Proceedings of the 2007 IEEE Symposium on Computational Intelligence and Data Mining (CIDM 2007)

Using the Pre-FUFP Algorithm for Handling New Transactions in Incremental Mining

Chun-Wei Lin , Tzung-Pei Hong, and Wen-Hsiang Lu

***Abstract*—In the past, we proposed a Fast Updated FP-tree (FUFP-tree) structure to efficiently handle new transactions and to make the tree update process become easier. In this paper, we attempt to modify the FUFP-tree construction based on the concept of pre-large itemsets. Pre-large itemsets are defined by a lower support threshold and an upper support threshold. The proposed approach can achieve a good execution time for tree construction especially when each time a small number of transactions are inserted. Experimental results also show that the proposed Pre-FUFP maintenance algorithm has a good performance for incrementally handling new transactions.**

I. Introduction

Many algorithms for mining association rules from transactions have been proposed, most of which were based on the Apriori algorithm [1][2][3], which generated and tested candidate itemsets level-by–level. This may cause iterative database scans and high computational costs. Han *et al*. thus proposed the Frequent-Pattern-tree (FP-tree) structure for efficiently mining association rules without generation of candidate itemsets [7]. The FP-tree [7] was used to compress a database into a tree structure which stored only large items. Both the Apriori and the FP-tree mining approaches belong to batch mining. In real-world applications, new transactions are usually inserted into databases incrementally. In this case, the originally desired large itemsets may become invalid, or new large itemsets may appear in the resulting updated databases [4][5][11][13][15]. Designing an efficient algorithm that can maintain association rules as a database grows is thus critically important.

One noticeable incremental mining algorithm was the Fast-Updated Algorithm (called FUP), which was proposed by Cheung *et al*. [4] for avoiding the shortcomings mentioned above. Although the FUP algorithm could indeed improve mining performance for incrementally growing databases, original databases still needed to be scanned when necessary. A pre-large-itemset algorithm was thus proposed to further reduce the need for rescanning original database based on two support thresholds [8]. The algorithm did not need to rescan the original database until a number of new transactions have been inserted. Since rescanning the database spent much computation time, the maintenance cost could thus be reduced in the pre-large-itemset algorithm.

In the past, Hong *et al*. [9] modified the FP-tree structure and designed the fast updated frequent pattern trees (FUFP-trees) to efficiently handle newly inserted transactions based on the FUP concept. The FUFP-tree structure was similar to the FP-tree structure except that the links between parent nodes and their child nodes were bi-directional. Besides, the counts of the sorted frequent items were also kept in the Header_Table of the FP-tree algorithm.

In this paper, we attempt to further modify the FUFP-tree algorithm for incremental mining based on the pre-large concept [8]. Based on two support thresholds, the proposed approach can effectively handle cases in which itemsets are small in an original database but large in newly inserted transactions. Experimental results also show that the proposed maintenance algorithm has a good performance for incrementally handling new transactions.

II. Review of Related works

A. The FUFP-tree algorithm

The FUFP-tree construction algorithm is the same as the FP-tree algorithm [7] except that the links between parent nodes and their child nodes are bi-directional. Bi-directional linking will help fasten the process of item deletion in the maintenance process. Besides, the counts of the sorted frequent items are also kept in the Header_Table.

An FUFP tree must be built in advance from the original database before new transactions come. When new transactions are added, the FUFP-tree maintenance algorithm will process them to maintain the FUFP tree. It first partitions items into four parts according to whether they are large or small in the original database and in the new transactions. Each part is then processed in its own way. The Header_Table and the FUFP-tree are correspondingly updated whenever necessary.

Several other algorithms based on the FP-tree structure have been proposed. For example, Qiu *et al*. proposed the QFP-growth mining approach to mine association rules [12]. Mohammad proposed the COFI-tree structure to replace the conditional FP-tree [14]. Ezeife constructed a generalized FP-tree, which stored all the large and non-large items, for incremental mining without rescanning databases [6]. Koh *et al*. adjusted FP trees also based on two support thresholds [10], but with a more complex adjusting procedure and spending more computation time than the one proposed in this paper. Some related researches are still in progress.

This research was supported by the National Science Council of the Republic of China under contract NSC 95-2221-E-390-025.

C. W. Lin is with the Department of Computer Science and Information Engineering, National Cheng Kung University, Tainan, 701, Taiwan, R.O.C. (e-mail: p78951228@mail.ncku.edu.tw).

T. P. Hong is with the Department of Electrical Engineering, National University of Kaohsiung, Kaohsiung, 811, Taiwan, R.O.C. (corresponding author; phone: +886+7+5919191; fax: +886+7+5919374; e-mail: tphong@nuk.edu.tw).

W. H. Lu is with the Department of Computer Science and Information Engineering, National Cheng Kung University, Tainan, 701, Taiwan, R.O.C. (e-mail: whlu@mail.ncku.edu.tw).

B. The pre-large-itemset algorithm

A pre-large itemset is not truly large, but may be large with a high probability in the future. Two support thresholds, a lower support threshold and an upper support threshold, are used to realize this concept. The upper support threshold is the same as that used in the conventional mining algorithms. On the other hand, the lower support threshold defines the lowest support ratio for an itemset to be treated as pre-large.

Considering an original database and transactions which are newly inserted by the two support thresholds, itemsets may fall into one of the following nine cases illustrated in Figure 1.

Cases 1, 5, 6, 8 and 9 will not affect the final association rules according to the weighted average of the counts. Cases 2 and 3 may remove existing association rules, and cases 4 and 7 may add new association rules. If we retain all large and pre-large itemsets with their counts after each pass, then cases 2, 3 and case 4 can be handled easily. Also, in the maintenance phase, the ratio of new transactions to old transactions is usually very small. This is more apparent when the database is growing larger. It has been formally shown that an itemset in case 7 cannot possibly be large for the entire updated database as long as the number of transactions is smaller than the number f shown below [8]:

$$f = \left\lfloor \frac{(S_u - S_l)d}{1 - S_u} \right\rfloor,$$

where f is the safety number of the new transactions, S_u is the upper threshold, S_l is the lower threshold, and d is the number of original transactions.

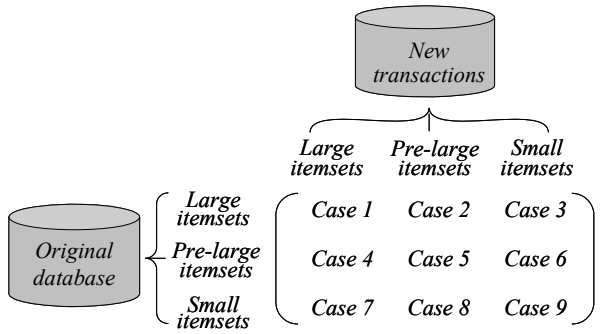

Fig.1. Nine cases arising from adding new transactions to existing databases

III. THE PROPOSED MAINTENANCE ALGORITHM

An FUFP tree must be built in advance from the initially original database before new transactions come. Its initial construction is similar to that of an FP tree. The database is first scanned to find the items with their supports larger than a predefined minimum support. These items are called large items. Next, the large items are sorted in descending frequency. At last, the database is scanned again to construct the FUFP tree according to the sorted order of large items. The construction process is executed tuple by tuple, from the first transaction to the last one. After all transactions are processed, the FUFP tree is completely constructed. Besides, a variable c is used to record the number of new transactions since the last re-scan of the original database with d transactions. The details of the proposed algorithm are described below.

The Pre-FUFP maintenance algorithm:

INPUT: An old database consisting of (d+c) transactions, its corresponding Header_Table storing the frequent items initially in descending order, its corresponding FUFP tree, a lower support threshold S_l, an upper support threshold S_u, its corresponding pre-large table storing the set of pre-large items from the original database, and a set of t new transactions.

OUTPUT: A new FUFP tree for the updated database by using the Pre-FUFP maintenance algorithm.

STEP 1: Calculate the safety number f of new transactions according to the following formula [8]:

$$f = \left\lfloor \frac{(S_u - S_l)d}{1 - S_u} \right\rfloor.$$

STEP 2: Scan the new transactions to get all the items and their counts.

STEP 3: Divide the items in the new transactions into three parts according to whether they are large, pre-large or small in the original database.

STEP 4: For each item I from STEP 3, which is large in the original database (appearing in the Header_Table), do the following substeps (Cases 1, 2 and 3):

Substep 4-1: Set the new count $S^U(I)$ of I in the entire updated database as:

$$S^U(I)=S^D(I)+S^T(I),$$

where $S^D(I)$ is the count of I in the Header_Table (original database) and $S^T(I)$ is the count of I in the new transactions.

Substep 4-2: If $S^U(I)/(d+c+t) \geq S_u$, update the count of I in the Header_Table as $S^U(I)$, and put I in the set of *Insert_Items*, which will be further processed in STEP 10;
Otherwise, if $S_u \geq S^U(I)/(d+c+t) \geq S_l$, remove I from the Header_Table, connect each parent node of I directly to its child node in the corresponding FUFP tree, set $S^D(I) = S^U(I)$, and keep I with $S^D(I)$ in the pre-large table;
Otherwise, item I is small after the database is updated; remove I from the Header_Table and connect each parent node of I directly to its child node in the corresponding FUFP tree.

STEP 5: For each item I from STEP 3 which is pre-large in the original database, do the following substeps (Cases 4, 5 and 6):

Substep 5-1: Set the new count $S^U(I)$ of I in the entire updated database as:

$$S^U(I)=S^D(I)+S^T(I).$$

Substep 5-2: If $S^U(I)/(d+c+t) \geq S_u$, item I will be large after the database is updated; put

I in the set of *Insert_Items* and *Branch_Items*, which will be further processed in STEP 8;
Otherwise, if $S_u \geq S^U(I)/(d+c+t) \geq S_l$, set $S^D(I) = S^U(I)$ and keep *I* with the new $S^D(I)$ in the pre-large table;
Otherwise, remove item *I* from the pre-large table.

STEP 6: For each item *I* from STEP 3 which is neither large nor pre-large in the original database but large or pre-large in the new transactions (Cases 7 and 8), put *I* in the set of *Rescan_Items*, which is used when rescanning the database in STEP 7 is necessary.

STEP 7: If $t+c \leq f$ or the set of *Rescan_Items* is null, then do nothing;
Otherwise, do the following substeps for each item *I* in the set of *Rescan_Items*:
Substep 7-1: Rescan the original database to decide the original count $S^D(I)$ of *I*.
Substep 7-2: Set the new count $S^U(I)$ of *I* in the entire updated database as:
$$S^U(I)=S^D(I)+S^T(I).$$
Substep 7-3: If $S^U(I)/(d+c+t) \geq S_u$, item *I* will become large after database is updated, put *I* in the set of *Insert_Items* and *Branch_Items*;
Otherwise, if $S_u \geq S^U(I)/(d+c+t) \geq S_l$, set $S^D(I) = S^U(I)$ and keep *I* with $S^D(I)$ in the pre-large table;
Substep 7-4: Otherwise, neglect *I*.

STEP 8: Insert the items in the *Branch_Items* to the end of the Header_Table according to the descending order of their updated counts.

STEP 9: For each original transaction with an item *I* existing in the *Branch_Items*, if *I* has not been at the corresponding branch of the FUFP tree for the transaction, insert *I* at the end of the branch and set its count as 1; Otherwise, add 1 to the count of the node *I*.

STEP 10: For each new transaction with an item *I* existing in the *Insert_Items*, if *I* has not been at the corresponding branch of the FUFP tree for the new transactions, insert *I* at the end of the branch and set its count as 1; Otherwise, add 1 to the count of the node *I*.

STEP 11: If $t+c > f$, then set $d = d+t+c$ and set $c = 0$; otherwise, set $c = t+c$.

In STEP 9, a *corresponding branch* is the branch generated from the large items in a transaction and corresponding to the order of items appearing in the Header_Table. After STEP 11, the final updated FUFP tree by using the Pre-FUFP maintenance algorithm is constructed. The new transactions can then be integrated into the original database. Based on the FUFP tree, the desired association rules can then be found by the FP-Growth mining approach as proposed in [7].

IV. AN EXAMPLE

In this section, an example is given to illustrate the proposed Pre-FUFP algorithm for maintaining an FUFP tree when new transactions are inserted. Table 1 shows a database to be used in the example. It contains 10 transactions and 9 items, denoted *a* to *i* .

TABLE 1
THE ORIGINAL DATABASE IN THE EXMAPLE

Old database	
Transaction No.	Items
1	*a, b, c, d, e, g, h*
2	*a, b, f, g*
3	*b, d, e, f, g*
4	*a, b, f, h*
5	*a, b, f, i*
6	*a, c, d, e, g, h*
7	*a, b, h, i*
8	*b, c, d, f, g*
9	*a, b, f*
10	*a, b, g, h*

Assume the lower support threshold S_l is set at 30% and the upper one S_u at 50%. For the given database, the large 1-itemsets are *a*, *b*, *f*, *g* and *h*, from which the Header_Table can be constructed. The FUFP tree is then formed from the database and the Header_Table, with the results shown in Figure 3. Besides, the sets of pre-large items for the given database are shown in Table 2.

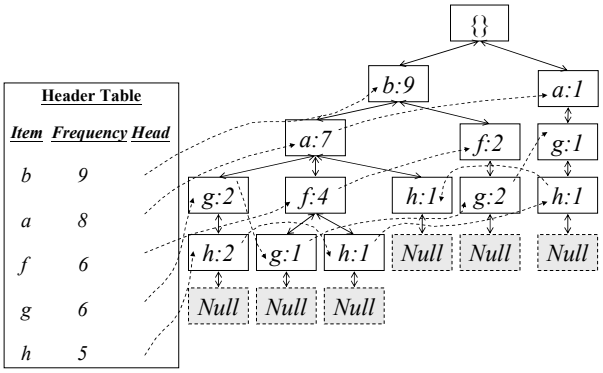

Fig. 3. The Header_Table and the FUFP tree constructed

TABLE 2
THE PRE_LARGE ITEMSET FOR THE ORIGINAL DATABASE

Pre-large itemset in the original database	
Items	Count
c	3
d	4
e	3

Assume the three new transactions shown in Table 3 appear.

TABLE 3
THE THREE NEW TRANSACTIONS

Transaction No.	Items
1	*a, b, d, f, i*
2	*a, b, d, i*
3	*a, c, d, h, i*

The proposed Pre-FUFP maintenance algorithm proceeds as follows. The variable *c* is initially set at 0.

STEP 1: The safety number *f* for new transactions is calculated as:

$$f = \left\lfloor \frac{(S_u - S_l)d}{1 - S_u} \right\rfloor = \left\lfloor \frac{(0.5 - 0.3)10}{1 - 0.5} \right\rfloor = 4$$

STEP 2: The three new transactions are first scanned to get the items and their counts.

STEP 3: All the items *a* to *i* in new transactions are divided into three parts, $\{a\}\{b\}\{f\}\{g\}\{h\}$, $\{c\}\{d\}\{e\}$, and $\{i\}$ according to whether they are large (appearing in the Header_Table), pre-large (appearing in the pre-large table) or small in the original database. Results are shown in Table 4, where the counts are only from the new transactions.

TABLE 4
THE THREE NEW TRANSACTIONS

Large items in the original database		Pre-large items in the original database		Small items in the original database	
Items	Count	Items	Count	Items	Count
a	3	*c*	1	*i*	3
b	2	*d*	3		
f	1	*e*	0		
g	0				
h	1				

STEP 4: The items in the new transactions which are large in the original database are first processed. In this example, items *a*, *b*, *f*, *g*, and *h* (the first partition) satisfy the condition and are processed. The support ratios of items *a*, *b* and *f* are larger than 0.5. Take item *a* as an example to illustrate the substeps. The count of item *a* in the Header_Table is 8, and the count in the new transactions is 3. The new count of item *a* is thus 8+3 (= 11). The new support ratio of item *a* is 11/(10+0+3) 0.5. Item *a* is thus still a large item after the database is updated. The frequency value of item *a* in the Header_Table is thus changed as 11, and item *a* is then put into the set of Insert_Items. Items *b* and *f* are similarly processed. Next, both the support ratios of items *g* and *h* are smaller than 0.5 but larger than 0.3. Items *h* and *g* will become pre-large after the database is updated. Take item *h* as an example. Item *h* is removed from the Header_Table and its corresponding FUFP tree, and put in the pre-large table with its updated count as 6. In this case, the FUFP tree needs to be processed as well. The results after item *h* is processed are shown in Figure 4.

STEP 5: The items in the new transactions which are pre-large in the original database are processed. In this example, items *c*, *d* and *e* satisfy the condition and are processed. Take item *d* first as an example to illustrate the supsteps. The count of item *d* in the pre-large itemset is 4, and its count in the new transactions is 3. The new count of item *d* is thus 4+3 (= 7). The new support ratio of item *d* is 7/(10+0+3) 0.5. Item *d* will thus become a large item after the database is updated. *d* is then put into the set of *Insert_Items* and *Branch_Items*. The new support ratio of item *c* is 0.4, which is between the lower and the upper thresholds. Item *c* is then put into the pre-large table and its count is updated as 4. At last, the new support ratio of item *e* is small than 0.3. Item *e* is thus removed from the pre-large table. After STEP 5, we can get *Insert_Items* = {*a*, *b*, *f*, *d*} and *Branch_ Items* = {*d*}.

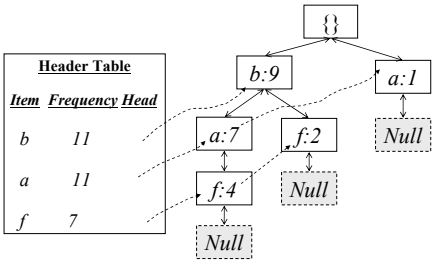

Fig.4. The Header_Table and the FUFP tree after STEP 4

STEP 6: Since the item *i* is neither large nor pre-large in the original database but large in the new transactions, it is put into the set of *Rescan_Items*, which is used when rescanning in STEP 7 is required. After STEP 6, *Rescan_Items* = {*i*}.

STEP 7: Since $t+c = 3+0 < f$ (= 4), rescanning the original database is unnecessary. Nothing is done in this step.

STEP 8: The items in the set of *Branch_Items* are sorted in descending order of their updated counts and then inserted into the end of the Header_Table. In this example, the set of *Branch_Items* contains only *d*, and no sorting is needed. Item *d* is thus inserted into the end of the Header_Table.

STEP 9: The FUFP tree is updated according to the original transactions with items existing in the *Branch_Items*. In this example, *Branch_Items* = {*d*}. The corresponding branches for the original transactions with *d* are show in Table 5.

TABLE 5
THE CORRESPONDING BRANCHES FOR THE ORIGINAL TRANSACTIONS WITH ITEM *d*

Transaction No.	Items	Corresponding branches
1	*a, b, c, d, e, g, h*	*b, a, d*
3	*b, d, e, f, g*	*b, f, d*
6	*a, c, d, e, g, h*	*a, d*
8	*b, c, d, f, g*	*b, f, d*

The first branch is then processed. This branch shares the same prefix (*b*, *a*) as the current FUFP-tree. A new node (*d*:1) is thus created and linked to (*a*:7) as its child. The same process is then executed for the other three corresponding branches. The final results are shown in Figure 5.

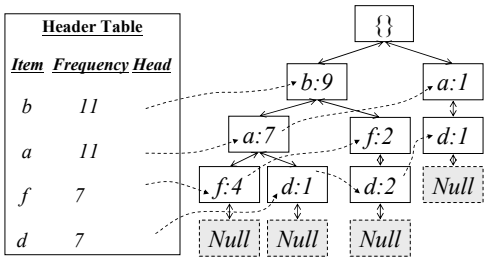

Fig. 5. The Header_Table and the FUFP tree after STEP 9

STEP 10: The FUFP tree is updated according to the new transactions with items existing in the *Insert_Items*. In this example, *Insert_Items* = {*a*, *b*, *f*, *d*}. The corresponding branches for the new transactions with any of these items are shown in Table 6.

TABLE 6
THE CORRESPONDING BRANCHES FOR THE NEW TRANSACTIONS

Transaction No.	Items	Corresponding branches
1	*a*, *b*, *d*, *f*, *i*	*b*, *a*, *f*, *d*
2	*a*, *b*, *d*, *i*	*b*, *a*, *d*
3	*a*, *c*, *d*, *h*, *i*	*a*, *d*

The first branch shares the same prefix (*b*, *a*, *f*) as the current FUFP tree. The counts for items *b*, *a*, and *f* are then increased by 1 since they have not yet counted in the construction of the previous FUFP tree. The same process is then executed for the other two branches. The final results are shown in Figure 6.

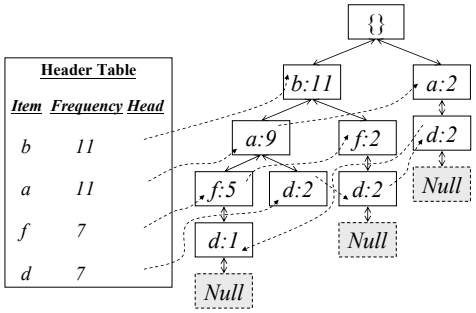

Fig. 6. The Final FUFP tree after all the new transactions are processed

STEP 11: Since $t\,(=3) + c\,(=0) < f\,(=4)$, set $c = t+c = 3+0 = 3$.

After STEP 11, the FUFP tree are updated. Note that the final value of *c* is 3 in this example and *f* - *c* = 1. This means that one more new transaction can be added without rescanning the original database for Case 7. Based on the FUFP tree shown in Figure 5, the desired large itemsets can then be found by the FP-Growth mining approach as proposed in [7].

V. Experimental Results

Experiments were made to compare the performance of the batch FP-tree construction algorithm, the FUFP-tree maintenance algorithm and the Pre-FUFP maintenance algorithm. When new transactions came, the batch FP-tree construction algorithm integrated new transactions into the original database and constructed a new FP-tree from the updated database. The process was executed whenever new transactions came. The incremental FUFP-tree maintenance algorithm and the Pre-FUFP maintenance algorithm processed new transactions incrementally in the way mentioned in Sections 2.A and 3.

The experiments were performed in C++ on an Intel x86 PC with a 3.0G Hz processor and 512 MB main memory and running the Microsoft Windows XP operating system. A real dataset called BMS-POS [16] was used in the experiments. This dataset was also used in the KDDCUP 2000 competition. The BMS-POS dataset contained several years of point-of-sale data from a large electronics retailer. Each transaction in this dataset consisted of all the product categories purchased by a customer at one time. There were 515,597 transactions with 1657 items in the dataset. The maximal length of a transaction was 164 and the average length of the transactions was 6.5.

The first 500,000 transactions were extracted from the BMS-POS database to construct an initial FP-tree. The value of the minimum threshold was set at 1% to 5% for the three algorithms, with 1% increment each time. The next 2,000 transactions were then used in incremental mining. For the Pre-FUFP maintenance algorithm, the upper minimum support threshold was set at 1% to 5% (1% increment each time) and the lower minimum support threshold was set at 0.5% to 2.5% (0.5% increment each time). The execution times and the numbers of nodes obtained from the three algorithms were compared. Figure 7 shows the execution times of the three algorithms for different threshold values.

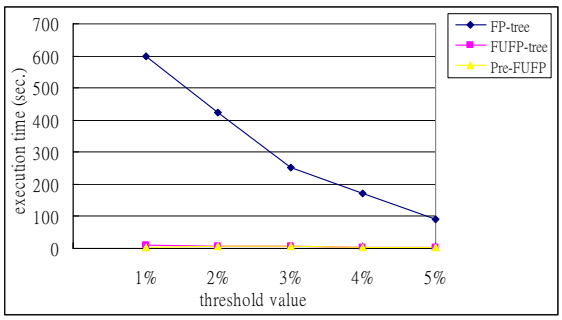

Fig. 7. The comparison of the execution times for different threshold values

It can be observed from Figure 7 that the proposed Pre-FUFP maintenance algorithm ran faster than the other two. The comparison of the numbers of nodes for the three algorithms is given in Figure 8. It can be seen that the three

algorithms generated nearly the same sizes of trees. The effectiveness of the Pre-FUFP maintenance algorithm is thus acceptable.

VI. CONCLUSION

In this paper, we have proposed the Pre-FUFP maintenance algorithm for incremental mining based on the concept of pre-large itemsets. It first partitions items of new transactions into three parts according to whether they are large, pre-large or small in the original database. Each part is then processed in its own way. The Header_Table and the FUFP-tree are correspondingly updated whenever necessary. Experimental results also show that the proposed Pre-FUFP maintenance algorithm runs faster than the batch FP-tree and the FUFP-tree construction algorithm for handling new transactions and generates nearly the same tree structure as them.

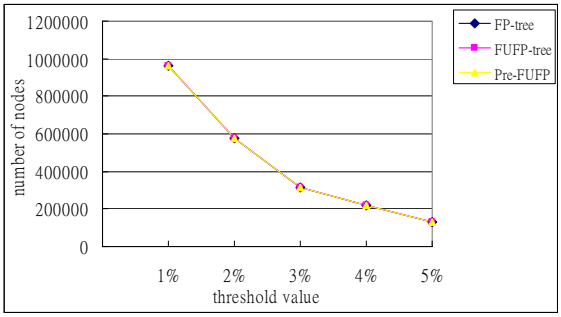

Fig.8. The comparison of the numbers of nodes for different threshold values

REFERENCES

[1] R. Agrawal, T. Imielinksi and A. Swami, "Mining association rules between sets of items in large database," *The ACM SIGMOD Conference*, pp. 207-216, Washington DC, USA, 1993

[2] R. Agrawal, T. Imielinksi and A. Swami, "Database mining: a performance perspective," *IEEE Transactions on Knowledge and Data Engineering*, pp. 914-925, 1993.

[3] R. Agrawal and R. Srikant, "Fast algorithm for mining association rules," *The International Conference on Very Large Data Bases*, pp. 487-499, 1994.

[4] D.W. Cheung, J. Han, V.T. Ng and C.Y. Wong, "Maintenance of discovered association rules in large databases: An incremental updating approach," *The Twelfth IEEE International Conference on Data Engineering*, pp. 106-114, 1996.

[5] D.W. Cheung, S.D. Lee and B. Kao, "A general incremental technique for maintaining discovered association rules," *In Proceedings of Database Systems for Advanced Applications*, pp. 185-194, 1997.

[6] C. I. Ezeife, "Mining Incremental association rules with generalized FP-tree," *Proceedings of the 15th Conference of the Canadian Society for Computational Studies of Intelligence on Advances in Artificial Intelligence*, pp. 147-160, 2002.

[7] J. Han, J. Pei and Y. Yin, "Mining frequent patterns without candidate generation," *The 2000 ACM SIGMOD International Conference on Management of Data*, pp. 1-12, 2000.

[8] T. P. Hong, C. Y. Wang and Y. H. Tao, "A new incremental data mining algorithm using pre-large itemsets," *Intelligent Data Analysis*, Vol. 5, No. 2, 2001, pp. 111-129.

[9] T. P. Hong, J. W. Lin and Y. L. Wu, "A fast updated frequent pattern tree," *The IEEE International Conference on Systems, Man, and Cybernetics*, pp.2167-2172, 2006.

[10] J. L. Koh and S. F. Shieh, "An efficient approach for maintaining association rules based on adjusting FP-tree structures," *The Ninth International Conference on Database Systems for Advanced Applications*, pp. 417-424, 2004.

[11] M. Y. Lin and S. Y. Lee, "Incremental update on sequential patterns in large databases," *The Tenth IEEE International Conference on Tools with Artificial Intelligence*, pp. 24-31, 1998.

[12] Y. Qiu, Y. J. Lan and Q. S. Xie, "An improved algorithm of mining from FP- tree," *Proceedings of the Third International Conference on Machine Learning and Cybernetics*, pp. 26-29, 2004.

[13] N. L. Sarda and N. V. Srinivas, "An adaptive algorithm for incremental mining of association rules," *The Ninth International Workshop on Database and Expert Systems*, pp. 240-245, 1998.

[14] O. R. Zaiane and E. H. Mohammed, "COFI-tree mining: A new approach to pattern growth with reduced candidacy generation," *IEEE International Conference on Data Mining*, 2003.

[15] S. Zhang, "Aggregation and maintenance for database mining," *Intelligent Data Analysis*, pp. 475-490, 1999.

[16] Z. Zheng, R. Kohavi and L. Mason, "Real world performance of association rule algorithms," *The International Conference on Knowledge Discovery and Data Mining*, pp. 401-406, 2001.