Maintenance of Pre-large FUSP Trees in Dynamic Databases

Chun-Wei Lin1, Tzung-Pei Hong1,2
1Dept of Computer Science and Information Engineering
National University of Kaohsiung
Kaohsiung, Taiwan
2Department of Computer Science and Engineering
National Sun Yat-sen University
Kaohsiung, Taiwan
Email: jerrylin@ieee.org, tphong@nuk.edu.tw

Hong-Yu Lee1 and Shyue-Liang Wang3
1Department of Computer Science and Information Engineering
Kaohsiung, Taiwan
3Department of Information Management
National University of Kaohsiung
Kaohsiung, Taiwan
Email: staryculturesky@gmail.com,
slwang@nuk.edu.tw

Abstract—In the past, pre-large fast-updated sequential pattern trees (pre-large FUSP tree) were proposed for efficiently mining large sequences for record insertion and deletion, respectively. In this paper, we thus proposed a maintenance approach for efficiently maintaining pre-large FUSP trees and effectively deriving desired large sequences when data in databases are modified. Experimental results also show that the proposed algorithm has a better performance in execution time.

Keywords—pre-large FUSP tree; large sequences; maintenance; record modification; tree structure

I. INTRODUCTION

Data mining is used to extract the desired information or rules for efficiently making the decisions from large databases [1, 2]. Among the different derived knowledge, mining the sequential patterns (large sequences) from temporal database is a critical issue for modeling customer behavior in business.

Sequential patterns mining was firstly proposed by Agrawal et al. in 1995, and it takes trivial tasks to derive the desired information [3]. It finds customer purchase sequence to predict whether there is a high probability that when customers buy the products, they will buy the other products in later transactions. In the past, the algorithms for record insertion [4, 6, 10, 11] and record deletion [7] were respectively proposed to maintain the derived sequences in dynamic databases. In real-world application, transactions may be modified due to some incomplete information or manual errors [8]. Thus, it is an important task to effectively maintain the mined sequential patterns in the updated database. Conventional approach [3] is to re-mine the updated database to get the correct sequential patterns in maintenance, thus requiring considerable computational time when the database is massive in size.

In this paper, we propose a maintenance algorithm based on the pre-large FUSP tree [6] for efficiently handling modification of records. When records are modified from the database, the proposed algorithm is to maintain tree structure, partitioning the 1-itemsets in modified transactions into nine parts according to whether they are large, pre-large, or small in the original database and whether their count difference is positive, zero or negative based on pre-large concept [8]. Each part is thus processed on its own way to maintain and update the constructed pre-large FUSP tree. Experimental results then showed the performance of the proposed approach in execution time.

II. REVIEW OF RELATED WORKS

A. Sequential Patterns Mining

Sequential patterns [3] are mainly used to mine user actions over time, which is harder than mining association rules [1, 2] since the former must consider both itemsets and sequences. Although customer behavior models can be efficiently extracted using Agrawal et al.’s mining algorithm [3], the sequential patterns may become invalid or inappropriate when databases are updated. Many approaches have been proposed to improve maintenance performance. For the maintenance of sequential patterns, Lin and Lee proposed the FASTUP algorithm for inserted records [10]. Cheng et al. proposed the IncSpan (Incremental mining of sequential patterns) algorithm for efficiently mining sequential patterns from a tree structure [4]. Lin et al. also modified the FP-tree structure [5] and designed the fast updated sequential pattern tree (FUSP tree) to efficiently handle newly inserted customer sequences [11]. Hong et al. then designed a pre-large FUSP tree and respectively proposed the maintenance algorithms for record insertion [6] and record deletion [7] based on the pre-large concept.

B. Pre-large Concept for Record Modification

In addition to record insertion [6] and deletion [7], record modification in databases is also common. Considering an original customer sequences and the modified customer sequences in terms, the nine cases illustrated in Figure 1 may occur [8]. When large and pre-large sequences are kept, all the above cases except case 7 can be easily handled. It has been formally shown that a sequence in case 7 cannot possibly be large for the entire updated database as long as the number of modified customers (with all their records are modified) is smaller than a safety bound shown below [8]:

$$q \leq \lfloor (S_a - S_r)d \rfloor$$
where \( q \) is the number of modified customers, \( d \) is the number of customer sequences from the database, \( S_u \) is the upper threshold, and \( S_l \) is the lower threshold.

III. THE PRE-LARGE FUSP TREES

The designed pre-large FUSP-tree structure is shown in Figure 2. Two different relations in the tree are the sequence-extended sequence (SES), marked as \( s \), and the itemset-extended sequence (IES), marked as \( i \). The former represents sequences and the latter represents itemsets [4].

IV. THE PROPOSED MAINTENANCE ALGORITHM

The detail of the proposed maintenance algorithm is described below. The global variable, \( c \), is used to accumulate the number of modified customers since the last re-scan of the entire updated database.

The maintenance algorithm for record modification:

**INPUT:** An old database consisting of \( d \) records, its corresponding Header Table, Pre_Header Table, and the pre-large FUSP tree, a lower support threshold \( S_l \), an upper support threshold \( S_u \), and a set of \( r \) modified customer sequences transformed from modified records.

**OUTPUT:** An updated pre-large FUSP tree.

**STEP 1:** Calculate the safety number \( f \) for modified records according to the following formula [8]:

\[
f = [(S_u - S_l) \times d]^{1/2}.
\]

**STEP 2:** Find all 1-itemsets in the \( t \) modified records before and after modification. Denote them as a set of modified 1-itemsets, \( M \).

**STEP 3:** Find the count difference (including zero) of each 1-itemset in \( M \) for the modified records.

**STEP 4:** Divide the 1-itemsets in \( M \) into three parts according to whether they are large, pre-large or small in the original database.

**STEP 5:** For each 1-itemset \( I \) in \( M \) which is large in the original database (appearing in the Header Table), do the following substeps (for cases 1, 2 and 3):

Substep 5-1: Set the new count \( S^I_u(I) \) of \( I \) in the entire updated database:

\[
S^I_u(I) = S^I_u(I) + S^I_M(I),
\]

where \( S^I_u(I) \) is the count of \( I \) in the Header Table (from the original database) and \( S^I_M(I) \) is the count difference of \( I \) from record modification.

Substep 5-2: If \( S^I_u(I) \geq (S \times d) \geq S^I_M(I) \), update the count of \( I \) in the Header Table as \( S^I_u(I) \), and put \( I \) in both the set of Increased_Seqs and Decreased_Seqs, which will be further processed to update the pre-large FUSP tree in STEP 9; Otherwise, if \( (S \times d) \geq S^I_u(I) \geq (S \times d) \), remove \( I \) from the Header Table, put \( I \) at the front of the Pre_Header Table with its updated count \( S^I_u(I) \), and keep \( I \) in both the set of Increased_Seqs and Decreased_Seqs.

Otherwise, if \( I \) is small after the database is updated; remove \( I \) from the Header Table and connect each parent node of \( I \) directly to its corresponding child node in the pre-large FUSP tree.

**STEP 6:** For each 1-itemset in \( M \) which is pre-large in the original database (appearing in the Pre_Header Table), do the following substeps (for cases 4, 5 and 6):

Substep 6-1: Set the new count \( S^I_u(I) \) of \( I \) in the entire updated database as:

\[
S^I_u(I) = S^I_u(I) + S^I_M(I).
\]

Substep 6-2: If \( S^I_u(I) \geq (S_u \times d) \), 1-itemset \( I \) will become large after the database is updated; remove \( I \) from the Pre_Header Table, put \( I \) with its new count \( S^I_u(I) \) at the end of the Header Table, and put \( I \) in both the sets of Increased_Seqs and Decreased_Seqs.

Otherwise, if \( (S_u \times d) \geq S^I_u(I) \geq (S_u \times d) \), 1-itemset \( I \) is still pre-large after the database is updated; update \( I \) with its new count \( S^I_u(I) \) in the Pre_Header Table and put \( I \) in the sets of Increased_Seqs and Decreased_Seqs.
Otherwise, 1-itemset \( I \) is small after the database is updated; remove \( I \) from the Pre_Header_Table and connect each parent node of \( I \) directly to its corresponding child node in the pre-large FUSP tree.

**STEP 7:** For each 1-itemset \( I \) which is neither large nor pre-large in the original database but has positive count difference in \( M \) (for case 7), put \( I \) in the set of Rescan_Seqs, which is used when rescanning the database in **STEP 8** is necessary.

**STEP 8:** If \( (t+c) \leq f \) or the set of Rescan_Seqs is null, do nothing;
Otherwise, do the following substeps for each 1-itemset \( I \) in the set of Rescan_Seqs:

Substep 8-1: Rescan the original database to decide the original count \( S^O(I) \) of \( I \) (before modification).
Substep 8-2: Set the new count \( S^I(I) \) of \( I \) in the entire updated database as:
\[
S^I(I) = S^O(I) + S^{dl}(I).
\]
Substep 8-3: If \( S^I(I) \geq (S_c \times d) \), 1-itemset \( I \) will become large after the database is updated; put \( I \) at the end of the Header_Table and in both the sets of Increased_Seqs and Branch_Seqs; Otherwise, if \( (S_c \times d) \geq S^I(I) \geq (S_s \times d) \), 1-itemset \( I \) will become pre-large after the database is updated; put \( I \) at the end of the Pre_Header_Table and in both the sets of Increased_Seqs and Branch_Seqs; Otherwise, neglect \( I \).

**STEP 9:** For each updated record before modification (\( T' \)) and with a 1-itemset \( J \) existing in the Decreased_Seqs, find the corresponding branch of \( J \) in the pre-large FUSP tree for the record, and subtract 1 from the count of the \( J \) node in the branch; if the count of the \( J \) node becomes zero after subtraction, remove node \( J \) from its corresponding branch and connect the parent node of \( J \) directly to the child node of \( J \).

**STEP 10:** For the updated records after modification (\( T \)) with a 1-itemset \( J \) existing in Increased_Seqs, if \( J \) has not been at the corresponding branch of the pre-large FUSP tree, insert \( J \) at the end of the branch and set its count as 1; Otherwise, add 1 to the count of the \( J \) node.

**STEP 11:** If the set of Branch_Seqs is null, nothing is done in this step;
Otherwise, for each record in the unmodified records (\( D \)) and with an item \( J \) existing in Branch_Seqs, if \( J \) has not been at the corresponding branch of the pre-large FUSP tree, 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 12:** If \( (t+c) > f \), set \( c = 0 \); otherwise, set \( c = t + c \).

In **STEP 9**, a corresponding branch for a record is the branch generated from the large and pre-large 1-itemsets in the record and corresponding to the order of 1-itemsets appearing in the Header_Table and Pre_Header_Table. After **STEP 12**, the final updated pre-large FUSP tree is obtained. Based on the pre-large FUSP tree, the desired large sequences can then be found by the FP-growth-like mining approach [5] on the large nodes in Header_Table.

**V. AN EXAMPLE**

The dataset in Table 1 is used to construct the initially pre-large FUSP tree and shown in Figure 2. The lower support threshold \( S_l \) and the upper support threshold \( S_u \) are set at the 30% and 60%, respectively. Assume the last two records (with TIDs 9 to 10) in the original database are modified, as shown in Table 2. The proposed pre-large FUSP-tree algorithm proceeds as follows. The variable \( c \) is initially set at 0.

| **TABLE I.** A DATABASE FOR CONSTRUCTING THE PRE-LARGE FUSP TREE |
| --- | --- |
| **Customer ID** | **Sequence** |
| 1 | (a)(b) |
| 2 | (ac)(ei) |
| 3 | (ac)(df) |
| 4 | (b)(cd)(e) |
| 5 | (a)(b)(e)(f) |
| 6 | (ac)(b) |
| 7 | (a)(b)(d)(ef) |
| 8 | (ac)(e)(fg) |
| 9 | (b)(d)(g) |
| 10 | (de)(g)(h) |

| **TABLE II.** TWO RECORDS AFTER MODIFICATION |
| --- | --- |
| **Customer ID** | **Sequence** |
| 9 | (b)(d)(gh) |
| 10 | (ac)(i) |

The safety number \( f \) for modified records is firstly calculated as [8]:
\[
f = \left( (S_c - S_l) \right) d = \left[ (0.6 - 0.3) \right] \times 10 = 3.
\]

The 1-itemsets in the two records before and after modification are found as \{a\}, \{b\}, \{c\}, \{d\}, \{e\}, \{g\}, \{h\} and \{i\}, which are denoted as \( M \). The count difference of each 1-itemset in \( M \) is found. The results are \{a:1\}, \{b:0\}, \{c:1\}, \{d:-1\}, \{e:-1\}, \{g:-1\}, \{h:1\}, and \{i:1\}. All the 1-itemsets are then divided into three parts, \{a, b, e\}, \{c, 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. Each part is then processed on its own way according to the designed algorithm. Also, the pre-large FUSP tree is thus updated according to the maintenance approach in each
processed step. After all steps are processed, the final pre-large FUSP tree is thus shown in Figure 3.

![Diagram of the pre-large FUSP tree]

Figure 3. The final pre-large FUSP tree.

Also, \((t=2) + (c = 0) < (f = 3)\), set \(c = t + c = 2 + 0 = 2\). Note that the final value of \(c\) is 2 in this example and \((f - c) = 1\). This means that one more record can be modified without rescanning the original database for case 7. Based on the pre-large tree shown in Figure 3, the desired large sequences can then be found by the FP-growth mining approach as proposed in [5].

VI. EXPERIMENTAL RESULTS

Experiments were made to compare the performance of the AprioriAll algorithm [3], FUSP-tree algorithm for record modification [12], and the proposed pre-large FUSP-tree algorithm. The IBM data generator [9] was used to generate the sequence data S4I2N1KD25K in the experiments. The value of the upper and lower support thresholds were respectively set at 2.3% and 2.7%. The execution times obtained from the three algorithms were compared and the results shown in Figure 4. The numbers of 150 transactions are then sequentially modified from the original database each time.

![Graph of execution times for three algorithms]

Figure 4. The execution times for three compared algorithms.

In Figure 4, it easily observed that the execution of the proposed pre-large FUSP-tree maintenance algorithm has better execution time than that by the batch AprioriAll algorithm and the FUSP-tree maintenance algorithm for record modification. The effectiveness of the proposed approach for record modification is thus acceptable.

VII. CONCLUSION

In this paper, we have proposed the pre-large FUSP tree maintenance algorithm for record modification. Based on pre-large concept for record modification, it helps reduce the rescanning of the original customer sequences if the modified customer sequences do not exceed the safety bound. From the experimental results, the proposed maintenance algorithm has a better performance than the batch AprioriAll algorithm and the FUSP-tree maintenance algorithm for record modification.

REFERENCES