



ISSN:1991-8178

## Australian Journal of Basic and Applied Sciences

Journal home page: www.ajbasweb.com



### Top-K Frequent Itemsets Mining Using Compressed Poc Tree

<sup>1</sup>A.Vaishnavi and <sup>2</sup>B.Anantharaj<sup>1</sup>PG Scholar, Dept. of CSE Thiruvalluvar College of Engg. & Tech.<sup>2</sup>Assistant Professor, Head of the department in CSE Thiruvalluvar College of Engg. & Tech.

#### ARTICLE INFO

##### Article history:

Article Received 12 January 2015

Revised 1 May 2015

Accepted 8 May 2015

##### Keywords:

Data mining, frequent item set mining, Node sets, Algorithm and Performance

#### ABSTRACT

Node-list and N-list, two novel data structure proposed in recent years, have been proven to be very efficient for mining frequent item sets. The main problem of these structures is that they both need to encode each node of a PPC-tree with pre-order and post-order code. This causes that they are memory-consuming and inconvenient to mine frequent item sets. In this paper, we propose Node set, a more efficient data structure, for mining frequent item sets. Node sets require only the pre-order (or post-order code) of each node, which makes it saves half of memory compared with N-lists and Node-lists. Based on Node sets, we present an efficient algorithm called FIN to mining frequent item sets. For evaluating the performance of FIN, we have conduct experiments to compare it with PrePost and FP-growth, two state-of-the-art algorithms, on a variety of real and synthetic datasets. The experimental results show that FIN is high performance on both running time and memory usage.

© 2015 AENSI Publisher All rights reserved.

To Cite This Article: A.Vaishnavi and B.Anantharaj., Top-K Frequent Itemsets Mining Using Compressed Poc Tree. *Aust. J. Basic & Appl. Sci.*, 9(21): 52-58, 2015

### INTRODUCTION

Frequent itemset mining, first proposed by Agrawal, Imielinski, and Swami (1993), has become a fundamental task in the field of data mining because it has been widely used in many important data mining tasks such as mining associations, correlations, episodes, and etc. Since the first proposal of frequent itemset mining, hundreds of algorithms have been proposed on various kinds of extensions and applications, ranging from scalable data mining methodologies, to handling a wide diversity of data types, various extended mining tasks, and a variety of new applications (Han, Cheng, Xin, & Yan, 2007).

In recent years, we present two data structures called Node-list (Deng & Wang, 2010) and N-list (Deng, Wang, & Jiang, 2012) for facilitating the mining process of frequent itemsets. Both structures use nodes with pre-order and post-order to represent an itemset. Based on Node-list and N-list, two algorithms called PPV (Deng & Wang, 2010) and PrePost (Deng *et al.*, 2012) are proposed, respectively for mining frequent itemsets. The high efficiency of PPV and PrePost is achieved by the compressed characteristic of Node-lists and N-lists. However, they are memory-consuming because Node-lists and N-lists need to encode a node with pre-order and post-order. In addition, the nodes' code

model of Node-list and N-list is not suitable to join Node-lists or N-lists of two short itemsets to generate the Node-list or N-list of a long itemset. This may affect the efficiency of corresponding algorithms. Therefore, how to design an efficient data structure without pre-order and post-order is an interesting topic. To this end, we present a novel data structure called Nodeset, for mining frequent itemsets. Different from Node-lists and N-lists, Nodesets require only the pre-order (or post-order code) of each node without the requirement of both pre-order and post-order. Based on Nodesets, we propose FIN, an efficient mining algorithm, to discover frequent itemsets. FIN directly discovers frequent itemsets in a search tree called set-enumeration tree (Rymon, 1992). For avoiding repetitive search, it also adopts a pruning strategy named promotion, which is similar to Children-Parent Equivalence pruning (Burdick, Calimlim, Flannick, Gehrke, & Yiu, 2005), to greatly reduce the search space. For evaluating the performance of FIN, we conduct a comprehensive performance study to compare it against PrePost and FP-growth. The experimental results show that FIN is efficient on both running time and memory consumption. The rest of this paper is organized as follows. In Section 2, we introduce the background and related work for frequent itemset mining. Section 3 introduces the Nodeset structure and its basic properties. We describe FIN in Section 4.

Experiment results are shown in Section 5 and conclusions are given in Section 6.

## 2. Related work:

Formally, the task of frequent itemset mining can be described as follows. Let  $I = \{i_1, i_2, \dots, i_m\}$  be the universal item set and  $DB = \{T_1, T_2, \dots, T_n\}$  be a transaction database, where each  $T_k$  ( $1 \leq k \leq n$ ) is a transaction which is a set of items such that  $T_k \# I$ .  $P$  is called an itemset if  $P$  is a set of items. Let  $P$  be an itemset.

A transaction  $T$  is said to contain  $P$  if and only if  $P \# T$ . The support of itemset  $P$  is the number of transactions in  $DB$  that contain  $P$ . Let  $n$  be the predefined minimum support and  $|DB|$  be the number of transactions in  $DB$ . An itemset  $P$  is frequent if its support is no less than  $n \cdot |DB|$ . Given a transaction database  $DB$  and a threshold  $n$ , the task of mining frequent itemsets is to find the set of all itemsets

whose supports are not less than  $n \cdot |DB|$ .

Most of the previously proposed algorithms for mining frequent itemsets can be clustered into two groups: the Apriori-like method and the FP-growth method (Deng *et al.*, 2012). The Apriori-like method is based on anti-monotone property (Agrawal & Srikant, 1994), called Apriori, which states that if any length  $k$  itemset is not frequent, its length  $(k + 1)$  super-itemset also cannot be frequent.

The Apriori-like methods employ candidate-set-generation- and-test strategy to discover frequent itemsets. That is, it generates candidate length  $(k + 1)$  itemsets in the  $(k + 1)$ th pass using frequent length  $k$  itemsets generated in the previous pass, and counts the supports of these candidate itemsets in the database. A lot of studies, such as (Agrawal & Srikant, 1994; Savasere, Omiecinski, & Navathe, 1995; Shenoy *et al.*, 2000; Zaki, 2000; Zaki & Gouda, 2003), adopt the Apriori-like method. The Apriori-like method achieves good performance by reducing the size of candidates.

However, previous studies reveal that it is highly expensive for Apriori-like method to repeatedly scan the database and check a large set of candidates by itemset matching (Han *et al.*, 2007). Different from the Apriori-like method, the FP-growth method mines frequent itemsets without candidate generation and has proven very efficient. The FP-growth method achieves impressive efficiency by adopting a highly condensed data structure called FP-tree (frequent itemset tree) to store databases and employing a partitioning-based, divide-and-conquer approach to mine frequent itemsets. Some studies, such as (Grahne & Zhu, 2005; Han, Pei, & Yin, 2000; Liu, Lu, Lou, Xu, & Yu, 2004; Pei *et al.*, 2001), adopt the FP-growth method.

The FP-growth method wins an advantage over the Apriori-like method by reducing search space and generating frequent itemsets without candidate generation. However, the FP-growth method only achieves significant speedups at low minimum supports because the process of constructing and

using the FP-trees is complex (Woon, Ng, & Lim, 2004). In addition, recurrently building conditional itemset bases and trees makes the FP-growth method inefficient when datasets are sparse (Deng *et al.*, 2012). In recent years, we propose N-list (Deng *et al.*, 2012) and Nodelist (Deng & Wang, 2010), two novel data structures, to represent itemsets. Both of the two structures are based on a tree structure called PPC-tree, which store the sufficient information about frequent itemsets.

A N-list or Node-list is a sorted set of nodes in the PPC-tree. Usually, the nodes in a N-list or Node-list are sorted by the ascending order of the pre-order of nodes. Their main difference is that Node-lists are built by descendant nodes while N-lists are built by ancestor nodes. N-lists or Node-lists have two important properties. The first one is that the N-list or Node-list of a length  $(k + 1)$  itemset can be constructed by joining the N-lists or Node-lists of its subset with length of  $k$ . The other one is that the support of an itemset is the sum of counts registering in the nodes of its N-list or Node-list.

The high efficiency of PrePost and PPV, which based on N-lists and Node-lists, respectively, is achieved by these two properties. Extensive experiments show that PrePost and PPV are about an order of magnitude faster than state-of-the-art Apriori-like algorithm, such as dEclat and eclat<sub>goethals</sub>, and are not slower than the algorithms based on FP-growth (Deng & Wang, 2010; Deng *et al.*, 2012). Compared with Node-lists, N-lists have two advantages.

The first one is that the length of the N-list of an itemset is much smaller than the length of its Node-list. The other one is that N-lists have property called single path property. The first advantage makes the time for joining two N-lists is much shorter than that for joining two Node-lists. This causes the efficiency of PrePost is higher than that of PPV. In addition, the single path property of N-lists is employed by PrePost to directly mine frequent itemsets without generating candidates in some cases while PPV must generate and test all candidates as the Apriori-like algorithms do. Therefore, PrePost is more efficient than PPV (Deng *et al.*, 2012). Since 2012, NC<sub>set</sub> (Deng & Xu, 2012), a structure similar to N-list and Node-list, has been proposed to mine erasable itemsets (Deng, Fang, Wang, & Xu, 2009) and the experimental results show that NC<sub>set</sub> is very efficient (Deng & Xu, 2012; Le & Vo, 2014; Le, Vo, & Coenen, 2013).

Although N-list and Node-list are efficient structures for mining frequent itemsets, they need to encode a node of a PPC-tree with pre-order and post-order code, which is memory-consuming and inconvenient to mine frequent itemsets. In this paper, we propose Nodeset, a novel structure, where a node is encoded only by preorder or post-order code. Nodesets deal with the problem inherent in N-list and Node-list and is proven to be more efficient by

extensive experiments. To the best of our knowledge, No such structure has been proposed in previous work.

### **Basic principles:**

In this section, we introduce relevant concepts and properties about Nodesets. We adopt some notations used in (Deng & Wang, 2010; Deng *et al.*, 2012). For more details, please refer to (Deng & Wang, 2010; Deng *et al.*, 2012). Note that, for simplicity, we denote an itemset of length  $k$  as a  $k$ -itemset in this paper.

### **3.1. POC-tree definition:**

Because Nodesets are based on a POC-tree, we first introduce the definition of POC-tree in brief. Here, POC-tree is the abbreviation of Pre-Order Coding tree.

#### **Definition 1. POC-tree is a tree structure:**

(1) It consists of one root labeled as “null”, and a set of item prefix sub trees as the children of the root.

(2) Each node in the item prefix sub tree consists of five fields: item-name, count, and children-list, pre-order. Item-name registers which item this node represents. count registers the number of transactions presented by the portion of the path reaching this node. Children-list registers all children of the node. Preorder is the pre-order rank of the node.

According to Definition 1, the structure of POC-tree is almost the same as the structure of PPC-tree (Deng & Wang, 2010; Deng *et al.*, 2012). The only difference of these two kinds of tree lie in that each node of POC-tree is encoded by its pre-order while each node of PPC-tree is encoded by both its pre-order and its post-order. In a POC-tree, the pre-order of a node is determined by a preorder traversal of the tree. In other word, the pre-order records the time when node  $N$  is accessed during the pre-order traversal. A POC-tree is only used to generate the Nodesets of frequent 2-itemsets. Later, we will find that after building these Nodesets, the POCtree is useless and can be deleted. In fact, we can also use the postorder to encode each node and build a similar tree to generate the Nodesets. That is, the post-order is equivalent to the pre-order for our method. For simplicity, we use the pre-order in this paper.

Based on Definition 1, we have the following POC-tree construction algorithm.

#### **Algorithm 1 (POC-tree Construction):**

**Input:** A transaction database  $DB$  and a minimum support  $n$ .

**Output:** A POC-tree and  $F1$  (the set of frequent 1-itemsets).

#### **1. [Frequent 1-itemsets Generation]:**

According to  $n$ , scan  $DB$  once to find  $F1$ , the set of frequent 1-itemsets (frequent items), and their

supports. Sort  $F1$  in support descending order as  $L1$ , which is the list of ordered frequent items. Note that, if the supports of some frequent items are equal, the orders can be assigned arbitrarily.

#### **2. [POC-tree Construction]:**

The following procedure of construction POC-tree is the Same as that of constructing a FP-tree (Han, Pei, & Yin, 2000).

Create the root of a POC-tree,  $Tr$ , and label it as “null”. For each transaction  $Trans$  in  $DB$  do the following.

Select the frequent items in  $Trans$  and sort out them according to the order of  $F1$ . Let the sorted frequent-item list in  $Trans$  be  $[p | P]$ , where  $p$  is the first element and  $P$  is the remaining list. Call insert tree ( $[p | P], Tr$ ).

The function insert tree ( $[p | P], Tr$ ) is performed as follows.

If  $Tr$  has a child  $N$  such that  $N.item-name = p.item-name$ , then increase  $N$ 's count by 1; else create a new node  $N$ , with its count initialized to 1, and add it to  $Tr$ 's children-list. If  $P$  is nonempty, call insert tree( $P, N$ ) recursively.

#### **3. [Pre-code Generation]:**

Scan the POC-tree to generate the pre-order of each node by the pre-order traversal. Note that Algorithm 1 is the same as the construction algorithm of PPC (Deng *et al.*, 2012) except the third part of Pre-code Generation. For better understanding the concept and the construction algorithm of POC-tree, let's examine the following example.

Example 1 (4). Let the transaction database,  $DB$ , be represented by the information from the left two columns of Table 1 and  $n = 0.4$ .

The frequent 1-itemsets set  $F1 = \{a, b, c, e, f\}$ .

Fig. 1 shows the POC-tree which is constructed from the database shown in Example 1 after executing Algorithm 1. The number outside of a node is the pre-order of the node. In fact, the pre-order of a node is its identification. Note that the POC-tree is constructed using the right most column of Table 1 in Algorithm 1.

Obviously, the secondcolumn and the last column are equivalent for mining frequent itemsets under the given minimum support. In the rightmost columns of Table 1, all infrequent items are eliminated and frequent items are listed in support-descending order. This ensures that the  $DB$  can be efficiently represented by a compressed tree structure.

Now there are a different set of research works that focus on the same problem of frequent set item mining but use contrastingly different approach. Based on the approaches which are used for scanning the transactional databases and mining out the

frequent patterns we can divide the algorithms used for mining frequent itemsets into two categories:

- Mining Using Horizontal Data Format
- Mining Using Vertical Data Format

The first category corresponds to the methods discussed above which involve mining from a

transactional databases in which each record belongs to a transaction, i.e the first column is the transaction ID and the second column is the item set in the transactions. This will be in the form of {TID: {itemset}} which can be shown as in the figure below:

**Table 2.1:** Horizontal Data Format

TID	ITEM LABELS
T01	{"Bread", "Butter", "Milk", "Biscuits"}
T02	{"Stationary", "Biscuits", "Chocolates", "Milk"}
T03	{"Apples", "Banana", "Eggs", "Soap", "Potato"}
T04	{"Eggs", "Bread", "Soap", "Biscuits", "Butter"}
T05	{"Book", "Stationary", "Soap", "Apples", "Carrot"}

A typical Transactional database is given in the Table 2.1. This transaction is similar to the transactions in any departmental store where the transaction Id gives a unique set of items that a customer buys in one attempt. Here in this figure we can see that each tuple belongs to a particular transaction which is done by a single customer. It is not necessary that all transactions belong to different customers. Since a customer can make as many transactions as possible, more than one transaction may belong to a single customer.

The first category of the frequent itemset mining i.e., the apriori algorithm and its derivatives work on these types of transactional databases. They scan each of the record of the database and generate a candidate set for the each iteration. Some of the approaches scan these records and create an in memory data structure as in and then start mining the in-memory tree for the frequent itemsets. Thus we can further categorize this kind of approach into two into

Mining with candidate generation

Mining without candidate generation

Here the apriori and its extensions and enhancements can be categorized in to the first

category which is dependent on multiple scans of the database for generation of the candidate sets. The other categories of algorithms include the algorithms which scan the database once and then mines out the patterns which are frequent.

The other category is that of which uses the vertical data format of the transactional database. If the vertical data format is not available it is formatted by changing the columns as the item and the set of Transaction identifiers. Here each row corresponds to a particular item which is present in union set of all the transactions. The entries corresponding to each item will be a set of transaction identifiers in which they appear. The basic format of the records will be as {item: TID set}. This can be seen with an example as shown in Table 2.2.

Here in Table 2.2, the same example is taken and represented in the Verticaldata format. The difference here from the former is that first column consists of the items instead of the transaction identifiers. For each entry of the item in the table we have a corresponding set of transaction identifiers. This set of transaction identifiers denote the transactions in which the particular item appears.

**Table 2.2:** Vertical Data Format

ITEM	TID Set
Bread	{T01, T04, T06}
Butter	{T01, T04, T06, T07}
Milk	{T01, T02, T06}
Biscuits	{T01, T02, T04}
Stationary	{T02, T05}
Chocolates	{T02, T06}
Apples	{T03, T05}
Banana	{T03, T07}
Soap	{T03, T04, T05}
Eggs	{T03, T07}
Potato	{T03}
Carrot	{T05}
Book	{T05}

This type of approach was given first by zaki et. al [19] in 2000 wherein they proposed the Equivalence Class Transformation algorithm also known as ECLAT.

This algorithm scans the transactional database and creates a vertical data set as shown in Table 2.2. Mining is done by taking each time a different combination of items and then intersecting their

corresponding TID sets which will give the set consisting of the transaction identifiers in which all of these items are present.

This method is easier to compute the support of the items and the itemsets. At each step the infrequent itemsets are removed and then the remaining item sets are again combined and their TID sets intersected and infrequent itemsets pruned.

It takes one scan of the horizontal database and converts it to a vertical data format. However this also has an overhead of generating new itemsets which has to again maintain a database of TID sets.

This issue is solved by using a Diffsets which maintains only the differences of the TID sets of the  $k$ th itemset and the  $k+1$ th itemset. This would be very helpful when the datasets are dense, i.e. each item is present in a lot of the transactions thus making the TID sets huge. Thus in this case instead of storing the entire TIDs one can simply use diffset and store the differences between the candidate generations.

#### **System Analysis:**

##### **4.1 Existing System:**

###### **Drawbacks Of Nodsets Approach:**

The nodesets are constructed using a pre-order coding tree which is actually a version of an FP-tree. In fact approaches similar to FP-growth copies the same tree construction algorithm but with certain differences such as in case of PPC-tree, the tree nodes are coded with both pre-order and post-order accompanied with their label and count whereas in case of POC-tree, the tree nodes are coded with either pre-order or a post-order code. Albeit its repeated use in various approaches it still has more scope for compression, i.e. the nodes in the POC-tree can be reduced further. Thus the memory efficiency of the approach can be improved.

If there are a lot of items with the same support, for example a 1000 items with the same support then, there may be 1000! Combinations of the items which when appear in the transactional database, they are left unsorted because of their similar support. Thus in this case the tree will be a lot heavier when compared to the one in which the items appear in the same order. This condition will occur when the datasets are huge and consistent.

Also if the nodes are more redundant, the time taken to traverse them will be more than necessary to mine out enough frequent items.

##### **4.2 Proposed System:**

We will now describe the slight drawbacks of the POC tree construction algorithm in the above work and will also give a solution to overcome the drawback. Using the Nodeset data structure and the modified POC-tree we give a maximal frequent itemset mining algorithm. We are modifying the FIN algorithm proposed in the earlier approach to mine out only the top-most  $k$  frequent items. This is because most of the times the interesting patterns are seen in the highest cardinality itemsets. A lot of algorithms have been proposed in the past and the recent for mining out top-most  $k$  frequent itemsets [36], [37], [32], [38]. The algorithm is provided with a pre-determined cardinality of the itemsets we require. If the cardinality is not

specified, then the top-most frequent itemsets is returned.

The proposed system can be divided into three parts as follows:

1. POC-tree Construction
2. POC-tree Compression
3. Top-k frequent itemset mining (TKFIN)

###### **Poc-Tree Construction:**

The construction of the POC-tree is the same as that of the approach given in the POC-tree construction algorithm as given in the algorithm 1 in previous section.

The general outline of the approach is as follows:

1. The Transactional database is scanned and the infrequent itemsets are removed.
2. After removal of infrequent items, the items in each of the transaction are sorted according to their descending support count which is also called as L-order of the items.
3. Construction of POC-tree starts from the root of the tree which is labeled as "null" and its pre-order code being 0.
4. For each transaction the following step is repeated recursively.
5. Each transaction is scanned from left to right in L-order and child node is created, if it does not exist in the children nodes of the parent node. The first item in the transaction is made as the direct child node to the root and further next nodes are made as children to this node. This process goes on till the transaction ends.
6. If there are no more transactions in the database, then return the POC-tree created.

###### **Compressed Poc-Tree:**

POC-Tree construction algorithm as given in algorithm 1 is a good approach to compress the transactions into a tree in such a way that we can mine them using Nodesets which removes the necessity of using pointers to the nodes from a conditional database as in the FP-tree approach. Although the algorithm does a good job in compressing the redundant prefixes in the transactions, there remains more space for removal of more nodes from the tree and compressing them within the POC-tree.

This can be explained by considering the given example and its POC-tree constructed using the algorithm 1. Here the observable property of the algorithm 1 is that when it does the sorting of the transactions according to the L order, it arbitrarily assigns the positions of the items for which the support is same.

Consider the Table 5 of sorted transactions, wherein the items have been sorted according to their L orders. For transaction T05 where the items are {"Soap", "Stationary", "Apples"}, since the item labeled "Soap" has the highest support it stands front

in the L order. Then the items labeled “Stationary”, “Apple” has the same support i.e. 2 and thus the algorithm can chose an arbitrary order of these two. Let us assume that the algorithm selects an ordering as {“Soap”, “Stationary”, and “Apples”} as seen in the case of the POC-tree constructed in figure 2. Also consider transaction T03 where the items are {“Soap”, “Apples”, “Banana”, “Eggs”}. Using the same approach we have “Soap” as the first element and the rest, i.e. {“Apples”, “Banana”, and “Eggs”} have same support which is 2. Now again an arbitrary order of these three is assumed as seen in the POC-tree, {“Soap”, “Apple”, “Banana”, “Eggs”}. Now since both the sets have the items “Soap” and “Apple” which can be legibly placed in the same order in both the sets, there is a little space for compression of these nodes in the POC-tree by keeping the nodes for “Soap” and “Apple” as common nodes for two sub-trees, thereby reducing the number of nodes in the POC-tree.

The above observation clearly states a requirement for either a new algorithm to replace the older one or an add-on algorithm which will compress the tree after obtaining it from the first scan of the database. We will be using the later approach. The algorithm proposed is Compress Tree (Subtree), which takes as input the sub-tree of a node. The algorithm 10 is recursive and it starts from the root node of the tree.

Eventually, encrypted in the same manner. Then, they compute the union of those subsets in their encrypted form. Finally, they decrypt the union set and remove from it item sets which are identified as fake and broadcast the results.

**Algorithm 2: POC – TreeCompression:**

**Input:** A POC – Tree obtained from the Algorithm 7

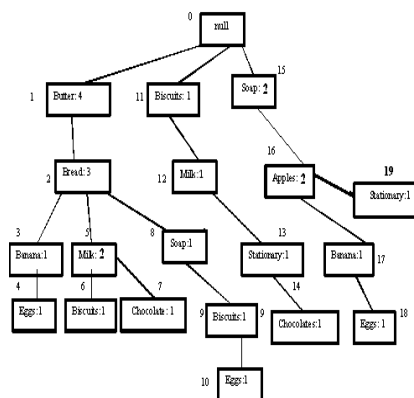
**Output:** A Compressed POC – Tree

1. root.label:= “null”;
2. root.pre:= 0;
3. **for** each child\_node ∈ root.children representing its subtree,

4. Call Compress\_Tree (child\_node, POC – Tree) recursively.
5. Return POC\_tree;

**Algorithm3:Compress\_Tree (Current\_node, POC\_Tree):**

1. if parent(Current\_node).label = “null” || parent(parent(Current\_node)).label = “null”
2. for each child\_node such that child\_node ∈ Current\_node.children
3. Compress\_Tree ( child\_node , POC – Tree)
4. else if Current\_node.support = parent(Current\_node).support then foreach
5. parent(current\_node).siblings do
6. **if** current\_node.label = parent(current\_node).sibling.label **then**
7. for each node ch ∈ POC – Tree.current\_node.children
8. subtree.add(ch)
9. copySubTree(ch, POC-tree, subtree)
10. POC-tree.remove(ch)
11. attachSubTree(Parent\_node, subtree.root, subtree, POC-tree)
12. end for
13. POC-tree.remove(node)
14. subtree.add(Parent\_node)
15. copySubTree(Parent\_node, POC-tree, subtree)
16. POC-tree.remove(Parent\_Node)
17. attachSubTree(parent(current\_node).sibling, Parent\_Node, subtree, POC-tree)
18. parent(current\_node).sibling.partialSupport = parent(current\_node).sibling
19. Current\_node.partialSupport
20. Compressed := true;
21. if compressed = “true”
22. for each ch ∈ Parent\_Node.children
23. Compress\_Tree(ch, POC-tree)
24. end for
25. else
26. for each ch in Current\_Node.children
27. Compress\_Tree(ch, POC-Tree)
28. End for.



**Fig. 3.1:** Compressed Poc-Tree

**Table 3.1:** COMPARISONS BETWEEN FIN AND TKFIN

FIN	TKFIN
Uses a pre order coding tree which cannot compress the tree when the supports of some items are equal.	Uses a compressed pre-order coding tree, which can handle the uncertainty in the permutation of the itemsets
Cannot handle denser data sets, the algorithm may get memory deprived.	Can handle denser data sets where a large number of items have equal supports. The burden on memory is lowered
Time taken to traverse a poc tree generated from a denser and huge dataset will be more relatively, but the overall time can be equivalent because there is no compression step	Time taken to traverse a huge and dense dataset will obviously be less except for the fact that the whole TKFIN algorithm will take almost the equivalent amount of time due to the extra compression step
It returns all the frequent itemsets thus obtained from the pattern tree	It returns only the k-cardinal frequent itemsets requested, otherwise returns the top most cardinal frequent itemset

**Conclusion and Future Enhancement:**

In this document we have discussed about the research problem of frequent itemset mining which forms the essential part of any association rule mining algorithm. We have defined the problem and have highlighted the pioneering research works done in this field and also have evaluated them in a general manner. We have divided the algorithms for frequent itemset mining approach into two categories. One which uses a Horizontal Data Format and other category uses the Vertical Data Format. Since from the past works it the Vertical Data Format has been proven to be the best approach, we choose this field.

We also consider the importance of the prefix tree based methods because of the most popular and successful approach known as fp-growth. We evaluate the works done using the vertical approaches in detail describing the merits and downfalls of each approach and how they can be overcome. We select an approach that efficiently combines the elegance of prefix-tree based approach and the quick support counting facility of ECLAT approach which uses a tree called POC-tree and a data structure called nodeset. Finally we propose our algorithm for compressing the POC-tree and propose an algorithm for mining top-k cardinality frequent itemsets using the same FIM approach.

Further this algorithm can be used in an association rule miner which will be an efficient approach for mining association rules between the frequent itemsets. Also the compressed POC-tree can be used for mining maximal frequent itemsets and closed frequent itemsets.

**REFERENCES**

- Jiawei Han, Micheline KaKBer, 2006. " *Data Mining Concepts and Techniques*".
- Agrawal, R., 1993."Mining Association Rules between Sets of Items in Large Databases",no. pp: 1-10.
- Brin.S and Tsur, U.S., "Dynamic Itemset Counting and Implication for Market Basket Data", pp: 255-264.
- Lee, W., S.J. Stolfo and K.W. Mok, 1998. "Mining Audit Data to Build Intrusion Detection Models".
- Kolda, T., "The TOPHITS Model for Higher-Order Web Link Analysis"
- Giannella, C., J. Han, X. Yan and P.S. Yu, "Mining Frequent Patterns in Data Streams at Multiple Time Granularities", pp: 191-212.
- Dinuca, C.E., 2012. "An application for clickstream analysis", 6(1): 0-7.
- Alves, R., D.S. Rodriguez-Baena and J.S. Aguilar-Ruiz, 2010. "Gene association analysis: a survey of frequent pattern mining from gene expression data", *Brief. Bioinform*, 11(2): 210-24.
- Leuven, K.U and U. Kingdom, 1997. "Finding frequent in chemical compounds.
- Jose, S., "Mining Quantitative Association Tables Rules in Large Relational".
- Yu, S and J. Watson, "Effective Algorithm for and Association", pp: 175-186.
- Omiecinski, E., "An Efficient Algorithm for Mining Databases Association Rules in Large", pp: 432-444.