2001

Mining incremental association rules with generalized FP-tree.

Yue. Su
University of Windsor

Recommended Citation

Su, Yue., "Mining incremental association rules with generalized FP-tree." (2001). *Electronic Theses and Dissertations.* Paper 2741.

Mining Incremental Association Rules With Generalized FP-tree

By

Yue Su

A Thesis
Submitted to the Faculty of Graduate Studies and Research
through the School of Computer Science
in Partial Fulfillment of the Requirements
for the Degree of Master of Science at the
University of Windsor

Windsor, Ontario, Canada

2001

0-612-67602-1

952553

Abstract

Mining association rules among items in a large database have been recognized as one of the most important data mining problems. New transaction insertions and old transaction deletions may lead to previously discovered association rules no longer being interesting, and new interesting association rules may also appear. The process of generating association rules in the updated database using mostly only the updated part of the database and the previous association rules is called incremental association rules maintenance.

The most straightforward approach for mining incremental association rules in the updated database starts from scratch to mine the entire database again when update occurs. This approach is very time consuming because it uses the entire database and has to repeat many computations previously done. Some algorithms that utilize the previously discovered frequent patterns have been presented in order to improve the maintenance efficiency by reducing the computation time. However, they still suffer some shortcomings which include: (1) scanning the updated part of the database several times (at each level) to confirm previous large itemsets still large; (2) scanning the entire database several times when some previous small itemsets now become large in the updated part of the database.

This thesis proposes two new methods that use the frequent patterns tree (FP-tree) structure to reduce the required number of database scan. One is DB-tree algorithm which stores all the information in a tree structure and requires (1) no scanning of the original database, (2) to only scan the updated transactions once without involving candidate sets generation. Another method is FPUP algorithm, which predicts possible large itemsets for future mining. This FPUP approach also uses FP-tree structure to scan the old database less times than the existing FP algorithms for improved performance.

Key words: Apriori algorithm, Maintaining Incremental association rules, FP-tree algorithm, DB-tree algorithm, FPUP algorithm

To my husband.

Acknowledgements

There are many people I would like to thank for their help in writing this thesis.

I would like to express my deep appreciation to my supervisor Dr. Ezeife for her invaluable comments, encouragement, guidance and support during the thesis work. Without her help, this thesis would not have been completed.

I would also thank my internal reader Dr. Ahmed Tawfik; his pieces of advice contributed a lot to the quality of my thesis. I am thankful to my external reader Dr. Robert Schurko for being very accommodating and finding time to read my thesis. Special thanks also go to Dr. Peter Tsin for chairing my thesis defense.

I would give my sincere gratitude to my beloved husband for his endless love and everlasting support. His love supports me for finishing the graduate studies. Thank you and I love you.

I love to thank my parents who always stand behind me, love me and support me. Thank you, mom and dad.

Finally, I would like to thank my friends Hydi Dou and Jeffery Liu, who has been encouraging me and helping me during my thesis work.

Table of Contents

Abstract .... **IV**

Acknowledgements .... **VI**

Table of Contents .... **VII**

List of Figures .... **IX**

1. Introduction **1**

1.1 What is Association Rule? 3

1.2 The Measures of Association Rules 4

1.3 What are Itemsets and Frequent Itemsets 4

1.4 The Steps for Mining Association Rules 5

1.5 Advantages and Disadvantages of Association Rules 5

1.6 What is Incremental Association Rule Mining? 6

1.7 Existing Methods for Maintaining Incremental Association Rules 6

1.8 The Motivation for Thesis 7

1.9 Contributions of Thesis 8

2.0 Outline of Thesis 8

2. Previous/ Related Works **9**

2.1 The Association Rules Mining Review 9

2.1.1 Template Algorithm 9

2.2.2 Apriori Algorithm 9

2.1.3 DHP Algorithm 13

2.1.4 Partition Algorithm 13

2.1.5 Random Sampling Algorithm 14

2.1.6 DIC Algorithm 14

2.1.7 Carma Algorithm 15

2.2 The FP-tree Algorithm 15

2.3 The Incremental Updating Techniques 20

3. Mining Incremental Association Rules with FP-tree **28**

3.1 Using Database Tree (DB-Tree) 30

3.1.1 Constructing DB-tree 31

3.1.2 Mining Incremental Rule with DB-tree 34

3.2 Mining Incremental Association Rules Using Potential Large Itemsets (FPUP) ... 41

4. Experimental Evaluation and Performance Analysis 51

4.1 Experiment 1: Execution Time for Different Algorithms 51

4.2 Experiment 2: The Effect of Different Changed Transaction Size Testing 53

4.3 Experiment 3: Tolerance Range Testing 57

4.4 Correctness of Algorithm Implementations 58

5. Conclusions and Future Work 61

References 63

APPENDIX:Correctness Testing Results 69

VITA AUCTORIS 82

List of Figures

1.1 Customer Purchase data........3

2.1-1 Transaction Database........10

2.1-2 Apriori Algorithm........12

2.2-1 The Database TDB........16

2.2-2a The First Branch of FP-tree........18

2.2-2b The Second Branch of FP-tree........18

2.2-3 The FP-tree of TDB........18

2.2-4 Mining of All- Patterns by Creating Conditional (Sub)-Pattern Base........19

2.3-1 The Database after Transaction Insertion of FUP Algorithm........22

2.3-2 The Database after Transactions Deletion of FUP2 algorithm........24

2.3-3 The Incremental Mining Process........26

3.1-1 DB-tree........33

3.1-2 The Updated Database after Transaction Insertion of DB-tree Algorithm........34

3.1-3 DB-tree after Inserted Transactions........35

3.1-4 FP-tree of The Updated Database D'........37

3.1-5 The Updated Database after Transactions Deletion of DB-tree Algorithm........38

3.1-6 Algorithm for Updating The DB-tree........40

3.1-7 Algorithm for Projection FP-tree........41

3.2-1 The FP-tree of FPUP Algorithm........43

3.2-2 The Database after Transaction Insertion of FPUP........44

3.2-3 Updated FP-tree after Transaction Insertion........45

3.2-4 The Database after Transactions Deletion of FPUP........46

3.2-5 The Updated FP-tree after Transaction Deletion........47

3.2-6a FPUP Mining Algorithm for Transaction Insertion........48

3.2-6b FPUP Algorithm for Transactions Deletion........49

4.1-1 Experimental Data- Execution Time (seconds) for Different Algorithms........52

4.1-2 Execution Times on Different Support Level........52

4.2-1 Experimental Data-Execution Times(seconds) with Different Changed Transaction Size on Minimum Support 0.5%........54

4.2-2 Execution Times with Different Changed Transaction Size on Minimum Support 0.5%........54
4.2-3 Experimental Data- Execution Times (seconds) with Different Changed Transaction Size on Minimum Support 1%........55
4.2-4 Execution Times with Different Changed Transaction Size on Minimum Support 1%........55
4.2-5 Experimental Data- Execution Times (seconds) with Different Changed Transaction Size on Minimum Support 6%........56
4.2-6 Execution Times with Different Changed Transaction Size on Minimum Support 6%........56
4.3-1 Experimental Data-Execution time (seconds) of FPUP on Different Tolerance 57
4.3-2 Execution Time of FPUP on Different Tolerance........58
4.4-1 The Dataset to Testing The Correctness........59
4.4-2 The Results of Different Algorithms........60

1. Introduction

Databases are more widely used in today's applications because of the availability of powerful and affordable database systems and the huge amounts of information that need to be collected and analyzed. This explosive growth in data and databases has generated an urgent need for new techniques and tools (e.g., data mining) that can intelligently and automatically transform the processed data into useful information and knowledge. Thus, data mining has become a research area of increasing importance [UGP+96] [Sh91]. Data mining is a process that finds relationships and patterns within a large amount of data stored in a database. Data mining is also referred to as knowledge discovery in databases, which means a process of nontrivial extraction of implicit, previously unknown and potentially useful information from data in databases [Sh91]. It helps end users extract useful business information from large databases. By knowledge discovery in databases, interesting knowledge, regularities, or high-level information can be extracted from relevant sets of data in databases and be investigated from different angles. Large databases thereby serve as rich and reliable sources for knowledge generation and verification [CHY96]. Data mining algorithms are used to discover patterns, clusters and models from data. These patterns and hypotheses are then rendered in operational forms that are easy for people to visualize and understand.

Association rule mining is a data mining technique. The concept of the data mining association rule was first introduced by Rakesh Agrawal in [AIS93]. It is based on supermarket data with a large collection of items. An association rule from this domain is a rule such as "80% of all customers who buy product A and B also buy products C and D". Discovering such customer buying patterns is useful for customer segmentation, cross-marketing, catalog design and product placement [Hi99]. The Apriori [AS94] is the foundation algorithm for the general association rules mining. Following this, many algorithms have tried to enhance the efficiency and performance of the Apriori. These algorithms include [AY98] [To96] [Hi99] [AIS93] [AS94] [SON95] [BMU+97] [SVA97] [PCY95]. In all of these algorithms, Apriori [AS94] algorithm is the foundation for the others.

Databases are usually 'dynamic' instead of 'static' because there are always some transaction insertions, deletions or modifications happening to the existing database. When the database is updated, the discovered association rules may change. Some old rules may no longer be interesting and yet new rules may emerge. The maintenance of incremental association rules is a technique to maintain the association rules discovered by data mining using mostly only the updated part of the database.

A straightforward method to deal with updating association rules is based on the Apriori algorithm [AS94] that generates association rules in the new database from the scratch. Obviously, it is not efficient. Thus, more efficient algorithms dealing with the incremental maintenance of association rules include [CHN+96, CLK97, TBA+97, LCK98, ZE01]. In [CHN+96], an algorithm named FUP is presented. It updates the discovered association rules in a database when new transactions are inserted into the database. It cuts down the amount of repetitive calculations performed scanning the database table to obtain the supports of candidate itemsets by utilizing the previous association rules mining result to generate new association rules. The algorithm FUP2 in [CLK97] is complementary to FUP and more general than FUP. It updates the discovered association rules when new transactions are inserted into the original database or old transactions are deleted from the original database. The FUP2 algorithm can generate the precise association rules in the updated database. An algorithm in [TBA+97] saves the computation and scanning time by storing all the calculated results of the original database making it easy to update rules when the database is updated. All the above algorithms are Apriori based. DELI in [LCK98] discusses when is the best time to update the incremental association rules. It sacrifices accuracy to gain the speed. Because FUP and FUP2 are Apriori like algorithms, they need much time to scan the updated part of the database and in some cases, original database, many times. The MAAP algorithm presented in [ZE01] begins to calculate candidate itemsets from certain higher n-level, to avoid the calculation of many lower level large itemsets that involve huge table scans.

In [HPY00] a new mining method is proposed. This method is basically different from the Apriori. It stores all the frequent patterns information in a tree like structure, which is called FP-tree, and it is a substantially compact structure compared to the huge database. Then the mining process consists of recursively building the conditional pattern base and FP-tree for each node in the FP tree. This algorithm avoids generating large amounts of candidate itemsets and scanning the entire database each level thus achieving drastic improvement in comparison to the Apriori algorithm.

Given a database as Figure 1.1, we use this to introduce some definitions:

Transaction ID (TID)	Item
1	sugar, milk, bread, egg
2	bread, apple
3	sugar, bread, milk
4	sugar, milk
5	egg, milk, orange

Figure 1.1 Customer Purchase Data

1.1 What is Association Rule?

Mining association rules in transactional or relational databases has attracted a lot of attention recently. A transactional database is a table where records are super market kind of transactions. The format of a transaction representing items bought during a market visit is (TID, item1, item2, ..., item n), where TID stands for transaction id and the other attributes of the table represent the items sold in the super market. Association rules find the relationships among the different items (attributes) in a database transaction record. An association rule is an implication of the form $X \Rightarrow Y$ for itemsets X and Y, where $X \subseteq I$ and $Y \subseteq I$, $X \cap Y = \varnothing$ and $I = \{i_1, i_2 ..., i_n\}$ is the set of all items in the store. For example, from a large set of transaction data, one may find such an association rule as when 40% of customers buy milk and sugar, they usually also buy bread in the same transaction. This rule is expressed as:

milk, sugar $\Rightarrow$ bread and this rule applies to 40% of all transactions.

This rule is read as "if milk and sugar, then bread". The left side of the rule is called antecedent while the right hand side is the consequent.

1.2 The Measures of Association Rules

There are two measures for assessing the usefulness of association rules. They are support and confidence. The rule $X \Rightarrow Y$ has support s in the transaction set D if s% of transactions in D contains itemsets $X \cup Y$. In other words, $s\% = |D'| / |D|$ where $|D|$ represents the number of transactions in D and $|D'|$ represents number of transactions in D containing itemsets $X \cup Y$. For example, using transaction set in Figure 1.1, assume there is a set of items {sugar, milk, bread}. Since transaction 1 and 3 contain {sugar, milk, bread}, the support for the set of items {sugar, milk, bread} = 2/5*100%=40%. The rule $X \Rightarrow Y$ holds in the transaction set D with confidence c if c% of transaction in D that contains X also contains Y. The confidence $c\% = |X \cup Y| / |X|$. For example, because transactions 1, 3 and 4 from Figure 1.1 contain {sugar, milk}, the confidence of the rule milk, sugar $\Rightarrow$ bread = 2/3 * 100%=66.7%. Generally, support is used to determine how often the rule is applied while confidence is used to determine how often the rule is correct.

1.3 What are Itemsets and Frequent Itemsets

An itemset is a set of items, an itemset that contains k items is called a k-itemset. For example, itemset {sugar, milk, bread} is a set of items {sugar}, {milk} and {bread}, and this itemset is 3-itemset. Given a minimum support defined as a threshold, an itemset which has a support equal to or larger than this minimum support is called a frequent itemset or large itemset. For example, using a minimum support 3, sets of frequent 1-itemsets from Figure 1.1 (also called large 1-itemsets) are {(sugar), (milk,), (bread)}. Similarly, frequent 2-itemsets are {(sugar, milk)}. In this case, only itemsets with support greater than or equal to 3 are frequent.

1.4 The Steps for Mining Association Rules

Existing algorithms proceed in 2 steps to compute association rules:

1. Find all large itemsets.
2. For each large itemset Z, find all subsets X, such that the confidence of

X⇒Z-X is greater than or equal to the confidence threshold. For example, assume we find the itemset (sugar, milk) large after step 1, using the Apriori algorithm, to generate association rules from this one large itemset, we find the confidences of the rules sugar ⇒ milk, milk ⇒ sugar, which are the only rules we can get from this large itemset. We then keep only the rules which have confidence greater than the minimum confidence. For example, using Figure 1.1 and a minimum confidence of 1.5, it can be seen that the rule sugar ⇒ milk has confidence 3/3=1, while milk ⇒ sugar has a confidence of 3/4 =0.75. Since these confidences are all greater than the minimum confidence of 0.5, both rules kept as results. The process is repeated for all large itemsets. Most algorithms address the first step because it is the most time and memory space consuming. The optimistic approaches have to be found to improve the performance and efficiency.

Association rules problem has been extensively studied by many researchers and a number of fast variants have been proposed [AS94] [PCY95] [SON95] [Hi99] [To96] [BMU+97].

1.5 Advantages and Disadvantages of Association Rules

The discovery of interesting association relationships among huge amounts of data is useful in selective marketing, decision analysis, and business management. Such rules track the buying patterns in consumer behavior. The idea of an association rule is to develop a systematic method by which a user can figure out how to infer the presence of some sets of items, given the presence of other items in a transaction. Such information is useful in making decisions such as customer targeting, shelving, and sales promotions.

The large itemset method has proved as a useful tool for mining association rules in large database which are inherently sparse. However, in many cases the method is difficult to generalize to other scenarios for computational reasons. The first problem is spuriousness in itemset generation. For example, consider a retailer of breakfast cereal that surveys 5000 students on the activities they engage in the morning. The data shows that 3000 students play basketball, 3750 eat cereal, and 2000 students both play basketball and eat cereal. For a minimum support of 40%, and minimum confidence of 60%, we find the following association rules: *plays basketball* $\Rightarrow$ *eat cereal*. The association rule is misleading because the overall percentage of students eating cereal is 75% which is even larger than minimum confidence 60%. Thus, playing basketball and eating cereals are not positively associated. Another limitation of association rules is resulted by data skew. For a k-dimensional database, there are 2^k possibilities for itemsets. Some data sets may be such that a large number of these 2^k possibilities may qualify above the minimum support. For such situations, it may be necessary to set the minimum support to an unacceptably high level, This may result in a number of important rules being lost. Often, the value of minimum support is decided based on computational constraints or in restricting the number of rules generated to a manageable number. Thus, it is impossible to ascertain when important rules are being lost and when they are not.

1.6 What is Incremental Association Rule Mining?

As data is inserted into, deleted from or modified in a database, the previous rules may lose their interestingness, new interesting rules may appear in the updated database. The process of generating association rules in the updated database using mostly only the updated part of the database and the previous association rules is called incremental association rules mining.

1.7 Existing Methods for Maintaining Incremental Association Rules

Apriori algorithm [AS94] is a straightforward method to deal with incremental association rules mining. FUP algorithm proposed in [CHN+96] is a more efficient incremental mining method than Apriori. It utilizes the previous association rules mining result to generate new association rules. Algorithm FUP2 [CLK97] is complementary to

FUP and more general than FUP. [TBA+97] proposed an algorithm saves the computation and scanning time by storing all the calculated results of the original database making it easy to update rules when the database is updated. In [LCK98], DELI algorithm uses a method that sacrifices accuracy to gain the speed. The MAAP algorithm presented in [ZE01] begins to calculate candidate itemsets from certain higher n-level to avoid the calculation of many lower level large itemsets that involve huge table scans.

1.8 The Motivation for Thesis

The existing incremental association rules mining algorithms are based on Apriori and try to reduce the database scan time and reduce the number of generated candidate itemsets. However, most of them have to calculate the candidate itemsets at each level and all of these algorithms have to scan the original database and the updated part of database a lot of times. The existing algorithms have the following drawbacks that affect the performance: (1) scanning the updated part of the database several times (at each level) to confirm that previously large itemsets are still large; (2) scanning the entire database several times when some previously small itemsets now become large in the updated part of the database.

The FP-tree structure is very efficient for association rules mining because using the FP-tree to mine the association rules, no candidate sets need to be generated. The second drawback above is dealt with. However, the first drawback is still remaining. The FP-tree stores the frequent patterns for the original database. If the database is updated, the frequent patterns may not be the same as stored in the FP-tree. In other words, the FP-tree may lose some frequent patterns for the new database. In this case, the scan on the old database is needed to update the FP-tree structure to make it consistent with the new database. Constructing the FP-tree is quite a big overhead in the FP-tree mining. It is impossible and impractical to scan the old database to update the FP-tree each time the database is updated.

In this thesis, two methods named DB-tree and FPUP (FP-tree based UPdating incremental association rules) are proposed. These two methods apply the FP-tree into the incremental association rules mining without scanning the old database even once, thus achieve the better response time. DB-tree stores all the information of the database in an FP-tree like structure. When mining association rules, it just projects the FP-tree from this structure. Another method, FPUP stores some information which may predicates the large itemsets of the future in the existing FP-tree.

1.9 Contributions of Thesis

This thesis presents two new algorithms (DB-tree and FPUP algorithms) for efficiently mining incremental association rules in updated database. They take advantage of the FP-tree technique of not generating the candidate set, but unlike the FP-tree algorithm, they 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. Usually, the algorithms only need scan the updated part of transactions once (at most twice) and do not generate the candidate sets and thus obtain good performance.

2.0 Outline of Thesis

The rest of the thesis is organized as follows: Chapter 2 reviews the existing work related to the thesis. Chapter 3 presents the detailed description of the new algorithms (DB-tree and FPUP) for maintaining incremental association rules on transaction data. Chapter 4 presents system implementation and performance analysis. Chapter 5 gives a conclusion and discusses future work.

2. Previous/ Related Works

In this chapter, previous and current research on mining incremental association rules are reviewed. The association rules mining techniques are generally discussed and basic data mining algorithm Apriori is introduced in section 2.1 and FP-tree in section 2.2. The incremental association rules mining algorithms are introduced and discussed in section 2.3.

2.1 The Association Rules Mining Review

The problem of generating association rules was first introduced in [AIS93] and an algorithm called AIS was proposed for mining all association rules. This is the prototype of Apriori algorithm. Agrawal and Srikant in [AS94] presented the Apriori algorithm, which is the basis for many association rule algorithms.

2.1.1 Template Algorithm

The concept of data mining association rules was first introduced by Rakesh Agrawal in 1993 in [AIS]. It was based on supermarket data with a large collection of items. The algorithm makes multiple passes over the dataset. In each pass, the supports for certain itemsets are measured. These itemsets are candidate itemsets. The candidate itemsets are generated during the passes when data is being read. Specially, after reading a transaction, it is determined which of the itemsets were large in the previous pass are present in the transaction. New candidate itemsets are generated by extending these large itemsets with other items in the transaction.

2.2.2 Apriori Algorithm

Apriori algorithm for discovering large itemsets makes multiple passes over the data. In the first pass, it counts the support of individual itemsets and determine which of them are frequent (have minimum support). In each subsequent pass, it starts with the itemsets found to be large in the previous pass. This is to generate new potentially large itemsets, candidate itemsets, and counts the actual support for these candidate itemsets during the pass over the data. At the end of the pass, it determines which of the candidate itemsets are actually large, and these determined large itemsets are used to generate new potentially large itemsets and candidate itemsets for the next pass. The algorithm stops when no more candidate or large itemsets can be generated.

Given a transaction database *D*, Figure 2.1-1, the example is shown for illustrating the Apriori algorithm. Suppose that all items I= {A, B, C, D, E} in *D*.

Transaction Database *D*

TID	Items
100	A C D
200	B C E
300	A B C E
400	B E

Figure 2.1-1 Transaction Database

In the first iteration, Apriori simply scans all the transactions to count the number of occurrences for each item. The set of candidate 1-itemsets, C_1={{A},{B},{C},{D},{E}} is obtained. Assuming that the minimum support is 2 (i.e., s=40%), the set of large 1-itemsets is L_1 ={A, B, C, E}, where large 1-itemsets denotes each large itemset in L_1 contains only one item. Apriori exploits the observation that all subsets of a large itemset are large themselves. So, if any subset is not large, this itemset can be pruned off. So Apriori uses the Apriori-gen function (L_1 join L_1) to generate a candidate set of itemsets C_2. In joining L_1 and L_1, which means when one itemset X is chosen from the first L_1 , another itemset Y is chosen from the second L_1, the last item of X must be less than the last item of Y, the rest of items of X must be the same as those of Y. It ensures that all the subsets of the candidate sets in C_2 is large themselves. C_2 consists of 2-itemsets,

C_2={{AB}, {AC}, {AE}, {BC}, {BE}, {CE}}. Next, the four transactions in D are scanned and the support of each candidate itemset in C_2 is counted. The set of large 2-itemsets, L_2, is therefore determined based on the support of each candidate 2-itemset in C_2. L_2 ={{AC}, {BC}, {BE}, {CE}}. The set of candidate itemsets, C_3, is generated from L_2 by using the Apriori-gen function as follows. When doing the join operation, the first k-2 items in the two set (L_k join L_k) must be the same accordingly, the k-1 item are not the same, and the (k-1) item for the first L_k must be less than the (k-1) item in the second L_k participating in the join. From L_2, two large 2-itemsets with the same first item, such as {BC} and {BE}, are identified first. The join of L_2 = {(B C), (B E)} and L_2 = {(B C), (B E)} will give C_3 ={(B C E)} since the first itemset {B C} of first L_2 apriori-gen joins with the second itemset (B E) of the second L_2 . Next, it goes through a pruning step to remove all candidate sets with any of its subsets not large in the previous iterations. Since all subsets (B C), (B E) and (C E) are large, the (BCE) remains a candidate 3-itemset. Apriori then scans all the transactions and confirms large 3-itemsets L_3 is {B C E} because its support is 2. Since there is no candidate 4-itemset to be constituted from L_3, Apriori ends the process of discovering large itemsets. The Apriori algorithm is presented in Figure 2.1-2

The Apriori is a multi-pass algorithm, it makes one pass over the database for each level. There are thus k or k+1 passes over the database, where k is the size of the largest frequent itemset. (k+1) passes are needed if there are candidates of size (k+1). This algorithm makes many passes over the transaction database, so it limits the performance and efficiency.

```
$L_1$ = {large 1-itemsets};
For (k=2; $L_{k-1} \neq 0$; k++) do begin
    $C_k$ = apriori-gen ($L_{k-1}$);    //New candidates
    forall transactions $t \in D$ do begin
        $C_t$ = subset($C_k$, t);    // candidates contained in t
        forall candidates $c \in C_t$ do
            c.count++;
        end
        $L_k = \{c \in C_k \mid c.count \geq minsup\}$
    end
Answer = $\cup_k L_k$;

Apriori-gen function
Step1:
 Insert into $C_k$
 Select p.item$_1$, p.item$_2$, ..., p.item$_{k-1}$, q.item$_{k-1}$
 From $L_{k-1}$ p, $L_{k-1}$ q
 Where p.item$_1$ = q.item$_1$, ..., p.item$_{k-2}$=q.item$_{k-2}$, p.item$_{k-1}$< q.item$_{k-1}$;
Step2:
 Forall itemsets $c \in C_k$ do
     Forall (k-1)-subsets s of c do
         If ($s \notin L_{k-1}$) then
             Delete c from $C_k$;
```

Figure 2.1-2 Apriori Algorithm

There are some other algorithms that improve on the efficiency of the Apriori. The DHP algorithm [PCY95] uses hashing techniques to filter out unnecessary itemsets and is proposed for efficient large itemset generation. The sampling algorithm [To96] randomly selects the sample from the database to generate the candidate itemsets therefore reducing the database scan time. The Partition algorithm [SON95] divides the database into n partitions to fit in the main memory and finds the all-large itemsets in each partition, and then it combines all the large itemsets of same lengths from all partitions to generate the candidate itemsets of the entire database. Finally it counts the real support of each candidate itemset in order to get the large itemsets. The DIC algorithm [BMU+97] proposed a method for large itemset generation that reduces the number of passes over the transaction database by counting some (k+1)-itemsets and counting k-itemsets simultaneously.

The Apriori algorithm is a level-wise association rules mining algorithm, which had been extended to accommodate various kinds of association mining problems. There are problems to be solved which include (1) straight association rule mining [To96, Hi99, AIS93, SON95, BMU+97, PCY95]. (2) incremental association rules mining [CHN+96, CLK97, ZE01, TBA+97]. (3) multidimensional association rules mining [KHC97]. (4) web mining [SSC97, BL98]; (5) generalized association rules mining [SA95, HMW+98]; etc. Apriori was the algorithm only defined for solving straight association rules mining. However, in order to solve problems like (2) and others, existing algorithms are based on the Apriori method. FP-tree [HPY00] is proposed as a new method for solving the general mining problems, which removes the need for generating candidate sets and cuts down on number of database scans.

2.1.3 DHP Algorithm

The DHP algorithm is an effective hash-based algorithm for the candidate set generation which was proposed in [PCY95]. The DHP algorithm uses the technique of hashing to filter out unnecessary itemsets for the next candidate itemset generation. DHP has two major features: one is efficient generation of large itemsets by utilizing a hashing technique; another is effective reduction of transaction database size by employing pruning techniques. Generally, given a large database, the initial extraction of useful information from the database is the most costly part. DHP algorithm is very efficient for the generation candidate large itemset, in particular for the large 2-itemset.

2.1.4 Partition Algorithm

The partition algorithm is a more efficient algorithm for mining association rules which is fundamentally different from the Apriori and DHP algorithms. It reduces both CPU and I/O overheads. This algorithm was proposed in [SON95] for finding large itemsets by dividing the database into n partitions. The suitable size of each partition is that the set of transaction can be maintained in main memory. Then large itemsets are generated separately for each partition. Then it combines (unite) the large itemsets of same lengths from all partitions to generate the candidate itemsets of the whole database.

Finally count the occurrences of all the candidate itemsets to determine whether the candidate itemsets are large or not.

DHP algorithm reads the database at most twice to generate all significant association rules.

2.1.5 Random Sampling Algorithm

The use of random sampling [To96] to generate large itemsets may save considerable expense in terms of the I/O costs. The idea is to pick a random sample in order to use this sample to find association rules that probably hold in the database, and them to verify the results with the rest of the database. The size of the data collection has an essential role in database mining. Association rule algorithms require multiple passes over the whole database, and subsequently the database size is the most influential factor of the execution time for very large database. The random sampling algorithm uses only a random sample of the database to find approximate regularities to reduce the number of database scan times. The mining step can be divided into two steps: step one uses a random sample to efficiently find supersets of the collection of frequent sets; step two use the rest of the database to compute the exact frequencies of the sets to find the large itemsets.

2.1.6 The DIC Algorithm

An algorithm DIC (Dynamic Item Counting) algorithm was proposed in [BMU+97]. This algorithm uses fewer passes over the database than classic algorithms such as the Apriori algorithm, and yet uses fewer candidate itemsets than methods based on sampling.

DIC starts counting an itemset as soon as it suspects that it may be necessary to count it instead of waiting until the end of the previous pass. This algorithm generally makes around two passes. This makes it considerably faster than Apriori.

2.1.7 Carma Algorithm

The Carma algorithm [Hi99] is inspired by online aggregation. The user is free to change the support threshold any time during the first scan of the transaction sequence using Carma. The Carma algorithm will terminate with the precise support for each large itemset after at most 2 scans. Carma uses distinct Phase I and Phase II algorithms for the first and second scan of the transaction sequence. During the first scan Phase I continuously constructs a lattice of all potentially large itemsets. After each transaction, it inserts and/or removes some itemsets from the lattice. At the end of the transaction sequence, Phase I guarantees that the lattice contain a superset of all large itemsets relative to some threshold. Phase II initially removes all itemsets which are trivially small. By rescanning the transaction sequence, Phase II determines the precise number of occurrences of each remaining itemset and continuously removes all itemsets, which turn out to be small. Finally the algorithm ends up with the set of all large itemsets along with their support.

2.2 The FP-tree Algorithm

All the methods based on the Apriori use an important common lemma that is: if any length-k pattern is not frequent in the database, none of its length-(k+1) patterns can be frequent later. This property leads to the powerful pruning of the set of itemsets to be examined in the search for longer frequent patterns based on the existing ones.

Besides this property, the methods adopt the level-wise, candidate generation-and test approach which needs to scan the database multiple times. The first scan finds all the length-1 large patterns. The length-k (k>1) frequent patterns can be obtained by scanning the database the k-th time to check the length-k candidate itemsets, which is generated from the length-(k-1) frequent patterns found in the previous pass.

Although there have been many techniques [AY98, To96, Hi99, AIS93, SON95, BMU+97, PCY95] developed for reducing the number of database scans and the number of candidates to be generated, when the minimum support threshold is low or the length of the patterns to be generated is long, the candidate generation-based algorithm may still bear the following shortcomings:

1) The number of candidates to be generated may still be huge, especially when the length of the patterns to be generated is long.

2) Each scan is executed on the entire database, which is very costly when the database is very large and the number of candidates to be examined is huge.

A new efficient data structure for mining large itemsets without candidate set generation was proposed in [HK01][HPY00]. The previous Apriori-like, level-wise approaches have the bottleneck of the candidate set generation and testing. So, the new structure is proposed to avoid generating huge amounts of candidate sets in order to improve performance.

The novel, compact data structure named frequent pattern tree or FP-tree is an extended prefix-tree structure, which stores all the frequent patterns in it. Next, an example for mining frequent patterns using FP-tree algorithm is given.

Given the database TDB (first two columns) as Figure 2.2-1. Assume the minimum support is *s*=0.6 (support occurrence is 3).

D

TID	Items Bought	(Ordered) Frequent Items
100	f, a, c, d, g, i, m, p	f, c, a, m, p
200	a, b, c, f, l, m, o	f, c, a, b ,m
300	b, f, h, j, o	f, b
400	b, c, k, s, p	c, b, p
500	a, f, c, e, l, p, m, n	f, c, a, m, p

Figure 2.2-1 The Database TDB

The steps for mining association rules consist of:

(i) Constructing the FP-tree from the database as shown in Figure 2.2-2.

(ii) Mining the frequent patterns on FP-tree. This consists of first constructing frequent patterns for each frequent item (e.g. for frequent item p, it is <p:3>), then all the patterns in the FP tree that define this frequent pattern (e.g., <f:4,c:3,a:3,m:2,p:2> and <c:1,b:1,p:1>), and finally deriving the conditional pattern base of each frequent items from its FP patterns such that all of the paths can be defined as only one FP-tree branch that satisfies minimum support. For example, the conditional pattern base for frequent item p, constructed from paths <f:4,c:3,a:3,m:2,p:2> and <c:1,b:1,p:1> are <f:2,c:2,a:2,m:2,p:2> and <c:1,b:1,p:1>. This !eads to only one branch <cp:3> meaning that the only frequent pattern from this item is <cp:3>.

Details of the procedure for constructing the FP-tree are given in the following steps:

1) Scan the TDB once to find all the large-1 items, sort them in descending order. <(f:4), (c:4), (a:3), (b:3), (m:3), (p:3)>, (the number indicates the frequency or support). The FP-tree will be constructed in this order. The order is important because there are better chances that more prefix strings can be shared. The sorted frequent items are listed in the third column in Figure 2.2-1
2) Create the root of tree labeled with 'null'. Scan the database the second time. 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)> as shown in Figure 2.2-2a. The second transaction shares the (f, c, a) with the first branch (the existing one), so only increment the count by one of these shared nodes. But the frequent item b leads to a separate branch from b as shown in Figure 2.2-2b. The rest of the transactions are scanned and the tree constructed the same way.

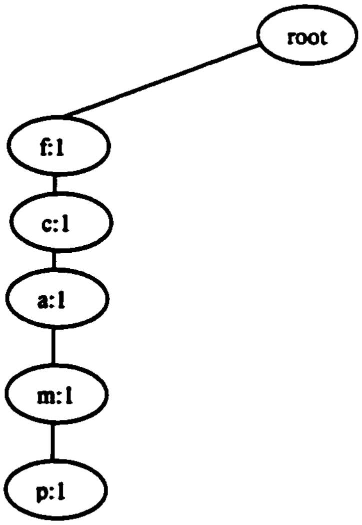

Figure2.2-2a The First Branch of FP-tree

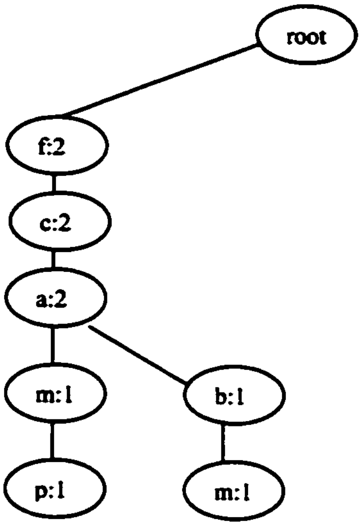

Figure2.2-2b The Second Branch of FP-tree

3) To facilitate tree traversal, an item header table is built in which each item points to its occurrence in the tree via a head of node link. Note the nodes with the same item name are linked in sequence via such node-links.

The constructed FP-tree is on Figure 2.2-3.

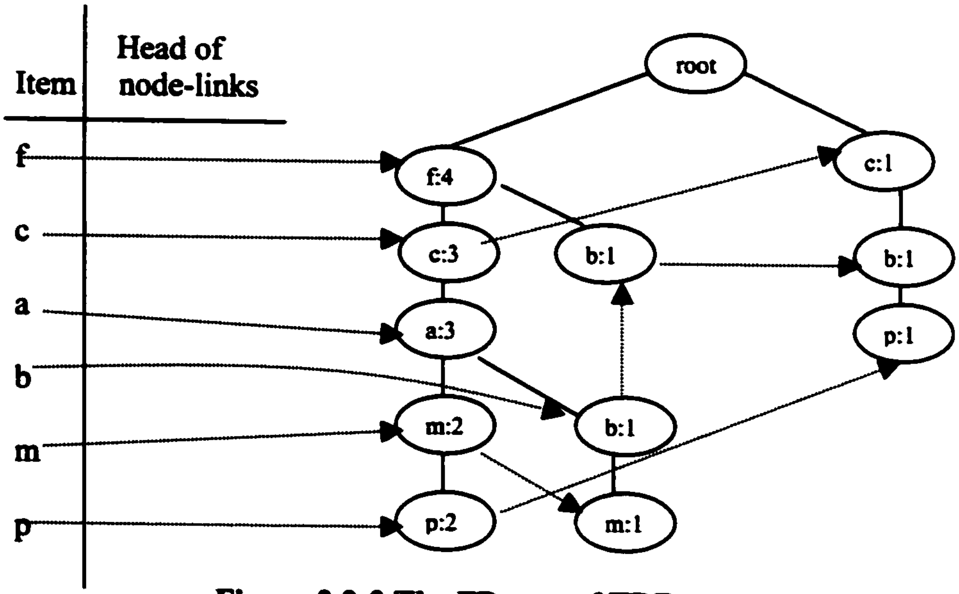

Figure 2.2-3 The FP-tree of TDB

It can be seen that to construct the FP-tree, one scans the entire database only twice. The first time to compute the frequent 1-items and the second time to build the tree. The complexity of constructing a transaction ***Trans*** into the FP-tree is **O** (|***Trans***|), where |***Trans*** | is the number of frequent items in the transaction ***Trans***.

The second step consists of mining the frequent patterns using the above FP-tree. From the constructed FP-tree, all the possible frequent patterns of each frequent item a_i can be obtained by following a_i 's head in the FP-tree header. The mining process starts from the bottom of the header table. For node p, it derives a frequent pattern (p: 3) and two paths in the FP-tree: (f: 4, c: 3, a: 3, m: 2, p: 2) and (c: 1, b: 1, p: 1). The prefix (f, c, m) happened together with p twice and prefix (c, b) happened with p once. So p's sub-patterns <f: 2, c: 2, a: 2, m: 2> and <c: 1, b: 1> consist of the **conditional pattern base** of p. Construction of an FP-tree on this conditional pattern base leads to only one branch (c: 3). Hence only one frequent pattern (cp : 3) is derived. So far the search for frequent patterns associated with p terminates. For item m, it derives a frequent itemset (m: 3) and two paths <f: 4, c: 3, a:3, m:2> and < f:4, c:3, a:3, b:1, m: 1>. Although p appears together with m as well, there is no need to include p here in the analysis since any frequent itemsets involving p has been analyzed in the previous examination of p. So the m's conditional sub-database is, {(f:2, c:2, a: 2), (f:1, c:1, a:1, b:1)}. Constructing an FP-tree on it, we derive m's conditional FP-tree, <f:3, c:3, a:3>, a single frequent itemset path. Since m's conditional FP-tree, < f:3, c:3, a:3>, has a single branch, instead of recursively constructing its conditional FP-trees, one can simply enumerate all the combinations of its components, i.e., {(m:3),(am:3), (cm:3), (fm:3), (cam:3), (fam:3), (fcam:3), (fcm:3)}. The conditional pattern base and conditional FP-tree of other items can be obtained in a similar way. The complete conditional pattern is in figure 2.3-4.

Item	Conditional pattern base	Conditional FP-tree
p	{(f:2, c:2, a:2, m:2), (c:1, b:1)}	{(c:3)}| p
m	{(f:2, c:2, a:2), (f:1, c:1 a:1, b:1)}	{(f:3, c:3, a:3)}|m
b	{(f:1, c:1, a:1, b:1), (f:1, b:1) (c:1, b:1)}	∅
a	{(f:3, c:3)}	{(f:3, c:3)}|a
c	{(f:3)}	{(f:3)}|c
f	∅	∅

Figure 2.2-4 Mining of All- Patterns by Creating Conditional (Sub)-Pattern base

By mining the conditional FP-tree recursively, the maximal frequent patterns (all their sub-patterns are also frequent) are obtained as: {cp}, {fcam}. The maximal frequent patterns of the whole database are {cp}, {fcam} and {b}.

From the above examples, the FP-tree has the following good properties: (1) FP-tree contains complete information of the entire database with respect to frequent itemset mining, every transaction in database is mapped onto one path in the FP-tree, and the frequent itemset information is completely stored in the tree, (2) FP-tree is a highly compact structure since there are often a lot of sharing of frequent items among transactions, therefore the size of the tree is usually much smaller than its original database.

2.3 The Incremental Updating Techniques

In this section, several incremental association rules mining algorithms are introduced and discussed.

In [CHN+96], the FUP algorithm is proposed for the maintenance of discovered association rules in large databases. It is only suitable for the maintenance problem in the case of insertion. If there are frequent itemsets in the updated database, which are not large itemsets of the original database, the FUP algorithm will scan the original database to check whether they are large itemsets or not.

In [CLK97], FUP2 algorithm is proposed as the generalized and complementary algorithm to FUP that is very efficient when the deleted transactions constitute a small part of the database, when inserted transactions happened, it is the same as FUP. It divides the candidate sets into two parts, one is the itemsets that are previously large, another is the itemsets that are previously small with respect to the old database. Thus, for all the candidates in the previously large itemsets, it is easy to calculate the support occurrences in the updated database to decide if they are large or not any more; for all the candidate sets in the previously small itemsets, some of them can be pruned off if they are large in the deleted part of the database (they can not be large in the updated database), the remaining ones are checked in the original database to decide their support.

The above two algorithms deal with insertion and deletion of transactions in the database. The modification of a database can easily be treated as deletion first followed by insertion of some transactions.

The algorithm presented in [TBA+97] reduces the I/O requirements for updating the set of large itemsets by maintaining the large itemsets and the negative border along with their support counts. It saves the computation and scanning time by storing the negative border (the itemsets were not large in the original database) of the large itemsets. When the database is updated, the occurrences of the itemsets in both original large itemsets and negative borders are easy to get. Thus, it need not scan the original database. However, if there is an itemset that is beyond the negative border, this algorithm still has to scan the original database.

DELI [LCK98] discussed when is the best time to update the incremental association rules. It sacrifices accuracy to gain speed. It applies sampling techniques to estimate the support counts and gives an approximate upper bound on the amount of changes in the set of association rules introduced by the new transaction. If the bound is low, it indicates that the amount of changes in association rules in the new database is small. So the old association rules can still be kept as an approximation of association rules in the new database. If the bound is high, it indicates that the amount of changes in association rules in the new database is large. So, it is necessary to update the association rules in the new database.

The MAAP algorithm [ZE01] starts computing the large itemsets from a high k-level, if the items in this high level are large, their subsets in the lower lever are also large and included. This removes the need to scan the database for support of all lower level new large itemsets. It also uses the same Apriori property to reduce the number of candidate sets generated.

Next the examples are given to explain FUP and FUP2 algorithms. The original database is still the same as the one given in figure 2.2-1.The minimum support is still 60%. We use D to denote the original database, Δ^+ denotes the set of inserted transactions, and Δ^- denotes the set of deleted transactions, D' denotes the updated database (after insertion, deletion or modification)

First, FUP algorithm is introduced when some transactions are inserted into the database. The updated database is like Figure 2.3-1:

D'

	TID	Items Bought
	100	f, a, c, d, g, i, m, p
	200	a, b, c, f, l, m, o
	300	b, f, h, j, o
	400	b, c, k, s, p
	500	a, f, c, e, l, p, m, n
Δ^+	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
	1000	f, t, a, b, p, l, o

Figure 2.3-1 The Database After Transaction Insertion of FUP Algorithm

Because the minimum is 0.6 and total transaction is 10, the minimum occurrence is 6. The large-1 itemsets of the original database D is already known as: L_1={(f:4), (c:4), (a:3), (b:3), (m: 3), (p:3)>.

In FUP2, it first scans the updated part of the database Δ^+. For all itemsets that are in the old large 1-itemset L_1, FUP simply updates their support count. So the support occurrences of the previous large itemsets are updated to: {(f: 9), (c: 5), (a: 5), (b: 7), (m: 6), (p: 7)}. It can be seen that, c and a are not large any more. So, these losers are pruned off. The rest of the large-1 items in Δ^+ are: {(o:3), (l:4)}. Their support counts in the updated database D' are o=2 (support count in D)+3=5, l=2 (support count in D)+4=6. So far, the large-1 items in the updated database D' are found: L_1'= (f, b, m, p, l).

In the second iteration, the L_2 of original D is: {(ac), (af), (am), (cf), (cm), (cp), (fm)}. Since a and c are not large in the new database D', it filters out the itemsets in L_2 which include these two items leaving only (fm). Applying apriori-gen function on L_1, the set{ (fb), (fm), (fp), (fl), (bm), (bp), (bl), (mp), (ml), (pl)} is produced. Scan the Δ^+ for the second time. Since (fm) is already known large in D, so just update its support count. The support count of (fm) in D' is 6. The occurrence of the rest in Δ^+ are: (fb:4), (fp: 3), (fl:4); (bm:2); (bp:3), (bl:3), (mp:2), (ml: 2), (pl:4). Because (bm), (mp) and (ml) are not large in Δ^+ , they can not be large in the updated D' and are pruned off. Next, because the itemsets { (fb:4), (fp: 3), (fl:4); (bp:3), (bl:3), (pl:4)} were not large in the original database, their original supports were not known, the algorithm has to scan the original database D to find support count to decide if they are large or not in D'. The result is L_2'={ (fl:6)}.

So far, how to find the large itemsets in the insertion-updated database has been explained by an example. Next the FUP2 algorithm is presented. As it is mentioned in [CLK97] that when insertion happens, the FUP2 algorithm is the same as FUP. FUP2 is the complementary and general algorithm of FUP. It can deal with insertion, deletion and modification however FUP can only deal with insertion. We demonstrate FUP2 by using deletion next.

Suppose the transaction 300, 400, 500, 600 and 700 are deleted from the database, the updated database D' and the deleted transactions Δ^- are in Figure 2.3-2:

D'

TID	Items Bought
100	f, a, c, d, g, i, m, p
700	q, f, b, p, l, w
800	f, c, b, e, m, o
900	f, b, p, m, a, l
1000	f, t, a, b, p, l, o

Δ^-

200	a, b, c, f, l, m, o
300	b, f, h, j, o
400	b, c, k, s, p
500	a, f, c, e, l, p, m, n
600	p, f, m, g, o, l

Figure 2.3-2 The Database After Transactions Deletion of FUP2 Algorithm

According to FUP2 algorithm, the mining procedure is:

1. The candidate C_1 of the D' is all the items in D', the large-1 itemsets of the original database D (before the deletion) are: L_1={(f:9), (b:7), (m:6), (p:7), (l:6)}.
2. Partition C_1 into P_k and Q_k, where $P_k = C_k \cap L_k$, which is the set of candidate itemsets that are previously large with respect to D; $Q_k = C_k - P_k$, which denotes the set of candidate itemsets that are previously small. So, the C_1= {a, b, c, d, e, f, g, h, i, j, k, l, m, n, o, p, q, s, t, w}, P_1= (f, b, m, p, l), Q_1 = {a, c, d, e, g, h, i, j, k, n, o, q, s, t, w}.
3. Scan Δ^- for all the items occurrences in $P_1 \cup Q_1$. For items in P_1, it is easy to know their occurrences in the updated database D'(just update their occurrence when scanning the Δ^-): (f:4), (b:4), (m:3), (p: 4), (l:3). Obviously, they are all large. The

occurrence of items in Q_1 in Δ^- are: (a:2), (c:3), (d:0), (e:1), (g: 1), (h:1), (i:0), (j:1), (k:1), (n:1), (o:3), (q:0), (s:1), (t:0), (w:0)

4. Delete from Q_1 where the items' occurrence are equal or larger than 3. The remaining ones are: {a, d, e, g, h, i, j, k, n, q, s, t, w}.
5. Scan the updated D' to find out the occurrence of the itemsets in 4 to decide the large-1 itemset. L_1'=(f, b, m, p, l, a)
6. Next, apply apriori-gen function on L_1'. C_2= {fb), (fm), (fp), (fl), (fa), (bm), (bp), (bl), (ba), (mp), (ml), (ma), (pl), (pa), (la)}. Also the large-2 itemsets of original database D is already known. L_2 = {(fl:6)}
7. Partition C_2 into P_2 and Q_2. P_2= {(fl)}, Q_2= {(fb), (fm), (fp), (fa), (bm), (bp), (bl), (ba), (mp), (ml), (ma), (pl), (pa), (la)}
8. Scan Δ^- the second time. (fl) happened 3 times in D' so it is large. L_2'={(fl)}. The occurrence of the itemsets in Q_2 in Δ^- are: {(fb:2), (fm:3), (fp:2), (fa:2), (bm:1), (bp:1), (bl:1), (ba:1), (mp:2), (ml:3), (ma:2), (pl:2), (pa:1), (la:2).
9. Delete from Q_2 the itemset with occurrence ≥3. And scan the D' second time to check the remaining in Q_2 if they are large or not. Add them into L_2 if their occurrences are larger or equal to 3. Finally the L_2' turns out to be: {(fl), (fb), (fm), (fp), (fa), (bp), (bl), (pl), (pa)}
10. Since there are not large-3 itemsets in the original database D, the rest of the procedure is the same as the apriori algorithm to generate and test the large-3, ..., large-k itemsets.

From the above example, we can see using FUP2 algorithm to find the updated large itemsets is actually an iteration from step 1 to step 5. In the first iteration, C_1 includes all the items (as the step 2 shows). In subsequent iterations, C_k is calculated from L_{k-1}', the large itemsets found in the previous iteration, using the apriori_gen function (as the step 6 shows).

The FUP, FUP2 and MAAP are all generate-and test algorithms. FUP and FUP2 are referred to as k- pass algorithms, because they have to scan the database k times (k is the item numbers of the longest frequent patterns). The MAAP algorithm computes the large itemsets from a certain high level that turns out to save some computation time.

To mine the incremental association rules, all the itemsets in the original database can be divided into four groups shown in figure 2.3-3. F and S represent the large itemsets and small itemsets in original database respectively; F' and S' represent the new large itemsets and small itemsets in the updated database:

1) The itemsets that are both large in the original database and the updated database (F- > F').
2) The itemsets that are large in the original database but not large in the updated database (F->S') .
3) The itemsets that are neither large in the original database nor in the updated database (S->S').
4) The itemsets that were not large in the original database but now are large in the updated database (S->F').

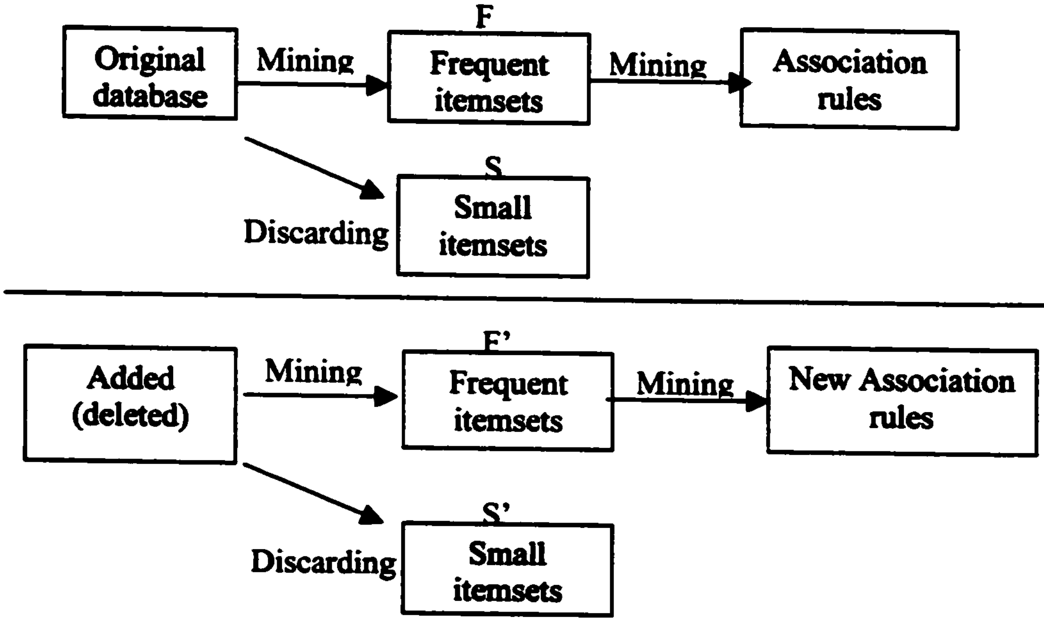

Figure 2.3-3 The Incremental Mining Process

When database is updated, if the itemsets are only in groups 1 and 2, all the algorithms need not scan the old databases and they do not care about the itemsets in group 3. However, if there are itemsets in group 4, the algorithms need to scan the old databases.

FUP and FUP2 calculate the candidate itemsets beginning from the level 1 and they have to calculate in each level, so whenever there are itemsets that are in group 4 in a level, these two algorithms need to scan the old database once each level to determine the candidate support. MAAP calculates the candidate itemsets from higher n level, it scans the old database once on this level if there are itemsets in group 4. Therefore, it need not scan the old database to find the sub-set of these itemsets in the level lower than n. But it needs to scan the old database if there are itemsets in group 4 in the level higher than n. Besides, if there are group 4 itemsets on the level lower than n which have no superset in level n, the MAAP algorithm still needs to scan the old database to determine their support. For example, if MAAP starts compute from level 4, however, there is a group 4 itemset {AB} in level 2 and there are no supersets of {AB} in level 4. The MAAP still has to scan the old database to determine if this itemset is large or not. FP-tree scans the entire database twice and stores the entire frequent patterns in the FP-tree, which is substantially more compact than the whole database. After database is updated, FP-tree needs to scan the old database once to maintain the FP-tree if there are new large itemsets that appear after the database update. Thus, for case 4 above, the FP-tree algorithm will also require at least one scan of the entire database and a scan of the updated part of the database. As the changed part is relatively small in comparison to the entire database and most frequent patterns are already in the tree, it is very impractical and time consuming to scan the old database each time the database is updated.

Next, the analysis of FP-tree is presented to indicate its advantages and limitations for solving the incremental association rules maintenance problem.

3. Mining Incremental Association Rules with FP-tree

We know that each node in the FP-tree is large itself. The FP algorithm scans the entire databases first time to find all the large-1 items in the database and sorts them in descending order. These sorted frequent items are the fundamental of the FP-tree. All the consequent frequent patterns are generated from these frequent items. Because one frequent pattern is always shared by many transactions, the frequent items are sorted in descending order so as to increase the chances of items with higher occurrence being shared.

The FP-tree is the fixed nodes and fixed pattern structure. The numbers of nodes and their orders are stable with respect to the database from which the tree is constructed, and all the frequent patterns of that base are in the tree. It is not necessary to reconstruct the FP-tree each time the database is updated. In [HPY00], it is mentioned how to incrementally update the FP-tree when daily new transactions are inserted into a database. It used the "watermark" concept. The watermark refers to the support upon which the FP-tree structure is formed. In general case, the occurrence of each item in the database is registered and tracked in updates. It is indicated that only when the FP-tree watermark is raised to some undesirable level, the reconstruction of the FP-tree for the new DB becomes necessary. For example, suppose an FP-tree was constructed based on support threshold (watermark) ψ=0.1% in a DB with 10^8 transactions. Suppose additional 10^6 transactions are added in, the frequency of each item is updated. If the highest relative frequency among the originally infrequent items goes up to 12%, the watermark will need to go up accordingly to ψ>0.12% to exclude such items. However, with more transactions added in, the watermark may even drop since an item's relative support frequency may drop with more transactions added in [HPY00]. For example, given a database of 100 transactions with minimum support 30% (minimum occurrence is 30), the FP-tree is constructed based on 30%, so this is the watermark. After 50 new transactions are inserted, the minimum occurrences will be 45. Suppose there is an item r that with occurrences 27 before the transaction insertion and now has 46 occurrences after the insertion, its support 46/150= 30.7% is only a little bit higher than the watermark

30%, it is not necessary to scan the original database to build this r into the FP-tree because this r may become small again after more transaction insertions later.

However, this method loses some information in the FP-tree. For example, the item r mentioned above was not a large item in the original database and thus it was not a node in the FP-tree. After the transactions are inserted, this r becomes large. Although there is no need to rebuild the FP-tree according to the watermark, when we mine the incremental association rules, the information about this r is not in the old FP-tree, how could we mine the patterns to include r?

When the database is updated, because of transaction deletions and insertions, the database has changed, and the frequent patterns may change also. The following situations are possible: firstly, some previously large patterns or nodes may not be large in the new database. Under this situation, all FUP, FUP2 and MAAP do not need to scan the original database. However, FUP and FUP2 scan the changed part of database k-times to compute the candidates on each level and produce many candidate itemsets. MAAP computes from higher level thus it does not scan the changed part all the time to compute each level thereby saving some computation time over FUP and FUP2. FP-tree also need not scan the original database but scans only the changed part once to maintain the FP-tree. The following mining process is fully based on this tree structure and no candidate itemsets generated. Secondly, the previously small patterns or nodes may become large. In this situation, both FUP and FUP2 become clumsy because besides scanning the changed part many times to compute the candidate itemsets, they all need to scan the original database to check the support of these itemsets. Although MAAP can save some computation time over FUP and FUP2, it still suffers the same problem. For the FP-tree structure, those previously large ones in the first situation are easy to deal with, because they are already stored on the FP-tree. But in this situation, for those that were not large before updating, their information is not stored in the existing FP-tree at all. How could we find these patterns? Thirdly, the frequent item order may change as the database is updated, some items with lower order may become higher and those which are higher may be lower finally. Do we need to change their order once the order change happened?

How do we resolve these problems in the second and third situations? Next two ways to deal with these problems are introduced.

3.1 Using Database Tree (DB-Tree)

We want to use a structure which stores the full information on a database to benefit the update of that database. The FP-tree is easy to get from this structure in order to execute the mining process.

FP-tree does not fully represent a database. So, if database is updated, this FP-tree may not fully represent all frequent patterns anymore, and there is need to update the FP-tree as well.

We have discussed before that it is not practical to reconstruct FP-tree each time the database is updated. If the minimum support of a database is 1, its FP-tree is just a compact version of the original database. To distinguish it from the FP-tree of higher support, we call the tree based on minimum support 1 the **DB-tree**. This DB-tree stores all the patterns of the database. When the transactions are deleted or inserted, the changed transaction instead of the whole large original database is scanned. Although this DB-tree also is more compact than the database itself, it may be undesirably larger than the FP-tree and mining on this DB-tree is impractical. Fortunately, this DB-tree includes the FP-tree inside it. So, if we can project the FP-tree from this DB-tree, we can mine the frequent patterns based on the FP-tree instead of large DB-tree. The DB-tree is just used for maintaining the frequent patterns when there is an update to the database since it keeps information regarding not only previously large itemsets, but also previously small itemsets. Thus, it eliminates the need to scan the database when certain types of updates occur. The DB-tree method is introduced next.

3.1.1 Constructing DB-tree

The original database D is the same as the one in Figure 2.2-1 and the minimum support s=60%.

First, scan D to get all the occurrences of all the items. Sort them in descending order of support. Next, use these occurrences to construct the DB-tree. The construction procedure is similar to the construction of the FP-tree, but it includes all the items instead of only the frequent 1-itemsets. Like constructing the FP-tree, it also needs to scan the database twice to construct a DB-tree. When constructing a DB-tree, the first scan is to count all the items and sort them in descending order, FP-tree only sorts the large items whereas the DB-tree sorts all the items. Second scan is to go through each item in each transaction, the FP-tree only puts the large items into the tree while the DB-tree puts each item in the tree structure. From the database scanning aspect, DB-tree construction is the same cost as the FP-tree construction because they both scan the database twice. However, DB-tree has more branches and more nodes than the FP-tree, so the DB-tree needs larger storage than the FP-tree, especially when the amount of items are quite large, e.g. 10,000. The lower the minimum support, the nearer the cost of constructing a DB-tree is to the cost of constructing an FP-tree because the low minimum support will cause large amounts of frequent items allowing the FP-tree to cover nearly as many items as the DB-tree. Although the DB-tree is larger than FP-tree, since almost all the items share common paths in the structure, this DB-tree is still much smaller than the database itself. The factor affecting the size of the DB-tree is its item combination. If more items share the branches commonly, the smaller the size of the DB-tree is. Comparing the size of the FP-tree to the size of DB-tree, it can be known that FP-tree constructed based on different minimum supports has different sizes. However, the DB-tree size remains same. So, the lower the minimum support, the more compact the FP-tree is, compared to the DB-tree. The size of an FP-tree is bounded by the overall occurrences of the frequent items in the database, and the height of the FP-tree is bounded by the maximal number of frequent items in any transaction in the database. The size of a DB-tree is bounded by the numbers of all items in the database, and the height of the DB-tree is bounded by the size of the longest transaction in the database. The worst case of the DB-tree is when there is no item combination. Each branch represents a transaction. Of course in this situation, there will be no frequent patterns in the database thus there no FP-tree exists and the size of the DB-tree is same as the size of database. Fortunately, there must be some combinations in a database, so the size of DB-tree is always smaller than the database itself. The size of DB-tree is nearly equal to the database size in the worst case and nearly to the FP-tree size in the best case. For example, given a database with 20 items, 10 rows. Suppose the average length of the transaction is 5, 10 items are frequent, the maximal length of the frequent pattern is 4, the maximal occurrence of the frequent item is 6 and the minimum occurrence of the frequent items is 3 (30% is the minimum support). Assume each item occupy 4 byte. Then the size of the database is 4(bytes) × 5(length) × 10(rows) = 200bytes. The width of the FP-tree is less or equal to 6 and the height of the FP-tree is less or equal to 4. So the size of the FP-tree is less or equal to 4(bytes) × 4 (height) × 6 (width) = 96 bytes. As DB-tree constructs all the items into the tree structure, it includes not only frequent patterns (FP-tree) but also the non-frequent patterns. The size of DB-tree is the size of the FP-tree + the size of the non-frequent patterns. In the best case, each non-frequent item only occupies one node in the DB-tree. In this example, the DB-tree size is FP-tree size + 10 (number of non-frequent items) × 4(bytes), that is 96 + 40 =136 bytes. In the worst case, each non-frequent item will occupies as many nodes as possible. In this example, since the minimum occurrence of the frequent item is 3, the maximal occurrence of the non-frequent item is 2. And there are 10 non-frequent items. So, the largest size of the non-frequent patterns is 4(bytes) × 2 ×10 = 80 (bytes). Then we know that the in this example the largest size of DB-tree is 96 (FP-tree size) + 80(non-frequent patterns size) = 176 bytes. The complexity of constructing a transaction ***DTrans*** into the DB-tree is **O** (|***DTrans***|), where |***DTrans*** | is the number of items in the transaction ***Trans***.

The DB-tree based on database in Figure 2.2-1 is like Figure 3.1-1.

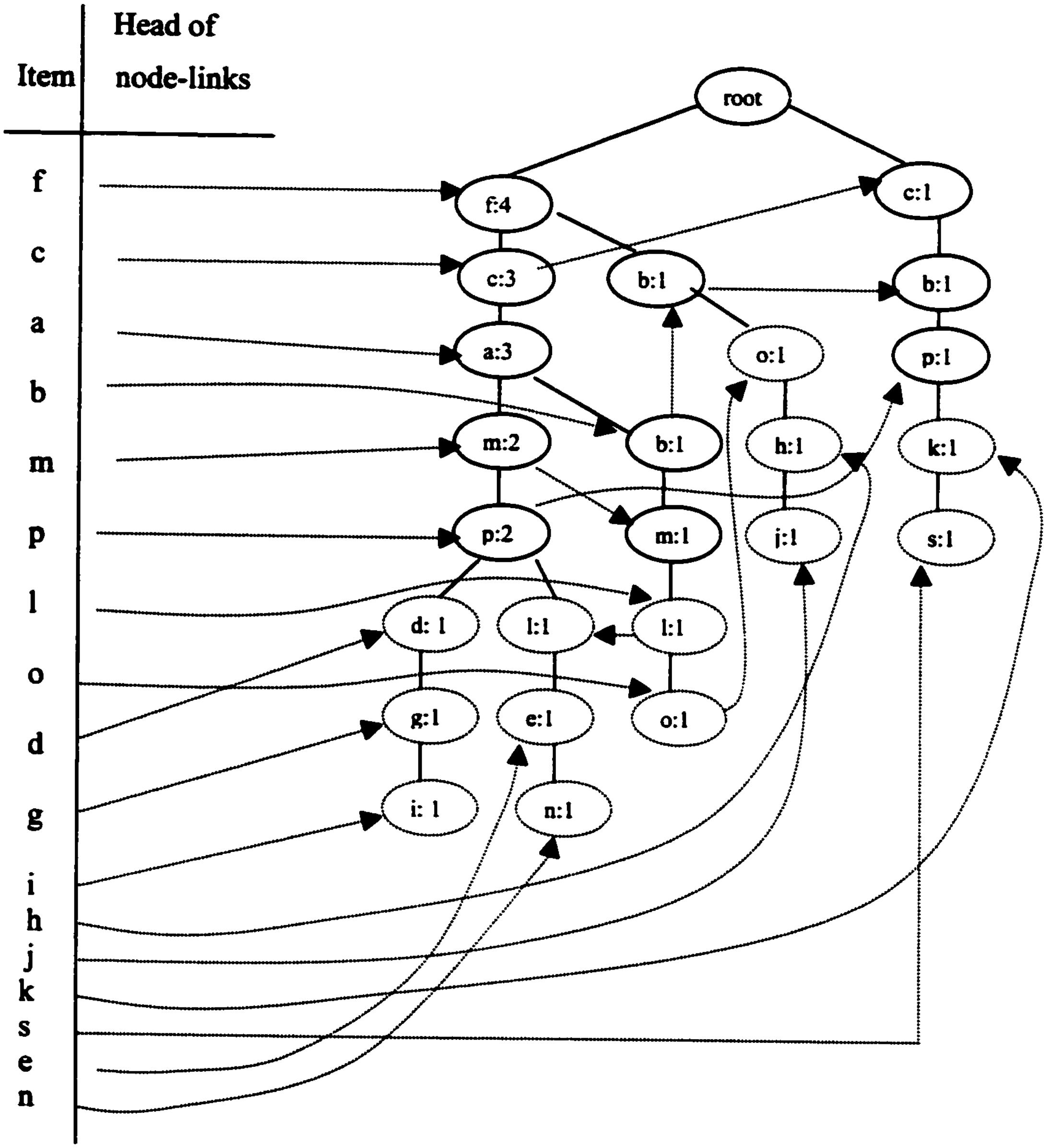

Figure 3.1-1 DB-tree

From the figure, it can be seen that DB-tree includes all the patterns of the database. In this figure, solid circles indicate large items and dotted circles indicate small items. On the top of the DB-tree is the FP-tree. When mining the frequent patterns, we can project FP-tree part from the DB-tree, which is substantially compact more than the database and smaller than the DB-tree that may fit the main memory. And then the mining task is fully based on this FP-tree.

An example that uses this DB-tree to mine the incremental association rules is given.

3.1.2 Mining Incremental Rule with DB-tree

Suppose two transactions, transactions 600 and 700 are inserted into D. The minimum support is still 60%. The updated database D' is as in Figure 3.1-2

D'

	TID	Items Bought
	100	f, a, c, d, g, i, m, p
	200	a, b, c, f, l, m, o
	300	b, f, h, j, o
	400	b, c, k, s, p
	500	a, f, c, e, l, p, m, n
Δ^+	600	a, c, f, m, g, o, l
	700	f, b, a, c, l, m, o, n

Figure 3.1-2 The updated Database After Transaction Insertion of DB-tree Algorithm

By using the DB-tree, the update will not cause any database scan in any situation (FUP, FUP2 and MAAP algorithms need to scan the original database many times when there are itemsets that were not large before but now turn out to be large in the updated part, the original FP-tree needs to scan the original database once to update the FP-tree when there are new items that become large). The DB-tree just scans the additional transactions (transaction 600, and 700) only once to update the DB-tree. When the new transactions share the same branch with the ones in the original database, it just increases the count of each item, otherwise it adds the new branch into the DB-structure. At the same time the occurrence of each item in the additional transactions are also obtained. The occurrences after insertion become <f:6>, <c:6>, <a:5>, <b:4>, <m:5>, <p:3>, <l:4>, <o:4>, <g:2>, <n:2>, <j:1> .... The DB-tree after the two transactions are inserted is shown in Figure 3.1-3. The minimum occurrence in the updated database D' would be 4.

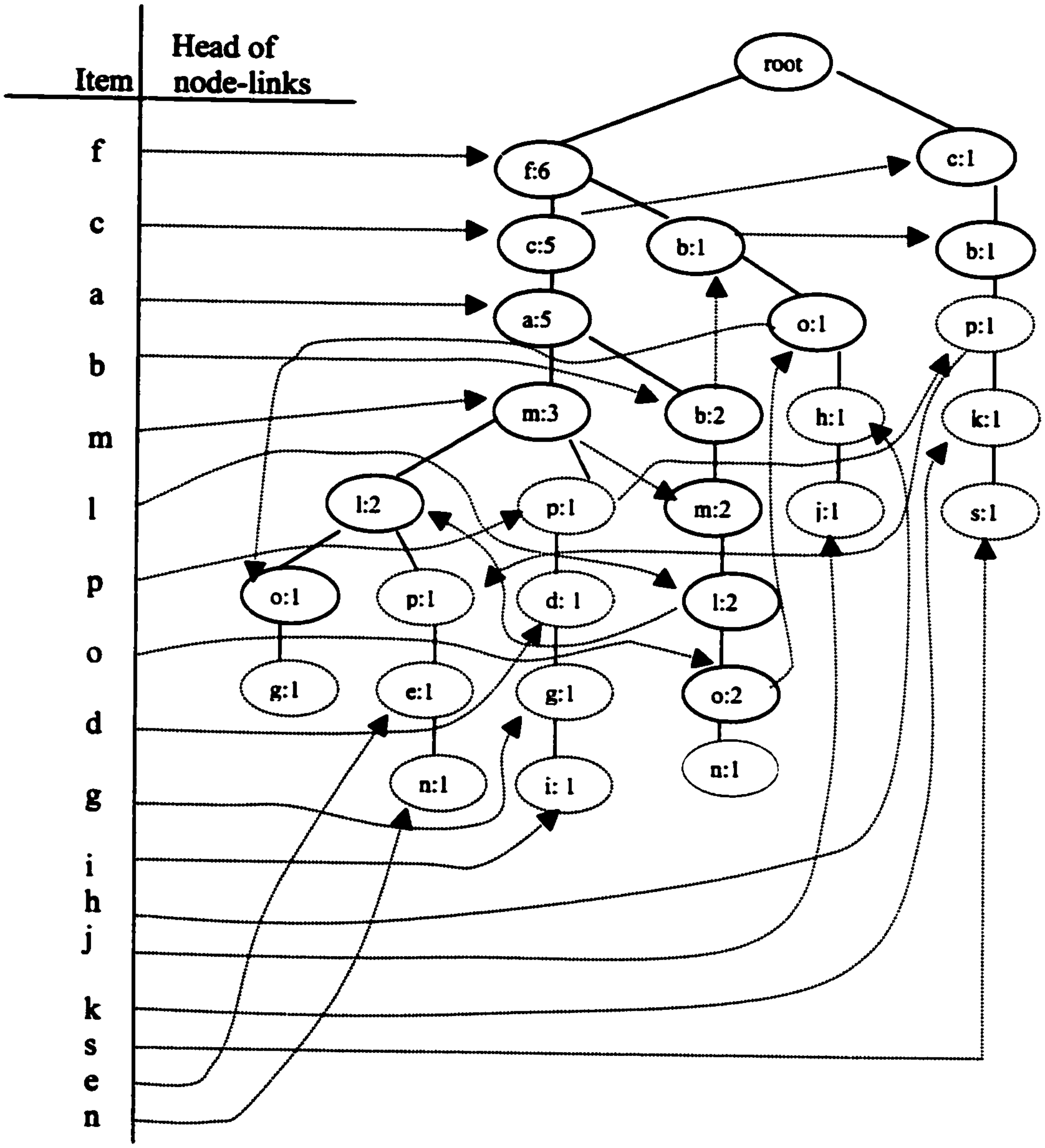

Figure 3.1-3 DB-tree After Inserted Transactions

When mining the updated association rules in the updated database, we need to project the FP-tree out from the updated DB-tree. We note that the frequent item order changed and item p is no longer large any more in D'. Do we need to rearrange the FP-tree? In another words, do we need to delete the node p from the FP-tree and restrictedly arrange the FP-tree structure according to their descending order? We notice that the updated part is usually dramatically smaller than the original database. The FP-tree is based on the original database therefore most amount of frequent patterns are already stored in the FP-tree when database is updated. The changed part would have comparatively fewer amounts of frequent patterns. Even the changed transactions affect the items' order, since most of the frequent patterns in the FP-tree are based on the old order, only small amounts of patterns (in updated part) follow the new order, it is not necessary to change the order strictly each time the order is changed. Otherwise the order change would cause a lot of overhead before mining. We don't care about the order change among the frequent items and among the small items. For example, the occurrences of item b becomes 4 and m becomes 5, but they are still large items and all their patterns are already in FP-tree, so there is no need to rearrange their orders. Only when small items become large or large items become small, it is necessary to rearrange the order. That means that if there is a small item that becomes large, it needs to be placed in the FP-tree part of the DB-tree; if there is a large item is now small, it needs to be lowered down out of the FP-tree part in the DB-tree. For example, the previous large item p becomes small and meanwhile the previous small item l becomes large, so their order changed.

To project FP-tree from DB-tree, start from the root, following each branch if its next node is large till there are no large items in a branch. For example, start from root, first it encounter f. Since f is large it still continue traversing the node under f in the same branch, the following nodes are c, a, m, l, o. The node g after o is small, so it stops at o. So far, one branch of the FP-tree is obtained. This process is repeated after all branches from root are traversed. The cost of projecting an FP-tree is equal to cost traversing FP-tree once.

The projected FP-tree of D' from the updated DB-tree is shown in Figure 3.1-4.

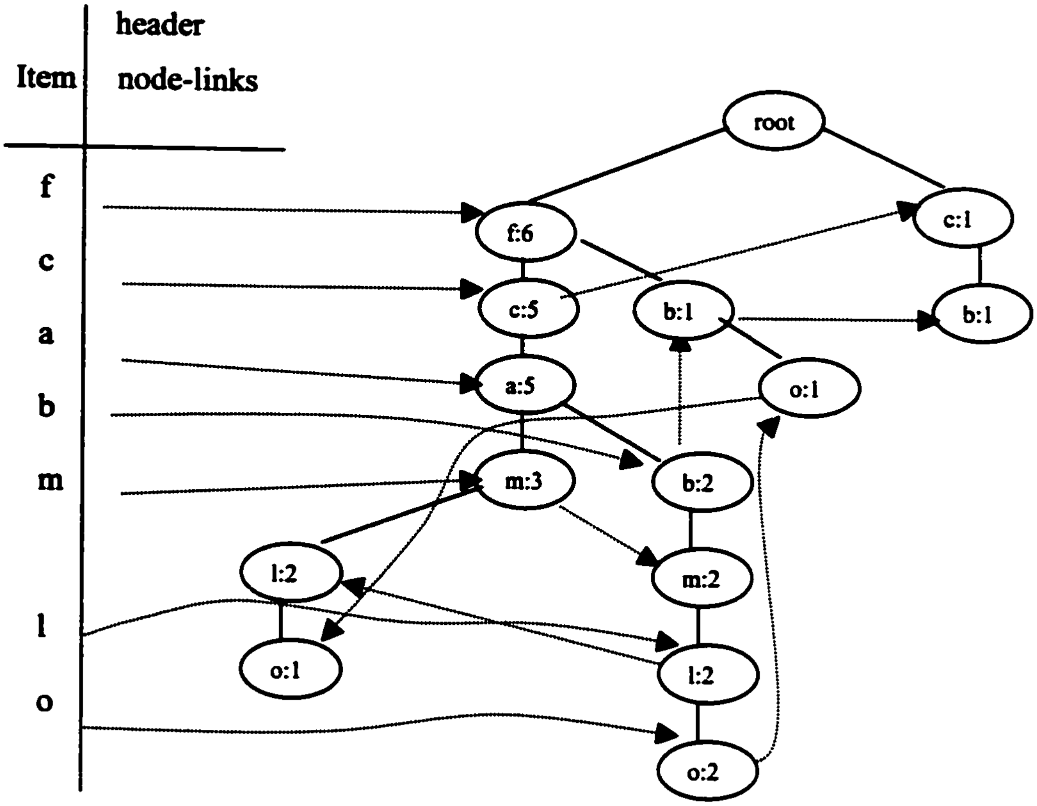

Figure 3.1-4 FP-tree of The Updated Database D'

The mining process is fully based on this FP-tree, and it is similar to the one in [HPY00]

Next we introduce when the transactions are deleted from the database, and how to mine the incremental rules with DB-tree in this situation. Suppose the deletion happened in database in Figure 3.1-2. If transactions 500 and 600 are deleted, the updated database D' is in Figure 3.1-5.

D'

TID	Items Bought
100	f, a, c, d, g, i, m, p
200	a, b, c, f, !, m, o
300	b, f, h, j, o
400	b, c, k, s, p
700	f, b, a, c, l, m, o, n

Δ^-

500	a, f, c, e, l, p, m, n
600	a, c, f, m, g, o, l

Figure 3.1-5 The Updated Database After Transactions Deletion of DB-tree Algorithm

The procedure to update the DB-tree in figure 3.2.1-3 is similar to that for insertion. The only difference is that deletion never would introduce new branches to the DB-tree. It only follows the branch and decreases the count of the nodes. By only scanning the deletion part once, the occurrences of the items are obtained. As we mentioned above in the transaction insertion, if the order among the frequent items and among the small items changed, we do not rearrange their order in the tree structure because it causes lots of overhead and will not affect frequent patterns. Only if there are small items becoming large or large items becoming small, their order in the tree need to be rearranged to ensure the FP-tree part of the DB-tree consists of all large items. The DB-tree is updated and then projects the FP-tree from the DB-tree and mining the updated association rules depends on the FP-tree.

From the above example, it can be seen that using the DB-tree to mine the incremental association rules has the following advantages: (1) no scanning of the original database is needed even when the small itemsets become large after updating the database, yet the original FP algorithm has to scan the original database to reconstruct the FP-tree in this situation, (2) the mining process is based on the FP-tree, the compact structure instead of generating many candidates, (3) it only scans the updated transactions once, yet the FUP2 need to scan on each level and MAAP also needs to scan many times.

The incremental association rules mining with DB-tree follows the following steps:

1) Construct the DB-tree using all items in the database at the beginning.
2) After the DB-tree is constructed, during the later database update just scan the updated part once to update the DB-tree.
3) Project the FP-tree from the DB-tree according to the large items.
4) Using the FP-growth procedure in the original FP-tree algorithm in [HPY00] to mine the frequent patterns in the FP-tree in order to find the frequent patterns.

Figure 3.1-6 and 3.1-7 give the procedures to update the DB-tree and project the FP-tree from DB-tree.

```
Input: original DB-tree, counters of all items, changed part of transaction, minimum support s,
transaction number of original database d, transaction number of changed part d', small items
group S, S'; large items group L, L'
Output: the updated DB-tree.
Begin
  if insertion {
    while (reading each inserted transaction){
      increase count of each node by 1 in the DB-tree or add the new branch;
      for each item j in transaction
        $counter_j = counter_j + 1$;
    }
    for each item j
     do{
         if $counter_j \geq (d+d') \times s$
            put j into L';
         else
            put j into S';
     }
    if $(S' \cap L) \neq \emptyset$ or $(S \cap L') \neq \emptyset$ {
      for all item $j \in (S' \cap L)$ or $j \in (S \cap L')$
         change their order in DB-tree;
    }
  }

  if deletion {
    while (reading each deleted transaction){
      decrease count of each node by 1 in the DB-tree or delete the node
      if its count is 0;
      for each item j
        $counter_j = counter_j - 1$;
    }
    for each item j
     do{
         if $counter_j \geq (d-d') \times s$
            put j into L';
         else
            put j into S';
     }
    if $(S' \cap L) \neq \emptyset$ or $(S \cap L') \neq \emptyset$ {
      for all item $j \in (S' \cap L)$ or $j \in (S \cap L')$
         change their order in DB-tree;
    }
  }
end
```

Figure 3.1-6 Algorithm for updating the DB-tree

```
Input: DB-tree, large items group L,
Output: FP-tree,
Begin
    Start from root;
    While (branch of root not null){
            Node n = next node;
            If (n $\in$ L)
            Do { read out node n of a branch;
                  n = next node;
            } while ( n $\in$ L)
            go to next branch;
    }
end
```

Figure 3.1-7 Algorithm for Projection of FP-tree

3.2 Mining Incremental Association Rules Using Potential Large Itemsets (FPUP)

In this section, we introduce a method that is not like the one in 3.1 that stores the tree structure, DB-tree, of the entire database. Here, when mining the original database, we also try to find those nodes that are not large at present but have high probability of being large after the database update, and keep these nodes in the FP-tree. These beforehand-stored nodes would benefit the incremental association rules mining after database is updated.

When updating the database, just follow the FP-tree to update each node's count (deletion or insertion) or add new branches into the FP-tree (insertion). After updating (deletion or insertion), even the nodes order change, or some nodes which were large before now become small, the structure need not be reconstructed because all the frequent patterns are still in the FP-tree. We still mine the frequent patterns based on the same FP-tree structure. However, for those items that were small but now become large after database update, their information are not stored in FP-tree and the question is on how to discover their patterns in the FP-tree?

The small items in the original database can be divided into two groups. One consists of those that are not large now but may be large after the database update, which we call potential large items, P. Another group includes those that are small now and with high possibility of still being small after update of the database, M. We can give a tolerance t when constructing and reconstructing the FP-tree, this t is equivalent to the wartermark given in [HPY00] because the FP-tree is constructed based on t instead of the minimum support s. However, the watermark in [HPY00] means the support that most mining processes that are based on over a period of time. For example, if the 60% mining processes use the minimum support of ≥20, then 20 is the watermark and it only indicates when there is a need to reconstruct the FP-tree. We choose the t with lower value than the minimum support here to benefit the incremental association rules mining. For those items with the support s, where t≤ s≤ minimum support, we construct them in the FP-tree, so the pattern including these potential large items are also stored into the FP-tree. They have the lower order than the large items and their positions are near the leaves of the tree. The reasonable value of t will be examined later in the experimental implementation. When mining the frequent patterns of the original database, we don't care about these items since they are small then. However, after updating the database, these potential large items play an important role in mining. Because their patterns are already stored, we do not have to scan the original database and need not reconstruct the FP-tree to find their patterns. Mining updated frequent patterns are fully based on the updated FP-tree.

The potential large items group P may include all items that change from small to large after the update. In fewer cases, it is possible that the items in group M (items with supports < t) may turn out to be large after update, or maybe these items become potentially large after update. However, their information is not in FP-tree. If the items in M only turn to be potentially large, we simply not consider them at the moment. Only when there are items in M that turn out to be large, there is a need to scan the old database once to construct those items in M now are either large or potentially large into the FP-tree. The original FP-tree needs to scan the old database once during each incremental mining to update its frequent patterns in the FP-tree if items become large after the database update. By using the potentially large items, the old database scan would barely happen or happen very less time compared to the original FP-tree.

Next we will use an example to explain this. We still use the database in figure 2.2-1 as the original database D. The minimum support minsuppt=60% (3 occurrences base on 5 transactions), the tolerance t=40% (2 occurrences based on 5 transaction). After scanning the entire database once, we find the occurrences of each item. The large-1 items L_1={(f:4), (c:4), (a:3), (b:3), (m:3), (p:3)}, the potential large items PL_1={(l:2), (o:2)}. M={ d, g, I, h, j, k, s, e} and their TID. So the FP-tree is constructed like Figure 3.2-1.

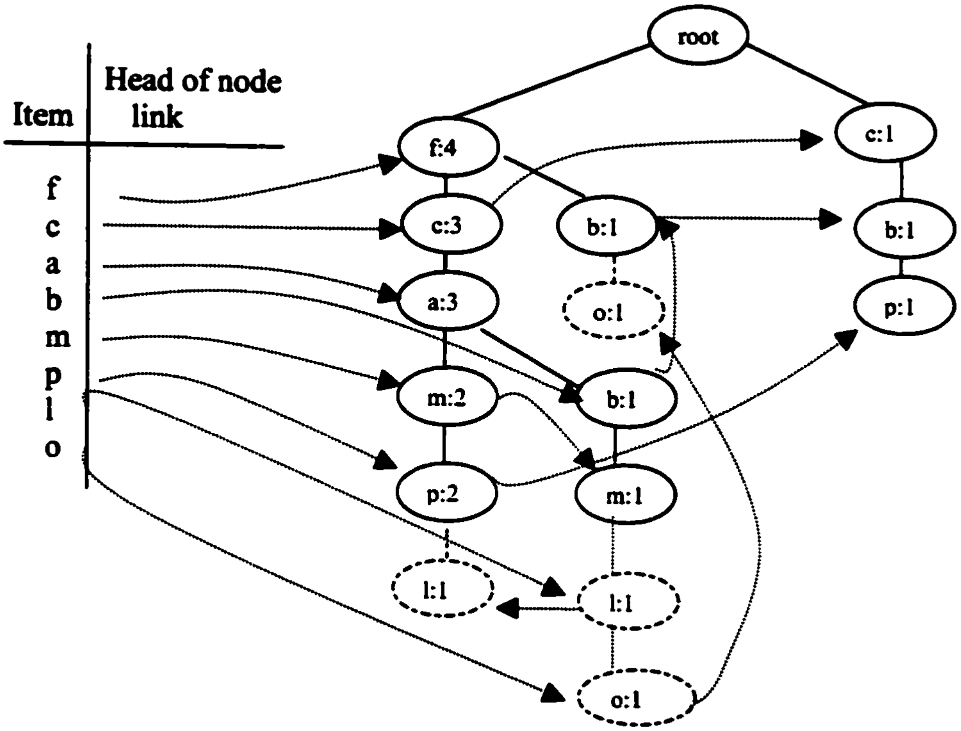

Figure 3.2-1 The FP-tree of FPUP Algorithm

The complexity of constructing a transaction ***Trans*** into the FP-tree of FPUP algorithm is O (|***PTrans***|), where |***PTrans*** | is the number of frequent and potential large items in the transaction ***PTrans***. When we mine the frequent patterns of the original D, we need not care about the two items with the dotted circle. The mining procedure is the same as it is described in section 2.2.

Next let some transaction be added into D, the updated database D' is like figure 3.2-2.

D'

	TID	Items Bought
	100	f, a, c, d, g, i, m, p
	200	a, b, c, f, l, m, o
	300	b, f, h, j, o
	400	b, c, k, s, p
	500	a, f, c, e, l, p, m, n
Δ^+	600	a, c, f, m, g, o, l
	700	q, f, b, a, c, l, w,o

Figure 3.2-2 The Database after Transaction Insertion of FPUP

Scanning the Δ^+ once we can calculate all the occurrences of each items in L_1 and PL_1. Therefore decide the updated L_1' and PL_1'. Next, scan the Δ^+ second time to update the FP-tree according to the new L_1'. The updated FP-tree is like Figure 3.2-3. Note, in this step, only if there are any item in M that now become large, select the transactions according TID of all the items with occurrences same or larger than t, traverse these transactions and construct these items as nodes (large and potential large) on FP-tree. If there is an item that only becomes potentially large, there is no need to spend time to construct it into FP-tree now. Once it becomes large or there are some other items that become large, all these items with frequency ≥ t will be constructed into FP-tree together at the same time. This will save some computation time. If there are items that were in L_1 or in PL_1, but now are neither in L_1' nor in PL_1', delete this node from the FP-tree and put this item into group M.

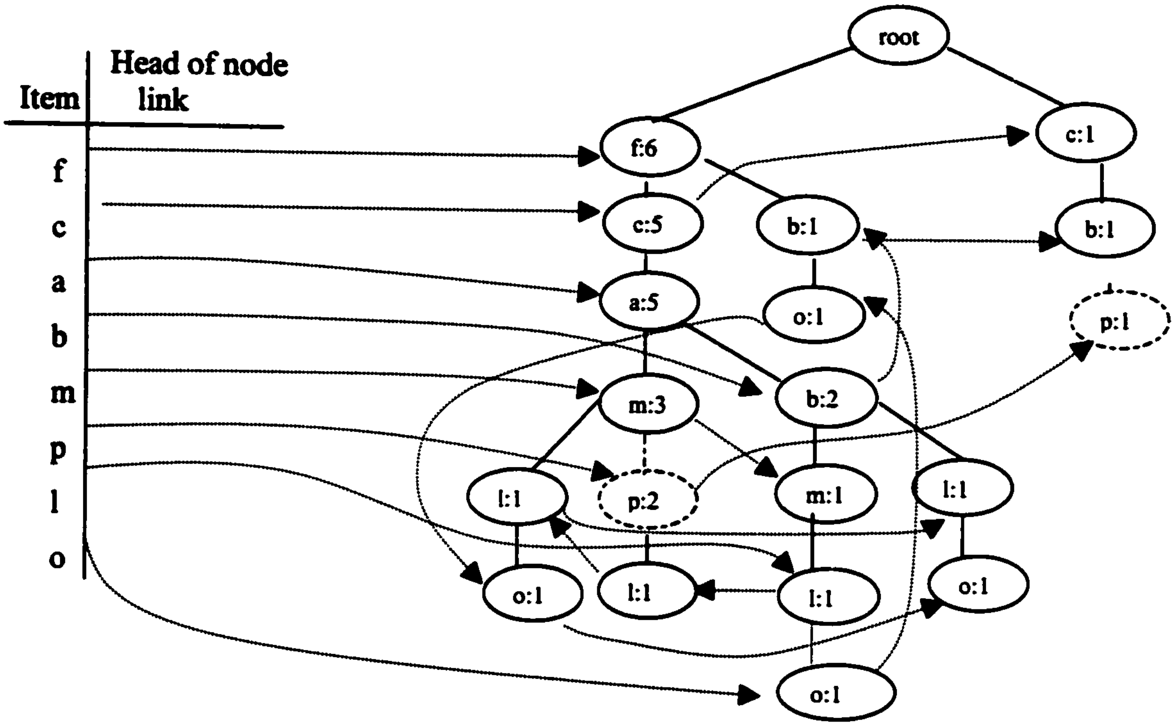

Figure 3.2-3 Updated FP-tree after Transaction Insertion

After the first scan of Δ^+, we know the occurrences of each potential and previously large items. The minimum occurrence of the items in D' is 60% multiplied by 7, that is 4, and we (have the frequent items and their occurrences (f:6), (c:6), (a: 5), (m:4), (b: 4), (p: 3), (l:4), (o: 4).) Therefore, item p is not large in the new D' and l and o turn out to be large finally. We can compute p's potential occurrence to see if p is still a potentially large item or not at next update. Note that t=40%x7=2.8, since 3>2.8, so p is potentially large now. If p is not potentially large either, it would be deleted from the FP–tree. In this example, there are no items in M that become large at present update.

The mining of updated frequent patterns now is fully based on the updated FP-tree in Figure 3.2-2. The procedure is the same as in [HPY00]. Follow each node link to find the conditional pattern bases for each node and construct conditional FP-tree for each node, perform this procedure recursively until find all patterns are found. Note for those nodes that are not large any more, but still in the FP-tree structure, we do not need to find conditional pattern base and construct the conditional FP-tree for it any more. And for the conditional pattern base which contain this present small node, just simply delete it from the conditional pattern. For example, the conditional pattern base of node l are:{<(fcamp):1>, <(fcam):1>, <(fcabm):1>, <(fcab):1>}, for pattern (fcamp), because p is not large any more, so delete it, the pattern turns out to be (fcam).

An example of deletion is presented next. Suppose transactions with TID 500 and TID 600 are deleted from the database in Figure 3.2-2. The updated database is in Figure 3.2-4. Similar to the transaction insertion, first scan the deletion transactions Δ^-, since the occurrences of items in the L_l and PL_l are already known, we can get the updated occurrences of each item by simple calculation. Also the updated PL_l is obtained. The updated L_l'={(f:4), (c:4), (a:3), (b: 4), (o:3)}, the updated PL_l'={(m:2), (p:2), (l:2)} and M' doesn't change. Note that when deleting the transactions, because all the frequent patterns are already stored in the FP-tree, we do not need to scan Δ^- the second time, during the first scan, we can follow the path to find and reduce the count of each traversed node. For the nodes with the counts '0', we simply delete these nodes as these nodes infer no information any more.

D'

TID	Items Bought
100	f, a, c, d, g, i, m, p
200	a, b, c, f, l, m, o
300	b, f, h, j, o
400	b, c, k, s, p
700	q, f, b, a, c, l, w,o

Δ^-

500	a, f, c, e, l, p, m, n
600	a, c, f, m, g, o, l

Figure 3.2-4 The Database after Transactions Deletion of FPUP

The FP-tree after deletion is as shown in Figure 3.2-5 (the dotted circle indicates the nodes that are potentially large). Note that the nodes in the solid frame now have counts of zero and they can be deleted.

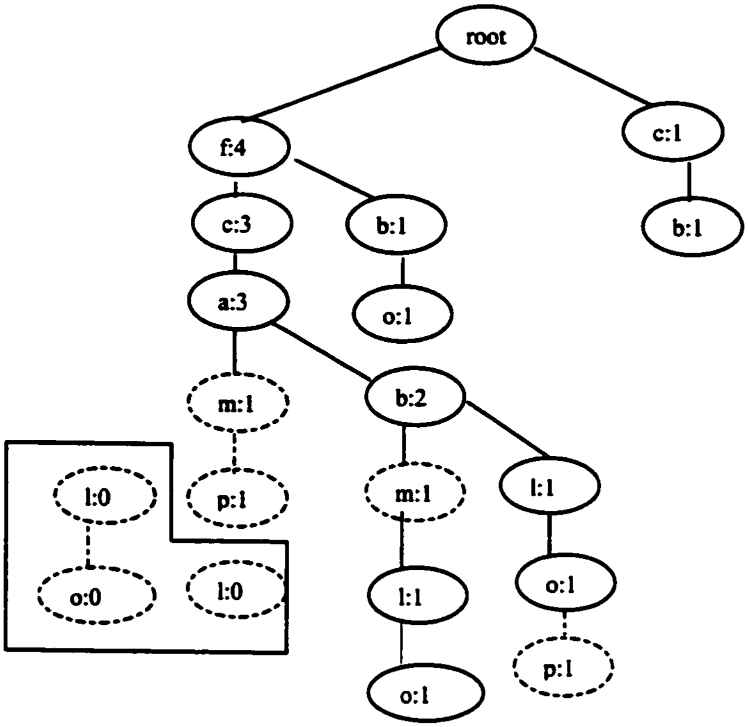

Figure 3.2-5 The Updated FP-tree after Transaction Deletion

The mining algorithm on this FP-tree is similar to the one introduced in section 2.2. Note for those potential large items (doted nodes), when constructing the conditional pattern base just simply ignore them.

Similarly, mining the database with some transactions modified is based on the existing FP-tree. We just need to scan the modified transactions once to update the FP-tree structure, reduce the existing nodes count or delete the nodes if their counts are '0'or increase the nodes counts or add new branches or nodes (if they are not in the previous FP-tree). After the FP-tree is updated, the mining method is the same as the one in [HPY00].

The complete FPUP algorithms for transactions insertion update and transactions deletion update are in Figures 3.2-6a and 3.2-6b:

```
Insertion:
Input: FP-tree; L, PL, M; original database DB and its transaction number d, updated part
of database db and its transaction number d'; minimum support s, potential tolerance t.
Output: updated frequent patterns
Begin
    Scan db first time, put each item into the new L', PL' and M';
    For all item (($i \in$ L-L') or ($i \in$ PL-PL')) and ($i \in$ M')
        Delete i from FP-tree;
  If ($M \cap L' \neq \emptyset$) {
      For all ($i \in (M \cap L')$) and ($i \in (M \cap PL')$)
          scan the old DB and insert i into FP-tree;
  }
  Else if ($M \cap PL' \neq \emptyset$){
     For all ($i \in (M \cap PL')$
          {Put i into M';
           remove i from PL';}
  }
  L=L';
  PL=PL';
  M=M';
  For all $i \in$ L or $i \in$ PL
     Do {insert i into FP-tree while scan the db second time;}
 FP-growth(FP-tree, pattern $\alpha$);
End
```

Figure 3.2-6a FPUP Mining Algorithm for Transaction Insertion

```
Deletion:
Input: FP-tree; L, PL, M; original database DB and its transaction number d, updated part
of database db and its transaction number d'; minimum support s, potential tolerance t.
Output: updated frequent patterns
Begin
  While scanning the db{
      decrease the occurrences of each node in FP-tree according to the corresponding
      transaction;
      calculate the occurrence of each item;
  }
  for all items i
  do {
        if occurrence (i)$\geq$ s
              put i into L';
        else if occurrence s$\geq$(i) $\geq$t
              put i into PL';
        else
              put i into M';
   }
  For all item (($i \in$ L-L') or ($i \in$ PL-PL')) and ($i \in$ M')
       Delete i from FP-tree;
  If (M $\cap$ L'$\neq \emptyset$) {
    For all ($i \in$ (M$\cap$L')) and ($i \in$ (M$\cap$PL'))
       scan the old DB and insert i into FP-tree;
  }
  Else if (M $\cap$ PL'$\neq \emptyset$){
    For all ($i \in$ (M$\cap$PL')
       {Put i into M';
        remove i from PL';}
  L=L';
  PL=PL';
  M=M';
  FP-growth(FP-tree, pattern $\alpha$);
End
```

Figure 3.2-6b FPUP Algorithm for Transactions Deletion

So far, the new idea of applying the FP-tree structure to incremental mining of association rules technique (we call it FPUP) has been introduced. By predicting some potentially large item, when the update happens to the original database, the FPUP need not scan the original database. Whereas the Apriori-like algorithms have to scan the original database k times and the original FP-tree algorithm need to scan the old database once whenever there is a small item that becomes large, FPUP saves more computation time than original FP-tree and Apriori-like algorithms. With insertion FPUP scans the inserted transactions twice, with deletion and modification, the FPUP scans the deleted transactions only once to update the FP-tree structure. After the FP-tree is updated, the following mining steps are fully based on this rather compact FP-tree like the algorithm introduced in [HPY00].

4. Experimental Evaluation and Performance Analysis

In this section, a performance comparison of DB-tree, FPUP algorithms with existing algorithms original FP-tree and Apriori is introduced. All these four algorithms are implemented in order to compare their performance.

All the experiments are performed on a 733MHz P3 PC machine with 256 megabytes main memory. The operating system is Mandrake Linux. All the programs are written in C++. All the reports on the runtime of FP-tree, DB-tree and FPUP exclude the original construction of tree structure from the original database because this is not a part of incremental mining process. The tree constructions later on in the incremental mining processes are included if the FP-tree has to be re-constructed in order to find an answer.

The synthetic data sets which are used here were generated using the program in [AS94]. The transactions mimic the transactions in the retailing environment. The parameters shown below are used to generate the data sets.

|D|: Number of transactions

|T|: Average size of the Transactions

| I |: Average size of the maximal potentially large Itemsets

| L|: Number of maximal potentially large itemsets

N: Number of items

The resource code for generating the synthetic data sets can be downloaded from : http://www.almaden.ibm.com/cs/quest/syndata.html.

Four experiments were conducted comparing the performance of DB-tree, FPUP, original FP-tree and Apriori.

4.1 Experiment 1: Execution Time for Different Algorithms

This experiment is to use a fixed size of database and fixed size of inserted (deleted) transactions, given different minimum support to compare the execution time of DB-tree, FPUP with FP-tree and Apriori. The size of the original database is one hundred thousand records. It is denoted as |D|=100,000, the average size of transactions (T) is 10. It is denoted as |T|=10, this means the average number of items in each transaction. There are totally 1000 different items in the whole database, that is |N|=1,000. The average length of maximal pattern is 6, denote as |I|=6. Assume the size of inserted dataset is 10,000 records, the size of deleted dataset is 10,000 records. These experiment conditions can be expressed as T10.I6.D100K+15K-15K with 1000items. The minimum support is increased from 0.1% to 6%. The experimental result is shown in Figure 4.1-1 and Figure 4.1-2. In this experiment, the tolerance t of FPUP algorithm is chosen to be between 80% and 90% of minimum support. That is, if the minimum support is 1, then the tolerance is 0.9 or 0.8.

Algorithms	Runtime (in seconds) at Supports of (%)										
	0.1	0.2	0.3	0.5	0.7	1	2	3	4	5	6
Apriori	—	—	—	—	—	—	—	258	43	27	22
FP-tree	54	51	49	43	39	33	19	9	6	4	4
DB-tree	44	44	44	43	43	43	42	42	42	40	40
FPUP	42	40	38	34	30	26	13	6	3	3	3

Figure 4.1-1 Experimental Data- Execution Time (seconds) for Different Algorithms

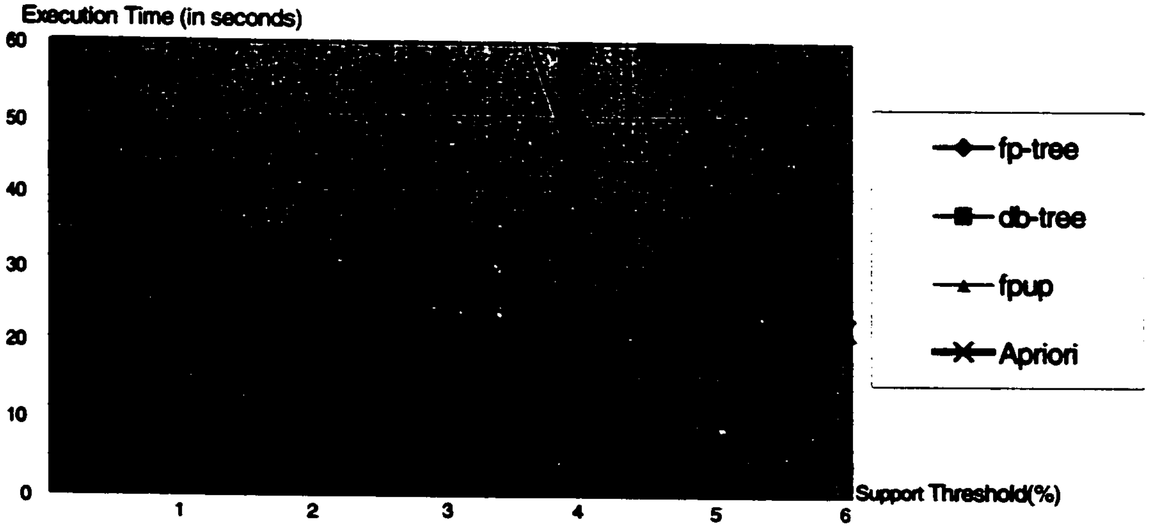

Figure 4.2 Execution Times On Different Support Level

Figure 4.1-2 Execution Times on Different Support Level

By observing the result, we can see that apart from the DB-tree, all the execution times decreases as the minimum support increases. For the Apriori algorithm, this is because the higher the minimum support, the less the candidate sets that are to be handled. For the FPUP and FP-tree structure, as the minimum support increases, the large 1-items become fewer which results in the smaller FP-tree, and also the larger the minimum support is, the less the chance of small items becoming large as the database changes. So it is with very small possibility to rescan the entire database to reconstruct the FP-tree. However, 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%). This is because the DB-tree is constructed based on each item in the database, so its size is quite large and stable. It can demonstrate although DB-tree omits the step to rescan the entire database when database is changed, the cost of maintaining the large scale of DB-tree is considerably large. Comparing DB- tree to FP-tree, when the minimum support is very small, the FP-tree more quickly rescans the entire database as the previously small items become large after the database is changed.

The FPUP algorithm shows better results compared to the other algorithms. It can be seen that it always uses less execution time than the FP-tree. By using the proper tolerance t, the appropriate size of the tree structure can be used in the FPUP which will not cost a lot to maintain as the DB-tree does. Also, it avoids scanning the entire database as the original FP-tree algorithm does. The proper value of t will be examined later in this chapter.

4.2 Experiment 2: The Effect of Different Changed Transaction Size Testing

This experiment still uses T10.I6.D100K with 1000items. The numbers of changed transactions are varied (from 10k to 50k). Comparing FP-tree, DB-tree and FPUP algorithms to see the effects the different size of changed transactions result in. Meanwhile, we choose three different minimum supports which have values that are low (0.5%), medium (1%) and very high (6%) to evaluate the performance of three algorithms.

First, the very low minimum support 0.5% is chosen. Figure 4.2-1 and Figure 4.2-2 show the execution time of three algorithms as the size of changed transactions increase. Figure 4.2-3 and 4.2-4 are showing the result with the minimum support 1%. Figure 4.2-5 and Figure 4.2-6 are the results with the minimum support 6%.

Algorithms	Runtime (in seconds) at Different Changed Transaction Size				
	10K	20K	30K	40K	50K
FP-tree	43	69	100	125	158
DB-tree	43	68	98	124	154
FPUP	34	54	71	89	111

Figure 4.2-1 Experimental Data- Execution Times (seconds) with Different Changed Transaction Size on Minimum Support 0.5%

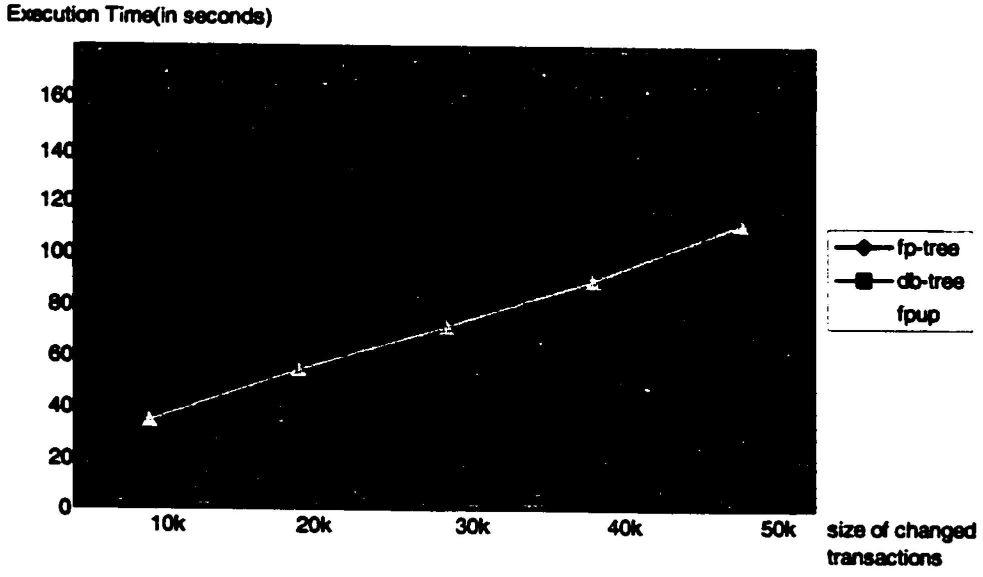

Figure 4.2-2 Execution Times with Different Changed Transaction Size on Minimum Support 0.5%

Algorithms	Runtime (in seconds) at Different Changed Transaction Size				
	10K	20K	30K	40K	50K
FP-tree	33	52	73	94	122
DB-tree	43	68	97	124	154
FPUP	26	36	50	62	83

Figure 4.2-3 Experimental Data- Execution Times (seconds) with Different Changed Transaction Size on Minimum Support 1%

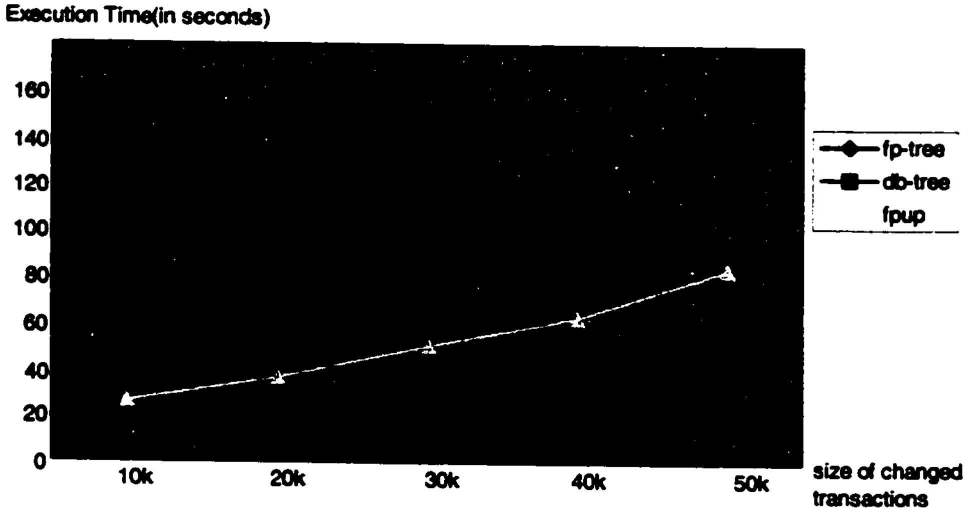

Figure 4.2-4 Execution Times with Different Changed Transaction Size on Minimum Support 1%

Algorithms	Runtime (in seconds) at Different Changed Transaction Size				
	10K	20K	30K	40K	50K
FP-tree	4	7	12	14	20
DB-tree	42	68	97	124	154
FPUP	2	3	8	7	12

Figure 4.2-5 Experimental Data- Execution Times (seconds) with Different Changed Transaction Size on Minimum Support 6%

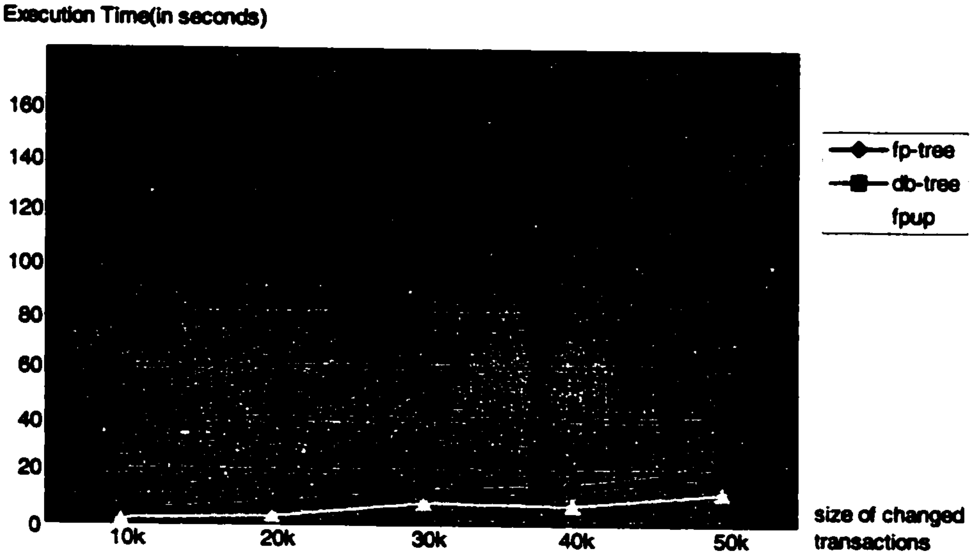

Figure 4.2-6 Execution Times with Different Changed Transaction Size on Minimum Support 6%

From the above, it can be observed that:

(1) As the changed transactions become large, all the execution times become larger.

(2) The large amount of changed transactions affect the execution times of original FP-tree and FPUP with lower minimum support much more than with higher minimum support. It can be seen that in figure 4.2, 50k transactions changed cause very long execution time because their minimum support is 0.1%. When the minimum support increased to 1% (figure 4.3), the situation becomes better. When the

minimum support becomes high, e.g., 6% in Figure 4.4, the execution times do not increase much as they did in Figure 4.2. This can be explained as: when the minimum support is very low, the large amount of changed transactions has high possibility to cause the items change from small to large that result in the entire database rescanning. When the minimum support is high, this possibility becomes very low.

(3) FPUP algorithm always has better performance than the DB-tree algorithm and original FP-tree algorithm.

(4) The execution time of DB-tree algorithm is almost stable on different minimum support. So in three figures, the line of DB-tree is almost the same. But the execution time increases when the size of changed transactions increases. Only when the minimum support is very low (≤0.5%), it shows some advantage over the original FP-tree algorithm (Figure 4.2) and the larger the changed transactions amount, the better the DB-tree is than original FP-tree.

4.3 Experiment 3: Tolerance Range Testing

Experiment 3: Given a dataset T10.I6.D100K with 1000items and fixed minimum support 1%. We use the different tolerance value t of FPUP algorithm to test the good tolerance range. We choose the tolerance from 90% of minimum support to 50% of minimum support. The experiment is conducted on different changed transactions size (from 10k to 50 k) and also the corresponding execution time of original FP-tree is shown as Figure 4.3-1 and Figure 4.3-2.

FPUP value	Runtime (in seconds) at Different Changed Transaction Size				
t	10k	20k	30k	40k	50k
90%	24	36	50	62	83
80%	26	38	56	65	85
70%	29	42	60	73	92
60%	31	46	65	78	100
50%	33	48	69	85	107

Fp-tree	33	52	73	94	122

Figure 4.3-1 Experimental Data-Execution time (seconds) of FPUP on Different Tolerance

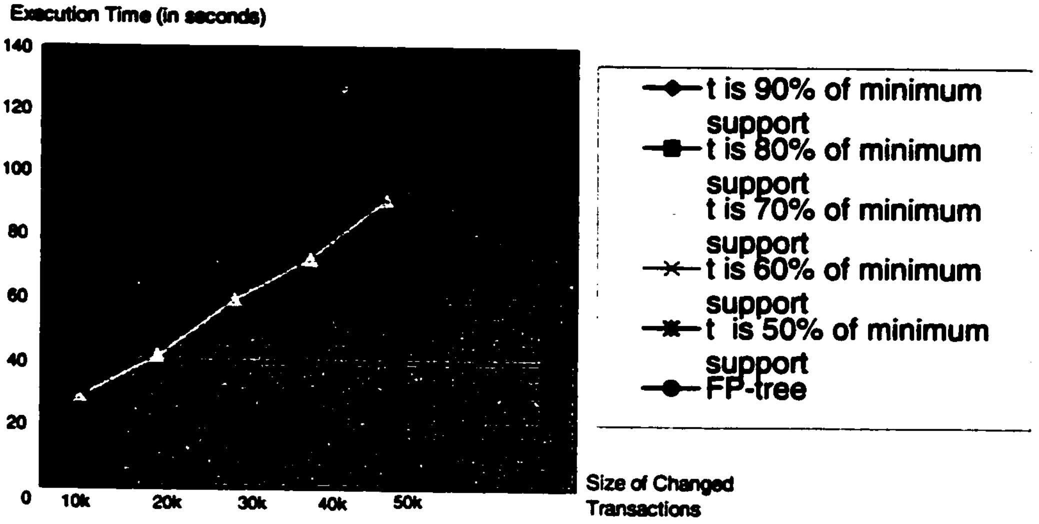

Figure 4.3-2 Execution Time of FPUP on Different Tolerance

From the figure, we can see that the tolerance is not less than 50% of the minimum support and for most cases, the larger the tolerance, the better the performance achieved. It achieves the best when the tolerance is 90% of the minimum support. The value that is more than 90% is not suggested because too high value has high-risk resulting in the entire database rescanning as the original FP-tree does.

4.4 Correctness of Algorithm Implementations

To show that the implementations of four algorithms are correct, a small sample database is used to test the correctness of four different algorithms.

The same programs used to run the experiments in section 4.1 for the four algorithms Apriori, FP-tree, DB-tree and FPUP were run on the dataset with 10 items [0...9] and totally 85 transactions given as Figure 4.4-1. The large itemsets produced as result by each of these algorithms are listed in Figure 4.4-2. The results are generated based on the minimum support 0.4.

TID	Items	TID	Items
1	0,6,7,8,9	44	3,4,7,8,9
2	4,5,6,8	45	3,8,9
3	0,7	46	5,8
4	4,6,7,8	47	4,7,8,9
5	0,5,7,8,9	48	0,8,9
6	0,1,3,4,5,7,8,9	49	0,6,8,9
7	2,3,5,6,7,9	50	3,7,8,9
8	0,3,4,5,6,8	51	3,5,8
9	9	52	0,3,4,6,7,8
10	0,5,6,8,9	53	0,4,5,8,9
11	0,3,7,8,9	54	3,4,7,8,9
12	0,3,6,8,9	55	7
13	0,3,4,5,6,8,9	56	3,5,8
14	3,8,9,	57	3,4,9
15	0,3,8,9	58	0,1,5,6
16	0,3,5,7,8	59	4,5,6,8,9
17	3,5,7,8,9	60	3
18	0.5,7,8	61	0,3,4,6,7,8,9
19	0,3,5,6,7	62	3,7,8,9
20	0,6	63	0,3,5,7,8,9
21	0,9	64	0,3,4,6,8,9
22	4,6,8,9	65	0,3,4,5,7,8,9
23	0,1,3,5,8,9	66	7,9
24	0,3,6,8,9	67	7,8
25	8,9	68	0,3,4,5,8,9
26	0,3,5,6,7,8,9	69	0,3,8
27	3,5,6,8,9	70	7,8,9
28	3,4,7,9	71	0,3,4,6,7,8,9
29	7,8	72	0,3,8,9
30	0,8,9	73	1,3,8
31	7,8	74	3,6,8,9
32	0,4,5,7,8	75	0,8,9
33	6,8,9	76	1,3,4,5,8
34	0,3,7,8,9	77	0,3,6,7,8,9
35	3,4,7	78	3,5,8,9
36	0,3,5,7,8,9	79	0,3,5,6,8,9
37	0,3,4,9	80	0,3,4,5,8,9
38	0,3,4,5,7,8,9	81	0,3,4,6,8,9
39	0,3,5,6,7,8,9	82	3,5,6,7,8,9
40	0,3,4,6,7,8,9	83	3,5,6,8
41	3,9	84	0,3,8,9
42	0,3,5,6,7,8,9	85	3,8
43	3,4,5,8		

Figure 4.4-1 the Dataset to Testing The Correctness

Algorithms	Large Itemsets Results

Apriori	{{0},{3},{5},{7},{8},{9}}
	{{0,8}, {0,9}, {3,8},{3,9}, {8,9}}
	{{3,8,9}, {0,8,9}}

FP-tree	{{9,8}, {8,3}, {9,3}, {3,8,9}, {8,0}, {9,0}, {0, 8, 9}, {7}, {5}}
DB-tree	{{9,8}, {8,3}, {9,3}, {3,8,9}, {8,0}, {9,0}, {0, 8, 9}, {7}, {5}}
FPUP	{{9,8}, {8,3}, {9,3}, {3,8,9}, {8,0}, {9,0}, {0, 8, 9}, {7}, {5}}

Figure 4.4-2 the Results of Different Algorithms

From the figure 4.4-2, it can be seen that all two new algorithms, FPUP and DB-tree, when executed produce the same correct results as the standard Apriori algorithm and the original FP-tree algorithm. This means that the experiments reported in section 4.1, 4.2 and 4.3 were based on correct implementations of these algorithms.

5. Conclusions and Future Work

Two new methods (DB-tree and FPUP) for maintaining discovered association rules in the updated database are proposed. Both of these two methods adapt the FP-tree structure that store the frequent patterns in the compact structure, and the following mining process are fully dependent on this structure. The complexity of constructing a transaction ***Trans*** into the FP-tree of original FP-tree algorithm is **O** (|***Trans***|), where |***Trans*** | is the number of frequent items in the transaction ***Trans***. The complexity of constructing a transaction ***DTrans*** into the DB-tree is **O** (|***DTrans***|), where |***DTrans*** | is the number of items in the transaction ***Trans***. The complexity of constructing a transaction ***Trans*** into the FP-tree of FPUP algorithm is **O** (|***PTrans***|), where |***PTrans*** | is the number of frequent and potential large items in the transaction ***PTrans***. Because DB-tree and FPUP algorithms both use the FP-growth() method introduced in [HPY00] to find the frequent patterns in the tree structures, they have the same complexity as the original FP-tree algorithm does. The FPUP algorithm achieves a better performance than the original FP-tree algorithm as it uses the compact tree structure and avoids rescanning the entire database when small itemsets become large due to database update. The DB-tree also avoids rescanning the entire database, however its tree structure is very large leading to high cost. So, it does not achieve better performance than FP-tree algorithm except when the minimum support is very low.

Future research should include looking for a theoretical method to decide the most beneficial tolerance value t for FPUP algorithm and improve DB-tree algorithm to save the maintenance cost thus to achieve better result.

The proposed method also can be applied in to Web usage mining. Web usage mining is the process of discovering relationships and global patterns that exist in large access log files. For example, by analyzing information from Web servers, the interesting relationships such as: 80% of clients who visited jdk1.1.6/docs/api/Package-java.io.html and /jkk1.1.6/doc/api/java.io.BufferedWriter.html in the same transaction, also accessed /jdk1.1.6/docs/relnotes/deprecatedlist.html. There are many efforts towards mining various patterns from Web Logs [CMS97, CMS99, CKL97, MJH+96, MPT00a, SF98, ZXH98]. Web access patterns are a sequential pattern in a large set of pieces of Web logs. Some research efforts try to use sequential pattern mining technology [AS95], which is mostly based on association rule mining. Other efficient mining techniques and extensions of sequential pattern mining methods [BWJ98, HDY99, MTV97] are also adopted into the web mining. All of the above studies used an Apriori like paradigm. [MPT00a, MPT00b] adopted the techniques introduced in [CHN+96, Chy97] to resolve the incremental Web usage mining problem. [PHM+00] use the FP-tree structure instead of Apriori like paradigm therefore save cost in finding the web access patterns. However, they still encounter the same problem, rescanning the original database, as it is in the transactional database incremental mining. If we can adopt the FPUP algorithm in the incremental web usage mining, we can avoid rescanning.

References

[AIS93] R. Agrawal, T. Imielinski and A. Swami. Mining Association Rules between Sets of Items in Large Databases. *Proc. of the ACM-SIGMOD 1993 Int'l Conference on Management of Data, Washington D.C., May 1993, 207-216*

[AS94] R. Agrawal and R. Srikant. Fast Algorithms for Mining Association Rules. *In Proc. of the 20th Int'l Conference on Very Large Databases, Santiago, Chile, September 1994.*

[AS95] R. Agrawal and R. Srikant. Mining sequential patterns. *In Proc. 1995 Int. Conf. Data Engineering. Pages 3-13. Taipei, Taiwan. March 1995.*

[AY98] C. Aggarwal and Philip S. Yu. Mining Large Itemsets for Association Rules. *Bulletin of the IEEE Computer Society Technical Comittee on Data Engineering, pages 23--31, March 1998*

[BL98] J. Borges, M. Levene. Mining Association Rules in Hypertext Databases. *In Proc. AAAI 1998*

[BMU+97] S. Brin, R. Motwani, Jeffrey D. Ullman and S. Tsur. Dynamic Item Counting and Implication Rules for Market Basket Data. *SIGMOD conference 1997 AZ, USA*

[BWJ98] C. Bettini, X. Sean Wang, and S. Jajodia. Mining temporal relationships with multiple gramularities in time sequences. *Data Engineering Bulletin, 21:32-38, 1998*

[CHN+96] D.W. Cheung, J. Han, V.T. Ng and C.Y. Wong. Maintenance of

Discovered Association Rules in Large Database: An Incremental Updating Technique. *In Proceedings of the 12th International Conference on Data Engineering, New Orleans, Louisiana, 1996.*

[CHY96] M.S. Chen, J. Han and Philip S. Yu. Data mining: An overview from Database Perspective. *IEEE Transactions on Knowledge and Data Eengineering. VOL. 8, NO. 6, December, 1996*

[CLK97] David W. Cheung, S.D. Lee and B. Kao. A General Incremental Technique for Maintaining Discovered Association Rules. *Proceedings of the 5th International Conference on Database Systems for Advanced Applications, Melbourne, Australia, April, 1997*

[CKL97] Discovering User Access Patterns on the World-Wilde Web. *In Proc. Of the 1st Pacific-Asia Conference on Knowledge Discovery and Data Mining, February 1997.*

[CMS97] R. Cooley, B. Mobasher, and J. Srivastava. Web Mining: Information and Pattern Discovery on the World Wide Web. *In Proc. Of The 9th IEEE Internation Conference on Tool with Artificail Intelligence, Novermber 1997.*

[CMS99] R. Cooley, B. Mobasher. And J. Srivastava. Data preparation for mining World Wide Web browsing patterns. *In Journal of Knowledge & Information Systems, Vol.1, No.1,1999*

[FSS+96] U. Fayyad, G. Piatetsky-Shapiro, P. Smyth and R. Uthurusamy. Advances in Knowledge Discovery and Data Mining. *editors (1996) MIT Press, Cambridge, MA.*

[HDY99] J. Han, G. Dong, and Y. Yin. Efficient mining of partial periodic patterns in time series database. *In Proc. 1999 Conf. Data Engineering (ICDE'99), pages 106-115, Sydney, Australia, April 1999.*

[Hi99] C. Hidber. Online Association Rule Mining. *SIGMOD conference, 1999. ACM 1999 1-581113-084-8/99/05*

[HK00] J. Han, M. Kamber. Data Mining: Concepts and Techniques. *Morgan Kaufmann Publisher, August 2000*

[HMW+98] J. Hipp, A. Myka, R. Wirth, and U. Guntzer. A New Algorithm for Faster Mining of Generalized Association Rules. *In Proceeding of the 2nd European Symposium on Principles of Data Mining and Knowledge Discovery (PKDD' 98), pages 74-82, Nantes, France, September 23-26, 1998*

[HPY00] J. Han, J. Pei and Y.Yin. Mining Frequent Patterns without Candidate Generation. *ACM SIGMOD 2000 Dallas, TX, USA*

[KHC97] M. Kamber, J. Han and J.Y. Chiang. Using Data Cubes for Metarule-guided Mining of Multidimensional Association Rules. *In proceeding of 3rd Int. Conference. Knowledge Discovery and Data Mining, Newport Beach, California, August 1997.*

[LCK98] S.D. Lee, David W. Cheung and Ben Kao. Is sampling Useful in Data Mining? A Case in the Maintenance of Discovered Association Rules. *Data Mining and Knowledge Discovery pages 233-262, 1998.*

[MJH+96] B. Mobasher, N. Jain, E. Han, and J. Srivastava. Web Mining: Pattern Discovery from World Wide Web Transactions. *Technical Report TR-96-050, Department of Computer Science, University of Minnesota, 1996*

[MPT00a] F. Masseglia, P. Poncelet, and M. Teisseire. Incremental Mining of Sequential Patterns in Large Databases. *Technical report, LIRMM, France, January 2000.*

[MPT00b] F. Masseglia, P. Poncelet, and M. Teisseire Web Usage Mining: How to Efficiently Manage New Transaction and New Clients. *Proceedings of the 4th European Conference on Principles and Practice of Knowledge Discovery in Databases (PKDD'2000)*

[MTV97] H. Mannila, H. Toivonen, and A. I. Verkamo. Discovery of frequent epsodes in event sequences. *Data Mining and Knowledge Discovery, 1:259-289, 1997.*

[PCY95] J. S. Park, M. S. Chen and Philip S. Yu. An Effective Hash_Based Algorithms for Mining Association Rules. *ACM-SIGMOD, San Jose, California, 1995.*

[PHM+00] J. Pei, J. Han, B. Mortazavi-Asl, and H. Zhu. Mining Access Patterns Efficiently from Web Logs. *Proc. 2000 Pacific-Asia Conf. on Knowledge Discovery and Data Mining (PAKDD'00), Kyoto, Japan, April 2000.*

[SA95] R. Srikant, R. Agrawal. Mining Generalized Association Rules. *In Proceeding s of the 21st VLDB Conference. Zurich, Switzerland, 1995*

[SF98] M. Spiliopoulou and L. Faulstich. WUM: A tool for Web utilization analysis. *In Proc. 6th Int'l Conf. On Extending Database Technology, Valencia, Spain, March 1998.*

[Sh91] G. Piatetsky-Shapiro. Knowledge Discovery in Databases. *AAAI Press, 1991.*

[SON95] A. Savasere, E. Omiecinski and S. Navalhe. An Efficient Algorithm for Mining Association Rules in Large Databases. *In 21st VLDB Conference, 1995.*

[SSC97] L. Singh, P. Scheuermann and B. Chen. Generating Association Rules from SemiStructured Documents Using an Extended Concept Hierarchy. *In Proceedings of the 6th International Conference on Information and Knowledge Management. Las Vegas, Nevada, USA,1997*

[SVA97] R. Srikant, Q. Vu and R. Agrawal. Mining Association Rules with Item Constraints. *Copyright 1997, American Association for Artificial Intelligence*

[TBA+97] S. Thomas, S. Bodagala, K. Alsabti, and S. Ranka. An Efficient Algorithm for the Incremental Updation of Association Rules in Large Databases. *In Proceedings of the 3rd International conference on Knowledge Discovery and Data Mining (KDD 97), New Port Beach, California, August 1997*

[To96] H. Toivonen. Sampling Large Databases for Association rules. *Proceedings of the 22nd VLDB conference Mumbai (Bombay), India,1996*

[ZE01] Z. Zhou, C.I. Ezeife. A Low-Scan Incremental Association Rule Maintenance Method Based on the Apriori Property. *Proceeding of the fourteenth Canadian Conference on Artificial Intelligence, AI 2001, June 7 to June 9, 2001, Ottawa.*

[ZXH98] O. Zaiane, M. Xin and J. Han. Discovering Web access patterns and trends by applying OLAP and data mining technology on Web Logs. *In Proc. Advances in Digital Libraries Conf., Melbourne, Australia, pages 144-158, April 1998*

APPENDIX: Correctness Testing Results

```
Script started on Sun Oct 14 16:22:21 2001
[yuesu@66-61-41-211 now]$ ./apri  please enter the
frequency here :0.4
0    46
3    56
5    35
7    39
8    71
9    59

 here is what you want0
 here is what you want3
 here is what you want5
 here is what you want7
 here is what you want8
 here is what you want9
 here is what you want0 8
 here is what you want0 9
 here is what you want3 8
 here is what you want3 9
 here is what you want8 9
 the 2-itemset has been generated!!!!!!!!!!!
The ani is: 2
 here is what you want0 8 9
 here is what you want3 8 9
The ani is: 2
begin time: 1003090947
end time  : 1003090950
3 is execution time
[yuesu@66-61-41-211 now]$ ./dbtreeinsert please enter the
minimum support:
0.4
8; 71
8; 71
9; 59
3; 56
0; 46
7; 39
5; 35
6; 31
4; 28
1; 5
2; 1
```

```
dbdbdbdbdbdbdbdbdbdbdbdbdbdbdbdbdbdbdbdbdbdbdbdbdbdbdbdbdbdbdbd
b
8:----



8with pattern {}  is frequent pattern!!!
---------------------------------------------------
8:----


---------------------------------------------------
9:----



/
{8;9} is  large
/

9with pattern {8,}  is frequent pattern!!!

---------------------------------------------------
3:----




/
{8;3} is  large
/
/
{9;3} is  large
/

3with pattern {8,9,}  is frequent pattern!!!


---------------------------------------------------
0:----
```

```
/
{8;0} is  large
/
/
{9;0} is  large
/

0with pattern {8,9,}  is frequent pattern!!!




------------------------------------------------
7:----








7with pattern {}  is frequent pattern!!!

7with pattern {}  is frequent pattern!!!
```

```
-------------------------------------------------
5:----




5with pattern {}  is frequent pattern!!!

5with pattern {}  is frequent pattern!!!




-------------------------------------------------
6:----
```

```
-------------------------------------------------
4:----
```

```
-------------------------------------------------
1:----
```

```
-----------------------------------------------
2:----



-----------------------------------------------
begin time: 1003090965
end time   : 1003090965
0 is execution time
[yuesu@66-61-41-211 now]$ ./fpinsert please enter the
minimum support:
0.4
markmarkmarkmarkmarkmarkmarkmarkmarkmarkmarkmarkmarkmarkmar
kmarkmarkmark
8:----



8with pattern {}  is frequent pattern!!!
-----------------------------------------------
8:----


-----------------------------------------------
9:----




/
{8;9} is  large
/

9with pattern {8,}  is frequent pattern!!!

-----------------------------------------------
3:----
```

```
/
{8;3} is  large
/
/
{9;3} is  large
/

3with pattern {8,9,}  is frequent pattern!!!


-----------------------------------------------
0:----







/
{8;0} is  large
/
/
{9;0} is  large
/

0with pattern {8,9,}  is frequent pattern!!!




-----------------------------------------------
7:----
```

```
7with pattern {}  is frequent pattern!!!

7with pattern {}  is frequent pattern!!!





-------------------------------------------------
5:----
```

```
5with pattern {}  is frequent pattern!!!

5with pattern {}  is frequent pattern!!!
```

```
------------------------------------------------
begin time: 1003090976
end time  : 1003090976
0 is execution time
[yuesu@66-61-41-211 now]$ ./fpupinsert please enter the
minimum support:
0.4
please enter the TOLERANCE:
0.34
8; 71
8; 71
9; 59
3; 56
0; 46
7; 39
5; 35
6; 31
4; 28
dbdbdbdbdbdbdbdbdmarkmarkmarkmarkmarkmarkmarkmarkmarkmarkma
rkmarkmarkmarkmarkmarkmarkmark
8:----



8with pattern {}  is frequent pattern!!!
------------------------------------------------
8:----


------------------------------------------------
```

```
9:----



/
{8;9} is  large
/

9with pattern {8,}  is frequent pattern!!!

---------------------------------------------------
3:----





/
{8;3} is  large
/
/
{9;3} is  large
/

3with pattern {8,9,}  is frequent pattern!!!


---------------------------------------------------
0:----








/
{8;0} is  large
/
/
{9;0} is  large
```

```
/

0with pattern {8,9,}  is frequent pattern!!!




-------------------------------------------------
7:----




7with pattern {}  is frequent pattern!!!

7with pattern {}  is frequent pattern!!!




-------------------------------------------------
5:----
```

```
5with pattern {}  is frequent pattern!!!

5with pattern {}  is frequent pattern!!!
```

```
------------------------------------------------
6:----
```

```
------------------------------------------------
4:----
```

```
------------------------------------------------
begin time: 1003091008
end time  : 1003091008
0 is execution time
[yuesu@66-61-41-211 now]$ exit
Script done on Sun Oct 14 16:23:31 2001
```

VITA AUCTORIS

Yue Su was born in Shenyang, China. She graduated from Shenyang No. 40 High school in 1988. Then, she went to Shenyang Institute of Aeronautical Technology where she obtained Honor Bachelor degree of Computer Science in 1992. She is currently a candidate for the Master's degree in school of Computer Science at the University of Windsor and will graduate in the fall term, 2001.