An incremental algorithm for frequent pattern mining based on bit-sequence

Wuzhou Dong, Juan Yi, Haitao He, Jiadong Ren
College of Information Science and Engineering, Yanshan University, P.R.China, 066004
dongwz@hebeea.edu.cn, fysdh@163.com, haitao@ysu.edu.cn, jdren@ysu.edu.cn

Abstract

In real-world applications, the target data is changed with time in association rules mining, and then existed association rules will also be changed, so the incremental mining algorithm should be developed. In this paper, we present IFPM-BS, a new incremental approach for mining frequent pattern. In this algorithm, we adopt the bit-sequence to compress the database to save the memory space, and the concept of the pre-large itemset is cited, the original database is not rescanned until a number of new transactions have been inserted; And then we define the concept of depth-branch and FLUPF-tree(Fast location Updated Frequent Pattern Tree) structure, the depth-branch is contained in the Header-Table of the FLUPF-tree to locate the nodes of the tree. If a pre-large item of the original database is changed into the large item, we can locate the nodes of the tree fast according to the matrix structure and the Header-Table information, the projected bit-sequences are contained in the matrix, and we don’t need to determine which transactions contain the item in the original database; Finally we can get the mining result by FP-growth algorithm. Experimental results also show that the proposed IFPM-BS algorithm can reduce the cost of time and improve mining efficiency.

Keywords: Incremental mining, Bit-sequence. Pre-large itemset

1. Introduction

Nowadays, association rules is one of the most important aspects in data mining\(^1\), on the one hand, we have a large number of data, which contain much information and knowledge of great potential value; on the other hand, the information update rate also reached very high speed\(^2\). But in the real world, the database is generally changed over time, so these static methods are no longer effective. FUP algorithm\(^3\) adopted the hash techniques, it calculated the support of large itemsets of newly inserted transactions, and compared them with previous large itemsets from original database. IPPFIM algorithm\(^4\) adopted the distortion technique, and computed the support of all combinations of the itemset through the basic formula from set theory. Though these algorithms used the information got from a previous mining, and the performance was indeed improved in incremental database, candidate itemsets were generated and original database needed to be rescanned, which would increase the cost of time and space. So the pre-large itemset\(^5\) was proposed, it defined a lower support threshold and an upper support threshold, the pre-large item and large item were stored in memory by two threshold, the algorithm didn’t need to rescan the original database until a number of new transactions had been inserted, execution time was obviously reduced, but a lot of candidate itemsets were still produced. The fast updated frequent pattern trees\(^6\) could handle newly inserted transactions efficiently based on FUP concept, FUFP-tree was similar to FP-tree except that the links between parent nodes and their child nodes were bi-directional. Besides, the counts of sorted frequent items were kept in Header_Table of FP-tree. Mining High Utility Patterns in Incremental Databases\(^7\) didn’t maintain count information, it used pattern growth mining approach to construct IIUUT tree, and efficiently capture the incremental data without any restructuring operation. The performance of these algorithms was improved without generation of candidate itemsets, but the original database still needed to be rescanned. CP-tree\(^8\) was improved based on FP-tree, it didn’t need to know frequent items in advance, it was constructed by lexicographical order if a new transaction was inserted, so the database only needed to be scanned once, but in restructuring phase, all branches needed to be restructured. DUA\(^9\) introduced a function to judge whether the itemset was frequent in updated database, and adopted the upper and lower support bounds, it could effectively reduce the number of the itemsets that needed to be scanned,
but it still needed to scan original database. Pre-FUFP modified FUFP-tree based on the concept of pre-large itemset, it didn’t need to scan the original database in a certain extent, but if a pre-large item was changed into a large item, the algorithm needed to determine which transactions contained the item in the original database when the item was inserted into the FUFP-tree, which could cost much time.

In order to improve the efficiency of incremental mining frequent pattern without rescanning the original database and generating the candidate itemsets, this paper uses the vertical data format based on bit-sequence to compress the database, which can reduce the memory space. And we define the depth-branch and FLUFP-tree structure, the depth-branch is contained in the Header-Table of the FLUFP-tree to locate the nodes of the tree, if a pre-large item is changed into the large item, we can locate the nodes of the tree fast according to the matrix structure and the Header-Table information, the projected bit-sequences are contained in the matrix, and we don’t need to determine which transactions contain the item in the original database, and we can get the mining results by FP-growth algorithm.

The rest of the paper is organized as following: Section 2 introduces some related definitions; Section 3 is the IFPM-BS algorithm and the detailed analysis of an example; Section 4 is the experimental results and performance analysis of the algorithm; Section 5 concludes this paper.

2. Problem definitions

Let $I=\{X_1, X_2, \ldots, X_m\}$ be a set of items $X$, $D=\{T_1, T_2, \ldots, T_n\}$ is a transaction database, each transaction $T$ is the set of items, $T \subseteq I$. Two support thresholds are cited: a lower support threshold and an upper support threshold. The support ratio of an itemset must be larger than the upper support threshold to be considered large. 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 a small itemset. 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.

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

Cases 1, 5, 6, 8 and 9 will not affect final frequent patterns; Cases 2 and 3 may remove existing frequent patterns, cases 4 and 7 may add new frequent patterns. Because we retain all large and pre-large itemsets with their counts, the cases 2, 3 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. Let $f$ be the safety number of new transactions, it has been formally shown that an itemset in case 7 can not possibly be large for the entire updated database as long as the number of transactions is smaller than $f$:

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

Where $S_u$ is the upper threshold, $S_l$ is the lower threshold, $d$ is the number of original transactions. In Case 4, if an item is frequent in the updated database, the original database need to be rescanned to search which transactions contain the item to get all frequent itemsets contained the item.

Let bit-sequence $\text{Bit}(X)$ represent the database $D$, if an item $X$ is in the $i$th transaction, the $i$th bit of $\text{Bit}(X)$ is set to be 1; otherwise, it is set to be 0. It is easy to know that $X$’s support is the number of ones in $\text{Bit}(X)$. For example, in Figure 2(a), an original database includes 9 different items and the total number of transactions is 10, thus the bit-sequence has ten bits. Because item $b$ appears in the transactions of 01,02,03,04,05,07,08,09,10, then $\text{Bit}(b)=1111101111$. $b$’s support is 9, by analogy, if the upper support threshold is set at 50%, the lower one is 30%, the bit-sequences of large items and the pre-large items are shown as Figure 2(b) by removing the small items’ bit-sequences.
Definition 2.1 (The Header-Table) It contains six domains: item, count, head, level, d-branch, *link*. item is a frequent item by frequency decreasing order, count is item’s support, head is the branch’s head node except the root, the branch contains the item, level is item’s depth in FLUFP-tree, d-branch is item’s depth-branch, *link* points into all nodes with the same item in the tree.

Definition 2.2 (depth-branch) In Header-Table, depth-branch’s initial value is the item of the parent node of the node pointed by *link*, if an item has the same head, the same level, the same depth-branch, we traverse corresponding depth-branches’ parent nodes by corresponding *link*, if items are also same, then we continue to traverse these items’ parent nodes until they aren’t the identical item, the depth-branch is the set of corresponding items traversed.

Definition 2.3 (Fast location Updated Frequent Pattern Tree (FLUFP-tree)) The root of FLUFP-tree is set Null, all nodes besides the root contain several domains: item, count, level, *p*, item is a frequent item, count is item’s support, level is item’s depth in FLUFP-tree, *p* points into all child nodes and parent nodes of the current node.

For example, in Figure 3, all pointers are bi-directional, the root is Null, b:9(1) shows the item is b, b’s count is 9, the depth is 1. The item and count of Header-Table are established before FLUFP-tree construction, but others are set during FLUFP-tree constructed. If node b is constructed, b.head=b, b.level=1, b.d-branch is null because b’s parent node is the root, b’s link points to node b. By analogy, if branch bfgh is constructed, h.head=b, h.level=4, h.d-branch=g, because branch bagh has the same h.head, h.level, and h.d-branch, then we traverse g’s parent nodes by corresponding *link* to check whether nodes’ items are the same or not. For bfgh, g’s parent node is f, but g’s parent node is a for branch bagh, the two items are not identical, then ga and gf are inserted into corresponding h.d-branch.

3. IFPM-BS

3.1 The updated FLUFP-tree construction

The bit-sequence is used to represent all items of original database first, these items are divided into large items, pre-large items and small items. FLUFP-tree is constructed before new transactions come. Header-Table is finished after the tree is constructed, new transactions’ all items are divided into three parts by items’ partition of original database. All cases are processed by Pre-FUFP except case 4. For case 4, if items are pre-large in updated database, they are inserted into pre-large table. Otherwise, they
are inserted into Branch-Items and the end of Header-Table with support descending order, then the original database is operated as follows: let the processed item be I, if ith bit of Bit(I) is 1, ith bits of all bit-sequences of large items in Header-Table are projected into a matrix, Bit(I) contained bit 1 is stored into the matrix’s left part, and projected bit-sequences are stored into right part by support ascending order, if there are many processed items, they are ordered by support descending order. Then the matrix is ordered according to bit 1 before bit 0 for each column from left to right of the matrix’s right part, and let num be the number of ones contained in the matrix’s right part; If first item is X in the matrix’s right part and first bit is 1, we search last bit in the same row: if last bit is 1 and we let corresponding item is Y, we search X.head=Y, X. level=num in Header-Table, if there is only a numth level, items whose corresponding bit is 1 in matrix’s left part are inserted into the node pointed by X.link by support descending order, and related information is updated, the row is deleted and we scan next row. If there is not only a numth level, let next item in right of X be Z, if corresponding bit is 1, items are inserted into the node pointed by X.link while XZ is equal to X.d-branch, and related information is updated, the row is deleted and we scan next row, otherwise, we search next item in matrix’s right part until there is an itemset equal to X.d-branch. If last bit is not 1, its left next bit is scanned until corresponding bit is 1, then we process next work with the same situation; If first bit of X is not 1, the projected bit-sequence of X is deleted, and next item at the matrix is null. while the corresponding bit is 1, then we process next work with the same situation. When the right part of the matrix is null, the items are directly inserted into the root of FLUFP-tree, the operation is finished while the matrix is null.

Table 1. The updated FLUFP-tree construction

| INPUT: | An old database D (d+c), original FLUFP-tree, corresponding Header-Table, an upper support threshold S_u, a lower support threshold S_l, the pre-large table, and t new transactions. |
| OUTPUT: | An updated FLUFP-tree. |

1: Calculate f, D and new transactions are transformed all items’ bit-sequences, and items of new transactions are divided into three parts, Pre-FUFP is used for all cases except case 4.
2: Do the following steps for case 4:
2.1: If items are pre-large in updated database, then insert them into the pre-large table;
   Else insert them into the end of Header-Table with support descending order
   for bit-sequences of all items of the original database existing in Branch-Items
   (for bit-sequences of all items in Header-Table into Φ and insert them into Φ’s left part by support descending order; Φ’s right part stores projected bit-sequences by support ascending order, and order Φ by 1 before 0 for the same item’s bit-sequence from left to right in the Φ’s right part; Calculate num(the number of ones contained in Φ’s right part));
   Repeat // traverse the ordered Φ
   Let first item is X in Φ’s right part;
   if X’s first bit is 1
   {while (the item of last bit in the same row is not X) //locate the node
    if the last bit is 1 and corresponding item is Y // the last bit is 1
    {search X.head=Y & X.level=num in Header-Table;
     if X only contain a numth level, insert items which corresponding bit is 1 in Φ’s left part by support descending order into the node pointed by X.link and update information; delete this row and scan next row in Φ’s right part;
     else search next item I right of X // X don’t contain a numth level
      let Bi={all X.d-branch in numth level } and Bi is the ith X.d-branch
      while (I != Y)
       {if I’s corresponding bit is 1 in Φ at the same row;
        if I=Bi, insert items which corresponding bit is 1 in Φ’s left part into the node pointed by X.link, and delete this row and scan next row in Φ’s right part;
        else Bi=Bi-1, search next item m in right of I and I=m;
        else search the next item m in the right of I and I=m;}}
      else search next bit in left of the last bit, and let the bit be the last bit;
      else delete X’s projected bit-sequences, and let next item in right of X be the first item
      if Φ’s right part is null, insert corresponding items in Φ’s left part as the root’s child; }
     else search X’s projected bit-sequences, and let next item in right of X be the first item
    } until Φ is null ;}
2.2: For bit-sequences of new transactions’ all items existing in Insert-Items, the process is the same as 2.1, but if the item is inserted into the node of FLUFP-tree, all items’ counts between the node and the root node are increased 1.
3: Finally if t+c > f, then set d = d + t + c and set c = 0; otherwise, set c = t + c.

After the above operation is finished, the set of Insert-Items is all large items contained in the Header-Table, while the new transactions are processed, the bit-sequences of these large items are
projected, the methods of the matrix scanning and the node locating of the FLUFU-tree are the same as above operation. While the items are inserted into FLUFU-tree, the support of all items from the leaf node to the root will be increased 1, and if an item is inserted and it is existed in the tree, then we only need to increase the item’s support.

The processing for all cases is similar to Pre-FUFP algorithm except case 4: In step 2.1, if a pre-large item is changed into a large item, projected bit-sequences contained in the matrix and Header-Table information can help us to locate nodes of FLUFU-tree directly, so we don’t need to determine which transactions contain items in Branch-Items in the original database. And the procedure of items contained in Insert-Items for new transactions is the same as step 2.1, but we still need to update items’ counts in FLUFU-tree. Final we reset the value of c; In the mining phase, we can locate the identical item by *link, and the procedure is the same as FP-growth.

3.2. An example of the updated FLUFU-tree construction

Assume the lower support threshold $S_l$ is 30% and the upper one $S_u$ at 50%, the original database and bit-sequences of large items and pre-large items are shown in Fig.2, initial FLUFU-tree is given in Fig.3, the new transactions and the pre-large table are shown respectively in Table 2 and Table 3.

**Table 2.** The three new transactions

<table>
<thead>
<tr>
<th>Transaction no.</th>
<th>Items</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>abcdfi</td>
</tr>
<tr>
<td>2</td>
<td>abcdi</td>
</tr>
<tr>
<td>3</td>
<td>acdh1</td>
</tr>
</tbody>
</table>

**Table 3.** The pre-large itemset for the original database

<table>
<thead>
<tr>
<th>Items</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>d</td>
<td>4</td>
</tr>
<tr>
<td>c</td>
<td>3</td>
</tr>
<tr>
<td>e</td>
<td>3</td>
</tr>
</tbody>
</table>

First variable c is initially set at 0, IFPM-BS algorithm is conducted as follows:

Step 1: The safety number $f$ for new transactions is 4 by the formula.

Step 2: Scan new transactions to get bit-sequences of all the items and their counts.

Step 3: The items are divide into three parts {a:3,b:2,f:1,g:0,h:1}, {c:3,d:3,e:0} and {i:3} by whether they are large, pre-large or small in the original database.

Step 4: The items in new transactions which are large in the original database are first processed, $S_U(a)=S_U(b)=3+1=4$, $S_U(f)=11$, $S_U(d+c+t)=11/13>0.5$, the count of item a in Header-Table is set 11, and item a is put into Insert-Items, by analogy, the counts of item g and h are 6 in updated database, and $0.3<6/11<0.5$, then item g and h are removed from Header-Table and FLUFU-tree, so the set of Insert-Items is \{a,b,f\}, which is shown in Figure 5(a).

Step 5: The items in the new transactions which are pre-large in the original database are processed, $S_U(c)=3+3=6$, $0.3<6/13<0.5$, item c is a pre-large item, so the count of item c is 6 in pre-large table; $S_U(d)=4+3=7/13>0.5$, item d is changed into a large item of updated database, then item d is put into Insert-Items and Branch-Items; $S_U(e)=3/13<0.3$, so item c is deleted from the pre-large table. After Step 5, we can get Insert-Items = \{a, b, f, d\} and Branch-Items = \{d\}.

Step 6: Since item i is neither large nor pre-large in original database but large in new transactions, it is put into the set of Rescan-Items, so the set of Rescan-Items is \{i\}.

Step 7: Since $t+c=3+4<4$, we don’t need to rescan the original database.

Step 8: The items in Branch-Items are sorted in support descending order and inserted into the end of Header-Table, so item d is inserted with its count, which is shown in Figure 5(b).

Step 9: Bit(d)=1010010100, the 1th, 3th, 6th and 8th are 1, then bit-sequences of large items in Header-Table are projected, and projected bit-sequences are stored in \#, the projected Bit(d) are put into left part of $\Phi$, the projected Bit(f), Bit(a) and Bit(b) are inserted into right part, and these bit-sequences are ordered support ascending order, $\Phi$ is ordered according to 1 before 0 for the same item’s bit-sequence from left to right in $\Phi$’s right part, num is the number of ones contained in right part of $\Phi$, which is shown in Figure 4. Since first item is f in right part of $\Phi$, in the first row, first bit is 1, last bit is also 1 and the corresponding item is item b, num=3, we search f.head=b and f.level=3 in Header-Table, because item f has only a 3th level, then item d is inserted into the node f:5(3) pointed by f.link, let item d be this node’s child node, d.count is set 1, and in Header-Table, d.head=b, d.level=4, d.branch=f, d.link points into item d, $\Phi$’s first row is deleted, next row is scanned, item f’s first bit is 1, the situation is the same as above, the result is shown in Figure 5(c).

Then we continue scan $\Phi$’s right part, since the first 2 rows are deleted, first bit is 0 of item f, the
projected Bit(f) is deleted, so first item is item a and first bit is 1, last bit is also 1 and the corresponding item is item b, num=2, we check a.level=b, a.level=2, in Header-Table, since item a has also only a 2th level, item d is inserted into the node pointed by a.link, and related information is updated, first row of $\Phi$ is deleted and next row is scanned, since first bit is 1 of item a, but last bit is 0 of item b, then left bit is checked of last bit, this bit is 1 and corresponding item is item a, num=1, item d is inserted into the node of FLUFP-tree pointed by a.link which item a has a.head=a, a.level=1 in Header-Table, the result is shown in Figure 5(d) after related information is updated.

Figure 4. The matrix $\Phi$ of the projected bit-sequences

![Header Table and FLUFP-tree](image)

(a) The Header-Table and the FLUFP-tree after item g, h are pruned
(b) The Header-Table and the FLUFP-tree after item d is added to the Header-Table
(c) The Header-Table and the FLUFP-tree after scanning the first 2 rows of $\Phi$
(d) The Header-Table and the FLUFP-tree after step 9

Figure 5. The updated FLUFP-tree construction

Step 10: Insert-Items={b, a, f, d}, for the bit-sequences of all the items of the new transactions existing in the Insert-Items, the process is the same as step 9, but if the item is inserted into the node of the FLUFP-tree, all the items’ count between the node and the root node is increased 1, and while the item is inserted, if the item has existed, then we only need to increase the item’s count. The final result is shown in Figure 6.

Step 11: Since $t+c (3+0=3)>f(4)$, then we set $c = t + c = 3 + 0 = 3$.

Figure 6. The final FLUFP-tree after all the new transactions are processed.

After Step 11, the FLUFP-tree is 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. The desired large itemsets can then be found by the FP-Growth mining approach, though there has no pointers during the identical item, we can locate the identical item according to the item’s *link in the Header-Table.
4. Experimental Evaluation

In this paper, we test the effectiveness and scalability of IFPM-BS, and let the algorithm compare with FP-tree and Pre-FUFP algorithm. All of our experiments are conducted on a PC with Intel Core 2 Duo 2.93 GHz CPU and 1GB RAM; FIM-BS is implemented in Visual C++ 6.0. A real dataset called BMS-POS is used in the experiments. There are 515,597 transactions with 1657 items in the dataset. The maximal length of a transaction is 164 and the average length of the transactions is 6.5.

4.1 Performance

The first 500,000 transactions are extracted from the BMS-POS database to construct the initial trees, the next 2000 transactions are then used in incremental mining.

The running time of FP-tree, Pre-FUFP and IFPM-BS in BMS-POS is shown in Figure 7, for Pre-FUFP and IFPM-BS, the upper minimum support threshold is set at 1-5% (1% increment each time) and the lower minimum support threshold is set at 0.5-2.5% (0.5% increment each time). It can be observed from Figure 7 that IFPM-BS ran faster than others, because FP-tree need to be restructured while new transactions come, Pre-FUFP uses existed FUFP-tree, but if a pre-large item is changed into a large item, it need to determine which transactions contain the item in original database to update FUFP-tree, while IFPM-BS adopts bit-sequence to compress transaction database and nodes of tree are located by the matrix contained projected bit-sequences, which don’t need to rescan original database.

The comparison result of the numbers of nodes for the three algorithms is shown in Figure 8. It can be seen that the three algorithms generate nearly the same sizes of trees, the effectiveness of IFPM-BS is thus acceptable, because the three algorithms adopt the pattern growth method to construct the corresponding trees, and the nodes of the trees only contain the large items, the large items are identical for the same dataset, so the number of the nodes for the three algorithms is nearly identical.

4.2 Scalability

The minimum support threshold is 4% for FP-tree, for Pre-FUFP and IFPM-BS, the upper and the lower support thresholds are set at 4% and 2%, respectively. The first 500,000 transactions are
extracted from BMS-POS database to construct initial trees, next 2000 transactions are then sequentially used each time as new transactions for the experiments.

Figure 9 shows execution times required by FP-tree, Pre-FUFP and IFPM-BS for processing each 2000 new transactions. From the graph we can see that the running time of IFPM-BS almost increases linearly as the number of transactions in the database increases, since the number of new transactions is no more than the safety number, then IFPM-BS needs not to be rescanned the original database, which can reduce the running time, this shows that the proposed algorithm can utilize the information carried from previous mining well and can incrementally generate frequent itemsets efficiently.

5. Conclusion

In this paper, an incremental algorithm for frequent pattern mining based on bit-sequence IFPM-BS is proposed. First the database is converted into bit-sequences, and the concept of pre-large itemset is cited, the original database is not rescanned until a number of new transactions have been inserted; And then we define depth-branch and FLUFP-tree structure, depth-branch is contained in Header-Table of FLUFP-tree to locate nodes of the tree, if a pre-large item is changed into the large item, we can locate nodes of the tree fast according to the matrix structure and Header-Table information, projected bit-sequences are contained in the matrix, and we don't need to determine which transactions contain the item in original database, and we can get mining results by FP-growth. So the cost of time is reduced. Our experiment results show that IFPM-BS has a linear scalability and better performance.

6. Acknowledgements

The authors also gratefully acknowledge to the reviewers whose the valuable comments and suggestions helped to improve the presentation. This research is supported by the Natural Science Foundation of Hebei Province P. R. China under Grant No.F2010001298.

7. References