *k* is called a *k-itemset*. The *support* of an itemset *X* is the percentage of transactions in *D* that contains *X*. If the *support* of an itemset is greater than or equal to a given *support threshold* σ, it is called a *frequent itemset* or *frequent pattern* otherwise it is *infrequent*. The objective of *frequent pattern extraction* is to find all frequent patterns, given an input database *D* and a *support threshold* σ.

The input database *D* can be represented as an *m x n* matrix where *m* is the number of transactions and *n* is the number of items (see Figure 1). We can denote the presence or absence of an item in each transaction by a binary value (1 if present, else 0). Counting the support for an item is the same as counting the number of 1s for that item in all the transactions. The sparseness or denseness can be determined by a *density measure* defined as the percentage of 1s in the total of 1s and 0s. If a dataset contains more 1s than 0s, it can be considered as a *dense* dataset, otherwise, as a *sparse* dataset.



Tid	Items
1	1 2 3 5
2	2 3 4 5
3	3 4 5
4	1 2 3 4 5
5	1 2 4 5

⇨

1	2	3	4	5
1	1	1	0	1
0	1	1	1	1
0	0	1	1	1
1	1	1	1	1
1	1	0	1	1

Figure 1. Binary representation of a transaction database

3 Compressed FP-Tree: Design and Construction

The prefix tree was first used for ARM in the TreeProjection algorithm [6] that outperformed Apriori. Such a tree is also called a Trie [10]. Han et al. employed a variant of the prefix tree named *FP-Tree* for their FP-Growth algorithm in [5]. Experiments showed that FP-Growth performed better than Apriori and TreeProjection. Another variant named *CATS Tree* was proposed in [9]. *CATS Tree* stores all the items present in transactions and so its size is bigger compared to *FP-Tree* that only stores frequent items. A variant of *FP-Tree* named *COFI-Tree* was proposed in [11]. The difference between *COFI-Tree* and *FP-Tree* is that *COFI-Tree* maintains bi-directional links allowing bottom-up scanning as well, but the number of nodes is the same as in *FP-Tree*. In [12], we introduced a more compact representation of the prefix tree (named *CT-Tree*). We showed that the more compact the data structure, the faster the algorithm for mining frequent patterns [13].

Now we propose a *Compressed FP-Tree* (*CFP-Tree*) that reduces the size of the *FP-Tree* by up to half. The items are in descending order of their frequency in a *CFP-Tree* (instead of ascending order in the *CT-Tree*), and there is a link to the next node with the same-item-node (while links are not present in the *CT-Tree*).

FP-Tree stores the item id in the tree while in *CFP-Tree* the item ids are mapped to an ascending sequence of integers that is actually the array index in *HeaderTable* (to use the same terminology as in *FP-Tree*). The frequency of each item is also stored in *HeaderTable*. We compress the *FP-Tree* by removing identical subtrees of a complete *FP-Tree* and by succinctly storing the information from them in the remaining nodes. All subtrees of the root of the *FP-Tree* (except the leftmost branch) are collected together at the leftmost branch to form the *CFP-Tree*. The *HeaderTable* contains a pointer to each node on the leftmost branch of the *CFP-Tree*, as these nodes are roots of subtrees starting with different items.

Figure 2 shows the *FP-Tree* and the *CFP-Tree* for a sample database. In this case, the *FP-Tree* is a complete tree for items 1-4. The *HeaderTable* of the *CFP-tree* has two additional columns compared to the *HeaderTable* of the *FP-Tree*. These are the frequency count of each item and an index that renames the items arranged in the descending order of frequency. Each node of the *CFP-Tree* contains an array of counts for item subsets in the path from the root to that node. The index of the cells in the array corresponds to the level numbers of the nodes above. The number of nodes in the *FP-Tree* is twice that of the corresponding *CFP-Tree*.

The *CFP-Tree* is constructed as follows: Suppose we want to mine the frequent patterns from the transaction

database shown in Figure 3a with a minimum support threshold σ = 40% (2 transactions). We need to first determine the frequent items by reading the database once. The frequent items are stored in descending order of item frequency in *HeaderTable*. In a second pass over the database, we select only the frequent items of each transaction, map each item to its index in *HeaderTable* on-the-fly, sort items of the transaction in ascending order of their index id, and insert them into the *CFP-Tree* (see Figure 3b and 3c). The result of this step is shown in Figure 4. The pointer in *HeaderTable* also acts as the start of the links to other nodes with

Tid	Items	Tid	Items	Tid	Items
1	1 2 3 4	6	2	11	1
2	2 4	7	1 4	12	2 3 4
3	1 3 4	8	1 2 3	13	1 2
4	3	9	3 4	14	1 2 4
5	2 3	10	4	15	1 3

(a) Sample Database

(b) *FP-Tree*

(c) *CFP-Tree*

Figure 2. *FP-Tree* and *CFP-Tree*

Tid	Items
1	3 4 5 6 7 9
2	1 3 4 5 13
3	1 2 4 5 7 11
4	1 3 4 8
5	1 3 4 10

(a) Sample Database

Tid	Items
1	3 4 5 7
2	1 3 4 5
3	1 4 5 7
4	1 3 4
5	1 3 4

(b) Frequent Items

Tid	Items
1	1 2 4 5
2	1 2 3 4
3	1 3 4 5
4	1 2 3
5	1 2 3

(c) Mapped

Figure 3. Mapping Frequent Items

Figure 4. Example Global *CFP-Tree*

```
1  /*Input: database Output: HeaderTable*/
2  Procedure ConstructHeaderTable
3  For each transaction in the database
4    For each item in a transaction
5      If item in HeaderTable
6        Increment count of item in
           HeaderTable
7      Else
8        Insert item into HeaderTable
           with count = 1
9       End If
10   End For
11 End For
12 Delete infrequent items and sort
   HeaderTable in frequency descending order
13 Assign an index for each frequent item

14 /* Input: database, HeaderTable, min_sup
   Output: Global CFP-Tree */
15 Procedure ConstructTree
16 Build_LeftMost_Branch_of_the_Tree()
17 For each transaction in the database
18   Initialize mappedTrans
19   For each frequent item in the trans
20   /*get index of items from HeaderTable*/
21    mappedTrans = mappedTrans ∪
        GetIndex(item)
22   End For
23   Sort(mappedTrans)
24   InsertToTree(mappedTrans)
25 End For

26 Procedure InsertToTree(mappedTrans)
27 firstItem = mappedTrans[1]
28 currNode = root of subtree pointed by
     HeaderTable[firstItem]
29 For each subsequent item i in mappedTrans
30   If currNode has child representing i
31    increment count[firstItem-1] of
       the child node
32   Else
33    create child node and
       set its count[firstItem-1]=1
34    Organise the same-item-node-link
35   End If
36 End For
```

Figure 5. Algorithms for Constructing *CFP-Tree*

same item ids (indicated by the dashed lines in Figure 4). For illustration, at each node we also show the index of the array, the transaction represented at each index entry and its count. In the implementation of *CFP-Tree*, however, we store only the second column that represents the count. The algorithm for constructing the *CFP-Tree* is given in Figure 5.

4 Mining the CFP-Tree

In this section, we describe the CT-PRO algorithm for mining complete frequent patterns using the *CFP-Tree*. The pointers in the *HeaderTable* are used as the starting points to mine all frequent patterns in the transaction tree. We go from the least frequent item to the most frequent so that a larger number of nodes can be pruned in early iterations letting the tree shrink quickly. The same-item-node-link is traversed to create a projection of all transactions ending with that item. The projection is also represented as a *CFP-Tree* that contains *local frequent items* with a new set of index numbers assigned to the items. For clarity, we call it a *local CFP-Tree* and its size will naturally be much smaller than that of the *global CFP-Tree*. The *local CFP-Tree* is traversed to extract the frequent patterns in the projection. The frequent patterns in a projection are represented in a *local frequent pattern tree*. The mining step of the algorithm is shown in Figure 6 and the following example illustrates the working of the algorithm on the sample database.

Example 1. Let us examine the mining process based on the *global CFP-Tree* shown in Figure 4.

Figure 7 shows the *local CFP-Tree* and *local frequent pattern tree* at each step during the mining process. We start from the least frequent item (index_id: 5 item_id: 7) in the *globalHeaderTable* (line 4). Item_id 7 is frequent and it will be the root of the *local frequent pattern tree* (line 5).

Then CT-PRO creates a projection of all transactions ending with 5. This projection is represented by a *local CFP-Tree* and only contains *locally frequent items.* Traversing the same-item-node-link of item-index 5 in the *global CFP-Tree* identifies the local frequent items that occur together with it. There are three nodes of index 5 and the path to the root for each node is traversed counting the other indexes that occur together with item-index 5 (lines 15-24). In all, we have 1(2), 2(1), 3(1) and 4(2) for index 5 (the count is shown in brackets). As indexes 1,4 (item_id: 4,5) are *locally frequent* (*support* $\geq$ 2) in projection of item-index 5, items 4 and 5 are registered in the *localHeaderTable* with new index id (see Figure 7a). Each entry in the *localHeaderTable* also becomes the child of the *local frequent pattern tree's* root (lines 7-9). Together, the root and its children form 2-frequent patterns (frequent pattern with length 2). On mapping to their item id, we get patterns 74(2), 75(2) as frequent.

```
1  /*  Input: HeaderTable, Global CFP-Tree */
2  /*  Output: frequent patterns */
3  Procedure Mining
4   For each item x ∈ globalHeaderTable from
       the least freq to the most freq
5    Initialize localFreqPattTree
      with x as the root
6    ConstructLocalHeaderTable(x)
7    For each freq item i in localHeaderTable
8        Attach i as a child of x
9    End For
10   ConstructLocalCFPTree(x)
11   MineRest(x)
12   Traverse the Local Frequent Pattern Tree to
       print the frequent patterns
13  End For
14 End For

15 Procedure ConstructLocalHeaderTable(i)
16 For each occurrence of node i in the tree
17  For each item in the path to the root
18   If item in localHeaderTable
19    Increment count of item
20   Else
21    Insert item with count = 1
22   End If
23  End For
24 End For

25 Procedure ConstructLocalCFPTree(i)
26 For each occurrence of node i in the tree
27  Initialize mappedTrans
28  For each freq item in the path
     to the root
29   mappedTrans=mappedTrans ∪ GetIndex(item)
30  End For
31  Sort(mappedTrans)
32  InsertToTree(mappedTrans)
33 End For

34 Procedure MineRest(x)
35 For each child i of x
36  Set all counts in localHeaderTable to 0
37  For each occurrence of node i in localCFPTree
38   For each item in the path to the root
39    Increment count of item in
        localHeaderTable
40   End For
41  End For
42  For each freq item j in localHeaderTable
43   Attach j as a child of i
44  End For
45  MineRest(node i)
46 End For
```

Figure 6. Mining Frequent Patterns in CT-PRO

After we identify *local frequent items* for the projection, the same-item-node-link in the *global CFP-Tree* is re-traversed and the path to the root from each node is revisited to get the *local frequent items* occurring together with index 5 in the transactions. These *local frequent items* are mapped to their index in the *localHeaderTable* on-the-fly, sorted in ascending order of their index id and inserted into the *local CFP-Tree* (lines 25-33). The first path of item-index 5 returns nothing. From the second path of index 5, we insert a transaction 14(1) to the *local CFP-Tree* and we

insert another transaction 14(1) from the third path of item-index 5. In total, we have two occurrences of transaction 14. Indexes 1 and 4 in *globalHeaderTable* represent items 4 and 5 respectively and the indexes for item 4 and 5 in *localHeaderTable* are 1 and 2 so in the *localCFP-Tree* transaction 14 will be mapped to 12. As the item index in *globalHeaderTable* and *localHeaderTable* are different, the item id (the second column) is always maintained for output purposes.

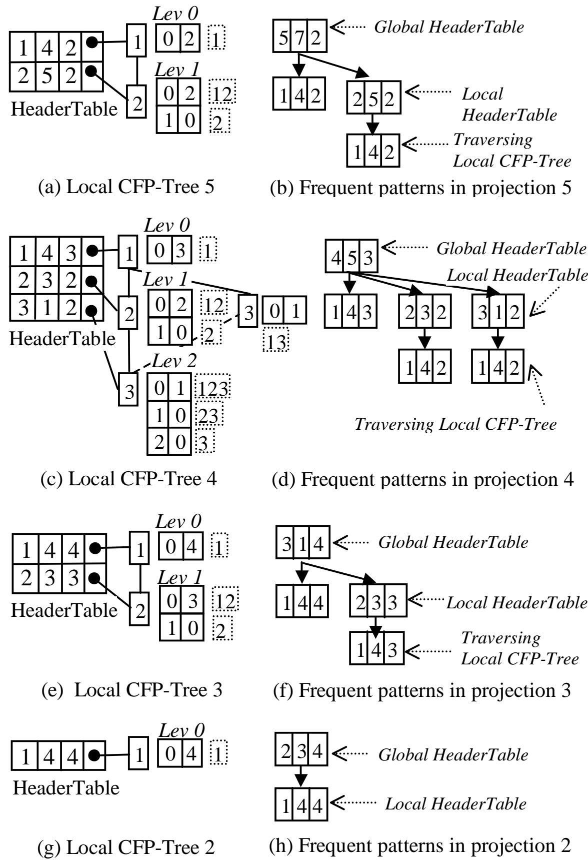


Figure 7. Local *CFP-Tree* during Mining Process

Longer frequent patterns (length > 2) will be extracted by calling another procedure (line 11). We have described this procedure (lines 34-46) using recursion but in the program, it is implemented as a non-recursive procedure for performance reasons. Starting from the least frequent item in the *localHeaderTable* (line 35), the same-item-node-link is traversed (line 37-41). For each node, the path to the root is traversed counting the other items that are together with the current item. For example, in Figure 7a, traversing the same-item-node-link of node 2 will return item 1(2), and since it is frequent, an entry is created and attached as the child of index 2 in the *local frequent pattern tree* (lines 42-44). All frequent patterns containing item 7 (index_id: 5) can be extracted by traversing the *local frequent pattern tree* (line 12).

The process is continued to mine the next item in *globalHeaderTable:* index_id 4 (item_id: 5), index_id 3 (item_id: 1), index_id 2 (item_id: 3) and finally, when the mining process reaches the root of the tree of Figure 4, it outputs 4(5). In all, 17 frequent patterns are extracted: 7(2), 74(2), 75(2), 754(2), 5(3), 54(3), 53(2), 51(2), 534(2), 514(2), 1(4), 14(4), 13(3), 134(3), 3(4), 34(4), 4(5).

As mentioned in Section 1, at each step of recursive mining, FP-Growth needs to create a *conditional FP-Tree*. This overhead adversely affects its performance, as the number of *conditional FP-Trees* will correspond to the number of frequent patterns. In CT-PRO, for each frequent item *f* (*not frequent itemsets*), only one *local CFP-Tree* is created and traversed non-recursively to extract all frequent patterns beginning with *f*. By doing this we avoid the cost of creating *conditional FP-Trees* as in FP-Growth.

5 Empirical Evaluation

In this section, we report the result of the experiments to evaluate performance of CT-PRO. All testing was performed on an 866 MHz Pentium III PC, 512 MB RAM, 110 GB HD running MS Windows 2000. The source codes were compiled using MS Visual C++ 6.0. The runtime include both CPU time and I/O time.

We compared CT-PRO with other efficient algorithms: Apriori, FP-Growth, OP and CT-ITL. Apriori 4.08 [7], generally acknowledged as the fastest Apriori implementation, also employs prefix trees to store the transactions and the frequent itemsets. CT-ITL [12] is an algorithm that combines the pattern growth approach with tid-intersection method [14] using *CT-Tree* data structure. OP is an extension of FP-Growth proposed in [8]. It is essentially a combination of FP-Growth and H-Mine [15]. Though not as well known as FP-Growth, OP is generally faster than FP-Growth.

For a comprehensive evaluation of the algorithm's performance, we generated ten datasets using the synthetic data generator [17]. The first five datasets contain 50,000 transactions on 100 items, with average number of items per transaction of 10, 25, 50, 75 and 100. The second five datasets contain 100,000 transactions on 100 items, with average items per transaction of 10, 25, 50, 75 and 100. The algorithms were extensively tested on these datasets, with support of 10 to 90 in increments of 10. As the average number of items gets larger and/or the support level gets lower, at some point, every algorithm 'hits the wall' (takes too long to complete).

Figure 8 shows the performance comparisons of CT-PRO, Apriori, FP-Growth, OP and CT-ITL on various

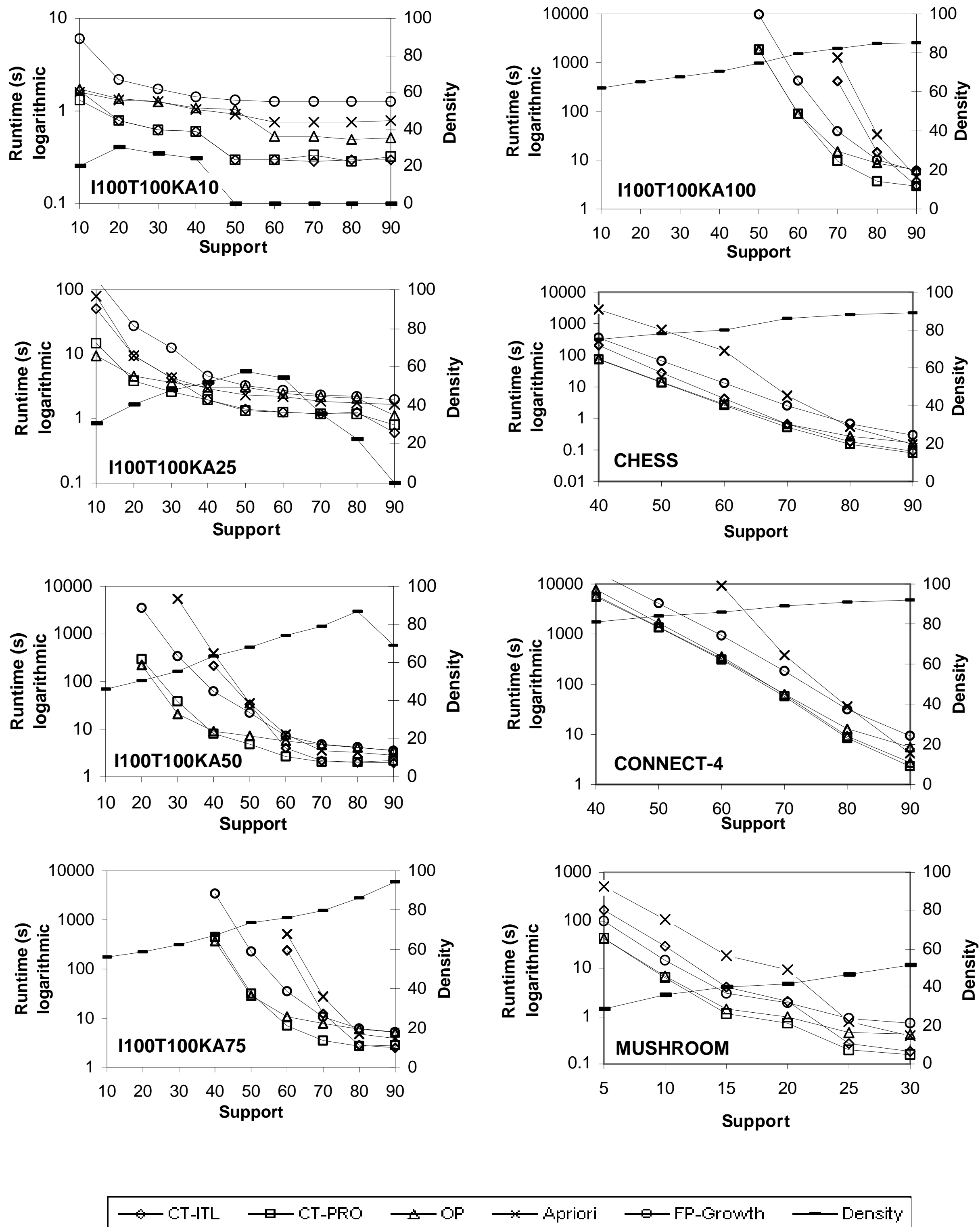


Figure 8. Performance Comparisons on Various Datasets

datasets. All the charts use a logarithmic scale for run time along the *y*-axis on the left of the chart. The *y*-axis on the right shows the *density measure* (see Section 2) at various support levels. We did not plot the result in the chart if the runtime is more than 10,000 seconds. Since items that are frequent depend on the minimum support level chosen, the density also varies at different minimum support levels. Therefore, whether the input data for ARM is dense or not can be decided only after frequent items are identified. The dataset name shows its characteristics. For example, I100T100KA10 means there are 100 items, and 100,000 transactions with an average of 10 items per transaction.

We note several interesting points from the performance results:

1. The performance characteristics on databases of 50,000 transactions to 100,000 transactions are quite similar. However, the runtimes increase with the number of transactions.
2. For sparse datasets (average items per transaction 10 and 25), the performance of FP-Growth is the worst at all support levels. This shows that the traditional approach (candidate-generation-and-test, used in Apriori) is not always worse than pattern growth.
3. As the average items per transaction gets larger and the support gets lower, the performance of the algorithms begin to deteriorate. We set 'the wall' where the runtime is more than 10,000 seconds. For average items per transaction of 50, Apriori can only mine down to support 30, but FP-Growth, OP and CT-PRO can mine down to support 20. At an average 100 per transaction, Apriori can only mine down to support 70, while CT-PRO, FP-Growth and OP can mine down to support 50.
4. CT-PRO generally performs the best at most support levels, but as the support gets lower, its performance is similar to OP. Only at a few points, OP runs slightly faster than CT-PRO.

As mentioned in Section 1, the objective of our empirical study was to determine the feasible performance range of the algorithm. Sample datasets such as real-world BMS datasets or UCI ML Repository datasets [18] do not cover the full range of densities as the synthetic datasets we generated for this purpose. However, to relate our experiments with previous work, Figure 8 also shows the performance comparisons on Connect-4 (129 items, 67557 transactions, average items per transaction 43), Chess (75 items, 3196 transactions, average items per transaction 37) and Mushroom (119 items, 8124 transactions, average items per transaction 23) that are commonly used for evaluating ARM algorithms [16][18]. Connect-4 and Chess are small dense datasets containing many long frequent patterns at various levels of support. The results show that CT-PRO outperforms others at *all* support levels we used on those datasets. On Connect-4, we did not continue the testing of Apriori below support 60 and support 50 for FP-Growth and CT-ITL as they passed the 10,000 seconds limit.

6 Mining Very Large Databases

We have assumed so far that the entire database would fit in the memory. Even though current memory sizes have reached gigabytes in size, many databases are even larger. A scheme for partitioning the database is needed to make CT-PRO scalable. We have enhanced the partitioning approach originally proposed in [5] by compressing transactions in the partitions.

We also added a memory tracking subsystem in the algorithm. During the second scan of the database, when we extract frequent items from transactions, sort on-the-fly and insert into the *CFP-Tree,* if the memory is full and the input data is not finished, CT-PRO will create partitions and write the transactions kept in the tree to those partitions. A transaction in a partition will represent a group of actual transactions with the same frequent items. Compared to the original proposal in [5], the total size of partitions will be much smaller as a result of grouping.

In each partition, the first number in the list of items shows the number of transactions belonging to the group. Transaction *T* with *i* as the first frequent item will be written into partition *i*. Each partition p_i will be used to mine all frequent patterns beginning with item *i.* While mining p_i, the set of frequent items occurring after item *i* in each transaction of p_i is copied to the next corresponding partition following the above rule. As an example, in the first partition of Figure 9 that contains all transactions beginning with 1, we have one occurrence of transaction 245, 234, 345 and two occurrences of transaction 23. After mining the first partition, transactions 245, 234, 23 will be copied to the second partition and transactions 345 will be copied to the third partition.

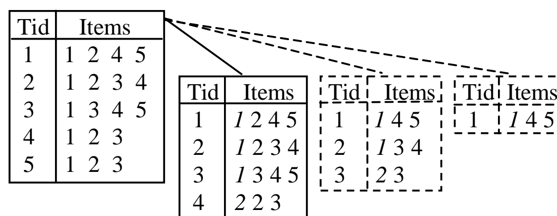


Figure 9. Partitioning the Database (The dashed-boxes partitions are created after mining the first partition)

We have assumed that individual partitions will fit into available memory. However, if any partition is still too large, it can be sub-divided into smaller sub-partitions before mining each sub-partition. For experimental evaluation, we generated large dense datasets ranging from 0.1 to 1 million transactions (increasing in steps of 0.1 million) and 1-11 millions (increasing in steps of 1 million) using the synthetic data generator [17] with similar characteristics to Connect-4. The algorithm was tested for scalability at various support levels. Figure 10 shows the experimental result of CT-PRO for large datasets at support 60 by setting the memory limit to 256MB. We also include the performance of OP for comparison. OP could only mine these datasets up to 7 million transactions, beyond which it failed. We tried to execute FP-Growth to mine these datasets, but surprisingly even for the dense dataset of 0.1 million transactions FP-Growth could not run to completion. These test results show that CT-PRO is scalable for very large dense datasets.

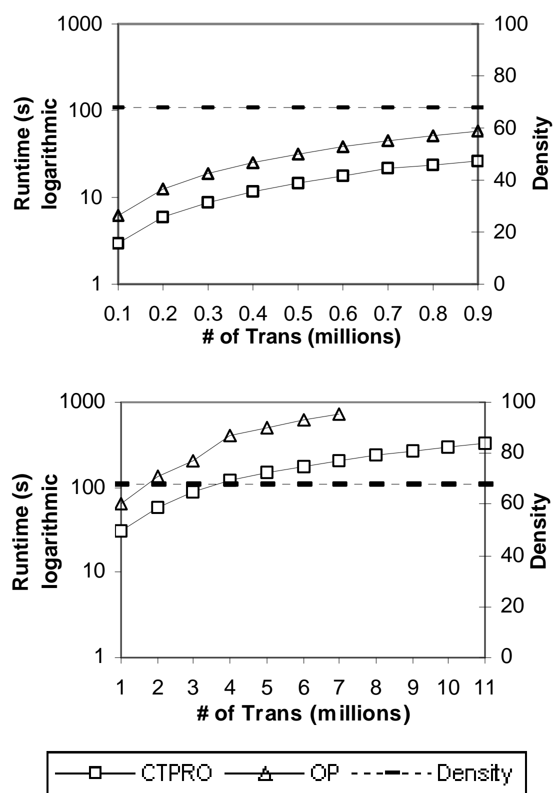


Figure 10. Scalability Testing at Support 60 Density 68%

7 Conclusions

We have presented a new data structure named *CFP-Tree* that is more compact than *FP-Tree* used in the FP-Growth algorithm. The number of nodes in an *FP-Tree* could be up to twice as many as in the corresponding *CFP-Tree* for a given database. We have used the *CFP-Tree* in our new algorithm named CT-PRO for mining complete frequent patterns. CT-PRO *divides* the *CFP-Tree* into several projections represented also by *CFP-Trees*. Then CT-PRO *conquers* the *CFP-Tree* for mining all frequent patterns in each projection.

We have compared the performance of CT-PRO against Apriori, FP-Growth, CT-ITL and OP. For the performance testing, we generated datasets that can show the operational range of all the algorithms. The results show that our algorithm generally outperforms others.

We have extended CT-PRO for mining very large databases and the results show that the algorithm is also scalable to very large datasets.

We have made the executable code of CT-PRO and the datasets generated by us available for research purposes at our website: http://mirror.cs.curtin.edu.au/other/dsrg.

We are currently adapting CT-PRO for parallel mining of large data warehouses in cluster environments and also extending this algorithm for mining frequent patterns from data streams.

Acknowledgments

We thank Jian Pei for the executable code of FP-Growth, Christian Borgelt for providing Apriori and Junqiang Liu for the OpportuneProject program.

References

[1] R. Agrawal, T. Imielinski, and A. Swami, "Mining Association Rules between Sets of Items in Large Databases," Proceedings of ACM SIGMOD, Washington DC, 1993.

[2] B. Liu, W. Hsu, and Y. Ma, "Integrating Classification and Association Rule Mining," Proceedings of ACM SIGKDD, New York, NY, 1998.

[3] K. Wang, X. Chu, and B. Liu, "Clustering Transactions Using Large Items," Proceedings of ACM CIKM, USA, 1999.

[4] R. Agrawal and R. Srikant, "Fast Algorithms for Mining Association Rules," Proceedings of the 20th International Conference on Very Large Data Bases, Santiago, Chile, 1994.

[5] J. Han, J. Pei, Y. Yin, and R. Mao, "Mining Frequent Patterns without Candidate Generation: A Frequent-pattern Tree Approach," *Data Mining and Knowledge Discovery: An International Journal, Kluwer*

Academic Publishers, vol. 8, pp. 53-87, 2004.

[6] R. Agarwal, C. Aggarwal, and V. V. V. Prasad, "A Tree Projection Algorithm for Generation of Frequent Itemsets," *Journal of Parallel and Distributed Computing (Special Issue on High Performance Data Mining)*, 2000.

[7] C. Borgelt, "Efficient Implementations of Apriori and Eclat," Proceedings of the IEEE ICDM Workshop on Frequent Itemset Mining Implementations, Melbourne, USA, 2003.

[8] J. Liu, Y. Pan, K. Wang, and J. Han, "Mining Frequent Item Sets by Opportunistic Projection," Proceedings of ACM SIGKDD, Edmonton, Alberta, Canada, 2002.

[9] W. Cheung and O. R. Zaiane, "Incremental Mining of Frequent Patterns without Candidate Generation or Support Constraint," Proceedings of Seventh International Database Engineering and Applications Symposium (IDEAS2003), Hong Kong, China, 2003.

[10] F. Bodon and L. Ronyai, "Trie: an Alternative Data Structure for Data Mining Algorithms," *Computers and Mathematics with Applications*, 2003.

[11] M. El-Hajj and O. R. Zaiane, "COFI-tree Mining: A New Approach to Pattern Growth with Reduced Candidacy Generation," Proceedings of the IEEE ICDM Workshop on Frequent Itemset Mining Implementations, Melbourne, USA, 2003.

[12] Y. G. Sucahyo and R. P. Gopalan, "CT-ITL: Efficient Frequent Item Set Mining Using a Compressed Prefix Tree with Pattern Growth," Proceedings of 14th Australasian Database Conference, Adelaide, Australia, 2003.

[13] R. P. Gopalan and Y. G. Sucahyo, "Improving the Efficiency of Frequent Pattern Mining by Compact Data Structure Design," Proceedings of the 4th International Conference on Intelligent Data Engineering and Automated Learning (IDEAL2003), Hong Kong, 2003.

[14] M. J. Zaki, "Scalable Algorithms for Association Mining," *IEEE Transactions on Knowledge and Data Engineering*, vol. 12, pp. 372-390, May/June 2000.

[15] J. Pei, J. Han, H. Lu, S. Nishio, S. Tang, and D. Yang, "H-Mine: Hyper-Structure Mining of Frequent Patterns in Large Databases," Proceedings of the 2001 IEEE International Conference on Data Mining (ICDM'01), San Jose, California, 2001.

[16] Blake, C.L. and Merz, C.J. UCI Repository of machine learning databases. 1998. UC, Irvine, CA: Dept of Information and Computer Science. http://www.ics.uci.edu/~mlearn/MLRepository.html

[17] Synthetic Data Generation Code for Associations and Sequential Patterns. http://www.almaden.ibm.com/software/quest/Resources/index.shtml. Intelligent Information Systems, IBM Almaden Research Center.

[18] Frequent Itemset Mining Implementations (FIMI'03) Workshop website, http://fimi.cs.helsinki.fi, 2003.