NOTE TO USERS

EFFICIENT MINING AND MAINTENANCE OF ASSOCIATION RULES IN LARGE DATASETS

YU SONG

A THESIS

IN

THE DEPARTMENT

OF

COMPUTER SCIENCE AND SOFTWARE ENGINEERING

PRESENTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

FOR THE DEGREE OF MASTER OF COMPUTER SCIENCE

CONCORDIA UNIVERSITY

MONTREAL, QUEBEC, CANADA

MARCH 2005

Your file Votre référence
ISBN: 0-494-04450-0
Our file Notre référence
ISBN: 0-494-04450-0

ABSTRACT

EFFICIENT MINING AND MAINTENANCE OF ASSOCIATION RULES

IN LARGE DATASETS

YU SONG

Data mining is the exploration and analysis of large quantities of data to discover meaningful patterns and rules. Mining frequent itemsets plays an essential role in many data mining tasks, which attempts to find interesting associations or correlations among a large set of data items. Efficient discovery of frequent large itemsets and its dual problem of mining association rules are well studied and efficient solution techniques have been developed and deployed in data analysis and mining tools. When new transactions are added to the dataset, it is important to maintain such discovered patterns and rules without requiring processing the whole dataset and re-computing from scratch.

In this research, we first focus on the maintenance problem and propose an in-memory technique to identify frequent large itemsets when the data set grows by addition of new transactions. The basic solution idea is to identify and use *negative borders* for maintenance. We then use this idea and develop a divide-and-conquer technique, based on *partitioning*, to compute frequent itemsets in large datasets, which do not fit into the main memory. Our experimental results show that the proposed techniques are efficient and scalable.

Acknowledgments

First of all, I would like to express deep gratitude to my supervisors Dr. V. S. Alagar and Dr. Nematallaah Shiri for their guidance and support throughout my research and study at Concordia University.

I also wish to thank my colleagues in the database lab for sharing idea, knowledge and discussion together. These include Ali Kiani, Zhi Hong Zheng, Wei Sheng Lin, Xi Deng, Hsueh-Ieng Pai, Aida Nemalhabib, Bhushan Suryavanshi, and Srividya Kadiyala.

Finally, I thank my families for their wholehearted support throughout this project.

This work was supported in part by grants from Natural Sciences and Engineering Research Council (NSERC) of Canada (Grant No. N00760), and by ENCS Faculty Research Support (Grants No. V01010 and VE0049). I really appreciate the support.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

Discovery and maintenance of association rules between items in large datasets of transactions have been identified as an important data mining problem [1, 2]. Mining association rules plays an essential role in many data mining tasks that tries to find interesting associations or correlation relationships among items in large datasets. Efficient discovery of association rules in large datasets is a well studied topic for which several solution approaches have already been proposed. However, it is nontrivial how to maintain such discovered associations when the dataset grows frequently or occasionally, and which may result in invalidating some association rules and/or introduction of some new such rule.

The problem of mining association rules over basket data was introduced in 1993 [3] for which several efficient algorithms have been proposed, including Apriori [1, 3, 4, 5], FP-growth [6, 7], Eclat [8, 9], DIC [10], etc. These algorithms provide efficient mechanisms for finding frequent itemsets, with a user specified minimum support, for which the association rules are computed. Addition of transactions to the database may invalidate existing rules or introduce new ones. The problem of updating the association rules can be mapped to first finding the new set of frequent itemsets. With large amounts of the

transaction data continuously being created and stored, the dataset becomes extremely large. A solution to the update problem is to re-compute the frequent itemsets of the whole dataset from scratch. Clearly, this could be an inefficient solution since it ignores all the computations done previously for finding the frequent itemsets. Mannila and Toivonen introduced the notion of the *Border* of a downward closed collection of itemsets [11], as a technique for sampling large databases for association rules [12]. It presented an efficient association rule maintenance algorithm to find the new frequent itemsets with minimal re-computation when new transactions are added to the dataset. Since negative border potentially may be turned into frequent itemset when the dataset is incremented, it is used to decide when to scan the whole dataset. We use the idea from sampling large databases to compute and/or maintain association rules by determining frequent itemsets and negative borders. Meanwhile, we lower the minimum support threshold to get more frequent itemset candidates. That is a way to reduce the need to rescan the original dataset.

1.1 Data Mining

The capability of data generation and collection has been increasing rapidly in the last few decades. With the widespread use of bar codes for most commercial products, the computerization of many businesses, scientific, and government transactions, and advances in data collection tools, we are flooded by all kinds of data, such as scientific data, medical data, demographic data, financial data, and marketing data. People have no time to read this data, but they would like to retrieve some useful information from it. It

is well known that necessity is the mother of invention. So, it is important to find some methods to analyze, classify, summarize the data automatically, further discover and characterize the business trends in the data. Data mining, as used in database research, is the exploration and analysis of large quantities of data in order to discover meaningful patterns and rules [2]. Data mining is one of the most active and exciting areas in the database research.

Data mining has attracted a great deal of attention in the information industry in recent years because of the wide availability of huge amounts of data and the imminent need for turning such data into useful information and knowledge. Data mining, also known as knowledge discovery in databases (KDD), is the process of discovering interesting knowledge from large amounts of data stored either in databases, data warehouses, or other information repositories.

In principle, data mining should be applicable to any kind of information repository, which includes relational databases, data warehouses, transactional databases, advanced databases, flat files, and the World Wide Web. Advanced database systems include object-oriented and object-relational databases, and specific application-oriented databases, such as spatial databases, time-series databases, text databases, and multimedia databases. In our work, we consider transactional databases. In general, a transactional database is a set of tuples, called transactions; each of tuples represents a transaction. A transaction typically includes a unique transaction identity number (trans_ID), and a list of the items making up the transaction (such as items purchased by a customer) as shown in Table 1.

trans_ID	List of item_IDs
T100	1, 3, 7, 16
T101	2, 7, 23
T102	3, 5, 12
...	...

Table 1: An example of a transaction database

There are various types of data stores and database systems on which data mining techniques can be applied. What kind of data patterns can be mined? Data mining functionalities are used to specify the kind of patterns to be found in data mining tasks. In general, data mining tasks can be classified into two categories: descriptive and predictive. Descriptive mining tasks characterize the general properties of the data in the database. Predictive mining tasks perform inference on the current data in order to make predictions. According to the kinds of knowledge, data mining systems can be categorized into the following six tasks based on data mining functionalities.

- Classification
- Estimation
- Prediction
- Association Rules
- Clustering
- Description and profiling

The goal of the first three is to find the value of a particular target variable. Association rules and clustering are undirected tasks where the goal is to uncover structure in data in respect of a particular target variable. Profiling is a descriptive task [2]. In this work, we focus on techniques for association rules mining and maintenance.

1.2 Association Rules Mining

With large amounts of data continuously being collected and stored, association or correlation relationships among items in such data are increasingly becoming more interesting to discover. This can be helpful in intelligent decision making processes, such as catalog design, cross-marketing, and loss-leader analysis. A typical example of association rule mining is "market basket analysis". A natural question that may arise in this business is which items are frequently purchased together? For example, if customers are buying chips, how likely is that they also buy drinks (and what kind of drink) on the same trip to the supermarket? Such kind of analysis can help increase sales by selective marketing, promotion, and arrange shelf space.

Association rule is a form of representing such kind of patterns. If we assume the universe as the set of items available, then each item is a Boolean variable representing the presence or absence of that item. A transaction records the items purchased on the receipt of a customer. For example, the information that clients who purchase chips also tend to buy drinks at the same time is represented as "an association rule" as follows:

$$\text{Chips} \Rightarrow \text{Drinks (support} = 3\%, \text{confidence} = 50\%)$$

Support and confidence are two threshold values denoting rule interestingness, set by a user or domain expert. They reflect the usefulness and certainty of discovered rules respectively. The above association rule with support of 3% means that 3% of all the transactions under analysis show that chips and drinks are purchased together. A confidence of 50% shows that 50% of the customers who purchased chips also bought the drink. Normally, association rules are considered interesting if they satisfy both a minimum support threshold and a minimum confidence threshold.

Market basket analysis is just one typical form of association rule mining. There are many other kinds of association rules. Association rules can be classified as follows:

- Based on the types of values handled in the rule: If a rule only considers associations between the presence and absence of items, it is a Boolean Association Rule. A rule concerns associations between quantitative items or attributes is a Quantitative Association Rule. The following rule is an example of a quantitative association rule, where X is a variable representing a customer:

  $$age(X, \text{“20…35”}) \wedge income(X, \text{“42K…58K”}) \Rightarrow buys(X, \text{Honda Car})$$

  In quantitative association rules, quantitative rules for items or attributes are partitioned into intervals.

- Based on the dimensions of data involved in the rule: If a rule reference only one dimension, then it is a single-dimensional association rule. If the items or

attributes in an association rule references two or more dimensions, it is a multi-dimensional association rule.

- Based on the levels of abstractions involved in the rule set: Some methods for association rule mining can find rules at different levels of abstraction. For example, suppose that a set of association rules mined includes the following rules:

$$\text{Chips} \Rightarrow \text{Drinks (support} = 3\%, \text{confidence} = 50\%)$$

$$\text{Chips} \Rightarrow \text{Sprint (support} = 2.5\%, \text{confidence} = 40\%)$$

"Drinks" is a higher-level abstraction of "Sprint". Thus, the association rules can be classified into multi-level association rules and single-level association rules.

In this research, we concentrate on single-dimensional, single-level, Boolean association rules discovery and maintenance. Some main characteristics of the association rules mining problem are as follows.

1. The size of the database is significantly large.
2. The rules discovered can hold only in statistical terms. Thus, the number of rules returned from a mining activity could be large.
3. The rules discovered from a database only reflect the current state of the database. For increased reliability and usefulness of such discovered rules, large volumes of data need to be collected and analyzed over different period of time.

4. The number of items and their association rules could be vary, but each rule involves only a few items.

1.3 Contributions

In this work, we propose a technique based on the notation of negative border and lowered minimum support to identify new association rules, when new transactions are added to the dataset. Then we develop this method based on partitioning for large datasets, which do not fit in main memory. The proposed technique can be used as a divide and conquer technique, similar to external sorting used in databases, for discovering association rules in large dataset of transactions. We have successfully designed and implemented this technique.

The contributions of our work are as follows.

1. Cheung et al. [13] propose an incremental updating technique, FUP (Fast Update), for maintenance of discovered association rules in large databases. It is based on the Apriori algorithm which requires O(n) passes over the whole database. Negative border [12], proposed as a method in sampling large database for association rules, is used to identify all association rules of a random subset of transactions that probably hold in the whole database, and then verify the results with the rest of the database. In this research, we first propose an efficient mining and maintenance algorithm for discovering association rules in large datasets based on both negative border and lowered minimum support threshold.

2. We extend the level-wise algorithm of association rules mining like Apriori to compute the frequent itemset along with the negative borders at the same time.
3. We propose an algorithm of association rules maintenance to "merge" the frequent itemset and the negative border information containing old and new transactions to get the actual frequent itemset for the whole dataset.
4. We develop the "Dataset Generator", the "Dataset merger", and the "Dataset Splitter" to generate, combine, and split the input dataset, respectively.
5. We build a platform in both Windows (using MFC as the interface) and Linux to test the applicability and efficiency of our proposed algorithm.
6. We further propose a partition-based algorithm to compute frequent itemsets in very large dataset which may not fit into main memory entirely. It first divides the large dataset into several non-overlapping partitions, for each of which we find all the local frequent itemsets and negative borders with their support counts. Then we "merge" them from all partitions and identify the global frequent itemsets of entire dataset. This algorithm is implemented in C/C++ and tested in Linux.

1.4 Outline of the Thesis

This thesis is organized as follows.

Chapter 2 presents formally the problem statement of frequent itemset mining, negative border and association rules mining and maintenance, and describes some basic properties. In Chapter 3, we review and survey all the related work of our research including the level-wise Apriori framework, FP-growth algorithm, incremental updating,

sampling large database for association rule mining, and partitioning algorithm [14]. Our system prototype and architecture are presented in Chapter 4. We also introduce each of the system functional modules and interfaces in this Chapter. We present our approach of mining and maintenance for association rules in large datasets based on negative border and lowered support minimum threshold in Chapter 5. In this chapter, we give the pseudocode of our algorithm for each of our system functional modules, some illustrative examples, the data structure we use in our implementation, and some technical features of our implementation details. Experimental results and performance comparison are presented in Chapter 6. We compare our algorithm with FUP, Sampling, partitioning in this chapter and demonstrate the executive time in different chart. We introduce our partition-based algorithm in Chapter 7 with algorithm pseudocode, analysis and performance evaluation. Chapter 8 contains conclusions and future work.

Chapter 2

Problem Statement

All the basic concepts and the formal problem statement concerning our research are given in this chapter [1, 2, 3, 15, 16]. A formal description of the problem of association rules mining is as follows.

2.1 Frequent Itemset

Let $I = \{i_1, i_2, \ldots, i_m\}$ be a set of items. A set $X = \{i_1, i_2, \ldots, i_k\} \subseteq I$ is called an *itemset*, or a *k-itemset* if it contains k items. For example, the itemset {Diaper, Beer} is a 2-itemset. A *transaction* over I is a pair T = <TID, X> where *TID* is the transaction identifier and I is an itemset. A transaction database D over I is a finite set of transactions over I. Each transaction identifier, TID, is unique in the transaction database D. We omit I whenever it is clear from the context.

A transaction T_i = (TID, X) is said to contain an itemset $Y \subseteq I$, if $Y \subseteq X$. The *cover* of an itemset Y in D consists of the set of identifiers of all transactions in D that contains Y:

$$\text{cover}(Y, D) := \{\text{TID} \mid (\text{TID}, X) \in D, Y \subseteq X\}.$$

The occurrence frequency of an itemset Y is the number of transactions in D that contain Y. This may also be called as *frequency*, *support count*, or *count of the itemset*. The *support count* of an itemset Y in D is the number of transactions in the over of Y in D:

$$\text{count}(Y, D) := |\text{ cover}(Y, D)|.$$

The *support* of an itemset Y in D is the probability of Y occurring in a transaction $T \in D$:

$$\text{support}(Y, D) := P(Y) = \frac{\text{count}(Y,D)}{|D|}$$

Note that $|D| = \text{count}(\{\}, D)$. We omit D whenever it is clear from the context.

The number of transactions required for an itemset to satisfy minimum support is therefore referred to as the *minimum support count.* An itemset is called frequent, or large, if its support count is no less than a given threshold min_count, with $0 \leq \text{min\_count} \leq |D|$. When working with support of itemsets instead of their support count, we use a minimum support threshold min_sup, with $0 \leq \text{min\_sup} \leq 1$. Obviously, $\text{min\_count} = \lceil \text{min\_sup} \times |D| \rceil$. The set of frequent itemsets is denoted by L and the set of frequent k-itemsets is commonly denoted by L_k. The notations are shown in Table 2.

I	Set of items
TID	Transaction identifier
T	A transaction over I is a couple T = (TID, X)
L	Set of frequent itemsets
L_k	Set of frequent k-itemsets

Table 2: Notations of frequent itemset

Definition 1. Let D be a transaction database over a set of items I, and min_sup a minimum support threshold. The collection of frequent itemsets in D with respect to min_sup is denoted by:

$$F(D, min\_sup) := \{X \subseteq I \mid support(X, D) \geq min\_sup\},$$

or simply F is the arguments D and min_sup are known from the context.

Problem 1. (Mining Frequent Itemset) Given a set of items I, a transaction database D over I, and minimum support threshold min_sup, find F (D, min_sup).

Proposition 1 [15, 16]. (Support Monotonicity) Given a transaction database D over I, let $X, Y \subseteq I$ be two itemsets. Then,

$$X \subseteq Y \Rightarrow support\ (Y) \leq support\ (X)$$

Proof. This follows immediately by observation that

$$\text{cover}(Y) \subseteq \text{cover}(X)$$

Hence, if an itemset is infrequent, all of its supersets must be infrequent. In the literature, this monotonicity property is also called as *downward closure property*, since the set of frequent itemsets is closed with respect to set inclusion.

Proposition 2 [15, 16]. Let $X, Y, Z \subseteq I$ be itemsets.

$$\text{support}(X \cup Y \cup Z) \geq \text{support}(X \cup Y) + \text{support}(X \cup Z) - \text{support}(X)$$

Proof.

$$\begin{aligned}
\text{support}(X \cup Y \cup Z) &= |\text{cover}(X \cup Y) + \text{cover}(X \cup Z)| \\
&= |\text{cover}(X \cup Y) \setminus (\text{cover}(X \cup Y) \setminus \text{cover}(X \cup Z))| \\
&\geq |\text{cover}(X \cup Y) \setminus (\text{cover}(X) \setminus \text{cover}(X \cup Z))| \\
&\geq |\text{cover}(X \cup Y)| - |(\text{cover}(X) \setminus \text{cover}(X \cup Z))| \\
&\geq |\text{cover}(X \cup Y)| - (|\text{cover}(X)| - |\text{cover}(X \cup Z)|) \\
&= \text{support}(X \cup Y) + \text{support}(X \cup Z) - \text{support}(X)
\end{aligned}$$

2.2 Negative Border

Let F be an itemset and P(F) be its powerset. Given a collection $L \subseteq P(F)$ of frequent itemsets, the *negative border* NB(L) of L consists of the minimal itemsets $X \subseteq F$, but not in L.

The collection of all frequent sets is always closed with respect to set inclusion. For instance, let R = {A, B, C, D} and assume the collection of frequent sets with some minimum support min_sup is

$$L = \{\{A\}, \{B\}, \{C\}, \{D\}, \{A, B\}, \{A, D\}, \{B, C\}, \{B, D\}, \{C, D\}, \{A, B, D\}$$

The negative borders of L contains {A, C}, since it is not in the collection L, but all its subsets are. The collection of negative borders includes:

$$NB(L) = \{\{A, C\}, \{B, C, D\}\}$$

Definition 2. (Border) Let F be a downward closed collection of subsets of I, the *border* B(F) consists of those itemsets $X \subseteq I$ such that all subsets of X are in F, and no superset of X is in F:

$$B(F) := \{X \subseteq I \mid \forall Y \subset X : Y \in F \wedge \forall Z \supset X: Z \notin F \}.$$

Those itemsets in B(F) that are in F are called the *positive border* of F, denoted as PB(F):

$$PB(F) := \{X \subseteq I \mid \forall Y \subseteq X : Y \in F \wedge \forall Z \supset X : Z \notin F\},$$

and those itemsets in BD(F) that are not in F are called the *negative border* of F, denoted as NB(F) :

$$NB(F) := \{X \subseteq I \mid \forall Y \subset X : Y \in F \wedge \forall Z \supseteq X : Z \notin F\},$$

Problem 2. (Mining Negative border) Given a set of items I, a transaction database D over I, and minimum support threshold min_sup, find NB(F(D, min_sup)).

Lemma 1. All 1-itemsets should be present in $L \cup NB(L)$.

The negative border consists of all itemsets that were candidates of the level-wise method which did not have enough support. That is, $NB(L_k) = C_k - L_k$, where C_k is the set of candidate k-itemsets, L_k is the set of frequent k-itemsets, and $NB(L_k)$ is the set of k-itemsets in NB(L). Therefore, $C_k = L_k \cup NB(L_k)$. The set C_k can be generated using only L_{k-1}.

Lemma 2. Let X be an itemset such that $X \in NB(L)$. Then all the possible subsets of X must be present in L.

For a contradiction, let Y be an itemset such that $Y \subset X$ and $Y \notin L$. By the definition of negative border, NB(L) consists of the minimal itemsets not in L. Since $Y \notin L$, X is not a minimal itemset not in L. Thus, X can not be in NB(L), which is a contradiction.

The lattice for the frequent itemsets from the example above, together with its positive border and negative border, is shown in Figure 1.

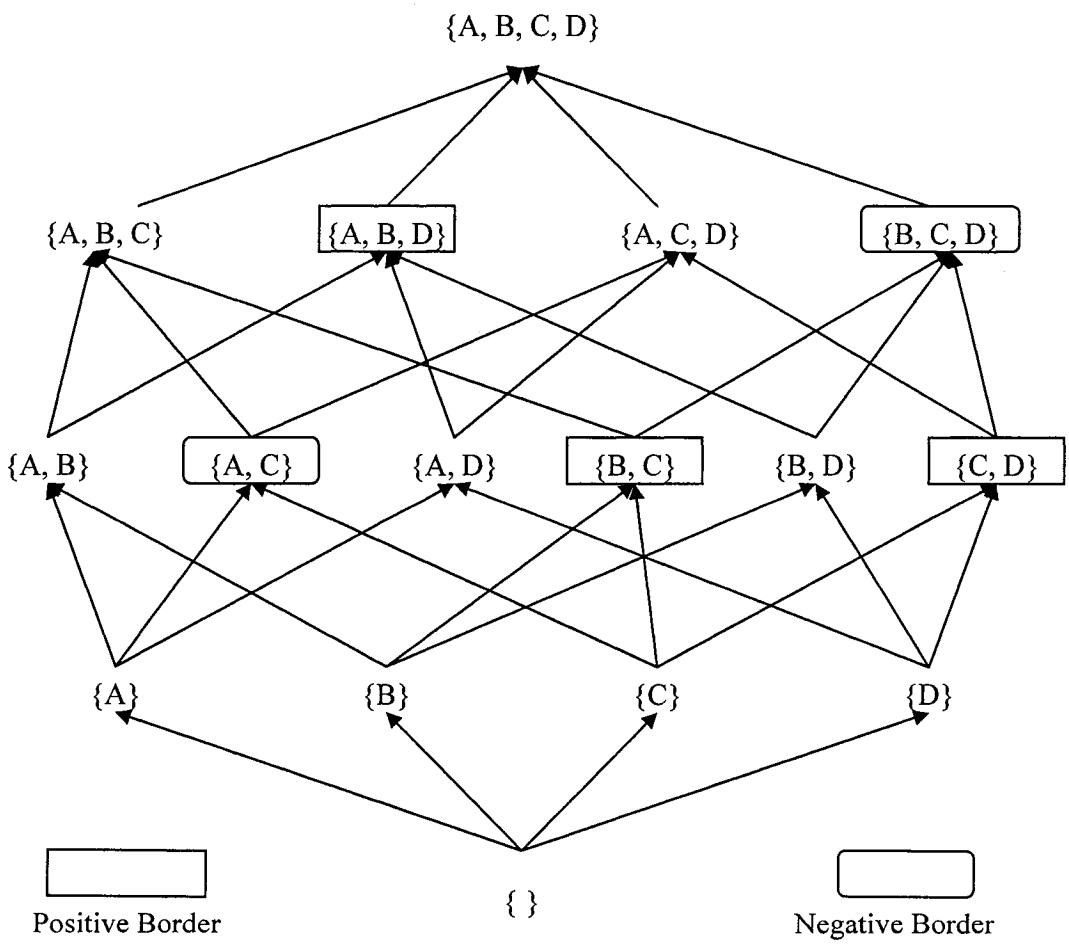


Figure 1: The lattice for the example and its border

2.3 Association Rule

An *association rule* is an expression of the form $X \Rightarrow Y$, where $X \subseteq I$, $Y \subseteq I$, and $X \cap Y = \varnothing$. Such a rule expresses the association that if a transaction contains all items in X, then that transaction also contains all items in Y. The set X is called the *body* or *antecedent*, and Y is called the *head* or *consequent* of the rule. The rule $X \Rightarrow Y$ holds in database D with support s, if the percentage of transactions in D that contain $X \cup Y$ is at least s. This is taken to be $P(X \cup Y)$, i.e., the probability of $X \cup Y$. The confidence of rule $X \Rightarrow Y$ in D is c, if c is the percentage of transactions T in D which T contains Y whenever T contains X. This is taken to be the conditional probability, $P(Y \mid X)$.

That is,

$$\text{support}\,(X \Rightarrow Y) = P(X \cup Y)$$

$$\text{confidence}(X \Rightarrow Y) = P(Y \mid X)$$

The problem of mining association rules is to generate all association rules that satisfy two user-specified thresholds: a minimum support (called *min_sup*) and a minimum confidence (called *min_conf*). A rule is strong if its support and confidence are not less than min_sup and min_conf, respectively.

The problem of mining association rules is to generate all rules whose have support and confidence are at least equal to some user specified minimum support and minimum

confidence thresholds, respectively. The problem can be decomposed into a two-step process:

1. Generate all frequent itemsets: In the first step, all itemsets that have support above or equal to the user specified minimum support are found. Each of these itemsets will occur at least as frequently as a pre-determined minimum support count.
2. Find strong association rules from the frequent itemsets: For each frequent itemset, all the rules that have minimum confidence are generated in step two as follows. For a frequent itemset X and any $Y \subseteq X$, if $\frac{\text{support}(X)}{\text{support}(X - Y)} \geq \text{min\_conf}$, then the rule $X - Y \Rightarrow Y$ is a valid rule.

Definition 3. Let D be a transaction database over a set of items I, min_sup be a minimum support threshold value, and min_conf be a minimum confidence threshold vlaue. The collection of frequent and confident association rules with respect to min_sup and min_conf is denoted by:

$$R(D, \text{min\_sup}, \text{min\_conf}) := \{X \Rightarrow Y \mid X, Y \subseteq I, X \cap Y = \varnothing,$$
$$X \cup Y \in F(D, \text{min\_sup}), \text{confidence}(X \Rightarrow Y, D) \geq \text{min\_conf}\},$$

or simply R if the arguments D, min_sup, and min_conf are all known from the context.

Problem 3. (Mining Association Rule) Given a set of items I, a transaction database D over I, and minimum support, and confidence threshold values min_sup and min_conf, find R(D, min_sup, min_conf).

For example, let T1 = {1, 3, 4}, T2 = {2, 3, 5}, T3 = {1, 2, 3, 5}, and T4 = {2, 5} be the only transactions in the database D. Let the minimum support and minimum confidence be 0.6 and 0.5, respectively. And then frequent itemsets are: {2}, {3}, {5}, and {2, 5}. The valid association rules are $2 \Rightarrow 5$ and $5 \Rightarrow 2$.

Proposition 3 [15, 16]. (Confidence monotonicity) Let $X, Y, Z \subseteq I$ be three itemsets, such that $X \cap Y = \varnothing$. Then,

$$\text{Confidence}(X \setminus Z \Rightarrow Y \cup Z) \leq \text{confidence}\ (X \Rightarrow Y).$$

Proof. Since $X \cup Y \subseteq X \cup Y \cup Z$, and $X \setminus Z \subseteq X$, we have

$$\frac{\text{support}\ (X \cup Y \cup Z)}{\text{support}\ (X / Z)} \leq \frac{\text{support}\ (X \cup Y)}{\text{support}\ (X)}$$

In other words, confidence is monotone decreasing with respect to extension of the head of an association rule. If an item in the extension is included in the body, then it is removed from the body of that rule. Hence, if the certain head of an association rule over

an itemset I cause the rule to be less confident, every superset of the head must also be less confident.

2.4 Association Rule Maintenance

Let DB1 be a table of transactions, called the original DB, and DB2 be a new set of such transactions, called the new DB. Let DB be the set of transactions in DB1 and DB2, i.e., DB = DB1 $\cup$ DB2, called the updated database. Let d1 be the number of transactions in DB1, d2 be the number of transactions in DB2, and d be the number of transactions in DB, that is, d1 = |DB1|, d2 = |DB2|, and d = |DB| = d1 + d2. Let min_sup be some fixed threshold value indicating the minimum support in DB1, DB2, and DB. For a given set X of items, we use count1(X) to denote the support count of X in DB1, count2(X) to denote the support count of X in DB2, and count(X) to denote the support count of X in DB. An itemset X is frequent in DB if count(X) $\geq$ d $\times$ min_sup.

Let L1 and L2 be tables that store all frequent itemsets X and their support counts in DB1 and DB2, respectively. We use *t_sup* to denote the "lowered" minimum support, where 0 < t_sup < min_sup. Let NB1 and NB2 denote all the negative borders and their support counts in DB1 and DB2, respectively.

Lemma 3. Let X be any itemset such that X $\notin$ L1. Then X $\in$ L only if X $\in$ L2.

Assume that there exists an itemset X such that $X \in L$, $X \notin L1$, and $X \notin L2$. Let count1(X) and count2(X), respectively be the number of transactions in DB1 and DB2 that contain the itemset X. Also let d1 and d2 be the total number of transactions in DB1 and DB2, respectively. Since $X \notin L1$ and $X \notin L2$, we have that

$$\frac{count1(X)}{d1} < min\_sup \text{ and } \frac{count2(X)}{d2} < min\_sup.$$

From above inequalities, it can be shown that

$$\frac{count1(X)+count2(X)}{d1+d2} = \frac{count(X)}{d} < min\_sup$$

Therefore, $X \notin L$, which is a contradiction.

DB1	The original transaction database
DB2	The new transaction database
DB	The updated transaction database, which DB = DB1 $\cup$ DB2
L1	Set of frequent itemsets in DB1
L2	Set of frequent itemsets in DB2
L	Set of frequent itemsets in DB
d1	The total number of transactions in DB1
d2	The total number of transactions in DB2
d	The total number of transactions in DB
NB(L1)	Set of negative border in DB1
NB(L2)	Set of negative border in DB2
NB(L)	Set of negative border in DB

Table 3: Notations for association rules maintenance

Theorem 1. Let X be an itemset such that $X \notin L1 \cup NB(L1)$ and $X \in L$. Then there exists an itemset Y such that $Y \subset X$, $Y \in NB(L1)$, and $Y \in L$. That is, some subset of X has been moved from NB(L1) to L.

Proof. Since $X \in L$, all possible subsets of X should be in L. But all the subsets of X can not be in L1 because in that case, X should be present in at least NB(L1), if not in L1 itself. By the assumption, $X \notin L1 \cup NB(L1)$. Therefore, there exists an itemset Y such that $Y \subset X$, and $Y \notin L1$.

Chapter 3

Related Work

Apriori is an influential algorithm for mining frequent itemsets for Boolean association rules. Since their introduction in 1993 by Argawal et al. [3], the frequent itemset and association rule mining problems have received a great deal of attention. During the past decade, numerous research result have been published presenting new algorithms or improvements on existing algorithms to solve these mining problems more efficiently. We implemented Apriori algorithm for mining association rules and extended it to compute the frequent itemsets along with their negative borders in each iteration at the same time. We will also compare Apriori algorithm with our proposed algorithm in Chapter 6.

Mining frequent patterns without candidate generation was proposed by Jiawei Han et al. in 2000 [6]. They compressed a large transaction database into a FP-tree structure, and then developed a FP-growth method to traverse the tree to get all the frequent patterns. It is faster than the Apriori algorithm and some new frequent pattern mining methods. It is interesting to study the performance of this algorithm extended to also compute negative borders while mining the frequent itemsets. We implemented this algorithm as well. Since Apriori algorithm relies on Breadth First Search (BFS) scanning the database but the generation of the FP-tree is done by Depth First Search (DFS), so the issue of how

FP-growth method computes frequent itemsets along with negative borders is discussed in the last chapter.

Cheung et al. [13] propose an algorithm, FUP (Fast Update), for maintaining association rules in large databases when new transactions are added to the database. It is based on the Apriori algorithm which requires O(n) passes over the database, where n is the size of the maximal frequent itemset. This algorithm identifies "promising" itemsets and "hopeless" itemsets in the incremental portion and reduces the size of the candidate set to be searched against the original database. The framework of FUP is similar to that of Apriori. It is an iterative procedure, starting at iteration one with size-one itemsets and proceeds to next iteration to complete candidate sets, based on the frequent itemsets found at the previous iteration.

In our work, we use the idea of negative border, proposed originally as a method for sampling large databases, and develop an efficient algorithm to identify new frequent itemsets when new transactions are added to the database. Sampling large databases for association rules is to consider a subset of transactions randomly, from which we identify all association rules that probably hold in the whole database, and then verify the results with the rest of the database.

Another efficient algorithm for mining association rules in large databases is based on partitioning the data [14]. It accomplishes mining association rules in two scans of the database. In the first scan, all potentially frequent itemsets in each partition are identified. This is referred to as local frequent itemsets. Then the actual supports for local frequent itemsets are identified which are then used to determine global frequent itemsets in the second scan of the database.

In what follows, we discuss details in each of the related works.

3.1 Apriori Algorithm

The reason why this algorithm is called Apriori is based on the fact that the algorithm uses prior knowledge of frequent itemset properties, namely that all non-empty subsets of a frequent itemset are frequent. Apriori algorithm is an iterative approach known as a level-wise search, where k-itemsets are explored from (k–1)-itemsets. The algorithm finds the set of frequent 1-itemset. According to our notation, this set is represented by L_1. L_1 is used to find L_2, which is the set of frequent 2-itemsets. Then we use frequent k-itemsets to find frequent (k+1)-itemsets. The iterative procedure will stop until no more frequent k-itemsets is found. Finding each L_k requires one full scan of the database.

An important property called the Apriori property is used to reduce the search space to improve the efficiency of the level-wise generation of frequent itemsets. This property is based on the following observation. If an itemset I does not satisfy the minimum support threshold, minsup, then I is not frequent, that is, P(I) < minsup. If an item X is added to the itemset I, then the new itemset ($I \cup A$) can not be more frequent than I. Therefore, $I \cup A$ is not frequent either, that is, $P(I \cup A) <$ minsup.

For the anti-monotone property, which belongs to a special category of properties, if a set can not pass a test, all of its supersets will fail the same test as well. It is named anti-monotone because the property is monotonic in the context of failing a test.

Here is an example to show how the Apriori property used in the algorithm is. The following two-step process, which consists of join and prune actions, will demonstrate how L_{k-1} is used to find L_k.

1. The join step: To find L_k, a set of candidate k-itemsets is generated by joining L_{k-1} itself. The candidates k-itemsets is denoted C_k. Let l_1 and l_2 be two frequent (k-1)-itemsets, which are in L_{k-1}. The notation $l_i[j]$ is used to represent the jth item in l_i. By convention, Apriori algorithm assumes that items within a transaction or itemset are sorted in lexicographic order. The members of L_{k-1} are joinable if their first $(k-2)$ items are common. Then the join, $L_{k-1} \bowtie L_{k-1}$, can be performed. That is, members l_1 and l_2 of L are joined if $(l_1[1] = l_2[1]) \wedge (l_1[2] = l_2[2]) \wedge \ldots \wedge (l_1[k-2] = l_2[k-2]) \wedge (l_1[k-1] \neq l_2[k-1])$. The condition $l_1[k-1] \neq l_2[k-1]$ is used to ensure that no duplicates are generated. The candidate k-itemset formed by joining l_1 and l_2 is $l_1[1]\ l_1[2]\ \ldots\ l_1[k-2]\ l_1[k-1]\ l_2[k-1]$.
2. The prune step: The candidate k-itemset C_k is a superset of the frequent k-itemset L_k, which its members may or may not be frequent. However, all the frequent k-itemsets are included in C_k by definition. A simple scan of the database to determine the support count of each candidate in C_k would help decide the frequent itemsets in L_k (i.e., by definition, all candidates having a support count no less than the given minimum support count are frequent, and therefore belong to L_k). However, C_k can be very large, which may result in a heavy computation. The property of the Apriori algorithm can be used to reduce the size of C_k. According to this property, a k-itemset can not be frequent if any of its $(k-1)$-subset is not in L_{k-1}. Hence, after the join step, if any $(k-1)$-subset of a candidate k-itemset is not in L_{k-1}, then this candidate k-itemset can not be frequent either and so can be removed from C_k before scanning the database.

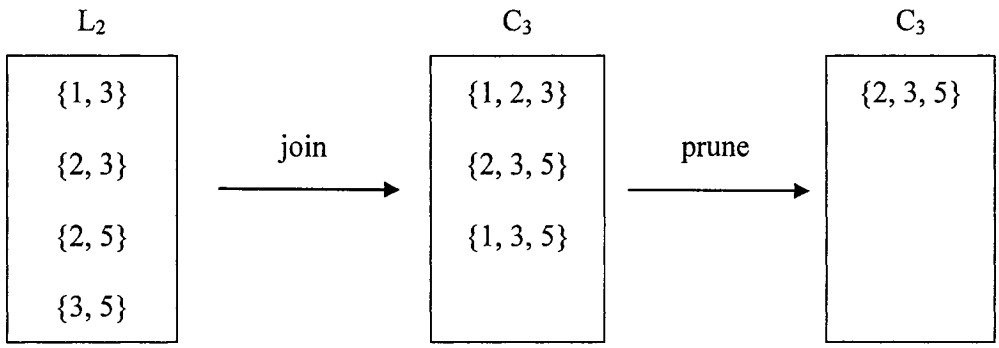


Table 4: An example of the two steps of the Apriori

For example, the set of frequent 2-itemsets, L_2, is determined as shown in Table 4. The set of candidate 3-itemsets, C_3, is generated by self join of L_2. That is, $C_3 = L_2 \bowtie L_2 = \{\{1, 3\}, \{2, 3\}, \{2, 5\}, \{3, 5\}\} \bowtie \{\{1, 3\}, \{2, 3\}, \{2, 5\}, \{3, 5\}\} = \{\{1, 2, 3\}, \{2, 3, 5\}, \{1, 3, 5\}\}$. Based on the Apriori property that all subsets of a frequent itemset must also be frequent, we can determine that {1, 2, 3} and {1, 3, 5} cannot be frequent because {1, 2} and {1, 5} are not in L_2. We therefore remove them from C_3 in the prune step, thereby saving the unnecessarily effort for computing their counts during the subsequent scan of D to determine L_3.

3.2 FP-Growth Algorithm

As we have seen, Apriori algorithm may need to generate a huge number of candidate sets. For instance, if there are 10^3 frequent 1-itemsets, the Apriori algorithm will need to generate more than 10^5 candidate 2-itemsets and accumulate and test their occurrence frequencies. And if we want to discover a frequent pattern of size 100, i.e., $\{i_1, \ldots, i_{100}\}$,

it will generate more than $2^{100} \approx 10^{30}$ candidate itemsets in total, making it intractable. An algorithm, called frequent-pattern growth, mines frequent itemsets without candidate generation [6]. Frequent-pattern growth method, or simply FP-growth, is a divide-and-conquer method which compresses the database representing frequent items into a frequent-pattern tree, or FP-tree, but retains the itemset association information, and then divides such a compressed database into a set of conditional databases, each associated with one frequent item, and compute each such database separately.

We implemented FP-growth algorithm as well, and report its performance in Chapter 6. Most of the association rule maintenance and updating techniques are based on the level-wise algorithm like Apriori. How can a FP-growth algorithm, which is based on DFS without candidate generation, work for association rule maintenance and updating, mining the frequent itemsets along with the negative borders, will be an interesting topic in the future work.

We give a simple example to show how to build the FP-tree and then how to traverse the FP-tree by FP-growth method to get the frequent itemsets. The transaction database D is as shown in Table 5. The first scan of the database yields the set of frequent 1-itemsets and their support counts, which can be the same as the Apriori algorithm. Let us assume that the minimum support count is 2. Then the set of frequent 1-itemsets is sorted in the descending order by their support count. The resulting set or list is denoted L^1. Thus, we have $L^1 = \{b:7, a:6, c:6, d:2, e:2\}$.

TID	List of Items
T100	a, b, e
T200	b, d
T300	b, c
T400	a, b, d
T500	a, c
T600	b, c
T700	a, c
T800	a, b, c, e
T900	a, b, c

Table 5: An example table of transaction data

A FP-tree is constructed as described in the following steps. First, create the root of the tree, labeled "null". The database D is then scanned to create branches for each transaction. The items in each transaction are processed in L^1 order. An item header table is also built so that each item points to its occurrences in the tree by a chain of node-links. The tree obtained after scanning all the transactions is shown in Figure 2 with the associated node-links. The problem of mining frequent itemsets in databases is then transformed to that of mining FP-trees.

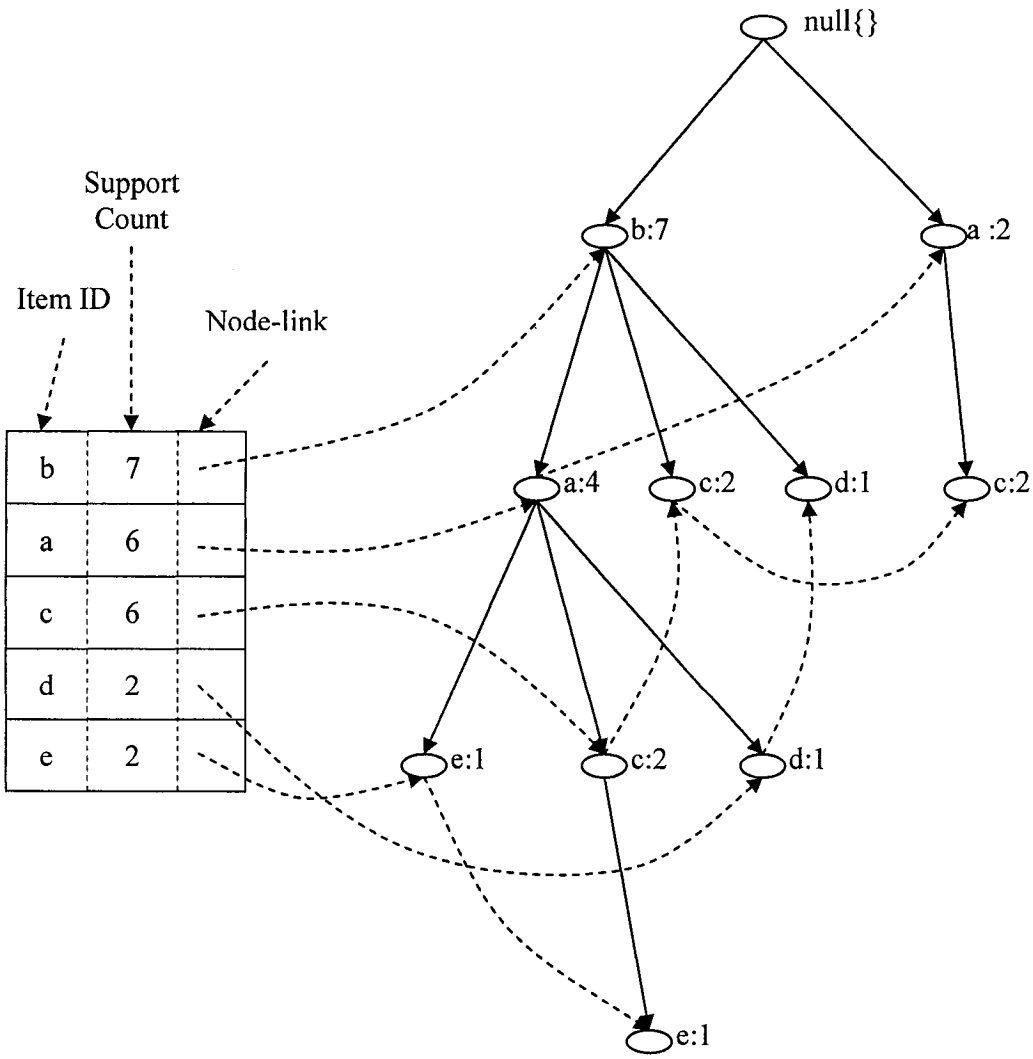


Figure 2: A FP-tree for the database D in Table 5

The mining of the FP-tree starts from each frequent 1-itemset, construct its conditional pattern base, then build its conditional FP-tree, and finally perform mining recursively on the tree. Mining of the FP-tree in our example is summarized in Table 6.

item	Conditional pattern base	Conditional FP-tree	Frequent patterns generated
e	{(b a:1), (b a c:1)}	<b:2, a:2>	{b, e}:2, {a, e}:2, {b, a, e}:2
d	{(b a:1), (b:1)}	<b:2>	{b, c}:2
c	{(b a:2), (b:2), (a:2)}	<b:4, a:2> <a:2>	{b, c}:4, {a, c}:4, {b, a, c}:2
a	{(b:4)}	<b:4>	{b, a}:4

Table 6: Mining the FP-tree

3.3 Incremental Update of Association Rules

The framework of Fast Update Algorithm (FUP, for short) is similar to that of Apriori. It is an iterative process, in which the iteration starts at size-one itemsets, and in each iteration, all the large itemsets of the same size are found. The candidate sets in each iteration are generated based on the large itemsets found at the previous iteration. The main features of FUP are listed as follows.

1. In each iteration, the supports of the size-k frequent itemsets in L are updated against the increment database to filter out the loser, i.e., those itemsets are no longer frequent in the updated database. Only the increment database has to be scanned for this filtering.

2. While scanning the increment, a set C_k of candidate sets is extracted from the transactions in increment database with their support counts. The supports of these sets in C_k are then updated against the original database to find the "new" frequent itemsets.
3. Many sets in C_k can be pruned away by a simple checking on their supports in the incremental database before the update against the original database starts.
4. The size of the updated database is reduced in each iteration by pruning away some items from the transactions in the updated database.

The increment database and the original database are both checked in each iteration, as not only some existing frequent itemsets may become invalid but also some potential new itemsets may become frequent.

3.4 Sampling for Association Rules

Sampling large databases for association rules is a method to mine on a subset of the given data. The basic idea of the sampling approach is to pick a random sample of the given data, and then compute the frequent itemsets in the sample, instead of the whole data. This sampling method provides a trade off between the accuracy and efficiency. The size of the sample is such that the search for frequent itemsets in the whole data can be done in main memory, and so only one scan of the transactions in the whole database is required overall.

It is often important to know the frequency and the confidence of association rules exactly, because even very small differences may be significant in business applications. When relying on results from sampling alone, one also takes the risk of loosing some valid information since their support count in the sample may be below the user specified threshold.

Using a random sample to get approximate frequent itemsets results is fairly straightforward, however, that exact frequencies can be found by analyzing the random sample first and the then compute for the whole database.

3.5 Partition Algorithm

The idea behind Partition algorithm is as follows [14]. Recall that the reason the database needs to be scanned multiple number of times is because the number of possible itemsets to be tested for support is exponentially large, if it must be done in a single scan of the database. However, suppose we are given a small set of potentially large itemsets, say a few thousand itemsets. Then the support for them can be tested in one scan of the database and the actual large itemsets can be discovered. Clearly, this approach will work only if the given set contains all the actual large itemsets.

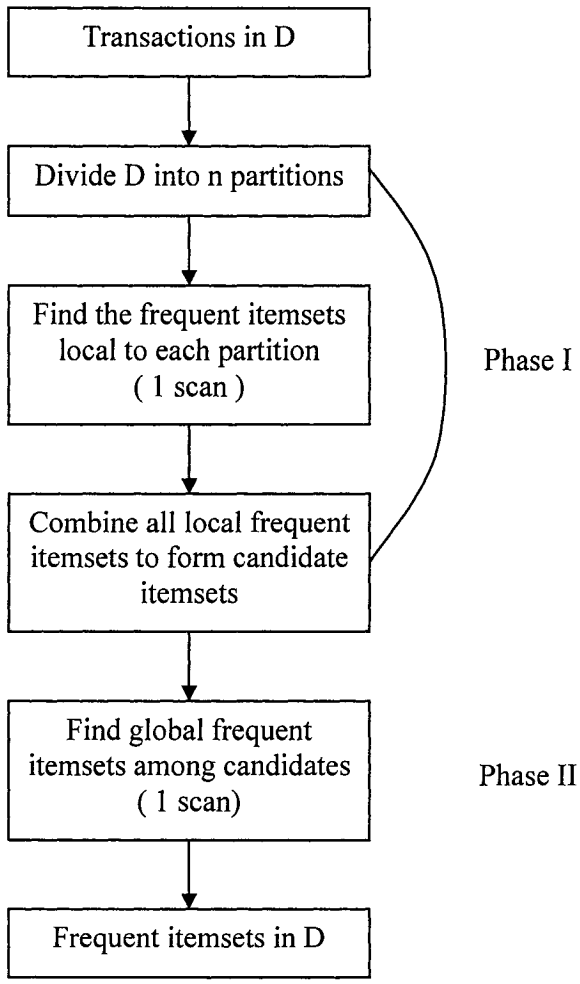


Figure 3: Mining by partitioning the data

Partition algorithm accomplishes this in two scans of the database. In one scan, it generates a set of all potentially large itemsets by scanning the database once. This set is a superset of all large itemsets, i.e., it may contain false positives. But no false negatives are reported. During the second scan, counters for each of these itemsets are set up and their actual support is measured in one scan of the database. The algorithm executes in two phases, as shown in Figure 3:

Phase I: The Partition algorithm logically divides the database into a number of non-overlapping partitions. The partitions are considered one at a time and all large itemsets for that partition are generated. At the end of phase I, these large itemsets are merged to generate a set of all potential large itemsets.

Phase II: The actual supports for these itemsets are determined and the large itemsets are identified. The partition sizes are chosen such that each partition can be accommodated in the main memory so that the partitions are read only once in each phase.

Assume that the transactions are in the form (TID, i_1, i_2, ..., i_k). It is straightforward to adapt the algorithm to the case where the transactions are kept normalized in (TID, itemsets) form. The TIDs are also assumed to be monotonically increasing. This is justified considering the nature of the application. They further assume the database resides on secondary storage and the approximate size of the database in blocks or pages is known in advance.

Chapter 4

System Prototype and Architecture

In this chapter, we describe the architecture of our system prototype, define each of functional modules and show how they are connected to each other in an abstract level in our system architecture.

4.1 System Architecture Overview

Our system is an association rules mining and maintenance system that can mine frequent itemsets along with negative borders from large datasets and maintain such association rules when new datasets are added. With a huge amount of transaction data collected, it is crucial to develop tools for discovery of interesting knowledge from large datasets. This system can be helpful for business decision, market basket analysis. Such kind of analysis can help increase sales by selective marketing, promotion, and arrange shelf space. The system architecture is one of the key elements of our mining and maintenance system. The system architecture defines how the each of the module interacts with each other, and what functionality each module is responsible for performing.

We use the object oriented design method to design our association rules mining and maintenance system. Our system is composed of 5 main functional modules for user-system interaction, which includes Dataset Generator, Dataset Merger, Dataset Splitter, Miner and Maintainer.

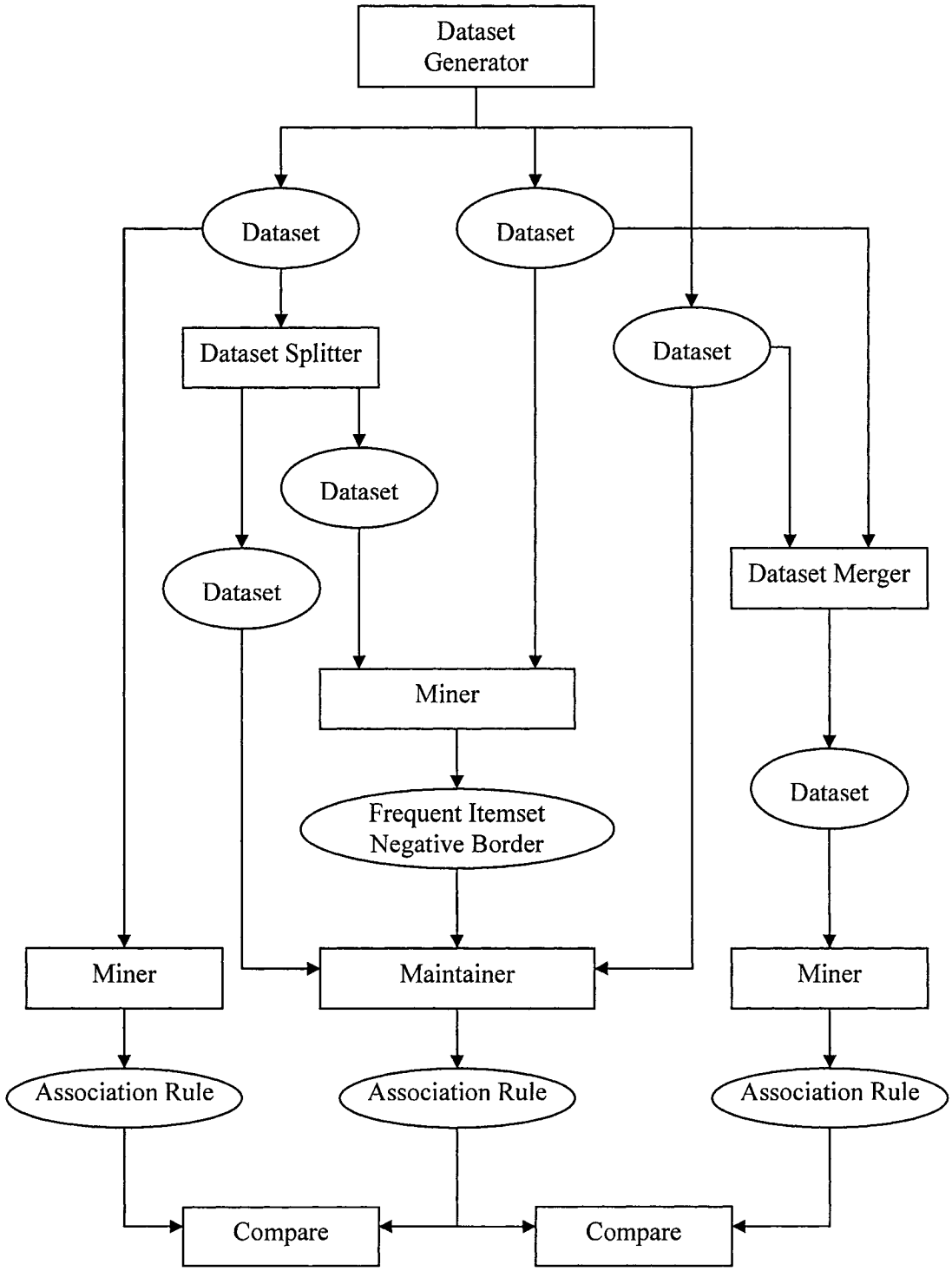


Figure 4: Maintenance System Work-flow Chart

The 5 modules are functionally supported by the component object modules. Each of them also can performance individual task independently. The architecture of our system prototype is shown in Figure 4, which is also a workflow chart of our system.

Our system architecture is based on a processing stream, in that mining or maintenance is broken down into a series of steps with results from each step passed to the next one. There are two main workflows for using this association rules mining and maintenance system and testing the system performance. In the first workflow, we create the desired dataset randomly from the Data Generator by user specified number of transactions and the maximum different items in one transaction. That generated dataset can be split into two parts (two datasets) by the Dataset Splitter. The first part, the bigger part, will be used as the original database. The second part, the smaller part, then is used as the updating database. Note that the new database is normally much smaller than the original database [13]. In the next step, the Miner will mine the first part, the original database, to compute frequent itemsets and negative borders. The computing result is one of the input components for the Maintainer. With the second part, the new database, as another input information, the Maintainer finds the whole association rules for the whole dataset. Mining the whole dataset for association rules directly by using the Miner should give us the same association rule results. We may compare the executive results to see the correctness and efficiency. The second workflow can process as follows. The Dataset Generator creates two datasets that one is used as the original database and another one is used as the new database. The Miner computes the frequent itemsets and negative borders in the original database. Then the Maintainer computes the association rules by given the new database with the frequent items and negative borders of the original database. The

Dataset Merger combines the two datasets, which are created by Data Generator into one entire dataset. Mining the whole dataset from beginning will compute all the association rules for the entire dataset. The last step is to compare the output results and the timing cost.

Each of those five functional modules may also work independently. The functionalities of the five existing modules are described as follows.

4.2 System Functional Modules

4.2.1 Miner

The module Miner is not only the core module in our system prototype architecture but also the main interface to each other module in our system. The Miner module is mainly responsible for mining a given dataset with user specified minimum support and minimum confidence threshold values to get the frequent itemsets, negative borders and association rules. It can be the preceding step of the module Maintainer to transform the frequent itemsets and negative borders of the original database to module Maintainer. It also can help the comparison phase by mining all the association rule for the whole database. Meanwhile, the module Miner is used as the main interface in our system. From it, user may access any of the other functional modules, Dataset Generator, Dataset Merger, Dataset Splitter, and Maintainer.

The main interface, also the interface of module Miner, in our system prototype is shown in Figure 5.

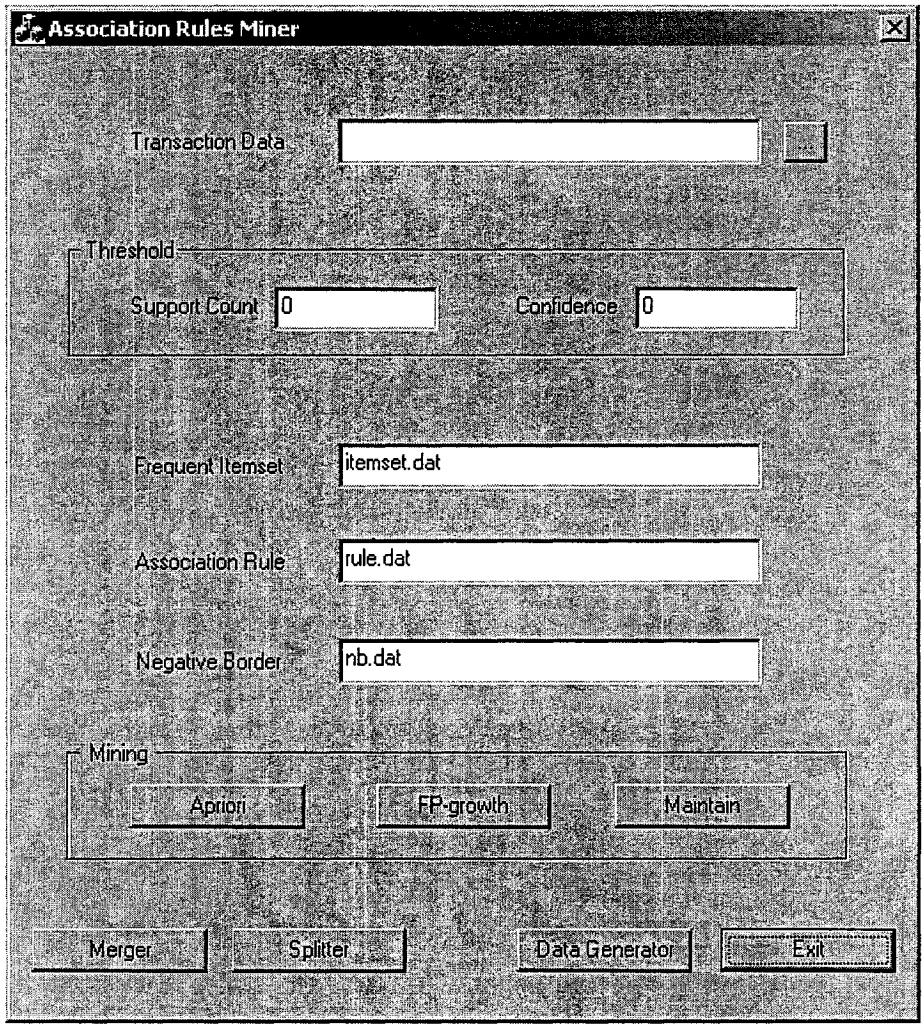


Figure 5: The main interface in our system prototype

4.2.2 Maintainer

The module Maintainer is another core part of our system prototype architecture. This module is used to maintain all the association rules for the whole dataset when a new dataset is added to the original one. With the frequent itemsets and negative borders of the original dataset, the minimum support and lowered minimum support, the module

Maintainer will compute the frequent itemsets and negative borders of the new dataset, then maintain the entire association rule for the whole dataset instead of mining it from the scratch.

The association rule maintenance interface is shown in Figure 6.

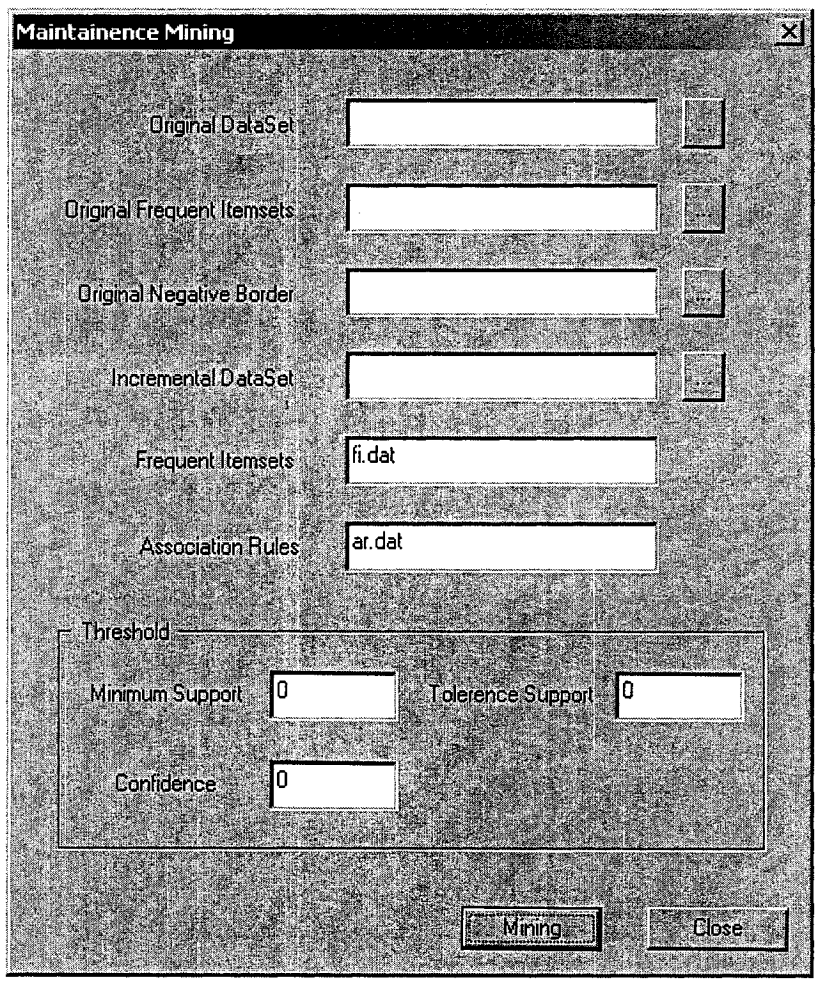


Figure 6: Association rule maintenance interface

4.2.3 Dataset Generator

Dataset Generator is a module in our system prototype to generate the transaction databases randomly. The number of transactions and the maximum numbers of item in each transaction are defined as parameters to this module. This functional module may generate the synthetic data as user desired and for the system usage. The interface of the Dataset Generator is illustrated in Figure 7.

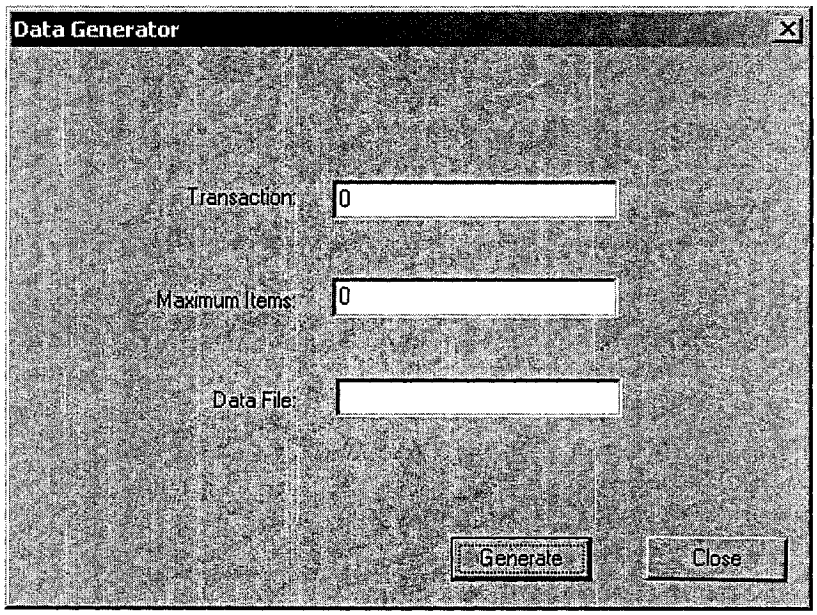


Figure 7: Dataset generator interface

4.2.4 Dataset Merger

Dataset merger is a module to merge two transaction datasets together. It can be used to combine original and new transaction database together for testing the final mining result of the whole database. The interface of the Dataset Merger is shown in Figure 8.

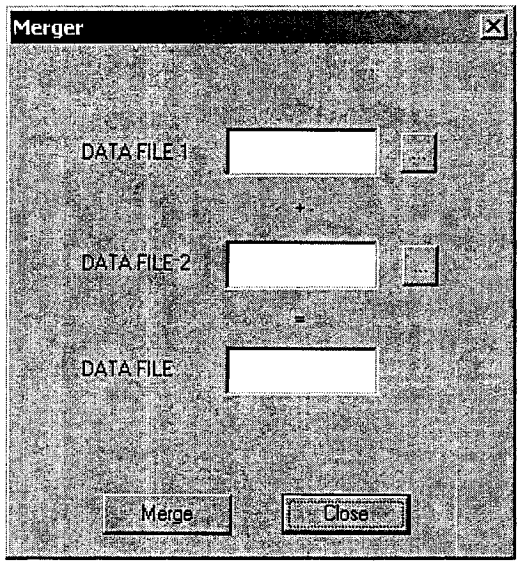


Figure 8: Dataset merger interface

4.2.5 Dataset Splitter

Dataset splitter is a module we developed to split transaction dataset randomly into two datasets by user specified percentage. It can be used to split the whole transaction database into two parts, the first part of which is used as the original database DB1 and the second part is used as the new database DB2. The interface of our Dataset Splitter is shown in Figure 9.

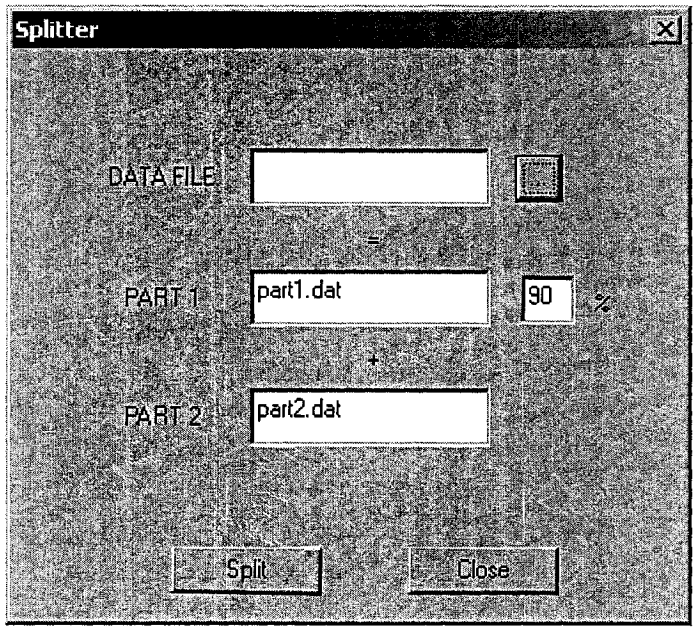


Figure 9: Dataset splitter interface

Chapter 5

Algorithm and Implementation

In this chapter, we present the algorithm in the major functional modules, the data structure we use in the implementation, and some implementation features and details.

5.1 Algorithm

In our algorithm, we first extend the level-wise algorithm like Apriori to compute the negative borders along with the frequent itemsets. We compute the frequent itemsets and negative borders at the same time in each iteration. We keep those frequent itemsets and negative borders of the original database for later maintaining. Then we compute the frequent itemsets and negative borders by using the same method on the transactions that are added to the original database. At last, our algorithm "merges" the frequent itemsets and negative borders containing new and old transaction dataset to identify the actual frequent itemsets for the whole database.

5.1.1 Mining Frequent Itemsets & Negative borders

Intuitively, given a (closed) collection S of itemsets that are frequent, the negative borders contains the itemsets that could potentially be frequent, too. The candidate collections of the level-wise algorithms are, in effect, the negative borders of the collection of all candidates that were not frequent is the negative borders of the collection of frequent sets. The fact is that negative borders need to be evaluated in order to ensure that no frequent sets are missed as special cases. The following algorithm computes the negative borders along with frequent itemsets, using an iterative level-wise approach based on candidate generation.

Procedure mine (D, min_sup; L, NB(L));

Input: D, database of transactions;

min_sup, minimum support.

Output: L, frequent itemsets in D;

NB(L), negative borders in D.

Method:

(1) C_1 = find_1-itemsets(D);

(2) **for each** $c \in C_1$

(3) **if** (count(c) $\geq$ min_sup)

(4) $L_1 = L_1 \cup c$;

(5) **else**

(6) $NB(L_1) = NB(L_1) \cup c$;

(7) **for** ($k = 2$; $L_{k-1} \neq \varnothing$; k++) **begin**

(8) C_k = generate(L_{k-1}, min_sup);

(9) **for each** transaction $t \in D$ **begin**

(10) C_t = subset(C_k, t);

(11) **for each** candidate $c \in C_t$

(12) count(c)++;

(13) **end**

(14) $L_k = \{c \in C_k \mid \text{count}(c) \geq \text{min\_sup}\}$;

(15) $NB(L_k) = \{c \in C_k \mid \text{count}(c) < \text{min\_sup}\}$;

(16) **end**

(17) **return** $L = \bigcup_k L_k$ and $NB = \bigcup_k NB(L_k)$;

In line 8, the above algorithm uses a procedure, *generate*, detailed as follows.

Procedure generate (L_{k-1}, min_sup; C_k);

Input: L_{k-1}, frequent (k-1)-itemsets;

min_sup, minimum support.

Output: C_k, candidate k-itemsets.

Method:

(1) **for each** itemset $l_1 \in L_{k-1}$

(2) **for each** itemset $l_2 \in L_{k-1}$

(3) **if** ($l_1[1] = l_2[1] \wedge l_1[2] = l_2[2] \wedge \ldots \wedge l_1[k-2] = l_2[k-2] \wedge l_1[k-1] < l_2[k-1]$)

(4) **then**

(5) $c = l_1 \bowtie l_2$; //join: generate candidates

(6) **if** has_infrequent_subset(c, L_{k-1}) **then**

(7) delete c;

(8) **else** add c to C_k;

(9) **return** C_k;

The generate function takes as argument, the set $L_{k-1}$of all large (k-1)-itemsets and returns a superset C_k of the set of all frequent k-itemsets.

The procedure uses *has_infrequent_subset* in line 6 above, detailed as follows.

Procedure has_infrequent_subset (c, L_{k-1}; True/False);

Input: c, candidate k-itemset;

L_{k-1}, frequent (k-1)-itemset.

Output: TRUE or FALSE, Boolean Value.

Method:

(1) **for each** (k-1)-subset s of c

(2) **if** s $\notin$ L_{k-1} **then**

(3) **return** TRUE;

(4) **return** FALSE;

The above pseudocode is the algorithm for mining frequent itemsets along with negative borders. We also provide the related procedures. Step 1 gets all the 1-itemsets with their support counts. In Step 2 to 6, we finds frequent 1-itemset L_1 and their negative borders $NB(L_1)$. In Step 7 to 16, L_{k-1} used to generate candidates C_k in order to find L_k and

$NB(L_k)$. The generate procedure generates the candidates and uses the Apriori property to eliminate those sets having a subset that is not frequent (Step 8). This procedure is described below. Once all the candidates have been generated, the database is scanned (Step 9) once again. For each transaction, a subset function is used to find all subsets of the transaction that are candidates (Step 10), and the count for each of these candidates is accumulated (Step 11 and Step 12). Finally, all those candidates satisfying minimum support form the set of frequent itemsets, L. All those candidates not in L form the set of negative borders NB(L).

The generate procedure performs two kinds of tasks, namely join and prune, as described in Chapter 3. In the join component, joins L_{k-1} with L_{k-1} to generate potential candidates (steps 1–5). The prune component (steps 6–8) uses the Apriori property to remove candidates that have a subset that is not frequent. The test for infrequent subsets is shown in procedure has_infrequent_subset.

We give a concrete example by using the transaction database D of Table 5. There are 9 transactions in this database, that is, $|D| = 9$. We use Figure 10 to illustrate the algorithm for finding frequent itemsets in D.

1. In the first iteration of the algorithm, each item is a member of the set of candidate 1-itemsets, C_1. The algorithm simply scans transactions in D in order to count the number of occurrences of each item.
2. We assume that the given minimum support count threshold is 2 (i.e., min_sup = 2 / 9 = 22%). The set of frequent 1-itemsets, L_1, and their negative borders, $NB(L_1)$, can then be determined by checking the support counts of candidate 1-itemsets with the minimum support count.

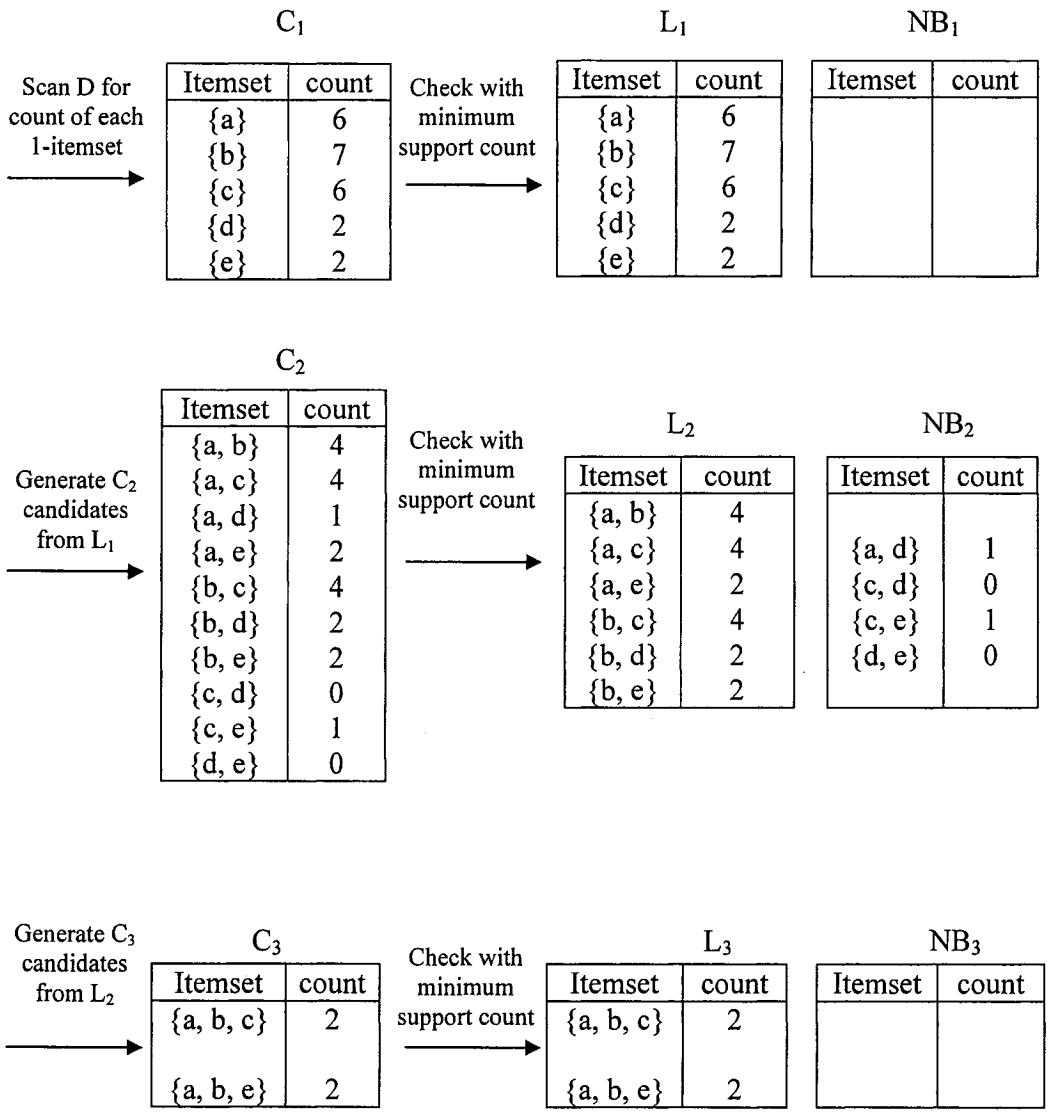


Figure 10: Generation of candidate itemsets, frequent itemsets and negatieve border

3. To discover the set of frequent 2-itemsets, L_2, the algorithm uses $L_1 \bowtie L_1$ to generate all the candidate set of 2-itemsets.
4. We then scan the transactions in D to accumulate the support count of each candidate itemset in C_2. The set of frequent 2-itemsets, L_2, and negative borders, $NB(L_2)$, can be determined, through those candidate 2-itemsets in C_2 respecting the minimum support.
5. The generation of the set of candidate 3-itemsets, C_3, is detailed in Figure 4. First, let $C_3 = L_2 \bowtie L_2 = \{\{a, b, c\}, \{a, b, e\}, \{a, c, e\}, \{b, c, d\}, \{b, c, e\}, \{b, d, e\}\}$. Based on the Apriori property that all subsets of a frequent itemset must also be frequent, we determine that the four candidate 3-itemsets, $\{\{a, c, e\}, \{b, c, d\}, \{b, c, e\}, \{b, d, e\}\}$, can not possibly be frequent. Thus, we remove them from C_3 directly, thereby saving the effort of unnecessarily obtaining their counts during the subsequent scan of D to determine L_3. We only need to check if the $(k - 1)$-subsets of a given candidate k-itemset are frequent because of the level-wise search strategy.
6. The transaction database D is scanned in order to identify the frequent 3-itemsets, L_3, and their negative borders.
7. The algorithm uses $L_3 \bowtie L_3$ to generate a candidate set of 4-itemsets, C_4. Although we get {a, b, c, e} after the join, this itemset is pruned since one of its subset {b, c, e} is not frequent. Therefore, C_4 is empty and the algorithm terminates. All of the frequent itemsets are found.

5.1.2 Maintenance in Association Rules Mining

We now apply the concept of negative borders and use lowered minimum support to find frequent itemsets for maintenance of association rules. There are four cases of itemsets we should consider for the maintenance of association rules mining:

Case 1: the itemsets which are frequent in both DB1 and DB2.

Case 2: the itemsets which are frequent in DB1 but not in DB2.

Case 3: the itemsets which are frequent in DB2 but not in DB1.

Case 4: the itemsets which are frequent in neither DB1 nor DB2

According to our analysis, case 4 cannot be used to generate any association rule, since the itemsets will not be frequent in DB in this case. For cases 1 and 2, the frequent itemset can be obtained easily by just scanning the new database DB2 once, then compute count(X) = count1(X) + count2(X). Case 3 may pose problem for the maintenance of association rules which needs further consideration. In certain situation, we may need to re-scan the original database once more, explained as follows.

In case 3, assume X is an itemset of DB2 such that X is neither in L1 nor in NB(L1). Let d1 be the number of transactions in DB1, d2 be the number of transactions in DB2, and t_sup denote the "lowered" minimum support, where $0 < t\_sup < min\_sup$. If $count2(X) < (d1 + d2) \times min\_sup - d1 \times t\_sup = d2 \times min\_sup + d1 \times (min\_sup - t\_sup)$, we do not need to scan DB1 again. Otherwise, this rescan of DB1 is inevitable.

The above steps of determining frequent itemsets L in DB are formally stated in the following algorithm.

Procedure maintain (DB1, L1, NB1, DB2, min_sup, t_sup; L);

Input: DB1, original database;

L1, frequent itemsets in DB1;

NB1, negative borders in DB1;

DB2, updating database;

min_sup, minimum support;

t_sup, lowered minimum support;

Output: L, frequent itemsets in DB;

Method:

```
(1)   mine (DB2, t_sup; L2, NB2);
(2)   for each candidate itemsets c ⊂ L2 ∪ NB2 begin
(3)     if (c ⊂ L1∪ NB1)
(4)       count(c) = count1(c) + count2(c);
(5)       L = L ∪{c | count(c) ≥ min_sup};
(6)     else if (count2(c) ≥ d2 × min_sup + d1 × (min_sup – t_sup))
(7)       DB1.scan(c);
(8)       count(c) = count1(c) + count2(c);
(9)       L = L ∪{c | count(c) ≥ min_sup};
(10)  end
(11)  return L;
```

5.1.3 Dataset Generator

Dataset Generator is a module in our system prototype to generate randomly the transaction databases. The number of transactions and the maximum numbers of item in each transaction are defined as parameters to this module.

Procedure data_generator (d, max_Item; DB);

Input: d, number of transactions;

max_Item, maximum items in each transaction.

Output: DB, database of transactions.

Method:

```
(1)   for (i = 0; i < d; i++) begin
(2)     curr_Item_Num = 1 + rand()%max_Item;
(3)     for (j = 1; j < curr_Item_Num; j++) begin
(4)       Ij = 1 + rand()%max_Item;
(5)       if (Ij ∉ Ti)
(6)         DB.input(Ij);
(7)     end
(8)     DB.input( '\n');
(9)   end
(10)  return DB;
```

5.1.4 Dataset Merger

Dataset merger is a module to merge two transaction datasets together. It can be used to combine old and new transaction database together for testing the final result of mining the whole database.

Procedure data_merger (DB1, DB2; DB);

Input: DB1, the old dataset file;

DB2, the new dataset file.

Output: DB, the whole dataset file.

Method:

(1) DB1.open();

(2) **for each** transaction T_i in DB1

(3) DB = DB $\cup$ T_i;

(4) DB1.close();

(5) DB2.open();

(6) **for each** transactions T_j in DB2

(7) DB = DB $\cup$ T_j;

(8) DB2.close();

(9) **return** DB;

5.1.5 Dataset Splitter

Dataset splitter is a module we developed to split transaction dataset randomly into two datasets by user specified percentage. It can be used to split the whole transaction database into two parts, the first part of which is used as the original database DB1 and the second part is used as the new database DB2. The interface of our Dataset Splitter is shown in Figure 9. It shows that the user wishes 90% of the *input* dataset to be randomly selected as part1.dat, and the rest 10% as part2.dat.

Procedure data_splitter (DB; DB1, DB2);

Input: DB, the whole dataset file;

percent, the percentage of the first part after splitting.

Output: DB1, the first part dataset;

DB2, the second part dataset.

Method:

(1) DB.open();

(2) **for each** transaction T_i in DB

(3) count++;

(4) count1 = count × percent;

(5) count2 = count × (1 – percent);

(6) **for each** transactions T_i in DB **begin**

(7) **if** (part2. is_full())

(8) part = part1;

```
(9)     if ( part1. is_full())
(10)        part = part2;
(11)    part = random_part_chosen();
(12)    if (part = part1)
(13)        DB1 = DB1 ∪ Ti;
(14)    else if (part = part2)
(15)        DB2 = DB2 ∪ Ti;
(16)    end
(17)    return DB1 and DB2;
```

5.2 Data Structure

The candidate generation and support counting processes require an efficient data structure in which all the candidate itemsets are stored. A central data structure we could use was trie or hash-tree. Concerning speed, memory need and sensitivity of the parameters, tries were proven to outperform hash-trees [17]. In this section, we introduce the trie data structure. All implementations of our frequent itemsets mining and maintenance algorithm presented in this thesis are implemented using the trie data structure.

5.2.1 Trie

The data structure that we used is a trie, which is a prefix-tree [18, 19, 20]. The data structure *trie* was originally introduced to efficiently store and efficiently retrieve words of a dictionary [21]. A trie is a rooted, downward directed tree like a hash-tree. The root is defined to be at depth 0, and a node at depth d can point to nodes at depth $d + 1$. A pointer is also called *edge* or *link*, and is labelled by a letter. If node u points to node v, then we call u the parent of v, and v is a child node of u.

Every leaf node l represents a word which is the concatenation of the letters in the path from the root to l. Note that if the first k letters are the same in two words, then the first k steps on their paths are the same. Tries are suitable to store and retrieve not only words, but any finite ordered sets. In this setting, a link is labelled by an element of the set, and the trie contains a set if there exists a path where the links are labelled by the elements of the set, in increasing order.

In a trie, every k-itemset has a node associated with it, as does its (k−1)–prefix. The empty itemset is the root. All the 1-itemsets are attached to the root, and the item they represent labels their branches. Every other k-itemset is attached to its (k−1)-prefix. Every node stores the last item in the itemset it represents together with its support and its branches. The branches of a node can be implemented using data structures such as a binary search tree or a vector.

At a certain iteration k, all candidate k-itemsets are stored at depth k in the trie. In order to find the candidate-itemsets that are contained in a transaction T, we may start at the root. To process a transaction for a node of the trie, we follow the branch corresponding to the first item in the transaction and process the remainder of the transaction recursively

for that branch, and then discard the first item of the transaction and process it recursively for the node itself [20].

Since all itemsets of size k have the same (k−1)-prefix that are represented by the branches of the same node, the join step of the candidate generation procedure becomes very simple using a trie. Indeed, to generate all candidate itemsets with (k−1)-prefix X, we simply copy all siblings of the node that represents X as branches of that node. Moreover, we can try to minimize the number of such siblings by reordering the items in the database in support ascending order [19, 20]. This reduces the number of itemsets that is generated during the join step, and hence implicitly reduces the number of times the prune step needs to be performed. Also, to find the node representing a specific k-itemset in the trie, we have to perform k searches within a set of branches. Obviously, the performance of such a search can be improved when these sets are kept as small as possible. An in depth study on the implementation details of a trie for Apriori can be found in [20].

5.2.2 Determining the Support Parameter

In our data mining context, the alphabet is the ordered set of all items I. A candidate *k*-itemset, $C = \{i_1 < i_2 < \ldots < i_k\}$, can be viewed as the word $i_1 i_2 \ldots i_k$ composed of letters from I. Every inner node can represent an important itemset [21].

We present a trie that stores the candidates {A, B, D}, {A, C, E}, {A, C, F}, {K, M, N}, {A, C, M} in Figure 11. Numbers in the nodes serve as identifiers and will be used in the implementation of the trie. The details of building a trie can be found in [21].

In support count method we consider the transactions one at a time. For a transaction t, we take all ordered k-subsets X of t and search for them in the trie. If X is found as a candidate, then we increase the support count of this candidate by one. Here, we may not generate all k-subsets of t, rather we perform early prune if possible.

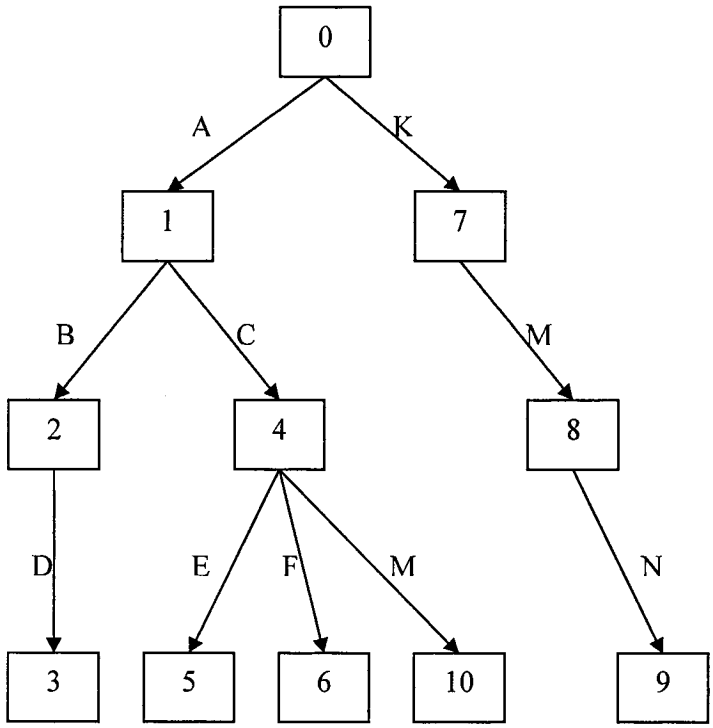


Figure 11: A trie structure containing 5 candidates

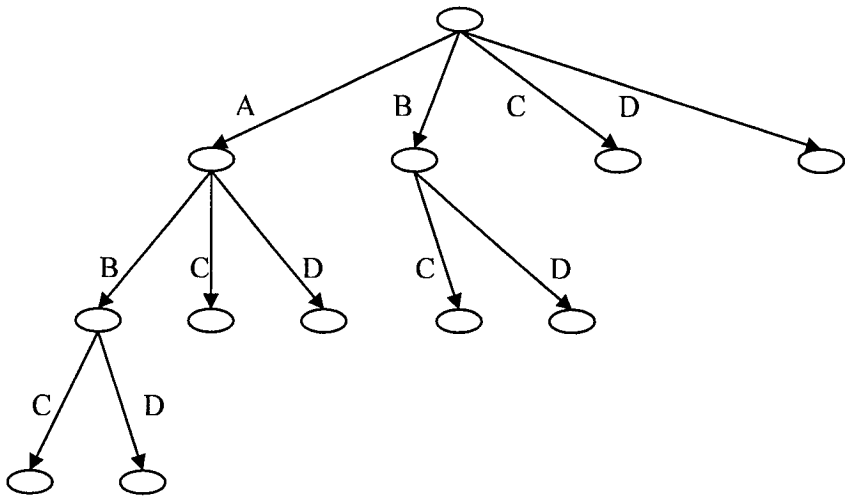
Tries may store not only candidates but also frequent itemsets. The advantages of this are as follows:

1. Candidate generation becomes easy and fast. We can generate candidates from pairs of nodes that have the same parents. That means the two sets are the same except the last item.

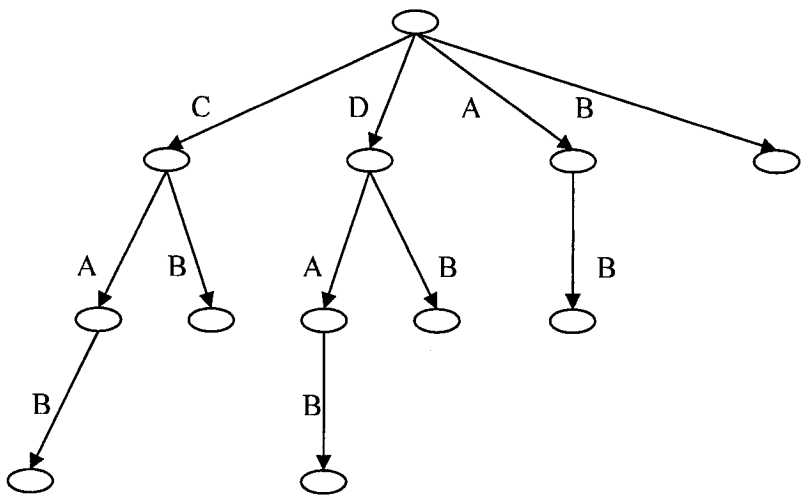
2. Association rules are produced much faster, since retrieving a support of an itemset is faster. Note that the trie was originally introduced to quickly decide if a word is in a dictionary.
3. Only one data structure has to be implemented, hence the code is simpler and easier to maintain.
4. We may immediately generate the *negative border*, which plays an important role in our association rule mining and maintenance algorithm [12, 22].

5.2.3 Frequency Order

It happens that we have to find the node that represents a given itemset. For instance, the subsets of the generated candidate have to be checked during candidate generation, or we want to determine the association rules. We traverse starting from the root, and we have to find the edge at depth d whose label is the same as the d^{th} element of the itemset. Theoretically, binary search is the fastest way to find an item in an ordered list. In linear search, we read the first item, and compare with the searched item. If it is smaller, then there is no edge with this item. If greater, we search forward, if they equal then the search is finished. If we are in the worst case, the most frequent item has the highest order, and we have to march to the end of the branch, whenever this item is searched for.



Order: <A, B, C, D>



Order: <C, D, A, B>

Figure 12: Different coding results in different tries

On the whole, the search will be faster if the order of items corresponds to a frequency order of items. We know exactly the frequency order after the first read of the whole database. Thus everything is ready to build the trie with the frequency order instead of the original order. The frequency order of an item i is f[i], if i is the $f[i]^{th}$ most frequent item. Storing frequency order, and their inverses, increases the memory need slightly. In return, it increases the speed of retrieving the occurrence of the itemsets.

Frequency order also affects the structure of the trie, and consequently the running time of support count. For example, we suppose that two candidate itemsets of size 3 are generated: {A, B, C}, {A, B, D}. Different tries are generated if the items have different order. Figure 12 presents the tries generated by the two different orders. If we want to find that which candidates are stored in a transaction {A, B, C, D}, then 5 nodes are visited in the first case and 7 in the second case. That does say that we will find the candidates faster in the first case, because nodes are not so "large" in the second case, which means that they have fewer edges. Processing a node is faster in the second case, but more nodes have to be visited.

In general, if the orders of the item correspond to the frequency order, then the result trie will be unbalanced while in the other case, the trie will be rather balanced. Neither one is clearly more advantageous than the other. We choose to build unbalanced trie, because it travels through fewer nodes. This means that fewer recursive steps, which is a slow operation compared to finding proper edges at a node.

It is advantageous to reorder frequent items according to ascending order of their frequencies because candidate generation will be faster. The first step of candidate generation is to find siblings and take the union of the itemset represented by them. It is

easy to prove that there are less sibling relations in a balanced trie, therefore fewer unions are generated and the second step of the candidate generation is invoked fewer times. For example in Figure 11, union would be generated and then deleted in the first case and none would be generated in the second.

Altogether frequency codes have advantages and disadvantages. They accelerate retrieving the support of an itemset, which can be useful in association rule generation or in frequent itemset mining. This, however, slows down candidate generation. Since candidate generation is much faster that support count, the speed decrease maybe not noticeable.

5.3 Implementation Features

Our association rules mining and maintenance system is implemented in an object-oriented manner in language C/C++. We heavily use STL (Standard Template Library), such as <vector>, <set>, <map>, and <algorithm> in our implementation.

The transactions in a file are first stored in a vector, then transform to a Class Transaction object. If we choose to store input, which is the default, the reduced transactions are stored in a map<vector, unsigned long>, where the second parameter is the number of times that reduced basket occurred. A better solution would be to apply trie, because map does not make use of the fact that two transactions can have the same prefixes. Hence insertion of a transaction would be faster, and the memory need would be smaller, since the same prefixes would be stored just once.

The class Item is used to implement trie data structure that actually can be programmed in many ways. We have chosen to implement it with vectors and arrays. It is simple, fast, and minimizes the memory need [21]. Each node is described by the same element of the vectors or a row of the arrays. The root belongs to the 0^{th} element of each vector according to the definition and description of trie data structure.

For vectors, we use vector class offered by the STL. But arrays are stored in a traditional C way because it is not a fixed-size array, which may cause the calloc, realloc commands in the code. Each row is as long as many edges the node has, and new rows are inserted as the trie grows during candidate generation. A better way would be if the arrays were implemented as a vector of vectors. The code would be easier to understand and shorter, because the algorithms of STL could also be used directly. However, the algorithm would be slower because determining a value of the array takes more time. In our implementation, we use a vector and an array, temp counter array, for 2-itemset candidates to determine the support count of 1-itemset and 2-itemset candidates efficiently.

The vector and array description of a trie makes it possible to give a fast implementation of the basic functions, like candidate generation, support count, and so on. For example, deleting infrequent nodes and pulling the vectors together is achieved by a single scan of the vectors.

Chapter 6

Experimental Results

To assess the merits of our proposed method for computing & maintenance of frequent itemsets, we conducted a number of experiments and studied the performance. For this, we used a Pentium® 4 2.80GHZ, 512M of RAM on Window XP Professional. In this section, we report our experimental results.

In our experiments, we used real data from FIMI [23] as well as synthetic data. The dataset we used from [23] in our experiments were T10I4D100K.dat with 100,000 transactions and retail.dat with 88,162 transactions.

As shown in Figures 13 and 14, the experimental results show that computing negative borders along with mining frequent itemset does not require much more time compare with the frequent itemset mining by using level-wise algorithm. Computing the negative borders of a set of frequent itemset can be accomplished by repeating the join and prune steps of the generate function. The negative borders consist of all itemsets that were candidates of the level-wise method which did not have enough support. And the generate function uses only frequent (k-1)-itemsets to compute candidate k-itemsets. The computation of negative borders can be done using only the set of frequent itemsets and the database need not be scanned again for this.

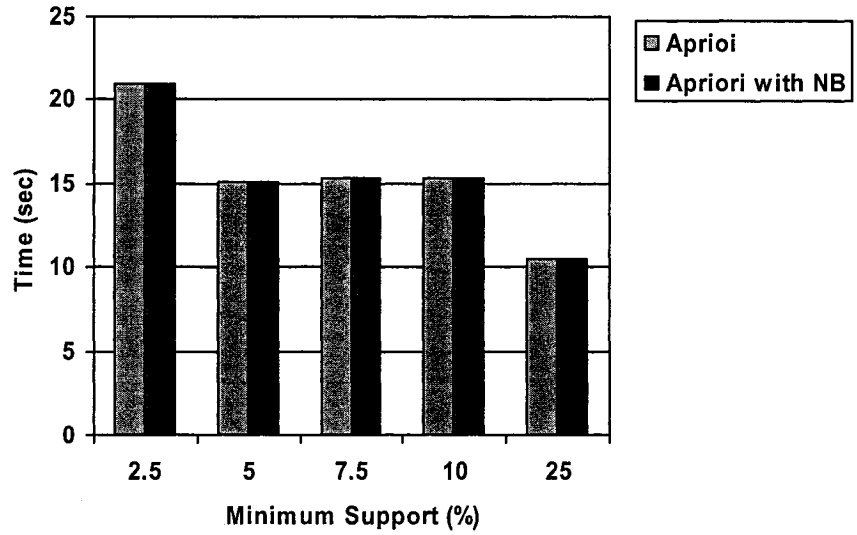


Figure 13: Execution time comparisons on dataset retail.dat

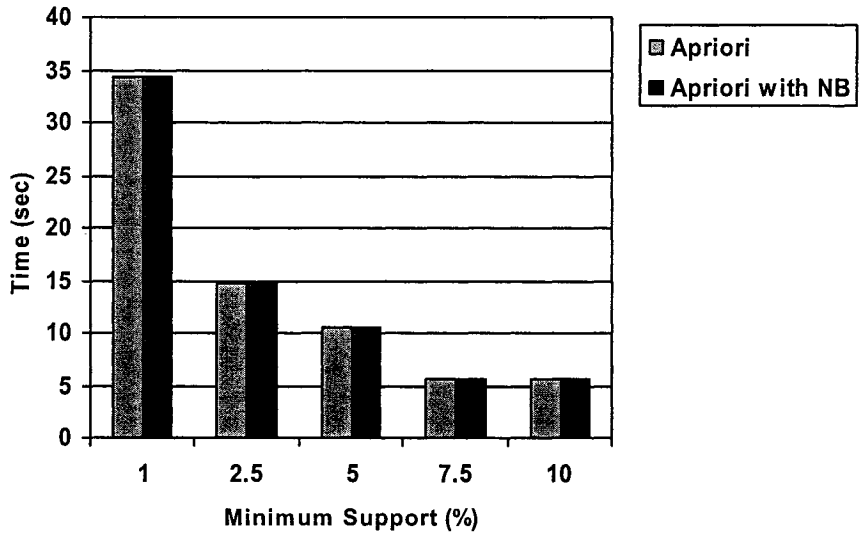


Figure 14: Execution time comparisons on dataset T10I4D100K.dat

Comparing the execution time of our algorithm to that of the Apriori algorithm on the whole data set, we also developed random file splitting by percentage. We split T10I4D100K.dat and retail.dat into 90% and 10%, where the 90% part was used as the original dataset DB1, and the 10% part was used as the new dataset DB2.

As expected, we can see from Figures 15 and 16, that the maintenance algorithm is far more efficient than the Apriori algorithm and FP-growth algorithm, on both synthetic data and real data. The algorithm shows better speed up for lower support threshold than higher support threshold. The speed up is higher for smaller size update dataset since the maintenance algorithm needs to process less data. For higher support threshold values, the number of frequent itemsets is less and hence it is less costly to run Apriori on the whole database.

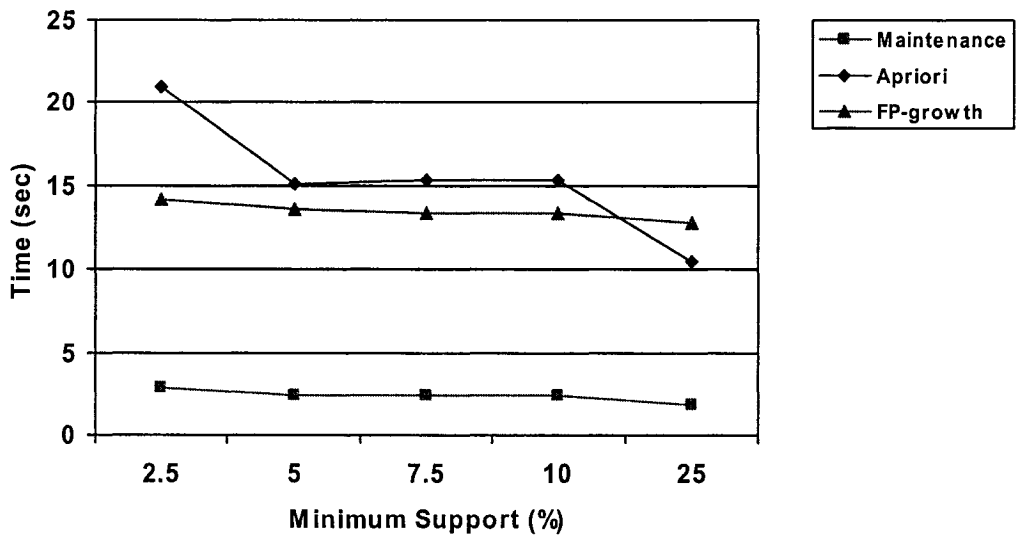


Figure 15: Execution times of Apriori and our algorithm on dataset retail.dat

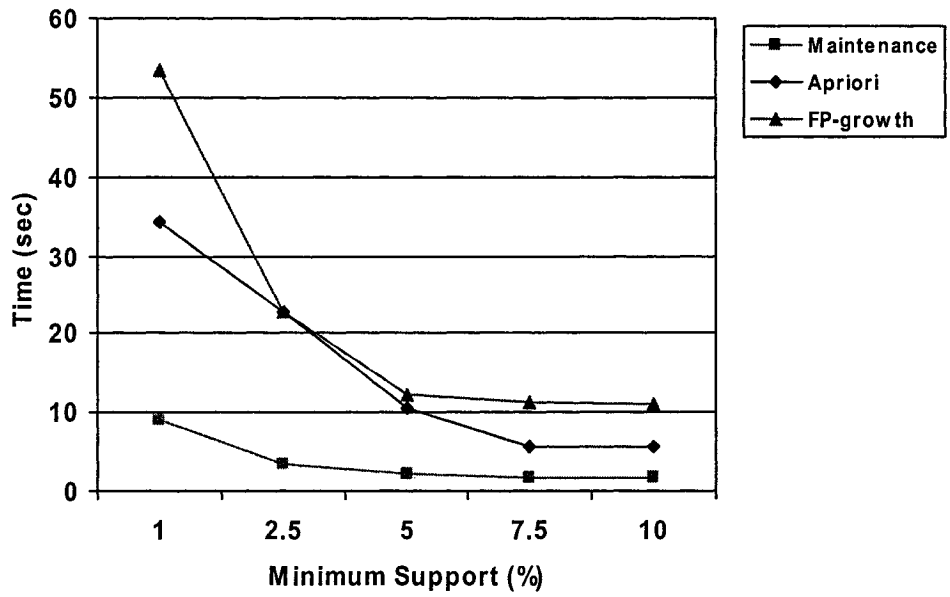


Figure 16: Execution times of Apriori and our algorithm on dataset T10I4D100K.dat

When the dataset is too huge to fit into the memory, our approach can be used to partition the dataset into pieces each of which fits into the main memory. After computing the frequent itemsets for each partition, we "merge" the results to get the frequent itemsets for the whole dataset. We also studied the performance for different percentage of the original database and the updating database on datasets T10I4D100K.dat and retail.dat. The size of DB1 and DB2 range from 90% and 10%, 80% and 20%, ..., to 50% and 50%. The results indicate that the time increase as the size of the update database increases, as shown in Figures 17 and 18. When DB1 and DB2 are about the same size, i.e., 50% and 50% or 60% and 40%, we see from both Figures 16 and 17 that there could be some performance degradation. The cost of our algorithm mainly depends on the size of the update datasets.

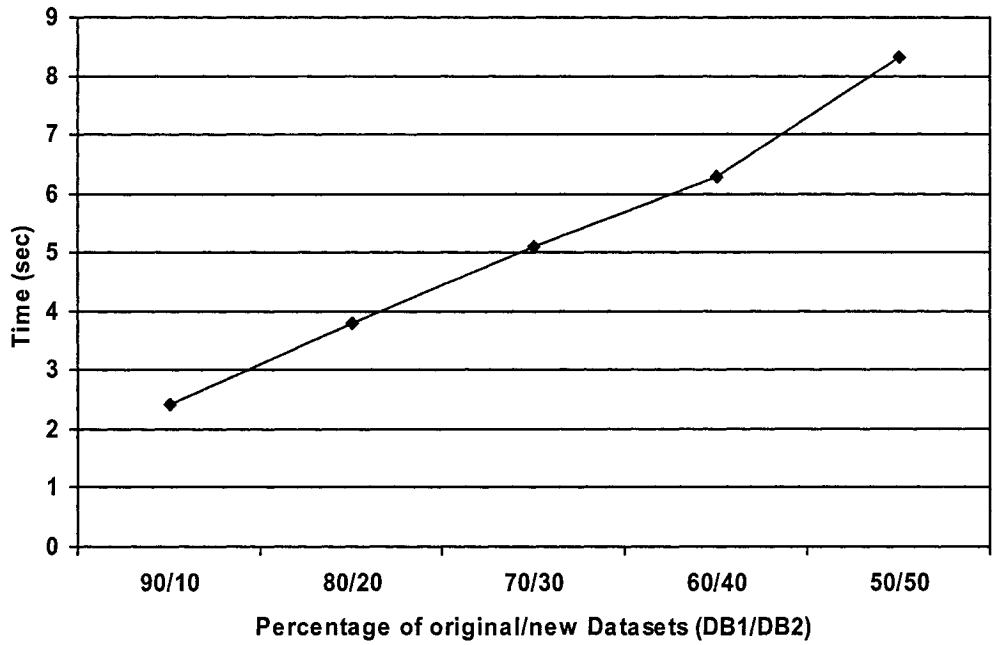


Figure 17: Execution times of different percentage on dataset retail.dat

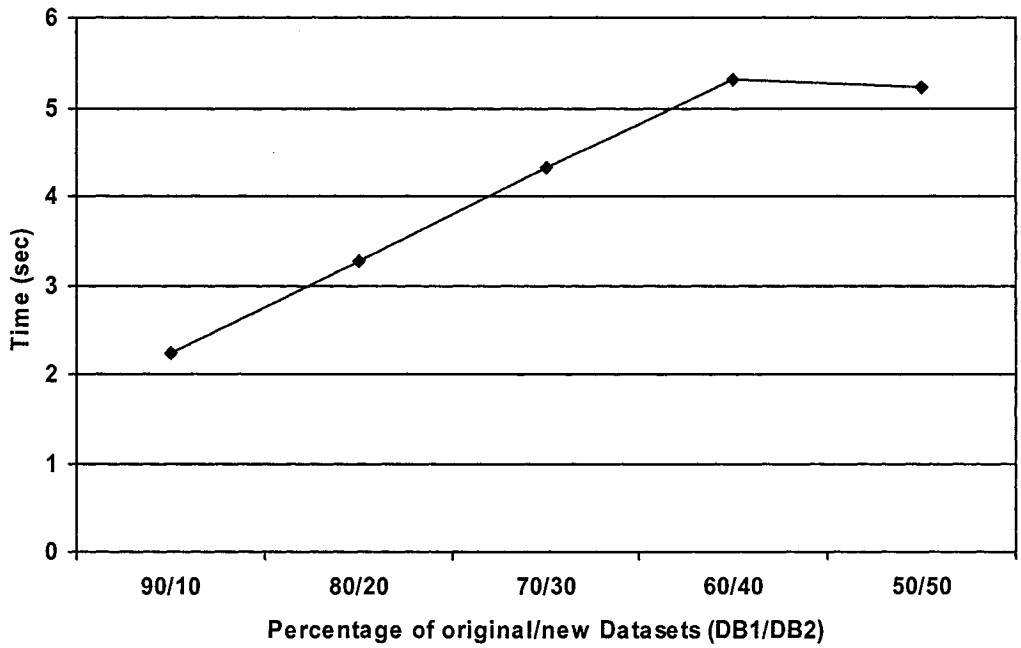


Figure 18: Execution times of different percentage on dataset T10I4D100K.dat

Comparison with the framework of FUP [13], which is similar to that of Apriori and contains a number of iterations, each iteration is associated with a complete scan of the whole database and in iteration k all the large k-itemsets are found. The candidate sets for iteration k+1 are generated based on the frequent itemsets found in iteration k. The speed-up of FUP over Apriori can be mainly attributed to the reduction in the number of candidate itemsets. It uses the frequent itemsets of the original database the filter and prune the candidate itemsets generated by Apriori. However, FUP may require *O(n)* passes over the database where *n* is the size of the maximal large itemsets. The most important feature of our maintenance algorithm is that the original database is scanned only when required. Computing the negative border closure may increase the size of the candidate. But only those itemsets which were not covered by the negative borders need to be checked against the whole database. Even in some rare cases, the size of the candidate set in the final scan could potentially be much smaller than in FUP.

Chapter 7

Efficient FIM Based on Partitioning

In previous chapters, we proposed a technique based on the concept of negative border, frequent itemsets and hence association rules in large datasets of transactions, when the dataset grows by addition of new transactions, which otherwise invalidate some association rules identified and/or introduce some new such rules.

We first extend the level-wise algorithm such as Apriori to compute the negative borders along with the frequent itemsets of the original dataset for later maintenance. Then we compute the frequent itemsets and negative borders by using the same method on the transactions that are added to the original database. At last, our algorithm "merges" the frequent itemsets and negative borders containing new and old transaction dataset to identify the actual frequent itemsets for the whole dataset. It is important to maintain such discovered association rules in large databases without re-computing the whole database from scratch when new data is added.

In this chapter, we develop a method to efficiently determine frequent itemsets for large datasets, which do not fit into main memory. The basic idea is to divide such datasets into partitions and apply the proposed technique to these partitions, then "merge" the result [24].

7.1 Previous Work on Partition

Savasere et al. [14] proposed an efficient algorithm for mining association rules in large databases, which is based on partitioning. In such cases, we need in general to scan the database multiple times to find frequent itemsets. However, suppose a small set of potentially large itemsets is given. Then the support for them can be determined in just one scan of the dataset to identify the actual large itemsets. Clearly, this approach will work only if the given set contains all the actual large itemsets.

Their partition algorithm accomplishes this in two scans of the database. It first generates a set of all potentially large itemsets by scanning the database once. This set is a superset of all large itemsets. During the second scan, counters for each of these itemsets are set up and their actual support is measured in the scan of the database.

Their algorithm therefore executes in two phases. In the first phase, their partition algorithm logically divides the database into a number of non-overlapping partitions. Each partition is then processed which results in generation of all the frequent itemsets for that partition. At the end of this phase, those large itemsets generated from different partitions are merged to produce a set of all potential large itemsets. In phase II, the actual supports for these itemsets are determined from the database and the large itemsets are identified. The partition sizes are chosen such that each partition can be accommodated in the main memory and only once in each phase.

Compare to their partition algorithm, our method keeps local frequent itemsets and negative borders along with their support counts for each partition using a lowered minimum support, but not by bookkeeping the transaction ID that is from which partition.

And we need only one scan of the entire dataset in the most cases while they need scan the whole dataset twice, where one scan in each of the two phases.

7.2 Our Partition Algorithm

The idea behind our partition algorithm is as follows. Since the dataset is assumed to be too large to fit into the main memory, we first partition it into a number of non-overlapping parts, and for each part we find all actual local frequent itemsets in phase I. As shown in Figures 13 and 14 in Chapter 6, computing negative borders along with mining frequent itemset does not require much more time compared to the time required to compute the frequent itemsets. We keep the local frequent itemsets and their negative borders along with their support counts in each partition. If the lowered minimum support is suitably chosen, we can keep enough information from each partition to use for generating frequent itemsets. In this case, our partition algorithm discovers the actual globe frequent itemsets in just one more scan of the entire dataset. That is, we compute frequent itemsets in no more than two scans, when the dataset is too large to fit into the main memory.

We therefore proceed as follows, each partition is read into the main memory for which we generate all frequent itemsets and negative borders using the lowered minimum support. After we have discovered all the local frequent itemsets and negative borders along with their support counts from all the partitions, we merge all these candidate itemsets, in the second phase, to identify the global support for each of them.

At the end of phase I, all those candidate frequent itemsets and negative borders are discovered rearranged in the lexicographic order. This is because there might be different

frequency order in each partition, so even the same itemset may have different order from different partitions. To speed up the merging process in phase II, we simply sort each of these candidate frequent itemsets and negative borders in lexicographic order, using some proper data structure.

We next provide some definitions. A partition p of a dataset D of transactions is any non-empty subset of D. Two different partitions of D are said to be non-overlapping if $p_i \cap p_j = \varnothing$, for $i \neq j$. For any itemset, we define lowered minimum support, low_sup, as the fraction of transactions containing that itemset in a partition. A local candidate itemset for a given partition is defined as a set of local frequent itemsets and negative borders in that partition, generated for some lowered minimum support. We define minimum support, min_sup, as the global support in the context of the entire dataset D. Our goal is to find all global frequent itemsets in D, by using local candidate itemsets found from partitions of D. Table 7 includes notations for our partition algorithm.

L^G	Set of global frequent itemsets
C^G	Set of global candidate itemsets
C^P	Set of local candidate in partition p which includes local frequent itemsets and negative borders

Table 7: Notations used in our partition-based mining algorithm

We represent sets of itemsets by capital letters and individual itemsets by small letters. We omit the partition number when it refers to the local itemset when there is no ambiguity. We next present our partition-based mining algorithm.

Procedure Partition-Mining(D, low_sup, min_sup, n; L^G)

Input: D, database of transactions;

low_sup, lowered minimum support;

min_sup, minimum support;

n, number of partitions.

Output: L^G, global frequent itemsets in D.

Method:

(1) P = partition_dataset(D, n);

(2) **for** i = 1 to n **begin** // Phase I

(3) read_in_partition($p_i \in P$);

(4) C^i = mine(p_i, low_sup);

(5) $C^G = \bigcup_i C^i$;

(6) **end**

(7) C = merge(C^G); // Phase II

(8) $L^G = \{ c \in C \mid count(c) \geq min\_sup\}$;

(9) **return** L^G;

This algorithm takes as the input parameters, the original large dataset *D*, lowered minimum support *low_sup*, global minimum support *min_sup*, and number of partitions

n. Initially the dataset D is divided into n partitions. Phase I of the algorithm requires n iterations. In each iteration i, partition p_i is processed. The procedure *mine*, which we introduced in section 5.1.1, takes a partition p_i as input and generates local frequent itemsets and negative borders as the output. The local frequent itemsets and negative borders from these n partitions are used to generate the global candidate itemsets. In phase II, the algorithm merges all the local frequent itemsets and negative borders from all n partitions by accumulating their supports, which would be their support count for the entire dataset. The global frequent itemsets will be generated from each itemset if its support is not less than the user defined minimum support. The algorithm could read the entire dataset once, during phase I, if the lowered minimum support was suitably chosen. That is, in such case, we do not even read the second scan of the dataset in phase II.

7.2.1 Local Candidate Itemsets Generation

The generation of local frequent itemsets and negative borders in our partition-based mining algorithm are exactly the same as the ones introduced before in section 5.1.1. The procedure, *mine*, takes a partition and generates all frequent itemsets and negative borders along with their support counts for that partition. We also use the sub-procedures *generate* and *has_infrequent_subset*, whose purposes should be clear from the names. The difference is that we organize each local frequent itemsets and negative borders in a lexicographic order at the end of phase I. That is because when we mine each partition, we sort each transaction in the order of frequency of its items. Since the search will be faster if the order of items corresponds to a frequency order of items, which we would know after the first read of the partition. This speeds up retrieving the occurrences of the

itemsets. We build the *trie* structure with the frequency order, instead of the original order. However, the frequency order could be different from each partition. To convenience the merge process in phase II, we reorganize all the local candidate itemsets in lexicographic order after they are generated.

7.2.2 Global Frequent Itemsets Generation

The global candidate itemset is generated as the union of all local frequent itemsets and negative borders from all partitions. In phase II of our algorithm, the global frequent itemsets are determined from the global candidate itemsets by user specified minimum support. We build a trie to store and merge all the local frequent itemsets and negative borders from all partitions. If the itemsets found from different partitions are the same, we just accumulate the support counts of each itemset. When we can not find an itemset in the trie, we need to add this itemset to the Trie. Since the partitions are non-overlapping, an accumulative count over all partitions gives the support for an itemset in the entire dataset. We then identify all the global frequent itemsets by comparing their support count with the minimum support threshold, *min_sup*, specified by the user.

The Apriori property is used to reduce the search space and hence improve the efficiency of generation of frequent itemsets in phase II. This property may be stated as follows. If an itemset I is not frequent, then none of its superset can be more frequent than I. According to this property, when we traverse the trie structure to identify those global frequent itemsets whose support counts are at least equal to the user defined minimum support, we do not need to search the trie below a node that corresponds such an itemset whose support is less than the minimum support.

7.2.3 Data Structures and Implementation

In this section, we describe the data structures and the implementation of our partition-based mining algorithm.

The main data structure we use in our algorithm is a trie structure. We already described this data structure and details of information it holds in section 5.2. To efficiently generate the candidate itemsets by joining the frequent itemsets, we store the itemsets in each partition in the ascending order of the frequencies of the items in that partition. Recall that at the end of phase I, we sort all the local candidate itemsets in lexicographic order.

The trie structure is also used in phase II of our partition-based mining algorithm. We use STL (Standard Template Library), such as <vector>, <set>, in our implementation. In addition to using Class *Item* to implement the trie data structure, we add a new Class, *Itemset*, to store each candidate itemset. All candidate itemsets are first read in a vector, then transformed to a Class Itemset object with its support count as the last element in this vector. Each of the Class Itemset object will be stored in the tree as an object of Class Item. After the trie is being built, we traverse it starting from the root. According to the Apriori property, if the support count of an itemset node in the tree is less than the minimum support threshold, we do not need to traverse its children. In this case, the search continues from the next sibling of such node. After the entire tree is traversed, all the global frequent itemsets are generated.

7.3 Performance Evaluation

In this section, we report the experiments and the performance results of our proposed algorithm. We also compare the performance with the Apriori algorithm and a previous mining algorithm based on partitioning [14].

The experiments were run on a Pentium® 4 1.7GHZ, 256M of RAM, 3.4G disk in the Linux environment. In our tests, we use the synthetic dataset, T10I4D100K.dat, from FIMI [23]. This dataset has 10^5 transactions and is the same as the dataset used in [14].

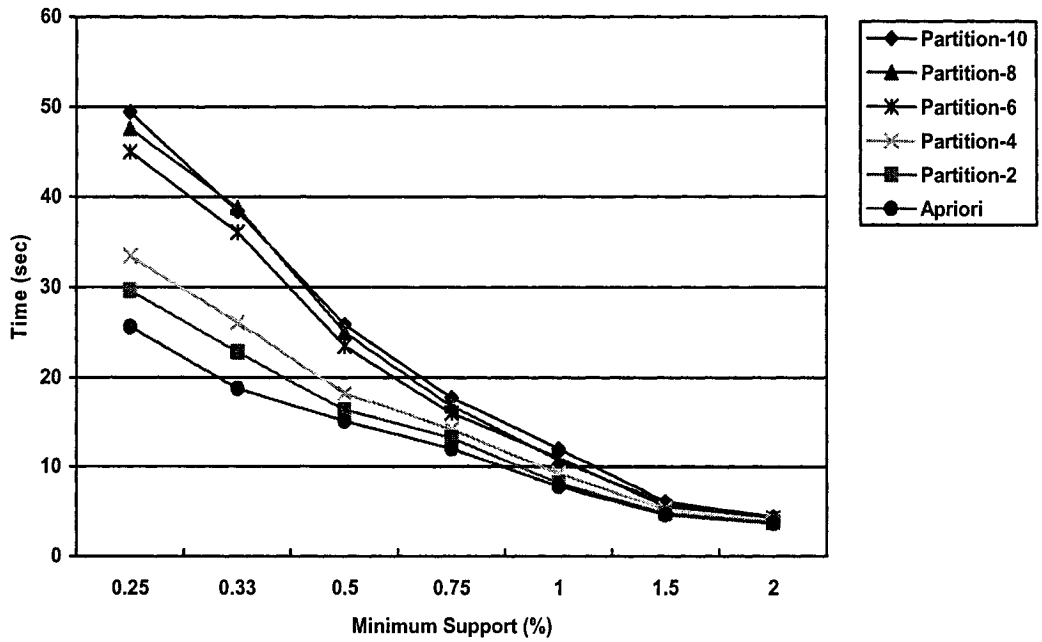


Figure 19: Performance comparison on dataset T10I4D100K.dat

Figure 19 shows the execution times for this synthetic dataset for increasing values of minimum support, which includes 0.25%, 0.33%, 0.5%, 0.75%, 1%, 1.5% and 2%. For comparison, we also carried out the experiments with different number of partitions,

which includes 2, 4, 6, 8, and 10. Since this dataset is not large enough and fits in the main memory, we compare the execution time to the one with the Apriori algorithm. We have not implemented buffer and disk management, so we did not include disk I/O times in the execution cost.

We used the command "*user/bin/time –a –f*" in Linux system to record the execution time of a process into a file. We also make a batch file to instruct the system to execute our mining algorithm with the above different minimum supports. The execution time in the figure is the *user time* in our record file. This is also the standard timing method used in FIMI [23].

The execution time decreases for both Apriori and our partition-based mining algorithms as the minimum support increases because the total number of frequent and candidate itemsets decrease. For this dataset, the fewer number of partitions the better our algorithm performance, in all cases. This was expected because more partitions means more sets of local candidate itemsets, which in turn requires more time for merging the results in phase II of our proposed method. The size of partitions is mainly chosen based on the size of the available main memory. For a given dataset, the number of partitions must be as few as possible, where each partition is as large as possible. This will give the best performance of our partitioning algorithm.

The improvement in the execution time for partition shown in Figure 19 is mainly due to the reduction in the CPU overhead and not due to the reduction in the disk I/O's. The reason is that the chosen dataset is not large enough to significantly affect the total execution time.

We also generated a large synthetic dataset, T10I4D1M.dat, with 10^6 transactions and carried out the experiments using the same computer platform mentioned above. Figure 20 illustrates the experimental results. Compared with the results in Figure 19, we note that when the dataset is increased 10 times larger, the execution time is almost 10 times it took by Apriori algorithm on the first dataset (Figure 19), for the minimum support 0.25%. Also using our technique, it took about 450 seconds to process the second dataset, while it took about 50 seconds to process the first dataset. That is, while the dataset was increased 10 times larger, the time increased was only 8 times more. For the minimum support 2%, our technique took almost the same time as Apriori algorithm, which is significant upon noting that the data was really accessed from the disk and not the main memory. From these experiments and the observations above, we conclude that the proposed technique is efficient and scalable when the dataset is too large to fit into the main memory.

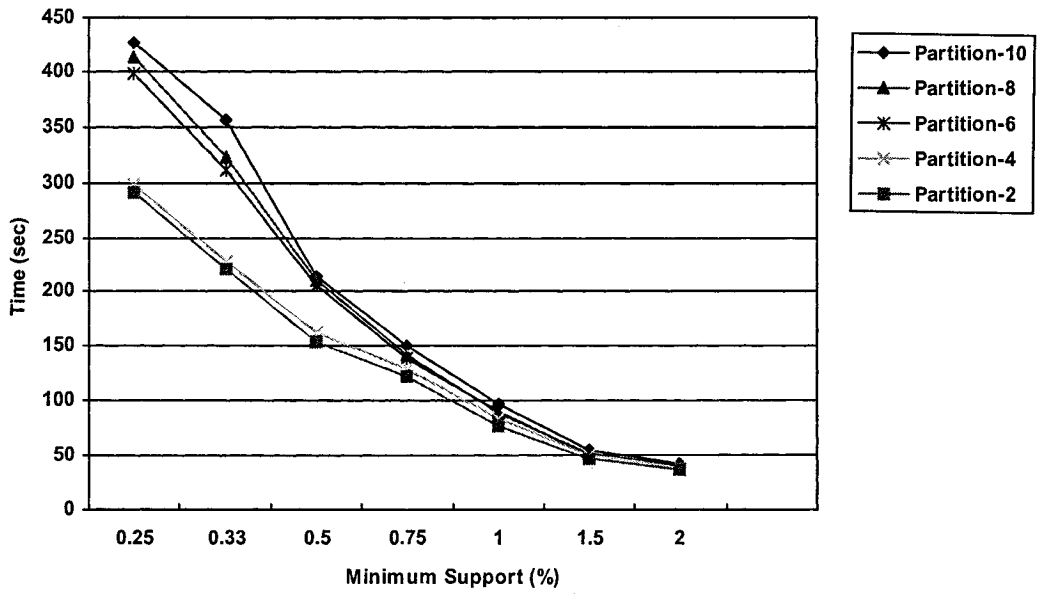


Figure 20: Performance comparison on dataset T10I41M.dat

Chapter 8

Conclusions and Future Work

In this work, we proposed an efficient algorithm for maintenance of association rules discovered in transaction databases. For this, we first studied the data mining problem, its classification, and the various solutions proposed. In this study, we were in particular interested in the problem of association rule mining, which has attracted lots of attention throughout the last decade. We then reviewed and provided formal definitions of frequent itemset, negative border, association rule, and together with a formal problem statement for our research. A number of related works are studied and discussed, including Apriori algorithm [1, 3, 4. 5], FP-growth algorithm [6, 7], incremental updating [13], sampling for association rule [12], and partitioning method [14]. We presented our system prototype and architecture, the five major functional modules of system prototype, which consists of Dataset Generator, Dataset Merger, Dataset Splitter, Miner, and Maintainer. We gave pseudocode of our various algorithms, described the data structures we use, and also provided some technical implementation details and features.

We showed that our maintenance algorithm reduces the time for updating the set of frequent itemsets. This is achieved by book keeping the frequent itemsets and the negative borders along with their support counts done "the lowered minimum support" threshold. Mining frequent itemsets along with negative borders will not require much time as shown in our experiment. The experimental results show that, as expected, our algorithm performs better than mining the whole datasets from scratch.

We developed a partition-based mining algorithm to compute frequent itemsets in very large datasets, which do not fit into main memory entirely. This is done by first dividing the dataset into several non-overlapping partitions. For each partition, we compute and store all the local frequent itemsets and the negative borders along with their support counts by lowered minimum support threshold. Then in the second phase, we "merge" the information obtained from all the partitions to determine the global frequent itemsets of the entire dataset. An important feature of our technique is that very often it requires only one full scan of the whole database when the lowered minimum support is suitably chosen.

Our maintenance and updating technique can be adapted to any level-wise algorithm for association rules mining. How a depth first search algorithm, like FP-growth, can efficiently compute the frequent itemsets along with negative borders is an interesting issue, which needs to be investigated in our context. Our work also raises three open issues. The first one concerns the lowered minimum support: "How lower the minimum support we could choose without missing any frequent itemsets and negative borders in the "merge" phase?" Another issue is, "Is there anyway to measure and control the probabilities of re-scanning the original database for adding new frequent itemsets". The third issue is, "How to suitably choose the number of partitions automatically?"

We did not discuss how to estimate the number of partitions for optimal computation, but believe arguments similar to external merge-sort technique applies in our context. Also, due to its nature of being a divide-and-conquer method, the proposed technique is also flexible to take advantage of parallel, shared nothing computer systems or in the network of computers.

Bibliography

[1] Jiawei Han, Micheline Kamber. Data Mining Concepts and Techniques. Morgan Kaufmann Publishers. 2000.

[2] Michael J. A. Berry and Gordon S. Linoff. Data Mining Techniques, Second Edition. Wiley Publishing, Inc. 2004.

[3] R. Agrawal, T. Imielinski, and A. Swami. Mining Association Rules between Sets of Items in Large Databases. In Proceedings of the ACM SIGMOD International Conference, Pages 207-216, May 1993.

[4] R. Agrawal and R. Srikant. Fast Algorithms for Mining Association Rules. In Proceeding of the 20th VLDB, Santiago, Chile, Pages 487-499, September 1994.

[5] R. Agrawal, H. Mannila, R.Srikant, H. Toivonen, and A.I. Verkamo. Fast discovery of association rules. In U.M. Fayyad, G. Piatetsky Shapiro, P. Smyth, and R. Uthnrusamy, editors, Advances in Knowledge Discovery and Data Mining. MIT Press, Pages 307-328, 1996.

[6] J. Han, J. Pei, and Y. Yin. Mining Frequent Patterns without Candidate Generation. In SIGMOD, Pages 1-12, 2000.

[7] R. Agrawal, C. Aggarwal, and V. Prasad. Depth first generation of long patterns. In SIGKDD, Pages 108-118, 2000.

[8] M.J. Zaki. Scalable algorithm for association mining. IEEE Transactions on Knowledge and Data Engineering, Pages 372-390, May/June 2000.

[9] M.J. Zaki, S. Parthasarathy, M. Ogihara, and W. Li. New algorithms for fast discovery of association rules. In Proceedings of the Third International

Conference on Knowledge Discovery and Data Mining. AAAI Press, Pages 256-265, 1997.

[10] S. Brin, R. Motwani, J. D. Ullman, and S. Tsur. Dynamic itemset counting and implication rules for market basket data. In SIGMOD, Pages 255-264, 1997.

[11] H. Mannila and H. Toivonen, and A.I. Verkamo. Levelwise search and borders of theories in knowledge discovery. Data Mining and Knowledge Discovery, Pages 241-258, November 1997.

[12] Hannu Toivonen. Sampling Large Databases for Association Rules. In Proceedings of the 22nd VLDB, Mumbai (Bombay), India, Pages 134-145, September 1996.

[13] D. W. Cheung, J. Han, V. T. Ng, and C. Y. Wong. Maintenance of Discovered Association Rules in Large Databases: An Incremental Updating Technique. In Proceedings of the 12th ICDE, New Orleans, Louisiana, Pages 106-114, February 1996.

[14] A. Savasere, E. Omiecinski, and S. Navathe. An Efficient Algorithm for Mining Association Rules in Large Databases. In Proceedings of the 21st VLDB Conference, Zurich, Swizerland, Pages 432-444, 1995.

[15] B. Goethals. Survey on frequent pattern mining. Technical report, Helsinki Institute for Information Technology, 2003.

[16] J. Hipp, U. Guntzer, and G. Nakhaeizadeh. Algorithms for Association Rule Mining – A General Survey and Comparison. SIGKDD Explorations. ACM SIGKDD, Pages 58-64, July 2000.

[17] F. Bodon and L. Ronyai. Trie: an alternative data structure for data mining algorithms. To appear in Computers and Mathematics with Applications, Pages 25-33, 2003.

[18] A. Amir, R. Feldman, and R. Kashi. A new and versatile method for association generation. Information Systems, Pages 333-347, 1997.

[19] R. J. Bayardo, Jr. Efficiently mining long patterns from databases. In Proceedings of the 1998 ACM SIGMOD International Conference on Management of Data. ACM Press, Pages 85-93, 1998.

[20] C. Borgelt and R. Kruse. Induction of association rules: Apriori implementation. In Proceedings of the 15th Conference on Computational Statistics. 2002

[21] D. E. Knuth. The Art of Computer Programming Vol. 3. Addisioin-Wesley, 1968.

[22] S. Thomas, S. Bodagala, K. Alsabti, and S. Ranka. An efficient algorithm for the incremental updation of association rules in large database. In Proceedings of the 3rd International conference on Knowledge Discovery and Data Mining (KDD 97), New Port Beach, California. Pages 263-266, August 1997.

[23] Workshop on Frequent Itemset Mining Implementations (FIMI'04). http://fimi.cs.helsinki.fi/fimi04/. 2004.

[24] N. Shiri, Y. Song, V. Alagar. Efficient and scalable techniques for mining and maintenance of association rules in large databases. Submitted for publication, April 2005.