CHAPTER 5

EFFICIENT CLOSED ITEMSET MINING

5.1 INTRODUCTION

A new scalable algorithm for discovering closed frequent itemsets is presented in this chapter. The closed frequent itemsets are lossless and compact representation of all the frequent itemsets that can be mined from a transactional database. A new algorithm called ECIM (Efficient Closed Itemset Mining) is proposed in this chapter. Section 5.2 presents proposed methodology to discover closed frequent itemsets. Section 5.3 provides problem definition. Section 5.4 develops an efficient algorithm, named ECIM to find all Closed Frequent itemsets. Section 5.5 provides the preparation of data, performance measures, and experimental setup. Section 5.6 presents thorough experimental results and discussions. Section 5.7 concludes this chapter.

5.2 PROPOSED METHOD

Association rule mining is a useful data mining technique to find frequent itemsets and generate useful association rules (Tan et al 2012). A frequent itemset is closed if none of its proper supersets have the same support. All closed frequent itemsets contain complete information to generate association rules.

The frequent closed itemset mining algorithms proposed by Grahne & Zhu (2005), Burdick et al (2001) and Wang et al (2003) must check the
closure at the end of the process or during the process. The closure checking verifies whether a newly found itemset is a subset or a superset of an already found frequent closed itemset with the same support. For this purpose, it is necessary to create a structure to store all frequent closed itemsets and closed itemset candidates. The closure checking step of the existing closed itemset mining algorithms takes more execution time and requires additional memory space for generating, storing and removing non-closed itemsets.

The proposed scheme in this research work completely eliminates the closure checking step during the closed itemset generation. The proposed algorithm discovers the closed itemsets efficiently without closure checking and unnecessary storage structure. The proposed algorithm ECIM will not generate unnecessary conditional FP-trees. The proposed approach use an attribute called record in the node of FP-tree to keep necessary information in each branch of the tree such that redundancy can be avoided.

5.3 PROBLEM DEFINITION

A transaction database TDB is a set of transactions, where each transaction, denoted as a tuple < tid, X >, contains a set of items (i.e., X) and is associated with a unique transaction identity tid. Let I = \{i_1, i_2, i_3, ..., i_n\} be the complete set of distinct items appearing in TDB. An itemset Y is a non-empty subset of I and is called an k-itemset if it contains k items. An itemset \{x_1, x_2, ... x_k\} is also denoted as x_1 ... x_k. A transaction < tid, X > is said to contain itemset Y if Y \subseteq X. The number of transactions in TDB containing itemset Y is called the support of itemset Y, denoted as sup(Y). Given a minimum support threshold, min_sup, an itemset Y is frequent if sup(Y) \geq min_sup.
**Definition for Frequent Closed itemset:** An itemset \( Y \) is a frequent closed itemset if it is frequent and there exists no proper superset \( Y' \supseteq Y \) such that \( \text{sup}(Y') = \text{sup}(Y) \).

### 5.4 ECIM ALGORITHM

The property of FP-tree is, mining itemsets in bottom-up order of its supports. It is not possible to find supersets of the found frequent closed itemsets by using this property. If it can avoid generating the subsets of known frequent closed itemsets, frequent closed itemsets can be generated directly.

According to the properties of FP-tree and closed itemset, it is easy to find that if an itemset is a subset of the other known itemset and both of their prefix items have the same support; then they both found in the same path. Therefore, if a branch has more than two items that have the same support, it may generate subsets with the same support. In order to avoid generating these subsets, it should keep off items which have been processed with the same support in the same path and avoid constructing them again.

A record is maintained in ECIM to identify if an item has been processed before. If the nodes of an item have the same item in the record, it means the item has completed the process. Using this idea, subsets with the same support cannot be generated. Thus, it can speed up the run time by not building the conditional FP-tree for invalid items. Figure 5.1 shows the framework of the proposed ECIM algorithm.
Figure 5.1 Framework of ECIM algorithm

1. Scan Database
2. Find Frequent Items using support threshold Value
3. Find Frequent Items Using Support threshold Value
4. Construct Hash Table for Frequent Items
5. Again Scan Database and Build FP-Tree
6. Link Hash Table and FP-Tree
7. Collect Items Those items does not have Prefix Item Path
8. Using Conditional Path base, set support threshold and Record of Prefix Node
9. Collect Similar Items in their record
10. Build the header table & Construct FP-Tree with Remaining Items
11. Assign next item to prefix (Repeat till the last item)
12. Item sets those who have no items in their record.
13. Frequent Closed item sets.
Algorithm: ECIM

Step 1:

Scan the database to find the counts of all items. Find all frequent items by using the min_sup and sort them in decreasing order of their supports. Build the header table with item-name, support-count and head-of-node-link fields.

Step 2:

Scan the database again to build the FP-tree, in which each node has four fields: item-name, count, node-link and record, according to the order of the header table. Each node with the same item-name is linked together.

Step 3:

According to the order of items in the header table, each item is used as the prefix item one at a time. If all nodes of the prefix item do not have the same items in their records, get all paths containing the prefix item being linked from the header table. When getting a path, the records in this path are registered if the prefix item is a leaf of the tree. Then set the support and record of the prefix node as the support and record of all items in the path to form the conditional pattern base.

Step 4:

Prune the items which have the same item in their records and the items which are not frequent from the conditional pattern base. Then, use the remaining items to build the header table. Finally, create the conditional FP-tree with the header table. Set the next item in the header table as the prefix
item. Use Step 5 to process the conditional FP-tree. Repeat Step 3 and Step 4 until the last item of the header table is processed.

**Step 5:**

According to the order of items in the header table of the conditional FP-tree, each item is used as the prefix item one at a time. Repeat Step 3 and Step 4 to create the conditional FP-tree for each prefix item until the conditional FP-tree becomes a single path.

**Step 6:**

Register the records for this path and obtain all itemsets whose prefix items have no item in their records. It can get closed itemsets from the single path with the prefix items of the conditional FP-trees which have been processed before.

**Step 7:**

After processing the last item in the original header table, it generates all frequent closed itemsets.

### 5.4.1 Checking Non-Closed Items

If the support of a prefix item is equal to the support of the parent item, it means the item has completed the process. After these items have been processed, the proposed approach uses the record of the node to register items that have the same support as their child items in the same branch. Using an array as a record in a node of FP-tree will waste much space. Instead, the linked list technique is used to store items. When considering an item if it is necessary to build the conditional FP-tree, simply check if the
records in all the nodes of this item have the same item. If they all have the same item, there is no need to process again because it has been done before. So itemsets generated from this item are not closed.

**Definition-1**

If all nodes of an item have the same items in their records, the itemsets generated by this item are not closed.

Let X be a prefix item and all the X nodes of the tree have the same item Y in their records. Assume X may generate a closed itemset, then the support of the itemset generated by X will not equal to the support of the superset generated by XY. Because all records of X contain Y, the itemsets generated by X must be subsets of the itemsets generated by XY, and their support must be the same. It contradicts the assumption. So X cannot generate any closed itemset.

After building the FP-tree, each item is considered as a prefix item following the order of items in the header table. When it searches the related paths of a prefix item by using the node-links of the header table, the records of these paths are completed.

The procedure to keep track of the record:

- When the node-link of a prefix item is linked to the first node from the header table, it can find the first path which contains the prefix item.

- When searching this path upward to look for items in the path, it can check if its child nodes have a total count that is equal to its count. If their counts are not the same, it does not register
anything. Otherwise, if the node has only one child node, it can register the item and record its child node in its record. If the node has more than one child node, it should check if any node has the same items in item-name or records. If each branch has the same items, it will register the item in the record of the node. Otherwise, nothing is registered. Repeat this process to the root until every node of the prefix items of the header table has been processed. If a node of a prefix item is not at the bottom of the path, it means this path has been registered before.

![Figure 5.2 Part of FP-Tree](image)

The Figure 5.2 is a part of FP-tree and shows that while searching the prefix item ‘p’, it can find the path c-a-m-p. The path is processed as shown in Figure 5.3. First, go up and find the item ‘m’, then register ‘p’ in the record of node ‘m’ because it has the same support as its child ‘p’. The next item ‘a’ has two child nodes, where their total support is equal to the
support of ‘a’. Since none of their item and record is the same, it does not register anything. Then it reaches item ‘c’. Because item ‘c’ has the same support as item ‘a’, its record is updated with item ‘a’.

![Figure 5.3 Path Process](image)

Next, the prefix item ‘m’ is searched and the result is shown in Figure 5.4. Item ‘b’ is the first item above item ‘m’ and its support is equal to the support of item ‘m’, so item ‘b’ is updated with a register of item ‘m’. The next item ‘a’ has two child nodes, where their total support is equal to the support of ‘a’. Since they both have a record of ‘m’, item ‘m’ is registered in node ‘a’. Because node ‘a’ adds a record ‘m’, the parent node ‘c’ is also updated with a register of ‘m’ to its node because the pair has the same support.
Usage of records:

- When searching the paths for a prefix item, it will set the support and records of the prefix item to all items in a path. If each record of the prefix item does not have the same items, the process of this item will continue, else it will stop.

- When getting the conditional pattern base of a prefix item, it can prune the infrequent items. An item will be pruned if all of its nodes have the same item(s) in their records, because they cannot generate any closed itemset by Definition-1. When getting a single path in the conditional FP-tree, it is easy to know if the itemsets have to be generated by the records.
The tree of Figure 5.4 is used to show a simple example. With a prefix item ‘m’, it can get two paths, c-a-m and c-a-b-m. The conditional pattern base has two prefix paths, ca: 2/p (ca: 2 with a record ‘p’) and cabm: 1. Because no item has the same records, it can derive m’s conditional FP-tree, <c:3, a:3>. If the prefix item is ‘a’ it can get a path ‘ca’. Because the conditional pattern base only has a path c: 3/m, it is unnecessary to generate a’s conditional FP-tree.

It is easy to prove that the proposed method will not generate ‘more or less itemsets’ by the following Definition 2.

**Definition 2:**

The proposed ECIM cannot generate non-closed itemsets or miss any closed itemsets. ECIM uses a record to identify if an item with the item in its record can generate a closed itemset by Step 3 and Step 4. CLOSET+ (Wang et al 2003) checks a prefix itemset from the two level hash indexed result tree to identify if the itemset can generate a closed itemset. This shows that ECIM can generate all closed itemsets as CLOSET+ does. Since CLOSET+ cannot generate extra itemsets that are not closed and miss any closed itemsets the proposed algorithm ECIM cannot generate extra or less itemsets.

**5.4.2 Example of Applying ECIM**

The proposed algorithm ECIM is explained using the database in Table 5.1. First the database is scanned. After scanning the database, the algorithm finds all frequent items with \( \text{min \_ sup} = 2 \). The supports of the items are a:3, c:4, f:4, m:3, p:3, b:3. Then sort these items based on support in decreasing order to become f:4, c:4, a:3, b:3, m:3, p:3. When the transactions are inserted into the FP-tree, items in each transaction are sorted in the order. The sorted order of items is shown in the last column of Table 5.1.
Table 5.1 Sample transaction database

<table>
<thead>
<tr>
<th>Tid</th>
<th>Items</th>
<th>Order based on support of frequent item list</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>a, c, f, m, p</td>
<td>f, c, a, m, p</td>
</tr>
<tr>
<td>200</td>
<td>a, c, d, f, m, p</td>
<td>f, c, a, m, p</td>
</tr>
<tr>
<td>300</td>
<td>a, b, c, f, g, m</td>
<td>f, c, a, b, m</td>
</tr>
<tr>
<td>400</td>
<td>b, f, i</td>
<td>f, c, a, b, m</td>
</tr>
<tr>
<td>500</td>
<td>b, c, n, p</td>
<td>c, b, p</td>
</tr>
</tbody>
</table>

The header table is created with sorted items to find each item in all branches easily. This is shown in Figure 5.5.

![Header Table](image)

**Figure 5.5 The FP-tree for Table 5.1**

The FP-tree is build after scanning the database again this is shown in Figure 5.5. All nodes which have the same item-name are linked from the header table. After constructing the FP-tree, it can be traversed to find all frequent closed itemsets. According to the FP-growth method, it can take item
‘p’ as a prefix itemset to get two paths, f-c-a-m-p and c-b-p, by using the node links of item ‘p’ in the header table.

When a node is linked from the header table, it will search the path to find what all items in this path are. At this time, it will register items in the record of each node if a node has the same support as the items below it.

In Figure 5.6, it registers ‘p’ in the record of item ‘m’ because item ‘m’ has the same support as prefix item p in the path f-c-a-m-p. Similarly, item ‘c’ has a record ‘a’ in its node. In the path c-b-p, it registers ‘p’ in the node ‘b’ and registers ‘pb’ in the node ‘c’. The conditional pattern base of prefix item ‘p’ has fcam: 2 and cb: 1 to create the conditional FP-tree shown in Figure 5.8.

![Figure 5.6 The recorded FP-tree with prefix item ‘p’](image)

Then it makes item ‘m’ as a prefix of the FP-tree. When searching the linked node of item ‘m’, it registers the records for nodes with item ‘m’ having no child. There are two paths with prefix ‘m’, f-c-a-m and f-c-a-b-m.
In Figure 5.7, the node ‘m’ in path f-c-a-m has a child, so it is unnecessary to register records in this path. In path f-c-a-b-m, it registers ‘m’ in node ‘b’, registers ‘a’ in node ‘c’ and adds ‘m’ to node ‘c’.

Figure 5.7 The recorded FP-tree with prefix item ‘m’

Figure 5.8 The conditional FP-tree for prefix ‘p’
The conditional FP-tree only contains frequent items and it also has a header table. We can see only one path in this tree. Following the bottom-up order in the header table, we first make item ‘m’ as a prefix item, and find what items are above it. At this time, we register the record in each node of this tree, as shown in Figure 5.8. It can find a frequent closed itemset cfamp: 2 in this tree with prefix ‘m’. Then it makes item ‘a’ as a prefix. Because it has an item ‘m’ in its record of the only one node, one knows it has been mined with prefix ‘m’. It does not generate itemsets with prefix ‘pa’ similarly for the prefix ‘pf’ also. Finally, item ‘c’ is the last item and its record is empty. So we can find cp: 3 as a frequent closed itemset. Because ‘c’ is the last item and the support of ‘c’ is equal to the support of ‘p’, p: 3 is not a closed itemset. At this time, all frequent closed itemsets with prefix ‘p’ have been found.

![Figure 5.9 The conditional FP-tree for prefix ‘m’](image-url)
The conditional pattern base can form a conditional FP-tree as shown in Figure 5.9. Then register the records in the tree when it makes item ‘a’ as a prefix item. It can find a frequent closed itemset fcam: 3, and itemsets with prefix items ‘c’ and ‘m’ are not generated. Because ‘f’ is the last item and the support of ‘f’ is equal to the support of ‘m’, m: 3 is not a closed itemset. At this time, all frequent closed itemsets with prefix ‘m’ have been found.

The third item ‘b’ is used as a prefix item. It can get three paths, fcab, fb and cb. Each one of the paths fcab and cb has a child and the supports in path fb are not the same, so it is unnecessary to register records of the tree. The conditional pattern base, fca: l/m, f: 1 and c: l/p, builds the conditional FP-tree as shown in Figure 5.10. According to the header table, item ‘c’ is the first prefix item. It is unnecessary to register records here because no support
is the same in each path. It can find two paths fc: 1 with record ‘m’ and c:1 with record ‘p’. Only item ‘c’ is frequent and the records of two paths are not the same, so cb: 2 is a frequent closed itemset. Since ‘f’ is the last item of the header table and it has no record, fb: 2 is a frequent closed itemset. Because ‘f’ is the last item and the support of ‘f’ is not equal to the support of ‘b’, b:3 is a frequent closed itemset. At this time, all frequent closed itemsets with prefix ‘b’ have been found.

   Because prefix item ‘a’ has only one path and it has a record ‘m’, it is unnecessary to create the conditional FP-tree with prefix item ‘a’. The prefix ‘a’ will not generate any closed itemset.

   For prefix item ‘c’, it has two paths, fc and c. The conditional pattern base, f: 3/am, and the other path have nothing, so it is unnecessary to create the conditional FP-tree for prefix item ‘c’ with item ‘f’. Then c: 4 is a frequent closed itemset.

   Finally, the last item ‘f’ has no record, so f: 4 is a frequent closed itemset. At this time, the mining process is completed.

5.5 EXPERIMENTAL STUDY AND PERFORMANCE EVALUATION

   This section presents a thorough analysis of the performance of the proposed ECIM algorithm on real datasets and compare its performance with the existing CLOSET+ and CHARM. The ECIM algorithm, CLOSET+ and CHARM algorithms implemented with Java version jdk 1.6. All the
experiments were conducted on an Intel i3 processor system with 4GB of RAM and the operating system is Windows-7.

5.5.1 Data Synthesis

To evaluate the run-time performance and memory usage real datasets are used. Table 5.2 shows the datasets and its characteristics used for the experiments.

Table 5.2 Real datasets

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Number of transactions</th>
<th>Number of Items</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mushroom</td>
<td>8124</td>
<td>120</td>
</tr>
<tr>
<td>Chess</td>
<td>3196</td>
<td>74</td>
</tr>
<tr>
<td>Accidents</td>
<td>11291</td>
<td>41</td>
</tr>
</tbody>
</table>

The proposed ECIM, CLOSET+ and CHARM tested on Mushroom dataset shown in the Table 5.2. The performance analysis in terms of runtime and memory usage is shown in the Figures 5.11 and 5.12 respectively on Mushroom dataset.

The proposed ECIM, CLOSET+ and CHARM tested on chess dataset shown in the Table 5.2. The performance analysis in terms of runtime and memory usage is shown in the Figures 5.13 and Figures 5.14 respectively on Chess dataset.

The proposed ECIM, CLOSET+ and CHARM tested on Accident dataset shown in the Table 5.2. The performance analysis in terms of runtime is shown in the Figures 5.15 and 5.16. Memory usage is shown in the Figure 5.17 on Accidents dataset.
Figure 5.11 Runtime on mushroom dataset
Figure 5.12 Memory usage (in MB) on mushroom dataset
Figure 5.13  Runtime on chess dataset
Figure 5.14 Memory usage (in MB) on chess dataset
Figure 5.15 Runtime on accidents dataset
Figure 5.16 Runtime on accidents dataset
Figure 5.17 Memory usage (in MB) on accidents dataset
5.5.2 Experiment Results and Discussion on Real Datasets

The experiment results on chess dataset in terms of execution time performance and memory usage is shown in Figure 5.13 and 5.14. The Figure 5.15 presents the performance analysis with larger minimum support threshold and Figure 5.16 presents with smaller support on accidents dataset.

It is observed from the experimental results the execution time of proposed ECIM method is better than CLOSET+ and Charm. When the minimum support is lower, the performance of proposed method gets better than CLOSET+ and Charm. It is so because; the number of closed itemset candidates is very big when the minimum support is very low. CLOSET+ needs more time to compare a lot of itemsets for closure checking. Since ECIM does not need to compare many itemsets, it can get better performance in lower minimum supports. When the support increases, the number of candidates becomes smaller. Then the run time of CLOSET+ approaches the run time of ECIM. The experimental results on real datasets show that the proposed ECIM, on average 4 times faster than CLOSET+ for lower minimum support threshold. On average ECIM is 1.5 times faster than CLOSET+.

Figure 5.12, 5.14 and 5.17 shows the memory consumption on mushroom, chess and accidents datasets respectively. The performance of ECIM is 1.3 times better than CLOSET+ on average. Because ECIM does not need to store candidates, it takes less memory space. Although noting records may take up some memory space, it is still less than the space of storing all candidates. Besides, the storage used to store candidates may be larger than the original FP-tree in the worst case. Since not every record needs to be the register items, ECIM takes less space than CLOSET+ and Charm. When the minimum support is higher the memory usage is less. This is because that the frequent items are reduced quickly, so the tree becomes very small. When the
minimum support still higher the required memory space is near to 0. The reason is that the number of frequent items becomes very small and only frequent 1-itemsets can be found, such that less memory space is needed for the FP-tree.

5.6 CONCLUSION

In this Chapter, an efficient approach called ECIM is proposed to mine frequent closed itemsets without the need of checking the closure. ECIM has several advantages over other approaches. First, it is not necessary to do closure checking. ECIM can achieve the effect of subset pruning and reducing the number of conditional FP-trees. Because ECIM does not generate any candidates, it is not necessary to store itemsets in the memory. It can give in the output frequent closed itemsets directly. The results of performance study show that ECIM is 36% runtime efficient and 45% memory efficient.