A Comparative Study of Association Rules Mining Algorithms

Cornelia Győrödi[*], Robert Győrödi[*], prof. dr. ing. Stefan Holban[**]

[*]Department of Computer Science, University of Oradea, Str. Armatei Romane 5, 3700, Oradea, Romania, Phone: +40 (0) 59 432-830, e-mail: rgyorodi@rdsor.ro

[**] Computer & Software Engineering Department, Politehnica University of Timisoara, Bd. Vasile Parvan 2, Timisoara, Romania, Phone: +40 (0) 56 204-333, e-mail: stefan@cs.utt.ro, WWW: http://www.cs.utt.ro/~stefan

Abstract: This paper presents a comparison between classical frequent pattern mining algorithms that use candidate set generation and test and the algorithms without candidate set generation. In order to have some experimental data to sustain this comparison a representative algorithm from both categories mentioned above was chosen (the Apriori, FP-growth and DynFP-growth algorithms). The compared algorithms are presented together with some experimental data that lead to the final conclusions.

Keywords – Data Mining, frequent pattern, Apriori, FP-growth, DynFP-growth.

1 Introduction

In recent years the sizes of databases has increased rapidly. This has led to a growing interest in the development of tools capable in the automatic extraction of knowledge from data. The term Data Mining, or Knowledge Discovery in Databases, has been adopted for a field of research dealing with the automatic discovery of implicit information or knowledge within databases [7]. The implicit information within databases, and mainly the interesting association relationships among sets of objects, that lead to association rules, may disclose useful patterns for decision support, financial forecast, marketing policies, even medical diagnosis and many other applications. This fact attracted a lot of attention in recent data mining research [7]. As shown in [1] , mining association rules may require iterative scanning of large databases, which is costly in processing. Many researchers have focused their work on efficient mining of association rules in databases ([1], [2], [6], [7], [8], [9],[11]).

A very influential association rule mining algorithm, Apriori [1], has been developed for rule mining in large transaction databases. Many other algorithms developed are derivative and/or extensions of this algorithm. A major step forward in improving the performances of these algorithms was made by the introduction of a novel, compact data structure, referred to as *frequent pattern tree*, or FP-tree [2], and the associated mining algorithm, FP-growth.

The main difference between the two approaches is that the Apriori-like techniques are based on *bottom-up generation of frequent itemset combinations* and the FP-tree based ones are *partition-based, divide-and-conquer methods*.
After conducting several performance studies Győrödi C., *et al.* (2003) developed an improved FP-tree based technique, named Dynamic FP-tree [11]. The developed method clearly indicates a performance gain mainly when applied on real world sized databases.
After this introduction, the paper is organised as follows. Section 2 presents the problem definition. The main aspects of Apriori, FP-growth and DynFP-growth algorithms are presented in Section 3. Section 4 shows a comparative study of the algorithms and the paper is concluded in Section 5.

2 Problem Definition

Association rule mining finds interesting association or correlation relationships among a large set of data items [8]. The association rules are considered interesting if they satisfy both a *minimum support* threshold and a *minimum confidence* threshold [3]. A more formal definition is the following [4]. Let $\Im = \{i_1, i_2, \ldots, i_m\}$ be a set of items. Let D, the task-relevant data, be a set of database transactions where each transaction T is a set of items such that $T \subseteq \Im$. Each transaction is associated with an identifier, called TID. Let A be a set of items. A transaction T is said to contain A if and only if $A \subseteq T$. An association rule is implication of the form $A \Rightarrow B$, where $A \subset \Im$, $B \subset \Im$, and $A \cap B = \varnothing$. The rule $A \Rightarrow B$ holds in the transaction set D with *support* s, where s is the percentage of transactions in D that contain $A \cup B$ (i.e., both A and B). This is taken to be the probability, $P(A \cup B)$. The rule $A \Rightarrow B$ has *confidence* c in the transaction set D if c is the percentage of transactions in D containing A that also contain B. This is taken to be the conditional probability, $P(B|A)$. That is,

$$support(A \Rightarrow B) = P(A \cup B) \quad (1)$$
$$confidence(A \Rightarrow B) = P(B \mid A) \quad (2)$$

The definition of a frequent pattern relies on the following considerations [5]. A set of items is referred to as an itemset (pattern). An itemset that contains k items is a k-itemset. For example the set {*name*, *semester*} is a 2-itemset. The occurrence frequency of an itemset is the number of transactions that contain the itemset. This is also known, simply, as the frequency, support count, or count of itemset. An itemset satisfies *minimum support* if the occurrence frequency of the itemset is greater than or equal to the product of *minimum support* and the total number of transactions in D. The number of transactions required for the itemset to satisfy *minimum support* is therefore referred to as the *minimum support count*. If an itemset satisfies *minimum support*, then it is a *frequent itemset* (*frequent pattern*).

The most common approach to finding association rules is to break up the problem into two parts [6]:

1. Find all frequent itemsets: By definition, each of these itemsets will occur at least as frequently as a pre-determined minimum support count [8].
2. Generate strong association rules from the frequent itemsets: By definition, these rules must satisfy minimum support and minimum confidence [8].

Additional interestingness measures can be applied, if desired. The second step is the easier of the two. The overall performance of mining association rules is determined by the first step. As shown in [2], the performance, for large databases, is most influenced by the combinatorial explosion of the number of possible frequent itemsets that must be considered and also by the number of database scans that has to be performed.

3 The Algorithms in Association Rules Mining

3.1 The Apriori Algorithm

Figure 1 gives an overview of the Apriori algorithm for finding all frequent itemsets, using the notation in Table 1. The first pass of the algorithm simply counts item occurrences to determine the large 1-itemsets. A subsequent pass, say pass k, consists of two phases. First, the large itemsets L_{k-1} found in the $(k\text{-}1)^{th}$ pass are used to generate the candidate itemsets C_k, using the Apriori candidate generation function (apriori-gen) described below. Next, the database is scanned and the support of candidates in C_k is counted. For fast counting, an efficient determination if the candidates in C_k that are contained in a given transaction t is needed. A hash-tree data structure [1] is used for this purpose. The Apriori algorithm is:

L_1 = {large 1-itemsets};
for (k = 2; $L_{k-1} \neq \emptyset$; k++) **do begin**
  C_k = apriori-gen(L_{k-1}); //New candidates
  forall transactions $t \in D$ **do begin**
    C_t = subset(C_k, t);
    //Candidates contained in t
    forall candidates $c \in C_t$ **do**
      c.count++;
  end
  L_k = { $c \in C_k$ | c.count ≥ minsup }
end
Answer = $\cup_k L_k$;

Figure 1. The Apriori algorithm.

The *apriori-gen* function takes as argument L_{k-1}, the set of all large (k-1)-itemsets. It returns a superset of the set of all large k-itemsets and is described in [1].

3.2 The FP-growth Algorithm

As shown in [2], the main bottleneck of the Aprioi-like methods is at the *candidate set generation and test*. This problem was dealt with by introducing a novel, compact data structure, called *frequent pattern tree*, or FP-tree then based on this structure an FP-tree-based pattern fragment growth method was developed, FP-growth. The definition, according to [2] is as follows.

Definition 1 (FP-tree) A frequent pattern tree is a tree structure defined below.

1. It consists of one root labeled as "root", a set of *item prefix sub-trees* as the children of the root, and a *frequent-item header table*.
2. Each node in the *item prefix sub-tree* consists of three fields: *item-name*, *count*, and *node-link*, where *item-name* registers which item this node represents, *count* registers the number of transactions represented by the portion of the path reaching this node, and *node-link* links to the next node in the **FP-tree** carrying the same item-name, or null if there is none.
3. Each entry in the *frequent-item header table* consists of two fields, (1) *item-name* and (2) *head of node-link*, which points to the first node in the **FP-tree** carrying the *item-name*.

The actual algorithm, according also to [2] is:

Algorithm 1 (FP-tree construction)

Input: A transactional database *DB* and a minimum support threshold ξ.

Output: Its frequent pattern tree, **FP-tree**

Method: The **FP-tree** is constructed in the following steps:

1. Scan the transaction database *DB* once. Collect the set of frequent items *F* and their supports. Sort *F* in support descending order as *L*, the *list* of frequent items.
2. Create the root of an **FP-tree**, *T*, and label it as "root". For each transaction *Trans* in *DB* do the following.
   a. Select and sort the frequent items in *Trans* according to the order of *L*. 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*], *T*).
   b. The function *insert_tree*([*p* | *P*], *T*) is performed as follows. If *T* has a child *N* such that *N.item-name = p.item-name*, then increment *N*'s count by 1; else create a new node *N*, and let its count be 1, its parent link be linked to *T*, and its node-link be linked to the nodes with the same *item-name* via the node-link structure. If *P* is nonempty, call *insert_tree*(*P*, *N*) recursively.

The FP-growth [2] algorithm for mining frequent patterns with FP-tree by pattern fragment growth is:

Input: a *FP-tree* constructed with the above mentioned algorithm;
D – transaction database;
s – minimum support threshold.

Output: The complete set of frequent patterns.

Method:

call FP-growth(*FP-tree*, *null*).

Procedure FP-growth(*Tree*, *A*)

{

if *Tree* contains a single path *P*

then for each combination (denoted as *B*) of the nodes in the path *P* **do**

```
        generate pattern B ∪ A with support=minimum support of nodes in B
    else for each ai in the header of the Tree do
    {
        generate pattern B = ai ∪ A with support = ai.support;
        construct B's conditional pattern base and B's conditional FP-tree TreeB;
        if TreeB ≠ Ø
            then call FP-growth(TreeB, B)
    }
}
```

3.3 The DynFP-growth Algorithm

As shown in [2] the main bottleneck of the Aprioi-like methods is at the *candidate set generation and test*. This problem was taken into consideration by introducing a novel, compact data structure, named *frequent pattern tree*, or FP-tree, then based on this structure an FP-tree-based pattern fragment growth method was developed, FP-growth. The completeness and compactness of this structure is also shown in [2]. Some observations on the way the FP-tree are constructed.

1. The resulting FP-tree is not unique for the same "logical" database.
2. The process needs two complete scans of the database.

A solution to the first observation was given by Győrödi C., *et al.*, (2003) [11], by using a support descending order together with a lexicographic order, ensuring in this way the uniqueness of the resulting FP-tree for different "*logically equivalent*" databases. The second observation was addressed also by Győrödi C., *et al.*, (2003) [11], by devising a dynamic FP-tree reordering algorithm, and employing this algorithm whenever a "promotion" to a higher order of at least one item is detected. Although the resulting FP-tree could be too large to be stored in its entirety in the main memory, because of its properties, and for a relatively high number of queries with different minimum supports, it would be more practical, from time consuming point of view, to store it on disk in its full form and using only the portions that are required from it. Using the dynamic reordering one doesn't have to rebuild the FP-tree even if the actual database is updated. In this case the algorithm has to be performed taking into consideration only the new transactions and the stored FP-tree. This approach can provide a very quick response to any queries even on databases that are being continuously updated – fact that is true in many cases. Because the dynamic reordering process, Győrödi C., *et al.*, (2003) [11] proposed a modification of the original structures, by replacing the single linked list with a doubly linked list for linking the tree nodes to the *header* and adding a *master-table* to the same *header*. All these modifications are presented in more details in [11]. The Dynamic FP-tree construction/reordering algorithm (Győrödi C., *et al.*, 2003) is described in the following.

Algorithm (Dynamic FP-tree construction)
Input: A transactional database *DB* and a minimum support threshold ξ.
Output: Its frequent pattern tree, **FP-tree**

Method: The **FP-tree** is constructed in the following steps:

1. Create the root of an **FP-tree**, *T*, and label it as "root". For each transaction *Trans* in *DB* the next steps must be followed.
    a. Add the items in *Trans* into the *header*.
    b. Select and sort the items in *Trans* according to the order of the *header*'s *master-table*. 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*], *T*).
    c. The function *insert_tree*([*p* | *P*], *T*) is performed as follows. If *T* has a child *N* such that *N.item-name* = *p.item-name*, then increment *N*'s count by 1; else create a new node *N*, and let its count be 1, its parent link be linked to *T*, and its node-link be linked to the nodes with the same *item-name* via the node-link structure. If *P* is nonempty, call *insert_tree*(*P*, *N*) recursively.
    d. If reordering is needed (i.e. a "promotion" was detected) then call *reorder*() on the FP-tree.

The *reorder*() function is performed as follows:

1. Gather the "promoted" items into a *reorderList* ordered according to their support (descending) and lexicographical order.
2. Call *checkpoint*() to update the insertion order into the FP-tree.
3. For each item from *reorderList* go through the list of linked nodes and for each of these nodes call *moveUp*(*node*) to place that node into the correct position in the FP-tree, according to the *header*'s *master-table*.

The *moveUp*(*node*) function is defined as:

1. Repeat the steps (a. to g.) until the *node* and its current *parent* are in the *properOrder*
    a. Take the *node*'s *parent*'s parent (*pparent*)
    b. If *parent* has the same support as the *node*, remove the *parent* from its parent's *childNodes* and assign it to *newNode*
    c. Else perform the following actions:
        i. Create a *newNode* with the same item as the *parent*, but having the *node*'s support.
        ii. Link it into the *parent*'s list of nodes with the same item.
        iii. Adjust the support of the *parent*, by subtracting the *node*'s support
        iv. Remove the *node* from the *childNodes* of the *parent*
    d. Replace the *childNodes* into the *newNode* with the *childNodes* from the *node* and update the *parent* link of the *childNodes* with their new parent (*newNode*).
    e. Set the parent link of the *node* to *pparent* (the original *parent*'s parent), initialise its *childNodes* with the *newNode*, and set the *newNode*'s parent to *node*.
    f. (*optional step*) If there is already an *existingNode* for the *node*'s item in the *pparent*'s *childNodes*, then call *merge*(*existingNode*, *node*), and continue with the *existingNode* as the current *node*.
    g. Otherwise insert the *node* into the *childNodes* of *pparent*.

The resulting FP-tree is compatible for mining purposes with the original FP-growth algorithm described in [2]. Because we use the Dynamic-FPtree construction algorithm we renamed the FP-growth in to DynFP-growth..

4 Comparative Study

The three frequent pattern mining algorithms were implemented in Java and tested on several data sets. The platform's specifications used for this test was:

Pentium 4 1.7GHz processor, with 256 MBRAM, Windows 2000. In order to obtain more realistic results a Microsoft SQL 2000 Server was used and accessed through the standard ODBC interface. To study the performance and scalability of the algorithms generated data sets with 10,000 to 500,000 transactions, and support factors between 5% and 40% were used. Any transaction may contain more than one frequent itemset. The number of items in a transaction may vary, as well as the dimension of a frequent itemset. Also, the number of items in an itemset is variable. Taking into account these considerations, the generated data sets depend on the number of items in a transaction, number of items in a frequent itemset, etc. The necessary parameters to generate the test data sets are defined in Table 1:

|D|	Number of transactions
|T|	Average size of the transactions
|L|	Number of maximal potentially large itemsets
N	Number of items

Table 1. Parameters

The test data set is generated for a number of items N = 100 and a maximum number of frequent itemsets |L| = 3000. |T| was chosen to be 10.

Some of the results of the comparison between the Apriori, FP-growth and DynFP-growth algorithms for support factor of 5% and for different data sets are presented in Table 2:

Transactions (K)	Exec time (sec)		
	Apriori	DynFP-Growth	FP-growth
10	13.94	2.32	3.76
20	21.98	3.98	6.88
30	48.37	8.23	14.63
40	66.50	12.10	20.90
50	107.65	19.50	34.30
80	198.30	37.90	64.80
110	1471.40	55.00	95.50
150	3097.20	98.90	174.60
190	5320.60	152.70	273.60
300	9904.80	284.00	526.70
400	17259.20	458.10	849.70
520	20262.60	610.20	1150.70

Table 2. The results for support factor of 5%.

Table 2 shows that the execution time of the algorithms grows with the dimension of the data set. The best performance is obtained by the FP-growth algorithm.

Figure 2 shows that the execution time for the FP-growth algorithm is constant for a certain data set when the support factor decreases from 40% to 5% while, in the same time, the execution time of the Apriori algorithm increases dramatically. For a support factor of 30% or greater and a data set of 40,000 transactions, the Apriori algorithm has better performances than the FP-growth algorithm, but for a support factor of 20% or less its performance decreases dramatically. Thus, for a support

factor of 5% the execution time for the Apriori algorithm is three times longer than the execution time of the FP-growth algorithm and up to five times longer than DynFP-growth.

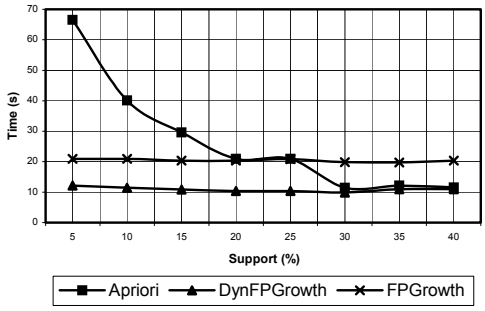

Figure 2. Scalability function of support for D1 40K database.

The execution time for the two algorithms for different values for the support factor on a data set with 150000 transactions is shown in Table 3. We notice that the Apriori algorithm has a lower performance than the FP-growth and DynFP-growth algorithms even for a support factor of 40%.

Support (%)	Exec time (sec)		
	Apriori	DynFP-Growth	FP-growth
5	3097.20	98.90	174.60
10	2186.10	91.20	170.80
15	1308.90	90.10	170.20
20	1305.60	89.00	170.90
25	870.50	89.00	171.30
30	875.00	89.50	172.50
35	440.00	88.90	169.80
40	441.60	90.10	175.20

Table 3. The results for D1 150K by support.

Figure 3 presents the execution time of the Apriori algorithm for different values of the support factor on different sized data sets. From here we notice that the performance of the algorithm is influenced by the dimensions of the data set and also by the support factor.

Figure 4 presents the execution time of the FP-growth algorithm for different values of the support factor on different sized data sets. From here we notice that the performance of the algorithm is depending only on the dimensions of the data set, the support factor having a very small influence.

It can be observed that the execution time of DynFP-Growth does not depend on support but only on the database size, this because the tree construction technique does not need the support information. In this way the tree will contain all the database transactions and depending on the required support the results will be

refined so that they will contain only the itemsets that have their frequency greater than the required support.

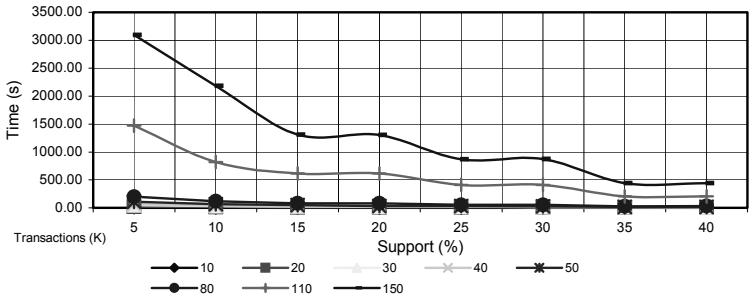

Figure 3. Apriori scalability by transactions/support.

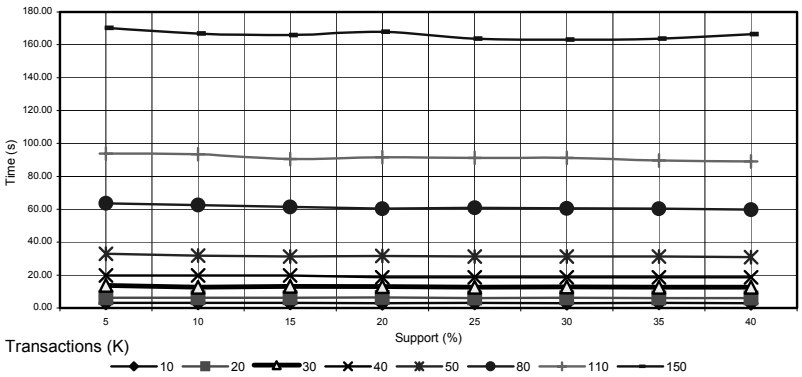

Figure 4. FP-growth scalability by transactions/support.

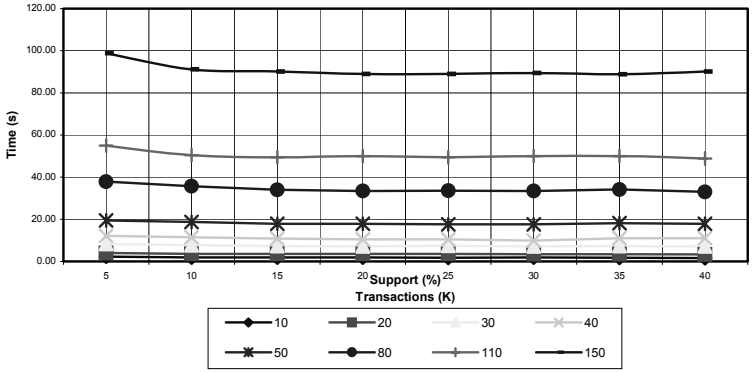

Figure 5. DynFP-growth scalability by transactions/support.

5 Conclusions

From the experimental data presented it can be concluded that the DynFP-growth algorithm behaves better than the FP-growth algorithm. First of all, the FP-growth algorithm needs at most two scans of the database, while the number of database scans for the candidate generation algorithm (Apriori) increases with the dimension of the candidate itemsets. Also, the performance of the FP-growth algorithm is not influenced by the support factor, while the performance of the Apriori algorithm decreases with the support factor.
Thus, the candidate generating algorithms (derived from Apriori) behave well only for small databases (max. 50,000 transactions) with a large support factor (at least 30%). In other cases the algorithms without candidate generation DynFP-growth and FP-growth behave much better.

References

[1] R. Agrawal, R. Srikant. "Fast algorithms for mining association rules in large databases". *Proc. of 20th Int'l conf. on VLDB*: 487-499, 1994.

[2] J. Han, J. Pei, Y. Yin. "Mining Frequent Patterns without Candidate Generation". *Proc. of ACM-SIGMOD*, 2000.

[3] C. Győrödi, R. Győrödi. "Mining Association Rules in Large Databases". *Proc. of Oradea EMES'02*: 45-50, Oradea, Romania, 2002.

[4] R. Győrödi, C. Győrödi. "Architectures of Data Mining Systems". *Proc. of Oradea EMES'02*: 141-146, Oradea, Romania, 2002.

[5] C. Győrödi, R. Győrödi, S. Holban, M. Pater. "Mining Knowledge in Relational Databases". *Proc. of CONTI 2002, 5th International Conference on Technical Informatics*: 1-6, Timisoara, Romania, 2002.

[6] M. H. Dunham. "Data Mining. Introductory and Advanced Topics". Prentice Hall, 2003, ISBN 0-13-088892-3.

[7] U.M. Fayyad, et al.: "From Data Mining to Knowledge Discovery: An Overview", *Advances in Knowledge Discovery and Data Mining*:1-34, AAAI Press/ MIT Press, 1996, ISBN 0-262-56097-6.

[8] J. Han, M. Kamber, "Data Mining Concepts and Techniques", Morgan Kaufmann Publishers, San Francisco, USA, 2001, ISBN 1558604898.

[9] L. Cristofor, "Mining Rules in Single-Table and Multiple-Table Databases", PhD thesis, University of Massachusetts, 2002.

[10] Ashoka Savasere, Edward Omiencinski, and Shamkant B. Navathe. "An Efficient Algorithm for Mining Association Rules in Large Databases." In Proceedings of the 21st International Conference on Very Large Databases, pag. 432 - 444, 1995.

[11] Cornelia Győrödi, Robert Győrödi, T. Cofeey & S. Holban – "Mining association rules using Dynamic FP-trees" – în Proceedings of The Irish Signal and Systems Conference, University of Limerick, Limerick, Ireland, 30th June-2nd July 2003, ISBN 0-9542973-1-8, pag. 76-82.