Mining Incremental Association Rules with Generalized FP-tree

C.I. Ezeife
School of Computer Science University of Windsor, Windsor, Ontario, Canada N9B 3P4
cezeife@uwindsor.ca
http://www.cs.uwindsor.ca/users/c/cezeife

Yue Su
School of Computer Science, University of Windsor

Abstract

New transaction insertions and old transaction deletions may lead to previously generated association rules no longer being interesting, and new interesting association rules may also appear.

Existing association rules maintenance algorithms are Apriori-like, which mostly need to scan the entire database several times in order to update the previously computed frequent or large itemsets, and in particular, when some previous small itemsets become large in the updated database.

This paper presents two new algorithms that use the frequent patterns tree (FP-tree) structure to reduce the required number of database scans. One proposed algorithm is the DB-tree algorithm, which stores all the database information in an FP-tree structure and requires no re-scan of the original database for all update cases. The second algorithm is the PotFP-tree (Potential frequent pattern) algorithm, which uses a prediction of future possible frequent itemsets to reduce the number of times the original database needs to be scanned when previous small itemsets become large after database update.

Keywords: Incremental maintenance, association rules mining, FP-tree structure

* This research was supported by the Natural Science and Engineering Research Council (NSERC) of Canada under an operating grant (OGP-0194134) and a University of Windsor grant.
1 Introduction

Databases and in particular, data warehouses contain large amounts of data that grow or shrink over time due to insertion, deletion and modification of transactions (records) of the database over a period of time.

Association rule mining is a data mining technique which discovers strong associations or correlation relationships among data. Given a set of transactions (similar to database records in this context), where each transaction consists of items (or attributes), an association rule is an implication of the form $X \rightarrow Y$, where $X$ and $Y$ are sets of items and $X \cap Y = \emptyset$. The support of this rule is defined as the percentage of transactions that contain the set $X$, while its confidence is the percentage of these “$X$” transactions that also contain $Y$. In association rule mining, all items with support higher than a specified minimum support are called large or frequent itemsets. An itemset $X$ is called an $i$-itemset if it contains $i$ items.

Agrawal et al. [1] presents the concept of association rule mining and an example of a simple rule is “80% of customers who purchase milk and bread also buy eggs”. Since discovering all such rules may help market baskets or cross-sales analysis, decision making, and business management, algorithms presented in this research area include [1, 7, 6]. These algorithms mainly focus on how to efficiently generate frequent patterns and how to discover the most interesting rules from the generated frequent patterns.

However, when the database is updated, the discovered association rules may change. Some old rules may no longer be interesting, while new rules may emerge. Four types of scenarios may arise with the generated frequent itemsets (frequent patterns) when an original database is updated: (1) frequent itemsets in the original database ($F$) remain frequent in the newly updated database ($F'$), (2) frequent itemsets in the original database ($F$) now become small itemsets in the new database ($S'$), (3) small itemsets in the original database ($S$) remain small in the new database ($S'$), and (4) small itemsets in the original database ($S$) now become frequent in the new database ($F'$). The symbols $F$ and $F'$ stand for frequent patterns in the original and new databases respectively, while $S$ and $S'$ stand for small itemsets in the original and new databases respectively. The four types of changes that may occur with frequent patterns following an update in the database can be summarized as: (1) $F \rightarrow F'$, (2) $F \rightarrow S'$, (3) $S \rightarrow S'$, (4) $S \rightarrow F'$.

Incremental maintenance of association rules involves a technique that uses mostly only the updated part of the database, not the entire
new database (consisting of the original data plus the updated part) to
maintain association rules. Existing work on incremental maintenance of
association rules with better performance than the Apriori algorithm [1]
include [3, 2, 8]. These incremental maintenance algorithms are still
Apriori-like and long lists of candidate itemsets are generated at each
level, while either the updated part of the new database (for $F \rightarrow F'$, $F
\rightarrow S'$, and $S \rightarrow S'$) or the entire new database (for $S \rightarrow F'$) has to be
re-scanned to obtain their supports.

Han et al. [5] proposes a tree structure for mining frequent patterns
without candidate generation, but applying this technique directly to the
problem of incremental maintenance of association rules does not produce
optimal results as the new database needs to be re-scanned for the $S \rightarrow
F'$ case.

In this paper, we present two new algorithms (DB-tree and PotFP-
tree algorithms) for efficiently mining association rules in updated database
using the FP-tree structure. These algorithms aim at removing the need
to re-scan the entire new database and re-construct the FP-tree when the
case $S \rightarrow F'$ arises. The algorithms work by only scanning the updated
parts of the database once (at most twice) without generating candidate
sets like previous incremental association rules algorithms. Thus, the
proposed algorithms achieve good performance.

1.1 Related Work

The problem of mining association rules is decomposed into two subpro-
blems, namely (1) generating all frequent itemsets in the database and (2)
generating association rules in the database according to the frequent
itemsets generated in the first step. Apriori algorithm [1] is designed for
generating association rules. The basic idea of this algorithm is to find
all the frequent itemsets iteratively. In the first iteration, it finds the
frequent 1-itemsets $L_1$ (each frequent 1-itemset contains only one item).

To obtain $L_1$, it first generates candidate set $C_1$ which contains all 1-
itemsets of basket data, then the database is scanned for each itemset in
the set $C_1$ to compute its support. The items with support greater than
or equal to minimum support (minsup) are chosen as frequent items
$L_1$. The minsup is provided by the user before mining. In the next
iteration, apriori_gen function [1] is used to generate candidate set $C_2$ by
joining $L_1$ with itself, $L_1$ and keeping all unique itemsets with 2 items
in each. The frequent itemsets $L_2$ is again computed from the set $C_2$ by
selecting items that meet the minsup requirement. The iterations go
on by applying apriori_gen function until $L_i$ or $C_i$ is empty. Finally, the
frequent itemsets $L$ is obtained as the union of all $L_1$ to $L_{i-1}$. FUP2 algorithm is proposed by [3] to address the incremental maintenance problem for association rule mining. This algorithm utilizes the idea of Apriori algorithm, to find the frequent itemsets iteratively by scanning mostly only the updated part of new database. It scans the entire new database when small itemsets become large. The MAAP algorithm [8] computes the frequent patterns from the higher level reusing the old frequent patterns and reducing the need to generate long lists of lower level candidate sets, thus, performing better than the FUP and the FUP2 techniques, but is still level wise.

In [4, 5], a compact data structure called frequent pattern tree (FP-tree) is proposed. The FP-tree stores all the frequent patterns on the tree before mining the frequent patterns using the FP-tree algorithm. Constructing the FP-tree requires two database scans (one scan for constructing and ordering the frequent patterns and the second for building branches of the tree). The FP-tree algorithm brings about better performance than the Apriori-like algorithms due to much reduced database scan. However, applying this technique directly for incremental maintenance of frequent patterns still requires the usual maximum of two database scans on the entire new database when previous small items become large.

1.2 Contributions

This paper contributes by proposing two new algorithms, DB-tree and Potential Frequent Pattern tree (PotFP-tree) algorithms, for efficiently mining incremental association rules in updated database. The proposed algorithms store frequent patterns on a more generalized FP-tree, which stores tree branches for all items in the database (in the case of DB-tree) and stores for all items that are frequent or are potentially frequent in the near future (in the case of PotFP-tree). The proposed algorithms eliminate the need to scan the old database in order to update the FP-tree structure when previously small itemsets become large in the new database.

1.3 Outline of the Paper

The organization of the rest of the paper is shown as follows: section 2 presents an example mining of a database and its update using the basic FP-tree algorithm; section 3 presents formal details of the proposed algorithms with examples; section 4 presents performance analysis of the
Table 1: The Example Database Transaction Table with Ordered Frequent Items

<table>
<thead>
<tr>
<th>TID</th>
<th>Items Bought</th>
<th>Ordered Frequent Items</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>{f, a, c, d, g, i, m, p}</td>
<td>{f, c, a, m, p}</td>
</tr>
<tr>
<td>200</td>
<td>{a, b, c, f, l, m, o}</td>
<td>{f, c, a, b, m}</td>
</tr>
<tr>
<td>300</td>
<td>{b, f, h, j, o}</td>
<td>{f, b}</td>
</tr>
<tr>
<td>400</td>
<td>{b, c, k, s, p}</td>
<td>{c, b, p}</td>
</tr>
<tr>
<td>500</td>
<td>{a, f, c, e, l, p, m, n}</td>
<td>{f, c, a, m, p}</td>
</tr>
</tbody>
</table>

algorithm; and finally, section 5 presents conclusions and future work.

2 Mining An Example Database with FP-Tree

The DB-tree and PotFP-tree algorithms being proposed in this paper are more generalized forms of the FP-tree algorithm, and we first apply the FP-tree algorithm to an example in section 2.1 before applying FP-tree to an updated database in section 2.2.

2.1 FP-tree Algorithm on a Sample Database

Suppose we have a database DB with set of items, \( I = \{ a, b, c, d, e, f, g, h, i, j, k, l, m, n, o, p \} \) and MinSupport=60% of DB transactions. A simple database transaction table for illustrating the idea is given in the first two columns of Table 1.

To compute the frequent itemsets in the old database (Table 1), FP-tree algorithm first constructs the FP-tree from the original database to obtain the extended prefix-tree structure (Figure 1), which stores all the frequent patterns in each record of the database as shown in Figure 1. Secondly, the algorithm mines or computes the frequent itemsets in the database by mining the frequent patterns on the FP-tree that satisfy minimum support.

To construct the FP-tree, the original DB (first two columns of Table 1) scans the database once to obtain and present in descending order of support all the frequent 1-itemsets \( L_1 \) (with support more than or equal to 3 records) as \( < (f : 4), (c : 4), (a : 3), (b : 3), (m : 3), (p : 3) > \). The number accompanying each item is its support in number of records of the DB it occurs. Using this sorted frequent pattern obtained after the first scan of the database, then, each record of the database is again scanned
to obtain the third column of Table 1 showing the frequent items that are present in each transaction in descending order of frequency. Starting from the first transaction, once the frequent patterns in a transaction is obtained, they are inserted in the appropriate branch of the FP-tree. For example, since the first transaction (100) has frequent items (f,c,a,m,p), a branch of the tree is constructed from the root with f as its immediate child, c is the child of f and a is the child of c and so on. Since these tree nodes have occurred only once so far, a count of 1 is recorded for each node. Reading the second transaction (200) with frequent items (f,c,a,b,m) causes the previous nodes f, c, and a of the first branch to have a count of 2 each. However, a child node of a is now b with a count of 1, while a child node of b is m with a count of 1. To handle the frequent items (f, b) of transaction 300, a branch b with a count of 1 is created from node f, which now has a count of 3. For the frequent item (c,b,p) of transaction 400, since there is no current branch on the tree to share path with, a branch c with count 1 is created from the root. Finally, the frequent items for transaction 500 (f,c,a,m,p) are inserted on the first branch of the tree and the counts of the nodes are incremented to give the final FP-tree shown in Figure 1.

Figure 1: The FP-tree of the Example Database

To mine the frequent patterns from the constructed FP-tree, starting from the lowest frequent item on the item header list (pointing to the nodes of the tree containing this frequent item), it derives the frequency of this item and continues to define all paths (branches) that occurred with this item and their counts. These paths make the item’s conditional pattern base. The frequent patterns derived from these pattern bases consist of only those prefix items (before the item) which make a combined support of more or less than the minimum support. For example, from the FP-tree of Figure 1, the lowest frequent item is p and following the header pointer, a frequent pattern (p:3) can be derived from the two
nodes that it appeared. The two p nodes are from the conditional pattern bases \(< (f : 2, c : 2, a : 2, m : 2) >\) and \(< (c : 1, b : 1) >\). Since only the prefix c has a combined support of 3 from these conditional pattern bases, it is the only prefix that can combine with p to form the frequent pattern cp:3. This concludes the search for frequent patterns of p. For items m, the conditional pattern bases are \(< (f : 2, c : 2, a : 2) >\) and \(< (f : 1, c : 1, a : 1, b : 1) >\). Only the items in the path before the considered frequent item are considered in the analysis. The frequent pattern of m that can be derived from these bases is \(< f : 3, c : 3, a : 3 ) >\). Thus, the derived frequent pattern is fcam:3. Obtaining fcam as a frequent item is equivalent to having all of its subsets confirmed frequent. Thus, \{ (m:3), (am:3), (cm:3), (fm:3), (cam:3), (fam:3), (fcam:3), (fcm:3) \} are all frequent. The conditional pattern base and conditional FP-tree of other items can be obtained in a similar way to give Table 2. Thus, the result of the original mining with FP-tree of the original DB is the set of maximal frequent patterns \{ cp \}, \{ fcam \} and \{ b \}. All the subpatterns of these frequent patterns are also frequent.

### 2.2 Mining Updated Database With FP-tree

If we update the DB such that transactions 600 \( (p,f,m,g,o,l) \), 700 \( (q,f,b,p,l,w) \), 800 \( (f,c,b,e,m,o) \), 900 \( (f,b,p,m,a,l) \) and 1000 \( (f,t,a,b,p,l,o) \) are added, we can get an updated database DB', which is suitable for applying the FP-tree algorithm. The number of items in the DB has also changed to include the items \{ q, t, w \}.

Since many of the incremental maintenance algorithms FUP [2], FUP2 [3], and MAAP [8] aim at improving performance by updating the generated frequent itemsets without scanning the entire database but
only the updated part, this section discusses how the FP-tree algorithm can be used to update the frequent itemsets without the need to re-scan the entire database.

With the above update (insertions) made to the example database of Table 1, scanning only the updated part of the database reveals that the frequent patterns from only the updated part of the database are (p:4, l:4, m:3, o:3), while the previous frequent patterns of the old database on which the original FP-tree is constructed are (f:4, c:4, a:3, b:3, m:3, p:3). Without the need to re-scan the entire database consisting of the old database and the updated part, the updated frequent patterns in descending order can be obtained from the two FP lists above as: (p:7, m:6, f:4, c:4, l:4, a:3, b:3, o:3). It can be seen that the order of frequency has changed for some items like p (going from the smallest frequent item to the highest frequent item), and m (moving from the second smallest frequent item to the second highest frequent item). Some previously unfrequent items like l and o are now frequent. With these kinds of changes to the database, the only way to update the FP-tree is to re-scan the entire database at least one more time to re-construct the tree since the correct mining is based on the order. Secondly, some transactions with previously low items like l and o in the old database have to be scanned to include these new frequent items in their tree branches.

The cost of re-scaning and re-constructing the FP-tree when thousands of items are in the database and millions of rows or transactions are in the database could be quite significant. Thus, this paper proposes two ways to eliminate the need to re-scan the entire database when the database is updated either through insertions, deletions or modifications of transactions.

3 The Proposed Incremental Generalized FP-Tree Algorithms

This section presents two algorithms being proposed for mining frequent itemsets incrementally using a more generalized FP-tree structure. Section 3.1 discusses the algorithm DB-Tree, while section 3.2 presents the PotFP-Tree, standing for potential FP-Tree algorithm. The PotFP-Tree algorithm keeps information about some previously unfrequent items which are predicted to have a high potential for being frequent soon in the future, and the DB-tree keeps the entire database on a generalized FP-tree with minimum support of 0 or 1.
3.1 Mining Incremental Rules With DB-Tree

DB-Tree is a generalized form of FP-tree which stores in descending order of support all items in the database, as well as counts of all items in all transactions in the database in its branches. The DB-tree is constructed the same way the FP-tree is constructed except that it includes all the items instead of only the frequent 1-itemsets. Thus, like the FP-tree, it takes two database scans to construct the DB-tree. DB-tree has more branches and more nodes than the FP-tree and thus needs larger storage than the FP-tree. However, the DB-tree is still much smaller than the database since items share paths in the tree structure. A DB-tree can be seen as an FP-tree with a minimum support of 0. This means that the DB-tree contains an FP-tree on top. At any point in time, the desired FP-tree could be projected from the DB-tree based on the minimum support. The lower the minimum support of a desired FP-tree, the closer to a DB-tree it is. The DB-tree of the database in Table 1 is given as Figure 2.

In Figure 2, all patterns of the database are included and solid circles indicate frequent items while dotted circles indicate small items. On top of the DB-tree is the FP-tree. Mining the frequent patterns from DB-tree requires first projecting the FP-tree from the DB-tree and mining
Figure 3: DB-Tree after Inserted Transactions

the patterns from the FP-tree.

Suppose two transactions, 600 (a, c, f, m, g, o, l) and 700 (f, b, a, c, l, m, o, n) are inserted into the database of Table 1, and we want to update the frequent itemsets with a support of 60%, this update will not cause any database scan using the DB-tree algorithm, while with the previous algorithms like FUP, FUP2 and MAAP, the original database will need to be scanned several times. The original FP-tree algorithm also needs to scan the original database once to update the FP-tree since some items like (l and m) that were not previously large have now become large. To handle this update, using the DB-tree algorithm, would require just scanning the two transactions to update the DB-tree. The occurrences after the two transaction insertions are: (f:6), (c:6), (a:5), (b:4), (m:5), (l:4), (o:4), (p:3), (g:2), (n:2), (j:1), (k:1), (s:1), (e:1), (n:1). The DB-tree after the insertion is given as Figure 3.

The minimum occurrence in the updated database is 4 for frequent itemsets. It can be seen that item p is no longer large in the new database. While it is not necessary to change the order when only frequent patterns order change and are out of order, it is necessary to re-order when either the frequent items become small or small items become frequent.

To project FP-tree from the DB-tree, we start from the root of the DB-tree and extract each branch where a next node is still large. The cost of projecting an FP-tree is equal to the cost of traversing FP-tree
Figure 4: FP-tree of the Updated DB projected from DB-tree

once. The projected FP-tree from the updated DB-tree of Figure 3 is given as Figure 4.

3.2 Mining Incremental Rules With PotFP-Tree Algorithm

The PotFP-tree adopts a more relaxed principle for picking items to store on the tree than both the FP-tree (storing only frequent items) and the DB-tree (storing all items). Thus, the PotFP-tree is based on a principle that is in-between the two extremes. The PotFP-tree also stores items that are not frequent at present but have high probability of being frequent after the database update. Updating the database would entail following the PotFP-tree to update each node's count (for deletion or insertion), or adding new branches into the PotFP-tree (some insertions).

The small items in the original database can be divided into two groups, namely, (1) those that are not large now but may be large after the database update (called the potentially frequent items, P), and (2) those that are small now and with high possibility of still being small after update of the database, M. We can give a tolerance $\ell$ when constructing and re-constructing the FP-tree, which is equivalent to the watermark, in [5]. Watermark is defined as the minimum support that most mining processes are based on over a period of time. For example, if over a year, 60% of all mining processes were based on minimum support of $\geq 20$, then 20 is the watermark, meaning that if more than 20 transaction update occurs, the FP-tree needs to be re-constructed. Thus, the idea here is to use a tolerance $\ell$ that is slightly lower than the support that most mining process (average min-support) have been based on recently. Keeping small items with support less than average min-support, but greater than
t, will benefit the incremental mining process.

Database items with support s, where \( t \leq s \leq \text{averageminsupport} \) are the potentially frequent items which are not part of the current FP-tree, but included in the PotFP-tree structure for purposes of eliminating the need to scan the entire old database in order to update the FP-tree when updates occur in the database. Like the DB-tree, the FP-tree sits on top in the PotFP-tree while the patterns involving potentially frequent items are near the leaves of the tree. The advantage of the PotFP-tree over the FP-tree is that if database update causes all potentially frequent items in group P to become large after database update, the Pot-FP tree algorithm would not require a scan of the original database. However, if some potentially frequent items in group M become large, it will need to scan the original database like the FP-tree. This algorithm has reduced the number of times the entire database needs to be scanned due to updates. How much is gained in response time due to non-scanning of the entire database depends on the choice of \( t \). An experiment in the next section is used to examine what would constitute reasonable values for \( t \).

4 Experimental and Performance Analysis

A performance comparison of DB-tree, PotFP-tree with original FP-tree and Apriori algorithms was conducted and the results of the experiments are presented in this section. All these four algorithms were implemented and run on the same datasets generated using the resource code [1] for generating synthetic datasets downloaded from http://www.almaden.ibm.com/cs/quest/syndata.html. The correctness of the implementations were confirmed by checking that the frequent itemsets generated for the same dataset by the four algorithms are the same. The experiments were conducted on a 733 MHz P3 PC machine with 256 megabytes of main memory running Linux operating system. The programs were written in C++.

The transactions in the dataset mimic the transactions in a retail environment. The result of two experiments are reported as follows.

- Experiment 1: Given a fixed size dataset (inserted and deleted parts of the dataset are also fixed), we test CPU execution time at different thresholds of support to compare DB-tree, PotFP-tree, FP-tree and Apriori algorithms. The aim of this experiment is to show that performance of PotFP-tree algorithm is better than that of FP and Apriori algorithms at different levels of support using the same dataset size. The number of transactions (D) in this dataset
Table 3: Execution Times for Dataset at Different Supports

<table>
<thead>
<tr>
<th>Algorithms</th>
<th>CPU Time (in secs) at Supports of</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0.1</td>
</tr>
<tr>
<td>Apriori</td>
<td>-</td>
</tr>
<tr>
<td>FP-tree</td>
<td>54</td>
</tr>
<tr>
<td>DB-tree</td>
<td>44</td>
</tr>
<tr>
<td>PotFP-tree</td>
<td>42</td>
</tr>
</tbody>
</table>

is one hundred thousand records, that is $|D| = 100,000$ records, the average size of transactions (number of items in transactions) $(T)$ is 10, $|T| = 10$, average length of maximal pattern (that is, average number of items in the longest frequent itemsets) $(I)$ is 6, or $|I| = 6$, number of items $(N)$ (the total number of attributes) is one thousand, $N=1000$. Assume the size of updated (inserted) dataset is 10,000 records, the size of updated (deleted) dataset is 10,000 records (these parameters are abbreviated as T10.16.D100K-10K+10K with 1000 items, the support thresholds are varied between 0.1% and 6%, meaning that for a support level of 0.1%, an itemset has to appear in 100 (one hundred) or more transactions to be taken as a frequent itemset, while with a support of 6%, an itemset has to appear in 6000 (six thousand transactions to be large). An experimental result is shown in Table 3, while its graphical representation is given in Figure 5.

From the observation of the experimental result, we can see that (i) as the size of the support increases, the execution time of all the algorithms decreases. (ii) for the same support, the execution time of PotFP-tree algorithm is less than that of FP-tree and Apriori algorithms. (iii) as the size of support increases, the difference in execution times of PotFP-tree algorithm and FP-tree diminishes. In this experiment, DB-tree only shows a little advantage over the FP-tree when the minimum support is very small (less than 0.5%).

- Experiment 2: Given a fixed size dataset (including inserted and deleted datasets) and a fixed support, we test CPU execution times when different numbers of old frequent itemsets are allowed to change in the new database. Since the number of frequent itemsets changed may affect CPU time of PotFP-tree algorithm, this experiment is conducted to observe the performance of both PotFP-tree
Figure 5: Execution Times At Different Support Levels

Table 4: Execution Times at Different Transaction Sizes on Support 0.5%

<table>
<thead>
<tr>
<th>Algorithms (times in secs)</th>
<th>Different Changed Transaction Size</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>10K</td>
</tr>
<tr>
<td>FP-tree</td>
<td></td>
</tr>
<tr>
<td>DB-tree</td>
<td></td>
</tr>
<tr>
<td>PotFP-tree</td>
<td></td>
</tr>
</tbody>
</table>

An experiment on what would constitute a reasonable tolerance
Figure 6: Execution Times at Different Sizes of Changed Transactions

shows that best result is achieved with a tolerance value that is equal to 90% of the minimum support and performs worst with a tolerance value equal to 50% of the minimum support. Thus, we can deduce that the closer the tolerance is to the minimum support, the better the performance. However, a tolerance value that is too close to the minimum support loses the advantage gained by using the PotFP-tree.

5 Conclusions and Future Work

This paper presents two new algorithms DB-tree and PotFP-tree algorithms, for incrementally maintaining association rules in the updated database. These algorithms are based on a generalized FP-tree structure that store more items on the tree than only those that are frequent. The contribution of these algorithms is better response time and in particular when minimum support is low.

Future work should include looking for a theoretical method to decide the most beneficial tolerance value \( t \) for the PotFP-tree scheme and consider using a partitioned version of the DB-tree to improve on its performance. Application of this method and incremental mining approaches in general, to web usage mining should be investigated.
References


[5] Han, J., Pei, J., Yin, Y.: Mining Frequent Patterns without Candidate Generation, ACM SIGMOD 2000, Dallas, TX, U.S.A.

