Efficient Modification of Fast Updated FP-trees Based on Pre-large Concepts

Chun-Wei Lin¹, Tzung-Pei Hong²,³* and Wen-Hsiang Lu¹

¹Department of Computer Science and Information Engineering
National Cheng Kung University
Tainan, 701, Taiwan, R.O.C.
{p7895122; whlu}@mail.ncku.edu.tw

²Department of Computer Science and Information Engineering
National University of Kaohsiung
Kaohsiung, 811 Taiwan, R.O.C.

³Department of Computer Science and Engineering
National Sun Yat-sen University
Kaohsiung, 804, Taiwan, R.O.C.
tphong@nuk.edu.tw

Abstract

The frequent pattern tree (FP-tree) is an efficient data structure for association-rule mining without generation of candidate itemsets. It is used to compress a database into a tree structure which stores only large items. It, however, needs to process all transactions in a batch way. In this paper, we modify the FUFP-tree maintenance based on the concept of pre-large itemsets for efficiently handling record modification. Pre-large itemsets are defined by a lower support threshold and an upper support threshold. It can help reduce the rescan of the original database. The proposed approach can achieve a good execution time for tree maintenance especially when each time a small number of records are modified. Experimental results show that the proposed Pre-FUFP modification algorithm has a good performance for handling updated records and generate nearly the same tree structure as the original FP tree algorithm.

Keyword: data mining, FP-tree, FUFP-tree, Pre-FUFP algorithm, pre-large itemsets, record modification, maintenance.

*corresponding author
1. Introduction

Years of effort in data mining have produced a variety of efficient techniques. Depending on the classes of knowledge derived, the mining approaches may be classified as finding association rules, classification rules, clustering rules and sequential patterns [4][13], among others. Among them, finding association rules in transaction databases is most commonly seen in data mining [1][2][3][6][7][14][15][17][18].

In the past, 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. Han et al. proposed the Frequent-Pattern-tree (FP-tree) structure for efficiently mining association rules without generation of candidate itemsets [8]. The FP-tree was used to compress a database into a tree structure which stored only large items. It was condensed and complete for finding all the frequent patterns. A recursive mining procedure called FP-Growth was executed to derive frequent patterns from the FP-tree. They showed the approach could have a better performance than the Apriori approach.

Hong et al. thus proposed an algorithm based on pre-large itemsets to handle the
inserted transactions in incremental mining, which can further reduce the number of rescanning databases [9]. It used a lower support threshold and an upper threshold to reduce the need for rescanning original databases and to save maintenance cost until a number of new transactions have been inserted or deleted. Since rescanning the database spent much computation time, the maintenance cost could thus be reduced in the algorithm.

Hong et al. also proposed the Fast Updated FP-tree (FUFP-tree) structure to efficiently handle the newly inserted transactions in incremental mining [10]. 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 propose a fast updated FP-tree modification algorithm based on the pre-large concept to efficiently handle modified records. When records are modified from the database, the proposed algorithm will process them to maintain the FUFP-tree and the Header_Table. The count difference of each item in modified records is first calculated. The proposed Pre-FUFP modification algorithm then partitions items into nine cases according to whether they are large, pre-large or small
in the original database and whether their item differences are positive, zero or negative. Each case is then processed in its own way. The Header_Table and the FUFP-tree are then correspondingly updated whenever necessary. The proposed approach can effectively handle the case in which items are small in the original database but have positive count differences due to record modification based on pre-large concepts. Experimental results also show that the proposed FUFP-tree modification algorithm based on the pre-large concept can achieve a good performance when records are modified.

2. Review of related works

In this section, some related researches are briefly reviewed. They are the FUFP-tree structure (which is based on the FP-tree structure) and the concept of pre-large itemsets.

2.1 The FUFP-tree structure

An FUFP tree must be built in advance from the original database before new transactions come. The counts of the sorted frequent items are also kept in the
Header_Table which the same as the FP-tree structure [8]. When new transactions are added, the FUFP-tree maintenance algorithm will process them to maintain the FUFP tree. The Header_Table and the FUFP-tree are correspondingly updated whenever necessary.

When an originally large item becomes small, it is directly removed from the FUFP tree and its parent and child nodes are then linked together. On the contrary, when an originally small item becomes large, it is added to the end of the Header_Table and then inserted into the leaf nodes of the FUFP tree. It is reasonable to insert the item at the end of the Header_Table because when an originally small item becomes large due to the new transactions, its updated support is usually only a little larger than the minimum support. The entire FUFP tree can then be re-constructed in a batch way when a sufficiently large number of transactions have been inserted.

Several other algorithms based on the FP-tree structure have been proposed [5][12][16][19]. Some related researches are still in progress.

2.2 The concept of pre-large itemsets
Hong et al. proposed an algorithm based on pre-large itemsets to handle the inserted transactions in incremental mining, which can further reduce the number of rescanning databases [9]. Two support thresholds, a lower support threshold and an upper support threshold, are used to realize this concept.

Considering an original database and some records to be modified 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 modified records

Cases 1, 2, 5, 6, 8 and 9 above will not affect the final large items. Case 3 may remove some existing association rules, and cases 4 and 7 may add some new
association rules. If we retain all large and pre-large itemsets with their counts after each pass, then cases 3 and 4 can be handled easily. Also, in the maintenance phase, the ratio of modified 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 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 [9]:

\[
f = \lfloor (S_u - S_l) d \rfloor,
\]

where \( f \) is the safety number of modified records, \( S_u \) is the upper threshold, \( S_l \) is the lower threshold, and \( d \) is the number of original transactions.

3. The Proposed Pre-FUFP Modification Algorithm for Handling Modified Records

An FUFP tree is built in advance from the initial original database. 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, which called large items. Next, the large items are sorted in descending frequency. At last, the database is scanned again to construct an FUFP tree according to the sorted order of large items. After all transactions are processed, an FUFP tree is completely
When records are modified from the database, the FUFP tree and the Header_Table need to be modified as well. The proposed algorithm first calculates the count difference by comparing the counts of each updated item before and after record modification. The proposed maintenance algorithm then partitions the items into nine cases according to whether they are large, pre-large or small in the original database and whether their item differences are positive, zero or negative. Each part is then processed in its own way. The Header_Table and the FUFP-tree are correspondingly updated whenever necessary. The notation used in the proposed Pre-FUFP modification algorithm for handling records modified is first described below.

### 3.1 Notation

- \( D \): the original database;
- \( T \): the set of modified records (after modification);
- \( T' \): the set of records to be modified (before modification);
- \( D' \): the set of unchanged records, i.e., \( D-T' \);
- \( U \): the entire updated database;
- \( M \): the set of items appearing in the updated records before and after modification;
- \( d \): the number of records in \( D \);
- \( t \): the number of records in \( T \);
- \( d' \): the number of records in \( D' \);
- \( S_l \): the lower support threshold for pre-large itemsets;
Su: the upper support threshold for large itemsets, $S_u > S_l$;
$I$: an item;
$S^D(I)$: the number of occurrences of $I$ in $D$;
$S^M(I)$: the count difference of $I$ from the updated records, $I \in M$;
$S^U(I)$: the number of occurrences of $I$ in $U$.

$\text{Decreased}_\text{Items}$: the set of items with their counts decreased when an FUFP-tree is modified;
$\text{Increased}_\text{Items}$: the set of items with their counts increased when an FUFP-tree is modified;
$\text{Rescanned}_\text{Items}$: the set of items to be rescanned to get their supports in the original database;
$\text{Branch}_\text{Items}$: the set of items to be rescanned to build new branches in an FUFP-tree;

3.2 The Proposed Modification Algorithm

The details of the proposed algorithm are described below. A variable $c$ is used to record the number of modified records since the last re-scan of the original database.

The Pre-FUFP modification algorithm:

INPUT: An old database consisting of $d$ records, 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 $t$ modified records.

OUTPUT: A new FUFP tree after record modification by using the Pre-FUFP
modification algorithm.

STEP 1: Calculate the safety number $f$ of modified records according to the following formula [9]:

$$f = \lfloor (S_u - S_i) / d \rfloor.$$

STEP 2: Find all the items in the $t$ records before and after modification. Denote them as a set of modified items, $M$.

STEP 3: Find the count difference (including zero) of each item in $M$ for the modified records.

STEP 4: Divide the items in $M$ into three parts according to whether they are large, pre-large or small in the original database.

STEP 5: For each item $I$ 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 $S'(I)$ of $I$ in the entire updated database as:

$$S'(I) = S^O(I) + S^M(I),$$

where $S^O(I)$ is the count of $I$ in the Header_Table (from the original database) and $S^M(I)$ is the count difference of $I$ from record modification.

Substep 5-2: If $S'(I)/d \geq S_u$, update the count of $I$ in the Header_Table as $S'(I)$, and put $I$ in the set of Increased_Items and
**Decreased_Items**, which will be further processed to update the FUFP tree in STEP 10;

Otherwise, if \( S_u > \frac{S(I)}{d} \geq S_l \), remove \( I \) from the Header_Table, connect each parent node of \( I \) directly to its corresponding child node in the 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 corresponding child node in the FUFP tree.

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

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

\[
S^U(I) = S^D(I) + S^M(I).
\]

**Substep 6-2**: If \( \frac{S^U(I)}{d} \geq S_u \), item \( I \) will be large after the database is updated; put \( I \) in both the sets of **Increased_Items** and **Branch_Items**, which will be further processed to update the FUFP tree in **STEP 9**;

Otherwise, if \( S_u > \frac{S^U(I)}{d} \geq S_l \), 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 7: For each item $I$ which is neither large nor pre-large in the original database but has positive item difference in $M$ (for case 7), put $I$ in the set of $Rescanned\_Items$, which is used when rescanning the database in STEP 8 is necessary.

STEP 8: If $t+c \leq f$ or the set of $Rescanned\_Items$ is null, then do nothing;

Otherwise, do the following substeps for each item $I$ in the set of $Rescanned\_Items$:

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:

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

Substep 8-3: If $S^U(I)/d \geq S_n$, item $I$ will become large after the database is updated; put $I$ in both the sets of $Insert\_Items$ and $Branch\_Items$;

Otherwise, if $S_n > S^U(I)/d \geq S_0$, set $S^D(I) = S^U(I)$ and keep $I$ with $S^D(I)$ in the pre-large table;

Otherwise, neglect $I$. 

STEP 9: For each updated record before modification \((T')\) and with an item \(J\) existing in the \textit{Decreased\_Items}, find the corresponding branch of \(J\) in the FUFP 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: Insert the items in the \textit{Branch\_Items} to the end of the Header\_Table according to the descending order of their updated counts.

STEP 11: For each record in the unmodified records \((D')\) with an item \(J\) existing in \textit{Branch\_Items}, if \(J\) has not been at the corresponding branch of the FUFP 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: For the updated records after modification \((T)\) with an item \(J\) existing in \textit{Increased\_Items}, if \(J\) has not been at the corresponding branch of the FUFP 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 13: If \(t+c > f\), then set \(c = 0\); otherwise, set \(c = t+c\).

In STEP 10, a \textit{corresponding branch} is the branch generated from the large items
in a record and corresponding to the order of items appearing in the Header Table. After STEP 13, the final updated FUFP tree by using the Pre-FUFP modification algorithm to process modified records is obtained. The modified records 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 [8].

4. An Example

In this session, an example is given to illustrate the proposed Pre-FUFP modification algorithm for maintaining an FUFP tree when records are modified. 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

<table>
<thead>
<tr>
<th>TID</th>
<th>Items</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>a, b, c, d, g, h</td>
</tr>
<tr>
<td>2</td>
<td>a, b, c, e, g, h</td>
</tr>
<tr>
<td>3</td>
<td>a, b, f, g, h</td>
</tr>
<tr>
<td>4</td>
<td>a, b, c, e, g, h</td>
</tr>
<tr>
<td>5</td>
<td>a, b, c, e, f, g</td>
</tr>
<tr>
<td>6</td>
<td>a, d, e, g, h, i</td>
</tr>
<tr>
<td>7</td>
<td>b, c, d, g, h, i</td>
</tr>
<tr>
<td>8</td>
<td>a, e, h, i</td>
</tr>
</tbody>
</table>
Assume the lower support threshold $S_l$ is set at 40% and the upper one $S_u$ at 70%.

For the given database, the large 1-itemsets are $a$, $c$, $g$ 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 2. Besides, the set of pre-large items for the given database is shown in Table 2.

![Figure 2: The Header_Table and the FUFP tree constructed](image)

**Table 2: The pre-large items for the original database**

<table>
<thead>
<tr>
<th>Items</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>$b$</td>
<td>6</td>
</tr>
<tr>
<td>$e$</td>
<td>5</td>
</tr>
<tr>
<td>$f$</td>
<td>4</td>
</tr>
</tbody>
</table>

Assume the last three records (with TIDs 8 to 10) in the original database are modified as shown in Table 3. The proposed Pre-FUFP modification algorithm
proceeds as follows. The variable $c$ is initially set at 0.

<table>
<thead>
<tr>
<th>Transaction No.</th>
<th>Items</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>$a, b, c, d, i$</td>
</tr>
<tr>
<td>9</td>
<td>$a, b, d, e, g, i$</td>
</tr>
<tr>
<td>10</td>
<td>$d, e, g, h, i$</td>
</tr>
</tbody>
</table>

Table 3: The three records after modification

STEP 1: The safety number $f$ for modified records is calculated as:

$$f = \lfloor (S_a - S_i) d \rfloor = \lfloor (0.7 - 0.4)10 \rfloor = 3. $$

STEP 2: The items in the three records before and after modification are found as $\{a, b, c, d, e, f, g, h, i\}$, which are denoted $M$.

STEP 3: The count difference of each item in $M$ is calculated. The results are shown in Table 4.

<table>
<thead>
<tr>
<th>Item</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a$</td>
<td>0</td>
</tr>
<tr>
<td>$b$</td>
<td>2</td>
</tr>
<tr>
<td>$c$</td>
<td>-1</td>
</tr>
<tr>
<td>$d$</td>
<td>3</td>
</tr>
<tr>
<td>$e$</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 4: The count difference of each item in $M$
STEP 4: All the items $a$ to $i$ in Table 4 are divided into three parts, $\{a\} \{c\} \{g\} \{h\}$, $\{b\} \{e\} \{f\}$ and $\{d\} \{i\}$ 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 5.

<table>
<thead>
<tr>
<th></th>
<th>Count</th>
<th></th>
<th>Count</th>
<th></th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a$</td>
<td>0</td>
<td>$b$</td>
<td>2</td>
<td>$d$</td>
<td>3</td>
</tr>
<tr>
<td>$c$</td>
<td>-1</td>
<td>$e$</td>
<td>1</td>
<td>$i$</td>
<td>2</td>
</tr>
<tr>
<td>$g$</td>
<td>1</td>
<td>$f$</td>
<td>-2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$h$</td>
<td>-1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 5: The three partitions of items in $M$

STEP 5: The items in $M$ which are large in the original database are first processed. In this example, items $a$, $c$, $g$ and $h$ (the first partition) satisfy the condition and are processed. Take the item $g$ as an example to illustrate the substeps. The count of item $g$ in the Header_Table is 8, and its count difference for item $g$ is 1. The new count of item $g$ is thus $8+1 (=9)$. Since the support ratio ($9/10$) of item $g$ is larger than
the upper support $S_u(0.7)$, item $g$ is thus still a large item after the database is updated. The frequency value of item $g$ in the Header_Table is thus changed as 9 and then put into both the sets of $Increased\_Items$ and $Decreased\_Items$. Similarly, items $a$ and $h$ satisfy the condition and are put into both the sets of $Increased\_Items$ and $Decreased\_Items$. On the contrary, the new count of item $c$ is $7 + (-1) = 6$. Its support ratio (6/10) is smaller than 0.7 but larger than 0.4. Item $c$ thus becomes pre-large after the database is updated. Item $c$ is directly removed from the Header_Table and its corresponding branches in the FUFP tree, and put in the pre-large table with its updated count as 6.

After STEP 5, the set of $Increased\_Items = \{a, g, h\}$ and $Decreased\_Items = \{a, g, h\}$. The updated FUFP tree after this step is shown in Figure 3.

![Figure 3: The Header_Table and the FUFP tree after STEP 5](image)

STEP 6: The items in $M$ which are in the pre-large table are processed. In this
example, items $b$, $e$ and $f$ satisfy the condition and will be processed. Since the count of item $b$ in the pre-large table is 6 and its count difference is 2, the new count of item $b$ is thus $6+2 \,(=8)$. Item $b$ will become large after the database is updated since its support ratio $(8/10)$ is larger than 0.7. Item $b$ is also put in both the sets of $Increased_Items$ and $Branch_Items$. Similarly, the new count of items $e$ and $f$ are updated as $5+1 \,(=6)$ and $4 + (-2) \,(=2)$, respectively. Since the support ratio $(6/10)$ of item $e$ is smaller than 0.7 but larger than 0.4, item $e$ is still a pre-large item after the database is updated. Item $f$ will become a small item since its support ratio $(2/10)$ is smaller than 0.4. The new count of item $e$ is thus changed as 6 in the pre-large table and item $f$ is removed from the pre-large table. After STEP 6, the set of $Increased_Items = \{a, g, h, b\}$ and the set of $Branch_Items = \{b\}$.

STEP 7: Since the items $d$ and $i$ are neither large nor pre-large in the original database, they are thus put into the set of $Rescanned_Items$, which is used when rescanning in STEP 8 is required. After STEP 7, $Rescanned_Items = \{d, i\}$.

STEP 8: Since $t + c = 3 + 0 \leq f \,(=3)$, rescanning the original database is unnecessary. Nothing has to be done in this step.
STEP 9: The FUFP tree is updated according to the records before modification $(T)$ with items existing in the set of $Decreased\_Items$. In this example, 

$Decreased\_Items = \{a, g, h\}$. The corresponding branches for the records before modification are shown in Table 6. The final results are shown in Figure 4.

<table>
<thead>
<tr>
<th>Transaction No.</th>
<th>Items</th>
<th>Corresponding branch</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>$a, e, h, i$</td>
<td>$a, h$</td>
</tr>
<tr>
<td>9</td>
<td>$a, c, f, g$</td>
<td>$a, g$</td>
</tr>
<tr>
<td>10</td>
<td>$c, f, h$</td>
<td>$h$</td>
</tr>
</tbody>
</table>

Table 6: The corresponding branches for the records before modification

Figure 4: The final FUFP tree after STEP 9

STEP 10: The items in the set of $Branch\_Items$ are sorted in the 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 $b$ and no sorting is needed. Item $b$ is thus inserted into the end of Header_Table. The Header_Table after this step is
shown in Figure 5.

STEP 11: The FUFP tree is updated according to the items in Branch_Items from the unmodified records. In this example, Branch_Items = \{b\} and the corresponding branches for item b in the unmodified records are shown in Table 7. After STEP 11, the final results are shown in Figure 6.

Table 7: The corresponding branches for the unmodified records in Branch_Items

<table>
<thead>
<tr>
<th>Transaction No.</th>
<th>Items</th>
<th>Corresponding branch</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>a, b, c, d, g, h</td>
<td>a, g, h, b</td>
</tr>
<tr>
<td>2</td>
<td>a, b, c, e, g, h</td>
<td>a, g, h, b</td>
</tr>
<tr>
<td>3</td>
<td>a, b, f, g, h</td>
<td>a, g, h, b</td>
</tr>
<tr>
<td>4</td>
<td>a, b, c, e, g, h</td>
<td>a, g, h, b</td>
</tr>
<tr>
<td>5</td>
<td>a, b, c, e, f, g</td>
<td>a, g, h, b</td>
</tr>
<tr>
<td>7</td>
<td>b, c, d, g, h, i</td>
<td>g, h, b</td>
</tr>
</tbody>
</table>
STEP 12: The FUFP tree is updated according to the items in the set of $Increase\_Item$ from the updated records (after modification). In this example, $Increased\_Items = \{a, g, h, b\}$. The corresponding branches with items in $Increased\_Items$ for the modified records are shown in Table 8. The final results are shown in Figure 7.

**Table 8: The corresponding branches with items in $Increased\_Items$ for the modified records**

<table>
<thead>
<tr>
<th>Transaction No.</th>
<th>Items</th>
<th>Corresponding branch</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>$a, b, c, d, i$</td>
<td>$a, b$</td>
</tr>
<tr>
<td>9</td>
<td>$a, b, d, e, g, i$</td>
<td>$a, g, b$</td>
</tr>
<tr>
<td>10</td>
<td>$d, e, g, h, i$</td>
<td>$g, h$</td>
</tr>
</tbody>
</table>
STEP 13: Since $t = 3 + c(0) \leq f(=3)$, set $c = t + c = 3 + 0 = 3$.

Note that the final value of $c$ is 3 in this example and $f - c = 0$. It means that no more record can be modified without rescanning the original database for Case 7.

Based on the FUFP tree shown in Figure 7, the desired large itemsets can then be found by the FP-Growth mining approach as proposed in [8].

5. **Experimental Results**

Experiments were made to compare the performance of the batch FP-tree construction algorithm, the FUFP-tree modification algorithm [11] and the Pre-FUFP modification algorithm for handling modified records. When records were modified,
the batch FP-tree construction algorithm constructed a new FP-tree from the updated
database. The process was executed whenever records were modified. The
FUFP-tree modification algorithm and the Pre-FUFP modification algorithm were
executed for record modification in the way mentioned in Sections 2.1 and 3.

The experiments were performed in the Ruby 1.8 programming language on an
Intel x86 PC with a 3.0G Hz processor and 512 MB main memory, running the
Microsoft Windows XP operating system. A real dataset called BMS-POS [20] 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 480,000 transactions in the BMS-POS database were used to construct
an initial FP-tree. For the Pre-FUFP modification algorithm, the upper minimum
support threshold was set at 2% to 6% (1% increment each time) and the lower
minimum support threshold was set at 1% to 3% (0.5% increment each time). The
execution times and the numbers of nodes obtained from the three algorithms were compared. Figure 8 shows the execution times of the three algorithms for different threshold values.

![Figure 8: The comparison of the execution times for the three modification algorithms](image)

It can be observed from Figure 8 that the proposed Pre-FUFP modification algorithm ran faster than the other two. Note that the FUFP-tree modification algorithm and the Pre-FUFP modification algorithm may generate a less concise tree than the FP-tree construction algorithm since the latter completely followed the sorted frequent items to build the tree. As mentioned above, when an originally small item becomes large due to modified records, its updated support is usually only a little larger than the minimum support. It is thus reasonable to put a new large item at the
end of the Header_Table. The difference between the FP-tree and the FUFP-tree structures will thus not be significant. For showing this effect, the comparison of the numbers of nodes for the three algorithms is given in Figure 9. It can be seen that the three algorithms generated nearly the same sizes of trees. The effectiveness of the Pre-FUFP modification algorithm is thus acceptable.

![Figure 9: The comparison of the numbers of nodes for the three modification algorithms](image)

Experiments were then made to show the execution times and the numbers of nodes of the three algorithms for different numbers of modified records. The minimum support thresholds were set at 2%, 4% and 6%, respectively, for the three algorithms and the lower support thresholds were set at 1%, 2% and 3% for the Pre-FUFP modification algorithm. The transactions from the BMS-POS database
were used to construct an initial FP-tree. Each time, 3,000 transactions were randomly chosen from the last updated database for modification. The next 3,000 transactions outside the 480,000 ones were used as the new contents in the modified records. The execution times for different threshold values are shown in Figures 10 to 12, respectively.

![Figure 10: The comparison of the execution times for modified records at the 2% threshold](attachment:image.png)
As mentioned before, when the modified records reach the safety number, the original database will be processed again. In the experiments, when the lower threshold value was set at 1% and the upper threshold value was set at 2%, the safety
number was calculated as $f = 480,000 \times (0.02-0.01) = 4,800$. Thus, the processing time for 6,000 and 120,000 modified records was more than that for the others in Figure 10. When the lower threshold value was set at 2% and the upper threshold value was set at 4%, the safety number was calculated as $f = 480,000 \times (0.04-0.02) = 9,600$. The processing time for 12,000 modified records was thus more than that for the others in Figure 11. At last, when the lower threshold value was set at 3% and the upper threshold value was set at 6%, the safety number was calculated as $f = 480,000 \times (0.06-0.03) = 14,400$. The processing time for 15,000 modified records was thus more than that for the others in Figure 12. Besides, it can be seen from these figures that the Pre-FUFP modification algorithm ran faster than the other two.

Next, Figures 13 to 15 show the numbers of nodes generated by the three algorithms with different threshold values.
threshold value: 2%

Figure 13: The comparison of the numbers of nodes for modified records at the 2% threshold

threshold value: 4%

Figure 14: The comparison of the numbers of nodes for modified records at the 4% threshold
Figure 15: The comparison of the numbers of nodes for modified records at the 6% threshold

It can also be seen from these figures that the Pre-FUFP modification algorithm had nearly the same node numbers as the other two.

6. Conclusion

In this paper, we have proposed the Pre-FUFP modification algorithm based on the pre-large concept to handle the effects of modified records on the FUFP-tree structure. Using the upper and the lower support thresholds, the pre-large itemsets act as a gap to avoid small itemsets becoming large in the updated database when transactions are modified. The proposed Pre-FUFP modification algorithm processes modified records to maintain the FUFP tree. It first partitions items in modified
records 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 modification algorithm runs faster than the batch FP-tree and the FUFP-tree modification algorithm for handling modified records and generates nearly the same tree structure as they do. The proposed approach can thus achieve a good trade-off between execution time and tree complexity.

References


