CHAPTER 3
MODIFIED FP GROWTH FOR FIM

3.1 Basics of Frequent Itemsets

The formal definition of frequent pattern and association rule mining problems is stated as follows: Let \( I = \{i_1, i_2, i_3, \ldots, i_n\} \) is a set of items, such as products like (computer, CD, printer, papers, ...and so on). Let \( DB \) be a set of database transactions where each transaction \( T \) is a set of items such that \( T \subseteq I \). Each transaction contains a unique identifier called TID. An association rule has the form \( X \rightarrow Y \), where \( X \cap Y = \emptyset \). \( X \) is called the antecedent and \( Y \) is called the consequent of the rule where \( X, Y \) is a set of items or an itemset or a pattern. The number of rows (transactions) containing \( X \) itemset in the given database is given as \( \text{freq}(X) \). The support of an itemset \( X \) is defined as the fraction of all rows containing the itemset, i.e. \( \text{freq}(X) / D \).

The support of an association rule is the support of union of \( X \) and \( Y \), i.e.

\[
\text{Support}(X \Rightarrow Y) = (X \cup Y) / D
\]

The confidence of an association rule is defined as the percentage of rows in \( D \) containing itemset \( X \) that also contain itemset \( Y \), i.e.

\[
\text{Confidence}(X \Rightarrow Y) = \frac{P(X|Y)}{\text{support}(X \cup Y)} = \frac{\text{support}(X \cup Y)/ \text{support}(X)}{D}
\]

An itemset (or a pattern) is frequent if its support is equal to or more than a user specified minimum support threshold. Association rule mining can be refined further by using constraints such as minimum support and minimum confidence.
However, a large number of these rules will be pruned after applying the support and confidence thresholds. Therefore the previous computations will be wasted. To avoid this problem and to improve the performance of the rule discovery algorithm, mining association rules may be decomposed into two phases:

1. Discovering the large itemsets, i.e., the sets of items whose support is above a predetermined minimum threshold and this is known as Frequent Itemsets.
2. Using this large itemsets, association rules for the database is generated that have confidence above a predetermined minimum threshold.

The overall performance of mining association rules is determined primarily by the first step. The second step is easy. After the large itemsets are identified, the corresponding association rules can be derived in a straightforward manner. The main consideration of the thesis is the first step, i.e. to extract frequent itemsets.

### 3.2 FPM Algorithms

There are various algorithms for FIM. Some of the efficient algorithms are:

- Apriori Algorithm
- Eclat Algorithm
- FP growth Algorithm

#### 3.2.1 Apriori Algorithm

Apriori is the very first algorithm for mining frequent items from transactional database. It was given by Agrawal. R and Srikant. R in 1994. It works on the horizontal layout based database. It is based on Boolean association rules which uses generate and test approach. It uses BFS (breadth first search). Using frequent k itemsets, Apriori finds a bigger itemset of k+1 itemset. Apriori
property: All subsets of a frequent itemsets which are non empty are also frequent.

Apriori algorithm states that if a set doesn’t satisfy the minimum support threshold, then all its supersets will fail to meet the minimum support threshold. In the first pass of the algorithm, candidate 1-itemsets is constructed by counting the occurrences of each item in the database. The resulting set is denoted as \( C_1 \). Set \( L_1 \) is generated by pruning the items whose support values are lower than the minimum support value. The resulting set is denoted as \( L_1 \). This property is also known as Apriori Pruning Principle. After the algorithm finds out all the frequent 1-itemsets, it joins the frequent 1-itemsets to construct the candidate 2-itemsets and prunes some candidate 2-itemsets whose support count are below the minimum support count to generate the frequent 2-itemsets. This process is repeated until no further candidate itemsets can be created. Figure 3.1 gives an example of generation of candidate itemsets and frequent itemsets when the minimum support count is 2. This algorithm

- Is easy to implement
- Can be easily parallelized
- Uses large itemset property
Limitations

There are major computational challenges faced by the Apriori algorithm. As the dimensionality of the database increases, it needs to scan the database multiple times iteratively and generate huge number of candidate itemsets. It checks a large set of candidates with pattern matching. As more search space is needed and I/O cost increases the computational cost becomes quite expensive. It is also a tedious workload to go over each transaction to determine support count of the candidate itemsets. Unfortunately, when the dataset becomes large, this algorithm leads to huge loss of time and more occupancy of memory space.
3.2.2  **Eclat Algorithm**

Eclat algorithm is basically a depth-first search algorithm using set intersection. It uses a vertical database layout i.e. instead of explicitly listing all transactions; each item is stored together with its cover (also called TID list) and uses the intersection based approach to compute the support of an itemset. It states that, when the database is stored in the vertical layout, the support of a set can be counted much easier by simply intersecting the covers of two of its subsets that together give the set itself. In this algorithm, each frequent item is added to the output set. After that, for every such frequent item $i$, the $i$-projected database $D_i$ is created. This is done by first finding every item $j$ that frequently occurs together with $i$. The support of this set $\{i, j\}$ is computed by intersecting the covers of both items. If $\{i, j\}$ is frequent, then $j$ is inserted into $D_i$ together with its cover. Reordering is performed in every recursion steps of the algorithm. Then the algorithm is called recursively to find all frequent itemsets in the new database $D_i$. The algorithm has good scalability due to the compact representation.

*Drawbacks*

When the database is very large and the itemsets in the database is also very large, then it is feasible to handle the Transaction-id list. Thus, it produces good results. But for small databases its performance does not scale well.

3.2.3  **FP growth Algorithm**

FP growth uses a combination of the vertical and horizontal database layout to store the database in main memory. It stores the actual transactions from the database in a tree structure and every item has a linked list going through all
transactions that contains that item. This new data structure is called FP-Tree. Steps involved in FPM: First it calculates the support count of each item in the database. Next, the items are ordered in the order the items in the database in support ascending order for the same reasons as before. Next, FP-Tree is formed. Create the root node of the tree, labeled as “null”. For each transaction in the database, the items are processed and a branch is created for each transaction.

Every node in the FP-Tree additionally stores a counter which keeps track of the number of transactions that share that node. When considering the branch to be added to a transaction, the count of each node along the common prefix is incremented by 1, and nodes for the items in the transaction following the prefix are created and linked accordingly. Additionally, an item header table is built so that each item points to its occurrences in the tree via a chain of node-links. Each item in this header table also stores its support. The reason to store transactions in the FP-Tree in support descending order is that in this way, it is hoped that the FP-Tree representation of the database is kept as small as possible since the more frequently occurring items are arranged closer to the root of the FP-Tree and thus are more likely to be shared. From the FP-Tree conditional pattern base is generated. Conditional FP-Tree is formed from conditional pattern base and finally, frequent itemsets are generated.

Advantages

- Mines frequent patterns without candidate generation
- Highly compressed structure because it compresses a large database into a compact FP-Tree structure.
- Avoid costly database scans
- Develop an efficient, FP-Tree based FPM method. It follows a divide and conquer methodology.
- Avoid candidate generation: sub-database test only
### 3.2.4 Comparison

Comparison of the above three algorithms are shown in Table 3.1.

**Table 3.1 Comparison of Apriori, Eclat and FP growth**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Apriori</th>
<th>Eclat</th>
<th>FP growth</th>
</tr>
</thead>
<tbody>
<tr>
<td>Technique</td>
<td>Breadth First Search &amp; Apriori property for pruning</td>
<td>Depth First Search and intersection of transaction id to generate candidate itemset</td>
<td>Divide and conquer</td>
</tr>
<tr>
<td>Approach</td>
<td>Horizontal layout based algorithm</td>
<td>Vertical Layout based algorithm</td>
<td>Projected layout based algorithm</td>
</tr>
<tr>
<td>Database</td>
<td>Suitable for sparse datasets as well as dense datasets</td>
<td>Suitable for medium and dense datasets but not for small datasets</td>
<td>Suitable for large and medium datasets</td>
</tr>
<tr>
<td>Database scan</td>
<td>Database is scanned for each time a candidate item is generated</td>
<td>Database is scanned few times</td>
<td>Database is scanned two times only</td>
</tr>
<tr>
<td>Data storage format structure</td>
<td>Horizontal array</td>
<td>Vertical array</td>
<td>FP-Tree(Horizontal tree)</td>
</tr>
<tr>
<td>Memory utilization</td>
<td>Due to large amount of candidates produced it requires large memory space</td>
<td>Requires less amount of memory compared to Apriori if items are less</td>
<td>Due to compact structure and no candidate generation, it requires less space</td>
</tr>
</tbody>
</table>
### Advantage

Easy to implement

As support count information will be obtained from the previous itemset, there is no need to scan the database each time a candidate itemset is generated.

The Database is scanned only two times. No candidate generation.

### Drawbacks

Too many candidate itemset generations. So more memory is required.

Requires virtual memory to perform the transformation.

Expensive.

<table>
<thead>
<tr>
<th>Advantage</th>
<th>Easy to implement</th>
<th>As support count information will be obtained from the previous itemset, there is no need to scan the database each time a candidate itemset is generated</th>
<th>The Database is scanned only two times. No candidate generation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Drawbacks</td>
<td>Too many candidate itemset generations. So more memory is required.</td>
<td>Requires virtual memory to perform the transformation.</td>
<td>Expensive.</td>
</tr>
</tbody>
</table>

Thus from the table 3.1 it is found that Apriori uses join and prune method, Eclat works on vertical datasets and FP growth constructs the conditional frequent pattern tree which satisfies the minimum support. The major drawback of the Apriori algorithm is that it produces too many candidate itemset generations. So more memory is required. It is very much expensive to scan large database. A true reason of Apriori failure is that it lacks an efficient processing method on database of small in size. Eclat is more efficient than Apriori algorithm in terms of running time but it requires virtual memory. Whereas FP growth is better than Apriori and Eclat in terms of execution time, which is shown in Figure 3.2 and it is more scalable. The comparison of the three algorithms are made with the database consisting of 5,000 transactions.
Modified FP growth Algorithm

The first step in FP growth is generating Frequent itemsets. A formalized and an efficient algorithm named support count tree has been proposed to find out Frequent-1 itemsets. This algorithm has been embedded in FP growth algorithm. So far a well defined algorithm has not been proposed to calculate Frequent 1-itemsets. In the proposed work an efficient tree structure has been proposed which finds the Frequent 1-Itemsets quickly and efficiently, which in-turn speeds up the generation of Frequent Itemsets of the entire database.
Assume all items in the database are numbered. If it is not numbered, then number each item in the database. Suppose the transactional database which is being considered consist of 45 items, then start the numbering from 1 to 45. Next step is to form a support count tree.

**Steps for generating the support count tree:**

Step 1: Find the mid value of the entire item set and make it as the root node.

Step 2: Items on the left side of the mid value forms the left sub tree of the root node and items on the right side of the mid value forms the right sub tree of the root node.

Step 3: With respect to the left sublist find the mid value and repeat step 1 and 2 until all items are included in the tree.

Step 4: With respect to the right sublist find the mid value and repeat step 1 and 2 until all items are included in the tree.

Step 5: Now scan each and every transaction from the transactional database.

Step 6: When each transaction is scanned search each item in the transaction from the support count tree. If an item is found, increment the respective count variable of that item.

Step 7: Repeat step 6 until all the transactions have been scanned.

Thus, from the count variable of each item, their support counts are known.

An example has been illustrated below on how to form an initial support count tree with 15 items using the above algorithm.
Step 1: Consider 1,2,3,4,5,6,7,8,9,10,11,12,13,14,15 item no for 15 items. Of these 15 items 8 is the mid value and so it is made as the root node.

Step 2: Left side of item 8 forms the left sub-list and right side of item forms the right sub-list.

Step 3: From the left sublist find the mid value. It is item 4 and link it as the left subtree of node 8. From the right sub-list find the mid value. It is item 12 and link it as the right subtree of node 8.

The above steps are repeated recursively until all the nodes are connected to the tree which is shown in Figure 3.3.
For even number of items, mid value is calculated by adding the position of the first and the last element and then dividing the result by 2, which is shown in algorithm 3.1. For example, consider there are 200 items in the dataset. Then
the middle value will be 100.5. Mid value is rounded-off and taken as 101. Left subtree will have 100 values and the right subtree will have 99 values. Thus, this algorithm invariably holds good for both even and odd number of items.

*Variables used:*

Min-points to the first item in the list
Max-points to the last item in the list
N-total number of items in the database
Let k be any item in the transaction.
Val – node value of the support count tree.
Y is the pointer pointing to the structure node. Each node consists of four fields.
Value(name of the node), count, left link and right link.
Right() – points to the right subtree
Left()-points to the left subtree
For each item in the transaction performs the following operation:

*Pseudocode for support count tree formation:*

```
SCTree(min, max)
1. t = (min+max) / 2
   Val -> y = t

2. For the left sublist
   Recursively call SCTree(min, (t-1))

3. For the right sublist
   Recursively call SCTree((t+1), max)
```

Algorithm 3.1: Support Count Tree formation
Steps involved in finding the support count from the support count tree:

Step 1: If the item to be searched is equal to the root node, then increment its count variable and search for the next item in the transaction else go to step 2.

Step 2: Check whether the value of the item to be searched is lesser than the root node. If yes, go to step 3 else go step 4.

Step 3: Check whether the root node value of the left sub tree is equal to the value of the item to be searched. If yes, repeat step 1 and 2.

Step 4: Check whether the root node value in the right sub tree is equal to the value of the item to be searched. If yes, repeat step 1 and 2.

Repeat the above steps until all items are searched.

Pseudocode to find the support count from the support count tree is given in algorithm 3.2. Initially assign the root node address to y.

```
SupportCount(y, k)
1. If val -> y = k
Then increment the count value of y

2. else if val->y < k
then y = right(y)
call recursively SupportCount (y, k)

3. else y = left(y)
Call recursively SupportCount (y, k)
Count value of each node gives the frequent-1 itemsets.
```

Algorithm 3.2: Calculation of Support Count
Steps for FIM using Modified FP growth:

1. After finding the frequent-1 itemsets remove the item which does not satisfy the minimum support. Sort the frequent items in each transaction in the descending order.

2. Create an FP-Tree with T as the root and label it as “null”. It also consists of a set of item prefix subtrees as the children of the root. Each node in the subtree consists of three fields -item-name, count, and node-link. A frequent-item header table is maintained for efficient access of FP-Tree. It consists of three fields namely -item-name, support count and head of node-link.

Algorithm for construction of FP-Tree:

Input: Transaction DB, minimum support threshold.
Output: FP-Tree

1. Let F be the set of items in the transaction. Initially, the support value is calculated using the support tree algorithm. Sort F in support order as prefix in each transaction.

2. Create the root T of an FP-Tree, and label it as "null".

3. Let the item list be [p|P], p is the first item and P is the remainder element in the list. For each item list call insertTree(Items, T);

4. call function insertTree([p|P], T). if T has child N and N.itemName = p.itemName then N.count++;
else create node N = p, N.count=1, be linked to T, node-link to the nodes with the same itemName;
if P is nonempty then call insertTree(P, N);

5. Starting at the bottom of frequent-item header table in the FP-Tree, by following the link of each frequent item traverse the FP-Tree.

6. Accumulate all of the transformed prefix paths of that item to form a conditional pattern base.
7. For each pattern base  
   - Accumulate the count for each item in the base  
   - Construct the conditional FP-Tree for the frequent items of the pattern base.
8. Repeat step 6 and 7 for each Frequent Item.
9. Repeat the above process on each newly created conditional FP-Tree until the resulting FP-Tree is empty, or it contains only one path

Algorithm for FP growth:

Input: FP-Tree, minimum support threshold, without DB.
Output: The complete set of Frequent Itemsets.
Method: Call FP growth (FP-Tree, null)
Procedure FP growth (Tree, α) {
   1. if P contains single path in tree then
   2. for each combination (denoted as β) of the nodes in P do
   3. generate pattern β \( \cup \) α with support as minimum support in β
   4. else for each a_i in the Header Table of Tree do {
   5. generate pattern β = α \{i\} \( \cup \) α with support = α_i.support
   6. construct β ’s conditional pattern base and β ’s conditional FP-Tree Tree β ;
   7. if Tree β \( \neq \) null then
   8. call FP growth (Tree β, β ); }
}
The above algorithm is used to mine frequent itemsets from FP-Tree.

3.4 Experimental Results

This support count tree has been used in the FP growth algorithm for the generation of Frequent Itemsets. The data set was taken from T20I7D500K. Number of transactions in this dataset is 5,00,000. The following experiment is carried out by varying the number of transactions taken from the
above dataset. Size of 15K dataset is 861KB, 25K dataset is 1.39MB, 50K dataset is 2.80MB and 75K dataset is 4.20MB. Experimental result shows that the run time of FP growth algorithm is more when compared to FP growth algorithm with support count tree which is shown in Figure 3.4 and Table 3.2.

![Performance Chart]

**Figure 3.4 Performance of FP growth and FP growth with support count tree**

**Table 3.2 Execution time of FP growth and FP growth with support count tree**

<table>
<thead>
<tr>
<th>S.No</th>
<th>Number of transactions(k)</th>
<th>Execution Time (sec) Of FP growth</th>
<th>Execution Time (sec) Of FP growth with support count tree</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>15</td>
<td>11</td>
<td>10</td>
</tr>
<tr>
<td>2</td>
<td>25</td>
<td>13</td>
<td>11</td>
</tr>
<tr>
<td>3</td>
<td>50</td>
<td>18</td>
<td>13</td>
</tr>
<tr>
<td>4</td>
<td>75</td>
<td>23</td>
<td>15</td>
</tr>
</tbody>
</table>
In Support count tree the items are kept in the sorted order. When an item value has to be incremented it is enough to search the item by traversing from root to leaf by making comparisons to the items stored in the tree. If the value to be searched is smaller than the root node then searching is continued in the left subtree otherwise searching is continued in the right subtree. Each comparison prunes half of the tree, so that the time taken to increment the count value of an item is proportional to the logarithm of the number of items stored in the tree. Thus the time complexity of support count tree to search an element and increment the count value of an item is $O(\log(n))$, where $n$ is the number of items. Whereas it takes $O(n)$ times in FP growth. Thus, to calculate the support count of each item, Support count tree is better than FP growth.