Maintenance of the Prelarge Trees for Record Deletion

Chun-Wei Lin[†], Tzung-Pei Hong[‡], and Wen-Hsiang Lu[†]
[†] Department of Computer Science and Information Engineering
National Cheng Kung University
Tainan, 701, Taiwan, R.O.C.

[‡] Department of Computer Science and Information Engineering
National University of Kaohsiung
Kaohsiung, 811, Taiwan, R.O.C.

Abstract: - The frequent pattern tree (FP-tree) is an efficient data structure for association-rule mining without generation of candidate itemsets. It, however, needed to process all transactions in a batch way. In addition to record insertion, record deletion is also commonly seen in real-application. In this paper, we propose the structure of prelarge trees for efficiently handling deletion of records based on the concept of pre-large itemsets. Due to the properties of pre-large concepts, the proposed approach does not need to rescan the original database until a number of records have been deleted. The proposed approach can thus achieve a good execution time for tree construction especially when a small number of records are deleted each time. Experimental results also show that the proposed approach has a good performance for incrementally handling deleted records.

Key-Words: - data mining, FP-tree, Prelarge-tree algorithm, pre-large itemsets, record deletion.

1 Introduction

Many algorithms for mining association rules from transactions were proposed, most of which were based on the Apriori algorithm [1], which generated and tested candidate itemsets level-by-level. This may cause iterative database scans and high computational costs. Han *et al.* proposed the Frequent-Pattern-tree (FP-tree) structure for efficiently mining association rules without generation of candidate itemsets [3]. It was condensed and complete for finding all the frequent patterns. The construction process was executed tuple by tuple, from the first transaction to the last one. Both the Apriori and the FP-tree mining approaches belong to batch mining. That is, they must process all the transactions in a batch way. In real-world applications, new transactions are usually inserted into databases incrementally.

One noticeable incremental mining algorithm was the Fast-Updated Algorithm (called FUP), which was proposed by Cheung *et al.* [2] 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.

In the past, Hong *et al.* thus proposed the pre-large concept to further reduce the need for rescanning original database [4]. A pre-large itemset was defined based on two support thresholds. The upper support threshold was the same as that used in the conventional mining algorithms. The lower support threshold defined the lowest support ratio for an itemset to be treated as pre-large. An itemset with its support ratio below the lower threshold was thought of as a small itemset. The algorithm did not need to rescan the original database until a number of new transactions had been inserted. Since rescanning the database spent much computation time, the maintenance cost could thus be reduced in the pre-large-itemset algorithm.

Hong *et al.* also 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 [5]. 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 proposed the structure of Prelarge tree for handling the deletion of records based on the concept of pre-large itemsets [4]. A structure of prelarge tree is to keep not only frequent items but also pre-large items. Based on the pre-large itemsets, the proposed approach can effectively handle cases in which itemsets are small both in an original database and deleted records. The proposed algorithm does not require rescanning the original

databases to construct the prelarge tree until a number of deleted records have been processed. Experimental results also show that the proposed algorithm has a good performance for incrementally handling deleted records.

2 Review of Related Works

In this section, some related researches are briefly reviewed. They are the FUFP-tree algorithm and the pre-large-itemset algorithm.

2.1 The FUFP-tree algorithm

The FUFP-tree construction algorithm is based on the FP-tree algorithm [3]. The links between parent nodes and their child nodes are, however, 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.

2.2 The pre-large-itemsets algorithm

Hong *et al.* proposed the pre-large concept to reduce the need of rescanning original database [4] for maintaining association rules. A pre-large itemset is not truly large, but may be large with a high probability in the future. 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. The support ratio of an itemset must be larger than the upper support threshold in order to be considered large. On the other hand, the lower support threshold defines the lowest support ratio for an itemset to be treated as pre-large. An itemset with its support ratio below the lower threshold is thought of as a small itemset. Pre-large itemsets act like buffers and are used to reduce the movements of itemsets directly from large to small and vice-versa.

Considering an original database and some records to be deleted by the two support thresholds, itemsets may fall into one of the following nine cases illustrated in Figure 1.

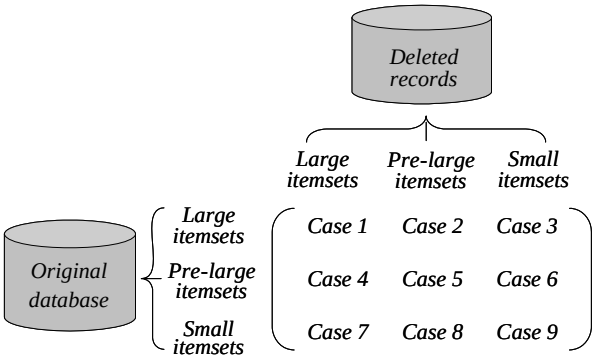

Figure 1: Nine cases arising from and the original database and the deleted records

Cases 2, 3, 4, 7 and 8 above will not affect the final association rules. Case 1 may remove some existing association rules, and cases 5, 6 and 9 may add some new association rules. If we retain all large and pre-large itemsets with their counts after each pass, then cases 1, 5 and 6 can be handled easily. Also, in the maintenance phase, the ratio of deleted records 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 9 cannot possibly be large for the entire updated database as long as the number of transactions is smaller than the number f shown below [4]:

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

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

3 The Proposed Deletion algorithm

A prelarge tree must be built in advance from the initially original database before the records are deleted from the original databases. Its initial construction is stated as follows. The database is first scanned to find the large items which have their supports larger than the upper support threshold and the pre-large items which have their minimum supports lie between the upper and lower support thresholds. Next, the large and the pre-large items are sorted in descending frequencies. The database is then scanned again to construct the prelarge tree according to the sorted order of large and pre-large items. The construction process is executed tuple by tuple, from the first transaction to the last one. After all transactions are processed, the prelarge tree is completely constructed. The frequency values of large items and pre-large items are kept in the Header_Table and Pre_Header_Table, respectively. Besides, a variable c is used to record the number of deleted records since the last re-scan of the original

database with d transactions. The details of the proposed algorithm are described below.

The prelarge-tree deletion algorithm:

INPUT: An old database consisting of (d-c) transactions, its corresponding Header_Table and Pre_Header_Table, its corresponding prelarge tree, a lower support threshold S_l, an upper support threshold S_u, and a set of t deleted records.

OUTPUT: A new prelarge tree after records is deleted by using the prelarge tree deletion algorithm.

STEP 1: Calculate the safety number f of deleted records according to the following formula [4]:

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

STEP 2: Scan the deleted records to get all the items and their counts.

STEP 3: Divide the items in the deleted records into three parts according to whether they are large (appearing in the Header_Table), pre-large (appearing in the Pre_Header_Table) or small (not in the Header_Table or in the Pre_Header_Table) in the original database.

STEP 4: For each item I which is large in the original database, 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 deleted records.

Substep 4-2: If $S^U(I)/(d\text{-}c\text{-}t) \geq S_u$, update the count of I in the Header_Table as $S^U(I)$, and put I in the set of *Reduced_Items*, which will be further processed in STEP 6;

Otherwise, if $S_u \geq S^U(I)/(d\text{-}c\text{-}t) \geq S_l$, remove I from the Header_Table, put I in the head of Pre_Header_Table with its updated frequency $S^D(I)$, and keep I in the set of *Reduced_Items*;

Otherwise, item I is still 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 prelarge tree.

STEP 5: For each item I 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 $S^U(I)/(d\text{-}c\text{-}t) \geq S_u$, item I will be large after the database is updated; remove I from the Pre_Hedaer_Table, put I with its new frequency $S^D(I)$ in the end of Header_Table, and put I in the set of *Reduced_Items*;

Otherwise, if $S_u \geq S^U(I)/(d\text{-}c\text{-}t) \geq S_l$, item I is still pre-large after the database is updated; update I with its new frequency $S^D(I)$ in the Pre_Header_Table and put I in the set of *Reduced_Items*;

Otherwise, remove item I from the Pre_Header_Table.

STEP 6: For each deleted record with an item J existing in the *Reduced_Items*, substract 1 from the count of J node at the corresponding branch of the prelarge tree.

STEP 7: For each item I which is neither large nor pre-large in the original database but small in the deleted records (Cases 9), put I in the set of *Rescan_Items*, which is used when rescanning the database in STEP 8 is necessary.

STEP 8: 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 8-1: Rescan the original database to decide the original count $S^D(I)$ of I.

Substep 8-2: 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 8-3: If $S^U(I)/(d\text{-}c\text{-}t) \geq S_u$, item I will become large after the database is updated; put I in the set of *Branch_Items* and insert the items in the *Branch_Items* to the end of the Header_Table according to the descending order of their updated counts;

Otherwise, if $S_u \geq S^U(I)/(d\text{-}c\text{-}t) \geq S_l$, item *I* will become pre-large after the database is update; put *I* in the set of *Branch_Items*, and Insert the items in the *Branch_Items* to the end of the Pre_Header_Table according to the descending order of their updated counts.
Otherwise, do nothing.

Substep 8-4: For each original transaction with an item *J* existing in the Branch_Items, if *J* has not been at the corresponding branch of the prelarge tree for the transaction, insert *J* at the end of the branch and set its count as 1; Otherwise, add 1 to the count of the node *J*.

STEP 9: If $t + c > f$, then set d = d - t - c and set $c = 0$; otherwise, set $c = t + c$.

In STEP 8, a corresponding branch is the branch generated from the large and pre-large items in a transaction and corresponding to the order of items appearing in the Header_Table and the Pre_Header_Table. After STEP 9, the final updated prelarge tree is maintained by the proposed algorithm. The records can then be deleted from the original database. Based on the prelarge tree, the desired association rules can then be found by the FP-Growth mining approach as proposed in [3] on only the large items.

4 An Example

In this session, an example is given to illustrate the proposed deletion algorithm for maintaining a prelarge tree when records are deleted. 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 example

Old database	
TID	Items
1	*b, c, e*
2	*b, c, e, g*
3	*a, b, d, e, h*
4	*a, b, e, g, h*
5	*a, e, g*
6	*a, b, e*
7	*b, d, e, g*
8	*a, b, c, f*
9	*a c, d, f*
10	*c, f, i*

Assume the lower support threshold S_l is set at 30% and the upper one S_u at 50%. Here, not only the frequent items are kept in the prelarge tree but also the pre-large items. For the given database, the large items are *b*, *e*, *a* and *c*, and the pre-large items are *d*, *g* and *f*, from which the Header_Table and the Pre_Header_Table can be constructed. The prelarge tree is then formed from the database, the Header_Table and the Pre_Header_Table. The results are shown in Figure 2.

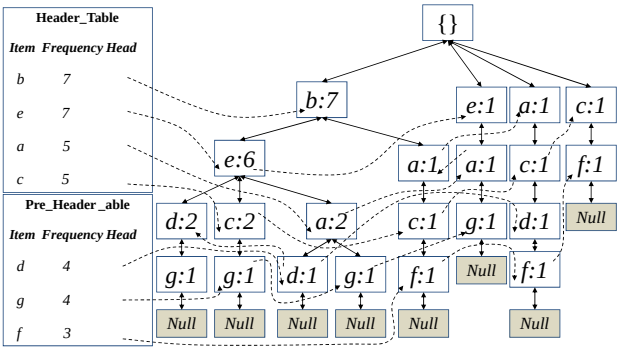

Figure 2: The Header_Table, Pre_Header_Table and the prelarge tree constructed

Assume the last three records (with TID 8 to 10) are deleted from the original database. The proposed prelarge-tree maintenance algorithm proceeds as follows. The variable *c* is initially set at 0.

STEP 1: The safety number *f* for deleted records is calculated as:

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

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

STEP 3: All the items *a* to *i* are divided into three parts, *{a}{b}{c}{e}*, *{d}{f}{g}*, and *{h}{i}* according to whether they are large (appearing in the Header_Table), pre-large (appearing in the Pre_Header_Table) or small in the original database.

STEP 4: The items in the deleted records which are large in the original database are first processed. In this example, items *a*, *b*, *c* and *e*(the first partition) satisfy the condition and are processed. Take item *a* as an example to illustrate the substeps. The count of item *a* in the Header_Table is 5, and its count in the deleted records is 2. The new count of item a is thus 5 - 2 (= 3). The new support ratio of item *a* is 3/(10-0-3), which lies between 0.3 and 0.5. Item *a* is removed from the Header_Table and put into the head of the Pre_Header_Table with its updated frequency value and into the set of Reduced_Items. The new count of item *c* is thus 5 - 3 (= 2). The new support ratio of

item *c* is 2/(10-0-3), which lower than 0.3. Item *c* will become small after database is updated. The item *c* is thus removed from the Header_Table and prelarge tree. The new count of item *b* is 7 – 1 (= 6). Item *b* is thus still a large item after database is updated. The frequency value of item *b* in the Header_Table is thus changed as 6, and item *b* is then put into the set of *Reduced_Items*. Item *e* is similarly processed. After STEP 4, the *Reduced_Items* = {*a, b, e*}.

STEP 5: The items in the deleted records which are pre-large in the original database are processed. They include items *d, f* and *g*. Take item *d, f* and *g* as an example to illustrate the substeps, respectively. The count of item *d* in the Pre_Header_Table is 4, and its count in the deleted records is 1. The new count of item *d* is thus 4 - 1 (= 3). The new support ratio of item *d* is 3/(10-0-3), which lies between 0.3 and 0.5. Item *d* is thus still a pre-large item after the database is updated. The frequency value of item *d* in the Pre_Header_Table is thus changed as 4, and item *d* is then put into the set of *Reduced_Items*. The count of item *f* in the Pre_Header_Table is 3, and its count in the deleted records is 3. The new count of item *f* is thus 3 - 3 (= 0). The new support ratio of item *f* is then 0/(10-0-3), which is smaller than 0.3. Item *f* will become small after database is updated. Item *f* is thus removed from the Pre_Header_Table and from the prelarge tree. The count of item g in the Pre_Header_Table is 4, and its count in the deleted records is 0. The new count of item *g* is thus 4 - 0 (= 4). The new support ratio of item *g* is then 4/(10-0-3), which larger than 0.5. Item *g* will become large items after database is updated. Item g is removed from the Pre_Header_Table and put in the end of Header_Table with its new frequency. The frequency value of item *g* in the Header_Table is thus changed as 4, and item *g* is then put into the set of *Reduced_Items.* After STEP 5, *Reduced_Items* = {*a, b, d, e, g*}.

STEP 6: The prelarge tree is updated according to the deleted records with items existing in the *Reduced_Items*. In this example, *Reduced_Items = {a, b, d, e, g}*. The corresponding branches for the deleted records with any items in the set of *Reduced_Items* are shown in Table 2.

Table 2: Three partitions of the items from the deleted records

TID	Items	Corresponding branches
8	*a, b, c, f*	*b, a*
9	*a, c, d, f*	*a, d*
10	*c, f, i*	ϕ

The first branch shares the same prefix (*b, a*) as the current prelarge tree. The count for item *b* and *a* are then substracted by 1 since they have to be deleted from the previous prelarge tree after database is updated. The same process is then executed for another branch. The final results after STEP 6 are shown in Figure 3.

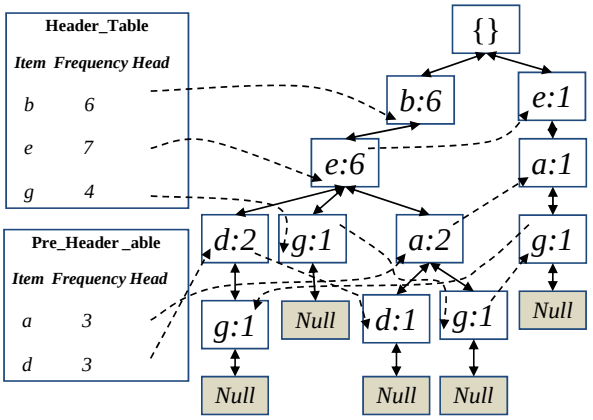

Figure 3: The Header_Table, the Pre_Header_Table and the prelarge tree after STEP 6

STEP 7: Since the item *h* and *i* is neither large nor pre-large in the original database (not appearing in the Header_Table and in the Pre_Header_Table), and small in the deleted records, it is put into the set of *Rescan_Items*, which is used when rescanning in STEP 7 is required. After STEP 7, *Rescan_Items = {h, i}*.

STEP 8: Since $t + c = 3 + 0 < f\ (= 4)$, rescanning the original database is unnecessary. Nothing is done in this step.

STEP 9: Since $t\ (= 3) + c\ (= 0) < f\ (= 4)$, set $c = t + c = 3 + 0 = 3$.

After STEP 9, the prelarge tree is updated. Note that the final value of c is 3 in this example and $f - c = 1$. This means that one more record can be added without rescanning the original database for Case 9. Based on the prelarge tree shown in Figure 5, the desired large itemsets can then be found by the FP-Growth mining approach as proposed in [3] on only the large items.

5 Experiments

Experiments were made to compare the performance of the batch FP-tree construction algorithm, the FUFP-tree deletion algorithm and the Prelarge-tree deletion algorithm for record deletion. 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* [7] were 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 transactions in the *BMS-POS* database were first used to construct an initial FP-tree. The minimum threshold was set at 1% to 5% for the three algorithms, with 1% increment each time. 2,000 transactions were then deleted from the database. For the deletion algorithm of prelarge tree, the upper minimum support threshold was set at 1% to 5% (1% increment each time) and the lower minimum support threshold was set at 0.2%, 1.2%, 2.2%, 3.2% and 4.2%, respectively. The execution times and the numbers of nodes obtained from the three algorithms were compared. Figure 4 shows the execution times of the three algorithms for different threshold values.

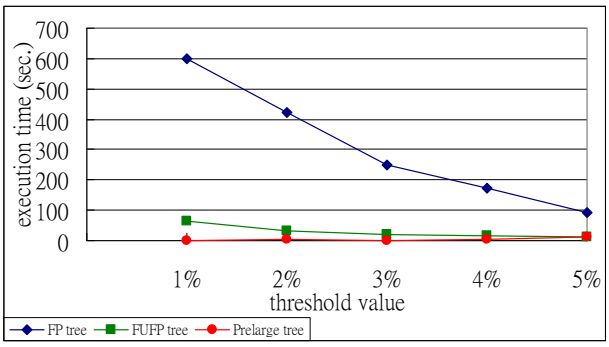

Fig. 4. The comparison of the execution times for different threshold values.

The comparison of the numbers of nodes for the three algorithms is given in Figure 5.

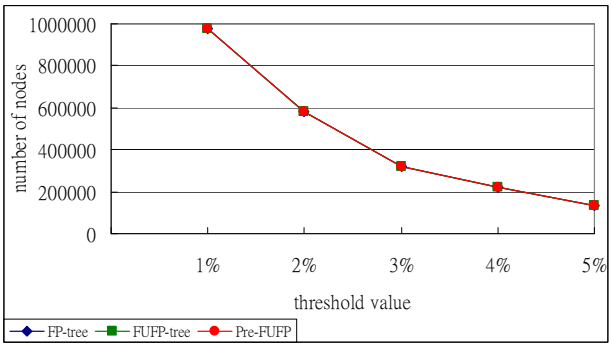

Fig. 5. The comparison of the number of nodes for different threshold values.

It can be seen that the three algorithms generated nearly the same sizes of trees. The effectiveness of the Prelarge tree deletion algorithm is thus acceptable.

6 Conclusions

In this paper, we have proposed the prelarg-tree maintenance algorithm for record deletion based on the concept of pre-large itemsets. The prelarge-tree structure is used to efficiently and effectively handle new transactions. Using two user-specified upper and lower support thresholds, the pre-large items act as a gap to avoid small items becoming large in the updated database when transactions are deleted.

Experimental results also show that the proposed prelarge-tree maintenance algorithm runs faster than the batch FP-tree and the FUFP-tree algorithm for handling deleted records and generates nearly the same number of frequent nodes as them. The proposed approach can thus achieve a good trade-off between execution time and tree complexity.

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, 1993.

[2] 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.

[3] 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.

[4] 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.

[5] T. P. Hong, C. W. Lin and Y. L. Wu, "*Incrementally fast updated frequent pattern trees*", Expert Systems with Applications, 2007 (accepted and to appear).

[6] H. Mannila, H. Toivonen and A. I. Verkamo, "*Efficient algorithm for discovering association rules*," The AAAI Workshop on Knowledge Discovery in Databases, pp. 181-192, 1994.

[7] 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.