Concurrent Data Mining with a Large Number of Users

Li Zhiheng

A Dissertation Submitted in Partial Fulfilment

of the Requirements for the Degree of

Master of Philosophy

in

Systems Engineering and Engineering Management

Supervised by

Prof. Jeffrey Xu, Yu

June 2004

Abstract

Data mining has attracted a lot of research efforts during the past decade. However, little work has been reported on supporting a large number of users with similar data mining tasks. In this thesis, we present a data mining proxy approach that provides basic services so that the overall performance of the system can be maximized for frequent itemset mining. This work is motivated by the fact that the pattern-growth method is one of the most efficient methods for frequent pattern mining which constructs an initial tree and mines frequent patterns on top of the tree. Concerning the proxy needs to provide the service for different clients, some methods to protect privacy and security should also been applied on the proxy.

Our data mining proxy is designed to fast respond a user's request by constructing the required tree using both trees in the proxy that are previously built for other users' requests and trees stored on disk that are pre-computed from transactional databases. We define a set of basic operations on pattern trees including tree projection and tree merge. We show that the proposed operations guarantee that the trees constructed are smallest and sufficient to support frequent pattern mining tasks. In addition, several strategies are proposed to manage trees when memory of the proxy becomes full. Our performance study indicated that the data mining proxy significantly reduces the I/O cost and CPU cost to construct trees. The frequent pattern mining costs with the trees constructed can be minimized. More importantly, the tree representation we adopted can be easily converted into many widely-used tree representations so that the proxy can indeed provide services to various applications that implements differ-

ent mining algorithms.

On the privacy protecting aspect, we observe that in the environment with multiple proxies the user only want to get the patterns in the whole data and each proxy wants to protect its own data so that the patterns of itself will not be known by the others. Therefore, suppose one user requests data from a group of proxies, the privacy protecting method aims at that the user can get the whole data while it protects that the user does not know the exact proxy where the specific datum come. To achieve it, we propose a novel method that the proxy provide their own encrypted data and after all data are collected together, these data are decrypted. Thus the user can only get data, but do not know their specific provider.

摘　要

在過去的十年中，很多的研究力量投入到了資料挖掘這一領域。然而尚未有研究提出如何解決多個用戶提出多個相似的資料挖掘查詢的問題。在本篇論文中，我們設計了一個資料挖掘伺服器的方案。此方案可以使全系統提供頻繁模式挖掘的能力被最佳化。設計此方案是基於模式增長的方法已經被證明在頻繁模式挖掘中是最有效的，而此類方法需要首先建立一個初始樹，然後在這棵樹上進行頻繁模式的挖掘。考慮到伺服器需要對不同的客戶提供服務，我們還需要在伺服器中加入隱私保護的措施。

我們的資料挖掘伺服器通過利用爲先前的資料挖掘查詢建立的樹以及爲全資料庫在硬碟上建立的樹來快速地爲新的資料挖掘查詢建立一棵新樹。爲此，我們定義了一組樹運算元來對已建立的樹進行投影以及合併的操作。我們可以證明這些運算元可以保證所建立的樹對於所需要進行的挖掘是最小的但又是足夠的。另外我們還提出了當伺服器的存儲空間用盡時如何進行處理的幾種策略。我們的實驗證明，資料挖掘伺服器可以明顯的降低輸入輸出流量和 CPU 建樹的工作量。同時在這些樹上進行的挖掘的工作量也被最小化了。更重要的是，我們所採用的這一種樹的形式可以被輕易地轉化爲其他各種樹的形式，這樣此伺服器實際上可以爲不同的應用提供不同的資料挖掘演算法。

在隱私保護方面，我們觀察到用戶只需要獲得整體資料中的頻繁模式，而在多個伺服器的環境中，每個伺服器都希望保護自己所擁有的頻繁模式而不被他人所知道。因此，如果用戶需要從多個伺服器獲得資料，我們只需要保證用戶可以獲得全部的資料，而不需要讓他知道這些資料是分別由誰提供的。爲了實現這一

目標，我們設計了一個方法讓每一個伺服器提供資料時對資料進行加密，而當用戶獲得所有的加密資料之後，再對整體進行解密。這樣用戶可以獲得全部的資料，而又不知道資料的具體來源。

Acknowledgement

I would like to show my deep thankfulness to my supervisor Professor Jeffrey Yu who has taught me not only the knowledge but also the real connotation of research in these two years. His impeccable supervision has taught me a lot, not only the sophisticated knowledge and techniques but also the earnest and studious manners in research work. His encouragement is also the great power to help me get through the difficulties in research.

I also need to thank to my friends of our database group, Yabo Xu, Fiona Choi, Gabriel Fung, Xiaolei Yuan, Jiefeng Cheng, Zheng Liu and Gang Gou. They shared their thinking with me, helped me to solve many problems and gave me warm support.

I am grateful to all my friends anywhere who shared the life with me in these two years. Only in the pleasant time with them, can I get the success in research.

The most appreciation from me should be given to my dearest parents. Their endless love and trust are the powerful source of energy to get through the difficulties and reach the success.

Li Zhiheng

The Chinese University of Hong Kong

June 2004

Contents

Abstract (English) i

Acknowledgement iii

Contents iv

List of Figures vii

List of Tables ix

List of Algorithms x

1 Introduction 1

1.1 Review of frequent itemset mining 1

1.2 Data mining proxy serving for large numbers of users 3

1.3 Privacy issues on proxy service 4

1.4 Organization of the thesis 6

2 Frequent itemsets mining 7

2.1 Preliminaries 7

2.2 Data mining queries 8

2.3 A running example 10

3 Data Mining Proxy 13

3.1 Load data for mining 14

3.2 An Overview 16
3.3 Tree Operations 16
3.4 Data Mining Usages and Observations 18

4 Implementation of Proxy **23**

4.1 Problems in implementation 23
4.2 A Coding Scheme 24
4.3 On-disk/In-Memory Tree Representations and Mining 27
4.4 Tree Operation Implementations 29
4.4.1 Tree Projection Operation Implementations: $\pi_{d2m}()$ and $\pi_{m2m}()$ 31
4.4.2 Tree Merge Operation Implementations: 33
4.4.3 Frequent Itemset/Sub-itemset Tree Building Request . . . 37
4.4.4 The Tree Projection Operation $\hat{\pi}$ and Frequent Super-itemset Tree Building Request 39

5 Performance Studies **45**

5.1 Mining with Different Sizes of Trees in Main Memory 47
5.2 Constructing Trees in Main Memory 48
5.3 Query Patterns and Number of Queries 50
5.4 Testing Sub-itemset Queries with Different Memory Sizes 51
5.5 Replacement Strategies 51

6 Privacy Preserving in Proxy Service **61**

6.1 Data Union Regardless Privacy Preserving 61
6.2 Secure Data Union 65

6.2.1 Secure Multi-party Computation 65
6.2.2 Basic Methods of Privacy Preserving in Semi-honest Environment 67
6.2.3 Privacy Preserving On Data Union 70
6.3 Discussions 73

7 Conclusion **75**

Bibliography **77**

List of Figures

2.1 An Example 10

2.2 *PP*-tree for Example 1 11

3.1 An Overview 17

3.2 Four Tree Projections (Suppose T_1 is the *PP*-tree in Figure 2.2) . 20

4.1 A coded *PP*-tree for Example 1 25

4.2 A PP_D-tree for Figure 2.2 ($\tau_m = 4$) 28

4.3 In-Memory PP_M-trees 41

4.4 Merging an in-memory tree with a projected on-disk tree 42

4.5 In-Memory Tree Merging 42

4.6 An example for $\hat{\pi}$ 43

5.1 Disk tree sizes with different τ_m 47

5.2 Tree sizes vs mining cost (excluding the time for constructing the initial tree) 48

5.3 Constructing initial trees 53

5.4 Various Queries Patterns I (1,000 queries, 60% are mini-support queries, 20% sub-itemset queries (5-10 items), and 20% super-itemset queries (1-5 items) 54

5.5 Various Queries Patterns II (1,000 queries, 60% are mini-support queries, 20% sub-itemset queries (5-10 items), and 20% super-itemset queries (1-5 items) 55

5.6 Various Queries Patterns III (1,000 queries, 60% are mini-support queries, 20% sub-itemset queries (5-10 items), and 20% super-itemset queries (1-5 items) 56

5.7 Sub-itemset Queries I (non-overlapping sliding window of $[R_{min}, R_{max}]$, 1,000 queries, all sub-itemset queries) 57

5.8 Sub-itemset Queries II (fixed R_{min} and enlarging R_{max} of $[R_{min}, R_{max}]$, 1,000 queries, all subitemsets queries) 58

5.9 Sub-itemset Queries III (fixed R_{max} and varying R_{min} of $[R_{min}, R_{max}]$, 1,000 queries, all subitemsets queries) 59

5.10 Replacement Strategies (1,000 queries, all sub-itemset queries) . 60

6.1 Sharing Data 64

6.2 Example of securely calculate average salary 68

6.3 Example of securely sharing data 69

6.4 Example of sharing counts 72

List of Tables

5.1 Dataset Sizes 46

5.2 Code Compression 46

List of Algorithms

1 $\pi_{d2m}(r, p, d, V, k)$ 32

2 $\pi_{m2m}(r, o, V, X)$ 34

3 SubTree-Request(τ, V) 38

4 Share MPIs among n sites ($n \geq 3$) 71

5 Get the count of the nodes on global *PP*-tree's from n sites ($n \geq 3$) 73

Chapter 1

Introduction

Summary

The thesis proposes a novel method to implement a data mining proxy to support a multi-user environment with quantities of queries. The privacy issues are also considered in this proxy.

1.1 Review of frequent itemset mining

Data mining is a powerful technology being widely adopted to help decision makers to focus on the most important nontrivial/predictive information/patterns that can be extracted from large amounts of data they continuously accumulate in their daily business operations. Finding frequent itemsets in a transactional database is a common task for many applications.

In the past decade, many methods have been proposed to find frequent itemsets. The Apriori [3] method utilize the anti-monotone Aprioiri heuristic: if

any k-itemset X is not frequent, any super itemset of X of length $k+1$ is not frequent. Therefore, the Apriori [3] conducts a level-wise procedure: (1) generate candidate itemsets of length k and scan the database to prune the candidates that are not frequent w.r.t. τ, min-support threshold; (2) generate candidate itemsets of length $k+1$ from the frequent k-itemset and return to step 1 if the number of candidate $k+1$-itemsets is not zero. The crucial shortcoming of [3] is that it needs to generate a vast number of candidates when number of items is very large and to scan database several times which leads to a large I/O cost. [2] wants to speed up the procedure by using a matrix to enumerate the support so as to save the time of counting support. [2] also propose a method of deep first traversing the lexicography tree so that the number of the candidates is small at a time. Many other methods are also proposed to improve the efficiency of Apriori. But these methods still can not avoid scanning database repeatedly. To solve this problem, one simplest way is to project the frequent items in each transaction to memory. But even only the frequent part, it may still be very large and is difficult to be held in memory. [22] solve this problem by using the bit set to represent itemset. But it still needs to access disk from time to time. The FP-growth methods [9] utilize the FP-tree structure to avoid accessing the disk again and again. It only needs to read the data on disk twice to build up the FP-tree in memory and then traverse the tree to calculate the support. There are many algorithms using this data structure, including [25, 13, 10, 14, 21, 23, 26]. For example, PP-mine proposed in [25] simplified the tree structure and improved the mining speed by avoiding building sub-tree recursively. Anymore, [16], whose mining procedure is similar to FP-growth, uses H-struct to load the data into memory . There are also many other methods, like [1, 12, 27, 24]. The algorithms of mining close

frequent itemsets and maxium frequent itemsets also use the above methods to find frequent patterns, [23, 26, 17, 4, 8, 5, 1, 12]. The only difference is that they do not need to find all frequent itemsets, so they can use some techniques to minimize the searching space.

1.2 Data mining proxy serving for large numbers of users

All the above methods just focus on the problem of single user and single query problem. In this thesis, we study a data mining proxy in a multi-user environment where a large number of users issue similar but different data mining queries from time to time. This work is motivated by the fact that in a large business enterprise a large number of users need to mine patterns according to their needs. Their needs may change from time and time, and the characteristics of data they have accumulated may change from time to time as well. The need for finding interesting patterns is not in a style of once-for-ever. Therefore, users need to register their data mining queries as a continuous query in a data mining system, and expect to see the results when a transactional database is updated periodically. In fact, the transactional database is highly possible to be updated frequently. Whenever a transactional database is updated, the system needs to run a large number of data mining queries registered. It is undesirable to process these data mining queries one-by-one.

In the thesis, we focus on how to fast respond a user's data mining query by constructing a smallest but sufficient tree using both trees in a data mining proxy that are previously built for other users' requests and trees stored on disk that are

pre-computed for transactional databases. In the thesis, we propose some tree operations on PP-tree both in memory and on disk so as to utilize the pre-built trees. To implement the operations, bitmap code scheme is designed to encode the path information on PP-trees and some fast algorithms are implemented for each operation. The data mining proxy has significant impacts on data mining. First, the I/O cost can be minimized, because trees do not need always to be constructed by loading data from disk. Second, the data mining cost can be reduced, because the mining cost largely depends on the tree size.

1.3 Privacy issues on proxy service

The privacy and security are also very important in the implementation of the proxy. Considering the situation that the data are distributed in different proxies and the user needs to mine patterns from these distributed data, the owners of the data do not want the user to get the patterns which only belong to themselves. Therefore, the privacy preserving policy proposed in this thesis aims that each site only shares a part of their own encrypted data and the result of the decryption will get the whole data the same as the original ones for data mining. Using encryption is to guarantee that the others will not know the real content of the part of shared data. In our method, we utilize the properties of tree structure so as to minimize the requirement for data to be shared, because less data to be shared, less probably the others can guess the useful information from the encrypted data, e.g. the size of the data. But we still guarantee that the mining results are correct. Anymore, the communication cost can largely be cut down.

Up to now, there are also some methods proposed to preserve the privacy in

data mining. [20] preserve the privacy by distorting the data and the user need to try his best to recover the distorted data, but this methods can only find the approximative patterns since the original data can never be recovered. [15] wants to mine the global patterns by sharing the patterns belonging to each site. To protect the privacy, this method let the users define some patterns which need to be preserved and do not share these patterns and some other patterns related. This methods also can not promise to find the whole patterns since some local patterns are hidden by the individual sites. Another disadvantage is that there are very large quantities of patterns, so it is very difficult to let the users to define some patterns to be hidden. [11] propose a method to share the patterns on each site by firstly encrypting the them and then decrypting the patterns after the user get the whole patterns. Thus users will not know which site the specific pattern is belong to. This method need to use the Apriori-like approach where each site first generates local patterns of length k, shares the local k-length patterns to find global frequent ones and then uses them to generate local patterns of length $k+1$. Therefore the communication cost is very large since the number of patterns will explode when the length k increase. Secondly, when the length of patterns k increases, some sites will have no patterns of length k. If there are less than three sites have patterns of length k, this method can not guarantee the privacy. Anymore, this method will leak the size of the overlapping part of the local patterns between two sites. It also have danger of leakage of the length of patterns in each site. The patterns that are only locally frequent will also be revealed. That is to say it is not good to sharing local patterns even when they are encrypted. Comparing to these methods, our method only need the user to share very little data (not local patterns) in privacy preserving and finally get the

correct results of the patterns. Comparing to the other methods, we utilize the tree structure, so that only a little data need to be shared so as to minimize the probability of leakage useful information. Anymore, the I/O cost is also largely cut down. However, we still guarantee the mining results are correct.

1.4 Organization of the thesis

The rest of the thesis is organized as follows. Chapter 2 introduces frequent itemsets mining and tree building requests. Chapter 3 gives an overview of the data mining proxy, followed by the definition of tree operations. Several observations and remarks will be given using examples. Chapter 4 discusses the implementation details of the proxy. We will report our experimental results in Chapter 5. The privacy preserving issues and the solutions will be discussed in Chapter 6. The conclusion of the thesis will be given in Chapter 7.

□ End of chapter.

Chapter 2

Frequent itemsets mining

Summary

In this chapter, some basic terms on association rules mining and frequent itemset mining are introduced. A brief view of *PP*-tree and the definition of three types of data mining queries are presented. An running example is presented to explain these terms.

2.1 Preliminaries

Let $I = \{x_1, x_2, \cdots, x_n\}$ be a set of items. An itemset X is a subset of items I, $X \subseteq I$. A transaction $T_X = (tid, X)$ is a pair, where X is an itemset and tid is its unique identifier. A transaction $T_X = (tid, X)$ is said to contain $T_Y = (tid, Y)$ if and only if $Y \subseteq X$. A transaction database TDB is a set of transactions. The number of transactions in TDB that contains X is called the support of X, denoted as $sup(X)$. An itemset X is a frequent pattern, if and only if $sup(X) \geq \tau$,

where τ is a threshold called a minimum support. The frequent pattern mining problem is to find the complete set of frequent patterns in a given transaction database with respect to a given support threshold, τ.

A prefix-path tree is an order tree. Let F be a set of frequent items (1-itemsets) in a total order ($\preceq$).[1] A node in the tree is labelled for a frequent item in F. The root of the tree represents "null" item. The children of a node are listed following the order. A path of length l from the root to a node in the tree represents a l-itemset. We use a sequence of items to represent a path. The rank of a prefix-path tree is the number of frequent 1-itemsets. In this paper, we adopt a prefix-path tree, called *PP*-tree, reported in our early work [25]. The *PP*-tree is much simpler than FP-tree [9], because it is node-link free and does not need to link all nodes with the same item names when construction. The *PP*-mine algorithm [25], which mines patterns without constructing any conditional trees, outperformed the FP-growth algorithm, which mines patterns by recursively constructing conditional FP-trees.

2.2 Data mining queries

A *data mining query* is to mine frequent patterns from a transaction database TDB. Let τ be a given threshold and V be an itemset. We consider three types of data mining queries below.

- **Frequent Itemset Mining Query**: mining frequent patterns whose support is greater than or equal to τ.

[1]The order can be any order like frequency order, lexicographic order.

- **Frequent super-itemset Mining Query**: mining frequent patterns that include all items in V, and have a support that is greater than or equal to τ. Examples include how to find causes of a certain rule, for example, $* \rightarrow X$, where $*$ indicates any sets.
- **Frequent sub-itemset Mining Query**: mining frequent patterns that are included in V, and have a support that is greater than or equal to τ. Examples include mining rules for a limited set of products, for example, daily products.

All data mining queries can be one of these three types or a combination of them. Each data mining query is processed in two steps: i) constructing an initial *PP*-tree, and ii) mining on top of the *PP*-tree being constructed. We call the former a *frequent tree building request*, which can be done by constructing a tree from either TDB or a previously built tree. The three corresponding frequent tree building requests are given as follows.

- **Frequent Itemset Tree Building Request**: constructing a tree in memory which is smallest and sufficient to mine frequent patterns whose support is greater than or equal to τ.
- **Frequent super-itemset Tree Building Request**: constructing a tree in memory which is smallest and sufficient to mine frequent patterns that include items in V and have a support that greater than or equal to τ. The tree being built can be used to find all possible causes of a certain rule, for example, $* \rightarrow V$, where $*$ indicates any sets.
- **Frequent sub-itemset Tree Building Request**: constructing a tree in memory which is smallest and sufficient to mine frequent patterns that are

included in V and have a support that is greater than or equal to τ. The tree being built can be used to find all possible patterns for a limited set of itemsets, $V_1 \rightarrow V_2$ where $V_1 \cup V_2 = V$ and $V_1 \cap V_2 = \emptyset$. An example is to mine all frequent patterns among daily products in a supermarket.

2.3 A running example

In this paper, we concentrate ourselves on the efficient tree building requests in a data mining proxy. A running example is given below.

Example 1 Let the first two columns of the table in Figure 2.1 be our running transaction database TDB. The frequent items are shown in the third column where the minimum support threshold is $\tau = 4$.

TID	Items	Freq items
100	c,d,e,f,g,i	c,d,e,f,g
200	a,c,d,e,f,m	a,c,d,e,f
300	a,b,d,g,k	a,b,d,g
400	a,c,h	a,c
500	a,c,e,f,g	a,c,e,f,g
600	a,b,e,n	a,b,e
700	a,b,c,d,m	a,b,c,d
800	a,e,k	a,e
900	a,b,d,g,h	a,b,d,g
1000	c,e,f,m	c,e,f
1100	c,d,e,g,h,j	c,d,e,g

Figure 2.1: An Example

A *PP*-tree example is shown in Figure 2.2 . The rank of this *PP*-tree is 7, because 7 frequent 1-itemsets, *a*, *c*, *e*, *d*, *g*, *b* and *f*, are presented in frequency

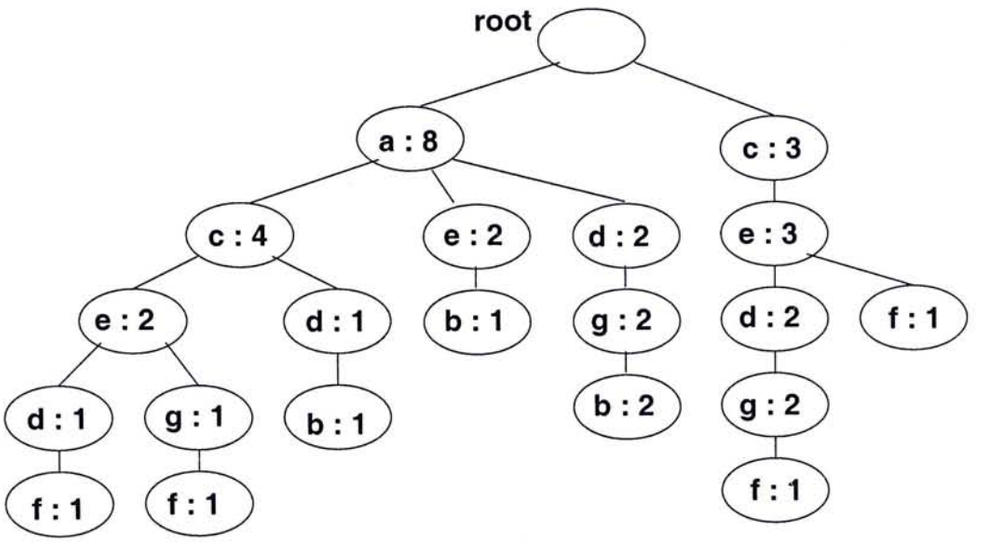


Figure 2.2: *PP*-tree for Example 1

order. Their corresponding supports are, 8, 7, 7, 6, 5, 4 and 4, respectively, which are greater than or equal to the minimum support ($\tau = 4$). The root is for "null" item. Each non-root node, v, has a pair $i{:}s$ shown inside the node, where i and s represent an item and a support. The support is the support for an itemset represented by the items along the path from the root to the node v. For example, an itemset of $\{a,\ c,\ e\}$ is represented by a path ace with a support of 2.

□ **End of chapter.**

Chapter 3

Data Mining Proxy

Summary

In this chapter, we will have an overview of the data mining proxy. Three tree operations are introduced here. We will explain how the data mining proxy works with these tree operations.

The data mining proxy is designed and developed to support a large number of users with similar data mining queries by fast responding a user's tree building request. The efficiency of tree building requests is achieved by utilizing both trees in the data mining proxy that are previously built for other users' requests and trees stored on disk that is pre-computed for transactional databases. In addition to efficiency issues, the effectiveness of data mining proxy is achieved by responding a smallest and sufficient tree for a users data mining query. It is because the cost of mining all possible patterns in a tree heavily depends on the tree size. The smaller the tree is, the less mining cost it occurs.

3.1 Load data for mining

In a transaction database, not all data in it are required for a data mining query. According to the Apriori property, which says if any itemset I is frequent, any sub-itemset of I must be frequent, any item i in a frequent itemset I, w.r.t. min-sup τ specified by a user, must be frequent. Therefore, in each transaction, only the frequent items will contribute to the support of the frequent itemsets. If the user submit a *frequent sub-itemset data mining query*, only the items that are frequent, w.r.t. τ, and in the user specified sub-itemset will contribute to this query. If the user submit a *frequent supper-itemset data mining query*, only the items that are frequent, w.r.t. τ, and in the transactions that contains all the items in the user specified supper-itemset will contribute to this query. Thus, for each transaction, only those items contributing to the query are required to be projected into memory. For a user specified query q, we denote $\pi_q(T)$ as the projection part of the transactions in T for the query q, where T is a TDB. If the user submit a *frequent itemset data mining query*, the projection can be denoted as $\pi_\tau(T)$, where τ is user-specified min-support. If the user submit a *frequent sub-itemset data mining query* (*frequent sub-itemset data mining query* resp.) it can be denoted as $\pi_{\tau,V}(Trans(T))$ ($\hat{\pi}_{\tau,V}(Trans(T))$ resp.), where V is the sub-itemset (super-itemset resp.) specified by the user. For example, in Figure 2.1, let T be the TDB in Example 1, the content of column two is the transactions in T and the content of column three is $\pi_4(T)$, where $\tau = 4$. For any transaction t, the projected part of t for a query q is denoted as $\pi_q(t)$. In Example 1, the projection of transaction 200 for the query with $\tau = 4$ is a, c, d, e, f.

Because the prefix path tree is a good compress form to store itemsets in

memory, the projection of a TDB, which is also a set of itemsets, can be stored in memory in form of a prefix path tree. Anymore, if we need to use the other algorithms, the prefix path tree containing all the necessary information can be transferred into other data structure.

For a new query q_{new}, an old query q_{old} and a transaction t in TDB T, $\pi_{q_{new}}(T)\hat{\cap}\pi_{q_{new}}(T) = \{s|s$ is an itemset and $s = \pi_{q_{new}}(t) \cap \pi_{q_{new}}(t)$ for any transaction t in T $\}$. If $\pi_{q_{new}}(T)\hat{\cap}\pi_{q_{new}}(T) \neq \Phi$, it is to say that we can project $\pi_{q_{new}}(T)\hat{\cap}\pi_{q_{new}}(T)$ from the tree for q_{old} to build up a tree for q_{new} and avoid loading this part from disk. Therefore, for a query q, the projection $\pi_q(T)$ can be the merge of the projections from TDB and the trees built for the previous queries.

But even if we only load the projection part from TDB, it also needs us to check every transaction in it, which means we need to scan the whole TDB, so it can not improve efficiency. We consider to reorganize the TDB into a tree structure so that we can only need to access a part of the tree to get the projection and have no need to scan the rest part. The method of materialize a tree on disk will be introduced in the next chapter.

In the implementation, we choose *PP*-tree structure proposed in Xu02 [25]. This is because *PP*-tree is the simplest prefix path tree structure since it only need the links from parent to its children and no any other additional links while others, like FP-tree, need to link up the nodes with the same item name. The experiments also show that *PP*-mine performs better than other algorithms in many cases. Finally and most importantly, we implement a method to materialize *PP*-tree on disk.

3.2 An Overview

An overview of the data mining proxy is shown in Figure 3.1. Here, we assume that a PP_i-tree exists for a transaction database, TDB_i, on disk, whose minimum support, τ_{m_i}, is small enough to support all user requests. All frequent patterns in TDB_i that are greater than τ_{m_i} in TDB_i are materialized in PP_i-tree. As shown later in this paper, such a PP_i-tree is considerably small. Several small subtrees of PP_i-tree can co-exist in data mining proxy. Using the operations introduced later, these co-existing trees in proxy and on disk can be utilized to construct a new tree for a new data mining query efficiently.

3.3 Tree Operations

In this section, we will propose some operations on the PP_M-tree and PP_D-tree to achieve the function of the proxy.

Let TDB be a transaction database that includes items in $I = \{x_1, x_2, \cdots, x_n\}$, where x_i is a 1-itemset.

Definition 3.1 : *Given a minimum support τ and a set of items $V \subseteq I$. Let T, T_1 and T_2 be PP-trees. Three tree operations are defined.*

- *A sub-projection operation, denoted $\pi_{\tau,V}(T)$, is to project a subtree from the given tree T. The resulting tree includes all itemsets in V whose minimum support is greater than or equal to τ.*
- *A super-projection operation, denoted $\hat{\pi}_{\tau,V}(T)$, is to project a subtree from the given tree T. The resulting tree includes all itemsets which are a super set of V and have a minimum support that is greater than or equal to τ.*

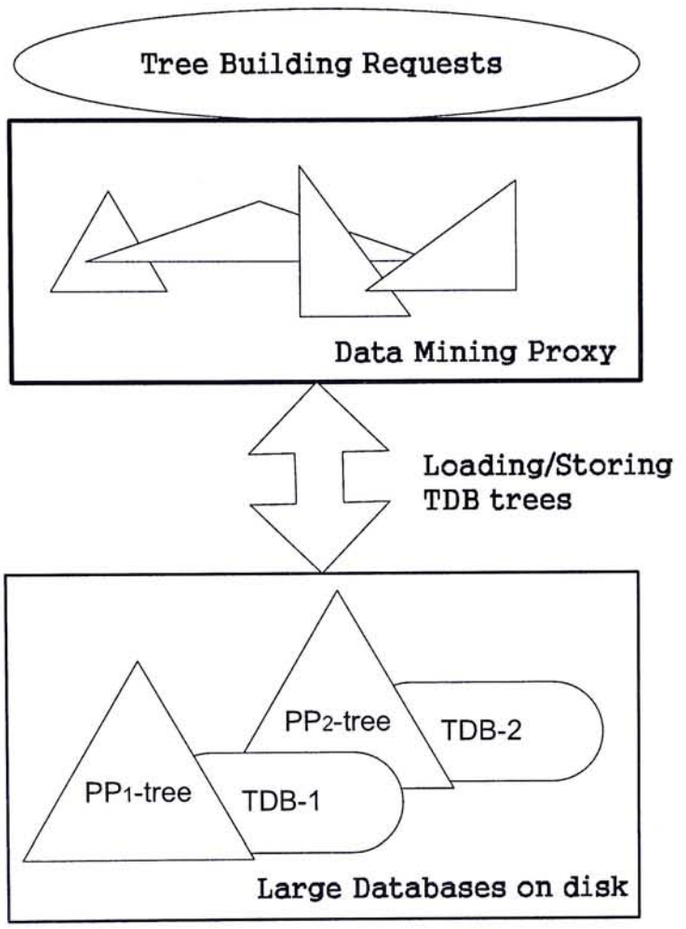


Figure 3.1: An Overview

- *A merge operation, denoted $T_1 \oplus T_2$, is to merge two trees, T_1 and T_2, and results a new tree.*

Given two *PP*-trees, T_i and T_j, we say $T_i \subseteq T_j$ if T_i is a subtree of T_j. More precisely, by $T_i \subseteq T_j$ we mean that every itemset, X, represented in T_i is also in T_j. As examples, if $T_i = \pi_{\tau,V}(T_j)$ then $T_i \subseteq T_j$, and if $T_k = T_i \oplus T_j$, then $T_i \subseteq T_k$ and $T_j \subseteq T_k$. Let T_I be the largest *PP*-tree that include every single $x_i \in I$ appearing in TDB. Obviously, any tree is a subtree of T_I. Consider a data mining query, q, and a mining algorithm, M. We denote the resulting frequent

patterns for q as $M_q(\cdot)$. For two *PP*-trees, T_i and T_j, we say $T_i \equiv_q T_j$ if $M_q(T_i)$ is the same as $M_q(T_j)$. In other words, the two trees, T_i and T_j, will give the same frequent patterns for the same query q. The smallest tree for a data mining query is defined below.

Definition 3.2 : *For a data mining query q with a minimum support τ and a set of items V, the smallest tree, T ($\subseteq T_I$), is a tree that satisfies the condition of $T \equiv_q T_I$, but does not satisfy $T \equiv_q T_I$ if any item is removed from T.*

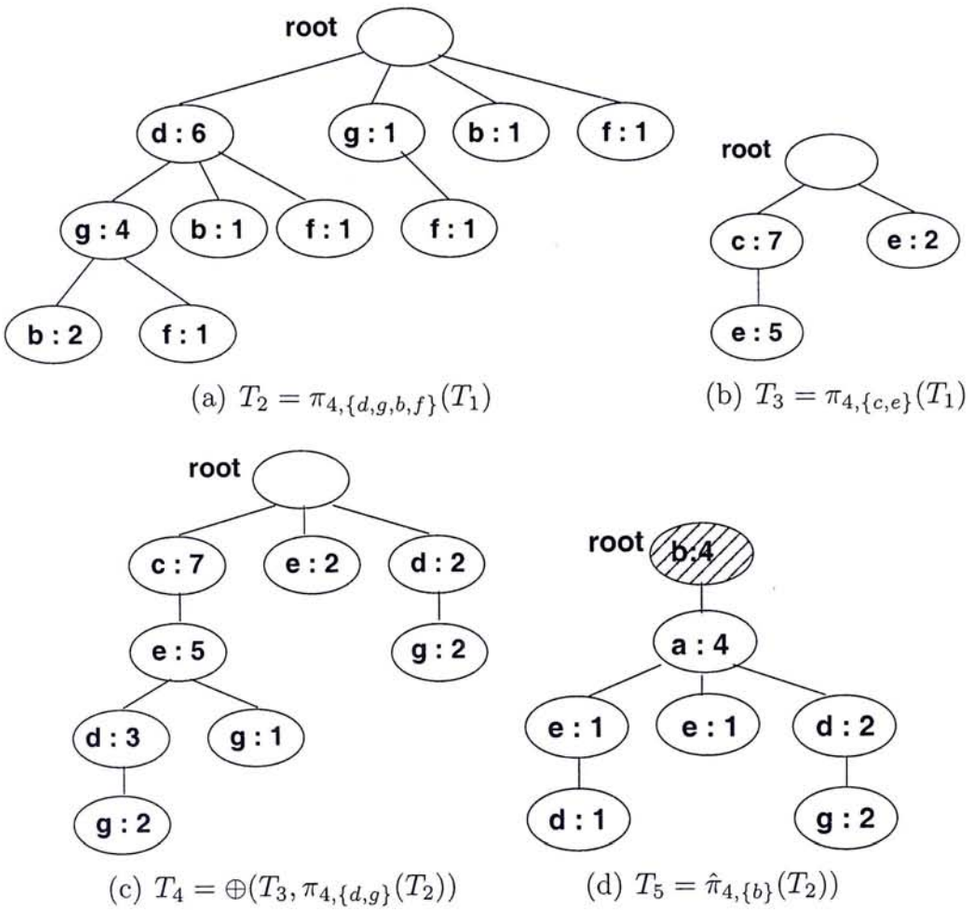
3.4 Data Mining Usages and Observations

A simple scenario using the tree operations to show the proxy functions is given below. Here, consider Figure 2.2, which represents a *PP*-tree, T_1, on disk for the transaction database in Figure 2.1 ($\tau = 4$). Assume the memory space in the proxy is limited, and the data mining proxy is initially empty. First, suppose the first data mining query is a frequent sub-itemset mining query with $\tau = 4$ and $V = \{d,\ g,\ b,\ f\}$. The corresponding frequent sub-itemset tree building request is issued to the data mining proxy with the same τ and V. The data mining proxy will process it as $T_2 = \pi_{4,\{d,g,b,f\}}(T_1)$ by loading a subtree from disk and constructing T_2 in the data mining proxy (Figure 3.2 (a)). Second, suppose another corresponding tree building request comes and can be processed as $T_3 = \pi_{4,\{c,e\}}(T_1)$. The data mining proxy loads a subtree from disk, and constructs the tree in memory (Figure 3.2 (b)), because the requesting items do not exist in any tree in memory. Third, suppose that a user issues another frequent sub-itemset tree building request with $\tau = 4$ and $V = \{c,\ e,\ d,\ g\}$. At this stage, there are two trees, T_2 and T_3, in the proxy. T_2 contains two

requested items, c and e, whereas T_3 contains the other two requested items, d and g. The requested tree can be constructed by merging T_2 and a subtree of T_3. The data mining proxy process the request as $T_4 = \oplus(T_3, \pi_{4,\{d,g\}}(T_2))$ (Figure 3.2 (c)), which does not involve any I/O costs. Fourth, assume that a frequent super-itemset tree building request comes with $\tau = 4$ and $V = \{b\}$. The data mining proxy processes the request as $T_5 = \hat{\pi}_{4,\{b\}}(T_2)$, which requests no I/O costs. The constructed tree, T_5, is shown in Figure 3.2 (d). The root node in Figure 3.2 (d) represents $X{:}s$ where X represents the itemset that is included in every itemset in the tree and s is its support. Next, suppose a user issues a frequent sub-itemset tree building request with $\tau = 4$ and $\{a\}$. Although T_5 contains a, it is incorrect to process the request as $\pi_{4,\{a\}}(T_5)$, because the support for a shall be 8, where the support for a in T_5 is 4. That is $\pi_{4,\{a\}}(T_5) \not\equiv \pi_{4,\{a\}}(T_1)$. It needs to be processed as $\pi_{4,\{a\}}(T_1)$.

Remarks *It is important to know that, for a given data mining query q, a sequence of tree operations can be easily identified that results in a smallest PP-tree for q. All the resulting trees shown in Figure 3.2 are the smallest trees to respond the corresponding data mining query.*

Three observations can be made below. First, the trees being built for frequent sub-itemset tree building requests can be possibly reused to support other frequent tree building requests. That is, given two trees T_i and T_j, where T_i (T_j) is a tree being built for a frequent sub-itemset mining query (with V_i (V_j) and τ_i (τ_j)). When a new frequent sub-itemset mining query (with τ_k and V_k) comes, a corresponding tree T_k can be built using T_i and T_j if $V_k \subseteq V_i \cup V_j$. An example is $T_4 = \oplus(T_3, \pi_{4,\{d,g\}}(T_2))$ (Figure 3.2 (c)). Second, frequent itemset tree building requests are a special case of the frequent sub-itemset tree



(a) $T_2 = \pi_{4,\{d,g,b,f\}}(T_1)$ (b) $T_3 = \pi_{4,\{c,e\}}(T_1)$

(c) $T_4 = \oplus(T_3, \pi_{4,\{d,g\}}(T_2))$ (d) $T_5 = \hat{\pi}_{4,\{b\}}(T_2))$

Figure 3.2: Four Tree Projections (Suppose T_1 is the *PP*-tree in Figure 2.2)

building requests, where V can be considered as to include all possible frequent 1-itemsets. It possesses the same property as the frequent sub-itemset tree building requests. Third, a tree being built for responding frequent super-itemset tree building request, T_l (with V_l and τ_l), is difficult to be reused, because itemsets $X - V_l$ represented in T_l is partial. There exist occurrences of items in $X - V_l$ that do not exist in T_l. T_l does not have a closure. As an example, there are four a's that co-exist with b as for the frequent super-itemset tree building request $V_l = \{a, b\}$ (Figure 3.2 (d)). But there are other four a's which do not co-exist with b (see Figure 2.1).

□ **End of chapter.**

Chapter 4

Implementation of Proxy

Summary

In this chapter, we discuss a coding scheme which plays a very important role in the implementations of tree operations. We will discuss in-memory and on-disk representations of *PP*-tree , and will illustrate how to implement tree operations so that the advantages of the tree structure, in memory and on disk, and the coding scheme can be fully utilized.

4.1 Problems in implementation

To implement the data mining proxy, there are some problems need to be solved. The first is how to determine the ancestor-descendent relationship between two nodes. For example, in Figure 3.2, when we project the node "$d : 6$" from tree in sub-figure (a) to the tree in sub-figure (c), "$d : 3$" is divided into three nodes:

"d : 3" as the child of "e : 5", "d : 1" as the child of "c : 7" and "d : 2" as the child of root node. Therefore, when doing projection, we need a method to determine how to divide the node and then insert the divisions to be a child of a appropriate ancestor in the new tree. In this chapter, we will introduce a bitmap code scheme to solve this problem.

The second problem is concerning the efficiency of the operation. Since the operations are to be done on trees, it will make the time complexity exploded if we recursively visit the tree to find next projection part, especially when the tree is stored on disk. In this chapter, we will try to find the methods to traverse the tree to be projected from, either on disk or in memory, one and only once to get all data need to be projected. It needs the bitmap codes to be well organized in the trees on disk and in memory. We also need to skip some parts in the trees if we can determine there is no data to be projected in these parts.

4.2 A Coding Scheme

Definition 4.1 Bitmap-Code: *Given a PP-tree of rank l for a set of frequent 1-itemsets in a total order. A bitmap-code of length l-bits is assigned to a node v in PP-tree, denoted code(v). The i-th bit in code(v) indicates whether the i-th frequent 1-itemset is included in the itemsets represented by the path from the root to the node v. The code for a node is unique such that code(v) $\neq$ code(u) if $v \neq u$ in a PP-tree.*

Consider the *PP*-tree of rank 7 shown in Figure 2.2 for the 7 frequent 1-itemsets, a, c, e, d, g, b and f, in a total order. Here, a is the first 1-itemset, and f is the 7th 1-itemset. The corresponding coded *PP*-tree is given in Figure 4.1.

Here, $code(v)$:s is given inside the node v, where $code(v)$ is a bitmap-code and s is the support of the itemsets represented by $code(v)$. For example, the leftmost leaf node v is coded as *1111001* which represents the itemsets *acedf* along the leftmost path of the coded *PP*-tree. The support of this itemset is 1.

Given two nodes v and u, and let $c_v = code(v)$ and $c_u = code(u)$. We define an operator $ancestor(c_v, c_u)$ which returns true if the node v is an ancestor of u, otherwise false. This function is implemented as follows. Let $lastOne(c_v)$ be a function that returns the bit position of the last 1-bit from the left, and $firstK(c_v, k)$ be a function that returns the first k bits from the left of c_v. The function $ancestor(c_v, c_u)$ returns true if $lastOne(c_v) < lastOne(c_u)$ and $firstK(c_v, lastOne(c_v)) \oplus firstK(c_u, lastOne(c_v)) = 0$, where $\oplus$ is an XOR bit-operator. Consider the coded *PP*-tree in Figure 2.2 where the codes for *ac*, *ae*, and *acegf* are *1100000*, *1010000* and *1110101*, respectively. Here, *ac* is not an ancestor of *ae* because $firstK(1100000, 2) \oplus firstK(1010000, 2) = 11 \oplus 10 \neq 0$, and *ac* is an ancestor of *acegf* because $firstK(1100000, 2) \oplus firstK(11110101, 2) =$

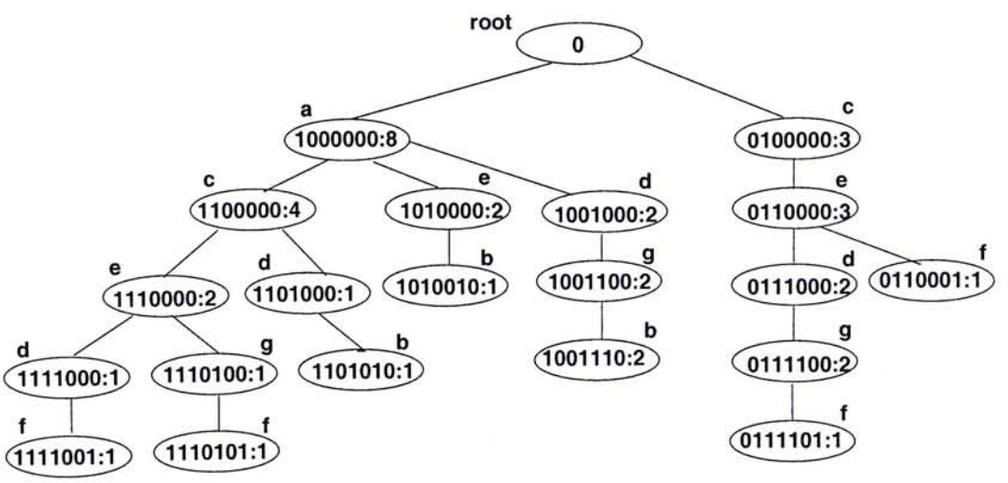


Figure 4.1: A coded *PP*-tree for Example 1

$11 \oplus 11 = 0$. We all defined a function $left(c_v, c_u)$ which returns true if the integer representation of the bitmap-code c_v is less than that of c_u.

The bitmap coding scheme we used in this paper is different the coding scheme we proposed in our early work [25]. In [25], each node is encoded with a number (of the *pre-order* of traversal of the complete *PP*-tree). We call it non-bitmap coding scheme. Given a rank l of a *PP*-tree (l frequent 1-itemsets are supported in the tree), both coding schema need l bits. The advantages of bitmap coding scheme are given below. First, the pre-order ($\preceq$) used in the non-bitmap coding scheme is also maintained in the bitmap coding scheme. Let c_v and c_u be two bitmap codes, then $c_v \preceq c_u$ when $ancestor(c_v, c_u)$ is true or otherwise $left(c_u, c_v)$ is true. We use the pre-order to store a *PP*-tree on disk, and use the pre-order to efficiently load a subtree from the *PP*-tree on disk. Second, the bitmap coding scheme can be effectively compressed since most bits in a code are 0. We conducted some testing. Our result shows that the compression ratio for bitmap coding scheme is very high, especially for sparse transaction databases. Third, we only need XOR bit-operator, $\oplus$, to identify the position of a node in the tree. The non-bitmap coding scheme needs to use a formula to calculate the position recursively, which is rather extensive.

In order to further reduce the space for maintaining the bitmap codes in memory, we only keep one copy of a bitmap code in memory when it appears in many different positions in many *PP*-trees. A code recorded in a node of a *PP*-tree is replaced with a pointer pointing to the bitmap. A bitmap-code will be removed when it is not used in any *PP*-trees in memory with assistance of a counter attached to the bitmap code.

4.3 On-disk/In-Memory Tree Representations and Mining

A *PP*-tree has an on-disk representation, denoted PP_D-tree. A PP_D-tree is represented as (T, F, I, τ_m). Here, T is a heap for the tree structure in which an element consists of a bitmap-code and its count. F stores N (the rank of the *PP*-tree) frequent 1-itemsets with their counts in frequency order. I is an index indicating the ranges of bitmap-codes in disk-pages. τ_m is the minimum support used to build PP_D-tree on disk. This PP_D-tree can be used for mining frequent itemsets with a minimum $\tau \geq \tau_m$. The codes in T are organized following the pre-order ($\preceq$) among bitmap-codes. In the following, we use $order(i)$ to indicate the position of an item i in F, and use $item(i)$ to indicate the item name of the i-th item in F such that $i = item(order(i))$ for $i \in F$.

The on-disk PP_D-tree for the *PP*-tree (Figure 2.2) is shown in Figure 4.2. In Figure 4.2, F stores the 7 frequent 1-itemsets whose supports are greater than or equal to $\tau_m = 4$. For example, $order(d) = 4$, because it is the fourth in F. The on-disk tree is stored on 7 consecutive disk-pages in the heap T. I maintains a pair of bitmap-codes as the boundary for each disk-page in T.

Remarks *Given two nodes v and u. Let $c_v = code(v)$ and $c_u = code(u)$, and let $d_v = addr(v)$ and $d_u = addr(u)$ where $addr(v)$ indicates the location of node v in one of the consecutive disk-pages. The pre-order $\preceq$ implies the order of nodes on disk. That is, $c_v \preceq c_u$ implies $d_v < d_u$. Furthermore, it means that an ancestor of a node v can only exist before v, and a child of v can only exist after v. This property is fully utilized in order to load a subtree from disk into memory.*

It needs to be pointed out that because there are no pointers in the on-disk representation and the codes can be compressed, the disk space required is not large.

F : TABLE

a	c	e	d	g	b	f
8	7	7	6	5	4	4

I : INDEX

1	1000000:1110000
2	1111000:1110100
3	1110101:1101010
4	1010000:1001000
5	1001100:0100000
6	0110000:0111100
7	0111101:0110001

T

PAGE 1		PAGE 2		PAGE 3	
01	1000000:8	04	1111000:1	07	1110101:1
02	1100000:4	05	1111001:1	08	1101000:1
03	1110000:2	06	1110100:1	09	1101010:1
PAGE 4		**PAGE 5**		**PAGE 6**	
10	1010000:2	13	1001100:2	16	0110000:3
11	1010010:1	14	1001110:2	17	0111000:2
12	1001000:2	15	0100000:3	18	0111100:2
PAGE 7					
19	0111101:1				
20	0110001:1				

Figure 4.2: A PP_D-tree for Figure 2.2 ($\tau_m = 4$)

A prefix-path tree has an in-memory representation. In [25], we proposed an in-memory tree, denoted PP_M-tree. Like FP-tree, despite the pointers to the children nodes, a node in PP_M-tree includes a pair of item-name and count and a node-link. The count registers the number of transactions represented by the path reaching from the root to this node. Unlike FP-tree, PP_M-tree is node-link free. That is, there are no other links to link all node with the same item names. Figure 2.2 shows a PP_M-tree. Also, in [25], we proposed a reconstructive mining algorithm, called *PP*-Mine, that mines frequent patterns by reconstructing PP_M-tree without constructing any conditional tree. We showed that *PP*-Mine outperformed FP-growth [9]. In this paper, in order to support tree projection

and tree merging, we modify the PP_M-tree in [25] by adding a pointer to a list of pairs of bitmap-code and count. Because *PP*-Mine only needs to mine patterns using the pair of item-name and count but not the bitmap-codes, which are only needed to construct a tree, the mining performance is not affected. In the following, we use PP_M to refer to the modified in-memory tree in this paper.

Figure 4.3 shows three in-memory trees. Figure 4.3 (a) shows the same PP_M-tree (T_1) as shown in Figure 2.2. Figure 4.3 (b) shows the image of a PP_M-tree $T_2 = \pi_{4,\{d,g,b,f\}}(T_1)$ for mining. The shaded nodes in T_1 (Figure 4.3 (a)) are projected for constructing T_2 (Figure 4.3 (b)). Note that the leftmost subtree of T_2 is constructed by merging four paths as shown in the four dotted ovals in Figure 4.3 (a). Figure 4.3 (c) shows the internal image of T_2 using bitmap-codes, which maintains all pieces of the path information for further tree merging and projection.

4.4 Tree Operation Implementations

In this section, we will propose some algorithms to construct PP_M-tree for tree building request using the operations introduced above.

If the request belongs to the type of *Frequent Itemset Tree Building Request*, it can be written in the form that $T_0 = \pi_{\tau,V_0}(P)$, where T_0 is the result PP_M-tree, τ is the min-support threshold, V_0 is ϕ and P represents the proxy that contains a set of *PP*-trees. Suppose in P the PP_D-tree's min-support is larger or equal to τ and the set of items in PP_D-tree is I, T_0 can be expressed as the form $T_0 = \pi_V(P)$, where $V \subseteq I$ and support of all the items in V is larger than or equal to τ. Because T_0 can be constructed from many trees in P, in memory or

on disk, we can to write $T_0 = \pi_{V_1}(T_1) \oplus \pi_{V_2}(T_2) \oplus \pi_{V_3}(T_3) \oplus ... \oplus \pi_{V_n}(T_n)$, where T_is are *PP*-trees in P, $\bigcup_{i=1}^{n} V_i = V$ and $V_i \cap V_j = \phi$, for any $i \neq j, i, j = 1, 2, 3, ..., n$. In this equation, all min-support are omitted because if T_i contains the items in V_i and $V_i \subseteq V$, all the items in V_i satisfy the min-support threshold τ since V only contains the items whose support is larger than or equal to τ.

Remarks *Because τ has no use in the procedure of building trees when the items not satisfying this threshold τ have been pruned, we do not need to write τ in the expressions for tree-building requests.*

If the request belongs to the type of *Frequent Sub-itemset Tree Building Request*, similarly, the returned PP_M-tree T_0 can be expressed as $T_0 = \pi_{\tau,V_0}(P) = \pi_V(P) = \pi_{V_1}(T_1) \oplus \pi_{V_2}(T_2) \oplus \pi_{V_3}(T_3) \oplus ... \oplus \pi_{V_n}(T_n)$, where P is the proxy contains of a lot of *PP*-trees in memory or on disk, τ is min-support threshold, V_0 is the user specified sub-itemset, V is a subset of V_0 and only contains the items whose support is larger or equal to τ, T_i is *PP*-tree in P, $\bigcup_{i=1}^{n} V_i = V$ and $V_i \cap V_j = \phi$, for any $i \neq j, i, j = 1, 2, 3, ..., n$.

If the request belongs to the type of *Frequent Super-itemset Tree Building Request*, it can be expressed as $T_0 = \hat{\pi}_{\tau,\hat{V}}(P)$, where T_0 is the result PP_M-tree, τ is the min-support threshold, $\hat{V}$ is user specified super-itemset and P represents the proxy that consists of a set of trees. Let $\hat{I}_n$ be the item with the largest order in $\hat{V}$. The expression can be changed into the form $T_0 = \hat{\pi}_{V,\hat{V}}(P)$, where V is the union of $\hat{I}_n$ and the items whose order is larger than $\hat{I}_n$ and support is larger than or equal to τ. Because the nodes in the result tree for the items whose order is less than $\hat{I}_n$ can be constructed using the codes of the codes of $\hat{I}_n$. The method will be introduced later. Therefore, T_0 also can be write as the merge of projections of some trees: $T_0 = \hat{\pi}_{V_1,\hat{V}}(T_1) \oplus \hat{\pi}_{V_2,\hat{V}}(T_2) \oplus \hat{\pi}_{V_3,\hat{V}}(T_3) \oplus ... \oplus \hat{\pi}_{V_n,\hat{V}}(T_n)$, where T_is

are PP-trees in P, $\bigcup_{i=1}^{n} V_i = V$ and $V_i \cap V_j = \phi$, for any $i \neq j, i, j = 1, 2, 3, ..., n$.

The implementations of the operations π, $\hat{\pi}$ and $\oplus$ are given in the following sections.

4.4.1 Tree Projection Operation Implementations: $\pi_{d2m}()$ and $\pi_{m2m}()$

In this section, we address implementations of sub-projection operation (π). Two basic algorithms are given. They are tree projection from disk to memory (π_{d2m}) and tree projection from memory to memory (π_{m2m}). Both are implemented as to be able to project a tree, and merge the projected tree with a given tree in memory. When the given tree in memory is empty, both $\pi_{d2m}()$ and $\pi_{m2m}()$ act as tree projection operation. The reason we adopt this implementation is that it makes easier for us to implement the tree merge operation, $T' \leftarrow T_1 \oplus T_2$, d it can be implemented using two tree projection operations as follows. First, let T' be empty. Second, project T_1 to T'. Third, project T_2 to T'.

Given a frequent sub-itemset tree building request with τ and V_0, where a set of items V, $V \subseteq V_0$, needs to be projected from a PP_D-tree using project operation, the implementation of π_{d2m} for loading a subtree consisting of items in V from a PP_D-tree on disk to an empty PP_M-tree in memory is illustrated in Algorithm 1. Five parameters will be passed in $\pi_{d2m}(r, p, d, V, k)$. Here, the first two parameters, r and p specify tree traversal on PP_M-tree. The third parameter d specifies tree traversal on PP_D-tree. V is the set of items to be projected from PP_D-tree. The details of these parameters are described in Algorithm 1.

Initially, we call $\pi_{d2m}(r, 0, 0, V, 0)$, where r points to the root node of the

PP_M-tree to be built, and the third parameter, 0, points to the first record in the head T of the PP_D-tree. As an example, Let T_1 denote the PP_D-tree shown in Figure 4.2, indicated. The result of the projection $\pi_V(T_1)$, $V_\tau = \{d, g, b, f\}$, is shown in Figure 4.3 (c), as the result of $\pi_{d2m}(r, 0, 0, \{d, g, b, f\}, 0)$.

Algorithm 1 $\pi_{d2m}(r, p, d, V, k)$

Input: the current position in a PP_M-tree (r), the bitmap-code of r (p), the order of the item represented by r in the frequent 1-itemsets maintained in F of the PP_D-tree (k), the requested sub-itemset (V, which is sorted by the ascending order of $order(i)$ for $i \in V$), which is sorted by the ascending order of the items), and the current reading position on disk (d, which is passed as call-by-reference).
Output: a PP_M-tree.

```
1: let l be the largest order(j) for any j ∈ V;
2: for i from k + 1 to l do
3:    c_i ← p, and set the i-th bit in the bitmap-code c_i to be 1;
4:    if item(i) ∈ V then
5:       if c_i is not in the current disk page then
6:          d ← readPage(c_i); {using the index I in the PP_D-tree}
7:       end if
8:       Search c_i in the page by increasing d in the current disk page;
9:       if c_i exists then
10:         if there exists a node v_i of item(lastOne(c_i)) as a child of r then
11:            add c_i to v_i's code list and increase v_i's support;
12:         else
13:            create a new node of v_i as r's child with code c_i and its support;
14:         end if
15:         π_d2m(v_i, c_i, d, V, i);
16:      end if
17:   else
18:      π_d2m(r, c_i, d, V, i);
19:   end if
20: end for
```

Given frequent sub-itemset tree building request with τ and V_0, where a set of items V, $V \subseteq V_0$, need to be loaded from a PP_M-tree, the implementation

of π_{m2m} for projecting a PP_M-tree to a PP_M-tree is illustrated in Algorithm 2. Four parameters will be passed in $\pi_{m2m}(r, o, V, X)$. Here, the first parameter, r, specifies the root of the tree to be projected to, whereas the second parameter, o, specifies the tree traversal on the tree to be projected from. The third parameter V is a subset of V containing the items whose support is larger than or equal to τ and need to be loaded from the tree rooted at r. X is an itemset containing the items in the tree rooted at r and satisfying that for any item i_x in X, $order(i_x) \leq order(item(o))$. The details of these parameters are described in Algorithm 2. We will explain it when we discuss tree merging in next section.

4.4.2 Tree Merge Operation Implementations:

The tree merge operation can be implemented at the same time when doing projection operations. In this section, we use examples to illustrate how to implement tree merge operations.

The tree merge operation is in the form of $T_1 = T_2 \oplus T_3$, where T_2 and T_3 can be either a PP-tree in memory or on disk. But in practice we often need to merge the projection of one tree with that of another, which is $T_1 = \pi_{\tau,V_1}(T_2) \oplus \pi_{\tau,V_2}(T_3)$. In order to make the procedure of building a tree more efficient,we first let T_1 store the result of projection $\pi_{\tau,V_1}(T_2)$, then do projection and merge operation $\pi_{\tau,V_2}(T_3) \oplus T_1$ using $\pi_{d2m}()$ and/or $\pi_{m2m}()$. If T_2 is a PP_M-tree, $\pi_{m2m}()$ can directly be used to implement the projection from T_2 to T_1 which has stored the result of $\pi_{\tau,V_1}(T_2)$. If T_2 is the tree on the disk, we use two different methods for *Frequent Itemset Tree Building Request* and *Frequent Sub-itemset Tree Building Request* respectively. For the previous one, we modified $\pi_{d2m}()$ a bit so as to do projection and merge at one time. For the later one, we first do projection and

Algorithm 2 $\pi_{m2m}(r, o, V, X)$

Input: the root of a PP_M-tree, T_t, to be projected to (r), the current position in a PP_M-tree, T_f, to be projected from (o), the requested sub-itemset (V), and items have have already been loaded in T_t (X). Note V (X) is sorted by the ascending order of $order(i)$ for i in V (X).
Output: a PP_M-tree.

```
1: if item(o) ∈ V then
2:   for every set of bitmap-codes of o, b_i, that share the same prefix path do
3:     identify a position node for b_i in the tree rooted at r, denoted v, by
       following the prefix path from r;
4:     if v has a child, u, having the same item name as represented by o then
5:       add b_i to the bitmap-code list of u and increase u's support;
6:     else
7:       create a new node u for b_i as a child of v;
8:     end if
9:   end for
10: end if
11: let l be the largest order(i) for i ∈ V;
12: for each child node n_i of o do
13:   if order(item(n_i)) ≤ l then
14:     if item(o) ∈ V then
15:       π_m2m(r, n_i, V, X ∪ item(o));
16:     else
17:       π_m2m(r, n_i, V, X);
18:     end if
19:   end if
20: end for
```

save the result as a temporary tree in memory using $\pi_{d2m}()$, and then merge this tree with T_1 using $\pi_{m2m}()$. We explain the detail of implementation using some examples below.

First, we consider how to merge an in-memory tree with a projected on-disk subtree for a *Frequent Itemset Tree Building Request*. The strategy of tree merging is to traverse the in-memory tree and extending the in-memory tree by loading the unloaded items from the on-disk tree. As shown below, it can be

efficiently processed because every disk-page of the on-disk tree will be accessed at most once, and can be done using sequential I/Os. An example is shown in Figure 4.4 for a frequent sub-itemset tree building request $\tau = 6$ and $V = \emptyset$. Let T_d be the on-disk tree as shown in Figure 4.2 where $\tau_m = 4$. In practice, we need to load a subtree from T_d with items $\{a, c, e, d\}$ because their supports are greater or equal to $\tau = 6$. Here, assume there is an in-memory tree, T_m, as shown in Figure 4.4 with solid circles that include two items, $I_m = \{a, c\}$. Therefore, we only need to load a subtree from T_d with $I_d = \{e, d\}$. Traversing from the root of T_m, we first visit its leftmost leaf node *ac* with a bitmap-code *1100000* following the pre-order. Assume the current reading position, d, in T_d is at the address *01* (Figure 4.2). We will try to find whether the first child of *ac*, *ace* (with a bitmap-code *1110000*), resides in T_d by searching from d. We can find the same bitmap-code *1110000* with support 2 at the address *03*, and will add a node *ace* in T_m. Then we will continuously search for *ace*'s child, *aced* (with a bitmap-code *1111000*), and will find it at the address *04*. A new node will be added into T_m accordingly. Later on, we will back-track to the node *ac*, and try to see whether the 2nd child of *ac*, *acd* (with a bitmap-code *1101000*) exists in T_d by searching from the current reading position d, which is pointing to *05*. Suppose that we have created a node *ae* in T_m and is now trying to check whether *aed* with a possible bitmap-code *1011000* in T_d. When we search through and move the current reading position d to the address *12*, we can determine that *aed* does not exist in T_d because the bitmap code of *aed* (*1011000*) is greater than the bitmap-code at the address *12* (*1001000*), which implies that *aed* cannot exist after the address 12 following the pre-order. Therefor, we can efficiently skip some checking. We will repeat the process while traversing T_m. Every disk-page

will be accessed at most once.

Second, we consider how to merge an non-empty in-memory tree with a projected on-disk subtree when the specified sub-itemset V is non-empty. Here, we also assume the given $\tau \geq \tau_m$. However, we can not use the same traversing-and-extending approach, because any item $i \in V$, that we need to obtain from the on-disk tree, may exist at different places in the on-disk tree. Obtaining all occurrences of i from the on-disk tree will result random I/O and there is no guarantee that the disk-pages will be read at most once. In order to read a disk-page at most once, we first load those items that do not exist in the in-memory tree from the on-disk tree. Then, we merge two in-memory trees. The in-memory tree merging will be discussed next.

Third, we use an example to explain how to merge two in-memory trees using Algorithm 2. In Figure 4.5, there are two trees in memory, T_1 rooted at *root1* containing items of $\{c,\ e\}$, and T_2 rooted at *root2* containing items of $\{g,\ b,\ f\}$. We show how to project the itemset $V = \{g,f\}$ from T_2 to T_1. Initially, we call $\pi_{m2m}(root1,\ root2,\ V,\ X)$, where $X = \{c,e\}$. Thus, starting from *root2*, it will recursively call $\pi_{m2m}(root1,\ g,\ V,\ X)$ to check the first child of *root2*, the node g. There are three bitmap-codes with this node. The first and the third bitmap-codes *1110100* and *0111100* should be handled as a descendant of *ce* *0110000* in T_1, because both g and f have lower frequency than any item in $X = \{c,\ e\}$ and both bitmap-codes indicate that they contain both c and e. Therefore, we traverse from *root1* and follow the path *ce* until we reach the node *ce*. We then find that it has no children that contains g at this stage. Thus a new node *ceg* is created with support 2 and both bitmap-codes *1110100* and *0111100* are attached to the newly created node. Comparing the second bitmap-code of node

g in T_2 (*1001100*) and X, we can determine that this occurrence of g does not have any ancestors in T_1, so a new node g is created as the last child of *root1* with the same bitmap-code *1001100*. At the end, we add g into X, so X becomes $\{c, e, g\}$. We continuous tree merging by checking the node g's children in T_2. Because its first child node b is not included V, we will call $\pi_{m2m}(root1, gf, V, X)$ to project gf from T_1 to T_2. The similar procedure will repeat until T_1 is completely merged with T_2.

4.4.3 Frequent Itemset/Sub-itemset Tree Building Request

The implementation of the frequent Itemset/sub-itemset tree building request is sketched in Algorithm 3. Recall a frequent itemset tree building request with τ can be supported as a frequent sub-itemset tree building request with the same τ, where V is set to be F as supported in the PP_D-tree. Therefore, the same method can be used for these two types of requests. The strategy we used in the procedure $selectTreesInMemory$ is to select a tree, T_i, in-memory that contains the longest consecutive itemsets in V. Let V be the intersection of the itemsets contained in the in-memory T_i and V. We repeat the same procedure until V becomes empty or we can not handle any more 1-itemsets. If V is empty, we say it is sufficient to construct a tree for the frequent sub-itemset tree building request using in-memory trees only. The procedure $loadSubtreeFromDisk$ is very similar to the one we used in our previous work [25]. The procedure $p\&mMem$ merges subtrees as follows. First, it sorts V following the $order(i)$ for any $i \in V$. Second, we create an empty tree T_r. Third, we attempt to find an in-memory tree T that contains all the top consecutive high frequency k items in V, denoted by V'. Let $V \leftarrow V - V'$. We project V' from T and merge the projected subtree

into T_r. We repeat the second step until V becomes empty. The procedure $p\&mMemDisk$ takes the similar approach. It needs projects all the items that can not be found any in-memory trees from the on-disk tree. Then, it can call the procedure $p\&mMem$ to complete the task.

Remarks *Suppose V_1, V_2 and V_3 are three itemsets all of which are subset of V, where $V_1 \bigcap V_2 = \phi$ and $V_2 \bigcap V_3 = \phi$. V_1 and V_3 can be projected from T_1 and V_2 can be projected from T_2. If for any item i_1 in V_1, any item i_2 in V_2 and any item i_3 in V_3, $order(i_1) < order(i_2) < order(i_3)$, we will do $\pi_{V_1}(T_1)$, $\pi_{V_2}(T_2)$ and $\pi_{V_3}(T_1)$ one after one. This is because if we project $V_1 \bigcup V_3$ at first, when projecting the nodes for V_2, which will be the ancestors of the nodes for V_3, the nodes for V_3 will need to be split into different branches. It is not efficient to do this additional work of splitting branches.*

Algorithm 3 SubTree-Request(τ, V)

Input: τ is a minimum support and V is a itemset.
Output: a PP_M-tree.

```
1: T_r ← ∅;
2: T_s ← selectTreesInMemory(τ, V); {T_s is a set of trees {T_1, T_2, ··· T_n}.}
3: if T_s ≠ ∅ then
4:   if T_s sufficient to construct the requested tree then
5:     if T_s = {T_1} then
6:       T_r ← π_{τ,V}(T_1);
7:     else
8:       p&mMem(T_r, τ, V, T_s);
9:     end if
10:  else
11:    p&mMemDisk(T_r, τ, V, T_s);
12:  end if
13: else
14:   T_r ← loadSubtreeFromDisk(τ, V);
15: end if
16: return T_r;
```

4.4.4 The Tree Projection Operation $\hat{\pi}$ and Frequent Super-itemset Tree Building Request

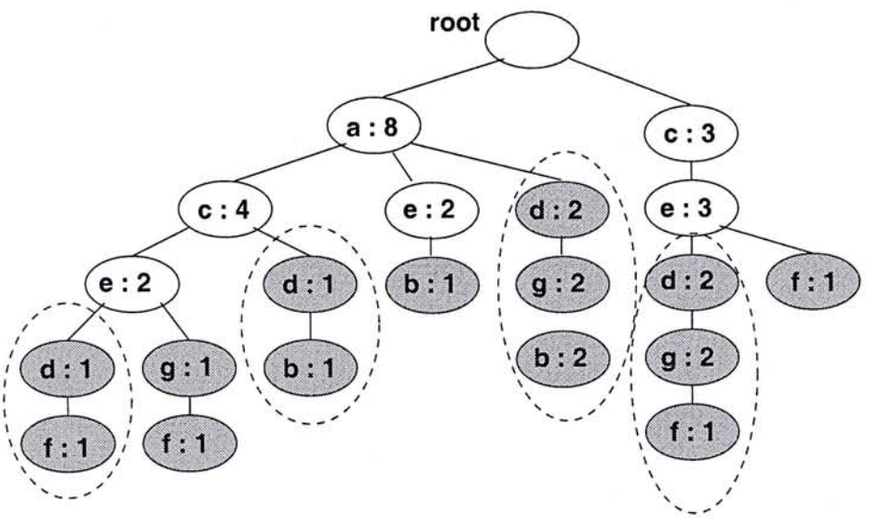
As is addressed above, given a frequent super-itemset tree building request with τ and itemset V, the tree constructed from the proxy is $T_0 = \hat{\pi}_{\tau,\hat{V}}(P) = \hat{\pi}_{V,\hat{V}}(P)$, where V consists of $\hat{I_n}$, the item with the largest order in $\hat{V}$, and the items whose order is larger than $\hat{I_n}$ and support is larger or equal to τ. After we have constructed the tree consisting of the items in V and satisfying the super-itemset constraint $\hat{V}$, the upper-part of the tree consisting of the items whose order is less than $\hat{I_n}$ can be constructed using the codes for $\hat{I_n}$ having been loaded in the previous step.

Therefore, the procedure of constructing a tree for a frequent super-itemset tree building request can be divided into two steps. In the first step, it projects a subtree from either memory or disk including all items in V. The resulting projected tree contains enough information to construct its upper half tree to support the super-itemset tree building request without any more I/O costs. This step can be implemented using some $\hat{\pi}$ operations on some different trees and merge the projection results. It can be expressed as $T_0 = \hat{\pi}_{V_1,\hat{V}}(T_1) \oplus \hat{\pi}_{V_2,\hat{V}}(T_2) \oplus \hat{\pi}_{V_3,\hat{V}}(T_3) \oplus ... \oplus \hat{\pi}_{V_n,\hat{V}}(T_n)$, where T_is are *PP*-trees in P, $V_1 \bigcup V_2 \bigcup ... V_n = V$ and $V_i \cap V_j = \phi$, $i \neq j, i, j = 1, 2, 3, ..., n$. The only difference is that when a code is to be loaded, we use $\hat{V}$ to check if all the bits in the codes corresponding to the items in $\hat{V}$ are set to be 1. If any bit is not set to be 1, the code can not be loaded and the branch rooted at it is not need to be traversed, because their ancestors do not include all the items in $\hat{V}$.

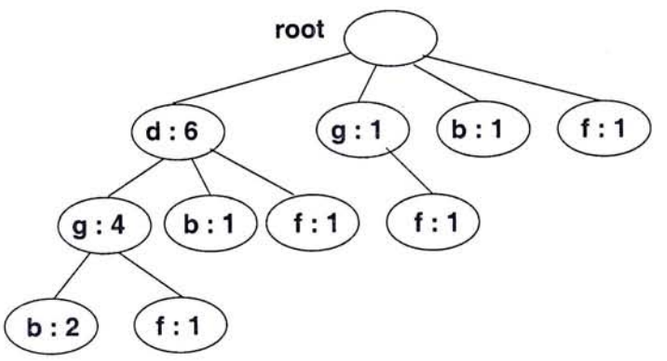
In the second step, we will construct its upper part tree using the projected

tree itself. Figure 4.6 shows an example for a super-itemset tree building request with $\tau = 4$ and $\hat{V} = \{c,\ d\}$ against an on-disk tree T_1 (Figure 4.2). In the first step, it loads the $V = \{d,\ g,\ b,\ f\}$ from the on-disk tree, where d is $\hat{I}_n$. Here, d has the largest $order(d)$ for the items in V. A subtree is projected from T_1 into memory such that it includes all of the four items in V if every path in the projected tree includes c *and* d. The resulting projected tree of the first step is shown on the left side in Figure 4.6. On the right side of Figure 4.6, we show the resulting $\hat{\pi}_{4,\{c,d\}}(T_1)$ where the shaded nodes are constructed using the bitmaps in the nodes in the left tree. As you can see from the figure, each code of d on the left side corresponds to a shaded path on the right side, and each bit set to be 1 of a code of d has one node on the right side to represent it excluding the bits corresponding to the item in $\hat{V}$. Note: because it is a tree for frequent super-itemset mining query, all itemsets (all paths) should include $\{c,\ d\}$, and therefore the root node indicates the fact. In Figure 3.2 (d), T_5 can be constructed from the in-memory tree T_2 because T_2 contains all the items $\{b,\ f\}$ that are needed to project from the on-disk T_1.

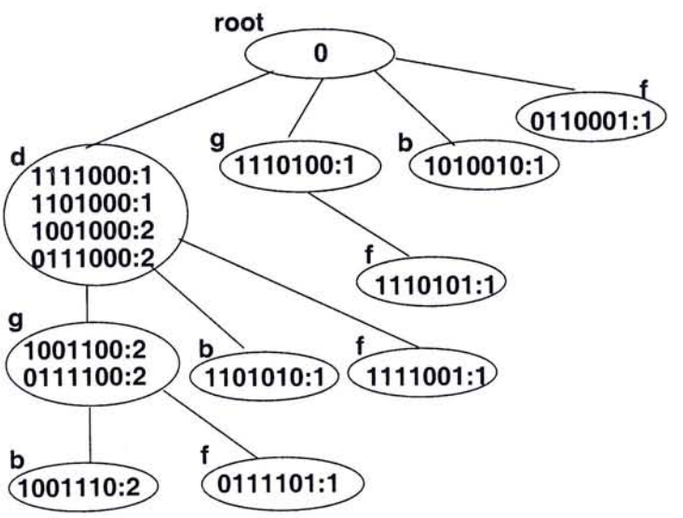
□ **End of chapter.**



(a) T_1



(b) $T_2 = \pi_{4,\{d,g,b,f\}}(T_1)$



(c) $T_2 = \pi_{4,\{d,g,b,f\}}(T_1)$

Figure 4.3: In-Memory PP_M-trees

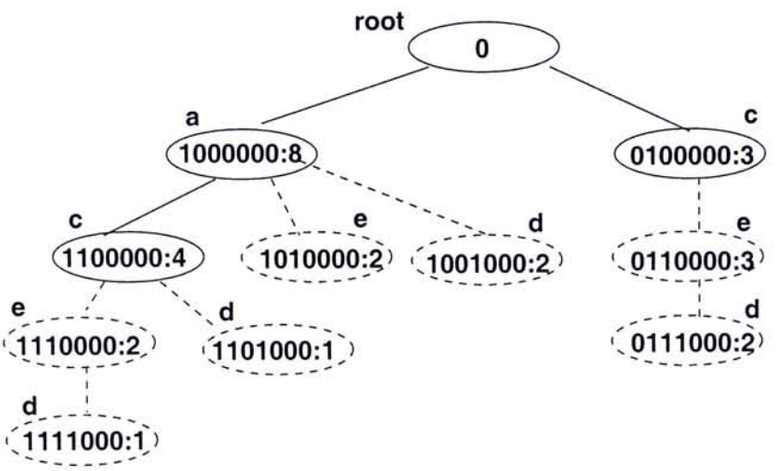


Figure 4.4: Merging an in-memory tree with a projected on-disk tree

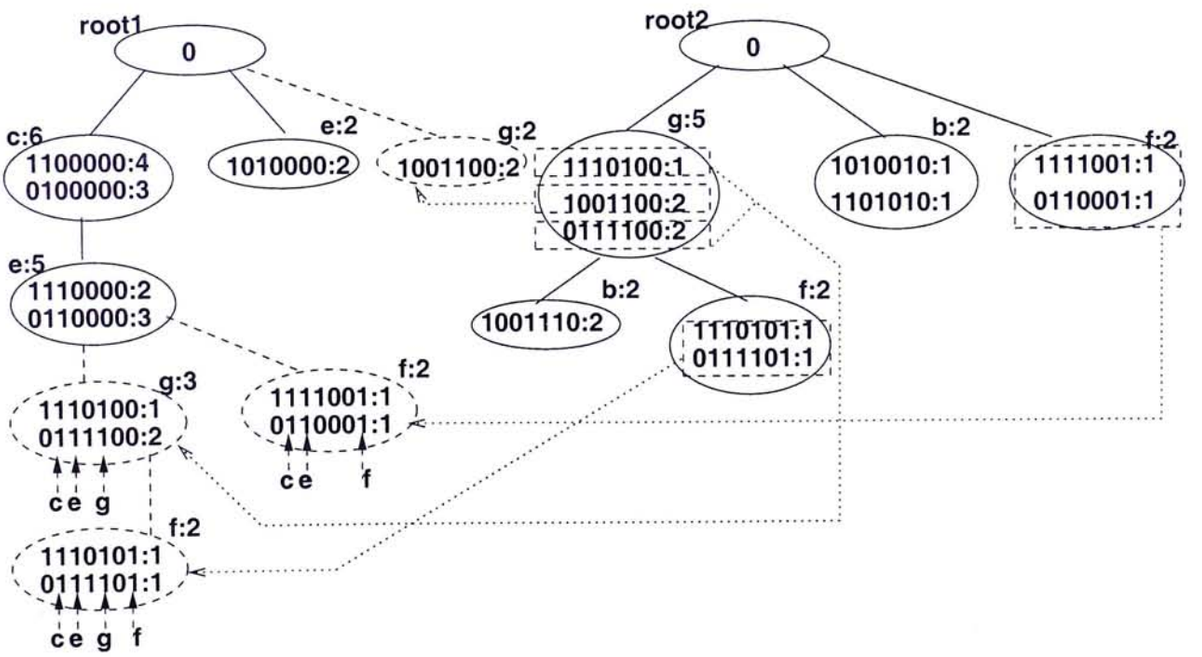


Figure 4.5: In-Memory Tree Merging

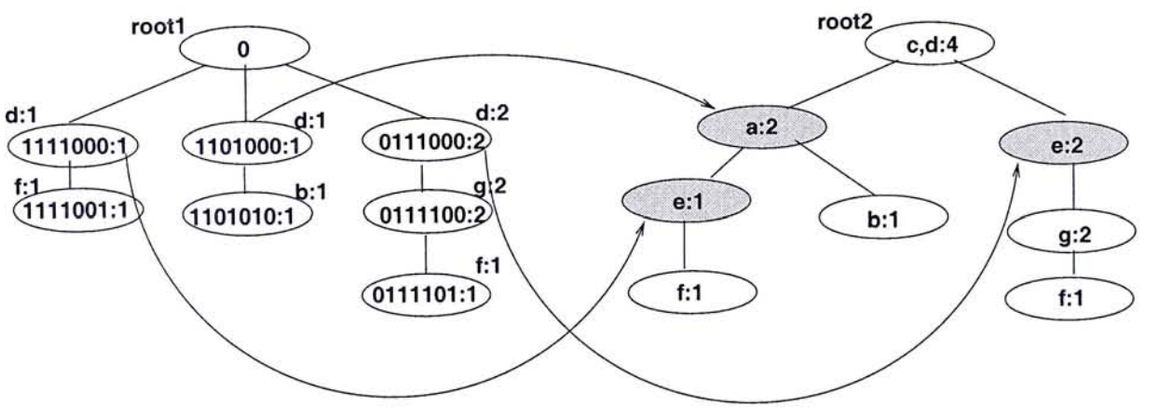


Figure 4.6: An example for $\hat{\pi}$

Chapter 5

Performance Studies

Summary

The extensive experiments have been done to test the data mining proxy. We report the experiment results in this chapter which show the good performance of the data mining proxy.

We conducted our experimental studies on a PC with a Pentium 4 2.0GHz, 1G Byte RAM running Windows XP Professional Edition. The method is implemented in *C++* using MS Visual *C++ 6.0*. Several datasets were used. We report our results using three datasets: *T25.I20.D100K* with 100K items (*Sparse*), with 10K items (*Medium*) and with 1K items (*Dense*). Table 5.1 shows the actual sizes of datasets being generated. The sizes of the three data files are 1,295 8KB disk-pages (*Sparse*), 1,290 8KB disk-pages (*Medium*) and 1,265 8KB disk-pages (*Dense*), after converting them into binary format (one integer 4 bytes).

Figure 5.1 shows the sizes of PP_D-trees (number of 8KB pages) using different τ_m with which the disk-based trees are built. Figure 5.1 (a) shows the sizes

Data set	Tran Number	Item Number	Tran Len	Pattern len	Size
Sparse	100K (89,180)	100K (58,874)	25	20	10,360K
Medium	100K (89,031)	10K (9,355)	25	20	10,320k
Dense	100K (99,847)	1K (990)	25	20	10,114K

Table 5.1: Dataset Sizes

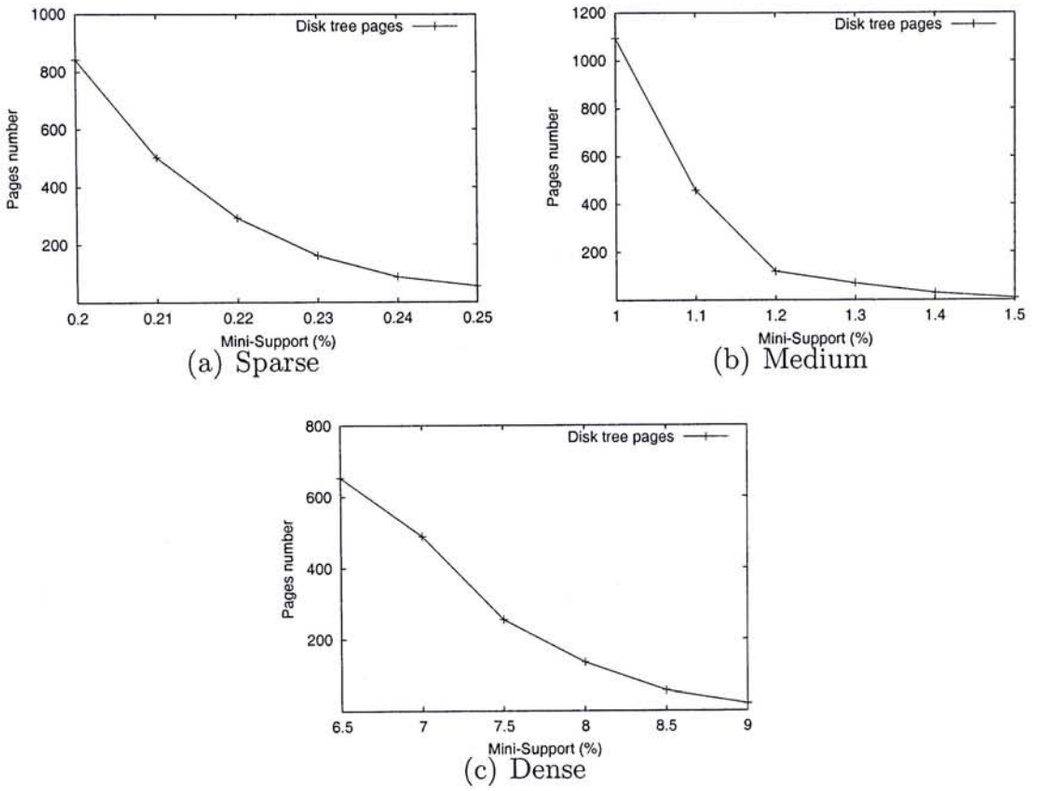
of PP_D-tree for *Sparse* in the rage of minimum supports, 0.2 and 0.25. Figure 5.1 (b) shows the sizes of PP_D-tree for *Medium* in the rage of minimum supports, 1.0 and 1.5. Figure 5.1 (c) shows the sizes of PP_D-tree for *Dense* in the rage of minimum supports, 6.5 and 9.0. As seen in Figure 5.1, the disk-based trees are considerably small.

Dataset(%)	Avg-Bits	Max-Bits	Std Diviation	Freq Item #
Sparse ($\tau_m = 0.01$)	320	1,056	278	48,977
Sparse ($\tau_m = 0.21$)	64	256	36	659
Medium ($\tau_m = 1.1$)	80	240	50.7	224
Dense ($\tau_m = 6.8$)	48	160	39.2	64

Table 5.2: Code Compression

In order to compress the bitmap codes, we adopted the compression method given in [22], which uses a list of integers to indicate in which position the bit is set to be 1. Since the number of "1" in the code is constrainted by the length of trasaction, the number of integer to express bitmap-codes is also limited. We show the compression ratio of bitmap-codes in Table 5.2. The numbers of items are 48977, 659, 224 and 64, respectively, when $\tau_m = 0.01$ and 0.21 for *Sparse*, $\tau_m = 1.1$ for *Medium*, and $\tau_m = 6.8$ for *Dense*, which means that the length of

bitmap-code would be 48977, 659, 224 and 64 without compression.



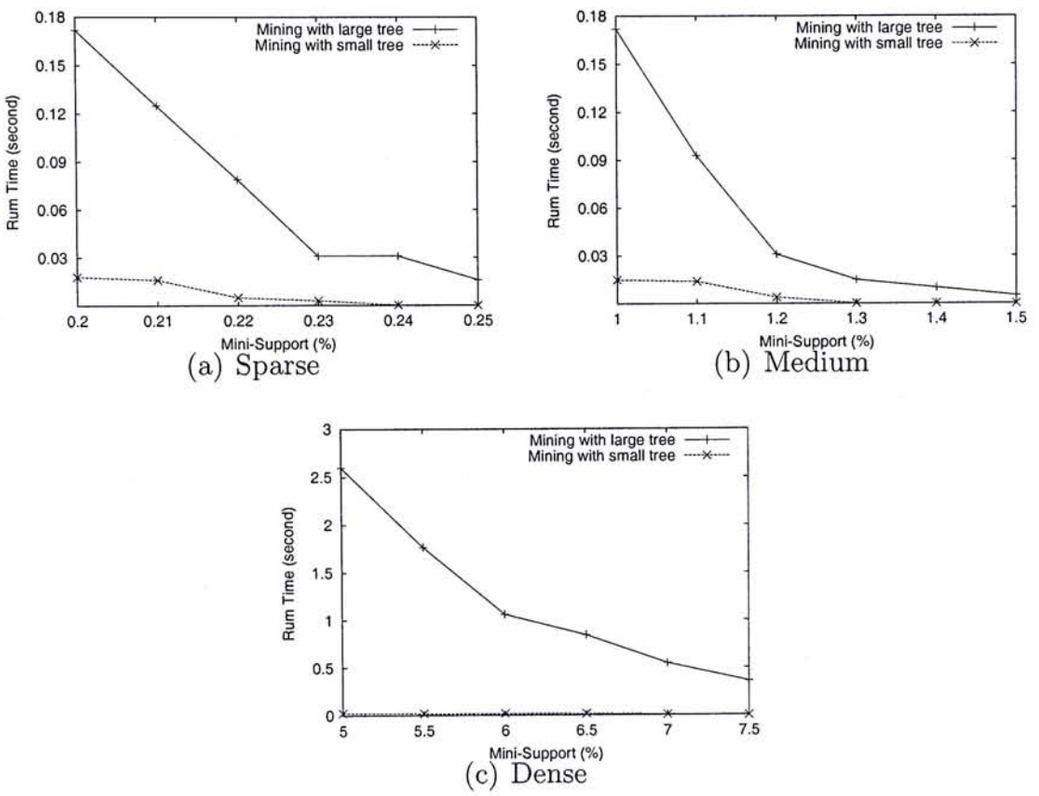
(a) Sparse (b) Medium

(c) Dense

Figure 5.1: Disk tree sizes with different τ_m

5.1 Mining with Different Sizes of Trees in Main Memory

First, we show the cost of data mining with different in-memory tree sizes (excluding the constructing the initial tree). The mining cost can be significantly reduced if a small but sufficient tree is used. We consider a frequent sub-itemset mining query with a minimum support τ and an itemset V of size of 10 items. First, we constructed a large in-memory tree with τ to mine patterns with V.



(a) Sparse (b) Medium

(c) Dense

Figure 5.2: Tree sizes vs mining cost (excluding the time for constructing the initial tree)

Second, we constructed a small in-memory tree to mine with the requested 10 items in V only. Figure 5.2 shows the differences of mining cost while τ is varying. The figures show that mining benefits from a smallest but sufficient subtree.

5.2 Constructing Trees in Main Memory

In this experimental study, for a frequent itemset tree building request with (τ), we show the time to construct a small tree in main memory using three different approaches: i) constructing the tree using TDB (TDB), ii) constructing the tree by loading the smallest tree from PP_D-tree on disk (*Disk-tree*), and iii) construct-

ing the tree by utilizing the tree(s) resident in main memory *Memory-tree*).

Figure 5.3 shows the case for constructing the initial in-memory tree using different approaches, namely, using *TDB* directly, using an on-disk tree, or using an in-memory tree. Three different datasets were used in this testing (*Sparse/Medium/Dense*).

First, in Figure 5.3 (a), (c) and (e), we built an individual disk-based PP_D-tree for a different minimum support (τ_m) on disk. There are 6 different PP_D-trees in each of the above three figures. For each PP_D-tree with τ_m, we construct a PP_M-tree in memory with the same τ_m. In particular for the case ii), it is to construct the tree by loading the whole PP_D-tree. We observe that, for *Sparse*, the *Memory-tree* approach outperforms *Disk-tree*, and *Disk-tree* outperforms *TDB*. For *Medium* and *Dense*, *Memory-tree* always performs best. But, *TDB* outperforms both *Disk-tree* and *Memory-tree* when the given minimum support is very small (1 for *Medium* and 6.5 for *Dense*), because the number of involved items are small.

Second, in Figure 5.3 (b), (d) and (f), the on-disk trees for (*Sparse/Medium/Dense*) were built using the smallest minimum support with which the on-disk trees performed better than using *TDB*, $\tau_m = 0.2$ for *Sparse*, $\tau_m = 1.1$ for *Medium*, and $\tau_m = 7$ for *Dense*. By comparing Figure 5.2 and 5.3, it also shows that the time for constructing tree is the main cost when using pattern growth methods. For example, in Figure 5.2 (b), when $\tau = 1.1\%$, the time for mining tree is only 0.1 second, while in Figure 5.3 (f), the time for constructing tree is 2.9 seconds.

Some observations can be made for using a disk-based tree. First, we can build a *Disk-tree* which at least performs the same as *TDB* when constructing the initial tree on disk. Recall such minimum supports are considerably small so

that most of data mining tasks require a larger minimum support. Second, the performance of *Memory-tree* does not depend on how *Disk-tree* is built, because *Memory-tree* utilizes the trees residing in memory already.

In the following testing, as default, we use the following settings: for *Sparse*, an on-disk tree is built with $\tau_m = 0.21$ and the proxy size size is 5MB; for *Medium*, an on-disk tree is built with $\tau_m = 1.1$ and the proxy size is 6.5MB; and for *Dense*, an on-disk tree is built with $\tau_m = 6.8$ and the proxy size is 18MB. The number of tree building requests is fixed to be 1,000.

5.3 Query Patterns and Number of Queries

In this set of experimental studies, we investigated two things, the number of queries and the mixture of three query patterns: frequent itemsets tree building request, frequent sub-items tree building request and frequent super-itemset tree building request. In Figure 5.4 5.5 and 5.6, P and NP mean with and without the data mining proxy. We fixed *M:N:L = 60:20:20* (*M:N:L* means the percentages of queries for frequent itemsets tree building requests (M), frequent sub-itemset tree building requests (N) and frequent super-itemset tree building requests (L)). For each test, we also selected a range of minimum supports, $R = [R_{min}, R_{max}]$, and controlled the percentage of the minimum supports in that range. Three cases are considered, 100%, 80% and 50%, denoted *P/NP-100*, *P/NP-80* and *P/NP-50* where P and NP indicate either with-proxy or without-proxy. In addition, two kinds of ranges were tested, i) non-overlapping sliding window (Figure 5.4), ii) fixed R_{min} but enlarging R_{max} (Figure 5.5), and iii) fixed R_{max} but varying R_{min} (Figure 5.6).

In all cases, the performance using the proxy outperforms the one without using the proxy with the same setting. When the dataset is *Sparse*, the cases with the proxy are 5 times faster than the cases without the proxy.

5.4 Testing Sub-itemset Queries with Different Memory Sizes

We focused on sub-itemset building requests in this experimental study. We considered three kinds of windows of the range of orders of the items as above i) non-overlapping sliding window, ii) fixed R_{min} but enlarging R_{max}, and iii) fixed R_{max} but varying R_{min}. We varied the memory sizes for the data mining proxy. The results are shown in Figure 5.7, 5.8 and 5.9 where the label of bar indicates the memory size. In the figures, *Number of Disk Pages Accessed* shows the total number of disk pages accessed in each test consisting of 1000 queries. As shown in these figures, if the memory size was too small, e.g. 600 KB, the proxy did not work well, while if the cache size was appreciate, e.g. 2,400 KB, the performance using the proxy outperformed the one without proxy. In fact, the tree of the sub-itemset are very small, even the largest one. Therefore the memory size needed is not very large. Note: The database size is larger than 10 MB.

5.5 Replacement Strategies

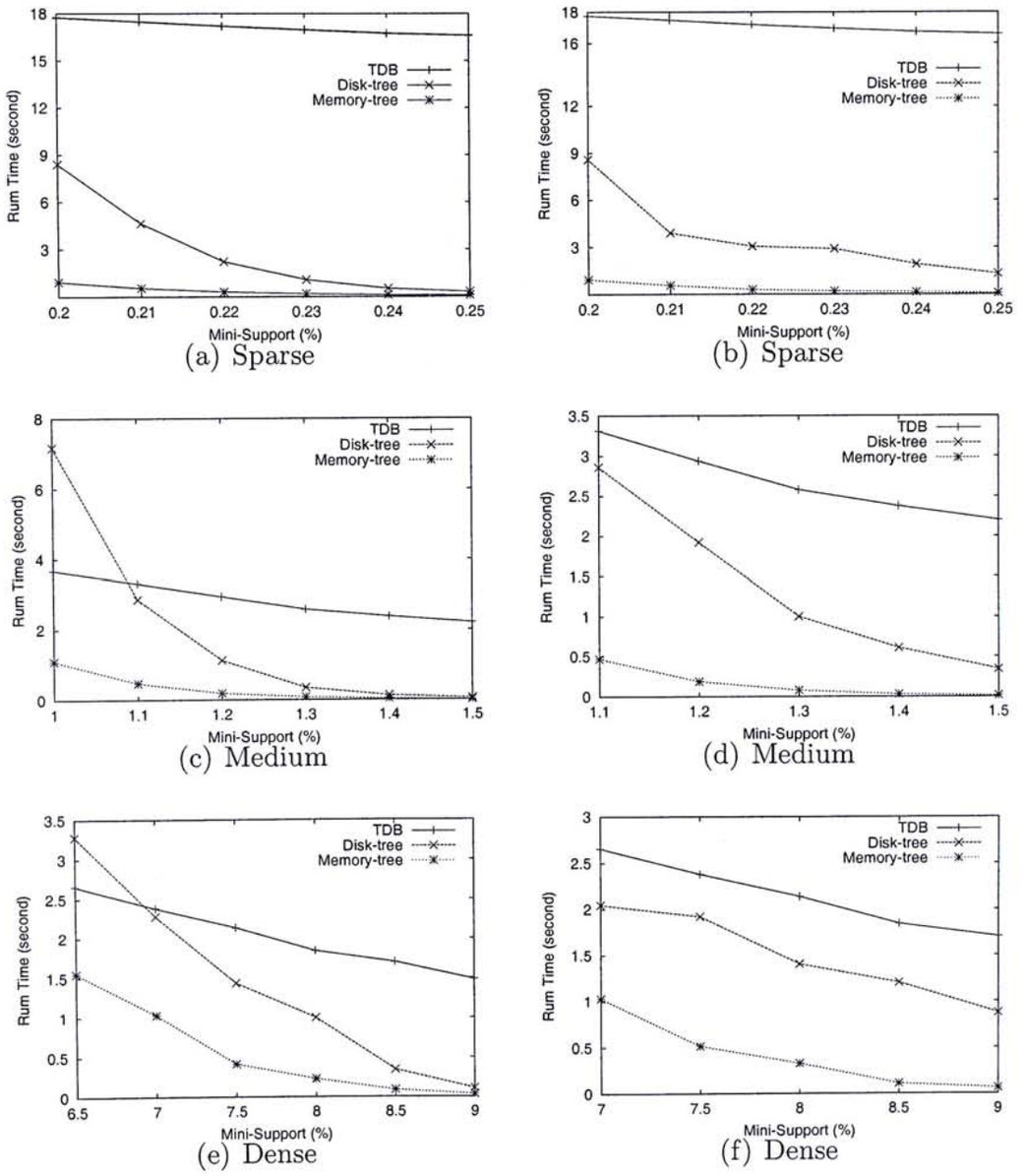
When the proxy becomes full, we need to remove some in-memory trees in order to support a new tree building request if we cannot use any in-memory trees. First, we choose the in-memory tree whose are a subtree of another in-memory

tree as a victim to be removed. Second, if more memory space is required, we consider three replacement strategies.

- *RP1*: Keep the trees in memory if the items in them are more frequently queried.
- *RP2*: Keep the trees in memory if the items in them appear in many nodes in the on-disk tree. It suggests that keeping them will reduce I/O costs, because the cost to reload these items is larger than the others.
- *RP3*: Keep the trees in memory if the items in them have higher support, because, generally, higher support items are more frequently queried.

Figure 5.10 shows the effectiveness of the three replacement strategies, for frequent sub-itemset tree building requests. The number along with the bar symbol (RPx) is the proxy size. We varied the items range in the experiment in two manners: sliding the range of items (Figure (a), (c) and (e)), and enlarging the range of items (Figure (b), (d) and (f)). The results showed that all three replacement strategies worked effectively in a similar manner.

□ End of chapter.

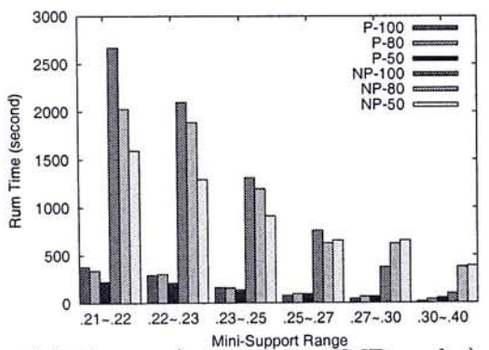


(a) Sparse (b) Sparse

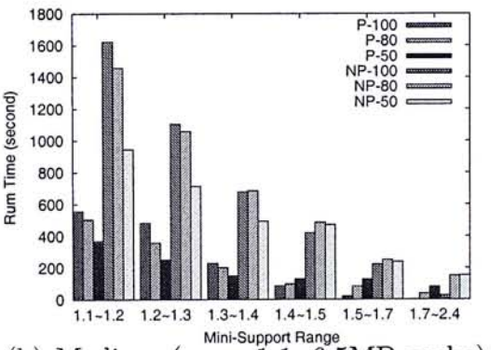
(c) Medium (d) Medium

(e) Dense (f) Dense

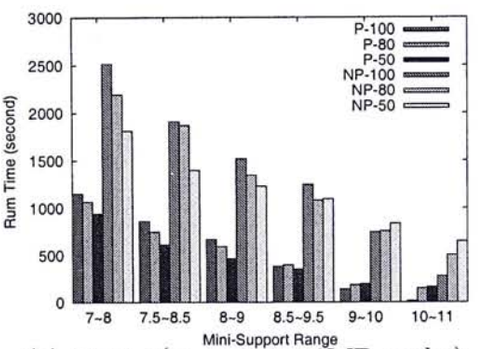
Figure 5.3: Constructing initial trees



(a) Sparse ($\tau_m = 0.21$, 5MB cache)

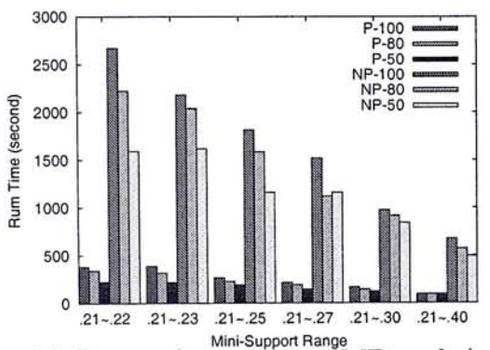


(b) Medium ($\tau_m = 1.1$, 6.5MB cache)

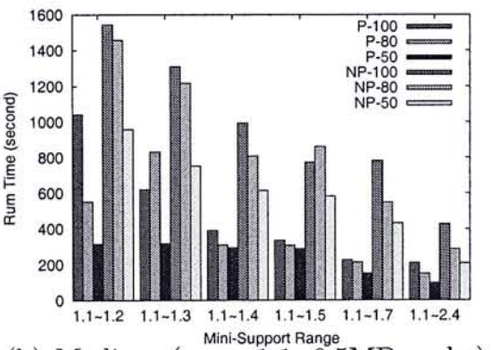


(c) Dense ($\tau_m = 6.8$, 18MB cache)

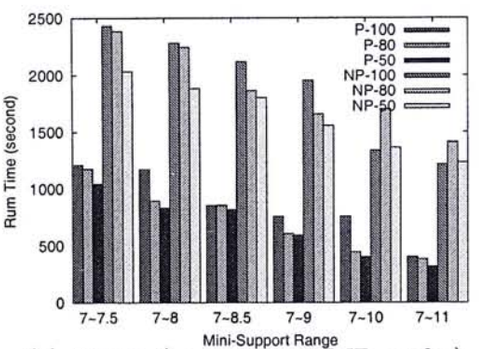
Figure 5.4: Various Queries Patterns I (1,000 queries, 60% are mini-support queries, 20% sub-itemset queries (5-10 items), and 20% super-itemset queries (1-5 items)



(a) Sparse ($\tau_m = 0.21$, 5MB cache)

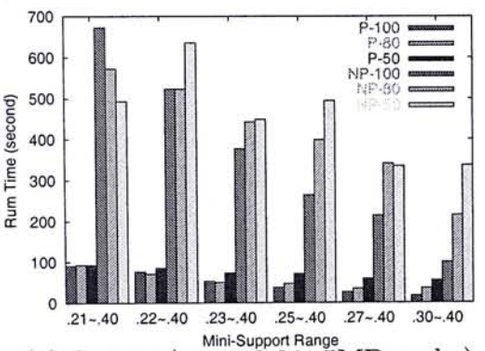


(b) Medium ($\tau_m = 1.1$, 6.5MB cache)

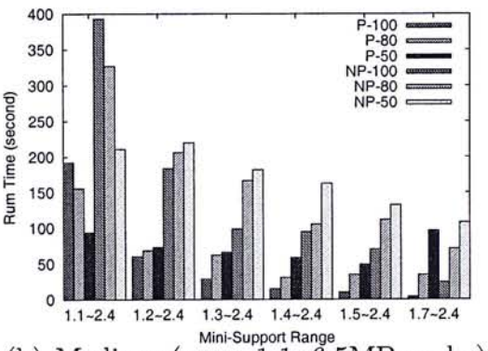


(c) Dense ($\tau_m = 6.8$, 18MB cache)

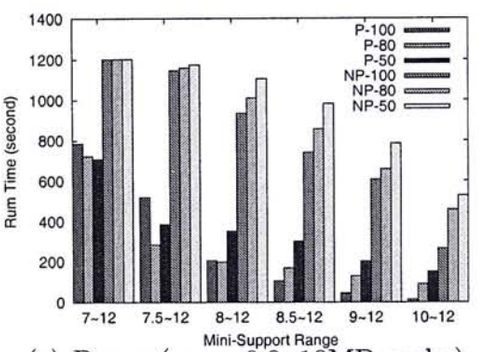
Figure 5.5: Various Queries Patterns II (1,000 queries, 60% are mini-support queries, 20% sub-itemset queries (5-10 items), and 20% super-itemset queries (1-5 items)



(a) Sparse ($\tau_m = 0.21$, 5MB cache)

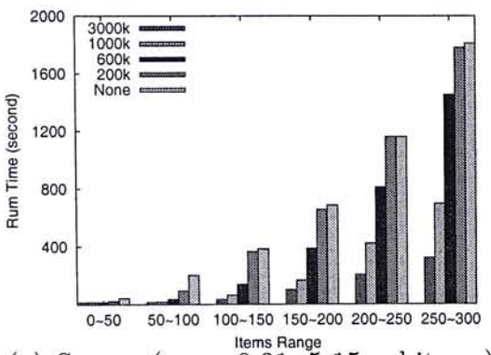


(b) Medium ($\tau_m = 1.1$, 6.5MB cache)

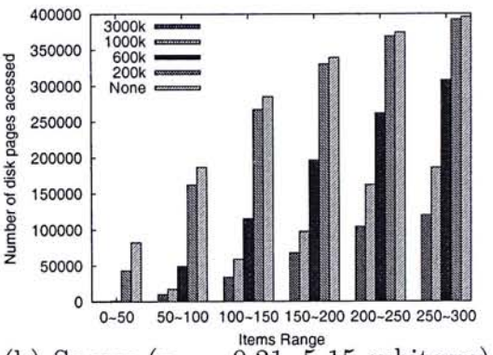


(c) Dense ($\tau_m = 6.8$, 18MB cache)

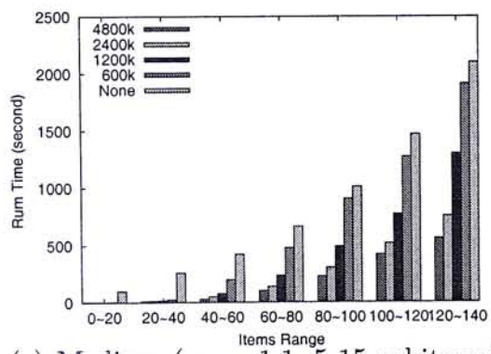
Figure 5.6: Various Queries Patterns III (1,000 queries, 60% are mini-support queries, 20% sub-itemset queries (5-10 items), and 20% super-itemset queries (1-5 items)



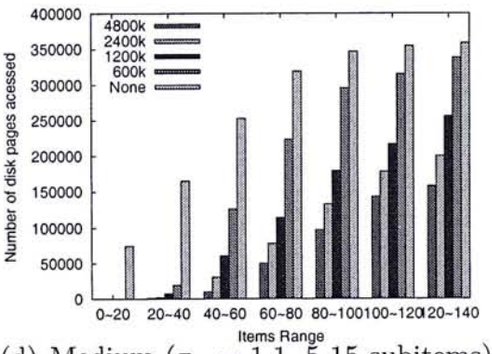
(a) Sparse ($\tau_m = 0.21$, 5-15 subitems)



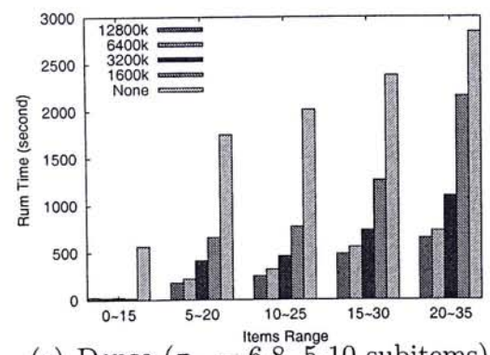
(b) Sparse ($\tau_m = 0.21$, 5-15 subitems)



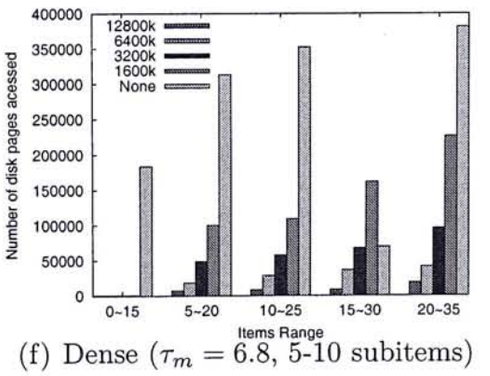
(c) Medium ($\tau_m = 1.1$, 5-15 subitems)



(d) Medium ($\tau_m = 1.1$, 5-15 subitems)

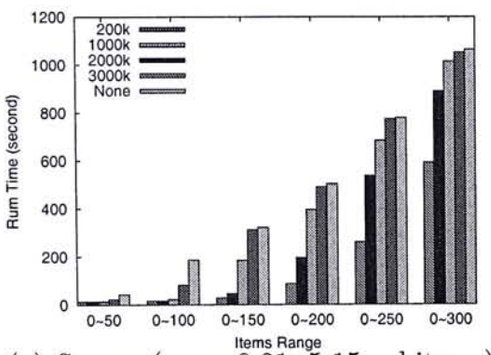


(e) Dense ($\tau_m = 6.8$, 5-10 subitems)

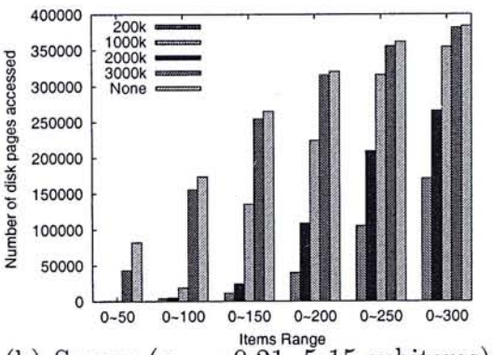


(f) Dense ($\tau_m = 6.8$, 5-10 subitems)

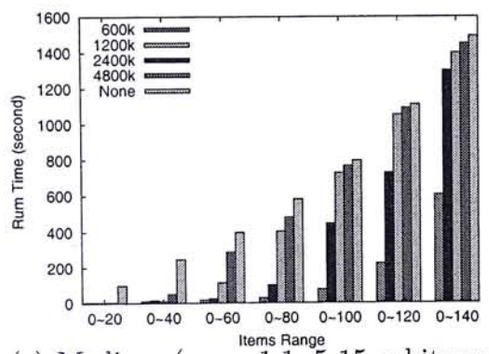
Figure 5.7: Sub-itemset Queries I (non-overlapping sliding window of $[R_{min}, R_{max}]$, 1,000 queries, all sub-itemset queries)



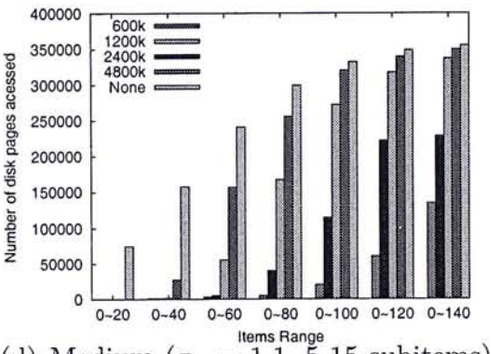
(a) Sparse ($\tau_m = 0.21$, 5-15 subitems)



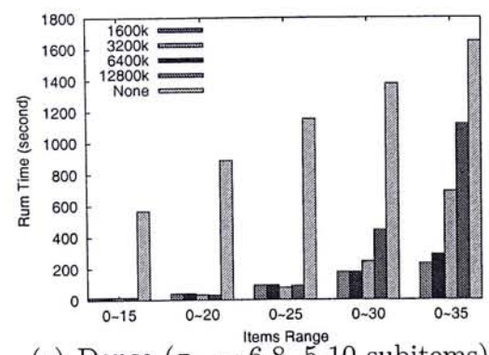
(b) Sparse ($\tau_m = 0.21$, 5-15 subitems)



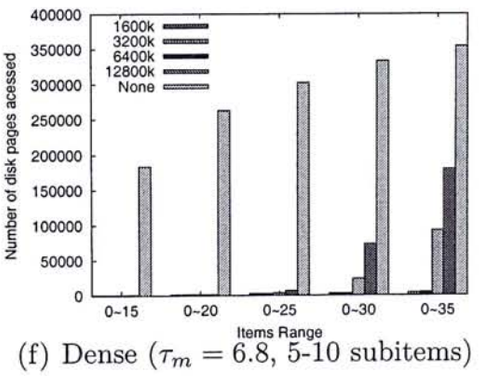
(c) Medium ($\tau_m = 1.1$, 5-15 subitems)



(d) Medium ($\tau_m = 1.1$, 5-15 subitems)

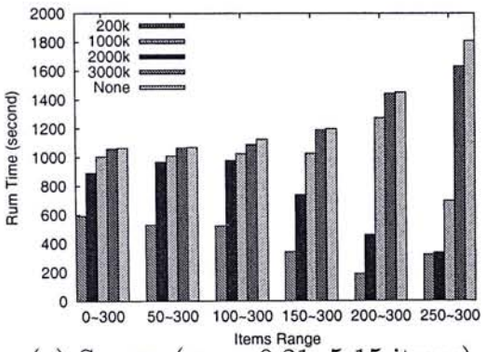


(e) Dense ($\tau_m = 6.8$, 5-10 subitems)

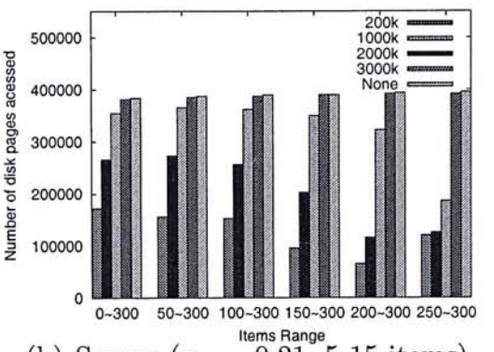


(f) Dense ($\tau_m = 6.8$, 5-10 subitems)

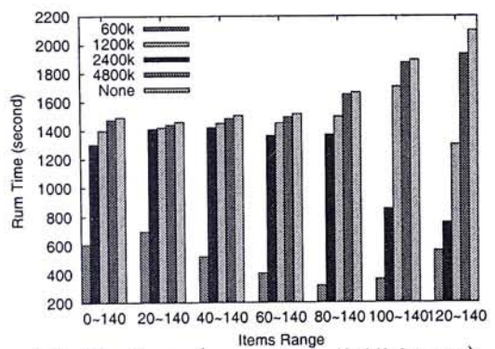
Figure 5.8: Sub-itemset Queries II (fixed R_{min} and enlarging R_{max} of $[R_{min}, R_{max}]$, 1,000 queries, all subitemsets queries)



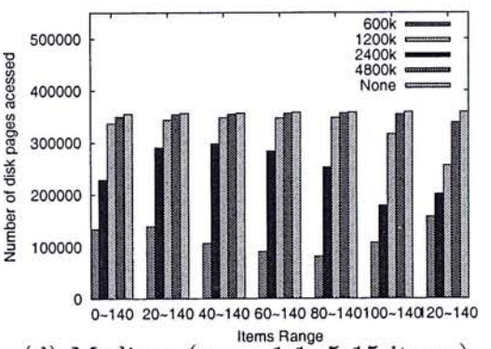
(a) Sparse ($\tau_m = 0.21$, 5-15 items)



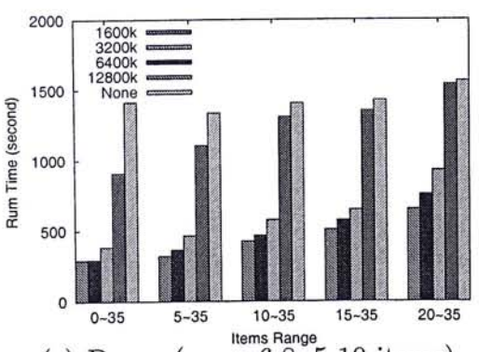
(b) Sparse ($\tau_m = 0.21$, 5-15 items)



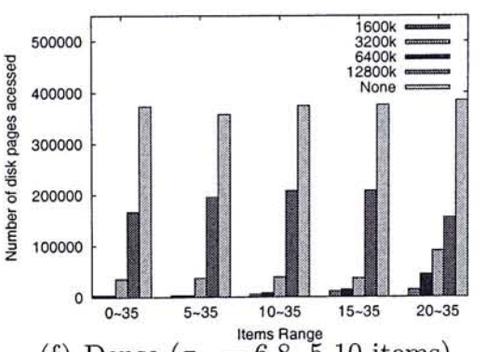
(c) Medium ($\tau_m = 1.1$, 5-15 items)



(d) Medium ($\tau_m = 1.1$, 5-15 items)

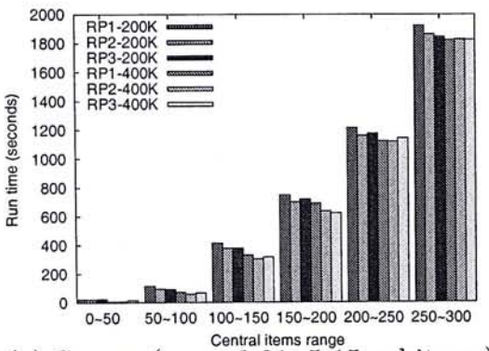


(e) Dense ($\tau_m = 6.8$, 5-10 items)

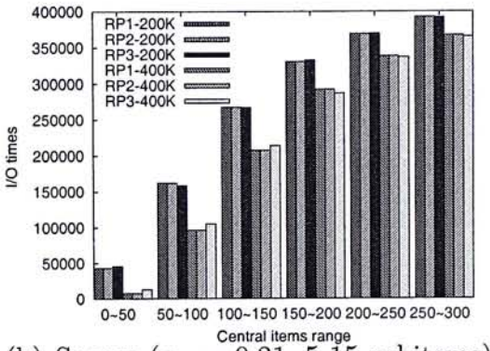


(f) Dense ($\tau_m = 6.8$, 5-10 items)

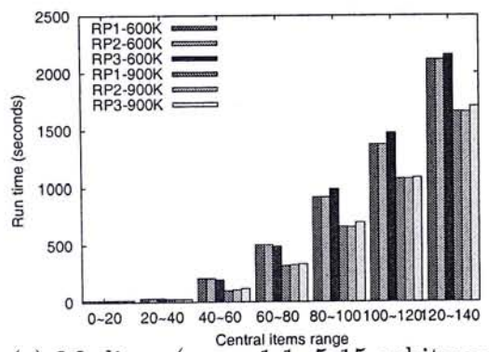
Figure 5.9: Sub-itemset Queries III (fixed R_{max} and varying R_{min} of $[R_{min}, R_{max}]$, 1,000 queries, all subitemsets queries)



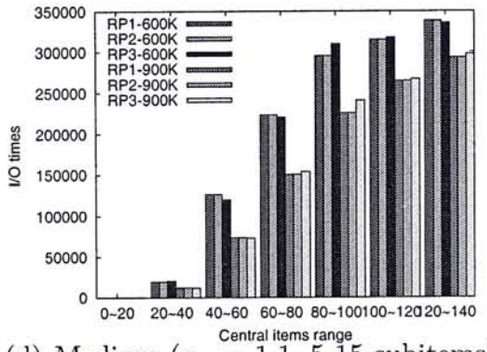
(a) Sparse ($\tau_m = 0.21$, 5-15 subitems)



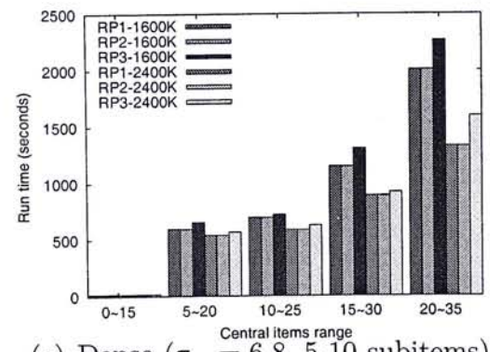
(b) Sparse ($\tau_m = 0.21$, 5-15 subitems)



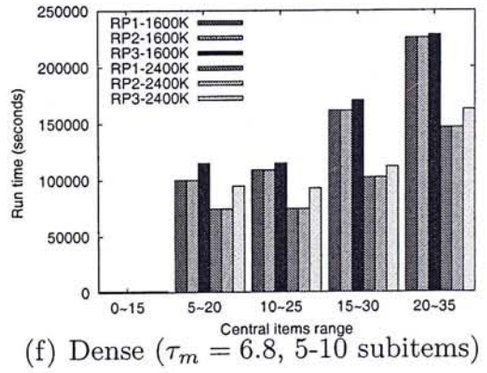
(c) Medium ($\tau_m = 1.1$, 5-15 subitems)



(d) Medium ($\tau_m = 1.1$, 5-15 subitems)



(e) Dense ($\tau_m = 6.8$, 5-10 subitems)



(f) Dense ($\tau_m = 6.8$, 5-10 subitems)

Figure 5.10: Replacement Strategies (1,000 queries, all sub-itemset queries)

Chapter 6

Privacy Preserving in Proxy Service

Summary

In this chapter, we propose a method to preserve privacy when the data for mining come from some different sites. The details of the method are illustrated. The advantages of our method are also discussed.

6.1 Data Union Regardless Privacy Preserving

In this section, we consider the problem that a user want to find the global frequent itemsets w.r.t. minimum support threshold τ from the transactional databases distributed in $n(\geq 3)$ different sites. If we do not concern the privacy problem, this task can be completed in the following four steps.

Step 1: According to Apriori [3] property, if an itemset I is frequent, any item i in I must be frequent. Thus, we need to find global frequent 1-itemsets at first. As stated in FDM [6], if an itemset I is globally frequent, it must be a local frequent itemset as least in one site. Therefore, first of all we need to request the local frequent 1-itemsets from all sites. These 1-itemsets make up of the candidates of global frequent 1-itemsets. Then we request each site to report its support to these candidates. For each candidates, we sum up the support from different sites and determine whether it is frequent in globe. Then the frequent 1-itemset are sorted following a certain total order $\preceq$.

Step 2: Let's call the global frequent items in a transaction the frequent projection of the transaction. Only the frequent projections of transactions will be used in data mining.

Definition 6.1 Max Path Itemset (MPI): *Suppose we have build a PP-tree for a transaction database,* Max Path Itemset *refer to the itemset represented by the path from the root of the tree to a leaf.*

In this step, the global frequent 1-itemsets sorted in a total order *preceq* are sent to all sites. Every site builds up a local *PP*-tree following the total order *preceq*. And then, each site send all MPIs in their own site to the user. The union of these MPIs are used to built up a global *PP*-tree while the counts of all nodes are set to be 0.

Step 3: The global *PP*-tree is sent to all sites. On each site, the counts of nodes on the tree are filled with local support. Then the local supports are sent back to the user. For each node on the global *PP*-tree, the counts of the corresponding node from different sites are added up to be the count of this node on global tree.

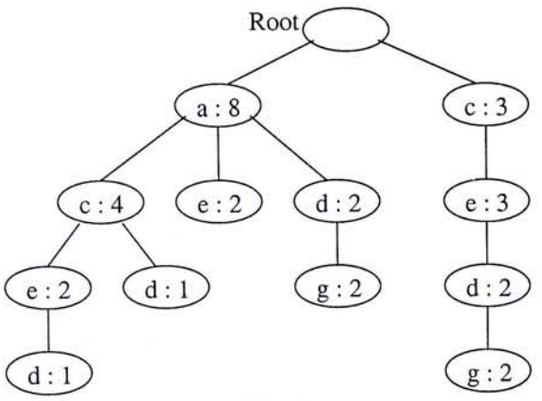
Step 4: Mining on top of the completely constructed *PP*-tree.

Let's use an example to explain these steps. A user submit a data mining query with $tau = 7$. In Figure 6.1, suppose site 1 holds the data which is shown in Figure 2.1 (Site 2 and Site 3 hold some other transaction databases). The global frequent items are a, c, d, e, g following frequency descending order. The tree shown in sub-figure (a) is the *PP*-tree storing the frequent projections of the transactions in Figure 2.1. Similarly, the tree in sub-figure (b)(resp., sub-figure (c)) consists of the frequent part of the transactions in Site 2 (resp., Site 3). Thus the MPIs in these three sites' make up of a set of itemsets. Following the procedure to build up a *PP*-tree, we use these itemsets to build up a global tree shown in sub-figure (d) regardless the count in each node. Finally, in sub-figure (e), we calculate the count for each node using the count of the same node on the local tree in each sites.

Theorem 6.1 *Given a minimum support threshold τ and n different sites, each of which holds a transaction database, following the step 1, 2 and 3 proposed above, we can build up a PP-tree equal to the PP-tree directly built up using the union of the transactions of the databases distributed in these n sites.*

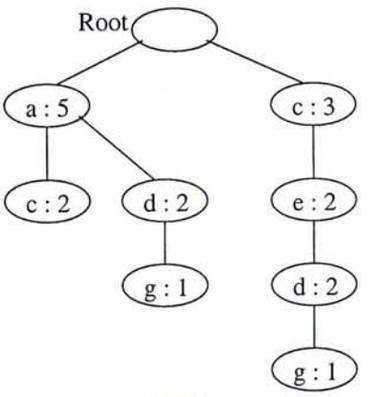
Proof: To prove this theorem, we only need to prove that any node on the directly built up tree (T_D), must has a same node with same count on the tree (T_S) built up following the above three steps proposed above.

Any node N_D with count k in T_D means that there are totally k transactions in all sites contains the itemsets I represented by the path from the root to N_D. Because there must be at least one MPI to be the superset of I, on the global *PP*-tree, there exists at least one path from the root to some node N_S represent the itemset I. Suppose the support of each site s_i for I is c_i, the count of N_D is



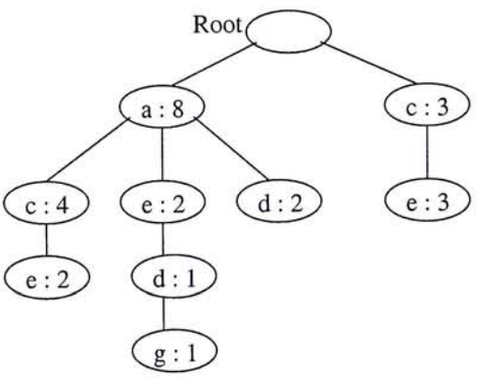
Site 1

(a)



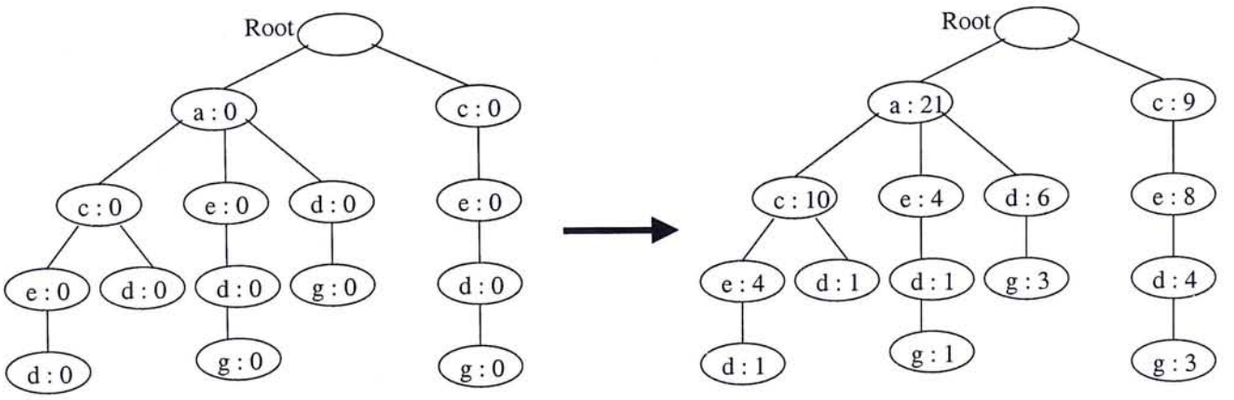
Site 2

(b)



Site 3

(c)



(d) (e)

Globe

Figure 6.1: Sharing Data

$\sum c_i$. In step 3, all sites fill in T_S's nodes with the local support for these itemsets and then the counts of the corresponding nodes are added up . Therefore, the count of N_S is also equal to $\sum c_i$. □

The details of the steps will be introduced in the next section together with the privacy preserving methods.

6.2 Secure Data Union

The security problems existing in the procedure proposed in Section 1 are: in step 1, when generating the global frequent 1-itemsets, the local frequent 1-itemsets and their support should not be revealed to the others; in step 2, when sharing the MPIs, the owner won't let the others know who owns it; in step 3, when sharing the count of each node, it obviously can not be revealed to the others.

6.2.1 Secure Multi-party Computation

In this thesis, we will try to protect the privacy under the semi-honest environment. In this section we will introduce the definition of semi-honest model and how to achieve privacy preserving w.r.t. semi-honest behavior.

The substantial work on secure multi-party computation prove that a wide class of computations can be computed securely under reasonable assumptions. We will have a brief view on this model, which is introduced in Goldreich [7] in detail. The definitions given below come from [7] and [11].

Security in semi-honest model: [11] A semi-honest party follows the rules of the protocol using its correct input, but is free to later use what it sees during execution of the protocol to compromise security. This requirement is

practical to be achieved in reality based on three observation: first, people need to follow the protocols so as to get the real data they want; second, it also allow the parties to freely use the any data they get since it is hard to control what the parties do on its own site; third, the embedded protocol in the software can not be easily altered.

Following is the definition of private two-party computation in the semi-honest model:

Privacy w.r.t. semi-honest behavior [7]:

Let $f : 0,1^* \times 0,1^* \longmapsto 0,1^* \times 0,1^*$ *be probabilistic, polynomial-time functionality, where* $f_1(x,y)$*(resp.,* $f_2(x,y)$*)denotes the first (resp., second) element of* $f(x,y)$ *and let* Π *be two-party protocol for computing* f.

Let the view of the first (resp., second) party during an execution of Π *on* (x,y), *denoted* $view_1^{\Pi}(x,y)$ *(resp.,* $view_2^{\Pi}(x,y)$*) be* $(x, r_1, m_1, ..., m_t)$ *(resp.,* $(y, r_2, m_1, ..., m_t)$*) where* r_1 *represent the outcome of the first (resp.* r_2 *second) party's internal coin tosses, and* m_i *represent the* i^{th} *message it has received.*

The output of the first (resp., second) party during an execution of Π *on* (x,y) *is denoted* $output_1^{\Pi}(x,y)$ *(resp.,* $output_2^{\Pi}(x,y)$*) and is implicit in the party's view of the execution.*

Π *privately computes* f *if there exist probabilistic polynomial time algorithms, denoted* S_1, S_2 *such that*

$$\{(S_1(x, f_1(x,y)), f_2(x,y))\}_{x,y\in\{0,1\}^*} \equiv^C \{(view_1^{\Pi}(x,y), output_2^{\Pi}(x,y))\}_{x,y\in\{0,1\}^*} \quad (6.1)$$

$$\{(f_1(x,y)), S_2(x, f_2(x,y))\}_{x,y\in\{0,1\}^*} \equiv^C \{(output_1^{\Pi}(x,y)), view_2^{\Pi}(x,y)\}_{x,y\in\{0,1\}^*} \quad (6.2)$$

where $\equiv^C$ *denotes computational indistinguishability.*

From the definition given above we can get that a computation is secure if the view of each party during the execution of the protocol can be effectively simulated by the input and output of the party. The detailed explanation and proof can be found in [7]

6.2.2 Basic Methods of Privacy Preserving in Semi-honest Environment

In this section, we will introduce two simple method to achieve privacy reserving in semi-honest environment. These two methods will help us to solve the privacy problem in Section 1 of this chapter.

Consider a very simple situation: a group of people want to get the average salary among them while them won't let any of the others know the salary of their own. One method is shown in Figure 6.2. Suppose there are three persons in a group, A, B and C, and the salaries of them are \$3000, \$2000 and \$1000 respectively. Since A, B and C won't let the others know their own salary, they will tell the others their salary by distorting it. After they get the sum of the distorted values, they will correct it one by one. The procedure is: A tells B that his salary is \$4000 (= \$3000 + \$1000); B tells C the total salary of A and himself is \$5500 (=\$4000 + \$2000 - \$500); C tells A the total salary of them is \$8000 (=\$5500 + \$1000 + \$1500); A corrects the sum by subtracting \$1000 from

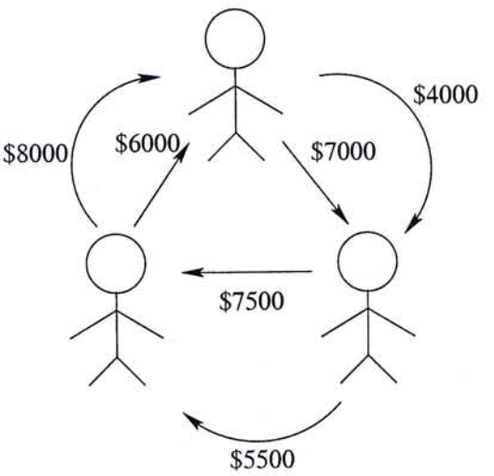


Figure 6.2: Example of securely calculate average salary

it and tells B the value \$7000; B corrects it by adding \$500 and tells C \$7500; C corrects the value by subtracting \$1500 and finally get the correct sum, \$6000, to calculate the average salary.

In some cases, we need a more complicated method. As shown in Figure 6.3, all sites use the same encryption method E. Suppose site A holds a datum K_A, starting from site A, each site uses its private encryption key to encrypt K_A. The result is $E_C(E_B(E_A(K_A)))$. In the same way, K_B, the datum held by B, and K_C, held by C, are encrypted to be $E_A(E_C(E_B(K_B)))$ and $E_A(E_B(E_C(K_A)))$ respectively. The three encrypted data are gathered together and sent to each site one after one to decrypt. Finally, we get the set A, B, C the same as their original value. This requires that the encryption methods E holds the property as the follows:

For any datum D_1 in domain D, any given feasible encryption keys $K_1, ..., K_n \in$

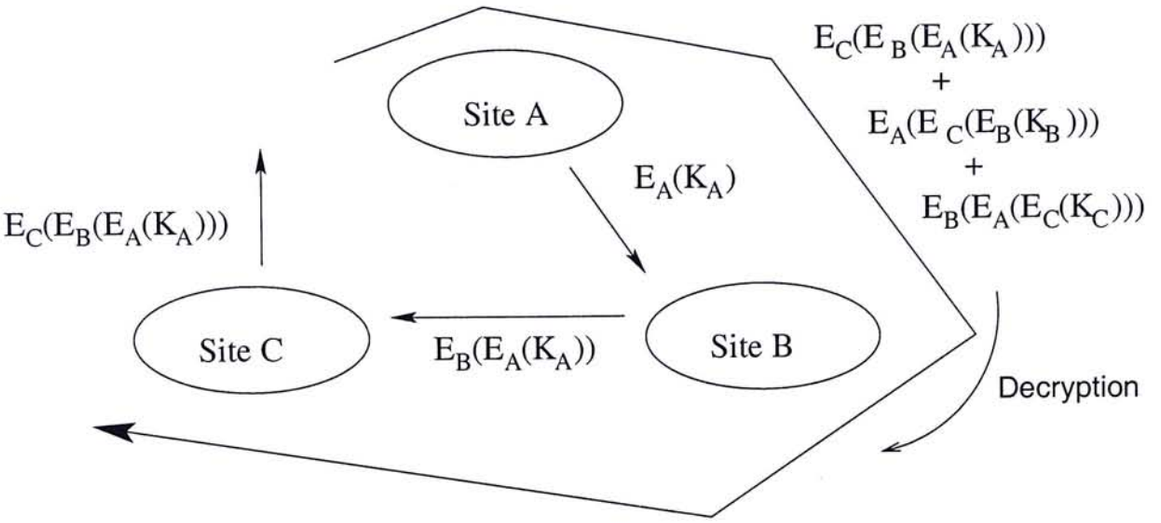


Figure 6.3: Example of securely sharing data

K and any permutations of i,j,

$$E_{K_{i_1}}(...E_{K_{in}}(D_1)...) = E_{K_{j_1}}(...E_{K_{jn}}(D_1)...) \tag{6.3}$$

$\forall D_1 and D_2 \in D$ such that $D_1 \neq D_2$ and for given k, $\epsilon < \frac{1}{2^k}$

$$Pr(E_{K_{i_1}}(...E_{K_{in}}(D_1)...) = E_{K_{j_1}}(...E_{K_{jn}}(D_1)...)) < \epsilon \tag{6.4}$$

An encryption algorithm holding the above two equations 6.3 and 6.4 is called commutative encryption. There are some examples of commutative encryption, like RSA [19] and Pohlig-Hellman encryption [18].

6.2.3 Privacy Preserving On Data Union

In this section, we will discuss how to add the privacy preserving methods into the steps of the procedure given in Section 1 of this chapter. We will first discuss step 2 and 3, and then step 1.

For the step 2, we can use the encryption method introduced in Figure 6.3. The details of the procedure are presented in Algorithm 4. In the algorithm, we use some commutative encryption to protect the MPIs of each site (from line 1 to line 8). On line 2 and line 3, we fake E_i by: (1) extending the length of some MPIs by appending some nonsensical items; (2) adding some itemsets to the set of MPIs. These two methods is to avoid the others getting the information of the real size of the local MPIs in one site. Applying these two methods will result in some nodes with count of 0 in the final global *PP*-tree. Obviously this will not affect the correctness of the mining results. On line 6 and some other lines, we permute the set of encrypted MPIs so as to prevent knowing exactly matching between the itemset before encryption and after encryption. From line 9 to line 15, all the encrypted MPIs from odd numbered sites are sent to one even numbered site and those from even numbered sites are sent to one odd numbered site. This is to prevent a site from getting the fully encrypted value of its own MPIs. From line 15 to the end are the procedure of decryption.

As is proved in [11], using this procedure to communicate itemsets between multi-sites is secure under the definition of multi-party computation except that: it reveals (1) the size of intersection of local MPIs between any subset of odd numbered sites; (2) the size of intersection of local MPIs between any subset of even numbered sites; (3) number of MPIs in at least one odd and one even numbered site. But it is different from the situations in [11]: (1) in [11] it reveals

Algorithm 4 Share MPIs among n sites ($n \geq 3$)

1: **for** each site S_i **do**
2: Extend some paths of the local *PP*-tree by appending some items to be the descendants of some leaves;
3: Let M_i be the set of MPIs of the faked tree; $E_i \leftarrow M_i$ and add some nonsensical itemsets to E_i;
4: **for** $j = 0$ to $n - 1$ **do**
5: $E_i \leftarrow E_i$ encrypted by $S_{(i+j)modn}$ use its own key;
6: Permute and send E_i to $S_{(i+j+1)modn}$;
7: **end for**
8: **end for**
9: **for** $j = 0$ to $n - 1$ **do**
10: **if** j is an odd number **then**
11: Permute and send E_j to S_0;
12: **else**
13: Permute and send E_j to S_1;
14: **end if**
15: **end for**
16: S_1 permute and send all the encrypted MPIs having been sent to it to S_0;
17: $E_{all} \leftarrow$ all the MPIs having been sent to S_0;
18: **for** $j = 0$ to $n - 1$ **do**
19: $E_{all} \leftarrow E_{all}$ decrypted by S_j use its own key;
20: Permute and send E_i to $S_{(i+j+1)modn}$;
21: **end for**
22: Using E_{all} to build up a *PP*-tree;

the intersection information of local frequent patterns which is really important, while here what it reveals is the intersection information of MPIs which do not have significant sense; (2) the MPIs can be faked by appending some nonsensical items but the patterns can not be faked in the procedure in [11].

For the step 3, we can use the encryption method introduced in Figure 6.2. The details of the procedure are presented in Algorithm 5. During the procedure of Algorithm 5, the arrays of the randomly generated numbers are transmitted and added up first (from line 3 to line 9), and then the array of sums is transmitted

and use the arrays of difference to correct it (from line 10 to line 12).

An example tree on the left side of Figure 6.4 is a local *PP*-tree after filled with the counts of the node by site 1 (Figure 6.1 (a)). On the right side of Figure 6.4, the first line is the array of the counts following deep-first order; the second line is the array of randomly generated numbers; and the third line is the array of the difference between the previous two arrays. Thus, the distributed sites can communicate using these arrays and correct the results with the array of the difference.

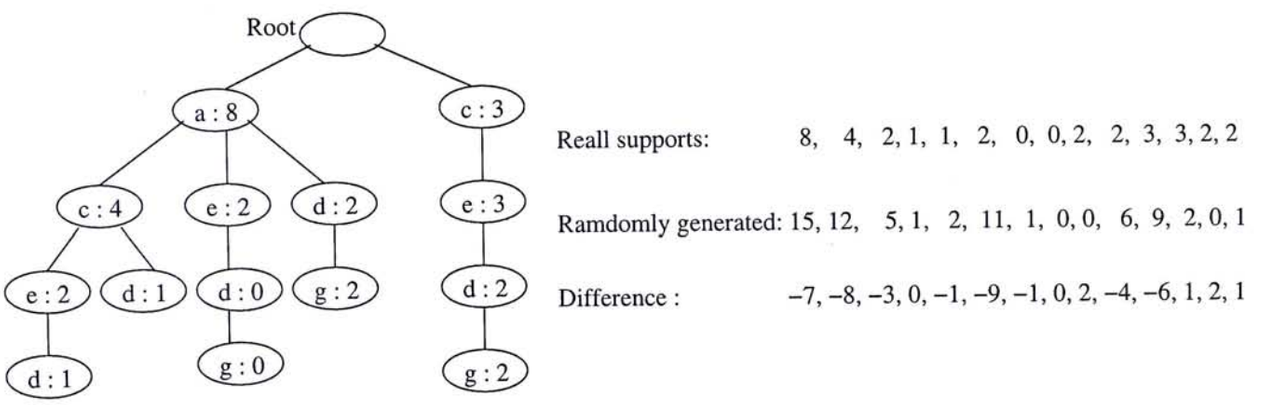


Figure 6.4: Example of sharing counts

For the step 1, firstly, we can use Algorithm 4 to unite the local frequent 1-itemsets so as to get the candidates of the global frequent 1-itemsets; Secondly, we can use Algorithm 5 to calculate the support of the candidate 1-itemsets so as to find the global frequent 1-itemsets.

Algorithm 5 Get the count of the nodes on global *PP*-tree's from n sites ($n \geq 3$)

1: Sent the global *PP*-trees to all sites;
2: Suppose the number of the nodes on the global *PP*-trees is k, generate an integer array, C, consisting of k zeros.
3: **for** each site S_i, where $i = 0$ to $n - 1$ **do**
4: Fill in the count of the nodes on the *PP*-tree it just received with the support of this site to the itemset represented by the node.
5: Sort the count of the nodes in deep-first order into an array N_i;
6: For each count, randomly generate an array, R, consisting of k integers and calculate the difference between N_i and R_i;
7: $C \leftarrow C + R_i$;
8: Send the array of the randomly generated numbers to next site;
9: **end for**;
10: **for** each site S_i, where $i = 0$ to $n - 1$ **do**
11: $C \leftarrow C + (N_i - R_i)$;
12: **end for**

6.3 Discussions

The privacy preserving methods introduced in this chapter fully utilize the advantages of the tree structure. In Algorithm 4, it only constructs the framework of the tree, so that it only needs very little information, the *max path itemsets* of each site. Anymore, MPI can easily be faked without affecting the correctness of the mining results so that the danger of leakage is very small.

The communication cost is also very small. During the whole procedure, each site only need to transmit data four times. In each time, they need to transmit encrypted data once and decrypted data once. The first and second time is to transmit the local frequent 1-itemsets and the local support for the global frequent 1-itemsets. occurs when . Obviously, in these two times, the communication cost is very small since the size of data is limited by the number of the distinct items in the databases. The third time of transferring data is to

transmit the local site's MPIs. And the forth time is to transmit the counts of the nodes on global tree. The size of these data is much smaller than that of the original database. But if we use the methods requiring to transmit local patterns, in the cases that the minimum support threshold is very low, the size of patterns will be hundreds or thousands times as large as that of the original database because the number of patterns explodes when the threshold is decreased.

□ **End of chapter.**

Chapter 7

Conclusion

In this thesis, we proposed a data mining proxy to support a large number of users' mining queries, and focused on how to build the smallest but sufficient trees in memory efficiently for mining.

The data mining proxy maintains the trees for different requests in memory and PP_D-trees resident in the hard disk. To utilize the trees in memory and on disk, three tree operations were proposed: sub-projection, super-projection and merge. A new bitmap coding scheme was also proposed to facilitate loading a subtree from disk and constructing a new tree in memory. Comparing to the previous non-bitmap coding scheme, it can be easily compressed so that the space cost on disk and in memory is largely reduced. Some algorithms are designed to efficiently implement all these tree operations. The advantages of tree structure and bit-map codes are fully utilized so as to minimize the time cost of the operations.

The advantage of the proxy is that we can maximize the usage of trees in memory, and minimize the I/O costs. The experiments shows that the proxy

largely reduced the cost of disk access. For instance, we only need to project the lower half tree to support frequent super-itemset mining queries because we can reconstruct the higher half in memory systematically. Several replacement strategies were also considered.

We conducted extensive testing. Our experiments showed that the data mining proxy is effective because in-memory tree operations can be processed much faster than loading subtrees from disk.

The privacy and security issues are also considered. Our method to preserve privacy fully utilize the advantages of trees structure so as to minimize the requirement of the information to be revealed. Anymore, the cost of the communication is also largely reduced. The method also guarantee that the final mining results are the same as those without the privacy preserving measures.

□ End of chapter.

Bibliography

[1] R. C. Agarwal, C. C. Aggarwal, and V. V. V. Prasad. Depth first generation of long patterns. In *Proceedings of 6th ACM SIGKDD Int. Conf. on Knowledge discovery and data mining*, pages 108–118, 2001.

[2] R. C. Agarwal, C. C. Aggarwal, and V. V. V. Prasad. A tree projection algorithm for generation of frequent item sets. *Journal of Parallel and Distributed Computing*, 61:350–371, 2001.

[3] R. Agrawal and R. Srikant. Fast algorithms for mining association rules. In *Proceedings of 20th Int. Conf. Very Large Data Bases*, pages 487–499, 1994.

[4] R. J. Bayardo. Efficiently mining long patterns from databases. In *Proceedigns of 1998 ACM SIGMOD Intl. Conference on Management of Data*, pages 85–93, 1998.

[5] D. Burdick, M. Calimlim, and J. Gehrke. MAFIA: A maximal frequent itemset algorithm for transactional databases. In *Proceedings of 2001 Intl. Conference on Data Engineering (ICDE'01)*, pages 443–452, 04 2001.

[6] Cheung, Han, Ng, Fu, and Fu. A fast distributed algorithm for mining association rules. In *PDIS: International Conference on Parallel and Distributed Information Systems*. IEEE Computer Society Technical Committee on Data Engineering, and ACM SIGMOD, 1996.

[7] O. Goldreich. Secure multi-party computation. Working Draft, Mar. 2000. Version 1.2, 108 pages.

[8] K. Gouda and M. J. Zaki. Efficiently mining maximal frequent itemsets. In *Proceedings of the 2001 IEEE ICDM*, pages 163–170. IEEE Computer Society, November 2001.

[9] J. Han, J. Pei, and Y. Yin. Mining frequent patterns without candidate generation. In *Proceedings of the 2000 ACM SIGMOD Conference*, pages 1–12, 2000.

[10] J. Han, J. Wang, Y. Lu, and P. Tzvethov. Mining top-k frequent closed patterns without minimum.

[11] M. Kantarcioglu and C. Clifton. Privacy-preserving distributed mining of association rules on horizontally partitioned data, 2002.

[12] D.-I. Lin and Z. M. Kedem. Pincer search: A new algorithm for discovering the maximum frequent set. In *Proc. of 6th Intl. Conference on Extending Database Technology, EDBT*, pages 105–119, 1998.

[13] G. Liu, H. Lu, W. Lou, Y. Xu, and J. X. Yu. Efficient mining of frequent itemsets using ascending frequency ordered prefix-tree. In *Proceedings of DASFAA'03*, pages 65–72, 2003.

[14] J. Liu, Y. Pan, K. Wang, and J. Han. Mining frequent item sets by opportunistic projection. In *Proceedings of the 8th KDD Conference*, 2002.

[15] S. R. Oliveira, O. R. Zaiane, and Y. Saygin. Secure association rule sharing. In *PAKDD 2004: Advances in Knowledge Discovery and Data Mining*, pages 74–85.

[16] J. Pei, J. Han, H. Lu, S. Nishio, and D. Y. S hiwei Tang. H-mine: Hyper-structure mining of frequent patterns in large databases. In *Proceedings of 2001 IEEE Conference on Data Mining*, 2001.

[17] J. Pei, J. Han, and R. Mao. Closet: An efficient algorithm for mining frequent closed itemsets. In *ACM SIGMOD Workshop on Research Issues in Data Mining and Knowledge Discovery*, pages 21–30, 2000.

[18] S. C. Pohlig and M. E. Hellman. An improved algorithm for computing logarithms over GF(p) and its cryptographic significance. *IEEE Transactions on Information Theory*, 24:106–110, 1978.

[19] R. L. Rivest, A. Shamir, and L. M. Adelman. A method for obtaining digital signatures and public-key cryptosystems. Technical Report MIT/LCS/TM-82, 1977.

[20] S. Rizvi and J. R. Haritsa. Maintaining data privacy in association rule mining. In P. A. Bernstein et al., editors, *VLDB 2002: proceedings of the Twenty-Eighth International Conference on Very Large Data Bases, Hong Kong SAR, China, 20–23 August 2002*, pages 682–693, Los Altos, CA 94022, USA, 2002. Morgan Kaufmann Publishers.

[21] M. Seno and G. Karypis. Lpminer: An algorithm for finding frequent itemsets using length-decreasing support constraint. In *1st IEEE Conference on Data Mining*, 2001.

[22] P. Shenoy, J. R. Haritsa, S. Sudarshan, G. Bhalotia, M. Bawa, and D. Shah. Turbo-charging vertical mining of large databases. In *2000 ACM SIGMOD Intl. Conference on Management of Data*, pages 22–33. ACM Press, 05 2000.

[23] J. Wang, J. Han, and J. Pei. CLOSET+: searching for the best strategies for mining frequent closed itemsets. In P. Domingos, C. Faloutsos, T. SEnator, H. Kargupta, and L. Getoor, editors, *Proceedings of the ninth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD-03)*, pages 236–245, New York, Aug. 24–27 2003. ACM Press.

[24] G. I. Webb. Efficient search for association rules. In R. Ramakrishnan, S. Stolfo, R. Bayardo, and I. Parsa, editors, *Proceedinmgs of the 6th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD-00)*, pages 99–107, N. Y., Aug. 20–23 2000. ACM Press.

[25] Y. Xu, J. X. Yu, G. Liu, and H. Lu. From path tree to frequent patterns: A framework for mining frequent patterns. In *Proceedings of IEEE ICDM'02*, pages 514–521, 2002.

[26] X. Yan, J. Han, and R. Afshar. Clospan: Mining closed sequential patterns in large datasets.

[27] M. J. Zaki and C. Hsiao. Charm: An efficient algorithm for closed association rule mining. Technical report, Rensselaer Polytechnic Institute, 1999.