METHODOLOGY

3. METHODOLOGY

The amount of data stored in databases has increased tremendously with the widespread use of databases and the rapid adoption of information systems and data warehousing technologies. An important type of database that contains huge knowledge of a business is the transaction database. A transaction database contains information about frequently used patterns of potential customers. The process of obtaining this information is called Frequent Pattern Mining and can be discovered using various data mining techniques, like clustering, classification, prediction and association analysis.

Association rules are repeatedly used for identifying frequent patterns from transaction database. Mining frequent patterns from transaction database is the most time consuming process due to a massive number of patterns generated and the large databases that are processed. By intelligently utilizing the limited memory and reducing the number of scans over the database, the performance of associative rule mining for pattern discovery can be improved.

In this research work two solutions are compared for this purpose. The first is to use a CT-Apriori (Compact Tree-Apriori) algorithm and the second is to use CFP-Tree (Compressed FP-Tree) algorithm. Both the algorithms are based on association rules and the working of both these algorithms is explained in this chapter. As associative rule mining forms the basis for both the algorithm, a brief description is provided in the following section. The algorithms are analyzed based on memory usage, time and scalability.

3.1. CT-APRIORI

Association rule mining algorithms consists of two tasks. The first task focus on generating all frequent itemsets that satisfy the user specified minimum support, while the second uses the frequent itemsets generated in the first task to discover all the association rules that meet a user defined confidence threshold. Out of the two tasks, the first task is more

computationally expensive and less straightforward and CT-Apriori is an algorithm that focuses on optimizing the I/O operations in finding frequent patterns.

3.1.1. Compact Transaction Database

This section explains the concepts and properties behind compact transaction database.

Let $I = \{i_1, i_2, \ldots, i_m\}$ be a set of m items. A subset $X \subseteq I$ is called an itemset. A k-itemset is an itemset that contains k items.

Definition 3.1: A transaction database $TDB = \{T_1, T_2, \ldots, T_N\}$ is a set of N transactions, where each transaction T_n ($n \in \{1, 2, \ldots, N\}$) is a set of items such that $T_n \subseteq I$. A transaction T contains an itemset X if and only if $X \subseteq T$.

An example transaction database TDB is shown in Table 1.1 and is reproduced below. In the table $I = \{A, B, C, D\}$, A, B, C and D are itemsets and N = 10.

TABLE 3.1

EXAMPLE TRANSACTION DATABASE TDB

TID	LIST OF ITEMIDs
001	A, B, C, D
002	A, B, C
003	A, B, D
004	B, C, D
005	C, D
006	A, B, C
007	A, B, C
008	B, C
009	B, C, D
010	C, D

The support or occurrence frequency of a pattern A is the percentage of transactions in D containing A: support(A)=||{t | t ∈ D, A ⊆ t}|| / ||{t | t ∈ D}||, where ||X|| is the cardinality of set X. A pattern in a transaction database is called as a frequent pattern if its support is equal to, or greater than a user-specified minimum support threshold, minsup.

Given a transaction database TDB and a minimum support threshold minsup, the problem of finding the complete set of frequent patterns is called frequent-pattern mining. The two most important performance factors of the frequent pattern mining are the number of passes made over the transaction database and the efficiency of those passes. As the data volume increases rapidly these days, the I/O read/write frequency plays an important role for the performance of database mining. Reducing the I/O operations during the mining process can improve the overall efficiency.

Definition 3.2: The Compact Transaction Database (CTDB) of an original transaction database TDB is composed of two parts: head and body. The head of CTDB is a list of 2-tuples (I_n, I_c), where $I_n \in I$ is the name of an item and I_c is the frequency count of I_n in TDB. All items in the head are ordered in frequency-descending order. The body of CTDB is a set of 2-tuples (T_c, T_s), where $T_s \in$ TDB is a unique transaction, T_c is the occurrence count of T_s in TDB and the items in each transaction of the body are ordered in frequency descending order.

The compact transaction database of the example transaction database TDB is shown in Table 3.2.

TABLE 3.2

COMPACT TRANSACTION DATABASE OF TDB

HEAD				
ITEM	C	B	D	A
COUNT	9	8	6	5
BODY				
COUNT	**LIST OF ITEMIDS**			
3	C, B, A			
1	C, B, D, A			
1	B, D, A			
1	C, B			
2	C, B, D			
2	C, D			

All four items in TDB, {A, B, C, D}, are listed in the head with their frequency count, ordered in frequency-descending order, {C:9, B:8, D:6, A:5}. The body consists of 6 unique transactions, instead of 10 in TDB (which is the total transaction count in the compact transaction database). The items in each transaction are ordered in frequency-descending order as well. In the next section, an efficient method to construct the compact transaction database by using a data structure compact transaction tree, denoted as CT-tree is described.

3.1.2. CT-Tree Generation – An example

This section explains the construction of CT for the Compact transaction database shown in Table 3.2. The CT-Tree generation process starts with the creation of the root of a tree called "ROOT". All the other node in the tree consists of two parts: item id and occurrence count of the path from root to this node. The CT-tree is constructed by scanning the example transaction database once. For the first transaction, after sorting the items of this transaction in lexicographic order, the first branch of the tree is constructed as {(A:0), (B:0), (C:0), (D:1)}. The last node (D:1) records the occurrence of the path ABCD. At

the same time, the frequency count of all these items are recorded in a list as [A:1, B:1, C:1, D:1].

For the second transaction, since its ordered item list {A, B, C} shares a common path {A, B, C} with the first branch, no new branch is created, but the occurrence count of the last shared node is incremented by 1 as (C:1). And the frequency count of each item in this transaction is incremented by 1 in the list as [A:2, B:2, C:2, D:1].

For the third transaction, since its ordered item list {A, B, D} shares a common path {A, B} with the first branch, one new node (D:1) is created and linked as a child of (B:0). And the frequency count list becomes [A:3, B:3, C:2, D:2]. The scan of the fourth and fifth transactions leads to the construction of two branches of the tree, {(B:0), (C:0), (D:1)} and {(C:0), (D:1)}, respectively. And the frequency count list becomes [A:3, B:4, C:4, D:4].

After the scan of all the transactions, the complete CT-tree for the example transaction database TDB is shown in Figure 3.1. And the frequency count list becomes [A:5, B:8, C:9, D:6], as shown in the head part of Table II in frequency-descending order.

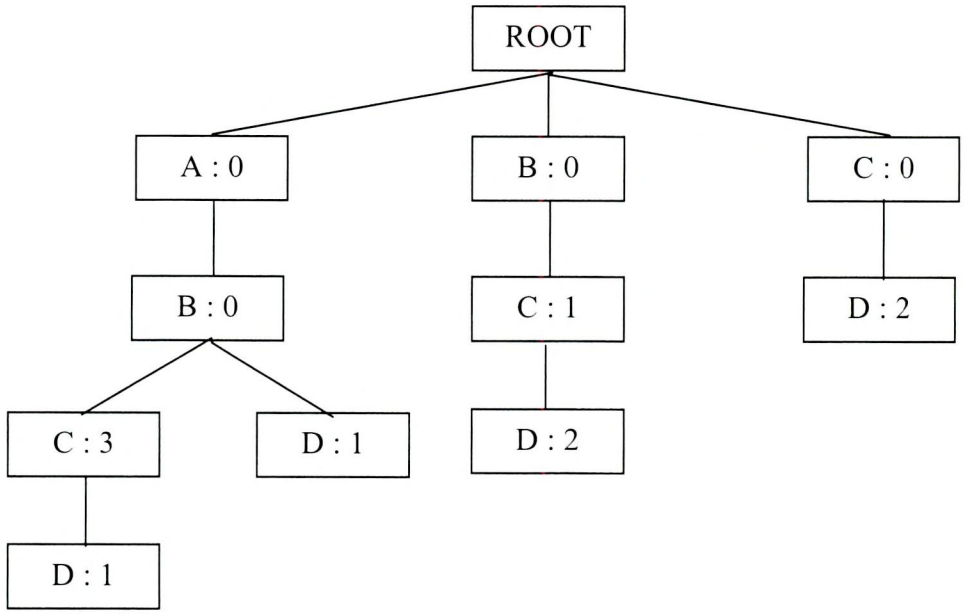


Figure 3.1 : CT-Tree for the database TDB in Table 3.2

After building the CT-tree, the body part of the compact transaction database is constructed and is explained below. For every node v whose count value is greater than 0 in the CT-tree, a unique transaction t is created in the body part of CTDB. The count value associated with the node is recorded as the occurrence count of t and the sequence of items labelling the path from the root to v is sorted in frequency-descending order and recorded as the item list of t. For example, no transaction is created for node A or B in the leftmost path because their count values are 0. Whereas transactions [3 C B A] and [1 C B D A] are created for nodes C and D, respectively, as shown in the first two rows in the body part of Table 3.2.

3.1.3. Algorithm description

The Compact Transaction Tree (CT-tree) of a transaction database TDB can be defined as a tree where each tree node V (except the root of the tree, which is labeled as "ROOT") is a 2-tuple (v, vc) (denoted by v : vc in the tree), where v is an item in TDB and vc is the number of occurrences in TDB of a unique transaction consisting of all the items in the branch of the tree from the root to node V. The algorithm for generating a CT-tree from a transaction database and for generating a compact transaction database from a CT-tree is described in Figure 3.2.

In the first two steps in the above method, the root of an empty CT-tree and a 2-dimension array list are initialized. All items in the original transaction database TDB will be stored in this list along with their support counts after constructing the CT-tree. From step 3 to step 6, a complete CT-tree is built with one database scan, where each transaction T in TDB is sorted and inserted into the CT-tree by calling the procedure insert (T, CT-tree).

Input: Original transaction database TDB;
Output: Compact transaction database CTDB.

```
1: root[CTtree] ← ROOT
2: list[item][count] ← null
3: for each transaction Tn in TDB do
4:          To ← sort items of Tn in lexicographic order
5:          insert(To, CTtree)
6: end for
7: if CTtree is not empty then
8:          list ← sort list[item][count] in count descending order
9:          for each item i in list[item] do
10:                    CTDB ← write i
11:                    CTDB ← write count[list[i]]
12:         end for
13:         startNode ← child[root[CTtree]]
14:         write(startNode, CTDB)
15: else
16:         output "The original transaction database is empty!"
17: end if

procedure insert(T, CTtree)
1: thisNode ← root[CTtree]
2: for each item i in transaction T do
3:          if i is not in list[item] then
4:                    list[item] ← add i
5:          end if
6:          list[count[i]] ← list[count[i]] + 1
7:          nextNode ← child[thisNode]
8:          while nextNode ≠ null and item[nextNode] ≠i do
9:                    nextNode ← sibling[nextNode]
10:         end while
11:         if nextNode = null then
12:                   item[newNode] ← i
13:                   if i is the last item in T then
14:                             count[newNode] ← 1
15:                   else
16:                             count[newNode] ← 0
17:                   end if
18:                   parent[newNode] ← thisNode
19:                   sibling[newNode] ← child[thisNode]
20:                   child[newNode] ← null
21:                   child[thisNode] ← newNode
22:                   thisNode ← newNode
23:         else
24:                   if item i is the last item in T then
25:                             count[thisNode]++
26:                   else
27:                             thisNode ← nextNode
28:                   end if
29:         end if
30: end for

procedure write(node, CTDB)
1: if count[node] ≠ 0 then
2:          count[newTrans] ← count[node]
3:          nextNode ← node
4:          while nextNode ≠ root[CTtree] do
5:                    newTrans ← insert item[nextNode]
6:                    nextNode ← parent[nextNode]
7:          end while
8:          if newTrans is not empty then
9:                    newTrans ← sort newTrans in list order
10:                   CTDB ← write newTrans
11:         end if
12: end if
13: if child[node] ≠ null then
14:         write(child[node], CTDB)
15: end if
16: if sibling[node] ≠ null then
17:         write(sibling[node], CTDB)
18: end if
```

Figure 3.2 : Compact Transaction Database Generator

Then, the list is sorted in frequency descending order and written as the head part of the compact transaction database CTDB, as shown in step 8 to step 12. After calling the procedure write(startNode, CTDB) in step 13 recursively, a unique transaction newTrans is written into the body of CTDB for each node whose count value is not equal to zero in the CT-tree. The occurrence count of newTrans is the same as the count value (step 2 of write), and the item list of newTrans is the sequence of items labelling the path from the node to the root (step 4 to step 7 of write), sorted in frequency-descending order (step 9 of write). Thus, a complete compact transaction database is generated.

3.1.4. CT-Apriori Algorithm

The Apriori algorithm is one of the most popular algorithms for mining frequent patterns and association rules. It introduces a method to generate candidate itemsets C_k in the pass k of a transaction database using only frequent itemset F_{k-1} in the previous pass. The idea rests on the fact that any subset of a frequent itemset must be frequent as well. Hence, C_k can be generated by joining two itemsets in F_{k-1} and pruning those that contain any subset that is not frequent. In order to explore the transaction information stored in a compact transaction database efficiently, the Apriori algorithm is modified and named as CT-Apriori. The pseudo code for CT-Apriori is shown in Figure 3.3. In the pseudo code, $X_{[i]}$ represents the ith item in X. The k-prefix of an itemset X is the k-itemset $\{X_{[1]}, X_{[2]},..., X_{[k]}\}$.

There are two essential differences between this method and the Apriori algorithm.

1) The CT-Apriori algorithm skips the initial scan of database in the Apriori algorithm by reading the head part of the compact transaction database and inserting the frequent 1-itemsets into F_1. Then candidate 2-itemset C_2 is generated from F1 directly, as shown in step 1 - 4 in Figure 3.3.

Algorithm: CT-Apriori algorithm
Input: CTDB (Compact transaction database) and min sup (minimum support threshold).
Output: F (Frequent itemsets in CTDB)

```
1: F1 ← {{i} | i ∈ items in the head of CTDB}
2: for each X,Y ∈ F1, and X<Y do
3:       C2 ← C2 ∪ {X∪Y}
4: end for
5: k ← 2
6: while Ck ≠ θ do
7:       for each transaction T in the body of CTDB do
8:             for each candidate itemsets X ∈ Ck do
9:                   if X ⊆ T then
10:                        count[X] ← count[X] + count[T]
11:                  end if
12:            end for
13:      end for
14:      Fk ← {X | support[X] ≥min sup}
15:      for each X,Y ∈ Fk, X[i]=Y [i] for 1 ≤i≤k and X[k]<Y [k] do
16:            L ← X ∪ {Y[k]}
17:            if ∀J ⊂ L, |J| = k : J ∈ Fk then
18:                  Ck+1 ← Ck+1 ∪ L
19:            end if
20:      end for
21:      k ← k + 1
22: end while
23: return F = UkFk
```

Figure 3.3 : CT-Apriori Algorithm

2) In the Apriori algorithm, to count the supports of all candidate k-itemsets, the original database is scanned, during which each transaction can add at most one count to a candidate k-itemset. In contrast, in CT-Apriori, as shown in step 10, these counts are incremented by the occurrence count of that transaction stored in the body of the compact transaction database, which is, in most of the time, greater than 1.

3.2. COMPRESSED FP-TREE (CFP-TREE)

Another algorithm that is popularly used during frequent pattern mining is the FP-Growth algorithm. It uses pattern growth approaches, which involves a divide and conquer strategy, where the database is divided into several

projections and each projection is mined independently. One main 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. In this section, a new data structure named Compressed FP-Tree (CFP-Tree for short) that is even more compact than FP-Tree is presented.

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. A new algorithm, called CT-PRO, is used to divide the database into several projections and then mine 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 the overhead of creating an extra data structure at each mining step is avoided.

The objective of frequent pattern extraction using CT-PRO is to find all frequent patterns, given an input database D and a support threshold s. 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 (Figure 3.4). The presence or absence of an item in each transaction can be denoted 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 3.4: Binary representation of a transaction database

3.2.1. Compressed FP-Tree

The main objective of compressed FP-Tree (CFP-Tree) is to reduce the size of the FP-Tree size by half. The items are in descending order of their frequency in a CFP-Tree and there is a link to the next node with the same-item-node. 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. The FP-Tree is compressed 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.

An example database is shown in Table 3.3 and Figures 3.5 and 3.6 show 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.

TABLE 3.3

SAMPLE DATABASE

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

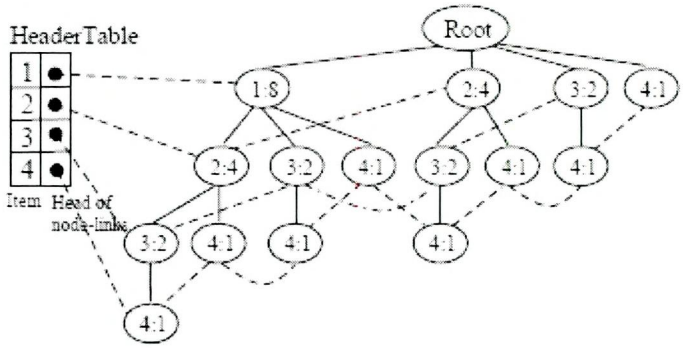


Figure 3.5 : FP-Tree

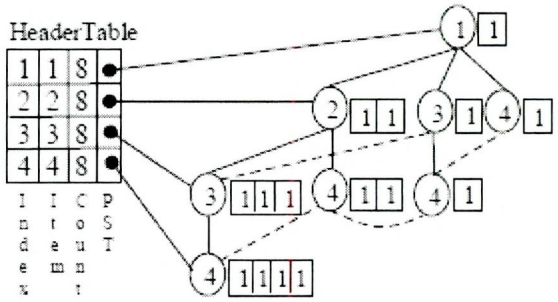


Figure 3.6 : CFP-Tree

To mine the frequent patterns from the transaction, two additional columns are 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.

3.2.2. Construction of CFP-Tree

To mine the frequent patterns from the transaction database shown in Table 3.4a with a minimum support threshold s = 40% (2 transactions), the frequent items has to be determined first 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, only the frequent items of each transaction are selected. In the next step, each item is mapped to its index in HeaderTable and is sorted in ascending order of their index id. After which, they are inserted into the CFP-Tree (Tables 3.4b and 3.4c).

The result of this step is shown in Figure 4. The pointer in HeaderTable acts as the start of the links to other nodes with same item ids (indicated by the dashed lines in Figure 3.7). For illustration, at each node the index of the array is shown and the transaction is represented at each index entry along with its count. However, in the implementation of CFP-Tree, only the second column that represents the count is considered. The algorithm for constructing the CFP-Tree is stored and the procedure is given in Figure 3.8.

TABLE 3.4

MAPPING FREQUENT ITEMS

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

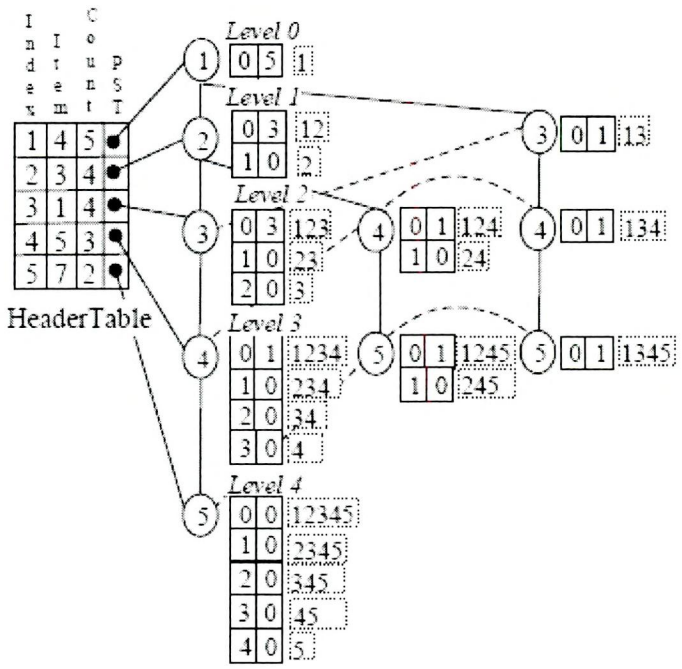


Figure 3.7 : Example Global CFP-Tree

3.2.3. Mining the CFP-Tree using CT-PRO

To mine all frequent patterns in the transaction tree using the CFP-Tree, the pointers in the HeaderTable are used as the starting points. The algorithm starts from the least frequent item and moves to the most frequent so that a larger number of nodes can be pruned in early iterations. This shrinks the tree the tree 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. This is termed as 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 3.9 and the following example illustrates the working of the algorithm on the sample database.

The mining process based on the gobal CFP-Tree shown in Figure 3.7 is discussed in this section. Figure 3.10 shows the local CFP-Tree and local frequent pattern tree at each step during the mining process. The algorithm

starts 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 itemindex 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 14-23). This results with 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 > 2) in projection of itemindex 5, items 4 and 5 are registered in the localHeaderTable with new index id (see Figure 4.11a). 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, the patterns 74(2), 75(2) are obtained as frequent.

After identifying 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, sorted in ascending order of their index id and inserted into the local CFP-Tree (lines 24-32). The first path of itemindex 5 returns nothing. From the second path of index 5, a transaction 14(1) is inserted to the local CFP-Tree and another transaction 14(1) is inserted from the third path of item-index 5. In total, there are 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.

```
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 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 È 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 3.8 : Algorithms for Constructing *CFP-Tree*

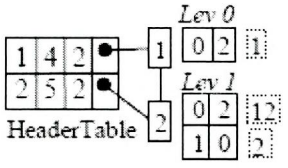
```
1. /* Input: HeaderTable, Global CFP-Tree */
2. Output: frequent patterns */

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

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

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

Figure 3.9 : Mining Frequent Patterns in CT-PRO

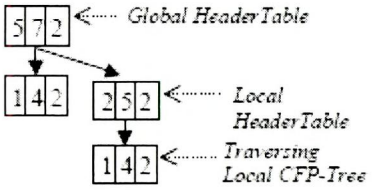
Longer frequent patterns (length > 2) are extracted by calling another procedure (line 11). This procedure (lines 33-45) is an recursive procedure which starts from the least frequent item in the localHeaderTable (line 34), the same-item-node-link is traversed (line 36-40). For each node, the path to the root is traversed counting the other items that are together with the current item. For example, in Figure 3.10a, 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).

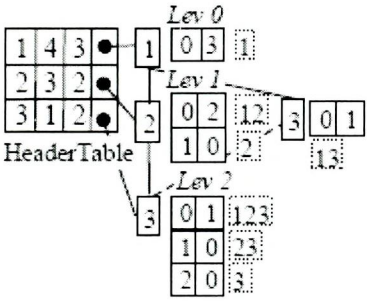
At each step of recursive mining, FP-Growth needs to create a conditional FP-Tree. This overhead adversely affects its performance, as the number of conditional FP-Trees will correspond to the number of frequent patterns. In CT-PRO, for each frequent item f (not frequent itemsets), only one local CFP-Tree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

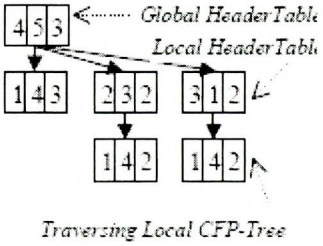


(a) Local FP-Tree 5

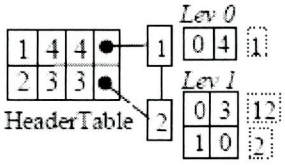


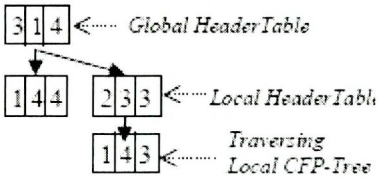
(b) Frequent Patterns in Projection 5



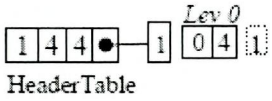


(c) Local FP-Tree 4 **(d) Frequent Patterns in Projection 4**





(e) Local FP-Tree 3 **(f) Frequent Patterns in Projection 3**



(g) Local FP-Tree 2 **(h) Frequent Patterns in Projection 2**

Figure 3.10 : Local CFP-Tree during Mining Process

3.3. FREQUENT PATTERN MINING IN WEB LOG FILES

The process of frequent pattern mining in web log files is shown in Figure 3.11.

The input to the process is the log data. The data has to be preprocessed in order to have the appropriate input for the mining algorithms. The preprocessing eliminates duplicate entries, irrelevant pages and retaining only pages with entries from GET and POST. From the resultant dataset, the IP address is used to identify unique users after session identification. Each session is differentiated using a timestamp of 30 minutes. The Unique URL is assigned a numeric identification number. The web log file is then converted to

a transaction database format with a Tid followed by the URL id's visited by a single user in a session.

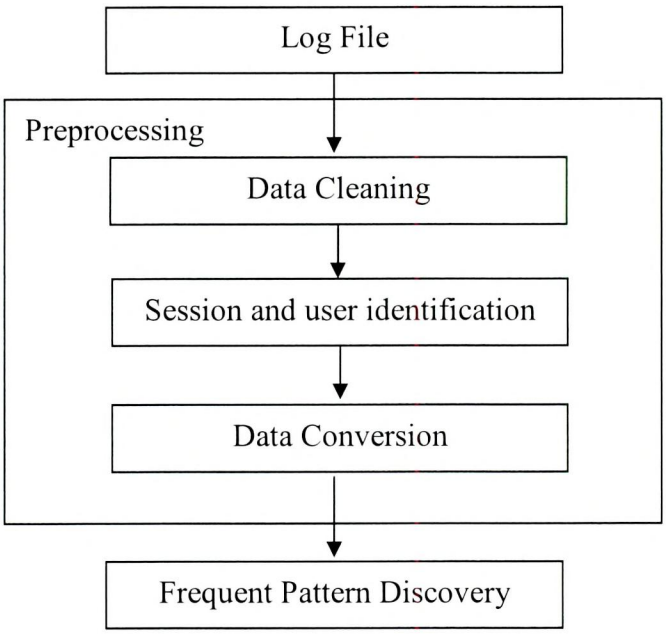


Figure 3.11: Frequent Pattern Mining in Web log data

While considering web log files, the main aim is to find the frequent pages visited at the same time, and to discover the page sequences visited by users. The results obtained by the application can be used to form the structure of a portal, satisfactorily for advertising reasons and to provide a more personalized Web portal.

3.4. CHAPTER SUMMARY

The two techniques proposed to use in frequent pattern mining using association rules were discussed in this chapter. Several experiments were conducted to test the performance of the two techniques selected. The results obtained are tabulated and discussed in the next chapter, Results and Discussion.