A Constraint Based Interactive Frequent Pattern Mining Algorithm for Large Databases

by

Tariqul Hoque

A thesis submitted to the Faculty of Graduate Studies of
The University of Manitoba
in partial fulfillment of the requirements for the degree of

Master of Science

Department of Computer Science
The University of Manitoba
Winnipeg, Manitoba, Canada

March 2007

THE UNIVERSITY OF MANITOBA

FACULTY OF GRADUATE STUDIES

*****

COPYRIGHT PERMISSION

A Constraint Based Interactive Frequent Pattern Mining Algorithm for Large Databases

BY

Tariqul Hoque

A Thesis/Practicum submitted to the Faculty of Graduate Studies of The University of Manitoba in partial fulfillment of the requirement of the degree

Master of Science

Thesis advisor

Dr. Carson K. Leung

Author

Tariqul Hoque

A Constraint Based Interactive Frequent Pattern Mining Algorithm for Large Databases

Abstract

Over the past decade, many frequent-pattern mining algorithms have been developed. However, many of them rely on the availability of large memory. Their performance degrades if the available memory is limited because of the overhead and extra I/O costs. Moreover, among the algorithms that mine large databases, many of them do not provide users control over the mining process through the use of constraints. Constraint based mining is very important because it encourages users focus on only those patterns that are interesting to the users. Furthermore, among the algorithms that handle user constraints, many of them do not allow users to interactively change the mining parameters during the mining process. As mining is usually an iterative process, it is important to have an algorithm that supports *constraint based mining* and allows users to *interactively* mine *large databases.*

In this thesis, we design and implement a constraint based interactive mining algorithm, named *Inverted Matrix++*, that uses a disk based data structure called *inverted matrix* for mining frequent patterns from large databases and constructs a conditional tree called *COFI*-tree* for each frequent item from the inverted matrix. Our algorithm facilitates constraint based mining and interactive mining from large databases. Experimental results show the efficiency of our algorithm in constrained interactive mining from large databases.

Acknowledgements

First of all, I express my profound gratitude to Dr. Carson K. Leung, my research supervisor. This thesis could not have been written and completed without Dr. Leung, who not only served as my research supervisor but also encouraged and guided me throughout my academic program.

I also thank my fellow research members for their valuable investigative questions and suggestions regarding my thesis work. Special thanks to my thesis examination committee members (Dr. Jeffrey E. Diamond and Dr. Pourang P. Irani) and the chair of my thesis defence (Dr. Peter C. J. Graham). I also thank members of the Graduate Studies Committee in Department of Computer Science for their useful comments and suggestions on my thesis proposal.

I deeply appreciate the financial support from my research supervisor (Dr. Leung), TRLabs Winnipeg and my parents. Without their financial support, I would not be able to start and continue my M.Sc. study at the University of Manitoba.

Finally, thanks to my family members for their constant support and perseverance to accomplish this endeavor. I also wish to thank my wife (Jobaida Begum) for unwavering support and pushing me to achieve this milestone in my life and career.

TARIQUL HOQUE
B.S., North South University, Bangladesh, 2001

The University of Manitoba
March 2007

To my parents who always dream about my better future.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Nowadays, most organizations have their own databases. These organizations are usually interested in manipulating these large databases to retrieve valuable information so that they can make decisions based on the retrieved information. An objective of data mining is to discover important or potentially useful but previously unknown patterns from a large amount of data, which help make predictions [Dun03, HK06, TSK06].

In general, transaction databases consist of a collection of transactions, where each transaction represents a set of items (aka an *itemset*) purchased by a customer. Association rules [AIS93, KMR+94] show how the presence of one item implies the presence of other items in the same transaction. In other words, association rules reveal relationships among items. A popular example used in the area of data mining is that researchers found an interesting association rule "customers who buy beer also buy diapers".

The task of mining association rules is to generate all association rules from a

transaction database *TDB*. Let $I = \{i_1, i_2, ..., i_m\}$ be a set of items. T is a transaction where $T \subseteq I$. T contains itemsets X and Y, where $X \subset I$, $Y \subset I$ and $X \subseteq T$, $Y \subseteq T$ and $X \cap Y = \emptyset$. Then, the rule "$X \Rightarrow Y$" is returned to the user if it satisfies two conditions: (i) $X \cup Y$ satisfies the minimum support threshold (minsup), and (ii) "$X \Rightarrow Y$" satisfies the minimum confidence threshold (minconf). Here,

$$\text{support} = \frac{\textit{The number of transactions containing all items in } X \textit{ and } Y}{\textit{The number of transactions in TDB}} \text{ and}$$

$$\text{confidence} = \frac{\textit{The number of transactions containing all items in } X \textit{ and } Y}{\textit{The number of transactions containing all items in } X}.$$

In other words, the support is the probability of having $X \cup Y$ in TDB, and the confidence is the conditional probability of finding Y having found X [AS94].

Association rule mining [AIS93, AS94] can be divided into two basic steps as follows. The first step is to find all frequent itemsets that satisfy a minimum support threshold, and the second step is to generate all interesting association rules. To get a better understanding of these two steps, let us consider a brief illustrative example below.

Example 1.1 *Consider the transaction database shown in Table 1.1. There are four transactions with five items A, B, C, D, and E. We count the support of all candidate itemsets (Table 1.2) by scanning all 4 transactions in the database. If the minimum support threshold is 50%, then we find all the frequent itemsets listed in Table 1.3. The support of* $\{A\}$ *is 75% (i.e., 3/4) since item A occurs in 3 out of 4 transactions. We check for all itemsets shown in Table 1.2, and find that only itemsets* $\{A\}$, $\{D\}$, $\{E\}$, *and* $\{A, E\}$ *are frequent (i.e., they satisfy the minimum support threshold).*

Table 1.1: Transaction database

Transaction-ID	**Itemsets**
T1	{A,B,E}
T2	{A,E}
T3	{A,D}
T4	{C,D}

Table 1.2: Sets of candidate itemsets

Cardinality	**#Itemsets**	**Candidate Itemsets**
1	5	{A},{B},{C},{D},{E}
2	10	{A,B},{A,C},{A,D},{A,E},{B,C}, {B,D},{B,E},{C,D},{C,E},{D,E}
3	10	{A,B,C},{A,B,D},{A,B,E},{A,C,D},{A,C,E}, {A,D,E},{B,C,D},{B,C,E},{B,D,E},{C,D,E}
4	5	{A,B,C,D},{A,B,D,E},{A,B,C,E}, {A,C,D,E},{B,C,D,E}
5	1	{A,B,C,D,E}

Table 1.3: List of frequent itemsets (with minsup=50%)

Frequent Itemsets	**Support**
{A}	75%
{D}	50%
{E}	50%
{A,E}	50%

Recall that association rule mining can be divided into two basic steps. In the first step (which is known as **frequent pattern mining**), for n items in the domain, we find frequent itemsets from $O(2^n)$ candidate itemsets by computing the frequency of these candidate itemsets. For instance, in Example 1.1, we generate a total of 31 candidate itemsets (as shown in Table 1.2) for five items (namely, A, B, C, D, and E) in the domain. In the second step of association rule mining (which is known as **rule generation**), for p frequent itemsets found in the first step, we generate $O(p^2)$

possible antecedent-consequence pairs for the rules "*Antecedent* $\Rightarrow$ *Consequence*". Among these rules, we return those satisfying the user-defined confidence threshold. For instance, in Example 1.1, for four frequent itemsets (namely, {A}, {D}, {E}, {A,E} shown in Table 1.3), we can generate rules such as "$\{A\} \Rightarrow \{E\}$" and "$\{E\} \Rightarrow \{A\}$". Hence, among the two steps in association rule mining, the first step is *computationally more expensive* than the second step. Frequent pattern mining is used not only for association rule mining but also for other data mining tasks (e.g., for mining correlations [BMS97], sequences [AS95], maximal and closed patterns [GZ04], and causality [SBMU98]. Therefore, frequent pattern mining is an important research task in data mining, and my M.Sc. research lies in frequent pattern mining.

Over the past decades, data miners have focused on the first step so as to find techniques to reduce the computation of this step. In the early years, most studies (e.g., algorithms like DHP [PCY95], Partition [SON95], Sampling [ZPLO96], and DIC [BMUT97]) used the Apriori framework. To speed up the mining process, the FP-growth algorithm [HPY00] was proposed, which is based on a novel tree structure called *Frequent Pattern tree* (*FPtree*) [HPY00] that captures the content of the database.

In general, the above mentioned algorithms find frequent patterns satisfying the minimum support threshold. The question is: What is the appropriate value for minimum support threshold? It is not an easy question to answer because appropriateness depends on the expectation of the user. Sometimes, the user may not even know what value is appropriate. Consequently, setting the threshold too high results in too few itemsets; setting it too low results in too many itemsets. Therefore, the user usually

needs to execute the mining algorithms for a number of times. Finding frequent patterns from scratch multiple times (each with a different threshold) can be expensive. To solve this problem, researchers introduce **interactive mining** [Hid99, Leu04], where the user can modify the mining parameter (e.g., the minimum support threshold) during the mining process based on the feedback provided by the system. Carma [Hid99] and iCFP [Leu04] are examples of interactive mining algorithms.

Even with an appropriate value for the minimum support threshold, the system may return a number of frequent patterns, out of which only a tiny fraction is interesting to the user. This calls for **constraint based mining** [NLHP98, LLN03]. In constraint based mining, users are allowed to focus the mining, where user can specify the pattern to be mined according to their intention using some restrictions. These restrictions are formally called *constraints.* For example, if the user is interested in finding itemsets of type "snack" and the price of each item in the itemsets is less than \$10, then he has to specify some constraints in addition to the support and confidence thresholds. Two classes of widely used constraints are *succinct constraints* and *anti-monotone constraints.* Some algorithms have been developed to facilitate the rules. CAP [NLHP98], FIC [PHL01], FPS [LLN02], and iCFP [Leu04] are examples of these constraint based mining algorithms.

Many of the above mentioned algorithms assume that there exists *sufficient* main memory space. Such an assumption is not too unrealistic *in many cases* due to the current trend that modern computing moves towards computers with large amounts of main memory (say, with gigabytes of memory) [EZ03b]. However, there are *situations* where the available memory is limited. One needs to bear in mind that, although

the amount of available memory keeps increasing, the volume of available data to be gathered and stored in the memory (e.g., data from the Internet, satellites, and other sources) can even grow faster. The volume is doubling every 18 months. This massive growth of database size may surpass the growth of hardware technology. When facing these situations (where the available memory is limited, i.e., there does not exist sufficient memory space), many of the above mentioned algorithms may encounter problems. For example, El-Hajj and Zaïane [EZ03a] reported that, when using a 733 MHz machine with a RAM of 256 MB, (a) the Apriori algorithm [AS94] was unable to mine databases with 5 million transactions (which required about 550 MB of space) and (b) the FP-growth algorithm [HPY00] was unable to mine databases with more than 5 million transactions when using a minimum support threshold of 0.01% and 100,000 items in the domain. However, in many data mining applications, it is not unusual to analyze large databases with more than 5 million transactions.

When facing these large databases, some researchers proposed the use of auxiliary data structures such as the *inverted matrix* [EZ03a] and the *co-occurrence frequent-item tree* (*COFI-tree*) [EZ03a, EZ03b, EZ04]. These auxiliary structures are designed to deal with limited memory, and require less in-memory space than the transaction database or the FP-trees (which capture the content of the database). *However, the algorithm that uses these auxiliary structures does not support interactive mining or constraint based mining.* To support mining large databases, we have identified the following research questions: Can one use the inverted matrix and the COFI-tree (i.e., the auxiliary structures designed for mining from large databases) to support constraint based mining so that users could specify their interest via the use of con-

straints? Can one use these auxiliary structures to support interactive mining in such a way that users could change the mining parameter (specifically, the support threshold) during the mining process?

1.1 Contributions

This thesis work is motivated by the above mentioned questions. We answer all of these questions in this thesis. We propose a memory-efficient mining technique to support constraint-based mining and interactive mining. Our **thesis statement** is:

> We develop an algorithm for *constraint based interactive* frequent pattern mining that can mine *large databases.*

Specifically, we focus our attention on the following aspects:

- **Constraint based mining** (e.g., how to find itemsets that satisfy the user specified constraints?),
- **Interactive mining** (e.g., how to find itemsets when users can change the minimum support threshold during the mining process?), and
- **Mining large databases** (e.g., how to use the inverted matrix and the COFI-tree to find itemsets from large databases, especially when memory is limited?).

To handle large databases, we propose a tree structure called **COFI*-tree**, which is similar but not identical to the COFI-tree. For instance, the COFI*-tree does not require to store extra information (e.g., participation value used in COFI-tree) for mining the tree.

In addition, we also propose an algorithm, called **Inverted Matrix++**, which uses both the inverted matrix and the COFI*-tree. It is designed to support constraint based mining with interactive mining from large databases.

1.2 Thesis Organization

The rest of this thesis is organized as follows.

In Chapter 2, some background materials and related work are presented. We describe the concepts, properties and classes of constraint mining. We also discuss some key frameworks of frequent pattern mining (Apriori and FP-tree), and describe some existing algorithms for constraint based mining, interactive mining and large database mining. We also elaborately describe two data structures: the inverted matrix (that we adopt for our system to support mining large databases) and the COFI-tree.

In Chapter 3, the methodology of our research work is described. We propose a projected FP-tree based data structure, called COFI*-tree. We also describe methods of pushing the constraints inside the mining process according to their properties and constraint classes. We also highlight the interactive mining feature of our proposed Inverted Matrix++ algorithm.

Experimental results on both synthetic and real-life datasets are presented in Chapter 4. The results show the efficiency of our proposed Inverted Matrix++ algorithm.

Finally, we conclude our thesis and propose some ideas for future work in Chapter 5.

Chapter 2

Related Work

This chapter describes some related work and background information on frequent pattern mining that helps readers understand our thesis problem. Section 2.1 discusses two major frameworks: the Apriori framework [AS94] and the FP-tree framework [HPY00], which are widely used and adopted as the basis of other algorithms for mining frequent patterns. While the Apriori-based algorithms (e.g., the Partition algorithm) and the FP-tree based algorithms (e.g., the FP-growth algorithm) are efficient in mining frequent itemsets from many databases, their performance degrades when the database is too large to fit into the available memory. Hence, we discuss in Section 2.2 how some previous algorithms (e.g., the Inverted Matrix algorithm [EZ03a] that uses the inverted matrix (Section 2.2.1) and the COFI-tree (Section 2.2.2)) handle the situation where the database is large. While these algorithms (i.e., Inverted Matrix) are efficient in mining large databases, they have not yet supported interactive mining or constraint based mining. In this chapter, we also present some relevant interactive and constraint based mining algorithms including a

brief introduction on "constraints".

2.1 The Mining Framework

In the literature, the two major frameworks that are used widely for developing frequent pattern mining algorithms are: the Apriori framework [AS94] and the FP-tree framework [HPY00]. The Apriori was developed in 1994, whereas a tree based mining method FP-tree in 2000 to speed up the mining process. Since most of the frequent pattern mining algorithms were developed based on either the Apriori framework or the FP-tree framework, we like to briefly discuss these two frameworks in the next two sections (2.1.1 and 2.1.2).

2.1.1 The Apriori Based Framework

The Apriori [AS94] is known as the classic algorithm in the family of frequent pattern mining algorithms. A very nice property of the Apriori algorithm is that "if an itemset is frequent, then all its subsets are also frequent". Suppose that the support of itemset {A, B} is 3. If itemset {A, B} is frequent, then both {A} and {B} are also frequent because the support of each of the itemsets is at least 3. The Apriori algorithm uses a generate-and-test approach. To elaborate, the algorithm first determines all candidate itemsets, and then tests these candidate itemsets against the database to find out whether they are frequent or not. With the Apriori property, the algorithm does not have to generate all possible combinations of items in the database. It prunes a significant number of candidate itemsets that are below the minimum support threshold. In the first pass of the algorithm, it scans the database

and computes all frequent 1-itemsets (i.e., itemsets of size 1). The algorithm does the following for passes $k > 1$:

> "It generates candidate k-itemsets using the frequent itemsets found in the previous pass. Then, it scans the database and counts the local supports of those candidate itemsets. After that, it outputs frequent k-itemsets. Then, it starts the next pass" [AS94].

This process is iterative, and it terminates when no more frequent itemsets can be generated. This is a candidate generate-and-test approach.

Note that the Apriori algorithm can be considered as a foundation to many algorithms including the Dynamic Hashing and Pruning (DHP) [PCY95], the Partition algorithm [SON95], the Sampling algorithm [ZPLO96], the Dynamic Itemset Counting algorithm (DIC) [BMUT97], and the Segment Support Map (SSM) [LLN00, LNM02] based algorithm. However, the candidate generate-and-test process is a bottleneck of these algorithms. Let us consider a database with 100,000 different items and 10 million transactions. In the worst case, the Apriori algorithm may need to generate $O(2^{100,000})$ candidate itemsets. This algorithm may not be able to mine this large database because of the following reasons: It takes a very long time to complete the task, and it needs huge memory to generate a huge number of candidate itemsets. The Apriori algorithm also suffers from the problems of the repeated I/O scans for scanning the database and the high computational cost for the candidate generate-and-test approach.

The DHP algorithm [PCY95] uses a hash table to shrink the number of candidates by pre-computing the approximate support. The algorithm also trims the transac-

tions that do not contain any frequent items. Though this Apriori based algorithm reduces some candidates, it still generates a huge number of candidates. The Sampling algorithm [ZPLO96] takes a small sample of the database, and it determines all the frequent itemsets based on the sample. The accuracy of the result heavily depends on the quality of sample. The SSM [LLN00, LNM02] divides the database into a number of non-overlapping segments to count and store the support of each singleton itemset in each segment. It speeds up the performance of the Apriori algorithm by reducing the number of candidates. The problem with this method is that it can only deal with singleton itemsets. All of these Apriori based algorithms try to improve performance, but they still generate candidate itemsets.

2.1.2 The FP-Tree Based Framework

To avoid generating a huge number of candidate itemsets, Han et al. [HPY00] proposed a FP-tree based algorithm called FP-growth. The algorithm is based on an FP-tree, which is a compact memory based tree structure representing frequent patterns.

The algorithm first scans the whole database to determine frequent 1-itemsets (singleton frequent itemsets, i.e., {A}, {B}, {C}, etc.). Then, it sorts the frequent items in descending frequency order. It scans the database once again, and constructs the FP-tree. The FP-tree consists of all frequent items. Therefore, there is no need to generate any candidates since it represents frequent patterns.

The FP-tree framework is advantageous over the Apriori in a sense that it avoids generating huge number of candidate itemsets. The FP-tree based algorithms are

significantly faster than the Apriori based algorithms. However, FP-tree based algorithms assume that tree structures (an FP-tree representing the transaction database TDB and conditional trees for subsets of TDB) fit into the main memory [HPY00]. This assumption may not hold when the algorithms deal with large databases.

2.2 Mining Large Databases

In this section, we discuss some of the existing methods that take care of the memory limitation problem. These methods support mining large databases. Before describing the memory efficient techniques, let us have an overview of some related data structures (an inverted matrix, a COFI-tree).

2.2.1 An Inverted Matrix

An ***inverted matrix*** [EZ03a] captures the content of a transaction database. In contrast with the usual transaction database (where items in a transaction are stored in one row), m items in a transaction are stored over m rows in an inverted matrix. Each row of the inverted matrix stores IDs of the transactions in which the items occur; each entry in the inverted matrix points to the next item on the same transaction. More precisely, each pointer consists of two elements: the first element points to the address of a row of the matrix and the second element indicates the address of a column of the matrix. In Example 2.1, we show a sample database in Table 2.1 and the inverted matrix for this database in Table 2.3.

Table 2.1: Sample database TDB

TID	**Itemsets**
T1	{A, B, C, D}
T2	{B, D}
T3	{B, C, D}
T4	{A, C, D}

Table 2.2: The inverted matrix for T1

Loc	**Index**				
1	(A,2)	(2,1)			
2	(B,3)	(3,1)			
3	(C,3)	(4,1)			
4	(D,4)	$(\emptyset,\emptyset)$			

Table 2.3: The inverted matrix for TDB

Loc	**Index**				
1	(A,2)	(2,1)	(3,3)		
2	(B,3)	(3,1)	(4,2)	(3,3)	
3	(C,3)	(4,1)	(4,3)	(4,4)	
4	(D,4)	$(\emptyset,\emptyset)$	$(\emptyset,\emptyset)$	$(\emptyset,\emptyset)$	$(\emptyset,\emptyset)$

Example 2.1 *There are four items A, B, C, and D in the sample database (Table 2.1). To construct an inverted matrix for this database, the database is scanned to find the frequency of each item. Then, the database is scanned again, and items in each row are sorted in ascending frequency order. At the same time, the column "Index" of the inverted matrix is filled with each item and its frequency. The first row in Table 2.1 contains items A, B, C, and D. Item A is located in line 1 in the inverted matrix and has a link to the first empty slot of the array of item B, since B is the next item after A. Therefore, entry (2, 1) is added to the first slot of the array of item A. Thus, entries (3, 1) and (4, 1) are added to the first slots of items B and C, respectively. For item D, a null entry* $(\emptyset,\emptyset)$ *is added since there is no more items*

after D. This results in Table 2.2.

Similarly, the same method can be applied to each of the remaining rows in the database. This results in the inverted matrix (shown in Table 2.3), which captures the content of the database shown in Table 2.1.

2.2.2 A COFI-Tree

A ***co-occurrence frequent-item tree (COFI-tree)*** [EZ03a, EZ03b, EZ04] is a compact memory based data structure. A branch in the COFI-tree can represent one or more transactions in a database; a node of the tree represents an item. In this tree structure, each parent node may have more than one child nodes; however, a child node has exactly one parent node. At each node, we store both its *frequency* and its *participation value*. Since the participation value is irrelevant to the remainder of this thesis, we do not describe it further.

A COFI-tree represents sub-transactions for a particular item. Suppose that we have the database in Table 2.1, and we want to construct a COFI-tree for item B. Then, sub-transactions for B are shown in Table 2.4. In other words, the sub-transactions for B include all the transactions containing B (i.e., T1, T2, and T3). Each sub-transaction for B only contains B and items that rank behind B (i.e., C and D).

Table 2.4: Sub transactions

{B, C, D}
{B, D}
{B, C, D}

The first row in the sub-transaction (T1) for B contains items B, C, and D. Hence, we get a tree branch with three nodes: B, C, and D with their frequency all set to 1. Then, we have items B and D in the second row (T2), and this leads to a new child D for B. The frequency of B is then incremented to 2, and the frequency of the new node D is set to 1. For T3, we increment nodes B, C and D in the left branch so that their frequencies are 3, 2 and 2 respectively. The final tree is shown in Figure 2.1.

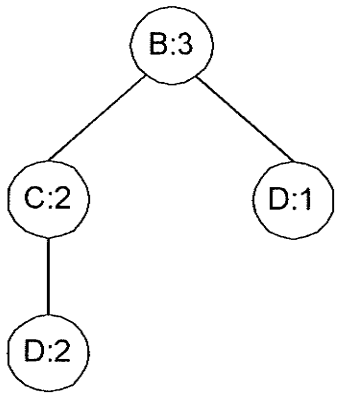

Figure 2.1: COFI-tree for item B

2.2.3 Existing Techniques to Overcome Memory Limitation

Recall that, when the transaction database (or the FP-tree capturing the content of the database) is so large that the available memory space is insufficient to hold the database or the FP-tree, many of the Apriori-based or FP-tree based algorithms may encounter problems. For example, El-Hajj and Zaïane [EZ03a] reported that, using a machine with a RAM of 256MB, (a) the Apriori algorithm [AS94] was unable to mine databases with 5 million transactions (which required about 550MB of space) and (b) the FP-growth algorithm [HPY00] was unable to mine databases with more

than 5 million transactions when using a minimum support threshold of 0.01% and 100,000 items in the domain.

To handle large databases (with insufficient memory space), several out-of-memory techniques can be applied. For example, when the entire database does not fit into the main memory, an Apriori-based algorithm called the Partition algorithm [SON95] divides the database into local partitions in such a way that each partition can fit into the main memory. The algorithm then finds all locally frequent itemsets in each local partition, and then scans the whole database once again to get globally frequent itemsets. Consequently, this Partition algorithm incurs lots of I/Os.

For tree-based algorithms, if an FP-tree is too large to fit in the main memory, then one can store the tree in the disk and to recursively partition and project the tree [HPY00]. When forming projections, the entire tree is read. As a result, lots of I/Os are incurred.

Besides the above two algorithms, one of the recently developed algorithms called **Inverted Matrix algorithm** [EZ03a] further reduces the I/O cost. This disk-based algorithm is based on the frequent conditional pattern concept. There are two phases in this algorithm. In the first phase, a disk-based data structure, called the *inverted matrix*, is generated. In the second phase, the inverted matrix is mined using a compact memory based data structure called *co-occurrence frequent-item tree (COFI-Tree)* [EZ03a, EZ03b, EZ04]. The inverted matrix captures the content of the transaction database. The algorithm first reads sub-transactions directly from the disk-resident inverted matrix. It then builds an individual memory-resident tree for each frequent item in the database. Each tree is mined independently, and is deleted

as soon as it is mined.

To gain a deeper understanding of the Inverted Matrix algorithm [EZ03a], let us consider the following example (Example 2.2).

Example 2.2 *Let us revisit Example 2.1. If the minimum support threshold is 2, then the first frequent item is A. Its first entry is (2, 1) indicating that the item represented by column 1 of row 2 (item B) is the item immediately after A in a transaction in the original database. Similarly, the entry in (2,1) is (3,1), which indicates that the item represented by column 1 of row 3 (item C) is the item immediately after {A, B}. Then, the entry in (3,1) is (4,1), which indicates that the item represented by column 1 of row 4 (item D) is the item immediately after {A, B, C}. Finally, the entry in (4,1) is (∅,∅), which indicates that no more item after {A, B, C, D}. Hence, we get the transaction {A, B, C, D} and a tree branch with nodes A, B, C, and D with frequency set to 1 (Figure 2.2(a)).*

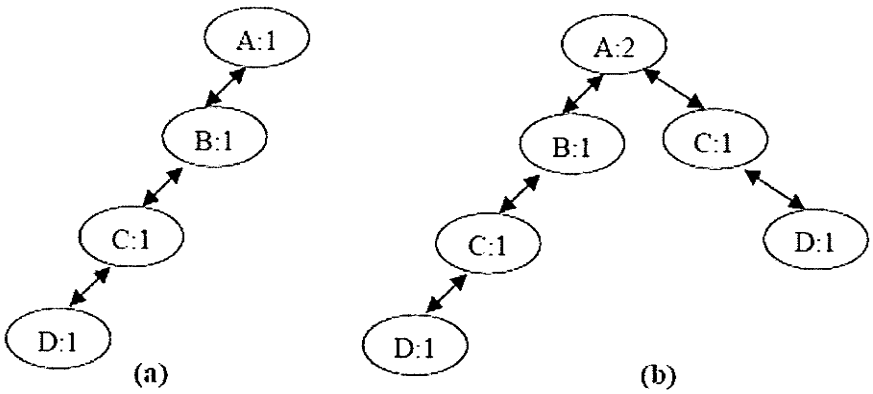

Figure 2.2: COFI-tree for item A

We then apply a similar procedure by following the link in the second entry of A, which points to (3,3), which then points to (4,4) and (∅,∅) afterwards. This link leads

to another tree branch A-C-D (Figure 2.2(b)). The frequency of A is incremented to 2 and the frequency of the new nodes C and D is set to 1. This conditional tree for A is completed since A has no more entry left.

After getting the complete tree, the algorithm finds all the possible frequent itemsets containing A: $\{A\}$, $\{A,C\}$, $\{A,D\}$, *and* $\{A,C,D\}$.

The same process is done for items B, C, and D. Thus, the algorithm finds all the frequent itemsets.

The following are some of the advantages of using the inverted matrix:

- The inverted matrix captures the content of the transaction database. Although inverted matrix does not represent the content in a compressed form (vs. FP-trees that represents the content in a compressed form), it is easier to access the content from the inverted matrix than FP-trees because one does not need to "uncompress" the information in the matrix.

- Rows are arranged in ascending frequent order of items. One can easily retrieve frequent items and skip infrequent items. In Example 2.1, if the minimum support threshold is set to 3, one can easily skip Row 1 (which represents the infrequent item A). Similarly, if the threshold is set to 4, one can easily skip Rows 1 to 3 (which represents the infrequent items A, B, and C).

- When constructing a COFI-tree for an item X, one only needs to retrieve those rows of the inverted matrix that are relevant for the construction. In other words, one does not need to scan the whole matrix. This saves I/Os.

The major advantage of using COFI-tree is that it requires less memory. Note that

both the FP-tree and the COFI-tree are stored in the memory. However, each COFI-tree, which is usually much smaller than the FP-tree (because the COFI-tree is basically the conditional FP-tree) is mined independently, and is deleted as soon as it is mined. Therefore, the Inverted Matrix algorithm [EZ03a] (which uses COFI-trees) requires less memory space than the FP-growth algorithm (which uses FP-trees).

However, while the Inverted Matrix algorithm [EZ03a] mines large databases, it does not support interactive mining or constraint based mining. We will show in the next chapter, how our propose algorithm uses the inverted matrix and a modified COFI-tree for interactive constraint based mining of frequent itemsets from large databases.

2.3 Interactive Mining

To support interactive mining, Hidber [Hid99] proposed the Carma algorithm, which provides the user with continuous feedback on frequent set computation so that the user can observe the mining progress continuously and can interactively change the minimum support during the runtime of the algorithm. However, like the Inverted Matrix algorithm, Carma also does not handle constraint based mining.

To support interactive constraint based mining, Leung [Leu04] proposed the iCFP system, which allows the user to modify the mining parameters (e.g., the minimum support threshold) during the mining process based on the feedback provided by the system. However, this system used FP-trees. In other words, it has not yet used the inverted matrix or COFI-trees (which were designed for handling large databases).

2.4 Constraint Based Mining

Even with an appropriate value for the minimum support threshold, mining algorithms may return a number of frequent patterns, out of which only a tiny fraction is interesting to the user. This calls for constraint based mining [NLHP98, LLN03].

2.4.1 Constraints

Constraints can be classified into overlapping classes depending on their properties. These classes include anti-monotone constraints, succinct constraints, and some other classes of constraints.

Definition 2.1 (Anti-monotone Constraints [NLHP98]) *A constraint C_{am} is **anti-monotone** if whenever an itemset S violates C_{am}, so does any superset of S.* □

For example, $C_{am} \equiv S.Price < 100$ is an anti-monotone constraint, because any superset of S violating C_{am} (e.g., containing any item with $Price \geq \$100$) also violates C_{am}.

Definition 2.2 (Succinct Constraints [NLHP98]) *Let Item denote the set of domain items. Succinctness is defined in several steps, as follows: Define SAT_C (Item) to be the set of itemsets that satisfy the constraint C. With respect to the lattice space consisting of all itemsets, SAT_C (Item) represents the pruned space (i.e., the solution space) consisting of those itemsets satisfying C.*

(a) An itemset $I \subseteq Item$ is a succinct set if it can be expressed as $\sigma_p(Item)$ for some selection predicates p, where σ is the selection operator (as in relational algebra).

(b) $SP \subseteq 2^{Item}$ *is a succinct powerset if there is a fixed number of succinct sets* $Item_1, ..., Item_k \subseteq Item$*, such that* SP *can be expressed in terms of the powersets of* $Item_1, ..., Item_k$ *using union and minus.*

(c) A constraint C *is* **succinct** *provided* $SAT_C(Item)$ *is a succinct powerset.* □

These overlapping classes of constraints are interesting because they have some nice properties that help optimize the mining process. For example, *a majority of constraints are succinct.* Moreover, for every succinct constraint C, there is a "formula" (called a member generating function (MGF)) that can generate precisely all those itemsets satisfying C. Hence, a succinct constraint can simply operate in a generate-only environment (by using an MGF), rather than in a generate-and-test environment. In other words, one does not need to generate lots of itemsets, test them, and then exclude those violating the constraints. Instead, one can easily enumerate (by using the MGF) all and only those itemsets that satisfy the succinct constraint. This explains why we focus on the succinct constraints in this thesis.

As an example, the constraint "$min(S.Price) \geq 10$" is succinct. Its pruned space (i.e., solution space) can be expressed as $2^{\sigma_{Price \geq 10}(Item)}$. Itemsets satisfying this succinct constraint can be generated using the MGF $\{X | X \subseteq \sigma_{Price \geq 10}(Item), X \neq \emptyset\}$, which generates all and only those itemsets comprising of items whose price is greater than or equal to 10. The succinct constraint "$min(S.Price) \geq 10$" is also anti-monotone. If the price of at least one item in the itemset S is less than 10, then the itemset does not satisfy the constraint. All supersets of this itemset also contain the invalid item having the price less than 10. Therefore, all supersets of such an invalid itemset are also invalid.

So far, we have mentioned that succinct constraints and anti-monotone constraints are interesting due to their nice properties [NLHP98]. Now, let us briefly discuss the following cases:

1. *Succinct and anti-monotone constraints*:

   For example, the constraint "$max(S.Price) \leq 20$" is succinct since an MGF $\{X | X \subseteq \sigma_{Price \leq 20}(Item),\ X \neq \emptyset\}$ can be applied to generate itemsets, where the price of each item in the itemset is less than or equal to \$20. The constraint is also anti-monotone because if the price of at least one item in the itemset is greater than \$20, then such an itemset does not satisfy the constraint and all supersets of this itemset also contain the invalid item (with price greater than \$20). Therefore, all supersets of such an invalid itemset are also invalid.

2. *Succinct but not anti-monotone constraints*:

   For example, the constraint "$max(S.Price) \geq 20$" is succinct. It means that the valid itemset must contain at least one item with price greater than or equal to \$20. The MGF is $\{X \cup Y | X \subseteq \sigma_{price \leq 20}(Item), X \neq \emptyset, Y \subseteq \sigma_{Price > 20}(Item)\}$. However, it is not anti-monotone because a superset of an invalid itemset S (say, all items in S are of price less than \$20) may be valid (e.g.., adding an item of price \$30 to S to form such a superset).

3. *Anti-monotone but not succinct constraints*:

   For example, the constraint "$sum(S.Price) \leq 100$" is anti-monotone but not succinct, because all supersets of an invalid itemset (i.e., having the total price greater than \$100) are invalid. The constraint is not succinct because there is no MGF to enumerate all and only those itemsets that satisfy the constraint.

4. *Constraints that are neither succinct nor anti-monotone*:

   For example, the constraint "$sum(S.Price) \geq 100$" is not anti-monotone and not succinct.

2.4.2 Constraint Based Algorithms

CAP [NLHP98], FIC [PHL01], FPS [LLN02], and iCFP [Leu04] are examples of constraint-based mining algorithms. The CAP algorithm [NLHP98] is an Apriori based algorithm framework. It exploits the property of constraints and pushes them inside the frequent itemset computation. For a class of constraints called the succinct and the anti-monotone constraints, CAP performs additional tests along with the frequency tests. The FIC, FPS, and iCFP are FP-tree based algorithms. All the above mentioned algorithms facilitate constraint based mining. However, they were not designed to effectively mine from very large databases. Hence, there is a need of an algorithm that can take care of the memory problem and can interactively mine constrained frequent itemsets from large databases.

2.5 Discussion

In this thesis, we develop a constraint based interactive frequent pattern mining for very large databases. Table 2.5 shows the key differences between my algorithm and some existing algorithms. The key difference is that my algorithm is able to interactively (i.e., where users can change the support threshold during the mining process) find frequent itemsets satisfying the user specified constraints from very large databases.

Table 2.5: Comparison of different algorithms

	Apriori, FP-growth	CAP, FIC, FPS	Carma	iCFP	Inverted Matrix	Inverted Matrix++
Mining large databases	×	×	×	×	√	√
Constraint based mining	×	√	×	√	×	√
Interactive mining	×	×	√	√	×	√

2.6 Summary

The Apriori framework [AS94] and the FP-tree framework [HPY00] are the two widely used frameworks for mining frequent patterns. The Apriori based algorithm suffers from the problems of the repeated I/O scans of the database and the high computational cost for the candidate generate-and-test approach. On the other hand, the FP-tree based algorithm avoids generating candidate itemsets. Hence, they are generally faster than Apriori based algorithms.

However, FP-tree based algorithms assume that the tree structure (an FP-tree representing the transaction database (TDB) and conditional trees for subsets of TDB) fits into the main memory [HPY00]. This assumption may not hold when the algorithms deal with large databases. Therefore, both the Apriori based algorithms and the FP-tree based algorithms may not be too efficient when mining large databases.

To take care of the memory limitation problem and to handle large databases, an inverted matrix (a disk based data structure) and a COFI-tree (a memory based data structure) were proposed. The COFI-tree has the same nice properties as the conditional FP-tree. However, we can build a COFI-tree from the inverted matrix

instead of from the original databases. In other words, the original databases are not needed. While the Inverted Matrix algorithm, which used inverted matrix, deals with mining from large databases, it does not support interactive mining or constraint-based mining.

On the other hand, some algorithms (e.g., Carma) support interactive mining by providing the user with continuous feedback on frequent itemset computation so that the user can monitor the mining progress continuously and can interactively change the minimum support during the runtime of the algorithm. However, they do not handle constraints. For constraint based mining algorithms (e.g., CAP, FIC, FPS), users are allowed to focus the mining by specifying the pattern to be mined using some restrictions called constraints. However, many of them do not support interactive mining. Fortunately, there are algorithms (e.g., iCFP) that support both interactive mining and constraint based mining. However, they have not yet designed to handle large databases. Hence, there is a demand for a frequent pattern mining algorithm that can efficiently mine large databases and also support interactive mining as well as constraint based mining. In this research, we design and develop a mining system for this purpose.

Chapter 3

Constraint Based Interactive Mining System

In the previous chapter, we reviewed some existing work; in this chapter, we start describing our new work. To elaborate, we describe our proposed Inverted Matrix++ algorithm. The algorithm uses two structures: an existing disk based data structure called an inverted matrix and our newly proposed memory based tree structure called a COFI*-tree. The resulting mining algorithm can handle constraints (constraint based mining) and changes of the minimum support threshold during the mining process (interactive mining), and it can find frequent itemsets from large databases.

3.1 Our Proposed Tree Structure: A COFI*-Tree

Recall from Section 2.1.2 that FP-tree [HPY00] is constructed from the database to keep all the frequent items. While mining from the FP-tree, the FP-tree algorithms

build a projected tree (also known as conditional tree) for all the frequent items in the FP-tree. However, by using an inverted matrix [EZ03a], one can directly build the projected tree for all the items found in the matrix. Each conditional tree built from the inverted matrix represents sub-transactions for a particular item. Therefore, branches in the tree contain co-occurrences of the frequent items. Our mining algorithm constructs memory efficient tree based data structure, called *COFI*-tree*, for all the *valid* (i.e., satisfying both the *minsup* and the *constraints*) frequent items. Note that the key difference between COFI-trees and COFI*-trees are as follows. COFI-trees keep all frequent items, and each node contains both frequency and participating value. As mentioned in Chapter 2 that the participation value is irrelevant to the remainder of this thesis, we do not describe it further. In other words, COFI*-trees keep only frequent *valid* items, and each node contains only its frequency (but not the participation value).

There are three major parts in our mining procedure:

1. Handling constraints (determining valid items to build COFI*-trees).

2. Constructing COFI*-tree for valid items.

3. Mining COFI*-tree.

Example 3.1 *Consider the database TDB shown in Table 3.1. It consists of five transactions and six items (A, B, C, D, E, and F). Our mining procedure then sorts the items in each transaction in the ascending order of the frequency of each item (Table 3.2). Afterwards, our algorithm builds an inverted matrix (Table 3.3). Given some auxiliary information (Table 3.4) about the price of each item, we can easily*

identify valid items.

Table 3.1: A sample database TDB

TID	**Itemsets**
T1	B, C, D
T2	A, B, C, D, E, F
T3	A, B, D, E
T4	A, C, D
T5	C, D

Table 3.2: The sorted database

TID	**Itemsets**
T1	B, C, D
T2	F, E, A, B, C, D
T3	E, A, B, D
T4	A, C, D
T5	C, D

Table 3.3: An inverted matrix

Loc	**Index**					
1	(F,1)	(2,1)				
2	(E,2)	(3,1)	(3,2)			
3	(A,3)	(4,2)	(4,3)	(5,3)		
4	(B,3)	(5,1)	(5,2)	(6,3)		
5	(C,4)	(6,1)	(6,2)	(6,4)	(6,5)	
6	(D,5)	$(\emptyset,\emptyset)$	$(\emptyset,\emptyset)$	$(\emptyset,\emptyset)$	$(\emptyset,\emptyset)$	$(\emptyset,\emptyset)$

Table 3.4: The auxiliary information about items

Item	A	B	C	D	E	F
Price	10	20	5	22	15	27

After reviewing the first part of our mining process, let us consider the second part below.

Example 3.2 *Reconsider the database TDB in Table 3.1. Let minsup be 3. Then, we show how our mining procedure constructs (and mines) a COFI*-tree for a valid item A. The first entry (4, 2) for the item A (note that 4 and 2 are row and column indices respectively) in the array for item A indicates that the item in row 4 (item B) is the item after item A in a transaction in the original database. Hence, we can follow the link (4,2), which indicates the entry in column 2 of row 4. The value in (4,2) is (5,2) indicating that the next item is in row 5 (i.e., item C). Similarly, the value in (5,2) is (6,2) indicating that the next item is in row 6 (i.e., item D); the value in (6,2) is (∅,∅) indicating that no more item after D.*

Therefore, we get the transaction {A, B, C, D} and a tree branch with nodes A, B, C, and D with frequency set to 1 (Figure 3.1(a)). Note that this tree is a projection tree for item A that includes all the co-occurrences of item A.

We then apply a similar procedure by following the link in the second entry of A: (4,3) - (6,3) - (∅,∅), which gives A-B-D (Figure 3.1(b)). The frequency of node A is incremented to 2. Then, the frequency of node B is also incremented to 2. Since C is the only child of B in the current COFI-tree, our mining procedure adds D as another child of B. The frequency of the node D is set to 1.*

Again, we start following the third entry for item A (5,3) - (6,4) - (∅,∅), which gives A-C-D. The frequency of node A is incremented to 3. The mining procedure adds C as another child of A, and D is the child of C (i.e., a branch with nodes C and D, where the frequency of both of the nodes are set to 1, is created). Now, the COFI-tree (Figure 3.1(c)) for item A is completed since A has no more entry left in the inverted matrix.*

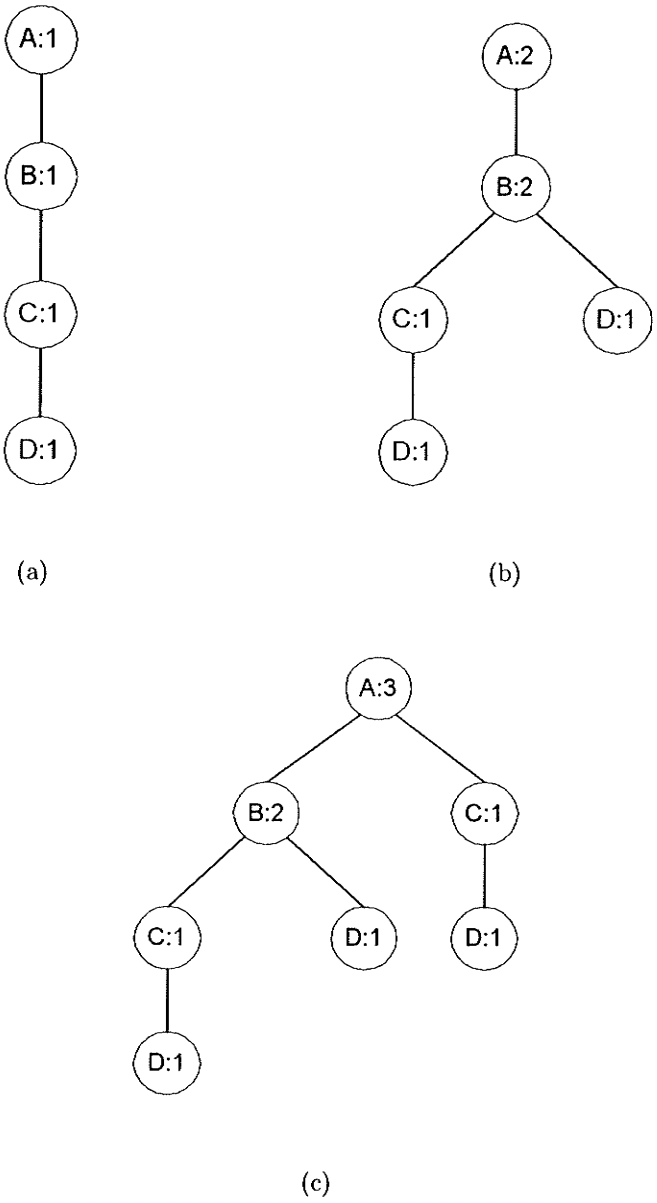

Figure 3.1: COFI*-tree for item A from inverted matrix in Table 3.3

After constructing a COFI-tree for item A, our algorithm finds all the possible frequent itemsets containing A:* $\{A\}$, $\{A,C\}$, $\{A,D\}$, *and* $\{A,C,D\}$. *Then, the algorithm carries out similar steps for COFI*-trees for other items, and finds all frequent*

itemsets: $\{A\}$, $\{C\}$, $\{D\}$, $\{A,C\}$, $\{A,D\}$, $\{C,D\}$, *and* $\{A,C,D\}$.

3.2 Constraint Based Mining: How Our Proposed Inverted Matrix++ Algorithm Handles Constraints with COFI*-trees?

We analyze the constraints and push them inside the mining process. We handle anti-monotone and/or succinct constraints. For constraints that are neither anti-monotone nor succinct, we induce these constraints into weaker anti-monotone or succinct constraints.

3.2.1 Mining Succinct Anti-monotone Constraints

All frequent itemsets that satisfy the **succinct anti-monotone (SAM) constraint** *must* contain only the valid items. For example, constraint "$max(S.Price) \leq 20$" indicates that a valid itemset must contain only those items that have price less than or equal to 20. Therefore, we *only* need to identify all the valid items at the beginning of the mining process. We do not need to construct any COFI*-tree for the invalid items. Our COFI*-trees also keep only valid items (i.e., satisfying both the frequency threshold and constraints). Therefore, we do not need any constraint checking in the mining phase. Here, the frequency test is required only to determine all the valid frequent itemsets. Let us consider Example 3.3 to see the mining process for SAM constraints.

Example 3.3 *Let us use the database shown in Table 3.1. The price of each item is shown in Table 3.4. We assume that the minsup is 3 and the SAM constraint is "$max(S.Price) \leq 20$". From Table 3.3, we note that items A, B, C and E satisfy the constraint. With minsup=3, items A, B, and C and D are frequent. Hence, only items A, B and C satisfy both the minsup and the constraint "$max(S.Price) \leq 20$". Therefore, we construct the COFI*-tree only for these items. The COFI*-tree for A includes only the valid items A, B and C; the COFI*-tree for B includes only the valid items B and C. We do not need to construct any COFI*-tree for C as it is the "last" valid item. All these COFI*-trees are built from the inverted matrix (Table 3.3). Figure 3.2 shows an example. Itemsets mined from these COFI*-trees contain all and only those valid items.*

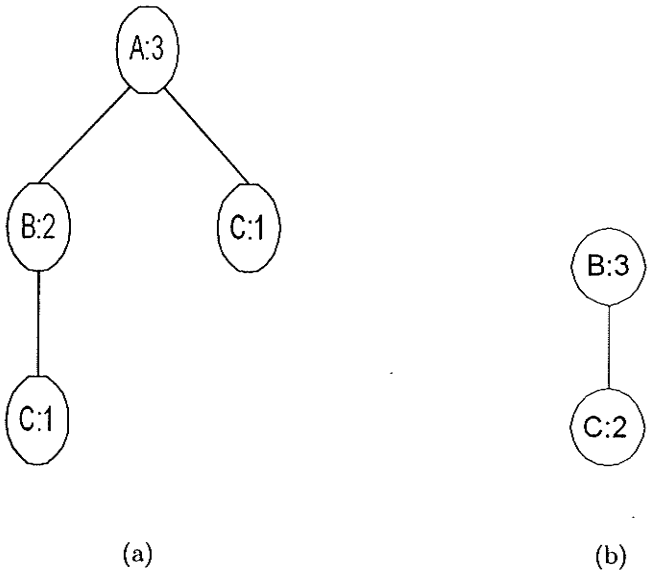

Figure 3.2: COFI*-trees for items A and B

3.2.2 Mining Succinct Non-anti-monotone Constraints

All frequent itemsets that satisfy the **succinct non-anti-monotone (SUC) constraint** *must* contain some mandatory items and may contain some optional items. With the MGF of the SUC constraint, we find some mandatory items that have to be present in the valid itemsets. For example, the MGF for constraint "$max(S.Price) \geq 20$" is $\{X \cup Y | X \subseteq \sigma_{price \geq 20}(Item), X \neq \emptyset, Y \subseteq \sigma_{price<20}(Item)\}$, where X is the mandatory part (which means all the valid itemsets must contain at least one item having price greater than or equal to 20). Therefore, we construct COFI*-trees only for each mandatory item that is frequent. We also construct COFI*-trees for any frequent optional item Z if there exists a frequent mandatory item located below Z in the inverted matrix. The root of the tree for an optional item contains a particular optional item, and the tree nodes are mandatory items or other optional items. All these trees are mined, and all the frequent itemsets are generated by computing the support of the itemsets.

Example 3.4 *Let the minsup be 1 and the succinct constraint be "$max(S.Price) \geq 25$". From Table 3.4, items E and F are mandatory, and items A, B, C and D are optional. We construct COFI*-trees for mandatory items E and F, but how about COFI*-trees for optional items? Since there is no mandatory item located below any of the optional items in the inverted matrix, we do not construct COFI*-trees for those optional items.*

3.2.3 Mining the Anti-monotone Non-succinct Constraint

For example, the constraint "$sum(S.Price) \leq 100$" is anti-monotone but not succinct, because all supersets of an invalid itemset (i.e., having the total price greater than \$100) are invalid. The constraint is not succinct because there is no MGF to enumerate all and only those itemsets that satisfy the constraint. If constraints are not succinct, then there is no way to predetermine the valid itemsets. So, our algorithm continues checking and pruning the itemsets for satisfying **anti-monotone non-succinct (AM) constraints** while mining the trees. The algorithm constructs COFI*-trees for all the valid (respect to the minsup) items. While generating itemsets from a branch of the COFI*-tree, our algorithm does not generate any superset with the invalid (respect to the constraints) itemsets. The algorithm starts analyzing the patterns from the leaves of a tree. Whenever it finds an invalid pattern, it stops generating supersets of that pattern in that branch and starts with another branch.

3.3 Interactive Mining: How Our Proposed Inverted Matrix++ Algorithm Handles Changes of the Support Threshold?

So far, we have shown how we handle constraints. In this section, we explain how an inverted matrix can facilitate interactive mining. Specifically, how an inverted matrix can be used to handle situations where users interactively change the support threshold (a mining parameter) during the mining process.

3.3.1 Handling an Increase of the Minimum Support Threshold

Recall that it is not easy to find an appropriate value for the minimum support threshold. If the value is set too high, just a few itemsets (and may be no itemsets) are returned. If the value is set too low, very large number of itemsets is returned. Hence, mining is supposed to be an interactive process. To enable the user to find an appropriate minimum support threshold value, interactive mining is desired. The question here is: How to handle the change of the support threshold? A naïve approach is to halt the current mining process (which uses the old support threshold), discard all itemsets satisfying the old threshold, and re-mine itemsets from scratch using the new support threshold. While this approach is correct, it is not efficient. This situation is worsened if the change occurs near the end of the mining process. Lots of computation is wasted.

The user can increase or decrease the support threshold to make the change of minsup during the mining process. When the user increases the support threshold, *itemsets satisfying the new threshold are subsets of itemsets satisfying the old threshold.* Therefore, discarding the itemsets satisfying the old threshold and re-computing the itemsets satisfying the new threshold is a waste of computation.

Instead, our proposed Inverted Matrix++ algorithm handles the increase of the minimum support threshold (minsup) as follows. If the user increases the threshold, we skip constructing COFI*-trees for items (in the inverted matrix) having frequency value between the old and the new thresholds. For example, with the inverted matrix in Table 3.3 and the minsup changing from 2 to 3, we skip constructing the COFI*-

tree for item E (which have a frequency of 2 that no longer satisfies the new threshold) because any itemsets that can be mined from this COFI*-tree for item E contains E itself (an item which is no longer frequent w.r.t. the new minsup). Any itemset containing any infrequent item is infrequent (i.e., will not satisfy the new minsup). Similarly, when minsup is changed from 2 to 4, we skip constructing the COFI*-trees for items A, B, E (which have frequencies < 4 and hence, no longer satisfies the new threshold). Since rows in the inverted matrix are arranged in ascending frequency order of items, it is easy to determine which items to be skipped. This deals with the "unprocessed" items in the inverted matrix, but how about the "processed" items? Itemsets mined from the COFI*-trees of the "processed" items satisfy the old minsup. This means that some itemsets satisfy the new minsup, and some do not. Hence, our proposed algorithm performs a post-processing step to check all processed itemsets to ensure that each of them satisfies the new minsup (and filter out those satisfying the old minsup but not the new minsup).

To summarize, the key steps of handling an increase of minsup are: (1) skip constructing COFI*-trees for items not satisfying the new minsup, and (2) perform a post-processing step to discard the "processed" itemsets that satisfy the old minsup but not the new minsup.

3.3.2 Handling a Decrease of the Minimum Support Threshold

The previous section (Section 3.3.1) showed how we handle an increase of minsup. Thus, for a decrease of minsup, we know that *itemsets satisfying the new threshold are*

supersets of itemsets satisfying the old threshold. Hence, all "processed" itemsets that satisfy the old minsup are guaranteed to satisfy the new minsup. We do not need to perform any post-processing step or to discard any "processed" itemsets. However, as itemsets satisfying the new minsup are supersets of itemsets satisfying the old minsup, the question is: How to find the "delta" itemsets (i.e., itemsets satisfying the new minsup but not the old minsup)?

When the user decreases minsup, we can halt the current mining process (which uses the old minsup) and resume it with the new minsup. In addition, we construct COFI*-trees for items satisfying the new minsup but not the old minsup. These trees help to find the "delta" itemsets. For example, when users change the minsup from 3 to 2 after processing COFI*-trees for items A and B in Table 3.3, we construct COFI*-trees for item E (which finds itemsets {E}, {A, E}, {B, E}, {D, E}, {A, B, E}, {A, D, E}, {B, D, E} and {A, B, D, E} satisfying the new minsup of 2).

On the surface, the above appears to be a good solution. However, a careful analysis reveals that we still miss some itemsets (e.g., {B, C}, {B, C, D}). Why? The reason is that when we process the COFI*-trees from items B and C, we use the old minsup of 3. At that time, itemsets {B, C} and {B, C, D} do not satisfy the old minsup of 3. After the change in minsup, the new minsup becomes 2. Hence, these two itemsets satisfy the new minsup. However, the trees have been processed (before the change)!

In order to solve this problem, we introduce an additional parameter called Pre-Minsup (where PreMinsup $<$ minsup). This parameter is set by the user. During the mining process, our proposed algorithm finds itemsets satisfying PreMinsup (instead

of minsup). Hence, when returning itemsets to the user, we only return those satisfying minsup (among those satisfying PreMinsup). When users change the minsup during the mining process, there are two cases. If the new minsup $\geq$ PreMinsup, we just need to use the new minsup when returning the answer (i.e., itemsets satisfying the new minsup). Otherwise (i.e., if the new minsup $<$ PreMinsup, we need to reconstruct COFI*-trees for the "processed" items. Let us reconsider the above example, with PreMinsup=2 $<$ minsup=3, we find itemsets {B, C} and {B, C, D} when processing COFI*-trees for items B and C.

To summarize, the key steps of handling a decrease of minsup are: (1) continue the mining process but with the new minsup, (2) construct COFI*-trees for "delta" items (i.e., items satisfying new minsup but not the old minsup), and (3) reconstruct COFI*-trees and re-mine itemsets for the "processed" items if the new minsup $<$ PreMinsup.

3.4 Summary

In this chapter, we described our new work—the Inverted Matrix++ algorithm—an algorithm for *constraint based interactive* frequent pattern mining that can mine *large databases*. The algorithm uses two structures: (i) an inverted matrix (a disk based data structure) and (ii) a COFI*-tree (our newly proposed memory based tree structure). The resulting mining algorithm allows the user to specify constraints, and handle constraints by pushing them in the mining process when finding itemsets that (satisfy the constraints. Moreover, our algorithm also allows the user to change the minimum support threshold. When the threshold is changed, our algorithm does not

need to find frequent itemsets from scratch. In other words, our proposed Inverted Matrix++ algorithm provides the user with constraint based mining and interactive mining, and it can find frequent itemsets from large databases.

Chapter 4

Experimental Results

This chapter presents the experimental results for our Inverted Matrix++ algorithm. We conducted four different sets of experiments and compared our developed Inverted Matrix++ algorithm with some existing algorithms. We used both synthetic and real-life data in the experiments.

4.1 Experimental Setup

We implemented the algorithms using the C programming language and analyzed the performance for large synthetic databases. We generated our sample synthetic datasets by using the IBM synthetic data generator [AS94]. We also used some real-life databases from the University of California - Irvine (UCI) Machine Learning Depository [BM98]. These testing databases are considered as the benchmark datasets in our research field. We implemented our algorithm and ran our experiment on a Pentium-IV machine with 2GHz processor, 512MB memory, and 30GB hard drive.

Since our goal is to provide constraint mining technique for large databases, our experiments were performed on the IBM datasets ranging from 1 million to 10 million (e.g., 1M, 5M, and 10M) transactions each containing an average of at least a dozen of items with a domain of approximately 1,000 items. From the UCI Machine Learning Repository, we used the mushroom dataset, which contains 137 distinct domain items and each transaction is of a fix length of 22 items.

To evaluate the effectiveness of our proposed Inverted Matrix++ algorithm, we first compared it with the Inverted Matrix algorithm [EZ04] in the experiment. We ran the Inverted Matrix algorithm to find all frequent itemsets and then conducted a post-processing step to check if the frequent patterns satisfy the constraints. In contrast, our proposed algorithm pre-pruned the itemsets that do not satisfy the constraints. Here, our question was: How much can we gain from pre-pruning instead of post-pruning? We picked the existing Inverted Matrix algorithm for comparison because it also used the inverted matrix (a disk based data structure). Evaluation results show the effectiveness of the constraint based mining aspect of our Inverted Matrix++ algorithm.

In addition, we also compared our algorithm with the FPS algorithm [LLN02], which is an FP-tree based algorithm. Note that it is also a constraint based algorithm. The key difference between this algorithm and our proposed algorithm is the use of different data/tree structures: The former uses FP-trees, whereas the latter uses the inverted matrix and the COFI*-trees. Evaluation results show the applicability and effectiveness of our Inverted Matrix++ algorithm in handling large databases.

Furthermore, we also compared the results of the proposed algorithm using con-

straints of other selectivity. All algorithms then gave the same results (i.e., the same set of valid frequent itemsets from large real world databases), though some of them took a longer time than the others.

Various forms of tests were conducted on the sample datasets to determine the execution time, scalability, and memory occupancy. In particular, we conducted the following experiments:

1. In our first set of experiments, we conducted the following experiments to test runtime and scalability of our algorithms when it mines large databases:

   (a) We varied the support threshold from 0.01% to 1%. The higher the support threshold, the higher was the number of frequent itemsets returned.

   (b) We also varied the size of the database from 1M to 10M. We studied the runtimes of those executions.

2. In the second set of experiments, we tested the applicability of our proposed algorithm (by observing whether or not the above algorithms can mine from large databases). In other words, we wanted to see if the algorithm be able to return all and only those valid itemsets. Our proposed algorithm is expected to be able to deal with large databases. In addition, we also measured the runtimes (i.e., the total CPU and I/O time) of our algorithm. Since our algorithm skips infrequent items and it does not need to build the entire tree (i.e., it does not require extensive memory space), it is expected to be well-suited to mine large databases.

3. In the third set of experiments, we measured the amount of required computa-

tion (i.e., to count the occurrences of constraint checking and support counting). Our algorithm pre-prunes the itemsets according to the user constraint by exploring the property of constraints (succinct anti-monotone constraints, succinct non-anti-monotone constraints, and anti-monotone non-succinct constraint) and by using an inverted matrix. Hence, our algorithm is expected to require less constraint checking and support counting.

4. In our fourth set of experiments, we also compared our proposed algorithm with the following algorithm:

   - **Rerun Inverted Matrix++**, where we first ran Inverted Matrix++ for a user-specified minimum support threshold, and then halted the program during the execution and reran Inverted Matrix++ with a different minimum support threshold.
   - **Rerun FPS**, where we first ran the FPS algorithm [LLN02] for a user-specified minimum support threshold, and then halted and reran FPS from scratch with a different minimum support threshold.

   Our proposed algorithm is expected to be faster than FPS because our algorithm does not start from scratch whenever users change the support threshold. Evaluation results show the effectiveness of the interactive mining aspect of our algorithm.

4.2 Experiment Set 1: Testing the Execution Time and the Scalability

In this first set of experiments, we focused on the effect on execution time of the tree building and mining. Here, both synthetic data (IBM dataset) and real life data from UCI data repository (mushroom data) were used to perform the experiment.

Experiment 4.1 (Testing with different minsup & the IBM dataset) In this experiment, if the user increased the minsup, then fewer items satisfied the minsup. Hence, fewer items would be selected. Therefore, it is expected that, if we increased the minsup, the total mining time would also be reduced. The graph in Figure 4.1 shows the effects on execution time with respect to the minsup on the IBM dataset (1000K transactions). In the graph, we show the total runtime for our Inverted Matrix++ algorithm and the breakdown (the runtime for building COFI*-trees and the runtime for mining COFI*-trees). The graph agrees with the expectation (i.e., the

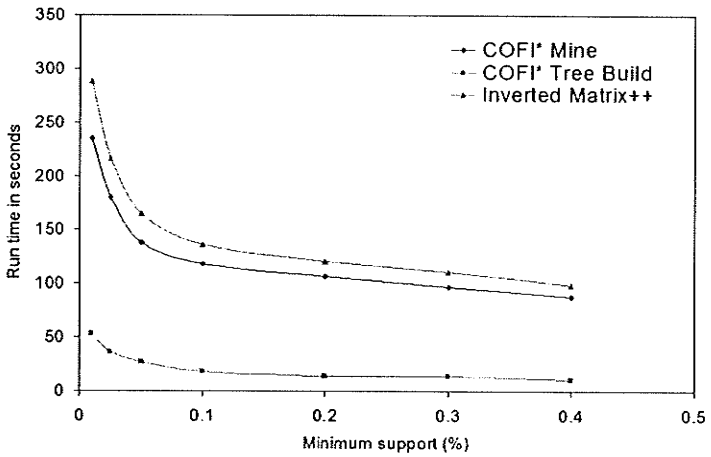

Figure 4.1: Runtime with respect to minimum support with the IBM dataset (Experiment 4.1)

execution time decreased with the increase of the minsup). In Figure 4.1, there is a high execution time with a low minsup of 0.01% and a quick dive in execution time with a minsup of at least 0.1%. The reason for the steep slope is that the number of frequent itemsets increased exponentially with the decrease of the minsup. We noticed that the time required for tree building was much less then the time required for mining the trees. Therefore, the trend of the total time of the execution depended on the mining time. Besides, the time required for building an inverted matrix did not change with different minsup. The reason is that the construction of the inverted matrix did not consider the minsup as parameters, and it stored all items in transactions irrespective of their frequencies.

Experiment 4.2 (Testing with different minsup & the mushroom dataset)

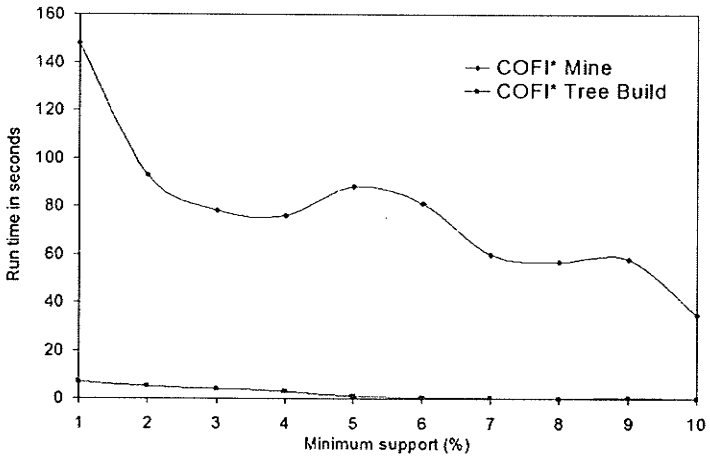

Figure 4.2: Runtime with respect to minimum support with the UCI mushroom dataset (Experiment 4.2)

The graph in Figure 4.2 shows the effects on execution time with respect to minsup on the mushroom dataset. In Figure 4.2, the graph is less smooth than that of Figure 4.1. The bumps occurred because the items in the mushroom dataset were not uniformly distributed as in case of the synthetic IBM dataset.

Experiment 4.3 (Testing with different sizes of the IBM dataset) In this experiment, we tested the scalability of our Inverted Matrix++ algorithm in terms of the change in data size. Here, we used the IBM dataset (i.e., 100K, 200K, ..., 1000K). Unlike the previous experiment, we changed the size of the datasets and kept the minsup constant at 0.01% for IBM dataset. The execution time increased with the increase of data size and showed a linear scale-up.

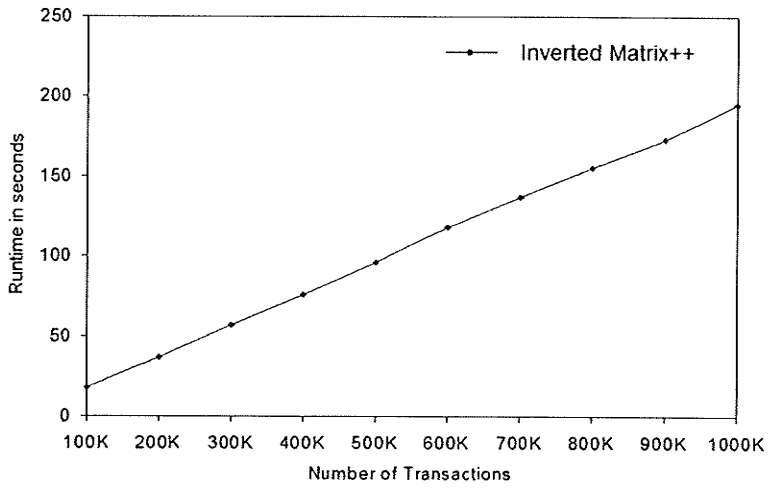

Figure 4.3: Runtime w.r.t. the size of the IBM data dataset (Experiment 4.3)

The graph in Figure 4.3 shows the effects on execution time with respect to the data size on the IBM dataset. We plotted the graph for the total runtime of the Inverted Matrix++. In Figure 4.3, we can see a gradual increase of the execution

time with the increase of the dataset size.

Experiment 4.4 (Testing with different sizes of the mushroom dataset) In this experiment, we tested the scalability of our Inverted Matrix++ algorithm in terms of the change in data size. Here, we varied the portions of the UCI mushroom dataset (i.e., 1K, 2K, ..., 8K) to be used in the experiment. We kept the minsup constant at 10% for UCI dataset.

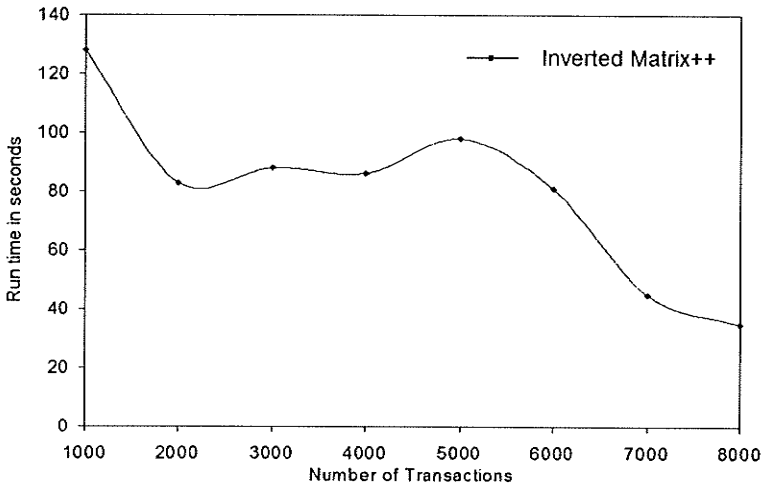

Figure 4.4: Runtime w.r.t. the size of the UCI dataset (Experiment 4.4)

The graphs in Figure 4.3 and Figure 4.4 show the effects on execution time with respect to the data size on IBM dataset and on UCI mushroom dataset respectively. We have plotted graphs for the total timing of the Inverted Matrix++. In Figure 4.3, we can see a gradual increase of the execution time with the increase of the dataset size. Figure 4.4 shows an interesting outcome. The non-uniform distribution of items in the dataset is the main reason behind this outcome. The bumps on the graphs are clearly shown with the increase in data size. Furthermore, there was an decrease

on execution time instead of increase with respect to data size. The reason for this trend of execution time is the distribution of frequent itemsets in the dataset. We have already mentioned that the results on a dataset with uniform distribution would most likely show the linear scalability. We also found that there was an decrease on the number of frequent itemsets instead of increase with respect to data size. Furthermore, the mining time shown in Figure 4.4 appeared to be proportional to the number of output frequent itemsets. This implies that the trend is due to the uneven distribution of frequent itemsets in the mushroom dataset which is effecting the mining time. However, Figure 4.3 where the IBM synthetic datasets with uniform distribution of items are used shows that the Inverted Matrix++ execution time linearly scales with the increasing data size.

4.3 Experiment Set 2: Testing the Effect of Constraints on Execution Time

In the second set of experiments, we evaluated our constraint based rule mining technique. Here, we compared (a) the FPS algorithm [LLN02] and (b) the Inverted Matrix algorithm [EZ04] followed by a post-processing step (for checking every frequent itemset to see if it satisfies the user-defined constraints) with our proposed Inverted Matrix++ algorithm. In the experiments, we fixed the minimum support threshold to 0.1%. We used the IBM dataset that contains 1000K transactions. We varied the type of constraints (SAM, SUC, AM) and the selectivity of the constraints. A constraint with p% selectivity means that p% of distinct items are selected (i.e.,

satisfying the constraint) for mining. Therefore, the higher the selectivity, the higher was the expected number of itemsets to be returned (and the longer would be the expected execution time).

Experiment 4.5 (SAM constraints) For this experiment, we selected the SAM (succinct and anti-monotone) constraint with selectivity ranging from 10% to 100%. We observed from Figure 4.5 that if the selectivity is lower, the gain is higher. The gain is significant (the reduction in execution time gained by our Inverted Matrix++ algorithm was almost 3 times) when compared with Inverted Matrix followed by the post-processing step when the selectivity is low (10% to 50%). The trend of the graphs for FPS and Inverted Matrix++ was similar though the Inverted Matrix++ algorithm outperformed FPS for low selectivity of the SAM constraint. The gap between the runtimes of the two algorithms decreased with the increase in the selectivity. We noticed that the runtime of our algorithm was slightly higher than the runtime of

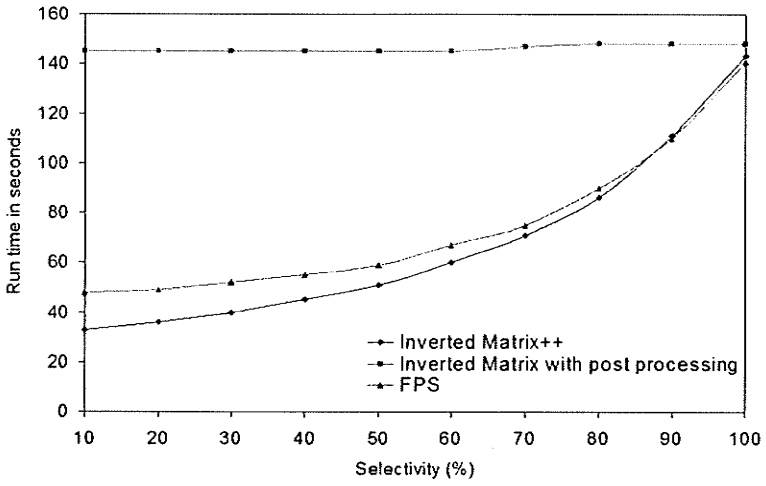

Figure 4.5: Changing the selectivity for SAM constraints (Experiment 4.5)

the FPS algorithm when the selectivity was 90% or above. This is because we need extra checks when reading the transaction from the inverted matrix. This overhead is lowered when more and more items are discarded. Therefore, the gap increased with decrease in selectivity. The results clearly highlighted the power of constraint based mining.

Experiment 4.6 (SUC constraints) In this experiment, we evaluated constraint based mining with SUC (succinct but not anti-monotone) constraints. Here, we had the similar experimental setup as the previous experiment (the experiment with SAM constraint). The result in Figure 4.6 shows a significant gain of using our proposed Inverted Matrix++ algorithm when compared with using Inverted Matrix plus the post processing step, when the selectivity is lower (less then 25% selectivity). This is because the pruning power for the non-anti-monotone constraint is less strict than the anti-monotone constraint. In the previous experiment (involving SAM constraint),

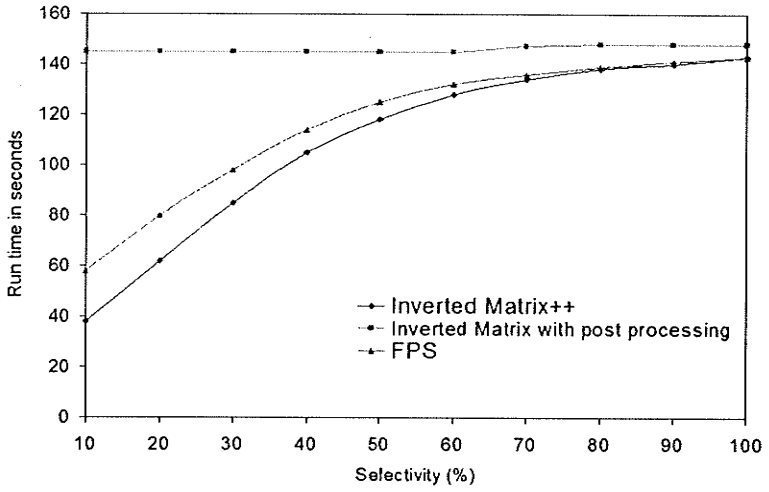

Figure 4.6: Changing the selectivity for SUC constraints (Experiment 4.6)

we achieved greater degree of pruning because the SAM constraint involves only the mandatory group whereas the SUC constraint involves both mandatory and optional groups. Recall from the algorithm described in Chapter 3 that for the SAM constraint, we only need to build the COFI*-tree for the mandatory item. On the other hand, for the SUC constraint, we need to build COFI*-tree for all the items regardless of being mandatory or optional. Hence, we only achieved greater gain for SUC constraint when we have lower selectivity. This is reflected in the results (comparing the graphs shown in Figure 4.5 and Figure 4.6) that with the same p% (30% to 70%) of selectivity the execution time for the SUC constraint was about twice than the execution time for the SAM constraint.

Though the experimental results show that the SAM constraint was more powerful than the SUC constraint, our Inverted Matrix++ algorithm showed better performance in both cases (SAM and SUC constraints) when compared with the two algorithms. Thus, the application of both constraints not only successfully enhanced the performance of the large database mining but also output frequent itemsets that are interested to the users.

Experiment 4.7 (AM constraints) For this experiment, we evaluated the effect of constraints on the execution time when the anti-monotone non-succinct (AM) constraint is used. However, we did not vary the selectivity. To find items satisfying the AM constraints, our algorithm continues checking and pruning the itemsets for satisfying *anti-monotone non-succinct (AM) constraints* while mining the COFI*-trees. We cannot ignore any item while building COFI*-tree. So, rather than varying the selectivity, we used different constraints (e.g., $sum(S.Price) \leq 50$, $sum(S.Price) \leq 100$,

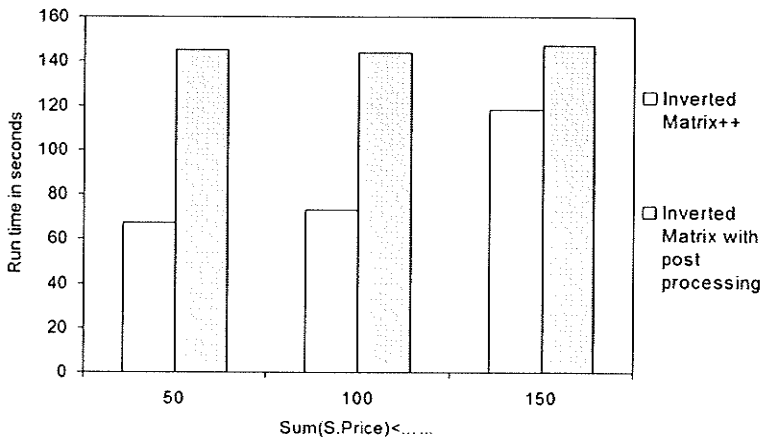

Figure 4.7: Experiments with different AM constraints (Experiment 4.7)

$sum(S.Price) \leq 150$). Figure 4.7 shows that our algorithm Inverted Matrix++ outperformed the Inverted Matrix algorithm.

4.4 Experiment Set 3: Testing the Effect of Interactive Mining

In this experiment, we tested the effect of interactive mining. Recall that the user can increase or decrease the support threshold to make the change of the minsup during the mining process. Therefore, regarding the change to the support threshold, there are two types: (1) increasing the minsup and (2) decreasing the minsup. We performed different experiments for these two types of *minsup* changes.

Experiment 4.8 (Increasing the minsup) We ran our algorithm using the IBM dataset with 1M transactions. To test the effect of interactively increasing the minsup, we first executed our Inverted Matrix++ algorithm with *minsup=0.1%*, and then

increased the *minsup* so that the new minsup became 0.2%, 0.3%, and 0.4%. We compared our algorithm with FPS and Inverted Matrix (w/o interactive mining). Figure 4.8 shows the experimental result.

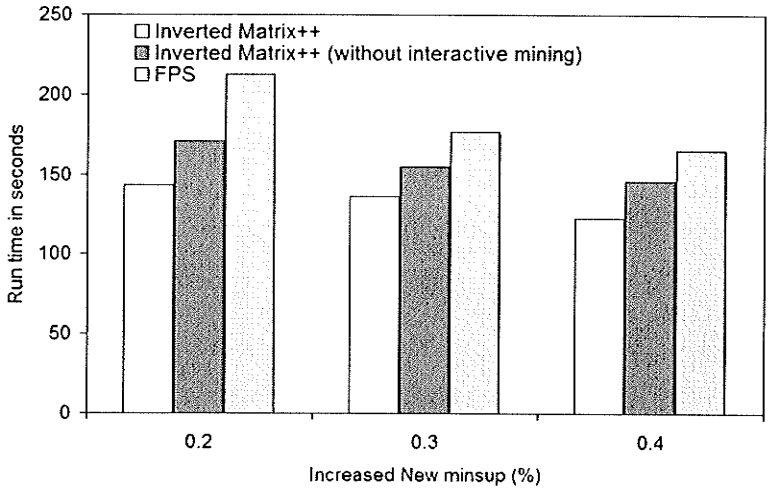

Figure 4.8: Interactive mining with an increasing minsup (Experiment 4.8)

Experiment 4.9 (Decreasing the minsup) To test the effect of interactively decreasing the minsup, we executed our algorithm with *minsup=0.4%* and then decreased the minsup so that the new minsup became 0.3%, 0.2%, and 0.1%. Figure 4.9 shows the experimental result.

To handle the problem of decreasing the minsup, we used the *PreMinsup* to find all the frequent itemsets though the itemsets satisfying the actual minsup were returned. For this reason, we need to generate more itemsets, which require extra time to find valid itemsets with the new minsup from the already generated itemsets using *PreMinsup*.

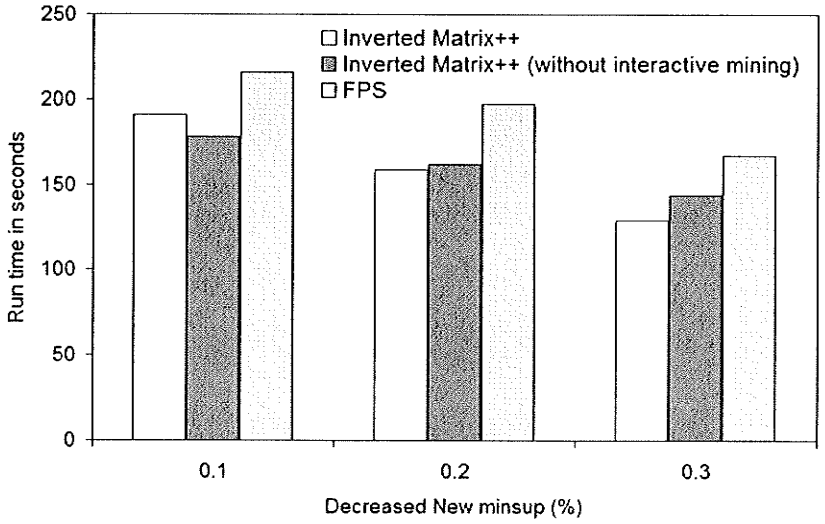

Figure 4.9: Interactive mining with a decreasing minsup (Experiment 4.9)

4.5 Experiment Set 4: Testing the Applicability for Mining Large Databases

Experiment 4.10 (Runtime for mining large databases) For this experiment, we ran our Inverted Matrix++ algorithm using large datasets with 1M, 5M, and 10M transactions. The *minsup* was 0.01%. Our algorithm efficiently mined all of the datasets. We also ran the FPS algorithm using the same datasets. Figure 4.10 clearly shows that our algorithm outperformed FPS. The successful execution shows the applicability of our constraint based interactive large database mining algorithm.

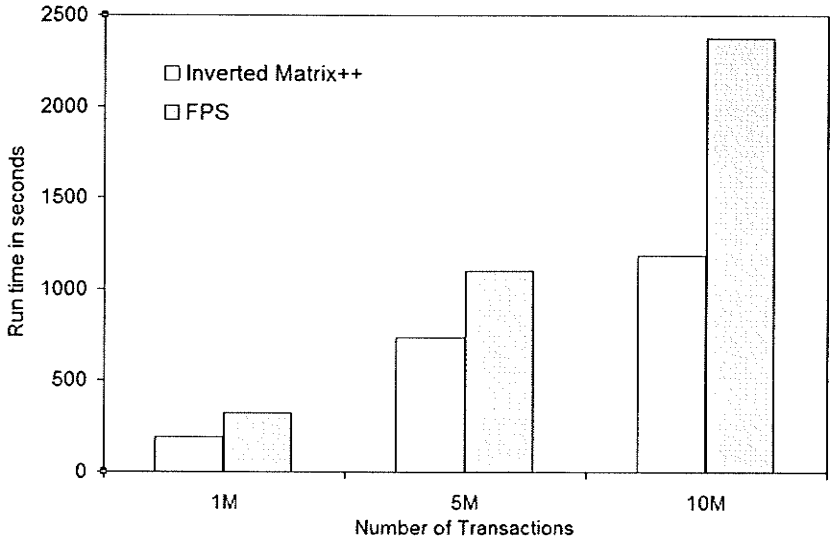

Figure 4.10: Runtime for mining large datasets (Experiment 4.10)

Experiment 4.11 (Memory space for mining large databases) We also measured the memory usage of the COFI*-tree and compared it with the FP-trees used in the FPS algorithm. The FP-tree captures all the frequent items and the associated transactions. On the other hand, the COFI*-tree captures the co-occurrences of a frequent items at a time. This explains why the FP-tree occupied more memory than the COFI*-tree. In the experiment, we found that the average memory usage for COFI*-tree was always less than that of FP-tree.

4.6 Summary

In this chapter, we showed our experimental results. We went through four different sets of experiments. In our first set, we tested runtime and scalability of our proposed Inverted Matrix++ algorithm when it mined large databases. The results

showed that the runtime (including the time required for building COFI*-trees and mining) decreased when the minsup increased. Our algorithm generally scaled up linearly w.r.t. the size of the dataset. In the second set, we measured the amount of required computation (i.e., to analyze the occurrences of constraint checking and support counting). The results showed the effectiveness of constraint mining of ours (when compared with FPS and Inverted Matrix plus post-processing step). Runtimes were proportional to the selectivity of constraints. In our third experiment set, we applied the interactive mining technique. The results showed the effectiveness of interactive mining of ours when users changed the minsup. Finally, we tested the applicability of our proposed algorithm. The results showed our proposed Inverted Matrix++ algorithm took a reasonable amount of runtime and memory space when mining large databases.

Chapter 5

Conclusions and Future Work

5.1 Conclusions

Over the past decade, many frequent-pattern mining algorithms have been developed. However, many of them rely on the availability of large memory. Their performance degrades if the available memory is limited because of the overhead and extra I/O costs. Moreover, among the algorithms that mine large databases, many of them do not provide users control over the mining process through the use of constraints. Constraint based mining is very important because it encourages users focus on only those patterns that are interesting to the users. Furthermore, among the algorithms that handle user constraints, many of them do not allow users to interactively change the mining parameters during the mining process. As mining is usually an iterative process, it is important to have an algorithm that supports *constraint based mining* and allows users to *interactively* mine *large databases.*

In this thesis, we developed the Inverted Matrix++ algorithm—an algorithm for

constraint based interactive frequent pattern mining that can mine *large databases.* Specifically, our algorithm pushes the user specified constraints in the mining process when finding itemsets that satisfy the constraints. The algorithm allows the user to change the minimum support threshold. When the threshold is changed, our algorithm does not need to find frequent itemsets from scratch. In addition, we proposed a new memory based tree structure called a COFI*-tree and used a disk based data structure called an inverted matrix in our algorithm for interactive mining from large databases.

Experimental results on both synthetic and real-life datasets showed the following. The runtime (including the time required for building COFI*-trees and mining) decreased when the minsup increased. Our algorithm generally scaled up linearly w.r.t. the size of the dataset. Our algorithm was effective in handling user specified constraints. Runtimes were proportional to the selectivity of constraints. The algorithm was also effective in handling user changes of the minimum support threshold. Moreover, our proposed Inverted Matrix++ algorithm took a reasonable amount of runtime and memory space when mining large databases.

5.2 Future Work

Our Inverted Matrix++ algorithm handles succinct and/or anti-monotone constraints effectively. To improve our proposed Inverted Matrix++ algorithm, we plan to directly handle those constraints that are neither succinct nor anti-monotone (e.g., the constraint "$sum(S.Price) \geq 100$", which finds all the itemsets where the total price of the items in each itemset is greater than \$100).

We also plan to develop a parallel version of our proposed Inverted Matrix++ algorithm. For example, the COFI*-tree for one valid item from an inverted matrix is independent of another item. We could improve the runtime by implementing a parallel version and obtain the benefits of parallel computation.

Furthermore, given that our proposed Inverted Matrix++ algorithm was designed to provide users with constraint based interactive frequent pattern mining from large databases, we also plan to extend it as follows. First, we would like to apply our algorithm to find frequent patterns from large real-life databases such as large amounts of health surveillance data. Second, we would like to make use of these frequent patterns to detect anomalies. To elaborate, any patterns that are deviated from the frequent patterns returned by our algorithm are likely to be anomalies. Third, we would like to modify our Inverted Matrix++ algorithm so that it could *directly* mine for abnormal patterns from these large health surveillance databases.

Bibliography

[AIS93] R. Agrawal, T. Imielinski, and A. Swami. Mining association rules between sets of items in large databases. In *Proc. ACM SIGMOD*, pages 207–216, 1993.

[AS94] R. Agrawal and R. Srikant. Fast algorithm for mining association rules. In *Proc. VLDB*, pages 487–499, 1994.

[AS95] R. Agrawal and R. Srikant. Mining sequential patterns. In *Proc. IEEE ICDE*, pages 3–14, 1995.

[BM98] C. L. Blake and C. J. Merz. UCI repository of machine learning databases, 1998. Department of Information and Computer Science, University of California, Irvine, CA, USA, `www.ics.uci.edu/~mlearn/MLRepository.html`.

[BMS97] S. Brin, R. Motwani, and C. Silverstein. Beyond market basket: generalizing association rules to correlation. In *Proc. ACM SIGMOD*, pages 265–276, 1997.

[BMUT97] S. Brin, R. Motwani, J. Ullman, and D. Tsur. Dynamic itemset counting

and implication rules for market basket data. In *Proc. ACM SIGMOD*, pages 255–264, 1997.

[Dun03] M. Dunham. *Data Mining: Introductory and Advanced Topics.* Prentice Hall, 2003.

[EZ03a] M. El-Hajj and O. R. Zaïane. Inverted matrix: efficient discovery of frequent items in large datasets in the context of interactive mining. In *Proc. ACM SIGKDD*, pages 109–118, 2003.

[EZ03b] M. El-Hajj and O. R. Zaïane. Non recursive generation of frequent k-itemsets from frequent pattern tree representations. In *Proc. DaWaK*, pages 371–380, 2003.

[EZ04] M. El-Hajj and O. R. Zaïane. COFI approach for mining frequent itemsets revisited. In *Proc. DMKD*, pages 70–75, 2004.

[GZ04] G. Grahne and J. Zhu. Mining frequent itemsets from secondary memory. In *Proc. IEEE ICDM*, pages 91–98, 2004.

[Hid99] C. Hidber. Online association rule mining. In *Proc. ACM SIGMOD*, pages 145–156, 1999.

[HK06] J. Han and M. Kamber. *Data Mining: Concepts and Techniques.* Morgan Kaufman, San Francisco, CA, USA, 2006.

[HPY00] J. Han, J. Pei, and Y. Yin. Mining frequent patterns without candidate generation. In *Proc. ACM SIGMOD*, pages 1–12, 2000.

[KMR+94] M. Klemettinen, H. Mannila, P. Ronkainen, H. Toivonen, and A. Verkamo. Finding interesting rules from large sets of discovered association rules. In *Proc. CIKM*, pages 401–408, 1994.

[Leu04] C. K.-S. Leung. Interactive constrained frequent-pattern mining system. In *Proc. IDEAS*, pages 49–58, 2004.

[LLN00] L. V. S. Lakshmanan, C. K.-S. Leung, and R. T. Ng. The segment support map: scalable mining for frequent itemsets. *SIGKDD Explorations*, 2(2):21–27, 2000.

[LLN02] C. K.-S. Leung, L. V. S. Lakshmanan, and R. T. Ng. Exploiting succinct contraints in FP-tree. *SIGKDD Explorations*, 4(1):40–49, 2002.

[LLN03] L. V. S. Lakshmanan, C. K.-S. Leung, and R. T. Ng. Efficient dynamic mining of constrained frequent sets. *ACM TODS*, 28(4):337–389, 2003.

[LNM02] C. K.-S. Leung, R. T. Ng, and H. Mannila. OSSM: a segmentation approach to optimize frequency counting. In *Proc. IEEE ICDE*, pages 583–592, 2002.

[NLHP98] R. T. Ng, L. V. S. Lakshmanan, J. Han, and A. Pang. Exploratory mining and pruning optimizations of constrained associations rules. In *Proc. ACM SIGMOD*, pages 13–24, 1998.

[PCY95] J. Park, M.-S. Chen, and P. S. Yu. An effective hash-based algorithm for mining association rules. In *Proc. ACM SIGMOD*, pages 175–186, 1995.

[PHL01] J. Pei, J. Han, and L. V. S. Lakshmanan. Mining frequent itemsets with convertible constraints. In *Proc. IEEE ICDE*, pages 433–442, 2001.

[SBMU98] C. Silverstein, S. Brin, R. Motwani, and J. Ullman. Scalable techniques for mining causal structures. In *Proc. VLDB*, pages 594–605, 1998.

[SON95] A. Savasere, E. Omiecinski, and S. Navathe. An efficient algorithm for mining association rules in large database. In *Proc. VLDB*, pages 432–444, 1995.

[TSK06] P.-N. Tan, M. Steinbach, and V. Kumar. *Introduction to Data Mining.* Addison-Wesley, Boston, MA, USA, 2006.

[ZPLO96] M. Zaki, S. Parthasarathy, W. Li, and M. Ogihara. Evaluation of sampling for data mining of association rules. Technical report 617, Computer Science Department, University of Rochester, NY, USA, 1996.