Discovery of Frequent Itemsets: Frequent Item Tree-Based Approach

A.V. Senthil Kumar[1] & R.S.D. Wahidabanu[2]

[1]Senior Lecturer, Department of MCA, CMS College of Science and Commerce
Coimbatore – 641 006. Tamilnadu. India
[2]Head, Department of CSE, Govt. College of Engineering
Salem, Tamilnadu, India.

Abstract. Mining frequent patterns in large transactional databases is a highly researched area in the field of data mining. Existing frequent pattern discovering algorithms suffer from many problems regarding the high memory dependency when mining large amount of data, computational and I/O cost. Additionally, the recursive mining process to mine these structures is also too voracious in memory resources. In this paper, we describe a more efficient algorithm for mining complete frequent itemsets from transactional databases. The suggested algorithm is partially based on FP-tree hypothesis and extracts the frequent itemsets directly from the tree. Its memory requirement, which is independent from the number of processed transactions, is another benefit of the new method. We present performance comparisons for our algorithm against the Apriori algorithm and FP-growth.

Keywords: *association rules; data mining; frequent items; frequent item tree; header table; minimum support.*

1 Introduction

Recent days have witnessed an explosive growth in generating data in all fields of science, business, medicine, military, etc. The same rate of growth in the processing power of evaluating and analyzing the data did not follow this massive growth. Due to this phenomenon, a tremendous volume of data is still kept without being studied. Data mining, a research field that tries to ease this problem, proposes some solutions for the extraction of significant and potentially useful patterns from these large collections of data. Association Rules, Sequential Patterns, Classification, Similarity Analysis, Summarization and Clustering are major areas of interest in data mining. Among these, mining association rules [1] has been a very active research area. The process of mining association rules consists of two main steps: (1) Finding the frequent itemsets that have a minimum support, and (2) Using the frequent itemsets to generate association rules that meet a confidence threshold. Step 1 is the more expensive of the two since the number of item sets grows exponentially with the number of items. A large number of increasingly efficient algorithms to mine frequently itemsets have been developed over the years [2], [3], [4]. There are several

Received June 18th, 2007, Revised August 18th, 2007, Accepted for publication August 18th, 2007.

algorithms contributed [5], [6], [7], [8] to improve the performance of the Apriori algorithm that use different type of approaches. An analysis of the best known algorithms can be found in [9].

There are two main strategies for mining frequent itemsets: the candidate generation-and-test approach and the pattern growth approach. Apriori [2] and its several variations belong to the first approach, while FP-growth [3] and H-Mine [4] are examples of the second. Apriori algorithm [2] suffers from the problem of spending much of their time to discard the infrequent candidates on each level. Another problem can be the high I/O cost, which is inseparable from the level-wise approach. In case of the Apriori algorithm the database is accessed as many times as the size of the maximal frequent itemset is. This problem is partly overcome by algorithms based on pattern growth.

The FP-growth (Frequent Pattern – growth) [3] algorithm differs basically from the level-wise algorithms, that use a "candidate generate and test" approach. It does not use candidates at all, but it compresses the database into the memory in a form of a so-called FP-tree using a pruning technique. The patterns are discovered using a recursive pattern growth method by creating and processing conditional FP-trees. The drawback of the algorithm is its huge memory requirement, which is dependent on the minimum support threshold and on the number and length of the transactions.

In this paper, we propose a new algorithm named FIT (Frequent Item Tree) for mining complete frequent itemsets directly from the database. The algorithm requires only one full I/O scan of the dataset to build the prefix tree in main memory and then mines directly this structure. Mining the FP-tree structure is done recursively by building conditional trees that are of the same order of magnitude in number as the frequent patterns, but mining the FIT structure is done recursively by building conditional trees that are of less order of magnitude in number as the frequent patterns.

The structure of the rest of this paper is as follows: Section 2 defines the association rule mining problem. The algorithm FIT proposed for generating frequent itemsets is described in Section 3. Experimental results are presented in Section 4 and conclusion is reported in Section 5.

2 Problem Statement

The problem consists of finding associations between items or itemsets in transactional data. The data could be retail sales in the form of customer transactions or any collection of sets of observations. Formally, as defined in [2], the problem is stated as follows: Let $I=\{i_1,i_2,\ldots i_m\}$ be a set of literals, called

items. *m* is considered the dimensionality of the problem. Let *D* be a set of transactions, where each transaction *T* is a set of items such that $T \subseteq I$. A unique identifier *TID* is given to each transaction. A transaction *T* is said to contain *X*, a set of items in *I*, if $X \subseteq T$. An *association rule* is an implication of the form "$X \Rightarrow Y$", where $X \subseteq I$, $Y \subseteq I$, and $X \cap Y = \phi$. An itemset *X* is said to be *large* or *frequent* if its *support s* is greater or equal than a given minimum support threshold σ. The rule $X \Rightarrow Y$ has a *support s* in the transaction set *D* if *s%* of the transactions in *D* contain $X \cup Y$. In other words, the support of the rule is the probability that *X* and *Y* hold together among all the possible presented cases. It is said that the rule $X \Rightarrow Y$ holds in the transaction set *D* with *confidence c* if *c%* of transactions in *D* that contain *X* also contain *Y*. In other words, the confidence of the rule is the conditional probability that the consequent *Y* is true under the condition of the antecedent *X*. The problem of discovering all association rules from a set of transactions *D* consists of generating the rules that have a *support* and *confidence* greater than a given threshold. These rules are called *strong rules*. This association-mining task can be broken into two steps: (1) A step for finding all frequent k-itemsets known for its extreme *I/O* scan expense, and the massive computational costs, and (2) A straightforward step for generating strong rules. In this paper, we are mainly interested in the first step.

In this paper an algorithm is developed to discover frequent itemsets from the transaction database. Also FIT algorithm is compared with the Apriori algorithm [2] as well as FP-growth algorithm [3] which are used to discover the large frequent itemsets from a transaction database.

2.1 Apriori Algorithm

The most commonly known, and the first presented association rule mining algorithm is the Apriori algorithm introduced by Agrawal et al [2]. The main idea of the algorithm is based on the a priori hypothesis, namely, an itemset can only be frequent if all its subsets are also frequent. In other words, if an itemset is not frequent, no superset of it can be frequent. Exploiting this knowledge makes possible to reduce the search space efficiently when discovering the frequent itemsets, because using this knowledge the number of the candidates can be reduced. The Apriori algorithm is a level-wise method, which means that it discovers the *k*-itemsets during k^{th} database scan.

The algorithm works as follows. During the first database scan the items in the transactions are counted and the infrequent ones are discarded. In this way the frequent 1-itemsets are found. From these frequent items two candidates are generated by creating all the combination of them by keeping the lexicographic order. Formally, items x and y form a candidate (x,y) when $x \leq y$. During the

second database scan the support of the 2-candidates are counted. After a database reading the counters of the candidates are checked whether they are over the minimum support threshold. If a value of a counter exceeds the threshold, the candidate belonging to it becomes frequent, otherwise it is filtered out. The 3-candidates are generated from the frequent 2-itemsets regarding the following rule. Let be given two itemsets (i_1,i_2) and (i_3,i_4) where $i_1<i_2$ and $i_3<i_4$ as mentioned earlier. The two itemsets can form a 3-candidate if $i_1=i_3$ and (i_2,i_4) is also frequent. Fulfilling the second condition means that the a priori hypothesis is fulfilled. The resulting 3-candidate is the following: (i_1,i_2,i_4). In general two k-itemsets are joined by keeping the lexicographic order to form a $(k + 1)$-itemset if the first k-1 items of them are in common and all the $(k$-1)-subsets of the resulting candidate are frequent as well. The algorithm terminates if no candidates can be generated or no frequent itemsets are found. The pseudo code of the Apriori algorithm [2] is depicted in Figures 1 and 2.

```
procedure Apriori(minsup)
L1=find frequent 1-itemsets
for (k=2;Lk-1!=null;k++)
    Ck=AprioriGen(Lk-1)
    for each transaction t do
      Ct = subset (Ck,t)
      for each candidate c in Ct do
          c.counter++
      for each c in Ck do
         if c.counter >= minsup then
           Lk.Add(c)
return Ck
```

Figure 1 Pseudo code of the Apriori algorithm.

```
procedure Apriori(minsup)
for each itemset l1 in Lk-1 do
   for each itemset l2 in Lk-1 do
      if l1[1] = l2[1]
        and l1[2] = l2[2]
        and… and  l1[k-2] = l2[k-2]
        and l1[k-1] < l2[k-1]
      then
        c=l1 join l2
        if c has infrequent subset
        then DELETE c
        else Ck.Add(c)
return Ck
```

Figure 2 Pseudo code of the AprioriGen procedure.

2.2 FP-Growth Algorithm

One of the algorithms that does not use any candidates to discover the frequent patterns is the FP-growth (Frequent Pattern Growth) algorithm proposed in [3]. The other main difference to the Apriori algorithm is the number of the database readings. While the Apriori is a level-wise algorithm, the FP-growth is a two-phase method. It reads the database only twice and stores the database in a form of a tree in the main memory.

The algorithm works as follows. During the first database scan the number of occurrences of each item is determined and the infrequent ones are discarded. Then the frequent items are ordered descending their support. During the second database scan the transactions are read and the frequent items of them are inserted into a so-called FP-tree structure. In this way the database is pruned and is compressed into the memory. The aim of using FP-tree is to store the transactions in such a way that discovering the patterns can be achieved efficiently.

```
procedure FPGrowth(Tree, ∝)
if Tree contains a single path P then
   for each β = comb. of nodes in P do
     pattern = β ∪ α
     sup = min(sup of the nodes in β )
else
   for each a_i in the header of Tree do
     generatepattern = β ∪ α
      sup  = a_i.support
     construct β's conditional pattern base
     FPTree = construct β's conditional FP-tree
      If FPTree != 0 then
           FPGrowth(FPTree, β)
```

Figure 3 Pseudo code of the FP-growth algorithm.

Each node in the tree contains an item, a counter to count the support, and links to the child nodes, to the parent nodes and to the siblings of the node. The rule for constructing the FP-tree is as follows. When reading a transaction, its infrequent items are omitted and the frequent ones are ordered regarding their support. The transaction is then inserted into the tree. If the tree is empty the transaction is inserted as the only branch in the tree. If it is not empty, while the first k items of the transaction fit the prefix of one of the branches of the tree, a counter is incremented in each referred node in the tree. From the $(k+1)^{th}$ item, a new branch is created as a child of the node, which corresponds to the k^{th} item in the transaction, and the further items in the transactions are inserted as this new branch with a support counter set to one. A header belongs to the FP-tree

which contains the sorted 1-frequent items, their supports and a pointer to the first occurrence of the given item in the tree. The other occurrences of the given item in the tree are linked together sequentially as a list.

The FP-tree is processed recursively by creating several so-called conditional FP-trees. This is the recursive pattern growth method of the algorithm. When a conditional FP-tree contains exactly one branch the frequent itemsets are generated from it by creating all the combinations of each items. When traversing the whole FP-tree, all the frequent itemsets are discovered. The pseudo code of the FP-growth algorithm [3] is depicted in Figure 3.

3 Frequent Item Tree Algorithm

The main motivation of Frequent Item Tree algorithm is to enhance the above mentioned algorithms both regarding the execution time behavior and the memory management. The aim was to develop an algorithm whose memory usage is significantly lower than that of the FP-growth algorithm, and its execution time is smaller than the execution times of both the algorithms described earlier. The Frequent Item Tree algorithm is a novel method to find all the frequent itemsets quickly. It discovers all the frequent itemsets in only one database scan.

The goal of Frequent Item Tree algorithm is to build a compact data structure called FI tree. The construction is done in two phases, where the first phase requires a full I/O scan of the dataset and the second phase requires only a full scan of frequent 2-itemsets. The first initial scan of the database identifies the frequent 2-itemsets. The goal is to generate an ordered list of frequent 2-itemsets that would be used when building the tree in the second phase.

The first phase starts by arranging the entire database in the alphabetical order using MSD Radix sort. Various string sorting algorithms have been intensively studied based on their speed, property of stability and order (relationship between the number of keys to be sorted and the time required). When compared with Quick sort, Ternary Quick sort and MSD Radix sort the speed of Merge sort is less, its order is N log N but it is stable. Quick sort when compared with Ternary Quick sort and MSD Radix sort its speed is less, order is N log N and it is not stable. Speed of Ternary Quick sort is less than MSD Radix sort, order is N log N but is it stable. MSD Radix sort which is a specialized one for strings is faster than Ternary Quick sort, order is O(N) and it is stable. Since MSD Radix sort doesn't work by comparing keys as used by other string sorting algorithms, the time taken for sorting strings is linearly proportional to the number of items which makes FI-tree algorithm more efficient. During the database scan the number of occurrences of frequent 2-

itemsets is determined and infrequent 2-itemsets with the support less than the support threshold are weeded out. Then the frequent 2-itemsets are ordered in the alphabetical order using MSD Radix sort.

```
procedure FI Tree(Tree, F )
add first item of the first freq2 list to H;
    for each 2-itemset entry (top down order) in freq2 list  do
      if  F(I) >= minsup, then
  create a link to the second item of freq2;
     if the first item of freq2 changes then
  add first item of freq2 to H;
          create a link to the second item of freq2;
     if  both the item of freq2 is not available in H then
              call buildsubtree(F);
  end procedure

procedure buildsubstree (F )
          add first item of freq2 list to H;
          create a new node for this first item;
          create a link to the second item of freq2;
end procedure
```

Figure 4 Pseudo code of the Frequent Item Tree algorithm.

Phase 2 of constructing FI tree structure is the actual building of this compact tree. This phase requires a complete scan of the ordered frequent 2-itemsets. The ordered frequent 2-itemsets are used in constructing the FI tree as follows: The first item in the first frequent 2-itemset in the frequent 2-itemsets list is added to the header file and this item will be the root for the child node. The support of frequent 2-itemset is assigned as the support to the item in the header table. In the first step, for each frequent 2-itemset, read the first item and compare with the item in the header table. If the item is already present in the header table a link is assigned to the second item in the frequent 2-itemset with the item in the child node. The support for the item in the header table will be compared with the support of the frequent 2-itemset and the highest support will be assigned to the item in the header table. The first step is repeated until the first item in the frequent 2-itemset list changes. If the first item in the frequent 2-itemset list changes, then the corresponding first item in the frequent 2-itemset is added to the header table with the support of the frequent 2-itemset and the first step is continued until the first item in the frequent 2-itemset changes. If both the items in the corresponding 2-itemset are not available in the header table, then the first item of the frequent 2-itemset is added to the header table with the support of the frequent 2-itemset and a new node is added as child for the root node, in which the first item of frequent 2-itemset acts as child for the root node and a link is assigned to the second item of the frequent

2-itemset. The process used in first step will be continued till the first item in the frequent 2-itemset changes. The above procedure will be repeated until the end of the frequent 2-itemset list. Finally the second item in the last frequent 2-itemset is added to the header table, assigned its maximum support from the frequent 2-itemset list and corresponding link is made. The pseudo code of the algorithm is depicted in Figure 4.

For illustration, we use an example with the transactions shown in Table 1. Let the minimum support threshold set to 2**.** Various steps used in phase 1 are shown in Figure 5. Phase 1 starts by accumulating the support for all possible 2-itemsets that occur in the transactions. Step 2 of phase 1 removes all non-frequent 2-itemsets, in our example (*ab,ae* and *af*), leaving only the frequent 2-itemsets (*be,ac,ag,bc,bf,bg,ce,cf,cg,ef,eg* and *fg*). Finally all frequent 2-itemsets are sorted alphabetically to generate the sorted frequent 2-itemset list. This last step ends phase 1 of the FI tree algorithm and starts the second phase. In phase 2, the first item *a* of the first frequent 2-itemset *ac* is added to the header table. The 2-itemset *ac* generates the first path of FI tree with item *a* as root node and item *c* as the child node with support for *a* as 2 in the header table. A link is established between items *a* and *c* and its corresponding item entry *a* in the header table. Since the first item *a* of the second 2-itemset in the frequent 2-itemset list is already present in the header table, a link is established between items *a,c* and *g.* Both the items *b*and *c* of the next 2-itemset *bc* are not present in the header table item *b* is added to the header table and forms the second path of the FI tree with item *b* as the root node and item *c* as the child node with support of *b* as 2 in the header table. A link is established between items *b* and *c* and its corresponding item entry *b* in the header table. For the next frequent 2-itemset, the first item *b* is already in the header table, so a link is established between *b, c* and *e.* The support 3 for 2-itemset *be* will be assigned for item *b* in the header table. The same process occurs for all frequent 2-itemsetss until we build the FI tree for the transactions given in Table 1. At last item *g* from the frequent 2- itemset *fg* is added to the header table and assigned a support of 3. Figure 6 shows the result of the tree building process.

Table 1 Transactional database.

T.No	Items
T1	b e
T2	a b c e f g
T3	b c e f g
T4	a c g

Item	Freq
be	**3**
ab	1
ac	**2**
ae	1
af	1
ag	**2**
bc	**2**
bf	**2**
bg	**2**
ce	**2**
cf	**2**
cg	**3**
ef	**2**
eg	**2**
fg	**2**

Step 1

Item	Freq
be	3
ac	2
ag	2
bc	2
bf	2
bg	2
ce	2
cf	2
cg	3
ef	2
eg	2
fg	2

Step 2

Item	Freq
ac	2
ag	2
bc	2
be	3
bf	2
bg	2
ce	2
cf	2
cg	3
ef	2
eg	2
fg	2

Step 3

Figure 5 Steps of Phase 1.

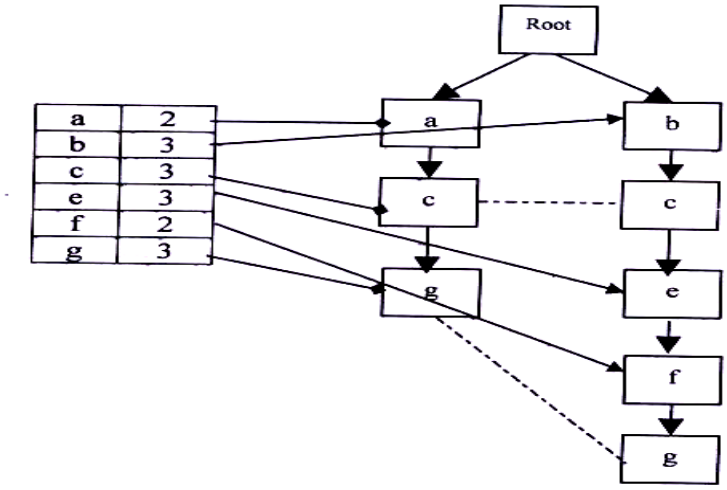

Figure 6 Frequent Item Tree.

4 Experimental Results

Experiments were conducted to test the efficiency of the FI-tree approach by comparing our approach with two well-known algorithms namely: Apriori and FP-growth. All experiments are performed on a 1.88 GHz P-IV Core 2DUO with 1 GB DDR2 800 MHz main memory, 80 GB HDD running on Microsoft Windows XP. All programs are written in Microsoft Visual Basic 6.0. We used

synthetic transactional databases generated using IBM Quest synthetic data generator [10]. All the experiments were conducted using T20I7D200k dataset. The naming conventions of the datasets are shown in Table 2. The number of the items that can occur in the transactions is 1000.

Table 2 Meaning of the parameters in the names of the datasets.

Parameter	Meaning
T	Average length of the transactions
I	Average size of maximal frequent itemsets
D	Number of transactions
K	Thousands

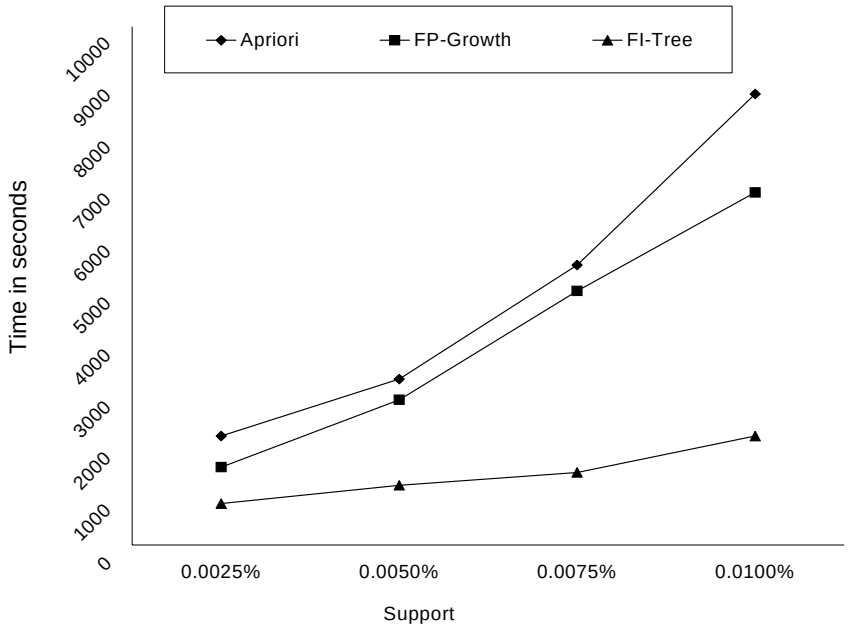

Figure 7 Performance at various support levels.

To test the behavior of the FI-tree vis-à-vis different support thresholds, a set of experiments was conducted. The mining process tested different support levels, which are 0.0025% that revealed almost 110k frequent patterns, 0.005% that revealed nearly 75k frequent patterns, 0.0075% that generated 40k frequent patterns and 0.01% that returned 22k frequent patterns. Figure 7 depicts the time needed in seconds for each one of these runs. It is clear that FI-tree algorithm is

the fastest of all the three methods. The execution time of the FI-tree method is always smaller than that of the Apriori and FP-growth algorithms. Figure 8 shows the execution times in seconds for the various datasets with transaction sizes 50, 100, 150, 200 and 250 of the three algorithms. It can be easily concluded that the execution time dependency of the Apriori algorithm on the number of transactions is linear. FP-growth algorithm reads the database twice and stores the database in the form of a tree in the main memory. FI-tree algorithm reads the entire database only once and the frequent 2-itemsets of them are used to build a tree structure so-called FI-tree structure.

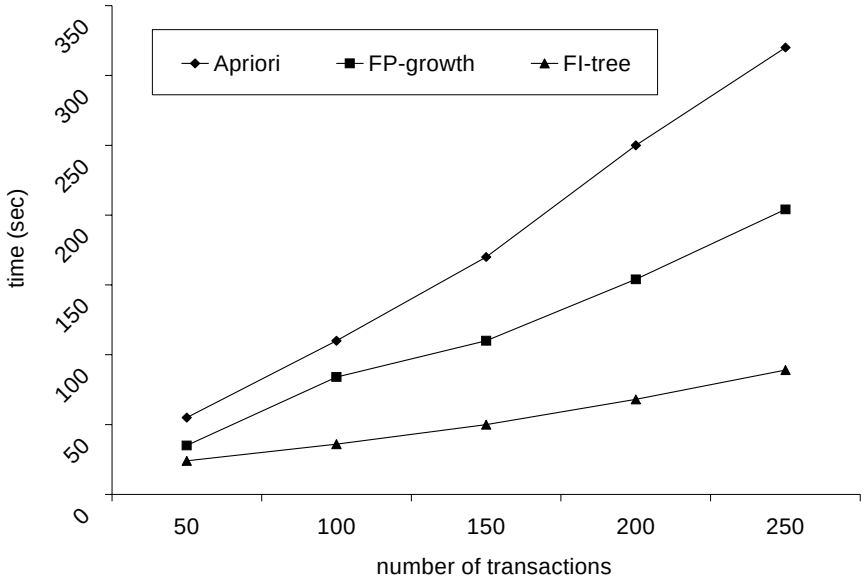

Figure 8 Execution time of the tree algorithm as a function of the number of transactions by 0.8% minimum support threshold.

In Figure 9 the peak memory sizes in megabytes are illustrated as a function of the number of transactions when the average size of the maximal frequent items is 7 and the average size of the transactions is 20. The minimum support threshold is set to 0.7%. The memory requirement for FI-tree algorithm is less for all datasets with transaction sizes 50, 100, 150, 200 and 250 when compared to FP-growth algorithm. The memory requirement of the FP-growth algorithm increases significantly with the growth of the number of transactions. The reason for this can be found when examining the sizes of the trees generated by the algorithm. If the algorithm mines two datasets with the same statistical properties but the one contains an order of magnitude more transactions than the other, the first FP-tree built by the FP-growth algorithm contains an order of

magnitude more nodes in the former case than in the latter. However the rules that have been found are nearly the same. From this fact we can draw the conclusion that several redundant nodes are in the FP-tree when increasing the number of the transactions. Its drawback is, however, that the memory requirement of the algorithm is huge. The memory requirement of the FI-tree algorithm depends only on the number of frequent 2-itemsets in the given transactions. Since FI-tree algorithm stores only the items needed for finding frequent 2-itemsets which are then used to form a tree in the main memory, the memory requirement of the FI-tree algorithm does not depend on the number of transactions.

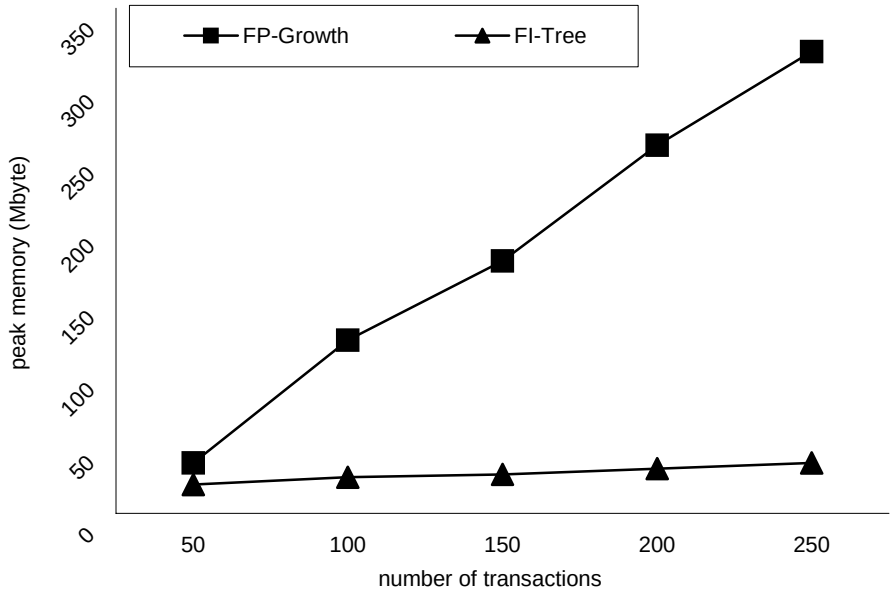

Figure 9 Peak memory of the algorithms as a function of transactions by 0.7% minimum support threshold.

The sizes of the first generated FP-trees in bytes are depicted in case of the FP-growth and of the FI-tree algorithms when used T20I7D200k dataset as a function of the minimum support threshold is illustrated in Figure 10. Apparently the sizes of the tree in case of all seven support thresholds are less in FI-tree algorithm when compared with FP-growth algorithm.

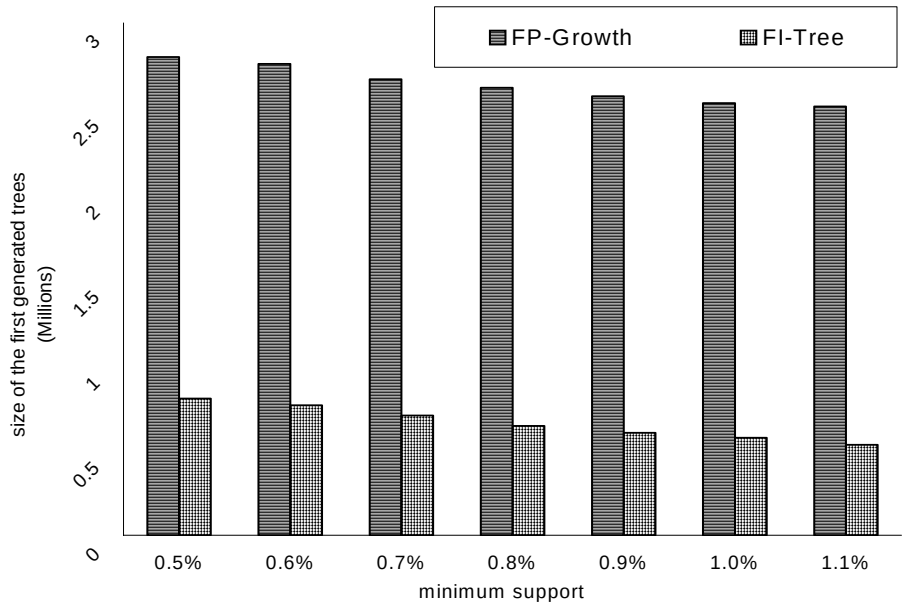

Figure 10 Sizes of the first generated tree of the FP-Growth and of the FI-Tree algorithm when using T20I7D200K.

5 Conclusion

This paper is concerned with the problem of efficiently discovering frequent itemsets in transactional databases. The algorithm identifies the main problem of the FP-growth algorithm which is the recursive creation and mining of many conditional pattern trees, and which are equal in number to the distinct frequent patterns generated. We have replaced this step by creating one FI-tree by scanning the database only once and by using the frequent 2-itemsets. The FI-tree algorithm is even faster than Apriori and FP-growth algorithms, and the memory requirement of the novel method does not depend on the number of transactions.

The advantage of the FI-tree algorithm is the quick mining process that does not use candidates. Its drawback is however, that the time needed to build an FI-tree and the memory requirement depends upon the number of frequent 2-itemsets. If the algorithm mines two datasets with the same number of transactions but the one contains more frequent 2-itemsets than the other, the first needs more time to build an FI-tree and the memory requirement will be more than in the latter. We are currently studying the possibility of using hashing techniques to find the efficient frequent 2-itemsets in order to reduce the time and memory requirements to build an FI-tree.

References

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

[2] R.Agrawal and R.Srikant, "Fast Algorithms for Mining Association Rules", *Proc.of the 20th Intl. Conf.* on VLDB, Santiago, Chile, 1994.

[3] J.Han, J.Pei, and Y.Yin, "Mining Frequent Patterns without Candidate Generation", *Proc. Of the ACM SIGMOD,* Dallas, TX, 2000.

[4] J.Pei, J.Han, H. Lu, S.Nishio, S.Tang, and D.Yang, "H-Mine: Hyper-Structure Mining of Frequent Patterns in Large Databases", *Proc. Of IEEE ICDM,* San Jose, California, 2001.

[5] M.H. Zaki, "Scalable Algorithms for Association Mining", *IEEE Transactions on Knowledge and Data Engineering,* May/June 2000, 372-390.

[6] V.S. Ananthanarayana, D.K.Subramanian and M.N. Murty, "Scalable, distributed and dynamic mining of association rules", *Proc.of the 7th Intl.Conf. on High Performance Computing,* Bangalore, India, pp.559-566.

[7] R.J. Bayardo, "Efficiently mining long patterns from databases", *Proc. Of the ACM SIGMOD Intl. Conf.on Management of Data,*Seatle, WA, pp. 85-93.

[8] P.Shenoy, J.R.Haritsa, S.Sundarshan, G.Bhalotia, M.Bawa and D.Shah, "Turbo-charging vertical mining of large databases", *Proc.of the ACM SIGMOD,* Dallas, TX, pp.22-33.

[9] R.Ivancsy, F.Kovacs and I.Vajk, "An Analysis of Association Rule Mining Algorithms", *In CD-ROM Proc.of Fourth International ICSC Symposium on Engineering of Intelligent Systems (EIS 2004),* Island of Madeira, Portugal.

[10] I.Almaden. Quest synthetic data generation code. http://www.almaden.ibm.com/cs/quest/syndata.html

R.S.D. Wahidabanu is presently Head, Department of CSE, Government College of Engineering, Salem, Tamilnadu, India. She has 24 years of teaching experience. Her research area includes Pattern Recognition, Artificial Intelligence and Data Mining.

A.V. Senthil Kumar is presently working as a Lecturer in the Department of MCA, CMS College of Science and Commerce, Coimbatore, Tamilnadu, India. He has more than 10 years of teaching and 5 years of industrial experience. During August 2006, he has participated and presented two papers in the area of Data Mining in an International Conference, ICTS 2006 in Indonesia and two papers in an International Conference, IMECS 2007 in Hongkong during March 2007. His research area include Data Mining and Image Processing.