Efficient Mining Algorithms of Finding Frequent Datasets

Lijuan Zhou
Department of Information Engineering College, Capital Normal University, Beijing, China
Zlj87@tom.com

Zhang Zhang
Department of Information Engineering College, Capital Normal University, Beijing, China
sherrytangtang@gmail.com

***Abstract*—This work proposes an efficient mining algorithm to find maximal frequent item sets from relational database. It adapts to large datasets.Itemset is stored in list with special structure. The two main lists called itemset list and Frequent itemset list are created by scanning database once for dividing maximal itemsets into two categories depending on whether the itemsets to achieve minimum support number. Sub itemsets whose superset is in itemset list are generated by recursion to make sure that each sub itemsets appeared before its superset. As current sub itemsets being joined to frequent itemset list, its sub itemsets are pruned from the itemset list. At last, all sub itemsets whose nearest superset is in frequent itemset list are pruned from the frequent itemset list to hold all maximal frequent itemsets.We compare our algorithms and FP-Growth by two sets of time-consuming experiments to prove the superiority of our efficient algorithm both not only with increasing datasets but also with changing mini-support.**

***Index Terms*—data mining; relational database; maximal frequent item sets**

I. Introduction

Association rules is an important data mining issue for describing the existing relationship among data items in the database. Association rules can reflect the complex and interesting links between the data in the database, digging out the useful links between data, for improving the practical work of great help. And the frequent item sets generation algorithm is the key to determine the efficiency of association rules mining.

Frequent itemset mining is a core data mining operation and has been extensively studied over the last decade [1]. Algorithms for frequent itemset mining form the basis for algorithms for a number of other mining problems, including association mining, correlations mining, and mining sequential and emerging patterns. Algorithms for frequent itemset mining have typically been developed for datasets stored in persistent storage and involve two or more passes over the dataset [2]. Recently, there has been much interest in reducing times of scanning and stored memory to improve algorithms.

Apriori is a seminal algorithm for finding frequent itemsets using candidate generation. It is characterized as a level-wise complete search algorithm using anti-monotonicity of itemsets, "if an itemset is not frequent, any of its superset is never frequent".

The most outstanding improvement over Apriori would be a method called FP-growth (frequent pattern growth) that succeeded in eliminating candidate generation. And there are several other dimensions regarding the extensions of frequent pattern mining such as using richer expressions than itemset. The algorithm proposed in this paper is based on the itemset set grid space theory and inherits the prefix-based search strategy of FP-growth.

Both Apriori and its improved algorithms spend lots of time generating candidates or take up large memory space to store redundant itemsets. This work describes a new algorithm of finding maximal frequent itemset by generating subsets instead of candidates. Specifically, our algorithm create itemset node from each IS only once and insert them in ISL or FISL only generating subset of those IS not frequent by travel ISL not scanning database again.

II. FP-growth Algorithms

It adopts a divide and conquer strategy by compressing the database representing frequent items into a structure called FP-tree (frequent pattern tree) that retains all the essential information and dividing the compressed database into a set of conditional databases, each associated with one frequent itemset and mining each one separately. It scans the database only twice. In the first scan, all the frequent items and their support counts (frequencies) are derived and they are sorted in the order of descending support count in each record. In the second scan, items in each record are merged into a prefix tree and items (nodes) that appear in common in different records are counted. Each node is associated with an item and its count. Nodes with the same label are linked by a pointer called node-link. Since items are sorted in the descending order of frequency, nodes closer to the root of the prefix tree are shared by more records, thus resulting in a very compact representation that stores all the necessary information. Pattern growth algorithm works on FP-tree by choosing an item in the order of increasing frequency and extracting frequent itemsets that contain

© 2012 ACADEMY PUBLISHER
doi:10.4304/jsw.7.4.727-732

the chosen item by recursively calling itself on the conditional FP-tree. FP-growth is an order of magnitude faster than the original Apriori algorithm [2].

III. EFFICIENT ALGORITHMS

This section describes our algorithm for once scanning database and time efficient frequent itemset mining.

We now describe our algorithm for maximal frequent itemset mining on relational database. The main algorithms description is:

```
(Database D)
 global List ISL;
 global List FISL;
 local Sequence IS;
 local SequencePoint ISP;
 ISL=NULL; FISL=NULL;
 foreach (IS∈ D)
       Join(IS);
 for all ISP of ISL
       for all sub sequences SIS of ISP->IS
             Join(SIS);
 for all ISP in n-level of FISL
       for all sub sequences SIS of ISP->IS
             Prune(SIS,FISL);
 Output(FISL);
END
```

And the important subroutines descriptions are:

```
Join (Sequence IS)
 local SequencePoint FISP;
 FISP=Find(IS,FISL,false);
 if (FISP!=NULL)
       FISP->count++;
    else
       Insert(IS);
END
Make_Fre (SequencePoint ISP)
 local int i;
 i=GetLevel(ISP->IS);
 if (ISP->count>=m_support_count)
       InsertToFISL(ISP);
       if (i==n)
             Prune(IS,ISL);
       else
          for all i-1 sub sequences SIS of ISP->IS
                ISP=Find(SIS,ISL,true);
                if (ISP!=NULL)
                      Prune(SIS,FISL);
                      ISP->count=0;
END
Find (Sequence IS, List L, bool ISLOrFISL)
 local int start_index, FD_index;
 local List lh;
 local SequencePoint ISP, FISP;
 start_index=GetStartIndex(IS);
 if (ISLOrFISL)
       lh=GetLevelHead(IS,ISL);
       ISP=lh->ISP[start_index];
       if (ISP==NULL)
             return ISP;
          else
             FD_index=start_index;
             while (ISP!=NULL)
                   FD_index=
GetFDIndex(ISP->IS,IS,FD_index);
                   if (FD_index==-1)
                         return ISP;
                   ISP=ISP->next[FD_index];
             return ISP;
    else
       lh=GetLevelHead(IS,FISL);
       FISP=lh->ISP[start_index];
       if (FISP==NULL)
             return FISP;
          else
             FD_index=start_index;
             while (FISP!=NULL)
                   FD_index=
GetFDIndex(FISP->IS,IS,FD_index);
                   if (FD_index==-1)
                         return FISP;
                   FISP=FISP->fre_next[FD_index];
             return FISP;
END
```

We initially introduce some terminology. We are mining a database of records D. Each record r in this database comprises a sequence of n items. The algorithm takes as input a parameter mini-support, i.e. the minimum frequency with which an itemset should occur to be considered frequent.

To store and manipulate the sub itemsets during any stage of the algorithm, two lists ISL and FISL is maintained.

IV. STORAGE AND CORE METHODS

In this section, we discuss the data structure and other optimizations used for efficiently implementing our algorithm. Particularly, we address the advantages in efficient execution of Insert and Delete operations.

A. *Stored level node and itemsets node*

An efficient data structure is required to maintain the itemsets. Frequent itemset mining implementations often use a prefix tree for this purpose. However, our algorithm requires the ability to delete the itemsets efficiently, which is not possible using a prefix tree. An obvious alternative is to use a multi-list. Firstly, itemsets of different lengths can be placed in sub lists in different level and stored there. However, this poses two problems. First, the time of finding an itemset can be quite high. Second, comparing two itemsets can be time consuming. Thus, we need a data structure that is compact, and can allow the following operations efficiently: 1) insertion of a new itemset 2) deletion of an itemset 3) incrementing the count of an itemset 4) traversal of the list. We have developed a new data structure, which we refer to as PrefixList.

Essentially, this data structure stores all itemsets using two lists. It has the benefit of easy deletion and insertion

© 2012 ACADEMY PUBLISHER

that a list allows, but it is also time-saving on finding itemsets.

The level node structure was shown as Figure1 and itemset node structure was shown as Figure 2.

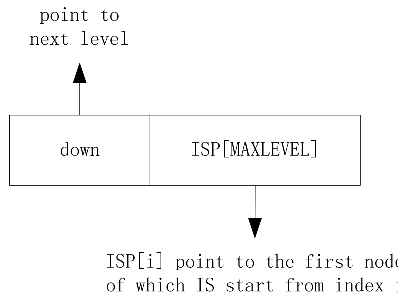

Figure 1. Level Node Structure

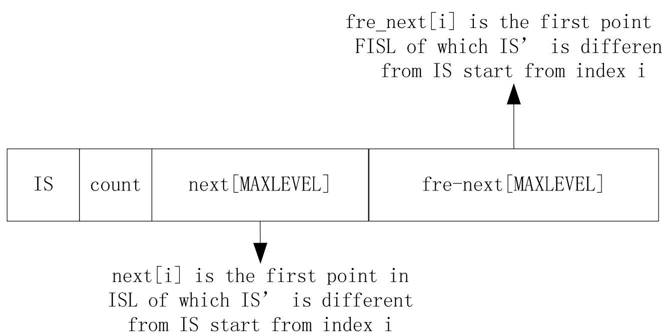

Figure 2. Itemset Node Structure

They are defined as following:

```
typedef struct ItemSeqNode
{
   ItemSequence IS[MAXLEVEL];
   int count;
   struct ItemSeqNode *next[MAXLEVEL];
   struct ItemSeqNode *fre_next[MAXLEVEL];
}*ItemSeqPoint;
typedef struct level
{
   struct level *down;
   ItemSeqPoint ISP[MAXLEVEL];
}*LevelPoint;
```

Mining dates are stored in rational database which is a set of tables containing data fitted into predefined categories. Each table (which is sometimes called a relation) contains one or more data categories in columns. Each row contains a unique instance of data for the categories defined by the columns.

By scanning the database, we are a first time for each IS to create an ISP and insert it to ISL. When the same IS appears to this ISP count is incremented by 1, and if the count is reaching to the mini-support count it is pruned from ISL at the same time joined to FISL. So in the scanning, we depart itemsets to two kinds one not frequent joining to itemset list and others to frequent itemset list.

And then travel itemsets list (ISL) in which the itemsets are not frequent to generate k-level (i range from 1 to n) sub itemsets (SIS) from small to large for finding frequent ones at the same time prune its k-1 sub itemsets from the frequent itemset list.

At the end of algorithm, all maximal frequent itemsets are stored in the FISL to output.

B. Core methods

Two key operations in implementing our algorithm are inserting itemsets that occur frequently into FISL and deleting ones that do not occur frequently.

Insert operations are divided into two steps: location and connection. Location through 2-3 steps to achieve the results divided into three kinds. First, locate to level node through the number of items in the sequences as 'lh'. Then, locate to the head node of sub list by getting index of item 'lh->ISP[start_index]'. Last, if needed locate to the insert position of list. In these steps, search results can be described as the following three kinds of situations: 1) not find the same or similar itemset 2) find the similar itemset with which it was different start at index i 3) find the same itemset. According to different location result, connection of current itemset point and list happens in the former two situations. For the first situation it would be the head point of the sub list and for the second one it would be the ith next point of the last similar itemset.

We also do not remove the infrequent itemset every time. Subsequence routine is invoked. The deletion happens in the following three ways. First, when we insert an existed n-itemset whose count reaching to mini-support count to FISL, it will be removed from ISL. Secondly, after each k-sub itemset generated we remove all frequent (k-1)-sub itemsets of it in the FISL by examining the k-1 level sub list. Finally, when the traversal of ISL is finished, frequent sub itemsets whose n-superset is frequent are removed.

V. EXPERIMENTAL RESULTS

We use two sets of experiments to compare our algorithm with FP-Growth in time performance. The first set of experiments fixed the minimum support to compare efficiency by the number of itemsets range from 200 to2000, and the second set of experiments fixed the number of itemsets to make comparison in different minimum support.

Data source is mushroom dataset. In the two sets of experiments we select the former 6 attributes of each record as itemsets.

Figure 3 to 6 shows the experimental results that compare the execution time with increasing dataset size. For comparing our algorithm against the fp-tree based algorithm, the mini-support range from 0.2 to 0.02, we can see that algorithm in time-efficient were significantly increased with FP-growth in variety mini-support, and as the data sets increases, the execution time of algorithm varying more steady than FP-growth.

Figure 3 shows that when the minimum support degree to take 0.2, execution time has been better than the FP-growth. As the number of itemsets increase, the FP-growth execution time is growing, while the MFISS-FP remaining at around 10ms. In Figure 4, the minimum support is taken to be 0.1, as the itemsets

© 2012 ACADEMY PUBLISHER

number increasing, the implementation cost of FP-growth significantly rises, while the able to maintain stable and far superior to FP-growth. As shown in Figure 5, as the itemsets number increases in the minimum support of 0.05, the implementation cost of FP-growth is almost linear growth, but is always less than and next to 15ms. In Figure 6 the minimum support degree taking to 0.02, we can see as the FP-growth implementation cost more and more time up to 125ms with increasing itemsets, at the same situation the can still remain the implementation cost at less than 15ms. Comparing Figure 3, 4, 5 and 6 we can sum up, as the minimum support degree between 0.02 and 0.2, the execution time of can be always maintained at less than 16ms, and ensure its stability to avoid obvious growth with the increasing number of itemsets; and the implementation cost of FP-growth is almost linear growth as the number of itemsets increasing, and the execution time up to 125ms when the itemsets number get to 2000. By contrast can be drawn, with changing minimum support, performance is far superior to FP-growth, and has a very high stability.

Figure 7 to 10 shows the experimental results that compare the execution time with changing mini-support. In this set of experiment, data sets were selected 8K, 4K, 2K and 1K from which we find that algorithm is more time-efficiency than FP-growth especially as mining large number of instances. This is because our algorithms have better stability.

As shown in Figure 7, in the experiment with size of 8k data set, and FP-growth has been stability in the implementation cost, but with the changing minimum support degree, implement to cost stably around 60ms, however the FP-growth has the execution time between 390ms-440ms. Figure 8 shows the comparison of against FP-growth in time consuming under the experimental data sets of size 4k, it can be seen the minimum support ranging from 0.2 to 0.05, the implementation cost of FP-growth has continued growth while remaining stable as the mini-support between 0.05 to 0.01. can maintain its execution time around 26ms in different level of mini-support. In the Figure 9 experimental data set size of 2k, with changing support, the implementation cost of FP-growth presents a growing state, while nearly hold about 15ms. Figure 10 describes the experimental results with 1k size of experimental data set, from which we can see that as minimum support changing, FP-growth keep its execution time about 63ms, but the maximum time of is 12.4ms when the minimum support being 0.1. Through the comparison of these four figures can be seen in different data set size have been markedly better than the FP-growth.

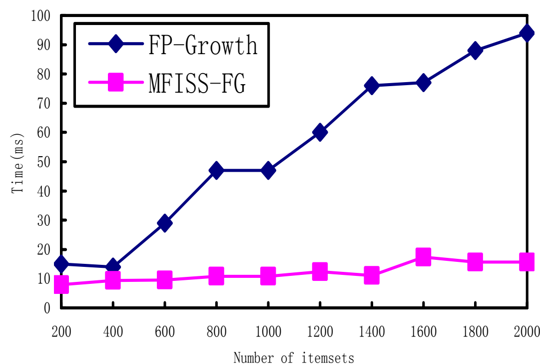

Figure 3. Execution Time with Increasing Dataset Size (mini-support=0.2)

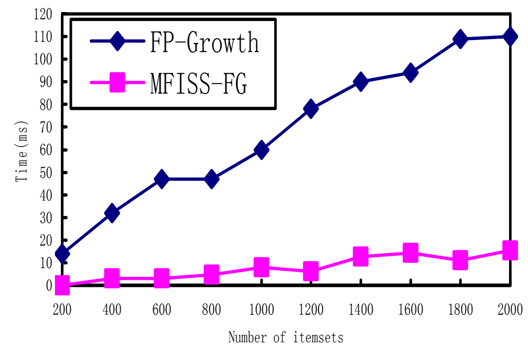

Figure 5. Execution Time with Increasing Dataset Size (mini-support=0.05)

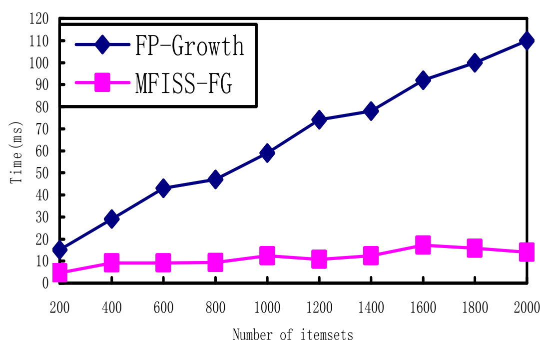

Figure 4. Execution Time with Increasing Dataset Size (mini-support=0.1)

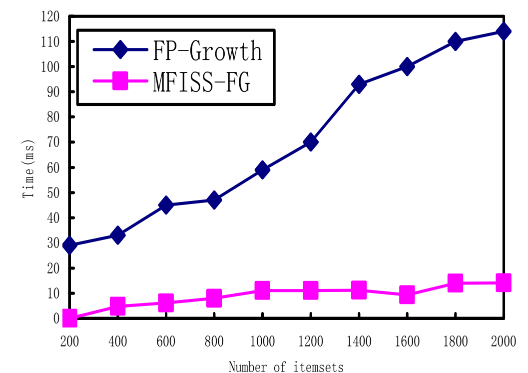

Figure 6. Execution Time with Increasing Dataset Size (mini-support=0.02)

© 2012 ACADEMY PUBLISHER

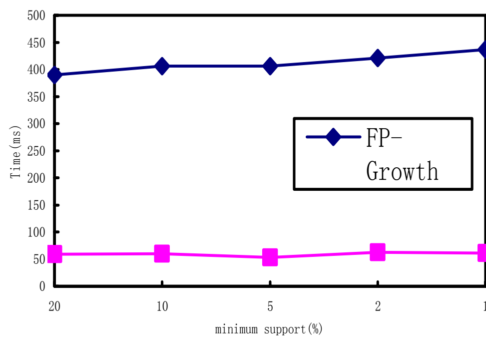

Figure 7. Execution Time with Changing Mini-support (8K Dataset)

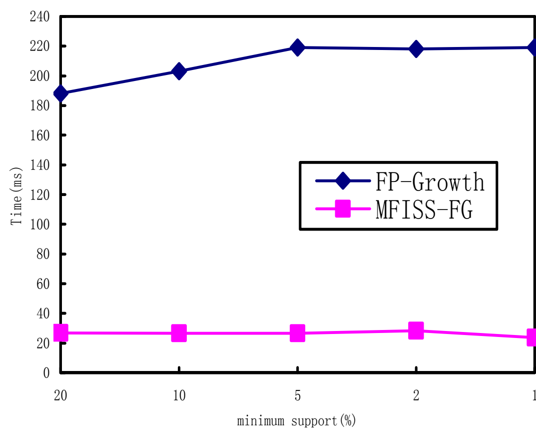

Figure 8. Execution Time with Changing Mini-support (4K Dataset)

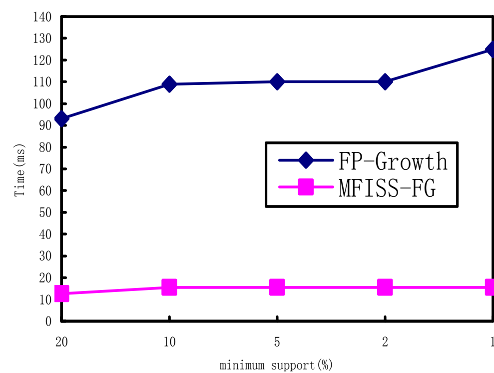

Figure 9. Execution Time with Changing Mini-support (2K Dataset)

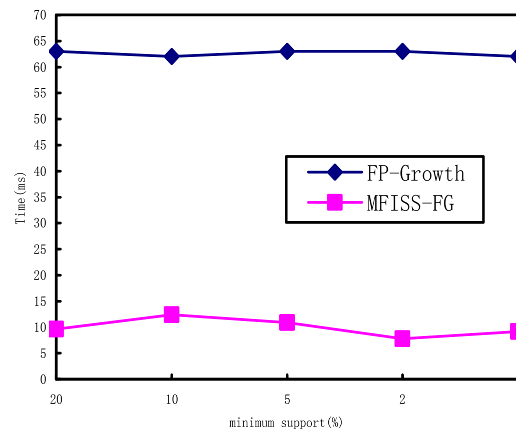

Figure 10. Execution Time with Changing Mini-support (1K Dataset)

VI. CONCLUSIONS

In this paper, one of the most important ideas to improve the speed of algorithm is to use a new data structure adapting to maximal frequent itemsets mining. And by using recursion to generate sub itemsets determine whether the current itemset is maximal frequent itemsets. We inherited the Join and Prune strategy of ISS-DM to create a new algorithm based on our own data structure. In addition, the searching speed was greatly increased by using prefix-list evolved from FP-growth. At last, we prove the superiority of our own new algorithm by comparing with FP-Growth through two sets of experiments on time-consuming.

ACKNOWLEDGEMENTS

This research was supported by China National Key Technology R&D Program (2009BADA9B02),

This research was supported by Beijing Nature Science Foundation (4092011) ，

This research was supported by Beijing Educational Committee science and technology development plan project(KM200810028016),

This research was supported by the Open Project Program of Key Laboratory of Digital Agricultural Early-warning Technology，Ministry of Agriculture, Beijing, 100037.

REFERENCES

[1] Jin R, Agrawal G. An algorithm for in-core frequent itemset mining on streaming data. In Proceeding of the 2005 international conference on data mining (ICDM'05), Houston, 2005, TX, pp 210–217.

[2] Wu Xindong, Vipin Kumar, J. Ross Quinlan, Joydeep Ghosh, Yang Qiang, Hiroshi Motoda, Geoffrey J. McLachlan, Angus Ng, Liu Bing, Philip S. Yu, Zhou Zhihua, Michael Steinbach, David J. Hand, Dan Steinberg. "Top 10 algorithms in data mining", Knowl Inf Syst (2008) 14:1–37.

[3] Agrawal Ret al. Fast algorithms for mining association rules. In: Proc the 20th International Conference on Very Large Data Bases. Santiago de Chile, 1994. 478-499.

[4] Nicolas Pasquieret al. Efficient mining if association rules using closed itermset lattices. Information Systems, 1999, 24(1):25-46.

[5] Agrawal Ret al. Mining association rule between sets of items in large database. In: Proc the ACM SIGMOD International Conference on Management of Data, Washington, 1993. 207-216.

[6] Cheung Det al. Efficient mining of association rules in distribut-ed databases. IEEE Trans Knowledge and Data Engineering, 1996,8(6):911-922.

© 2012 ACADEMY PUBLISHER

[7] Chen M Set al. Data mining:An overview from a database per-spective. IEEE Trans Knowledge and Data Engineering, 1996, 8(6):866-883.
[8] Han J W, Yin Y W. Mining frequent patterns without candidate generation. In: Proc SIGMOD Conference, 2000, 1-12.
[9] Cheng J Het al. Multi-strategy approach to mining interesting rules. Chinese Journal of Computers, 2000, 23(1): 47-51(in Chinese).
[10] Zhu Jiaxian. “A Mining Algorithm of Association Rule Based on Linked List”. Journal of Shaoxing University, 2004, 8(24):19-22, 59.
[11] Bao Zhengyi, Wang Zhoujing. “MLCI Algorithm for Mining Lower Closed Itemsets”, Computing Technology and Automation, 2005.12,4(24):73-76.
[12] Yang Qiang, Wu Xindong. “10 Challenging Problems in Data Mining Research”, International Journal of Information Technology & Decision Making, 2006, 4(5):597-604.
[13] Han Wang, Lingfu Kong, "A Constrained Maximum Frequent Itemsets Incremental Mining Algorithm", Network and Parallel Computing Workshops, IFIP International Conference on, pp. 743-747, 2007 IFIP International Conference on Network and Parallel Computing Workshops (NPC 2007), 2007.
[14] Mao Guojun, Liu Chunnian. Mining of Association Rules Based on the Operators of Set of Itemsets, Chinese Journal of Computers, 2002, 25(4): 417-422(in Chinese).
[15] Wang Xianjun, Song Jingjing, Jiang Baoqing. Mining frequent closed itemsets in unidirectional FP-tree. Computer Engineering and Applications, 2008, 44(10): 150- 153(in Chinese).

© 2012 ACADEMY PUBLISHER