

# Mining Frequent Patterns with the Pattern Tree\*

Hao Huang <sup>(+)</sup>, Xindong Wu <sup>(\*)</sup> and Richard Relue <sup>(.)</sup>

<sup>(+)</sup> Department of Computer Science  
University of Virginia  
Charlottesville, Virginia 22904, USA

<sup>(\*)</sup> Department of Computer Science  
University of Vermont  
Burlington, Vermont 05405, USA

<sup>(.)</sup> Department of Mathematical and Computer Sciences  
Colorado School of Mines  
Golden, Colorado 80401, USA

## Abstract

Mining frequent patterns with a frequent pattern tree (FP-tree in short) avoids costly candidate generation and repeatedly occurrence frequency checking against the support threshold. It therefore achieves much 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 is added to an existing database because two scans may be needed for not only the new data but also the existing data. In this research we propose a new data structure, the pattern tree (P-tree in short), 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 does not need to be re-scanned. Our experiments show that the P-tree method outperforms the FP-tree method by an factor up to an order of magnitude in large datasets.

## 1 Introduction

Data mining includes the activities of association analysis, clustering and classification. In this research, we focus on association rule mining. A canonical

---

\*A preliminary version of this paper has been published in the *Proceedings of the 2002 IEEE International Conference on Data Mining (ICDM '02)*, 629-632.

example is market basket analysis. A transaction consists of the items purchased in a market basket during a transaction. They could happen by a customer at a specific time point, or over a period of time. By finding the associations among thousands and thousands of transactions, the retailers could obtain and further make use of customer buying habits. An association rule is an implication of the form  $X \implies Y$ , 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  $X \cup Y$ ; and 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 [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12] was based on Apriori-like algorithms. They can be decomposed into two steps:

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

This kind of algorithms iteratively obtains 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, a frequent pattern tree based frequent pattern mining method [13, 14] developed by Han et al achieves high efficiency, compared with Apriori and TreeProjection [15] algorithms. It avoids iterative candidate generations.

The rest of the paper is organized as follows. We first describe the FP-tree data structure by an example 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. We deal with updating the P-Tree with new data in Section 5. Section 6 presents some further discussions about the P-tree data structure and its implementation and Section 7 contains the results of the tests we have performed. Finally, we will give a brief summary in Section 8.

Table 1: A Transaction Database

<i>TID</i>	<i>Transaction</i>	<i>Items in Frequency Descending Order</i>	<i>Frequent Items</i>
100	$f, a, c, d, g, i, m, p$	$f, c, a, m, p, d, g, i$	$f, c, a, m, p$
200	$a, b, c, f, l, m, o$	$f, c, a, b, m, l, o$	$f, c, a, b, m$
300	$b, f, h, j, o$	$f, b, h, j, o$	$f, b$
400	$b, c, k, s, p$	$c, b, p, k, s$	$c, b, p$
500	$a, f, c, e, l, p, m, n$	$f, c, a, m, p, e, l, n$	$f, c, a, m, p$

## 2 Frequent Pattern Mining and the Frequent Pattern Tree

The frequent pattern mining problem can be formally defined as follows. Let  $I = \{i_1, i_2, \dots, i_n\}$  be a set of items, and  $D$  be a transactions database, where each transaction  $T$  is a set of items and  $T \subseteq I$ . A 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 or percentage of transactions containing  $P$  in  $D$ . We say that  $P$  is a frequent pattern if the support of  $P$  is no less than a predefined minimum support threshold  $\xi$ .

In [13], frequent pattern mining consists of two steps:

1. Construct a frequent pattern tree, which can store more information in less space.
2. Develop an FP-tree based pattern growth method, FP-growth. We also adopt this method for further pattern mining after P-tree or FP-tree construction.

A frequent pattern tree is a prefix-tree structure storing frequent patterns for the transaction database, where the support of an item represented on each tree node is no less than the support threshold  $\xi$ . 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 items 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” [13]. For example, with five transactions in Table 1, we get an FP-tree in Figure 1.

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

Table 2: Mining the FP-tree in Figure 1

<i>Item</i>	<i>Conditional Paths</i>	<i>Conditional FP-tree</i>	<i>Frequent patterns</i>
p	$\langle (f : 2), (c : 2), (a : 2), (m : 2) \rangle$ $\langle (c : 1), (b : 1) \rangle$	$\langle (c : 3) \rangle$	$(c p : 3)$
m	$\langle (f : 2), (c : 2), (a : 2) \rangle$ $\langle (f : 1), (c : 1), (a : 1), (b : 1) \rangle$	$\langle (f : 3), (c : 3) \rangle$ $(a : 3) \rangle$	$(f c a m : 3)$ $(f c m : 3)$ $(f a m : 3)$ $(c a m : 3)$ $(f m : 3)$ $(c m : 3)$ $(a m : 3)$
b	$\langle (f : 1), (c : 1), (a : 1) \rangle$ $\langle (f : 1) \rangle, \langle (c : 1) \rangle$		
a	$\langle (f : 3), (c : 3) \rangle$	$\langle (f : 3), (c : 3) \rangle$	$(f c a : 3)$ $(f a : 3)$ $(c a : 3)$
c	$\langle (f : 3) \rangle$	$\langle (f : 3) \rangle$	$(f c : 3)$

1. Derive a set of conditional paths, which co-occurs with a prefix 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.

With the FP-tree in Figure 1, Table 2 shows the mining process with the FP-growth algorithm. The notation (*Item* : *Frequency*) represents the frequency of the item. For example, ( $f : 2$ ) indicates that  $f$  has a frequency of 2.

One can notice that there are two data reductions: the FP-tree is a projection of the entire database, and with the FP-tree the FP-growth algorithm again reduces the data through the whole process and thus the runtime as well. The study in [13] shows that the FP-growth algorithm is much more efficient and scalable than both Apriori and TreeProjection. Since we can dynamically generate an FP-tree from our pattern tree, which we will address later, the same algorithm also can be applied at pattern generation stage after the transformation of an FP-tree from our pattern tree. In other words, we can achieve better performance when we combine the advantages of both the pattern tree and FP-growth.

From the above discussions, the FP-tree based algorithm has some inherent advantages: the new data structure is desirably compact and the FP-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. The first scan gets a static list of frequent items, and another scan of the whole database is needed to construct an FP-tree.
2. Although a validity support threshold, watermark [13], is realizable, there is no guarantee of complete database information for the FP-tree when new data comes into the database.
3. If the specific threshold is reduced, one has 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. So it makes no use of the previous mining processes.

After some careful analysis, we have found that modifying the FP-tree construction makes a new data structure and a corresponding algorithm possible:

1. The FP-tree can contain all information of each transaction in a compact structure, so we can avoid repeatedly scanning the original database, not only in the stage of getting the frequency of each item but also of obtaining a frequency-sorted FP-tree. All we need to do is to restructure the FP-tree according to the frequency of each item.
2. More importantly, since multiple transactions in a database may share a common prefix, the cost of scanning the FP-tree and reconstructing a new one can be much less than scanning the original database the second time.

### 3 Patterns Generation with the Pattern Tree

The FP-tree based frequent pattern mining 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, unlike a frequent pattern 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 prefix in transactions, thereby constructing a compact tree structure. The construction and restructuring of the P-tree will be efficient because the tree algorithms in most programming languages are mature, and in most cases the P-tree can fit in main memory or virtual memory. Note that the P-tree can be bigger than the FP-tree and thus consumes more memory than the FP-tree. However, the FP-tree also needs a lot of memory too when dealing with a large database. We will address this implementation issue in Section 6.2 when it is impossible to implement a main memory based P-tree when the database is very large.

### 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  $[t \mid T_i]$

**ENDFOR**

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

```

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

```

Step 2: Restructure the initial P-tree *P*

- (1) *newP* ← *Root*
  - (2) **FOR** each path *p*<sub>*i*</sub> from the root to a leaf in the initial P-tree *P*
    - WHILE** *p*<sub>*i*</sub> ≠  $\phi$  **DO**
      - a. The common support of each item in *p*<sub>*i*</sub> is that of the node next to the last branching-node. If there is no branching-node in *p*<sub>*i*</sub>, the common support of each item is the actual support of each item in *p*<sub>*i*</sub>.  
 A branching-node is a node after which there exists more than one branch in the tree, such as items *a* and *c* in the path  $\langle (a : 3), (c : 2), (d : 1), (f : 1), (g : 1), (i : 1), (m : 1), (p : 1) \rangle$  in Figure 2. The last branching-node in this case is item *c*, and the common support is that of *d* next to *c* which is 1.
      - b. Obtain a sub-path *p'*<sub>*i*</sub> from *p*<sub>*i*</sub> with the common support for every item.
      - c. Sort *p'*<sub>*i*</sub> according to *L*.
      - d. Insert the sorted *p'*<sub>*i*</sub> into the new P-tree, by calling function *Insert*(*p'*<sub>*i*</sub>, *newP*).
      - e. *p*<sub>*i*</sub> ← *p*<sub>*i*</sub> - *p'*<sub>*i*</sub>
- ENDFOR**

### 3.2 Example 1: Constructing a P-tree from Table 1

First, initialize a tree  $P$  with a “Root” node and an empty list  $L$ . Then insert every sorted transaction into  $P$  and meanwhile update  $L$ . In our example,  $P$  is shown in Figure 2 and  $L$  is  $\langle (f : 4), (c : 4), (a : 3), (b : 3), (m : 3), (p : 3), (l : 2), (o : 2), (d : 1), (e : 1), (g : 1), (h : 1), (i : 1), (j : 1), (k : 1), (n : 1), (s : 1) \rangle$  after each transaction is processed.

Second, initialize a new tree  $newP$  similarly with a “Root” node. For the first path  $p'$  in Figure 2,  $\langle (a : 3), (c : 2), (d : 1), (f : 1), (g : 1), (i : 1), (m : 1), (p : 1) \rangle$ , we can notice that its common support is 1 and thus its subpath is  $\langle (a : 1), (c : 1), (d : 1), (f : 1), (g : 1), (i : 1), (m : 1), (p : 1) \rangle$ . According to  $L$ , we sort  $p'$  as  $\langle (f : 1), (c : 1), (a : 1), (m : 1), (p : 1), (d : 1), (g : 1), (i : 1) \rangle$  and then call function  $Insert(p', newP)$ . In the end of this iteration, we prune  $p'$  from the P-tree in Step 2, 2.e. For the second path  $\langle (a : 2), (c : 1), (e : 1), (f : 1), (l : 1), (m : 1), (n : 1), (p : 1) \rangle$ , its sorted subpath is  $\langle (f : 1), (c : 1), (a : 1), (m : 1), (p : 1), (l : 1), (e : 1), (n : 1) \rangle$ . After the same function  $Insert(p', newP)$  is performed, we need to subtract it from the P-tree. Finally the new P-tree  $newP$  is shown in Figure 3 after all five paths in Figure 2 have been inserted. If we cut off all nodes that do not have the minimum support 3, the P-tree will become exactly the same as the FP-tree in Figure 1. A formal algorithm will be discussed in Section 4.

### 3.3 Analysis

From the above P-tree construction steps, we need 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 highly 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 highly frequent patterns in the database, the slower the algorithm will be. The upper bound is the runtime of two database scans. In this subsection, we shall informally investigate how efficiently the P-tree can be constructed under the assumptions of how highly frequent patterns appear.

#### 3.3.1 The Worst Case

The worst case happens when every two transactions in the database share no prefix with each other, that is to say, every two transactions are totally different. Hence, the pattern tree data structure doesn't make any reduction from the database. It is just the same with the original database. In this case, the cost of P-tree construction will be the same as the cost of scanning the database twice, which is  $O(2nm \log m)$  where  $m$  is the maximum length of transactions,  $n$  the number of the transactions in the database, and  $\log m$  is need to sort each transaction. In this case, the FP-tree structure will not be able to compress the original database in any significant way either.

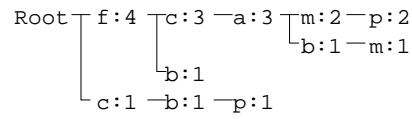


Figure 1: An FP-tree for Table 1

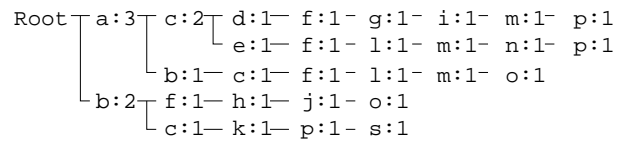


Figure 2: An Initial P-tree for Table 1

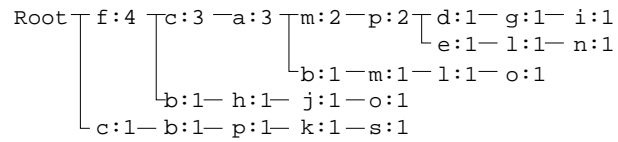


Figure 3: The Reconstructed P-tree for Table 1

### 3.3.2 The Best Case

If all transactions are the same, the first step of our algorithm in Section 3.1 remains unchanged, however, the second step just needs to process one single path. Thus, this best-case pattern distribution makes the algorithm run much faster.

### 3.3.3 Average Case

The worst case is unlikely to happen because there always exist some kinds of associations in the database and the transactions can't and will never be totally different. Thus, the average case runtime of our algorithm in Section 3.1 is better than the worst case to some extent depending on the associations in the database. The more these kinds of associations in the database and the more frequent they are, the more the P-tree data structure and the P-tree generation algorithm can benefit. In this case the time required to restructure the P-tree will be much less than that by scanning the database for the second time. One can notice that the initial P-tree is a compact version of the database, in other words, it contains the original database information with less space. So when restructuring the initial P-tree one processes more information per path. In addition, reading from the original database, which is possibly stored on local or remote disks, can consume more time than from the initial P-tree, which can possibly be stored in main memory due to its compactness.

## 3.4 Pattern Tree: A Formal Definition

A pattern tree 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 represented by the portion of the path from the root to the item, 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. An item represented on 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 the 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 descendants, 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 item frequency list  $L$ , and the support threshold  $\xi$

**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  $FIList$
  - ENDIF**
- ENDFOR**
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**  $c \in FIList$

**THEN**

$check(c)$

**ELSE**

Delete  $c$  (and the possible subtree starting from  $c$ )

**ENDIF**

**ENDFOR**

**RETURN**

**END**

## 4.2 Example 2: Constructing an FP-tree from a P-tree

Firstly, from Example 1 of Section 3 one can easily obtain a sorted frequent item list *FIList* as  $\langle (f : 4), (c : 4), (a : 3), (b : 3), (m : 3), (p : 3) \rangle$  when the minimum support  $\xi = 3$ .

Secondly, one checks every node in each path to check whether it is in the frequent item list *FIList*. For instance, in the path  $\langle (f : 4), (c : 3), (a : 3), (m : 2), (p : 2), (d : 1), (g : 1), (i : 1) \rangle$ , the node *m* and the following subtree (or path) are not in the *FIList*. Therefore only  $\langle (f : 4), (c : 3), (a : 3) \rangle$  is kept. Similarly,  $\langle (f : 4), (b : 1) \rangle$  is left after the node *h* and its subtree are pruned from the path  $\langle (f : 4), (b : 1), (h : 1), (j : 1), (o : 1) \rangle$ .

Finally, the P-tree in Figure 3 will become exactly the same as the FP-tree in Figure 1.

## 4.3 Analysis

In practice, we can compare the user-defined minimum support threshold with the occurrence of each item 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 one half 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 related to 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 the data mining field. For instance, when one searches for a specific title on Amazon.com, it will output the link to the book if it exists as well as the following links: what were also bought by customers who bought this book, and by customers who bought titles by the above author also bought titles by other authors and so forth. The latter would never be static; they are dynamic and updated according to customer buying patterns over a period of time. Therefore, one customer can be informed of the latest information and can easily access the most recent associated titles, and Amazon can thereby sell more books and gain more profit.

In existing research, when new data comes, one possible approach to update the discovered rules is to rerun Apriori algorithms on the whole updated database. The obvious disadvantage of this approach is that all the large item-sets have to be computed again from scratch, therefore all the previous computations are wasted. One alternative approach [16, 17] still requires  $k$  iterations although it reduces the size of the candidate set to be searched against the original large database.

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 [13]. 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.

How can we do that? 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  $T$

**ENDFOR**

- (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  $P1$ 

**WHILE**  $p_i \neq \phi$  **DO**

  - a. Let  $s$  be the common support of each item in  $p_i$
  - b. Obtain 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.  $p_i \leftarrow p_i - p'_i$

**ENDFOR**

## 5.2 Example 3: Updating a P-tree

Let's go through an example below with Algorithm 3.

Suppose the original transaction database,  $DB$ , contains 20 transactions shown in Table 3. During the first scan of the original database, we get the item frequency list  $L$  as follows:  $\langle (a:17), (f:15), (i:15), (b:13), (m:12), (o:12), (c:11), (l:10), (k:8), (h:7), (e:6), (p:6), (j:5), (d:4), (g:4), (n:4), (v:2), (s:1) \rangle$ . The initial P-tree  $P1$  is shown in Figure 4.

Assume that ten transactions in Table 4 are appended to the original database in Table 3. Each new transaction is inserted into  $P1$  according to the old frequency list  $L$  and the updated P-tree is shown in Figure 5. Meanwhile, we update the item frequency list  $L$ :  $\langle (a : 22), (i : 21), (f : 20), (b : 19), (c : 19), (l : 18), (m : 18), (o : 18), (k : 16), (h : 11), (j : 7), (d : 7), (e : 7), (p : 7), (g : 4), (n : 4), (v : 2), (y : 2), (q : 1), (s : 1) \rangle$ . Next, we reconstruct  $P1$  according to  $L$  and get the restructured P-tree  $P2$  in Figure 6 for the updated database.

Table 3: A Database (DB) of 20 Transactions

<i>TID</i>	<i>Transaction</i>	<i>Items in Frequency Descending Order</i>
1	f, a, c, d, g, i, m, p	a, f, i, m, c, p, d, g
2	a, b, c, f, l, m, o	a, f, b, m, o, c, l
3	b, f, h, j, o	f, b, o, h, j
4	b, c, k, s, p	b, c, k, p, s
5	a, f, c, e, l, p, m, n	a, f, m, c, l, e, p, n
6	m, e, j, h, l, a, o	a, m, o, l, h, e, j
7	a, b, k, f, c, l, i	a, f, i, b, c, l, k
8	o, a, i, e, m, h, p, f	a, f, i, m, o, h, e, p
9	k, a, i, j, c, f, b	a, f, i, b, c, k, j
10	f, j, i, l, c, a, o	a, f, i, o, c, l, j
11	a, f, l, n, b, i, e, p	a, f, i, b, l, e, p, n
12	k, i, a, e, l, b	a, i, b, l, k, e
13	a, o, f, i, b, v, m, h, c	a, f, i, b, m, o, c, h, v
14	l, o, c, a, m, k, n, f, i	a, f, i, m, o, c, l, k, n
15	o, b, m, k, c, d, f, j, i	f, i, b, m, o, c, k, j, d
16	a, b, i, g, m, o, c, h	a, i, b, m, o, c, h, g
17	i, o, d, m, f, k, g, h, a	a, f, i, m, o, k, h, d, g
18	m, d, e, l, v, i, a, b	a, i, b, m, l, e, d, v
19	i, a, b, g, l, m, o, f	a, f, i, b, m, o, l, g
20	n, b, h, p, f, i, k, o, a	a, f, i, b, o, k, h, p, n

Table 4: Ten New Transactions

<i>TID</i>	<i>Transactions</i>	<i>Items in Frequency Descending Order</i>
21	j, i, c, l, h, b, f, y, k	f, i, b, c, l, k, h, j, y
22	m, o, a, k, i	a, i, m, o, k
23	k, h, f, i, m, c, b	f, i, b, c, m, k, h
24	o, p, c, m, k, l, i, a	a, i, m, o, c, l, k, p
25	o, b, y, m, d, f, q, j, l	f, b, m, o, l, j, d, q, y
26	a, b, l, i, k, o	a, i, b, o, l, k
27	o, c, d, m, k, h, l	m, o, c, l, k, h, d
28	f, m, d, k, l, c, a	a, f, m, c, l, k, d
29	i, b, o, c, e, l, h	i, b, o, c, l, h, e
30	c, b, a, l, f, k	a, f, b, c, l, k

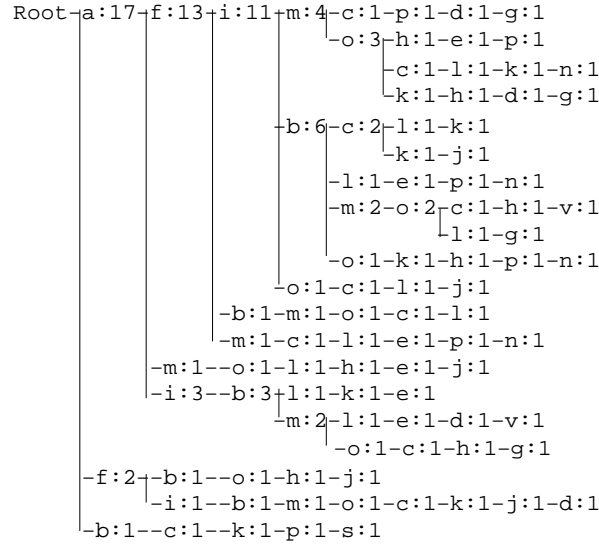


Figure 4: A P-tree for Table 3

### 5.3 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 [13, 14], an FP-tree is obtained by two scans of the entire database, including the existing and new database.
3. In the worst case (see Section 3.3.1), 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.

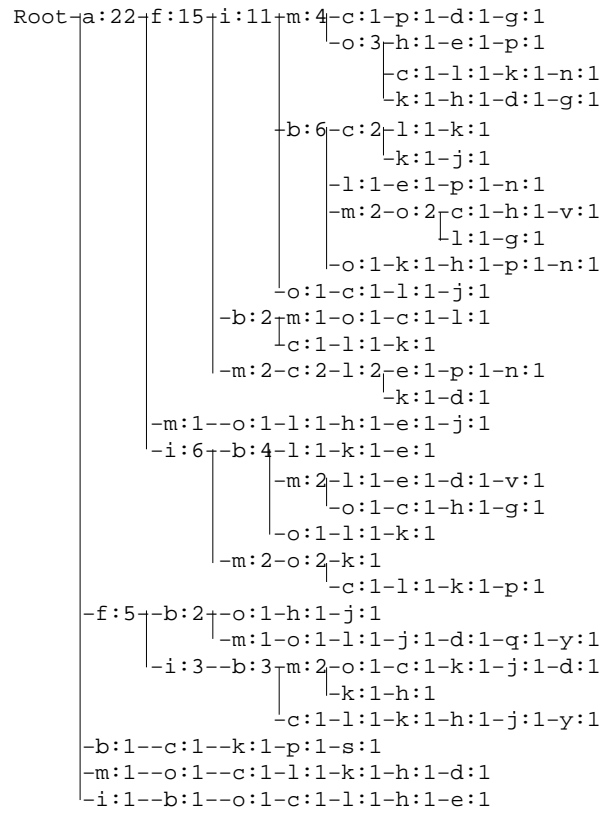


Figure 5: An Updated P-tree



## 6 Discussions

### 6.1 Lemmas

#### 6.1.1 Correspondence between the Pattern Tree and the Database

**Lemma 1** Given a transaction database DB, and a specified item order when some items have the same support in DB, there exists a unique P-tree corresponding to DB.

**Proof.** In each iteration, a transaction in the DB is inserted into the P-tree. Moreover, the items in a path will not have two kinds of sequences because there exists an order if two items in the path happen to have the same support values. Given those observations, the P-tree will contain the complete information of the DB without any redundancy; otherwise a contradiction will occur. If there is a transaction in the DB that is not in the P-tree, the item frequency list would be inconsistent with the DB. On the other hand, a path in the P-tree may represent one single transaction should the database contain no duplication. Even if the database does contain extra information, the item occurrence recorded in the P-tree will represent that. Similarly, the item frequency list would again be inconsistent with the DB if there is a path in the P-tree that does not exist in the DB.

#### 6.1.2 Correspondence between Pre-reconstructed and Post-reconstructed Pattern Trees

**Lemma 2** Given a transaction database DB, its corresponding restructured P-tree contains the same information as the P-tree before the process, i.e. the complete information of DB.

**Proof.** Based on the pattern tree reconstruction process, each path is inserted into the new P-tree and only sorted in a different order. In other words, there is one path in the P-tree corresponding to one path in the restructured P-tree and vice versa. Since we already prove that there is a one-to-one relationship between the P-tree and the database, the lemma holds.

#### 6.1.3 Correspondence between the P-tree and the FP-tree

**Lemma 3** Given a transaction database DB and a specified minimum support threshold, there exists only one unique FP-tree.

**Proof.** It is easy to see that the lemma holds because Algorithm 2 keeps the frequent items in the P-tree and prunes the infrequent items away. Sorting items in frequency descending order also ensures the uniqueness of the tree.

## 6.2 An Implementation Issue

### 6.2.1 Database Partitioning

A database partitioning technique can be useful when the database is huge and the entire pattern tree can not fit in main memory. This method can also be applied if the databases themselves are distributed. It consists of three phases as shown in Figure 7:

1. In Phase I, we divide the transaction database DB into  $n$  non-overlapping partitions given  $n$  available machines. For each partition, a pattern tree is constructed locally. Meanwhile, a local item frequency list is recorded. It is ensured that each partition size is much smaller than the size of the DB so that the pattern tree built upon each partition can fit in main memory.
2. In Phase II, we compute an overall item frequency list from the  $n$  local frequency lists and then restructure  $n$  pattern trees locally.
3. In Phase III, an overall FP-tree is built for further mining based on the user-specified minimum support threshold.

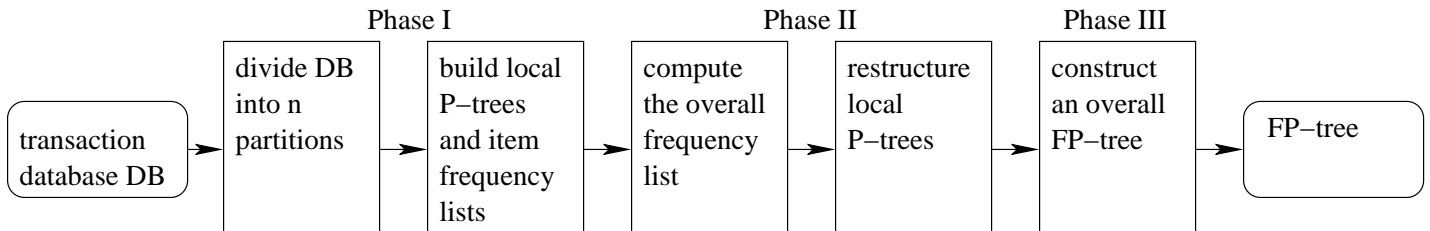


Figure 7: Database Partitioning

## 7 Tests and Results

This section presents our test results of multiple FP-tree generation and FP-tree updating while more data are added.

### 7.1 Test Environment and Test Databases

Our experiments are performed on a 500 MHz Pentium III PC with 640 MB of memory running on Red Hat Linux 6.1. All programs are written in Java. The FP-tree construction algorithm in [13] and [14] is implemented and run on the same machine in order to compare the P-tree approach in this paper with the FP-tree approach in the identical environment. We adopt the same datasets,

Table 5: Datasets

<i>DB</i>	<i>T</i>	<i>I</i>	<i>D</i>	<i>Size(KB)</i>
T25I10D10k	25	10	10k	1,094
T25I20D100k	25	20	100k	13,758

T25I10D10k (denoted *DB1*) and T25I20D100k (*DB2*), which were used in [13] for their experiments. These datasets can be generated using the algorithm in [6]. Their tests already show that the FP-tree method scales much better than Apriori and TreeProjection. Table 5 summarizes the dataset information, where *T* is the transaction size on average, *I* is the maximal potentially frequent itemset size on average, and *D* is the number of the transactions. We use *DB1* and *DB2* in Section 7.2 and *DB2* in Section 7.3.

## 7.2 Multiple FP-tree Generation

For datasets containing 10K to 100k transactions, suppose we need to generate four FP-trees with regard to four support thresholds, 0.25%, 0.5%, 0.75% and 1%. At first we generate an FP-tree with a support of 0.5%, then we need to generate FP-trees for support thresholds, 0.75%, 1% and 0.25%. When the support threshold increases from 0.5% to 0.75% or 1%, we can easily prune the FP-tree with the support threshold of 0.5% to obtain new FP-trees without having to scan the databases. However, when the support threshold decreases from 0.5% to 0.25%, we have to scan the database to obtain the new FP-tree. No matter how support thresholds change, our algorithm can generate four FP-trees easily after the generation of a P-tree. If support thresholds decrease, the FP-tree method consumes almost the same runtimes with different support thresholds. This is because the FP-tree method has to discard the previous computations and rescan the database. In this case, the more FP-trees are generated, the more time the FP-method takes. If support thresholds increase, the FP-tree method is similar to our method, which needs very little extra time to generate the FP-trees.

Figure 8 plots the scalability of each method as the number of transactions is increased from 10,000 to 100,000 transactions. Each method generates four FP-trees corresponding to the support thresholds 0.25%, 0.5%, 0.75% and 1%. Their runtimes are shown in Table 6. As shown, both the runtimes scale quite linearly. The P-tree method beats the FP-tree method for all dataset sizes by a factor of about 1.5.

Table 6: Runtimes (in seconds) with the Number of Transactions

<i>Method</i>	<i>10k</i>	<i>20k</i>	<i>30k</i>	<i>40k</i>	<i>50k</i>	<i>80k</i>	<i>100k</i>
P-tree	8.755	17.908	27.642	37.614	48.278	82.407	107.039
FP-tree	13.757	26.456	40.307	54.510	68.703	113.288	145.273

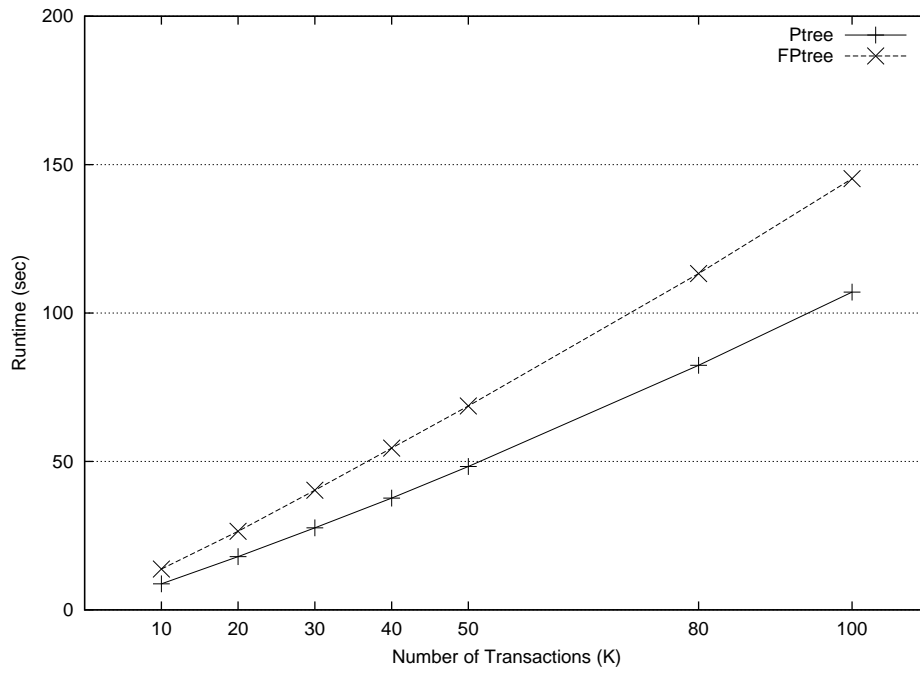


Figure 8: Runtimes with the Number of Transactions

Table 7: Runtimes with Single Update

<i>Method</i>	<i>10k</i>	<i>20k</i>	<i>30k</i>	<i>40k</i>
P-tree	19.540	41.142	71.863	100.260
FP-tree	40.213	80.966	124.862	167.798

Table 8: Runtimes with Multiple Updates

<i>Method</i>	<i>10k</i>	<i>20k</i>	<i>40k</i>	<i>50k</i>	<i>80k</i>
P-tree	8.755	19.842	41.363	59.536	108.724
FP-tree	13.757	40.213	94.723	163.426	276.714

### 7.3 FP-tree Updating

Figure 9 plots the results of each method when the same amount data is added into the original dataset. For instance, another 10k new transactions come into a dataset of 10k transactions, or 20k new transactions into a dataset of 20K transactions. Similarly, we need to generate an FP-tree with 0.5% support threshold, then FP-trees with 0.25%, 0.75% and 1% support thresholds. Their performances are also shown in Table 7. In this case, the P-tree method outperforms the FP-tree method by a factor of more than 1.5.

Table 8 shows the runtimes of each method when more than one update happens to a dataset, which originally contains 10k transactions. The number of transactions in each updating is 10k, 20k, 10k and 30k respectively. At last, the 10k dataset turns out to be a database of 80k transactions. We also generate four FP-trees with regard to 0.5%, 0.25%, 0.75% and 1% support thresholds. Figure 10 shows the performance of both methods for those updates. When the number of transactions increases from 10,000 to 80,000, the runtime of the P-tree method increases by 13 times while the FP-method by 20 times. As shown, the final runtime of the FP-tree method is about 2.6 times of that of the P-tree method.

Compared with the FP-tree method, our algorithm can generate FP-trees fast and efficiently especially when support thresholds decrease. The performance gap increases as the dataset becomes larger. One can imagine that the factor will become much larger to an order of magnitude if more FP-trees need to be generated or bigger datasets need to be processed.

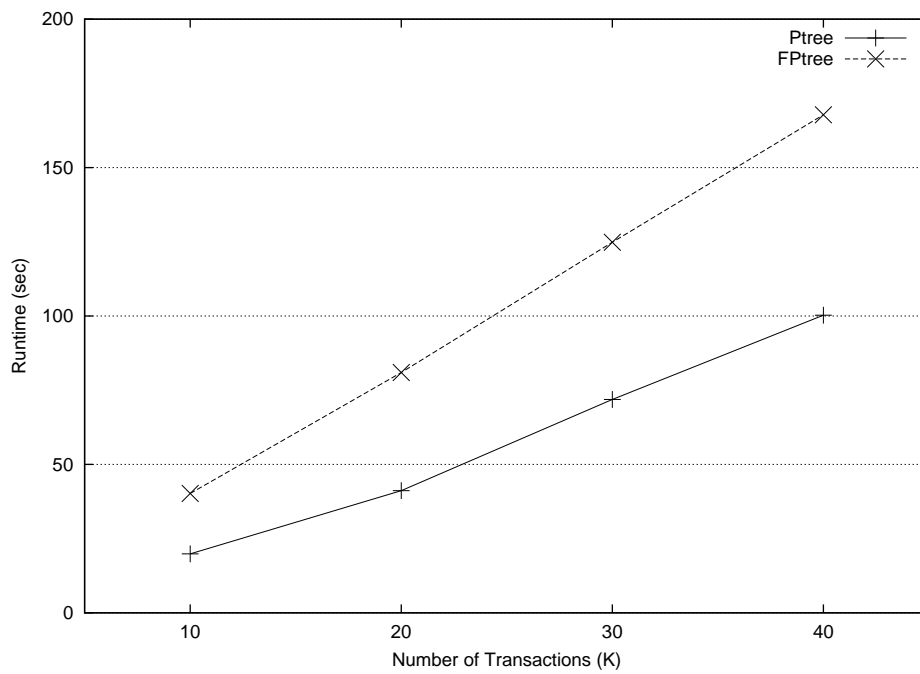


Figure 9: Runtimes with Single Update

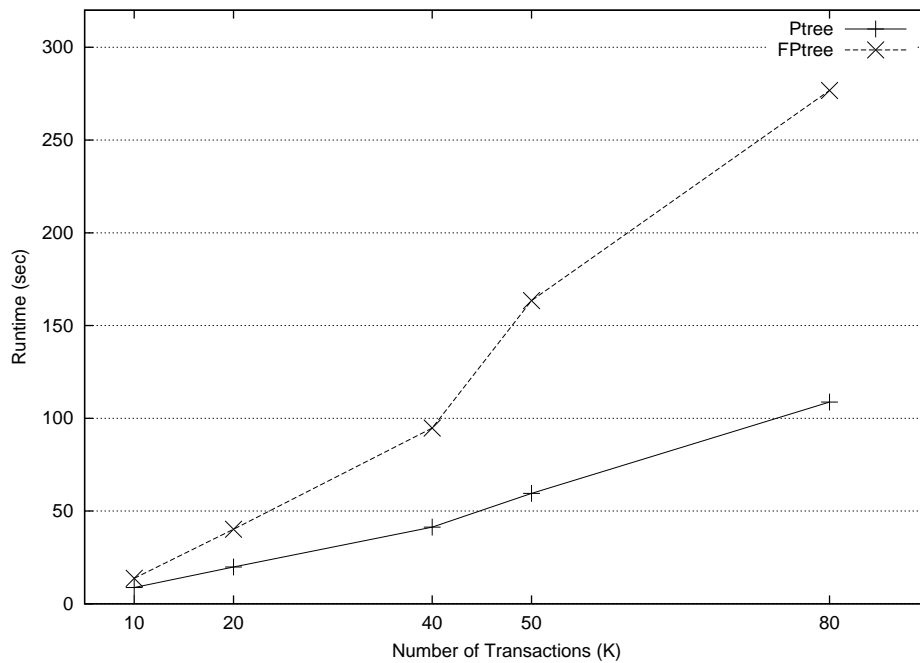


Figure 10: Runtimes with Multiple Updates

## 8 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, 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.

One possible direction for future work is to mine the desired and interesting rules from the P-tree structure given a specified rule antecedent.

## Acknowledgments

The authors would like to thank the two anonymous reviewers for their detailed constructive comments, which have helped improve the paper.

## References

- [1] R. Ng, L. V. S. Lakshmanan, J. Han, and A. Pang. Exploratory mining and pruning optimizations of constrained associations rules. *Proc. of ACM Int. Conf. on Management of Data (SIGMOD)*, pages 13–24, 1998.
- [2] S. Sarawagi, S. Thomas, and R. Agrawal. Integrating association rule mining with relational database systems: Alternatives and implications. *Proc. of ACM Int. Conf. on Management of Data (SIGMOD)*, pages 343–354, 1998.
- [3] R. Srikant, Q. Vu, and R. Agrawal. Mining association rules with item constraints. *Proc. of Intl. Conf. on Knowledge Discovery and Data Mining (KDD)*, pages 67–73, 1997.
- [4] R. Agarwal, C. Aggarwal, and V. V. V. Prasad. Depth-first generation of long patterns. *Proc. of Intl. Conf. on Knowledge Discovery and Data Mining (KDD)*, pages 108–118, 2000.
- [5] R. Agrawal, T. Imielinski and A. Swami. Mining Association Rules between Sets of Items in Large Database. *Proc. of ACM Int. Conf. on Management of Data (SIGMOD)*, pages 207–216, 1993.
- [6] R. Agrawal and R. Srikant. Fast algorithms for mining association rules. *Int. Conf. Very Large Data Base (VLDB)*, pages 487–499, 1994.

- [7] R. Agrawal and R. Srikant. Mining sequential patterns. *IEEE International Conference on Data Engineering (ICDE)*, pages 3–14, 1995.
- [8] R. J. Bayardo. Efficiently mining long patterns from databases. *Special Interest Group on Management Of Data (SIGMOD)*, pages 85–93, 1998.
- [9] B. Lent, A. Swami, and J. Widom. Clustering association rules. *IEEE International Conference on Data Engineering (ICDE)*, pages 220–231, 1997.
- [10] M. Klemettinen, H. Mannila, P. Ronkainen, H. Toivonen, and A.I. Verkamo. Finding interesting rules from large sets of discovered association rules. *Third International Conference on Information and Knowledge Management (CIKM'94)*, pages 401–408, 1994.
- [11] J.S. Park, M.S. Chen, and P.S. Yu. An effective hash-based algorithm for mining association rules. *Proc. of ACM Int. Conf. on Management of Data (SIGMOD)*, pages 175–186, 1995.
- [12] A. Savasere, E. Omiecinski, and S. Navathe. An efficient algorithm for mining association rules in large databases. *Int. Conf. Very Large Data Base (VLDB)*, pages 432–443, 1995.
- [13] J. Han, J. Pei, and Y. Yin. Mining frequent patterns without candidate generation. *Proc. of ACM Int. Conf. on Management of Data (SIGMOD)*, pages 1–12, 2000.
- [14] J. Han, J. Pei, Y. Yin, and R. Mao. Mining Frequent Patterns without Candidate Generation: A Frequent-Pattern Tree Approach. *Data Mining and Knowledge Discovery*. 8(1): 53–87, 2004.
- [15] R. Agarwal, C. Aggarwal, and V. V. V. Prasad. A tree projection algorithm for generation of frequent itemsets. *Journal of Parallel and Distributed Computing*, 2000.
- [16] D. W. Cheung, J. Han, V.T. Ng and C.Y. Wong. Maintenance of discovered Association Rules in Large Databases: An Incremental Updating Technique. *IEEE International Conference on Data Engineering (ICDE)*, pages 106–114, 1996.
- [17] D. W. Cheung, S.D. Lee and B. Kao. A General Incremental Technique for Maintaining Discovered Association Rules. *Proc. of 5th DASFAA Conf*, 1997.