I-Tree: A Frequent Patterns Mining Approach without Candidate Generation or Support Constraint

Syed Khairuzzaman Tanbeer, Jehad Sarkar, Byeong-Soo Jeong, Young-Koo Lee, Sungyoung Lee
Dept. of Computer Engineering, Kyung Hee University
e-mail: tanbeer@khu.ac.kr, jehad@oslab.khu.ac.kr, {jeong, yklee}@khu.ac.kr, sylee@oslab.khu.ac.kr

Abstract

Devising an efficient one-pass frequent pattern mining algorithm has been an issue in data mining research in recent past. Pattern growth algorithms like FP-Growth which are found more efficient than candidate generation and test algorithms still require two database scans. Moreover, FP-growth approach requires rebuilding the base-tree while mining with different support counts. In this paper we propose an item-based tree, called I-Tree that not only efficiently mines frequent patterns with single database scan but also provides multiple mining scopes with multiple support thresholds. The 'build-once-mine-many' property of I-Tree allows it to construct the tree only once and perform mining operation several times with the variation of support count values.

1. Introduction

Mining association rules to extract the relationships between different items in a database was first introduced by Agrawal et. al.[1]. The mining process includes two phases; mining the underlying frequent patterns and then generating association rules. Since generation of rules is rather straightforward, the main focus of researchers has been to optimize the complexity of frequent patterns detection steps. A good number of follow-up research works [5] have been published presenting new algorithms or improvements on existing algorithms to solve the frequent pattern mining problem more efficiently. In practice, as association rule mining is an iterative process, multiple frequent patterns mining processes with different values of minimum support count are sometimes required to get any acceptable result.

This paper proposes a 'build-once-mine-many' item-based tree technique, called I-Tree, which allows the users for multiple frequent patterns mining with different support counts. Moreover, allowing only one database scan I-Tree offers an enhancement in mining efficiency. The key contribution of the paper is the development of a simple, but yet efficient, tree structure for mining frequent patterns in an updated support constraint.

The construction of I-Tree requires the assumption that there is no limitation on the size of main memory. Since the current trend of modern computing is marching towards computers with huge amounts of main memory, the assumption is reasonable for a considerably large database. Moreover, deliberation of the same assumption has been found in several publications [2, 7, 3, 4, 8] in literature.

The rest of the paper is organized as follows. In section 2, we briefly review some techniques found in the literature related to the scope of the paper. Section 3 contains the design and detail construction process of I-Tree. The algorithm for constructing I-Tree is given in this section. I-Tree pruning and mining technique using I-Tree are discussed in section 4. In section 5, we discussed the major efficiency issues of I-Tree with respect to FP-Growth technique. This section also concludes the paper.

2. Related Works

Mining frequent patterns with only one database scan has been a challenging issue since its introduction as a research issue. Candidate-generation, say Apriori [1], and pattern-growth, say FP-Growth [6], methods are the prominent classes of algorithms found in literature. Although the shortcoming of candidate-generation has been well addressed by FP-growth approach it cannot reduce the number of database scan to one. FP-Growth being considered as one of the fastest algorithms to generate the frequent patterns depends on two database scans: one for generating the frequent distinct item list and the other for tree construction based on the frequent items.

Although CATS tree [2] and CanTree [7] are two approaches to perform the mining task with only one database scan both of them are suffering from computationally expensive tree construction steps. The former one requires merging and swapping of nodes during tree construction, and both downward and upward traversal during the mining phase.

3. I-Tree

I-Tree is constructed at a straightforward manner of inserting every transaction in database one after another into it. Before inserting any transaction, the transaction is preprocessed by sorting its items according to item's appearance order in the database. Unlike FP-Growth technique it prunes the constructed tree based on the support threshold given by user. I-Tree is an item-order tree, since it is built based on the sequence of occurrence of each item in database. It is a compact representation of database, as one or more transactions may share a single path in the tree.

3.1. Construction of I-Tree

Definition 1 (Sort-list). Sort-list is the list that includes each distinct item found in all transactions (i.e., in whole database) according to their appearance order. It also contains the frequency of each item in the database.

Sort-list is constructed dynamically while scanning any transaction starting from the very first one. The list is populated by the items in an order such that the most recent distinct item which has not appeared in any previous transaction(s) so far is placed at the bottom of the list. Each time any item is inserted or accessed in sort-list the count value of that item is incremented by one. Since scanning the first transaction it is being built and populated with 'new' items.

Initially the I-Tree is empty and starts construction with a 'null' root node. Like FP-Tree, I-Tree contains nodes representing items and total number of passes (i.e., count) of that item in the path upto that node. It follows the FP-Tree construction technique while inserts any sorted transaction into the tree. New transactions, being sorted according to the sort-list order, are added at the root level. Since at the beginning sort-list was empty, items of first transaction are inserted according to as-it-is order in transaction and hence the order of items of first transaction is not required to be altered.

Table 1: Transactions

TID	Original Transactions	Sort List	Sorted Transactions
1	<f, a, c, d, g, i, m, p>	f, a, c, d, g, i, m, p	<f, a, c, d, g, i, m, p>
2	<a, b, c, f, l, m, o>	f, a, c, d, g, i, m, p, b, l, o	<f, a, c, m, b, l, o>
3	<b, f, h, j, o>	f, a, c, d, g, i, m, p, b, l, o, h, j	<f, b, o, h, j>
4	<b, c, k, s, p>	f, a, c, d, g, i, m, p, b, l, o, h, j, k, s	<c, p, b, k, s>
5	<a, f, c, e, l, p, m, n>	f, a, c, d, g, i, m, p, b, l, o, h, j, k, s, e, n	<f, a, c, m, p, l, e, n>

The construction process of I-Tee is described using the database shown in Table-1. Table-1 not only shows the progress of sort-list population with 'new' items found in 'new' transactions but also gives the order of items in each transaction to be maintained to insert the transaction into I-Tree. Figure-1 shows the final I-Tree built based on the database of Table-1. TID1 <f,a,c,d,g,i,m,p> is inserted as-it-is manner that results the first branch of the tree (in Figure-1) being 'f' the initial node (just after the root node) and 'p' the last node. Before inserting TID 2, items of TID 2 are sorted from <a,b,c,f,l,m,o> order to <f,a,c,m,b,l,o> to maintain the sort-list order. It can be noticed that items 'b', 'l', and 'o' are 'new' items found in TID 2 and, therefore, are inserted at the tail of sort-list first according to the order of sequence they appeared in TID 2. The frequency count field of every item in sort-list is updated along with the increment of count field of each node of the transaction in tree. After mapping all transactions (say TIDs 3, 4, and 5) into the tree, nodes representing the same item are linked and the same item in sort-list points to the first node in the tree.

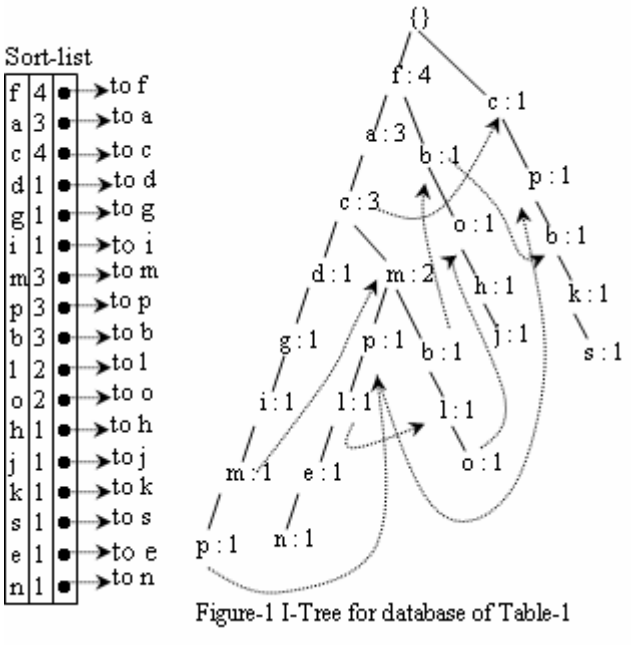

Figure-1 I-Tree for database of Table-1

Property 1 The total count of any node in I-Tree is greater than or equal to the sum of total count of its children.

3.2. I-Tree Construction algorithm

Step 1: Read a transaction.

Step 2: Populate the sort-list with 'new' items found in the transaction.

Step 3: Sort the items of the transaction according to item sort order of sort-list.

Step 4: Insert the sorted transaction into I-Tree.

Step 5: Increment count value of each node of the path of the transaction.

Step 6: Goto next transaction.

Step 7: If no transaction found then goto Step 8, otherwise goto Step 1.

Step 8: Point each item in sort-list to respective node that appeared first in the tree and link the nodes of same item all through the tree and terminate I-Tree construction.

4. Frequent Pattern Mining Using I-Tree

Once I-Tree is constructed, it can be used for mining frequent patterns with different minimum support (*min_sup*) values. The first step is to prune the I-Tree according to *min_sup* value given by the user. The sort-list is updated by removing all infrequent (items having count less than *min_sup*) items one after another and at the same time the tree is pruned by

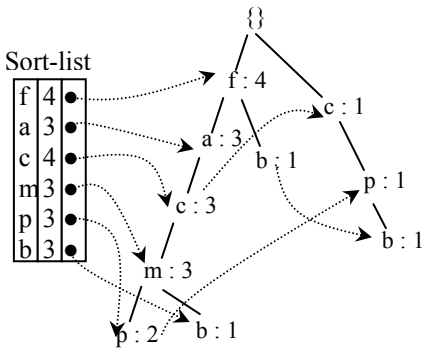

Figure-2 Pruned I-Tree (*min_sup* = 3)

deleting all nodes representing that item. The resulting pruned I-Tree can be, interchangeably, named as 'frequent I-Tree', since it only contains all items which are found frequent based on a given *min_sup*. Pruned I-Tree of Figure-2 represents the 'frequent I-Tree' for the I-Tree of Figure-1 for the value of *min_sup*, 3. In the pruning phase while searching the sort-list for infrequent items from top to bottom approach, item 'd' (as its support is less than *min_sup*) is the first item to be removed from the list and from the I-Tree, respectively. Therefore, record of item 'd' is deleted from sort-list and at the same time all nodes representing 'd' in I-Tree are also eliminated. As a result, node 'c:3' in the left-most path of the tree is made the parent node of node 'g:1' in the same path. As node 'd:1' does not have any link to same item in I-Tree, pruning for item 'd' is terminated and next infrequent item 'g' is picked from the sort-list and removed from both of sort-list and I-Tree in a similar fashion. Under the given *min_sup* all other infrequent items are 'i', 'l', 'o', 'h', 'j', 'k', 's', 'e', and 'n'. Removal of records of these items from sort-list and pruning all corresponding nodes from original I-Tree according to sort-list order results the pruned I-Tree of Figure-2.

Property 2 Pruned I-Tree only contains the nodes of items that satisfy the support threshold given by user.

Since pruned I-Tree holds property 1 and property 2, once it is constructed frequent patterns can be mined from it in a divide-and-conquer method similar to FP-growth.

5. Discussion and Conclusion

The main contribution of the paper is to provide an efficient mining technique using only one database scan. Like FP-tree, the pruned I-Tree contains only frequent items. The build-once-mine-many strategy allows the user to mine a single database with different support counts using only one database scan. While mining with different *min_sup*s I-Tree requires only rebuilding the pruned I-Trees (satisfying the minimum support value) from already constructed original I-Tree. On the other hand, FP-Growth technique demands for another pass while it is used to generate frequent patterns for different support thresholds on a single database.

The updated sort-list for pruned I-Tree contains exactly the same items (may not be in same order) the header table of FP-Growth approach would hold for the same dataset and same *min_sup* value. In the best case, structures of both trees are same but in other cases pruned I-Tree may contain some extra nodes, since items in sort-list are not ensured to be sorted according to support descending order which is maintained in header table of FP-Tree. Although these extra nodes may cause some mining computation overhead and memory consumption, the overall gain in time complexity is remarkably increased as I-Tree passes the database only once. Therefore, database rescanning and tree rebuilding problem in FP-growth technique on mining with different support thresholds for same database is focused and totally avoided in I-Tree and hence giving it more flexibility and efficiency in mining frequent patterns.

Reference

[1] R. Agrawal, T. Imielinski, and A. Swami. Mining association rules between sets of items in large databases. *Proc. SIGMOD 1993*, 207-216.

[2] W. Cheung, and O.R. Zaiane. Incremental mining of frequent patterns without candidate generation of support constraint. *IDEAS'03*. 2003.

[3] K. Wang, L. Tang, J. Han, and J. Liu. Top down FP-growth for association rule mining. *Proc. PAKDD 2002*, 334-340.

[4] I. Pei, J. Han, and R. Maro. CLOSET: An efficient algorithm for mining frequent closed patterns. *SIGMOD*. 2000.

[5] J. Han, H. Cheng, D. Xin, and X. Yan. Frequent pattern mining: current status and future directions. *Data Min Knowl Disc.* 10th Anniversary Issue. 2007.

[6] J. Han, J. Pei, Y. Yin, and R. Maro. Mining frequent patterns without candidate generation: A frequent-pattern tree approach. *Data Min Knowl Disc, 8*. 53-87. 2004.

[7] C. K. Leung, Q. I. Khan, and T. Haque. CanTree: A tree structure for efficient incremental mining of frequent patterns. *Proc. 5th IEEE Intl Conf. on Data Mining (ICDM'05)*. 2005.

[8] M. J. Zaki, and C.-J. Hsiao. CHARM: An efficient algorithm for closed itemset mining. *SIAM International Conference on Data Mining*. 2002.