Association Analysis with One Scan of Databases*

Hao Huang (+), Xindong Wu (*) and Richard Relue (+)
(+) Dept. of Math and Computer Science, Colorado School of Mines
Golden, Colorado 80401, USA
(*) Department of Computer Science, University of Vermont
Burlington, Vermont 05405, USA

Abstract

Mining frequent patterns with an FP-tree avoids costly candidate generation and repeatedly occurrence frequency checking against the support threshold. It therefore achieves better performance and efficiency than Apriori-like algorithms. However, the database still needs to be scanned twice to get the FP-tree. This can be very time-consuming when new data are added to an existing database because two scans may be needed for not only the new data but also the existing data. This paper presents a new data structure P-tree, Pattern Tree, and a new technique, which can get the P-tree through only one scan of the database and can obtain the corresponding FP-tree with a specified support threshold. Updating a P-tree with new data needs one scan of the new data only, and the existing data do not need to be re-scanned.

1 Introduction

An association rule is an implication of the form $X \Longrightarrow Y$, where X and Y are sets of items and $X \cap Y = \phi$. The support s of such a rule is that s% of transactions in the database contain $X \cup Y$; the confidence c is that c% of transactions in the database contain X also contain Y at the meantime. A rule can be considered interesting if it satisfies the minimum support threshold and minimum confidence threshold, which can be set by domain experts. Most of the previous research with regard to association mining was based on Apriori-like algorithms [1]. They can be decomposed into two steps:

1. Find all frequent itemsets that hold transaction support above the minimum support threshold.
2. Generate the desired rules from the frequent itemsets if they also satisfy the minimum confidence threshold.

Apriori-like algorithms iteratively obtain candidate itemsets of size $(k+1)$ from frequent itemsets of size k. Each iteration requires a scan of the original database. It is costly and inefficient to repeatedly scan the database and check a large set of candidates for their occurrence frequencies. Additionally, when new data come in, we have to run the entire algorithms again to update the rules.

Recently, an FP-tree based frequent patterns mining method [2] developed by Han et al achieves high efficiency, compared with Apriori and TreeProjection [3] algorithms. It avoids iterative candidate generations.

The rest of the paper is organized as follows. We review the FP-tree structure in Section 2. In Section 3, we introduce a new FP-tree based data structure, called pattern tree, or P-tree, and discuss how to generate the P-tree by only one database scan. How to generate an FP-tree from a P-tree is discussed in Section 4. Section 5 deals with updating the P-tree with new data, and Section 6 provides a reference for our experimental results.

2 Frequent Pattern Mining and the Frequent Pattern Tree

The frequent-pattern mining problem can be formally defined as follows. Let $I = \{i_1, i_2, ..., i_n\}$ be a set of items, and D be a transactions database, where each transaction T is a set of items and $T \subseteq I$. An unique identifier, called its *TID*, is assigned with each transaction. A transaction T contains a pattern P, a set of items in I, if $P \subseteq T$. The support of a pattern P is the number of transactions containing P in D. We say that P is a frequent pattern if P's support is no less than a predefined minimum support threshold ξ.

A frequent pattern tree is a prefix-tree structure storing frequent patterns for the transaction database, where the support of each tree node is no less than a predefined minimum support threshold ξ. The frequent items in each path are sorted in their frequency descending order. More frequently occurring nodes have better chances of sharing the prefix strings than less frequently occurring ones, that is to say, more frequent nodes are closer to the root than less frequent ones. In short, an FP-tree is a highly compact data structure, "which is usually substantially smaller than the original database, and thus saves the costly database scans in the subsequent mining processes" [2].

After the construction of an FP-tree, we can use this data

*This research is supported in part by the U.S. Army Research Laboratory and the U.S. Army Research Office under grant number DAAD19-02-1-0178.

structure to efficiently mine the complete set of frequent patterns with the FP-growth algorithm, which is a divide-and-conquer method performed as follows:

1. Derive a set of conditional paths, which co-occurs with a suffix pattern, from the FP-tree.
2. Construct a conditional FP-tree for each set of the conditional paths.
3. Execute the frequent pattern mining recursively upon the conditional FP-tree.

The study in [2] shows that the FP-growth algorithm is more efficient and scalable than both Apriori and TreeProjection [3]. The FP-tree based algorithm has some inherent advantages: the new data structure is desirably compact and the pattern growth algorithm is efficient with the data structure. But it also has the following problems:

1. A new FP-tree requires scanning the database twice.
2. Although a validity support threshold, watermark [2], is realizable, there is no guarantee of complete database information for the FP-tree when new data come into the database.
3. If the specific threshold is changed, we will have to rerun the whole FP-tree construction algorithm, that is, rescan the database twice to get the new corresponding frequent item list and a new FP-tree.
4. Even if the threshold remains the same, an FP-tree can't be constructed or updated at real-time. Each construction or updating needs to go from scratch, and scan the new and old data twice.

3 Patterns Generation with the Pattern Tree

The FP-tree based method has to scan the database twice to get an FP-tree, whose central idea is to get the list L of item frequencies in the first time and then construct the FP-tree in the second time according to L.

A Pattern Tree (P-tree for short), unlike FP-tree, which contains the frequent items only, contains all items that appear in the original database. We can obtain a P-tree through one scan of the database and get the corresponding FP-tree from the P-tree later.

The construction of a P-tree can be divided into two steps as well:

1. When retrieving transactions from a database, we can generate a P-tree by inserting transactions one by one after we sort the items of each transaction in some order (alphabetic, numerical or any other specific order), and meanwhile record the actual support of every item into the item frequency list L.
2. After the first and only scan of the database, we sort L according to item supports. The restructure of the P-tree consists of similar insertions in the first step. The only difference is that one needs to sort the path according to L before inserting it into a new P-tree.

This approach makes the best use of the occurrence of the common suffix in transactions, thereby constructing a more compact tree structure than FP-tree.

3.1 Algorithm

Algorithm 1 (P-Tree Generation)

Input: A transaction database DB and a minimum support threshold *minisup*

Output: A Pattern tree

The pattern tree can be created in two steps:

Step 1: Construct a P-tree P and obtain the item frequency list L.

(1) $P \leftarrow$ *Root*
(2) $L \leftarrow \phi$
(3) For each transaction T in the transaction database
   a. Sort T into $[t \mid T_i]$ in alphabetic order. Here in each sorted transaction $T = [t \mid T_i]$, t is the first item of the transaction and T_i is the remaining items in the transaction.
   b. *Insert*($[t \mid T_i]$, P)
   c. Update L with items in $[t \mid T_i]$

The function *Insert*($[t \mid T_i]$, P) performs as follows.

Function *Insert*($[t \mid T_i]$, P)
BEGIN
  FOR each of P's child node *N*
    IF *t.itemName* = *N.itemName*
      THEN
        N.frequency←*N.frequency*+1
        IF T_i is not empty
          THEN
          Insert(T_i,*N*)
        ENDIF
        RETURN
    ENDIF
  ENDFOR
  Create a new Node *N'*
  N'.itemName←*t.itemName*
  N'.frequency←1
  P.childList←*N'*
  IF T_i is not empty
    THEN *Insert*(T_i, *N'*)
  ENDIF
  RETURN
END

Step 2: Restructure the initial P-tree P

(1) $newP \leftarrow$ *Root*
(2) For each path p_i from the root to a leaf in the initial P-tree P, Until $p_i = \phi$ do:
   a. The common support of each item in p_i is that of the node next to the last branching-node. If there is no branching-node in p_i, the common support of each item is the actual support of each item in p_i.
   A branching-node is a node after which there exists more than one branch in the tree.
   b. Get a sub-path $p_i^{'}$ from p_i with the common support for every item.
   c. Sort $p_i^{'}$ according to L.
   d. Insert the sorted $p_i^{'}$ into the new P-tree, by calling function *Insert*($p_i^{'}$, *newP*).
   e. $p_i \leftarrow p_i - p_i^{'}$.

3.2 Analysis

The P-tree generation algorithm needs exactly one scan of the database and one scan of the initial P-tree. The running time depends on how the patterns distribute in the database. The more high frequent patterns in the database, the faster the algorithm will be. The lower bound is the runtime of one scan of the database. In the contrary, the less the high frequent patterns in the database, the slower the algorithm will be. The upper bound is the runtime of two database scans.

3.3 Pattern Tree: A Formal Definition

A pattern tree (or P-tree for short) is a rooted tree structure, which has the following properties:

1. The root is labeled as "*Root*". All other items are either its children or its descendants.
2. Each node except the root is composed of three fields: *itemName*, *frequency* and *childList*, where *itemName* stands for the actual item in the transaction database, *frequency* represents the transaction support of the item in the database, and *childList* stores a list of its child nodes.
3. A path in a P-tree represents at least one transaction and the corresponding occurrence(s), which is the *frequency* of its least frequent item(s).
4. A node holds more or equal frequency to its children or descendants. Note that the root node doesn't have the actual meaning in transactions, so we don't consider its frequency.
5. A prefix shared by several paths represents the common pattern in those transactions and its frequency. The more paths share the prefix, the higher frequency it has.

4 FP-Tree Generation from the P-Tree

From the definition of the P-tree, we can observe that an FP-tree is a sub-tree of the P-tree with a specified support threshold, which contains those frequent items that meet this threshold and hereby excludes infrequent items. We will propose an algorithm and analyze it in this section.

4.1 Algorithm

After the generation of the P-tree, we can easily get the frequent item list given a specific support threshold. All we need to do is to get rid of those infrequent items from item frequency list L. Next, we prune the P-tree to exclude the infrequent nodes by checking the frequency of each node along the path from the root to leaves. Because the frequency of each node is not less than that of its children or descendents, we delete the node and its subtrees at the same time if it is infrequent.

Algorithm 2 (FP Generation from the P-Tree)
Input: A P-tree P, the frequency list L, & the support threshold ξ
Output: An FP-tree

1. Frequent Item List $FIList \leftarrow \phi$
2. For each item *i* in *L*
   If *i.frequency* $\geq \xi$
   Add *i* to *FIList*
3. Sort *FIList* in frequency descending order
4. Invoke *check*(P). The function *check* is described as follows.

Function *check*(*N*)

```
BEGIN
  FOR each child c of the node N
    IF c ∈ FIList
      THEN
        check(c)
      ELSE
        Delete c (and the possible
        subtree starting from c)
    ENDIF
  ENDFOR
  RETURN
END
```

4.2 Analysis

In practice, we can compare the user-defined minimum support threshold with the occurrence recorded in the item frequency list. So the pruning could be done according to the following two rules:

1. If the minimum support threshold is higher than the occurrence of most items, then we can check the items along the path beginning from the root as mentioned in Section 3.1. Once an infrequent item is found, its subtree including itself is deleted from the pattern tree.
2. When the occurrence of most items is above the minimum support threshold, we can check the items along the path beginning from the leaves, the inverse order with the first rule. As long as a frequent item is found, we keep it and prune its subtree.

Regardless of which rule is applied, the algorithm checks at most half amount of items in a pattern tree. In the mining process, the users always need to adjust the support thresholds to achieve an appropriate one. If the support threshold is set too high, the process may produce fewer frequent items and some important rules can not be generated. On the other hand, if the support threshold is set too low, the process may produce too many frequent items and some rules may become meaningless. One advantage of our approach is that we can easily get different FP-trees corresponding to different support thresholds. When the support threshold is changed, no further database scans are needed.

5 Updating the Pattern Tree with New Data

One concern with the P-tree is how to update it with new data. In this section, we will propose an algorithm to solve the problem and illustrate the process with an example.

As the database can always be updated, how to update the old rules is an important problem in data mining. There

are two ways to update an FP-tree. One is to apply the construction algorithm to the new database, i.e. scan the updated database twice. In this case, the previous two scans of the old database are discarded. The other is to set "a validity support threshold (called watermark)" in [2]. The watermark goes up to exclude the originally infrequent items while their frequency goes up. But it may need to go down since the frequency of frequent items may drop when more and more transactions come in. This solution can't guarantee the completeness of the generated association rules. With new information the originally infrequent items may become frequent and vice versa.

Since we can generate the P-tree by scanning the database only once, we are also able to update the P-tree by one scan of new data without the need for two scans of the existing database and the second scan for the new data.

We can first insert the new transactions into the P-tree according to the item frequency list and meanwhile update the list. Then a new P-tree can be restructured according to the updated item frequency list. In the case there comes a new item, which does not appear in the existing database, we can assume its support is 0 and append it as a leaf node.

5.1 Algorithm

Algorithm 3 (P-Tree Updating)

Input: The original P-tree, *P1*, the original item frequency list, *L*, and a new transaction database *DB'*. (Note that with a compact format the original P-tree *P1* contains all items in the existing transaction database no matter whether or not they are frequent.)

Output: Updated pattern tree, *P2*

Step 1: Expand P1 using new data and meanwhile update L.

(1) For each transaction T in the new transaction database *DB'*

   a. Sort T according to the original frequency list *L*
   b. *Insert*(T, *P1*)
   c. Update *L* with items in T

(2) Sort *L* in frequency descending order.

Step 2: Restructure the expanded P-tree P1 into P2 according to the updated L.

(1) $P2 \leftarrow Root$

(2) For each path p_i in *P1*,
Until $p_i = \phi$ do:

   a. Let s be the common support of each item in p_i.
   b. Get a sub-path $p_i^{'}$ from p_i with the common support for every item.
   c. Sort $p_i^{'}$ according to *L*.
   d. *Insert*($p_i^{'}$, *P2*).
   e. $p_i \leftarrow p_i - p_i^{'}$.

5.2 Analysis

The most difficult problem concerning the FP-tree is to handle updates in the database. Once some new transactions are added, a new FP-tree has to be constructed to deal with these changes. The main advantages of the above algorithm in Section 5.1 are:

1. There is no further need to scan the existing database, because the original P-tree is already a compact version. Thus, the algorithm makes updating the P-tree more efficient by reusing the old computations on the original database.
2. We need to scan the new data only once. According to [2], an FP-tree is obtained by two scans of the entire database, including the existing and new database.
3. In the worst case, the cost of our algorithm is still $O(m * n)$, where m is the maximum length of transactions and n the number of the transactions in the database.

6 Tests and Results

We have performed experiments with multiple FP-tree generation and FP-tree updating while new data are added. Our test results show that the P-tree method outperforms the FP-tree method by an factor up to an order of magnitude in large datasets. The test environment, test databases, and detailed results are omitted in this paper due to size restrictions and can be found in [4].

7 Conclusions

We have proposed a new data structure, pattern tree or P-tree, and discussed how to obtain the P-tree by one database scan and how to update the P-tree by one scan of new data. Moreover, we have addressed how to get the corresponding FP-trees from the P-tree with different user-specified thresholds and also the completeness property of the P-tree. We have implemented the P-tree method and presented the test results in [4], showing that our method always outperforms the FP-tree method.

The key point of our method is to make best use of the P-tree structure, which presents a large database in a highly condensed format, and avoids the second database scan.

References

[1] M.-Y. Chen, J. Han, and P. Yu. Data Mining: An Overview from a Database Perspective. *IEEE Transactions on Knowledge and Data Engineering*. **8**(6): 866–883, 1996.

[2] J. Han, J. Pei, and Y. Yin. Mining Frequent Patterns Without Candidate Generation. *Proc. of ACM Int. Conf. on Management of Data (SIGMOD)*, 1–12, 2000.

[3] R. Agarwal, C. Aggarwal, and V. V. V. Prasad. A Tree Projection Algorithm for Generation of Frequent Itemsets. *Journal of Parallel and Distributed Computing*, **61**(3): 350–371, 2001.

[4] H. Huang, X. Wu and R. Relue. Association Analysis with One Scan of Databases. *University of Vermont Computer Science Technical Report CS-02-3*, 2002. `http://www.cs.uvm.edu/tr/CS-02-03.shtml`