Fast Discovery of Frequent Patterns in Market Basket Data

Rená taIvá ncsy
Department of Automation and Applied Informatics
Budapest University of Technology and Economics
Goldmann Gy. tér 3, H-1111 Budapest
Hungary
renata.ivancsy@aut.bme.hu

Istvá n Vajk
Department of Automation and Applied Informatics
Budapest University of Technology and Economics
Goldmann Gy. tér 3, H-1111 Budapest
Hungary
vajk@aut.bme.hu

Abstract – **Mining frequent patterns in large transactional databases is a highly researched area in the field of data mining. The different existing association rule mining algorithms suffer from various problems when mining large amount of data. One major problem of the Apriori-like algorithms, which uses candidates, is the computational cost when filtering out the candidates not to be proven frequent. Another disadvantage of them is the high I/O cost. The main problem of the other algorithms is the high memory dependency when the algorithm assumes that gigantic data structures fit into the main memory, like the FP-growth algorithm does. In this paper a novel approach is introduced for solving the aforementioned issues. The main idea of the new method is to count the small itemsets in a very quick way, namely, by using a specific index structure. The suggested algorithm is partially based on the Apriori hypothesis and exploits the benefit of a new index table-based cubic structure to count the occurrences of the candidates. Experimental results show that the execution time of the proposed algorithm is significantly smaller than that of the Apriori and of the FP-growth algorithms. Its memory requirement is significantly lower than that of the FP-growth algorithm and it does not depend on the number of the processed transactions.**

I. INTRODUCTION

The task of association rule mining algorithms is to find hidden, previously unknown, and potentially useful relations in large amount of data. Since it was first introduced by Agrawal et al in [1] the problem of mining associations has received a great deal of attention. The problem is widely known as the market basket analysis; however several other applications exist which are searching for associations.

Traditionally the task of market basket analysis is to take a survey of the customer' s behavior in a suprmarket. The basket is understood as a set of items bought by the customer. This is also called as a transaction. The objective of the analysis is to find the items which are frequently bought together. These item sets are called frequent itemsets. If the number of the items in the set is k then it is called k-frequent itemset. A rule can be for instance the following: "*20* percent of the customers buying bread and milk also buy lunch meat" .If it is true for *2* percent of all the transactions, we say that the support of the rule is *2%* and the confidence of it is *20%*.

The task of association rule mining can be divided into two main steps. The first one is to find the frequent itemsets in the original dataset. The second step is to generate rules from the itemsets found during the first step. Most of the existing algorithm' saim is to find the frequent itemsets i.e. the frequent patterns in the transactions because of two reasons. The first reason is the much higher computational complexity of the frequent pattern discovery task then that of the rule generation. The frequent itemsets are discovered from the original database, which can be terabytes in size; meanwhile the rules are generated from the relatively small number of itemsets found by the first step. The second reason is that the approach of discovering frequent patterns is utilized in wide range of applications, for example for mining sequential patterns, episodes, partial periodicity and many other important data mining tasks.

The existing frequent pattern discovering algorithms suffer from several problems regarding the computational and I/O cost, and high memory requirements. The candidate "generate and test" algorithms, such theApriori [2] algorithm, have the drawback spending much of their time to drop the non-frequent candidates in each level. Another problem can be the high I/O cost which is inseparable form the level-wise approach. Several algorithms were developed which were based on the Apriori algorithm in order to reduce the affects of these problems. One of them is the DHP (Dynamic Hash and Prune) [3] algorithm which also collects information about the $(i+1)$-frequent itemsets using hash tables during the i-frequent itemset discovering step. In this way the cost of generating and testing candidates in the $(i+1)^{th}$ level is reduced. Another enhancement of the Apriori algorithm is the DIC (Dynamic Itemset Counting) [4] algorithm that counts the support dynamically of more than one sized itemset at the same time. In this way the number of the database reads can be reduced. There are several other methods contributed to improve the Apriori algorithm [5, 6, 7, 8].

Other algorithms do not use candidates, like the FP-growth [9] algorithm does. It compresses the database into the main memory and discovers the frequent patterns using a recursive tree traversing method. Its drawback is, however, the huge memory requirement which is dependent on the minimum support threshold and on the number and length of the transactions. An enhancement of the FP-growth algorithm is suggested in [10].

The suggested method in this paper is a candidate generating algorithm, but it has the advantage counting and testing the candidates quickly using an index structure. Its other advantage is the relatively small memory requirement that is dependent on the minimum support threshold and on the item number.

The organization of the paper is as follows. Section 2 defines the association rule mining process. Section 3 introduces the most common frequent pattern mining algorithms. The execution behaviors of the presented algorithms are analyzed in Section 4. In Section 5 the new proposed algorithm is explained, and experimental results are shown as well. We conclude in Section 6.

II. PROBLEM STATEMENT

The frequent pattern discovering approach, beside several applications, is mainly used in the association rule mining algorithms. The association rule mining problem is defined as follows. Let $I=\{i_1, i_2, i_3, \ldots i_n\}$ be the complete set of items appearing in the transactional database. An itemset X is a non-empty subset of I and it is called k-itemset if it contains exactly k items. A transaction T is a set of items such that $T \subseteq I$. Each transaction in the database has an identifier, called *TID*. A transaction T contains the itemset X if and only if $X \subseteq T$. An association rule is an implication of the form $X \Rightarrow Y$ where both X and Y are itemsets, and there exists no item which appears both in X and in Y, formally $X \subset I$, $Y \subset I$ and $X \cap Y = \emptyset$. An association rule has two properties: the support and the confidence. The support of a rule, denoted with s, is the percentage of the transactions in the database which contain both X and Y. The confidence, denoted with c, is the percentage of transactions in the database containing X that also contain Y. This is taken as a conditional probability, P(Y|X).

In order to reduce the number of the discovered association rules two thresholds are introduced, the minimum support (minSup) and minimum confidence (minConf) thresholds. An itemset is frequent, if its support exceeds the minimum support threshold. A rule is created from frequent itemsets, and a rule is a valid rule, if its confidence is above the given threshold.

III. BASIC ALGORITHMS

The frequent pattern discovering algorithms can be classified regarding several aspects. One essential aspect is the number of disk scans because of the high cost of the I/O operation. Another major aspect is whether the algorithm uses candidates or not. Regarding this aspects two basic algorithms are explained in this paper which approaches are fundamentally different. The Apriori algorithm is a level-wise method which uses candidates to generate the frequent itemsets and the FP-growth algorithm is a two phase method which does not use any candidates.

A. The Apriori algorithm

The most commonly known frequent pattern discovering algorithm is the Apriori algorithm proposed in [2]. It is a level-wise algorithm, which means that it finds the k-frequent itemsets during the k^{th} database scan. The Apriori algorithm is the basis of the level-wise algorithms which were developed since the introduction of it.

The main idea of the algorithm is based on the Apriori hypothesis, which is the following. Each itemset can be only frequent, if all its subsets are frequent as well. 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 when discovering the frequent patterns.

The algorithm works as follows. During the first database scan the frequent items are determined. This is done by simply counting the occurrences of each single item in the transactions. The algorithm assumes that the items both in the transactions and in the candidates in the further steps are in lexicographic order. From the frequent items two-sized candidates are generated by creating all the two combination of them by keeping the lexicographic order. During the second database scan the support of the candidates are counted. At the end of the database reading the counters for the candidates are checked and if a counter is above the minimum support value, the candidate becomes frequent. From the *2*-frequent itemsets *3*-candidates are generated regarding the following rule. If the two first items in the 2-itemsets are the same, a *3*-candidate is generated by keeping the lexicographic order. The joined itemset will be only a candidate if all its two-sized subsets are also frequent. In general, the rule for creating the *(i+1)* candidates from the *i*-frequent itemsets is to join two itemsets, if all *(i-1)* items are in common by keeping the lexicographic order. The joined itemset will be only a candidate, if all its *(i-1)*-sized subsets are also frequent. The algorithm terminates if no candidates can be generated. The pseudo code of the algorithm is depicted in Fig. 1 and in Fig. 2.

B. The FP-growth algorithm

One of the algorithms which do not use candidates is the FP-growth algorithm. It basically differs from the Apriori algorithm, because it reads the database only twice and it stores the pruned transactions in the main memory.

The FP-growth (Frequent Pattern-growth) algorithm [9] works as follows. During the first database scan it counts the occurrences i.e. the support of the items and orders them descending the support. During the second database scan it builds a so-called FP-tree structure in the main memory. A node of the tree contains an item, a counter for counting the support of the item and links to the child nodes, to the parent node and to the next sibling. A tree is built regarding the following rule. If the tree is empty, the transaction is inserted as the only branch in the tree.

```
procedure Apriori(minSup)
  L1 := find frequent 1-itemsets
  for (k=2; Lk-1 != null; k++)
    Ck := Apriori_gen(Lk-2)
    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 Lk
```

Fig. 1. The pseudo code of the Apriori algorithm

```
procedure Apriori_gen(Lk-1)
  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
```

Fig. 2. The Apriori_gen procedure

If the tree is not empty, while the first k item in a transaction fits the prefix of one of the branches of the tree, a counter in the branch is incremented in each referred node. 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 conditional FP-trees. This process is called the recursive pattern growth method. The pseudo code of the algorithm is illustrated in Fig. 3.

IV. COMPARISON OF THE ALGORITHMS

The experimental results presented in this paper are performed on the semantic datasets generated by the dataset generator downloaded from the IBM website. The datasets generated with this program accomplish the conditions introduced in [2]. The algorithms were implemented in C#. The simulations were executed on a Pentium 4 CPU, 2.40 GHz, and 1GB of RAM computer on .NET Framework v1.1. The naming conventions of the dataset are shown in Table 1. The number of the items that can occur in the transactions is 1000.

In order to compare the two basic algorithms and to detect their drawbacks, their execution times and memory requirements are to be investigated. A major aspect is which parameters of the dataset affect the behavior of the algorithms significantly. The two main parameters of a dataset are the item number, denoted with n, which is the number of items that can appear in the transactions, and the number of the transactions, denoted with T. Fig. 4 shows the execution times of the two algorithms as a function of the transaction number. It can be easily concluded that the execution time dependency of the Apriori algorithms on the transaction number is linear, and that of the FP-growth algorithm is rather a polynomial of two degree.

```
procedure FP_growth(Tree, alpha)
if Tree contains a single path P then
  for each beta := combinations of nodes in P do
     pattern:=beat U alpha;
     minsup := min(minsup of the nodes in beta);
else
  for each a_i int he header of Tree do
    generate pattern :=beta U alpha
    minsup := a_i.support;
    construct beta's conditional pattern base;
    FPTree := construct betas
              conditional FP-tree;
    if FPTree != 0 then
      FP_growth(FPTree, beta);
```

Fig. 3. The pseudo code of the FP-growth algorithm

Table 1. Meaning of the parameters in the names of the datasets

Name of the parameter	Meaning
T	The average length of the transactions
I	The average size of the maximal frequent itemsets
D	The number of transactions
K	Thousand

The memory requirement of the two algorithms is depicted in Fig. 5 as a function of the transaction number. It is obvious, that the memory requirement of the Apriori algorithm does not depend on the transaction number. The reason for that can be found in the candidate " generate and test" approach. The number of the candidates does not depend on the transaction number; it depends only on the item number and on the minimum support threshold.

The memory requirement of the FP-growth algorithm increases significantly with the growth of the transaction number. The reason for this can be found when examining the sizes of the trees which are 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. The claim is laid to modify the algorithm so that the created tree does not contain as redundant nodes as in the original case. The function between the transaction number and the size of the first generated tree is linear, which is shown in Fig. 6 by different minimum support thresholds.

The advantage of the FP-growth algorithm is the quick mining process which does not use candidates. Its drawback is, however, that the memory requirement of the algorithm is huge, especially by lower minimum support threshold.

The main problem of the candidate " generate and test" methods is the computational cost when filtering out the infrequent itemsets. Fig. 7 shows the execution time of the Apriori algorithm by itemset level when using T20I7D200K dataset. When investigating the execution times by itemset levels the fact is proved that the algorithm uses most of its time to discover the small frequent itemsets. In general it uses more than 70% of its execution time to discover the 4-frequent itemsets, and more than 50% of this time is used to find the 2-frequent itemsets. Its reason is the huge number of candidates in the first four levels. The candidate numbers in each single level are depicted in Fig. 8. It can be seen well that the number of the candidates in the second level is two orders of magnitude higher than in the further levels, however the number of the frequent itemsets, depicted in Fig. 9, are about the same.

The Apriori algorithm stores the candidates in a hash tree in order to quick find whether an itemset is a candidate or not. During the database scan each transaction is processed and its subsets are checked whether a counter belongs to it in the hash tree or not. This method is faster than finding the candidates by linear search, but in case of huge candidate number, using a hash tree is inefficient.

The number of the database access of the Apriori algorithm is the same as the size of the maximal frequent itemset. It accesses the database k times even than when only one k-frequent itemset exists. If the dataset is huge, the multiple database scans can be one of the drawbacks of the Apriori algorithm.

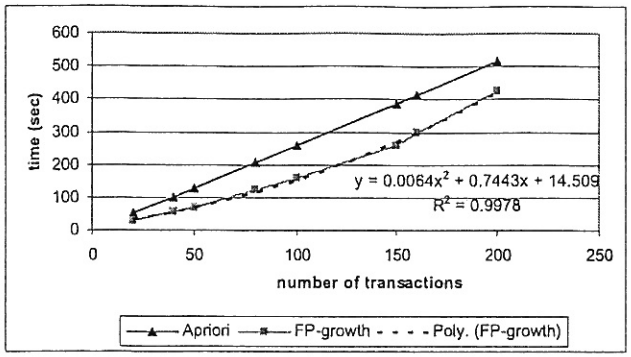

Fig. 4. Execution time of the two algorithms as a function of the transaction number by 0.9% minimum support threshold

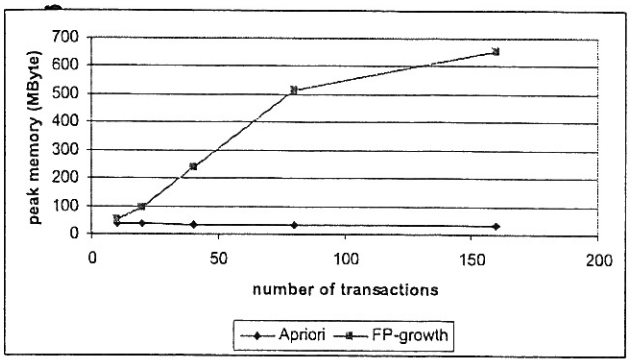

Fig. 5. Peak memory of the two algorithms as a function of the transaction number by 0.9% minimum support threshold

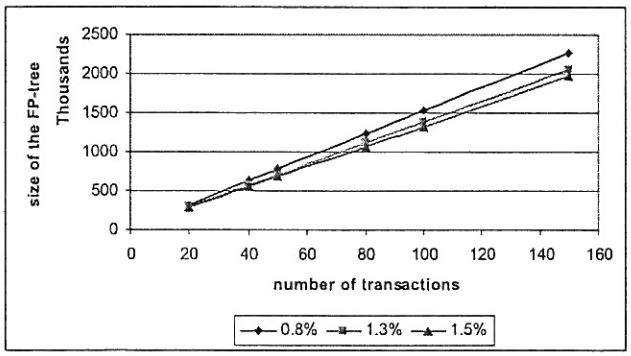

Fig. 6 . Sizes of the first generated FP-tree as a function of the transaction number by 0.8%, 1.3% and 1.5% minimum support thresholds

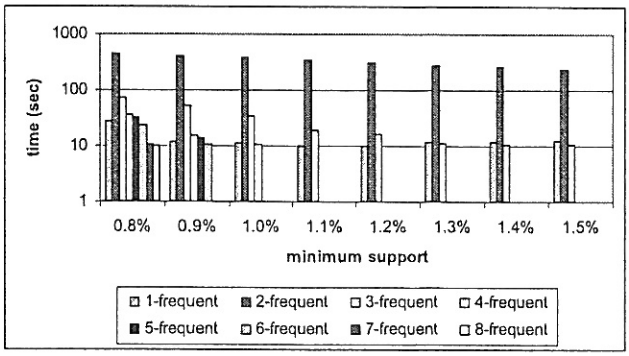

Fig. 7 Execution time on each level of the Apriori algorithm when using T20I7D200K dataset

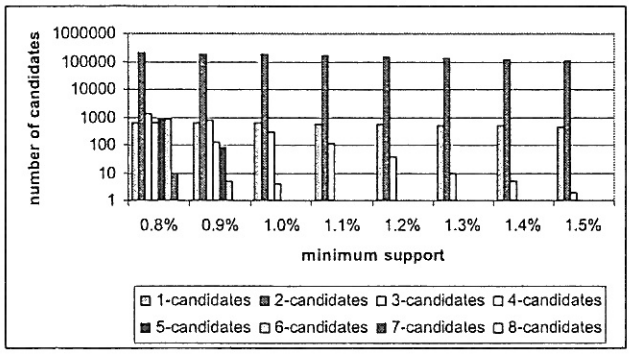

Fig. 8 Sizes of the candidates in each level when using T20I7D200K dataset

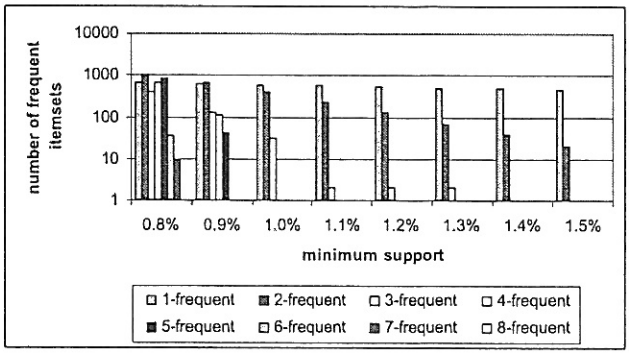

Fig. 9 Sizes of the frequent itemsets in each level when using T20I7D200K dataset

From these simulations the conclusion can be drawn that it is worth finding a method which discovers the small itemsets more quickly than the presented algorithms. In this case the way of finding all the itemsets is enhanced as well. Another expectation of the algorithm is that its memory requirement do not has to depend on the number of transactions.

V. THE CUBIC ALGORITHM

The main motivations of developing a new method are described in the previous section. The aim was to develop an algorithm, which memory usage is significantly lower than that of the FP-growth algorithm, and its execution time is much smaller than the execution time of both introduced algorithms. The main idea is to enhance the Apriori algorithm when discovering the small itemsets.

A. Description of the algorithm

The Cubic algorithm is a novel method to find the 4-frequent itemset quickly. The new method discovers the one and two frequent itemsets in one database reading using an upper triangular matrix, denoted with *M*. If the cardinality of the items in the database is denoted with *n*, the size of the matrix is $n*(n+1)/2$. The support counting of the candidates is made by a direct indexing method using this matrix and in this manner it is the fastest way.

The three and four frequent itemsets are counted during one further database scan. For efficient counting of the candidates, an index table-based cubic structure is built to hold them while the upper triangular matrix is traversed. One cube is created in order to store the 3 and 4-candidates which belong to the 2-frequent itemsets beginning with the same item. In this manner the first item of a candidate

selects the appropriate cube and the further items addresses the cells in the cube. The matrix M is processed by rows. The i^{th} row is only processed, if the value of the i^{th} diagonal element in M is greater than the minimum support threshold. In this case a new index table is created with size of n, and the values in the i^{th} row are checked whether they are over the threshold or not. If M[i,j] > minSup (i<j) the j^{th} element in the index structure is set to the number which will later index the cube. If all the elements of the i^{th} row are checked, a cube is created. The size of the cube is the number of the items in the i^{th} row, which value were over the minimum support threshold. A reverse index is created as well, in order to easy converting the index value, which addresses the cells in the cube, to the original item when traversing the cubes. This is used by the counter checking process.

During the second database scan every *3* and *4* subsets of the transactions are created, which has at least one *2*-frequent subset, and the appropriate element in the cube is incremented. The cube is selected by the first item of the subset. The other items address the counters in the cube using the index structure belonging to the selected cube.

The Apriori hypothesis is used only partially because of the following reason. The Apriori assumption is exploited when the algorithm creates different cubes for the itemsets having different first item. However it is not used when the edges of the cube are created. If the value of *M[i,j]* is greater than the minimum support threshold, the item *j* is added to the index table of the cube independently whether the elements *M[j,s]*, ($i<s<n$) are greater than the minimum support or not, where *s* denotes those items which satisfy the *M[i,s]* >minSup condition. The reason for this is that the storage space for the cube is rather compact, and there would not be any benefit discarding these items. In addition it would take more time to discard the item than to count its support. The main parts of the algorithm are depicted in Fig. 10 and in Fig. 11.

The Cubic algorithm discovers the 4-frequent itemsets. The further itemsets can be found in different ways. One of the possibilities is a level-wise approach, which simply invokes the Apriori algorithm. This is the easiest way and often a very quick solution because the Apriori algorithm finds the itemsets with cardinality greater than five relatively quick. Another way is to call the FP-growth algorithm. In this case the FP-tree must be generated only from thus transactions, which contains at least one 4-frequent itemset. In this way the profit is the smaller tree generated by the FP-growth algorithm thus, in general, the execution time is enhanced as well.

```
Procedure FillCubes()
  for each transaction t
    for (i=0; i<t.count; i++)
      if indexStruct[t[i]] == null continue
      for (j=i+1; j<t.count; j++)
        if M[t[i],t[j]] < minsup continue
        ind1 = IndexStruct[t[i]][t[j]]
        for (k=j+1; k<t.count; k++)
          ind2 = IndexStruct[t[i]][t[k]]
          if ind2 != -1
            CubeL[t[i]][ind1,ind2,0]++
              for (l=k+1; l<t.count; l++)
                ind3 = IndexStruct[t[i]][t[l]]+1
                if index3 != 0
                  CubeL[t[i]][ind1,ind2,ind3]++
```

Fig. 10 The candidate counting procedure in the Cubic algorithm

```
Procedure CheckCubes()
  for (i := 0; i < cubeL.count; i++)
    if cubeL[i] != null
      for (j=0; j<revInd[i].count; j++)
        for (k := j+1; k < revInd[i].count; k++)
          if cubeL[i][j; k; 0] >= minSup
            item2 = revInd[i][j]
            item3 = revInd[i][k]
            L3.Add(i,item2,item3)
            for (l=k+1; l<revInd[i].count; l++)
              if cubeL[i][j,k,l] > minSup
                item2 = revInd[i][j]
                item3 = revInd[i][k]
                item4 = revInd[i][l-1]
                L4.Add(i,item2,item3,item4)
```

Fig. 11 The candidate checking procedure of the Cubic algorithm

B. Simulation results

In Fig. 12 the execution time of the four algorithms is analyzed when using T20I7D200K dataset. It is clear, that the Cubic method continued by the Apriori algorithm, called Cubic Apriori algorithm, is the fastest of all the four methods. The execution time of the Cubic FP-growth method is always smaller than that of the Apriori algorithm but it is not always smaller, than the execution time of the FP-growth algorithm. The reason for that is illustrated in Fig. 13. The sizes of the first generated FP-trees are depicted in it in cases of the FP-growth and of the Cubic FP-growth algorithms when using T20I7D200K dataset as a function of the minimum support threshold. Apparently the sizes of the tree in case of small minimum support thresholds are near to each other, moreover by minimum support threshold of 0.5% they are about the same. It means that the Cubic FP-growth algorithm has to accomplish about the same recursive pattern growth process as the FP-growth algorithm does, but before this, the Cubic FP-growth algorithm has also to mine the 4-frequent itemsets using the Cubic method. In this case filtering the transactions by using the results of the Cubic algorithm causes no significant profit regarding the number of nodes in the tree. The saving in the node number is rather by minimum support threshold higher than 0.7%.

In Fig. 14 the peak memory sizes are illustrated as a function of the transaction number 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 0.9%. It is shown, that the memory requirement of the Cubic Apriori algorithm does not depend on the transaction number.

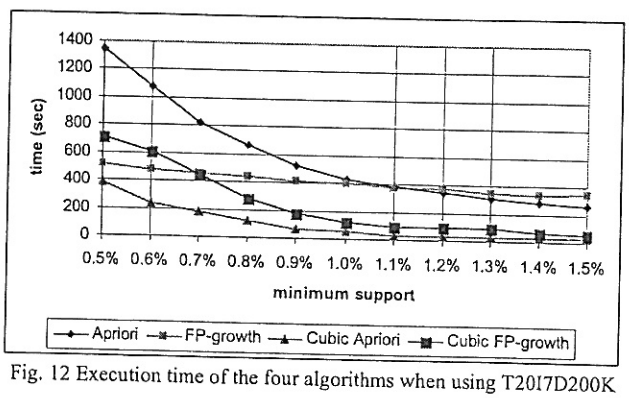

Fig. 12 Execution time of the four algorithms when using T20I7D200K

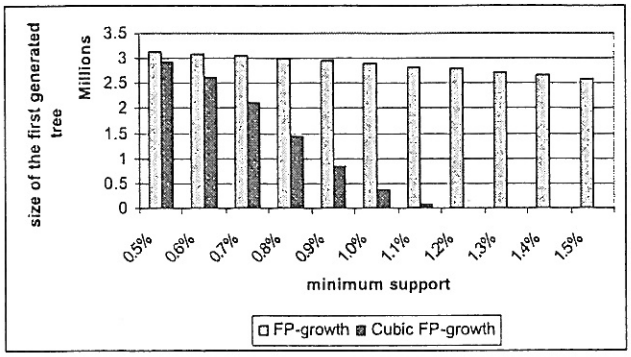

Fig. 13 Sizes of the first generated tree of the FP-growth and of the Cubic FP-growth algorithm when using T20I7D200K

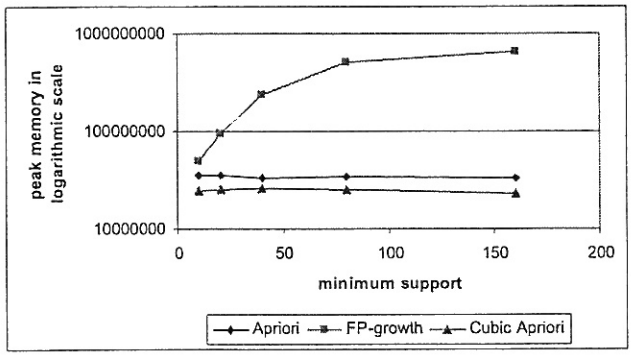

Fig. 14 Peak memory of the algorithms as a function of the transaction number by 0.9% minimum support threshold

VI. CONCLUSION

A novel method is proposed in this paper which finds the short frequent itemset quickly. After explaining the basic two association rule mining algorithm, the Apriori and the FP-growth algorithm in detail, the drawbacks of them were investigated. The main drawback of the Apriori algorithm is the slow candidate checking method using the hash tree data structure in case of small candidates, when the number of the candidates is high. The memory requirement dependency on the transaction number is proved as the major problem of the FP-growth algorithm. A novel method, the Cubic algorithm is presented in order to enhance the Apriori algorithm, by finding the short frequent patterns quickly, using an index table-based cubic structure. The algorithm exploits the benefits of direct indexing over the hash tree-based searching. Experimental result shows the time saving when replacing the first four steps of the Apriori algorithm with the novel method. In this way, the Cubic Apriori algorithm is even faster than the FP-growth algorithm, and the memory requirement of the novel method does not depend on the transaction number.

VII. ACKNOWLEDGMENT

This work has been supported by the fund of the Hungarian Academy of Sciences for control research and the Hungarian National Research Fund (grant number: T042741)

VIII. REFERENCES

[1] R. Agrawal, T. Imielinski and A.Swami: " Mining association rules between sets of items of large databases" *Proc. of the ACM SIGMOD Intl'l Conf. On Management of Data, Washington, D.C.,USA,* 1993, pp: 207-216

[2] R. Agrawal and R. Srikant: " Fast algorithms for mining association rules" *Proc. 20th Very Large Databases Conference, Santiago, Chile,* 1994, pp. 487-499

[3] J. S. Park, M. Chen, and P. S. Yu: " An effective hash based algorithm for mining association rules" *Proc. of the 1995 ACM Int. Conf. on Management of Data, San Jose, California, USA,* 1995, pp. 175-186

[4] S.Brin, R. Motawani, J.D. Ullman and S. Tsur: " Dynamic Item set counting and implication rules for market basket data" *Proc of the ACM SIGMOD Intl' l Conf. On Management of Data, Tucson, Arizona, USA,* 1997, pp. 255-264

[5] R. Agrawal and R. Srikant, "Fast algorithms for mining association rules" *Proc. 20th Very Large Databases Conference, Santiago, Chile,* 1994, 487-499

[6] M. J. Zaki, "Scalable algorithms for association mining" *IEEE Transaction on Knowledge and Data Engineering.* Vol 12. No 3. May/June 2000, 372-390

[7] V. S. Ananthanarayana, D. K. Subramanian and M. N. Murty "Scalable, distributed and dynamic mining of association rules" *Proceedings of the 7th International Conference on High Performance Computing - HiPC 2000 Bangalore, India,* December 17-20, 2000, pp 559-566

[8] R. J. Bayardo, "Efficiently mining long patterns from databases" *Proceedings of the ACM SIGMOD international conference on management of data, Seatle, WA,* June 1998, pp 85-93

[9] J.Han, J. Pei and Y. Yin: " Mining frequent patterns without candidate generation" *Proc. of the 2000 ACM-SIGMOD Int' l Conf.On Management of Data, Dallas, Texas, USA,* 2000, pp. 1-12

[10] M. El-Hajj and O. R. Zaï ane, "Non Recursive Generation of Frequent K-itemsets from Frequent Pattern Tree Representations" *in Proc. of 5th International Conference on Data Warehousing and Knowledge Discovery (DaWak'2003), Prague, Czech Republic,* September 3-5, 2003