ISSN 1746-7659, England, UK
Journal of Information and Computing Science
Vol. 9, No. 2, 2014, *pp*. 083-090

FP-growth Tree for large and Dynamic Data Set and Improve Efficiency

Rahul Moriwal

Department of computer science & Engineering
Indore Institute of Science & Technology Indore
rmoriwal@gmail.com
(*Received may08, 2013, accepted october 23, 2013*)

Abstract. FP-growth method is an efficient algorithm to mine frequent patterns, in spite of long or short frequent patterns. By using compact tree structure and partitioning-based, divide-and-conquer searching method, it reduces the search costs substantially. But just as the analysis in Algorithm, in the process of FP-tree construction, it is a strict serial computing process. Algorithm performance is related to the database size, the sum of frequent patterns in the database: ω. this is a serious bottleneck. People may think using distributed parallel computation technique or multi-CPU to solve this problem. But these methods apparently increase the costs for exchanging and combining control information, and the algorithm complexity is also greatly increased, cannot solve this problem efficiently. Even if adopting multi-CPU technique, raising the requirement of hardware, the performance improvement is still limited.

Keywords: Divide & Conquer, partitioning-based, parallel projection, data mining, AI

1. Introduction

(1). we can create a temp database for storing all the frequent items ordered by the list of frequent items, Lwe call this temp database as Projection Database (or PDB for short), which is used for projecting, reduce the expensive costs of individual node computation.

(2). we can project the PDB, two columns at a time.1 One column (called current column) is used to computer the count of each different item, the other (previous) column is used to distinguish the node's parent node of current column. we can insert one **level** of nodes into the tree at a time, not compute frequent items one by one. Then, the algorithm performance is only related to the **depth** of tree, namely the number of frequent items of the longest transaction in the database η,

(3). because we only project two columns at a time, only save the information of the current nodes and their parent nodes, if there exist the case as follows: the current nodes' parent nodes are identical, but their parent nodes' parent nodes are different, we couldn't judge how to deal with it. If we add their count regarding them as the same node,

2. DEFINITION AND BASE FORMULATION

conditional-pattern base (a "sub-database" which consists of the set of frequent items occurring with the suffix pattern), constructs its (*conditional*) FP-tree, and performs mining recursively with such a tree. The pattern growth is achieved via concatenation of the suffix pattern with the new ones generated from a conditional FP-tree. Since the frequent item set in any transaction is always encoded in the corresponding path of the frequent-pattern trees, pattern growth ensures the completeness of the result. our method is not *Apriori*-like *restricted generation-and-test* but *restricted test only*. The major operations of mining are count accumulation and prefix path count adjustment, which are usually much less costly than candidate generation and pattern matching operations performed in most *Apriori*-like algorithms. the search technique employed in mining is a *partitioning-based, divide-andconquer method* rather than *Apriori*-like *level-wise generation of the combinations of frequent itemsets*. This dramatically reduces the size of *conditional-pattern base* generated at the subsequent level of search as well as the size of its corresponding *conditional FP-tree*.

A performance study has been conducted to compare the performance of *FP-growth* with two representative frequent-pattern mining methods, *Apriori* (Agrawal and Srikant, 1994) and Tree *Projection* (Agarwal et al., 2001), *FP-growth* outperforms the *Tree Projection* algorithm. our Ftree-based mining method has been implemented in the DBMiner system and tested in large transaction databases in industrial applications. Although *FP-growth* was first proposed briefly in Han et al. (2000), this paper makes additional progress as follows.

– The properties of FP-tree are thoroughly studied. we point out the fact that, although it is often compact, FP-tree may not always be minima.

– Some optimizations are proposed to speed up *FP-growth*, for technique to handle single path FP-tree has been further developed for performance improvements.

– A database projection method has been developed to cope with the situation when an FP-tree cannot be held in main memory—the case that may happen in a very large database.

– Extensive experimental results have been reported. We examine the size of FP-tree as well as the turning point of *FP-growth* on data projection to building FP-tree.

3. FREQUENT-PATTERN TREE: DESIGN AND CONSTRUCTION

Let $I = \{a1, a2, \ldots am\}$ be a set of items, and a *transaction database DB*=$T1, T2, \ldots, Tn$, where Ti (i = [1 . . . n]) is a transaction which contains a set of items in I . The *support*1 (or occurrence frequency) of a *pattern A*, where *A* is a set of items, is the number of transactions containing *A* in *DB*. A pattern *A* is *frequent* if *A*'s support is no less than a predefined *minimum support threshold*, ξ .

A compact data structure can be designed based on the following observations:

(1). Since only the frequent items will play a role in the frequent-pattern mining, it is necessary to perform one scan of transaction database *DB* to identify the set of frequent items (with *frequency count* obtained as a by-product).

(2). If the *set* of frequent items of each transaction can be stored in some compact structure, it may be possible to avoid repeatedly scanning the original transaction database.

(3). If multiple transactions share a set of frequent items, it may be possible to merge the shared sets with the number of occurrences registered as *count*.

database

(1). If two transactions share a common prefix, according to some sorted order of frequent items, the shared parts can be merged using one prefix structure as long as the *count* is registered properly. If the frequent items are sorted in their *frequency descending order*, there are better chances that more prefix strings can be shared. one may construct a frequent-pattern tree as follows. a scan of *DB* derives a *list* of frequent items, (*f* :4), (*c*:4), (*a*:3), (*b*:3), (*m*:3), (*p*:3)(the number after ":" indicates the support), in which items are ordered in frequency descending order. the root of a tree is created and labeled with "*null*".

(1). The scan of the first transaction leads to the construction of the first branch of the tree:(*f* :1), (*c*:1), (*a*:1), (*m*:1),(*p*:1). (2). For the second transaction, since its (ordered) frequent item list *f, c, a, b,m* shares a common prefix *f, c, a*with the existing path *f, c, a,m, p* the count of each node along the prefix is incremented by 1, and one new node (*b*:1) is created and linked as a child of (*a*:2) and another new node (*m*:1) is created and linked as the child of (*b*:1). (3). For the third transaction, since its frequent item list *f, b*hares only the node *f* with the *f* prefix subtree, *f* 's count is incremented by 1, and a new node (*b*:1) is created and linked as a child of (*f* :3).(4). The scan of the fourth transaction leads to the construction of the second branch of the tree, (*c*:1), (*b*:1), (*p*:1). (5). For the last transaction, since its frequent item list *f, c, a,m, p*is identical to the first one, the path is shared with the count of each node along the path incremented by 1.

Definition (*FP-tree*). A *frequent-pattern tree* (or *FP- tree* in short) is a tree structure

(1). It consists of one root labeled as "*null*", a set of item-prefix sub trees as the children of the root and a frequent-item-header table.

(2). Each node in the item-prefix sub tree consists of three fields: *item-name*, *count*, and *node-link*, where *item-name* registers which item this node represents, *count* registers the number of transactions represented by the portion of the path reaching this node,