Techniques in Data Mining: Decision Trees Classification and Constraint-based Itemsets Mining

CHEUNG, Yin-ling

A Thesis Submitted in Partial Fulfillment
of the Requirements for the Degree of
Master of Philosophy
in
Department of Computer Science & Engineering

Techniques in Data Mining: Decision Trees Classification and Constraint-based Itemsets Mining

submitted by

CHEUNG, Yin-ling

for the degree of Master of Philosophy

at the Chinese University of Hong Kong

Abstract

Classification and Association Rules Mining are two important data mining techniques. These two techniques are complements of each other. Decision trees classification is a supervised learning that requires a training dataset to develop a classifier, while itemsets mining is an unsupervised learning that requires no apriori knowledge. Both of them are essential to practical applications. In this thesis, we aim at improving these two techniques for large databases.

Classification has been widely used to assist decision making processes in various applications. Among the techniques for classification, decision tree has caught most attention recently due to its conceptual simplicity and accuracy. In the first half of this thesis, we investigate several strategies to speed up the process for building decision trees under the database oriented constraint: the main memory space is limited and usually much smaller than the dataset. Our methods for building decision trees are all based on pre-sorting. We pay particular attention to the problem of how to minimize I/O operations under the limited memory space. Our study shows that by emphasizing on different aspects such as the order of hashing, allocation of memory buffers, the amount of disk space, and the tradeoff between I/O and CPU costs, we can obtain schemes with different performance characteristics. Thus they can meet different requirements for different applications.

On the other hand, mining association rules in a large database is another important data mining problem. In particular, mining the N-most interesting itemsets is a new technique that eliminates the requirement of a suitable user-specified support threshold, which is typically hard to be set by the users. In the second part of this thesis, we propose three algorithms, *LOOPBACK*, *BOLB*, and *BOMO*, for mining the N-most interesting itemsets by variations of the FP-tree approach. Experiments show that our new methods outperform the previously proposed Itemset-Loop algorithm, and the performance of BOMO can be an order of magnitude better than the original FP-tree algorithm even with the assumption of an optimally chosen support threshold. We further investigate the problem of mining association rules in a constraint-based approach. Realistically, users may only be interested in particular itemsets or rules that contain certain items. Thus, we define the item constraints in such a way that users can specify the number of required itemsets containing different types of items. Based on BOMO, we propose the *Double FP-trees* algorithm for mining the N-most interesting itemsets with item constraints.

論文題目：數據採集技巧：決策樹信息分類與約束性項集挖掘

作者：張燕玲

學校：香港中文大學

學系：計算機科學與工程

修讀學位：哲學碩士

日期：二零零一年六月三十日

摘要：

信息分類和關聯規則挖掘是兩種重要的數據採集技巧. 這兩種技巧具互補性質. 決策樹信息分類是一種監督學習，需要一些數據來訓練出一個分類器. 集項採集則是一種非監督學習，不需要任何已知的知識. 兩者在應用上均很重要. 此論文的目的在於改善這兩種在大數據庫中的採集技巧.

信息分類已廣泛地作支援決策的應用. 由於概念簡單而且準確度高，在眾多信息分類技術中，決策樹較為人矚目. 在論文的上部分，我們將研究數種在數據量大但記憶體不足的情況下加速建立決策樹的策略. 據我們的研究，這些策略在不同的環境條件下，如散列的次序，記憶體的分配，硬碟的容量，I/O 和 CPU 的速度和價錢等，都具有不同的特質. 因此，它們可供不同應用之需要.

另方面，關聯規則挖掘是另一重要數據採集之研究. 而挖掘 N 個最頻繁的項集則是一種無須用戶提供難以設定的合適支持度閾之新技術. 我們在論文的第二部分，提議三種由 FP-tree 演變出的算法：***LOOPBACK, BOLB,*** 和 ***BOMO*** 來挖掘 N 個最頻繁的項集.
實驗證明我們的方法勝越以前的 Itemset-Loop 方法；而且 BOMO 更以十倍的速度超越用最合適支持度閾來挖掘的 FP-tree 算法.
我們將這種技術加入限制性法作更深入的研究. 由於用戶可能只對個別的項集或規則感興趣，固我們定義了 *項目約束*. 此乃由用戶表明挖掘包含不同類項的項集之數目限制. 我們以 BOMO 為基礎，提議 ***Double FP-trees*** 算法來挖掘 N 個受項目約束限制的最有趣項集.

Acknowledgment

I would like to thank my supervisor, Prof. Ada Wai-chee Fu. With her constant encouragement, warm care, specialized support, and insightful advice, I have been well-developed in database research work, teaching assistant-ship, and self-development.

I am grateful to have Prof. Man-hon Wong and Dr. Yiu-sang Moon as my thesis markers. Thanks for their valuable suggestions. In addition, I would like to thank Prof. Jia-wei Han for his graciously consent to be my external marker. Also, thank to Prof. Jiang Tang for his collaboration in the decision tree part of my research work.

It's my pleasure to thank my fellow colleagues including Chun-hing Cai, Chun-hung Cheng, Po-shan Kam, Wai-chiu Wong, Wai-ching Wong, Wai-to Chan, Men-hin Yan, Xiao-hui Yu, Ka-ka Ng, Anny Ng, Yin-hung Kuo, Philip Chi-wing Fu, all the 1013's and 1005's guys, and other friends for their help, share, and advice. They give me a colorful and memorable postgraduate life. Also, I am in debt to Mr. Hon-man Law for his technical support.

I would also like to thank my parents for their love and support.

Finally, thanks GOD for my growth with the Holy Spirit in the past one year, and the constant care, and the prays from all my brothers and sisters including Calvy Chan, Myra Chan, Sally Tam, Kevin Hung, Wing Tse, Francis Wong, Inez Ko, David Lee, Mosze Mak, Florence Leung, Kenneth Leung, Circle Leung, and Sonia Chan, etc.

Without them, I cannot enjoy my campus life to such a great extent.

"Buy the truth and do not sell it, get wisdom, discipline and understanding."

– Proverbs 23:23

Contents

Abstract ii

Acknowledgement iv

1 Introduction 1

1.1 Data Mining Techniques 1

1.1.1 Classification 1

1.1.2 Association Rules Mining 2

1.1.3 Estimation 2

1.1.4 Prediction 2

1.1.5 Clustering 2

1.1.6 Description 3

1.2 Problem Definition 3

1.3 Thesis Organization 4

I Decision Tree Classifiers 6

2 Background 7

2.1 Introduction to Classification 7

2.2 Classification Using Decision Trees 8

2.2.1 Constructing a Decision Tree 10

2.2.2 Related Work 11

3 Strategies to Enhance the Performance in Building Decision Trees 14
3.1 Introduction . . . 15
3.1.1 Related Work . . . 15
3.1.2 Post-evaluation vs Pre-evaluation of Splitting Points . . 19
3.2 Schemes to Construct Decision Trees . . . 27
3.2.1 One-to-many Hashing . . . 27
3.2.2 Many-to-one and Horizontal Hashing . . . 28
3.2.3 A Scheme using Paired Attribute Lists . . . 29
3.2.4 A Scheme using Database Replication . . . 31
3.3 Performance Analysis . . . 32
3.4 Experimental Results . . . 38
3.4.1 Performance . . . 38
3.4.2 Test 1 : Smaller Decision Tree . . . 40
3.4.3 Test 2: Bigger Decision Tree . . . 44
3.5 Conclusion . . . 47

II Mining Association Rules 48

4 Background 49
4.1 Definition . . . 49
4.2 Association Algorithms . . . 51
4.2.1 Apriori-gen . . . 51
4.2.2 Partition . . . 53
4.2.3 DIC . . . 54
4.2.4 FP-tree . . . 54
4.2.5 Vertical Data Mining . . . 58
4.3 Taxonomies of Association Rules . . . 58
4.3.1 Multi-level Association Rules . . . 58

4.3.2 Multi-dimensional Association Rules 59
4.3.3 Quantitative Association Rules 59
4.3.4 Random Sampling 60
4.3.5 Constraint-based Association Rules 60

5 Mining Association Rules without Support Thresholds 62
5.1 Introduction 63
5.1.1 Itemset-Loop 66
5.2 New Approaches 67
5.2.1 A Build-Once and Mine-Once Approach, BOMO 68
5.2.2 A Loop-back Approach, LOOPBACK 74
5.2.3 A Build-Once and Loop-Back Approach, BOLB 77
5.2.4 Discussion 77
5.3 Generalization: Varying Thresholds N_k for k-itemsets 78
5.4 Performance Evaluation 78
5.4.1 Generalization: Varying N_k for k-itemsets 84
5.4.2 Non-optimal Thresholds 84
5.4.3 Different Decrease Factors, f 85
5.5 Conclusion 87

6 Mining Interesting Itemsets with Item Constraints 88
6.1 Introduction 88
6.2 Proposed Algorithms 91
6.2.1 Single FP-tree Approach 92
6.2.2 Double FP-trees Approaches 93
6.3 Maximum Support Thresholds 102
6.4 Performance Evaluation 103
6.5 Conclusion 109

7 Conclusion 110

A Probabilistic Analysis of Hashing Schemes **112**

B Hash Functions **114**

Bibliography **117**

List of Tables

2.1 A training database in an insurance company. 10

3.1 Notations for the parameters. 34

3.2 Description of attributes for synthetic data. 39

3.3 Parameters setting for the experiment. 41

4.1 A transaction database. 55

5.1 A transaction database. 74

5.2 Parameter setting. 79

5.3 Synthetic data description. 79

5.4 Ideal thresholds, ξ_{opt} (%), for different datasets with $k_{max} = 4$. . 80

5.5 Ideal thresholds, ξ_{opt} (%), for different datasets with $k_{max} = 10$. 80

5.6 Execution time (sec) with different N for different k. 84

6.1 Partition of items into different bins. 91

6.2 Item constraints. 91

6.3 A transaction database. 94

6.4 Elements in a constraint FP-tree node. 94

6.5 Constraints for synthetic data in the closed interpretation. 104

6.6 Synthetic data, closed interpretation. 105

6.7 Total number of itemsets mined. 108

List of Figures

2.1 A decision tree. 11

3.1 Attribute list and count matrix. 16
3.2 Scheme 0 (SPRINT). 16
3.3 SPRINT and the One-to-Many Hashing. 19
3.4 Pre-evaluation. 26
3.5 Horizontal Many-to-One Hashing. 29
3.6 Splitting attribute by attribute pairing. 31
3.7 Splitting attribute using replicated datasets. 33
3.8 Classification functions for synthetic data. 39
3.9 Dataset size=10M, total buffer size=5MB, load factor=80%, using Function 2. 42
3.10 Total buffer size=5MB, load factor=80%, using Function 2. 43
3.11 Total buffer size=5MB, load factor=80%, using Function 6. 44
3.12 Total buffer size=5MB, dataset size=10M, using Function 2. . . . 46

4.1 The Apriori Algorithm. 51
4.2 An example using Apriori-gen with threshold = 2. 52
4.3 FP-tree construction. 55
4.4 FP-tree. 56
4.5 Algorithm for mining frequent patterns using FP-tree. 57

5.1 Two correlated variables X and Y. 65

5.2 An example using Itemset-Loop. 67
5.3 NFP-tree algorithm for mining N-most interesting itemsets. . . 71
5.4 NFP-tree construction. 72
5.5 Algorithm for mining phase of N-most interesting itemsets. . . . 72
5.6 FP-tree. 73
5.7 FP-tree after round 2. 76
5.8 Real dataset: tiny.dat. 81
5.9 Real dataset: small.dat. 82
5.10 Synthetic datasets with $k_{max} = 4$. 83
5.11 Synthetic datasets with $k_{max} = 10$. 83
5.12 Small thresholds, synthetic dataset: T5.I2.D100K. 85
5.13 Large thresholds, synthetic dataset: T.5.I2.D100K. 86
5.14 Different decrease rate, synthetic datasets. 86

6.1 An FP-tree for a set of transactions. 94
6.2 An FP-tree for a set of constraints. 95
6.3 DFP-tree algorithm for mining N-most interesting itemsets with constraints. 99
6.4 Construction of constraint FP-tree. 99
6.5 Algorithm mining N-most interesting itemsets with constraints. 100
6.6 Matching itemset with constraints in constraint FP-tree. 100
6.7 Real dataset. 105
6.8 Real dataset, uniform $N = 5$. 106
6.9 Synthetic dataset, uniform $N = 5, 10, ..., 30$. 107
6.10 Synthetic dataset, $N = 5$, different k_{max}. 107

B.1 Hash table for coalesced chaining. 114
B.2 Resulting content of the hash table. 115

Chapter 1

Introduction

Data mining, which is also known as *knowledge discovery in databases*, is the exploration and analysis of large datasets to discover meaningful patterns and rules.

Nowadays, data mining is an important technology for business data processing. It is necessary to develop tools and techniques to retrieve useful and important information in an efficient, accurate, and user controllable way.

1.1 Data Mining Techniques

There are various interesting areas in data mining and we need different tools for studying and handling different areas. In this section, we introduce six kinds of data mining problems [9] including classification, association rules mining, estimation, prediction, clustering, and description.

1.1.1 Classification

Classification, also known as supervised learning, is a function that identifies a new object as belonging to one of several predefined classes based on a set of preclassified examples. For example: classify natural objects into different kingdoms such as animal, plant, and mineral.

1.1.2 Association Rules Mining

In market basket analysis, it is valuable to explore interesting associations between different items, e.g. Is it true that most of the customers who buy bread also buy milk? We can interpret this association using the rule: $buys(bread) \Rightarrow buys(butter)$. Mining association rules is to find such interesting rules according to some user defined thresholds such as the support of a rule, which is the fraction of the transactions in the database that contain all the items in the rule.

1.1.3 Estimation

While classification deals with discrete outcomes, estimation is a task to deal with continuously valued outcomes. Each record can then be ranked into one of the categories.

1.1.4 Prediction

Prediction is different from classification and estimation in a way that objects are classified according to some predicted future behaviors or estimated future values. Historical data are used to build models for predicting the future behavior.

1.1.5 Clustering

Clustering is an unsupervised task to identify classes for a set of unclassified objects according to some similarity measurement of their attributes. It is different from classification as there is no predefined set of classes.

1.1.6 Description

Description is a task of describing what happens in a complicated database. It helps users to understand people, products or processes in the database. It gives an explanation for the behavior of them.

1.2 Problem Definition

From business-oriented applications to personal-computer-based web users, mining interesting information from large databases is of paramount importance. However, it is usual that users are in general not experts in controlling the mining process. It is important that the data mining tools have to be user-friendly and easy to control. Another problem in data mining is the time efficiency in retrieving a small amount of useful information from very large databases. In this thesis, we handle these two problems by investigating two of the data mining techniques: classification and association rules mining. Both of them are NP-Complete [25, 61], but they are complements of each other and are essential to practical applications.

In Part I, we study the problem of classification. In particular, we focus on the decision tree classifier. To extract useful information from the increasing size of databases, we propose several strategies for building decision trees in a disk-based approach. We attempt to preserve the quality of the decision trees while minimizing the I/O operations under the assumption of limited amount of main memory space. This avoids the I/O bottleneck to inhibit the data mining.

In Part II, we pay attention to the problem of setting support threshold for conventional association rules mining. Since setting such threshold is non-user-friendly, difficult and indirect, finding frequent itemsets based on the support threshold is not practical. We propose new algorithms for mining N k-itemsets with the highest supports for k up to a certain value. We further handle the problem in a constraint-based approach so that we can mine N k-itemsets with the highest supports while at the same time, each of these itemsets satisfies certain conditions specified by the users. This problem re-definition is important since, realistically, users would like to focus on only certain types of interesting patterns.

With these two investigations, we are able to efficiently handle situations which need both the classification and the association rules mining techniques. This comes to the case in query by examples when we want to find the most interesting patterns or rules for all objects similar to the given object. We handle this problem by first classifying this object, and then using our proposed constraint-based association rules mining approach to find the interesting patterns for all the similar objects. We believe that this combined approach is valuable for many applications.

1.3 Thesis Organization

There are six chapters following this introduction chapter. Part I, classification, consists of two chapters while Part II, association rules mining, consists of 3 chapters.

Chapter 2 gives an introduction to the classification problem, and we particularly focus on decision tree classifiers. In Chapter 3, we propose several schemes for building decision tree classifiers for large databases in which the main memory space is limited. All these schemes benefit from the pre-evaluation technique for finding splitting points, and they provide different performance characteristics under different aspects, e.g. the order of hashing, allocation of buckets, the amount of disk space, and the tradeoff between I/O and CPU costs, etc.

A survey on association rules mining is given in Chapter 4. We study some of the state-of-the-art algorithms and the variety of association rules mining.

We then present our algorithms for mining large itemsets without support thresholds in Chapter 5. In Chapter 6, we further extend this technique for mining constraint-based association rules. We propose algorithms for mining large itemsets with item constraints. Finally, we draw the conclusion in Chapter 7.

Part I

Decision Tree Classifiers

Chapter 2

Background

Classification is a function that identifies a new object as belonging to one of several predefined classes. Since late 70's, it has been used to assist in decision making process in a variety of applications such as medical diagnosis, credit approval, weather prediction, etc. It has emerged now as an important branch of data mining in databases. Among the techniques for classification, decision tree has caught most attention recently due to its conceptual simplicity and accuracy. In this chapter, we give an introduction to decision tree classifiers and the recent work for this area.

2.1 Introduction to Classification

Since late 70's, *classification* has been used to assist in decision making process in a variety of applications such as medical diagnosis, credit approval, weather prediction, etc. It was not however until the 90's that data mining became an important discipline and classification emerged as one of the major branches in data mining. Early applications of classification were restricted only to small datasets, which were assumed to fit into main memories. Thus accuracy was the only concern. With the rapid growth of the computer's capability to collect and store large amounts of data, assuming the entire dataset to fit into the main memory is no longer realistic. This results in new issues such as fast classification, scalability, etc.

Simply put, classification is a function that identifies a new object as belonging to one of several predefined classes. It is performed using a tool called the *classifier*. A classifier is constructed through a learning process which uses as input a collection of sample objects (called the *training set*). An object is associated with a number of attributes, among which one is termed a **target** and the rest are called **predictors**. The domain of the target attribute consists of the class labels. During the learning process, both the values of the predictors and that of the target attributes for the sample objects are known. The classifier encodes the statistical dependencies between the target attribute and the predictors of the sample object set, and is then used to predict the target attribute value of a new object from its predictors values.

Several kinds of classifiers were proposed in the past [13, 46, 19, 10, 58]. Among them *decision tree* has caught most attention recently due to its simplicity, conceptual cleanness, and accuracy [10, 2, 41, 42, 43]. On the other hand, other methods like neural networks may require extremely long training times even for small datasets, and cannot explicitly return the characteristics of classes.

2.2 Classification Using Decision Trees

A decision tree is usually constructed in two phases, *growing* and *pruning* [10, 44]. In the growing phase, children of the nodes are generated and thus the tree grows. This process continues until the leaf nodes become "pure" (see the discussion in Section 2.1.). In the pruning phase, selected nodes are pruned bottom up, starting from the leaf level. It requires less computational time than the building phase, This pruning step is used to alleviate the impact caused by statistic irregularities. The mostly used technique for the pruning phase is based on minimum description length principle (MDL) [44, 15, 33].

Let t be a record. Let $X_1, \cdots, X_m$ be the predictor attributes and C be the target attribute of t. The domain of X_i, denoted as $dom(X_i)$, can be either *continuous* or *categorical*. In the former case, the attribute values are ordered and in the latter case, they are not. The domain of C is a set of class labels. A decision tree can be viewed as a function $dom(X_1) \times \cdots \times dom(X_m) \to dom(C)$. Each node in a decision tree is associated with a set of records. Let N be a node in the decision tree and $S(N)$ its associated set. Let D be the original training set. If N is the root, $S(N) = D$. Let $N_1, \cdots, N_k$ be the children of node P. We have $\cup_{i=1}^{k} S(N_i) = S(P)$.

Each node in a decision tree is associated with a set of predicates. A predicate is defined as $p(r) = \langle r.X_1, \cdots, r.X_m \rangle \in T$ where r is a record variable and $T \subseteq dom(X_1) \times \ldots \times dom(X_m)$. It is required that for any internal node N and any record $r \in S(N)$ exactly one predicate in the predicate set for N is true. Let $PRE(N)$ be the predicate set for N. We generate $\mid PRE(N) \mid$ children for N, each of which corresponds to a predicate in $PRE(N)$. Let $p_i \in PRE(N)$. N_i is $N's$ child corresponding to p_i if and only if $S(N_i) = \{r : r \in S(N) \ \& \ p_i(r) = true\}$. The predicates for the nodes in the decision trees is selected in such a way that each leaf node L is pure, namely, all the records in $S(L)$ belong to the same class (i.e, have the same value for the target attribute.)

One of the main issues in the construction of decision trees is how to select the set of the predicates for each node. To simplify our discussion, let us consider the case where each node is associated with two predicates, given by $p(r)$ and $\neg p(r)$, resulting in a binary decision tree. The ideas can be easily extended to non-binary cases. For any internal node N, let $p(r)$ be the predicate for it and N_L and N_R be its left child and right child, respectively. Then $S(N_L) = \{r : r \in S(N) \ \& \ p(r) = true\}$ and $S(N_R) = \{r : r \in S(N) \ \& \ p(r) = false\}$.

In addition, in most applications only a simplified form of predicate is used, that is, $p(r) = \langle r.X_i \rangle \in T$ where $1 \leq i \leq m$ and $T \subseteq dom(X_i)$. These

Table 2.1: A training database in an insurance company.

Age	Marriage_Status	Years_Driving	Risk_Class	Rid
40	M	15	L	1
25	S	3	H	2
55	M	30	L	3
55	M	35	L	4
20	S	2	H	5
26	M	1	L	6
65	M	40	H	7
70	S	20	H	8

assumptions are used to ease the task of constructing decision trees. Most applications have shown that the accuracy of decision trees can be satisfactory with these assumptions. In the following, we consider the decision trees only under these assumptions. In Figure 2.1, a decision tree is shown for the sample training set in Table 2.1.

2.2.1 Constructing a Decision Tree

A decision tree is constructed in an top down fashion. At each node, an attribute is chosen to 'split' the data set associated with that node. Let N be an internal node, and X be a continuous attribute. Each value x of X is a potential *splitting point* that partitions $S(N)$ into two sets: $S_L = \{r : r \in S(N) \; \& \; r.X \leq x\}$ and $S_R = \{r : r \in S(N) \; \& \; r.X > x\}$. If X is a categorical attribute, each subset B of its values is a potential splitting point: $S_L = \{r : r \in S(N) \; \& \; r.X \in B\}$ and $S_R = \{r : r \in S(N) \; \& \; r.X \notin B\}$. To determine the actual splitting point among the potential splitting points for all the attributes, a criterion *Gini Index* is used. For any node N, suppose that in $S(N)$, the fraction of data records belonging to class i is given by p_i. The Gini Index of N is defined as $\text{Gini}(S(N)) = 1 - \sum p_i^2$. If a splitting point partitions the set $S(N)$ into S_L and S_R, then the **Gini Index of N for split** at that splitting

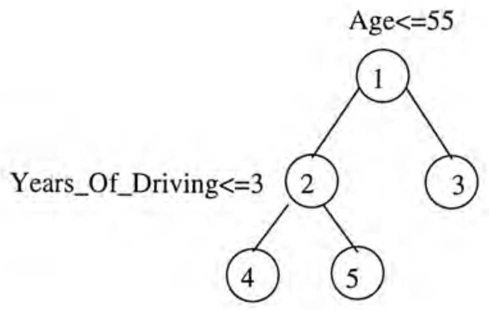


Figure 2.1: A decision tree.

point is defined as

$$Gini_{split}(S(N)) = \frac{| S_L |}{| S(N) |} Gini(S_L) + \frac{| S_R |}{| S(N) |} Gini(S_R). \qquad (2.1)$$

Among all the potential splitting points for all the attributes, the one with the lowest Gini Index for split is the best splitting point, called *final splitting point*. We call the attribute for which the final splitting point is chosen *splitting attribute* and other attributes *non-splitting attribute*.

Other than the Gini index, entropy is also a popular measurement for deciding on the splitting point.

2.2.2 Related Work

Almost all the work proposed for building decision trees sort attribute values. The methods in [10, 43] sort the data at every node, which would cause high computation overhead if the dataset is large. SLIQ [32] improves over this repeated sorting by employing a *pre-sorting* method, which sorts the attribute values only once, i.e., at the root. The advantages of pre-sorting attributes are twofold. First, it maintains the pre-sorted order of the attribute values without incurring sorting related overhead at each node. Thus the performance is improved. Second, compared with other methods, the input data it creates at each node is less sensitive in size to the inter-dependencies between different attributes. Thus its performance is relatively stable under different datasets with different data distributions.

Many attempts have been made to improve the performance over early pre-sorting algorithms. The weakness of the scheme proposed in [32] is its requirement for some data structure (called 'class lists') to be memory resident all the time. This has been improved by SPRINT [48], where no permanent residence in memory is required of any data structure. SPRINT separates 'attribute lists' from datasets and uses hashing to join them together if necessary. The work in [35] reduces the amount of computations by exploring the convexity property of impurity function used to split attribute lists. It is aimed at handling large categorical attributes. In CLOUDS and CMP [8, 55], the authors propose schemes to reduce the number of split points that must be evaluated in order to find the best split point for an attribute list. However, [8] introduces an extra scanning of the data set and [55] introduces restrictions which may introduce inaccuracy in the result.

An assumption made by many recent work is that some structures whose sizes depend on the datasets can fit into the main memory. For example, in [18] the authors propose a method called 'Rainforest'[1]. It uses a data structure called the AVC-set, instead of the attribute list. An AVC-set in some sense is a summary table, which groups entries with the same attribute value into one entry, recording the count. Therefore, the number of entries in the AVC-set is the number of distinct attribute values in the attribute list. The AVC-set is usually smaller than an attribute list, and therefore has a better chance to fit in memory. However, in this scheme the dataset must be scanned at every node to extract necessary information. The AVC-sets must be sorted at every node. This can be costly if the AVC-sets are large. Moreover, the chance that AVC-sets does not fit in the memory still exists.

In some other work, the main memory resident data is an sampling of the

[1]Rainforest is not basted on pre-sorting.

dataset. In BOAT [17], an optimistic approach is used, in which a sampling of the dataset will give us a solution that is the correct solution with high probability. However, the chance still exists when the estimated solution is not correct and we are back to the original problem. To reduce the cost, [35] uses a randomized sampling to evaluate the split. The work in [17] first built the tree using a small sample and then refine it. The performance of these schemes, therefore, depend on the samples selected. The work in [45] uses a different approach to attacking efficiency problem: it integrates the growing phase and the pruning phase. We observe that this technique is orthogonal to the technique used specifically for the growing phase.

Chapter 3

Strategies to Enhance the Performance in Building Decision Trees

Classification using decision tree is an important data mining technique. One class of methods for building a decision tree pre-sort the attribute values for each attribute. As the decision tree grows, the attribute values will be distributed (recursively) to each node in such a way that their relative orders are preserved. The advantages of this pre-sorting methods are twofold. First, it maintains the pre-sorted order of the attribute values without incurring sorting-related overhead at each node. Second, compared with other methods, the input data set it creates at each node is less sensitive in size to the interdependencies between different attributes. Thus, its performance is relatively stable under different data distributions. In this chapter, we study several strategies for pre-sorting based methods in building decision trees under the database oriented constraint: the main memory space is limited and is smaller than the dataset. We pay particular attention to the problem of how to minimize the I/O operations under the limited memory space. Our study also shows that by emphasizing on different aspects, we can obtain schemes with different performance characteristics. Thus, they can meet different requirements of different applications.

3.1 Introduction

In this chapter, we study strategies to speed up the pre-sorting based approaches for building decision trees under the database oriented constraint: the main memory space is limited, and is much smaller than the dataset size. We make no assumption that the main memory can hold any dataset dependent on the structures of the original dataset for the major evaluation steps. Also, we do not introduce any restriction that may compromise the accuracy. Therefore we provide absolute improvements on previous methods with no trade-off. Since in general the growing phase dominates the performance, we consider the growing phase only.

In our schemes, we pay special attention to the problem of how to minimize the I/O operations under limited main memory space. We use a technique, called 'pre-evaluation', whereby split points can be evaluated for an attribute list while the attribute list is being generated, instead of evaluating it after generation. This virtually reduces the amount of I/O operations required for evaluating split points to zero. Our study also shows that by emphasizing on different aspects, one can obtain schemes with different performance characteristics. Thus they can meet different requirements of different applications.

3.1.1 Related Work

One approach for constructing a decision tree for large dataset is proposed in SPRINT [48], which is based on the use of attribute lists, one for each predictor attribute. An **attribute list** for attribute X at node N is a projection of dataset $S(N)$ on X, C, and *Rid*. If X is a continuous attribute, then the attribute list is ordered by the values of X. Figure 3.1(a) is the attribute list

Age	Risk_Classes	Rid
20	H	5
25	H	2
26	L	6
40	L	1
55	L	3
55	L	4
65	H	7
70	H	8

(a)

$$\begin{array}{c c} & \begin{array}{cc} L & H \end{array} \\ \begin{array}{c} M \\ S \end{array} & \begin{pmatrix} 4 & 1 \\ 0 & 3 \end{pmatrix} \end{array}$$

(b)

Figure 3.1: Attribute list and count matrix.

```
Preprocessing: construct a set $A$ of sorted attribute lists.
BuildTree(Node $N$, AttriSet $A$)
0. If all data in $N$ are of the same class then return.
1. Further splitting is necessary: find splitting attribute list $L$ and splitting point $v$.
2. Generate $N_L$ and $N_R$ as the left child and right child of $N$;
3. Split($A, L, v, A_L, A_R$);
4. BuildTree($N_L$,$A_L$);
5. BuildTree($N_R$,$A_R$).
```

Figure 3.2: Scheme 0 (SPRINT).

for attribute *Age* in Table 2.1. A framework for this approach is shown in Figure 3.2. In the preprocessing, attribute lists are created for both the splitting attribute and non-splitting attributes. Then BuildTree(N,A) is called, where parameter A is the set of constructed attribute lists. Further splitting is necessary if all records in $S(N)$ do not have a unique class label. In this case we search for the best splitting point in any attribute list.

Let L be an attribute list for attribute X. Assume X is a continuous attribute and $x \in X$. Let $S_x = \{r : r.X \leq x \& r \in L\}$ and $R_x = \{r : r.X > x \& r \in L\}$. Then the Gini Index at x is

$$Gini_x = \frac{|S_x|}{|L|} Gini(S_x) + \frac{|R_x|}{|L|} Gini(R_x) \tag{3.1}$$

To evaluate all the splitting points for X, we scan L in a top-down fashion.

Suppose r and e are two entries, we say that $r \preceq e$ if and only if the value of the splitting attribute in r is less than or equal to that of e. For each scanned entry e, and for any class label i, the count $y_i = \mid \{r : r \in S(N) \& r \preceq e \& r.C = i\} \mid$ is accumulated. These counts can be used to calculate the Gini Index for each splitting point of X [1].

On the other hand, consider X as a categorical attribute and $x \subseteq X$. Let $U_x = \{r : r.X \in x \& r \in L\}$ and $V_x = \{r : r.X \notin x \& r \in L\}$. Then the Gini Index at x is

$$Gini_x = \frac{\mid U_x \mid}{\mid L \mid} Gini(U_x) + \frac{\mid V_x \mid}{\mid L \mid} Gini(V_x) \tag{3.2}$$

In SPRINT, a *count matrix* is constructed when the attribute list is scanned. Each row in the count matrix corresponds to a value in the domain of the attribute and each column corresponds to a class label of the target attribute. An entry in the matrix indicates the count of records with the class label for the column and the value of the attribute for the row. Figure 3.1(b) shows the count matrix for attribute *Marriage_Status*. After the count matrix is constructed, the Gini Index can be calculated for each possible partition of the attribute domain.

In either case, we can determine the best splitting point for each attribute list. Among all these splitting points, we choose the best one as the final splitting point and the associated attribute list as the splitting attribute list. Once the splitting attribute and the final splitting point are chosen, every attribute list can be split accordingly, with one portion going to N_L and the other to N_R. This task is carried out by the subroutine Split.

In the subroutine Split, we know the splitting point of the splitting attribute. Let L be the splitting attribute list. It will be split into L_L belonging to N_L and L_R belonging to N_R. This splitting is done by simply scanning the

[1] In SPRINT, these counts are stored and updated incrementally in a data structure called 'histogram'.

entries in L top down. For each entry scanned, we compare the attribute value it contains with the splitting point, and determine its destination (i.e., either L_L or L_R) immediately. After this splitting, we then split the non-splitting attribute lists. We load either L_L or L_R into the main memory. (For performance reason, we should bring in the shorter one). Without loss of generality, we assume L_L is shorter and we bring in L_L. Then we can bring in non-splitting attribute lists L_N one by one to do the splitting on them. For each record in L_N, we see if it exists in L_L by matching the record id (*rid*). If so, it belongs to N_L, else it belongs to N_R. To reduce the search time, we build a hash table for L_L using the *rid* as the key. Each record in L_N can then "probe" the hash table using the *rid* value.

If an attribute list cannot fit into the main memory, the attribute list is divided into **buckets**. The buckets are brought in one at a time to the main memory. For the splitting attribute, each time a bucket X is brought in from L_L, a hash table is built for the bucket only. With the hash table in the main memory, we bring in every bucket from each non-splitting attribute list one by one, and do the probing of the hash table to determine whether a record should go to N_L or N_R. Then we repeat the entire process with the next bucket in L_L. After the last bucket of L_L is brought into the memory, all the entries in each bucket of each non-splitting attribute list have their destinations. This essentially splits each non-splitting attribute list into two sub-lists, one for N_L and the other for N_R. A diagram for this scheme is shown in Figure 3.3.

The rest of this chapter is organized as follows. In Section 3.1.2, we introduce the pre-evaluation technique. In Section 3.2, we present several strategies for building decision trees. In Section 3.3, we analyze the performance for these strategies. We conclude the chapter by summarizing the main results in Section 3.4.

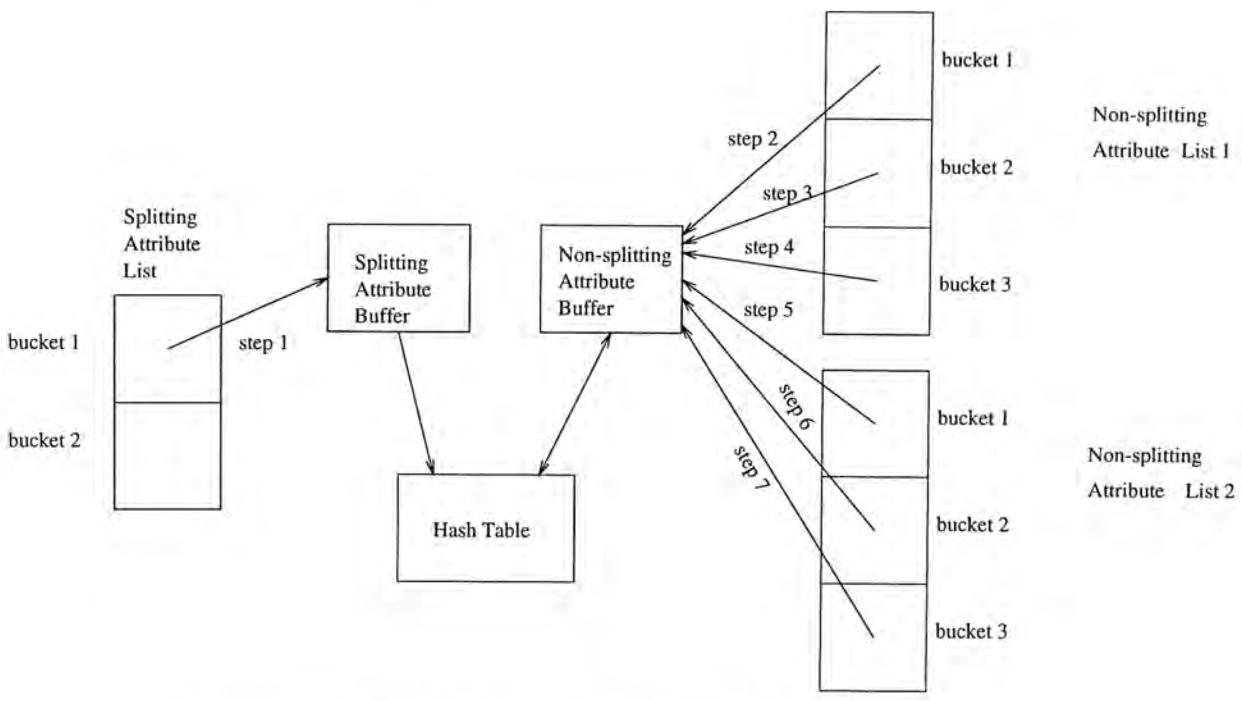


Figure 3.3: SPRINT and the One-to-Many Hashing.

3.1.2 Post-evaluation vs Pre-evaluation of Splitting Points

In the SPRINT framework, splitting points are evaluated after the attribute lists have been constructed, as indicated by the order of the related statements (i.e., preprocessing precedes step 1, and step 2 precedes step 3). We call this way of evaluating splitting points **post-evaluation**. The post-evaluation scheme has a negative impact on the performance. This is because if the attribute lists cannot fit into the main memory, they have to be written to disk, and I/O operations are needed for them to be fetched into the memory for splitting point evaluation.

Is it possible to evaluate splitting points without incurring extra I/O operations? For each attribute list, the required data such as the counts for class labels to evaluate the splitting point is obtained by accumulating those at the previous relevant entries. Therefore, the evaluation is possible only when all those entries have been generated. Those entries are, for a continuous attribute all the entries preceding the one with the splitting point value, and for a categorical attribute all the entries for each value in the splitting point set.

To make the evaluation possible, the counts for the entries that were already over-written must be kept in the memory when the current splitting point is being evaluated. We shall now give a more formal description of the idea. In the following discussion, we assume that any attribute list, starts from an (imaginary) empty entry ϵ, is always memory resident. We assume that ϵ is the 0-th entry in any attribute list.

Definition 1 Let L be an attribute list for attribute X and e be an entry in L. Let $c \in dom(C)$ be a class label. Then e is a *statistic point* at time t if

1. $e = \epsilon$, or
2. there exist $f \prec_L e$ and $t' \leq t$ such that f is a statistic point at t' and every entry g with $f \prec_L g \preceq_L e$ is in the memory at t [2].

If the second condition is true, we say that f is an *S-predecessor* for e (S for **Statistics).** ∎

Our intention here is to be able to collect incrementally various counts up to the statistic point using the information only residing in the memory. These counts are necessary to calculate the Gini Index for a splitting point.

Definition 2 Let L be an attribute list for a continuous attribute X and e be a statistic point in L at time t. Let $c \in dom(C)$ be a class label. Then the *statistics on c up to e*, denoted as $S_e(c)$, is

- if $e = \epsilon$, then $S_e(c) = 0$
- otherwise, let f be an S-predecessor of e, then $S_e(c) = S_f(c) + \mid \{g : f \prec_L g \preceq_L e \& g.C = c\} \mid$.

∎

[2]The statement 'Entry e of L is in memory' implies that not only e is physically in the memory but also it is known to belong to L.

Definition 3 Let L be an attribute list for a categorical attribute X and e be a statistic point in L at time t. Let $c \in dom(C)$ and $x \in dom(X)$. Then the *statistics on x and c up to e*, denoted as $S_e(x, c)$, is

- if $e = \epsilon$, then $S_e(x, c) = 0$
- otherwise, let f be an S-predecessor of e, then $S_e(x, c) = S_f(x, c)+ \mid \{g : f \prec_L g \preceq_L e \& g.X = x \& g.C = c\} \mid$.

∎

It is easy to see that statistics up to a statistic point at time t can be calculated using the data only residing in memory, as long as (recursively) the statistics of its S-predecessor is in the memory at that time. The following theorem shows that the statistics provides the exact information we need to calculate Gini Indexes.

Theorem 1 Let L, X, c, e and $S_e(c)$ be defined in the same way as in Definition 2. Then, $S_e(c) =\mid \{g : g \preceq_L e \& g.C = c\} \mid$.

Proof: Suppose e is the i-th entry in L. We prove the theorem by induction on i.

Base: $i = 0$. Thus, $e = \epsilon$. By the definition of statistics, $S_e(c) = 0$. On the other hand, $\mid \{g : g \preceq_L e \& g.C = c\} \mid= 0$.

Inductive step: $i > 0$. There exists a j, $0 \leq j < i$, such that the j-th entry, say f, is a statistic point and $S_e(c) = S_f(c)+ \mid \{g : f \prec_L g \preceq_L e \& g.C = c\} \mid$. By induction hypothesis, $S_f(c) =\mid \{g : g \preceq_L f \& g.C = c\} \mid$. Combining the last two equalities yields the desired value for $S_e(c)$. ∎

Theorem 2 Let L, X, x, c, e and $S_e(x, c)$ be defined in the same way as in Definition 3. Then, $S_e(x, c) =\mid \{g : g \preceq_L e \& g.X = x \& g.C = c\} \mid$.

Proof: Similar to the proof for Theorem 1, except that $S_e(c), S_f(c), \mid \{g : f \prec_L g \preceq_L e \& g.C = c\} \mid$, and $\mid \{g : g \preceq_L f \& g.C = c\} \mid$ are substituted, respectively, by $S_e(x,c)$, $S_f(x,c)$, $\mid \{g : f \prec_L g \preceq_L e \& g.X = x \& g.C = c\} \mid$ and $\mid \{g : g \preceq_L f \& g.X = x \& g.C = c\} \mid$. ∎

The two theorems state that the statistics up to a statistic point is the count of certain entries up to that point. We observe that these counts are all we need to calculate the Gini Indexes. For a continuous attribute X and an arbitrary value $x \in dom(X)$, if e is the last entry in L such that $e.X = x$, then the collection of $S_e(c)$ on all $c \in dom(C)$ enables us to calculate the Gini Index at splitting point x. For a categorical attribute X and an arbitrary set $y \subseteq dom(X)$, if e is the last entry in L, then the collection of $S_e(x,c)$ on all $x \in dom(X)$ and $c \in dom(C)$ enables us to calculate the Gini Index at splitting point y. Note however that our interest is not to just evaluate any particular splitting point. Rather, we need to evaluate all the splitting points in an attribute list among which we will choose the best. This is possible if every entry becomes a statistic point at the same point in time. Furthermore, for a continuous attribute list, whose domain may contain a large number of values, we would like the preceding entries to become statistic points earlier than the succeeding entries, so that their Gini Indexes can be compared incrementally. (For a categorical attribute list, we do not have a choice but selecting the best Gini Index after the last entry becomes the statistic point.) We shall now describe such a condition. We make the following assumptions. For any attribute list, in addition to the buckets it actually contains, there is a dummy bucket that contains a single entry, ϵ. We use $last(B)$ to denote the last entry in bucket B and B_i to denote the i-th bucket. The dummy bucket is the 0-th bucket. We assume ϵ is a special splitting point and $\text{Gini}_\epsilon = \infty$. We assume the existence of a secure storage area [3] that is large enough to store all the

[3]It is 'secure' in the sense that the data stored there is persistent with respect to the execution of any program.

statistics up to a single entry in each attribute list. (This means the statistics up to one entry may overwrite the statistics up to another entry if both entries belong to the same attribute list and compete for the storage area.) Also, we assume this storage area has sufficient capacity to store the value of the Gini Index at a single splitting point in each attribute list.

Lemma 1 Let L be an attribute list for attribute X and b be the number of buckets in L. If there exists a time sequence $t_0 \leq t_1 \leq \cdots \leq t_b$ such that for all i, $0 \leq i \leq b$, all the entries in bucket B_i are in memory at time t_i, then for all $e \in B_i$,

1. e is a statistic point at t_i;
2. if $i > 0$ then $last(B_{i-1})$ is an S-predecessor of e;
3. for all $c \in dom(c)$ and $x \in dom(X)$, $S_e(c)$ in case of X being continuous, and $S_e(x, c)$ in case of X being categorical, can be calculated while B_i is in memory.

Proof: We prove the claim by induction on i, $0 \leq i \leq b$.

Base: $i = 0$. We have $e = \epsilon$. By Definition 1, clause 1 is true. Clause 2 is trivially true. By Definitions 2 and 3, $S_\epsilon(c) = 0$ or $S_\epsilon(x, c) = 0$, depending on whether X is a continuous or a categorical attribute. Surely both of them can be obtained when B_0 is in memory.

Inductive step: $1 \leq i \leq b$. By induction hypothesis $last(B_{i-1})$ is a statistic point. Since all entries in B_i are in memory at time t_i and $last(B_{i-1})$ immediately precedes B_i, by Definition 1, any $e \in B_i$ is a statistic point and $last(B_{i-1})$ is its S-predecessor. Thus, clauses 1 and 2 are true. To prove clause 3, we first assume X is a continuous attribute. By induction hypothesis, $S_{last(B_{i-1})}(c)$ can be calculated while B_{i-1} is in memory. By the assumption about storage space in memory, it can be stored in memory. By Definition 2,

$S_e(c)$ can be calculated while B_i is in the memory. In case X is a categorical attribute, the argument is similar. ∎

The following theorem describes our main results.

Theorem 3 Let L be an attribute list for attribute X which meets the conditions given in Lemma 1. Then the Gini Index for the best splitting point in L can be obtained while B_b is in memory.

Proof: We first assume X is a continuous attribute. We prove by induction on i that the Gini Index for the best splitting point in the prefix preceding (inclusive) $last(B_i)$ can be obtained while B_i is in memory.

Base: $i = 0$. Note $last(B_0) = \epsilon$. By assumption, $Gini_\epsilon = \infty$ [4]. This is the only Gini Index and hence the best in the case. It can be stored at t_0. (Indeed, it can be put in memory at any time.)

Inductive step: $i > 0$. Let y be the best splitting point in the prefix preceding $last(B_{i-1})$. By induction hypothesis, $Gini_y$ can be obtained while B_{i-1} is in memory. By assumption about the secure storage capacity, $Gini_y$ can be stored in the memory until B_i is brought into the memory. Let $x \in dom(X)$ such that there exists $e \in B_i$ with $e.X = x$. We still use e to denote the last entry with $e.X = x$ in B_i. By Lemma 1, the statistics up to e (i.e., $S_e(c)$ for all $c \in dom(C)$) can be obtained while B_i is in memory. By Theorem 1 and Equation 3.1, $Gini_x$ can be calculated while B_i is in the memory. Thus among $Gini_y$ and $Gini_x$ for all such x we can select the smallest one when B_i is in the memory. This completes the induction. By substituting b for i in the claim we prove the theorem for continuous attribute X.

Now we assume X is a categorical attribute. Let e be the last entry in L. Thus $e = last(B_b)$. By Lemma 1, for all $c \in dom(C)$ and $x \in dom(X)$, $S_e(x, c)$ can be calculated while B_b is in memory. By Theorem 2 and Equation 3.2,

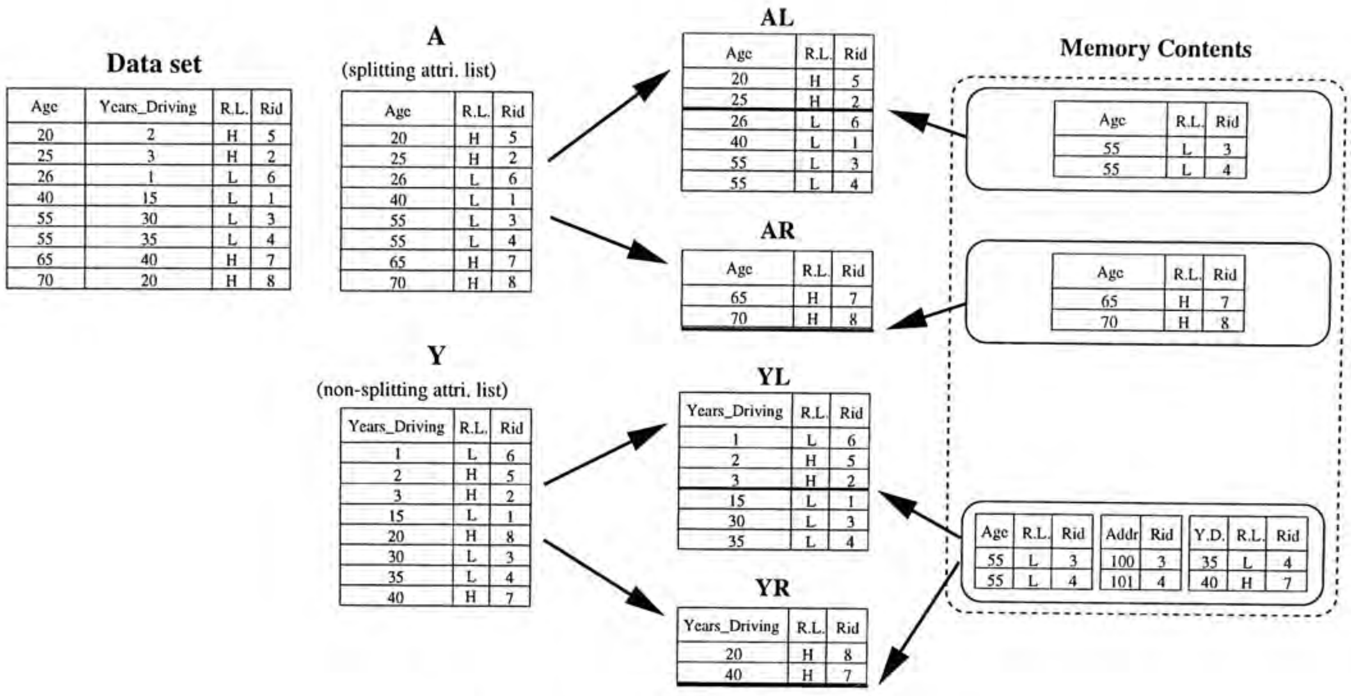
[4] In the implementation this is represented by the largest number that can be stored.

$Gini_z$ can be calculated for any $z \subseteq X$ while B_b is in memory. Thus, we can select the smallest one among them. ∎

To see how Theorem 3 is applied, we first consider the root. Continuous attribute lists are constructed at the root by sorting algorithms. We observe that almost all the popular sorting algorithms (such as selection sort, bubble sort, merge sort, quick sort, etc.) can ensure that any preceding element is inserted before any other element that follows it. As a result, we can ensure that for any attribute list, entries in preceding buckets are in memory before those in succeeding buckets. Thus the conditions in Theorem 3 are true. For a categorical attribute list, those conditions are guaranteed automatically: since the attribute list is not sorted, we are allowed to view the buckets created earlier as preceding buckets.

Now consider an arbitrary internal node that is not the root. Let it be N_L, its sibling be N_R, and its parent be N. Any attribute list, generated at N_L (N_R), is a result of splitting some attribute lists at N. If it is generated by splitting a splitting attribute list, say L^s, at N, then the conditions in Theorem 3 are clearly satisfied, since the entries are generated in the order they are placed in L^s.

Now suppose it is generated by splitting a non-splitting attribute list, say L^n. Note that we split L^n after L^s. Suppose L^s has been split into L^s_L and L^s_R, which are at N_L and N_R, respectively. As described in Section 3.1.1, we split L^n by bringing the buckets of L^s_L into memory. For each bucket that has been brought in we generate a hash table, and then we bring in the buckets of L^n one by one in a top-down fashion, in order to determine whether their entries belong to L^n_L or L^n_R (by probing the hash table). When the last bucket of L^s_L is brought into the memory, for each bucket of L^n that is brought in, all its entries will have their destinations determined. If we view all the entries in a bucket of L^n that belong to L^n_L (L^n_R) as a bucket of L^n_L (L^n_R), then for L^n_L (L^n_R) the order its buckets are generated in the memory are compatible with the



Data set

Age	Years_Driving	R.L.	Rid
20	2	H	5
25	3	H	2
26	1	L	6
40	15	L	1
55	30	L	3
55	35	L	4
65	40	H	7
70	20	H	8

A (splitting attri. list)

Age	R.L.	Rid
20	H	5
25	H	2
26	L	6
40	L	1
55	L	3
55	L	4
65	H	7
70	H	8

Y (non-splitting attri. list)

Years_Driving	R.L.	Rid
1	L	6
2	H	5
3	H	2
15	L	1
20	H	8
30	L	3
35	L	4
40	H	7

AL

Age	R.L.	Rid
20	H	5
25	H	2
26	L	6
40	L	1
55	L	3
55	L	4

AR

Age	R.L.	Rid
65	H	7
70	H	8

YL

Years_Driving	R.L.	Rid
1	L	6
2	H	5
3	H	2
15	L	1
30	L	3
35	L	4

YR

Years_Driving	R.L.	Rid
20	H	8
40	H	7

Memory Contents

Age	R.L.	Rid
55	L	3
55	L	4

Age	R.L.	Rid
65	H	7
70	H	8

Age	R.L.	Rid
55	L	3
55	L	4

Addr	Rid
100	3
101	4

Y.D.	R.L.	Rid
35	L	4
40	H	7

Figure 3.4: Pre-evaluation.

order they are stored. This means the conditions in Theorem 3 are satisfied.

Thus for both the root and internal nodes, we can apply Theorem 3. That is, after the last bucket of entries is created for each attribute list, the best splitting point for that list is determined while that bucket is still in memory. We then select the best of them as the final splitting point and the corresponding attribute list as the splitting attribute list.

To summarize, with the kind of pre-sorting mentioned above, the SPRINT framework can be improved in such a way that once the attribute lists are generated at a node we can determine for further splitting the splitting attribute list and the final splitting point without incurring any extra I/O. We call this scheme of evaluating splitting points **pre-evaluation**. (The prefixes 'post' and 'pre' are referring the time we write the attribute lists to the disk.)

Figure 3.4 shows an example of pre-evaluation. In this example, assume we have only two predictor attributes: *Age* and *Years_Driving*. Assume that at a certain node of the decision tree, *Age* and *Years_Driving* are, respectively, the splitting and non-splitting attributes, and A and Y are the corresponding attribute lists. Thus, at that node A is split into AL and AR first. Then, Y is split into YL and YR. The column under 'memory contents' shows the three occasions where some buckets have been brought into memory in the process of splitting A and Y. The inner box on the top row in that column indicates the last bucket of AL having been brought into the memory. At this time the best splitting point for AL is determined. Likewise the box on the second row determines the best splitting point for AR. The three inner boxes on the bottom row indicate that, from left to right, the last bucket of AL [5], the hash table and the last bucket of Y are in memory. Note that the last bucket of Y being in memory implies that the last bucket of YL, which is the entry with Rid of 4, and the last bucket of YR, which is the entry with Rid of 7, are in memory. Under such scenario, splitting Y into YL and YR is completed and at the same time the best splitting point for YL and that for YR are determined.

3.2 Schemes to Construct Decision Trees

In this section, we describe several schemes to construct decision trees (only for the growing phase) based on pre-sorting. All the schemes use pre-evaluations.

3.2.1 One-to-many Hashing

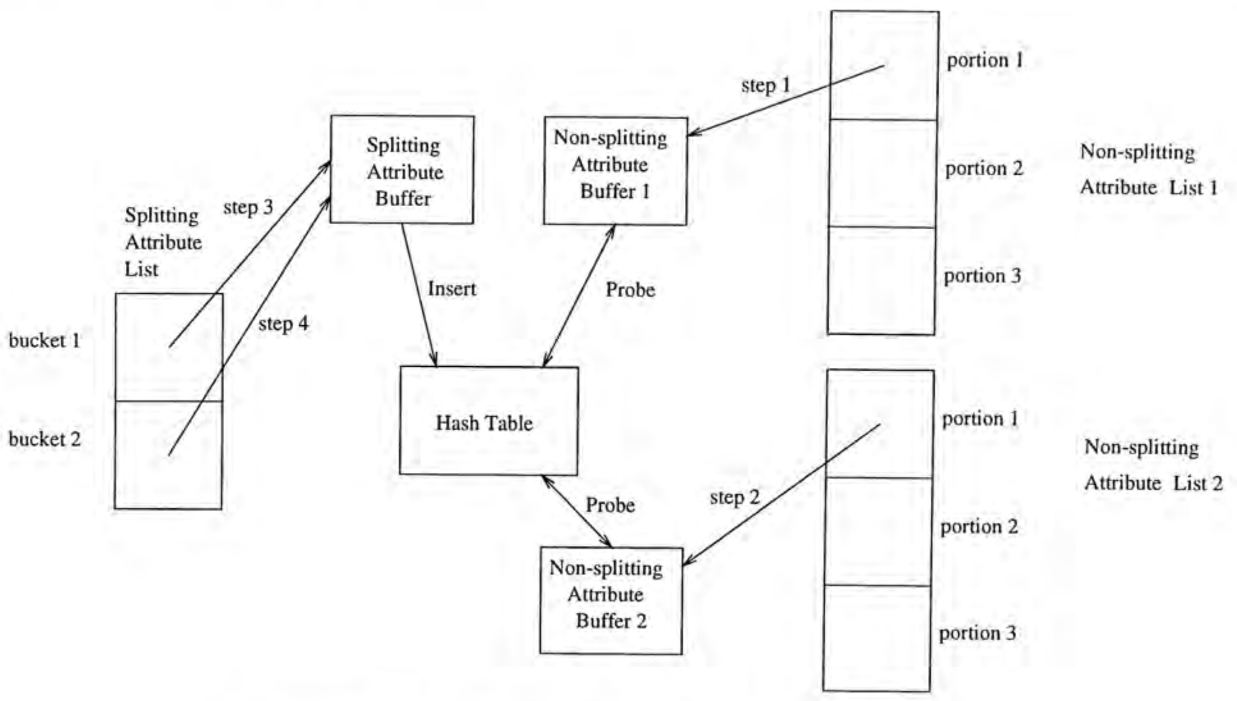
This is simply the SPRINT approach except that the pre-evaluation of splitting points is used. As described before, in this scheme for each bucket of records in the splitting attribute list, we create a hash table and then read every bucket from each non-splitting attribute list to probe the hash table. We use the phrase **one-to-many** because the buckets of the splitting attribute are loaded into the memory only *once* while the buckets of non-splitting attributes are loaded *many* times (see Figure 3.3).

[5] For performance reason we should use AR which is shorter than AL. We use AL here for better illustration.

3.2.2 Many-to-one and Horizontal Hashing

A merit of the one-to-many hashing is that the hash table is created only once in the memory. However, for each bucket from the splitting attribute list, each bucket of a non-splitting attribute list can possibly be brought into the memory multiple times. Each time it is brought into the memory, it must be subsequently written back to the corresponding disk file. To reduce the I/O cost, we can use an alternative to do the hashing, which we call **many-to-one hashing**. Like one-to-many hashing, we divide each attribute list into buckets. However we fetch the buckets from non-splitting attribute lists first. For each bucket fetched from a non-splitting attribute list, we bring in all the buckets from the splitting attribute list and create hash tables one by one. These hash tables are probed by the bucket of the non-splitting attribute list. When all the records in that bucket are resolved, we write them to the corresponding files, and then repeat this process for the next bucket from the non-splitting attribute list. Thus each bucket from a non-splitting attribute list will be read and written only once, while each bucket from the splitting attribute list can possibly be fetched multiple times, but without involving write operations. Our performance analysis in later sections will show that it indeed causes less I/O overhead compared to one-to-many hashing. We call this scheme many-to-one because the splitting attribute list has to be brought into the main memory *many* times while the non-splitting attribute lists are brought in only *one* time.

A question here is how to construct a bucket from the non-splitting attribute lists. One way of doing this is to let each bucket contain the records entirely from one list. We call the resulting scheme the **many-to-one simple hashing scheme**. The other is to divide the bucket into k slots, where k is the number of the non-splitting attribute lists, and let each slot contain the records from one list. We call this second method **horizontal hashing**.



step 1
portion 1
Splitting Attribute Buffer
Non-splitting Attribute Buffer 1
portion 2
Non-splitting Attribute List 1
Splitting Attribute List
step 3
Insert
Probe
portion 3
step 4
bucket 1
bucket 2
Hash Table
portion 1
Non-splitting Attribute List 2
Probe
step 2
portion 2
Non-splitting Attribute Buffer 2
portion 3

Figure 3.5: Horizontal Many-to-One Hashing.

A diagrammatical example of horizontal hashing is shown in Figure 3.5. In this example, Buffer 1 and Buffer 2 together make up the bucket for the non-splitting attributes. In Appendix A, we give some probabilistic comparison of the performance of the two methods.

3.2.3 A Scheme using Paired Attribute Lists

For this scheme, the overall framework is the same as that of the previous schemes. It differs from the previous schemes in the structure of the attribute lists and the way they are split. At each internal node N we maintain a set of **paired attribute lists**. Let X and Y be two attributes. The paired attribute list for **X paired with Y**, denoted as $\langle X, Y, C, Rid \rangle$, is a list of tuples of the values for these four attributes. Furthermore, if X is a continuous attribute the tuples are listed in the ascending order of the values of X. We say that X is the **host** and Y is the **guest** in the list.

The idea behind the attribute pairing is in the following. Suppose there exists another attribute list $\langle Y, Z, C, Rid \rangle$ at the same node. Then the values of

Y in $\langle X, Y, C, Rid\rangle$ can reference those of Y in $\langle Y, Z, R, Rid\rangle$. If $\langle Y, Z, R, Rid\rangle$ is the splitting attribute list at the left child, this reference can facilitate the process to distribute the entries in $\langle X, Y, C, Rid\rangle$, as described below. To make it possible for any attribute to reference any other attribute at a node, we maintain a cycle of m paired attribute lists: $\langle X_1, X_2, C, Rid\rangle$, $\cdots$, $\langle X_{m-1},, X_m, C, Rid\rangle$, $\langle X_m, X_1, C, Rid\rangle$.

The attribute lists at the root are created as a result of preprocessing. Again, pre-evaluation of splitting points can be made for its children. Now, consider how to construct the attribute lists at the two children N_L and N_R of P. Suppose the attribute lists for P have been created, and attribute X_1 has been chosen as the splitting attribute and x as the splitting point of X_1. There is no difficulty to split attribute list $\langle X_1, X_2, C, Rid\rangle$ for node N_L and N_R. Let the attribute list at N_L be $\langle X_1, X_2, C, Rid\rangle^L$ as a result of that splitting. To determine the entries that must go to node N_L (and respectively N_R) for the non-splitting attribute lists at P, we start from $\langle X_m, X_1, C, Rid\rangle$.

- First, assume X_1 is a continuous attribute. Since x is the splitting value for X_1 at node P, any entry in $\langle X_m, X_1, C, Rid\rangle$ with a value for X_1 being less than or equal to x goes to N_L and otherwise goes to N_R. Thus, we scan the entries in $\langle X_m, X_1, C, Rid\rangle$ top down. For each entry scanned, we can determine where it goes by examining its value for X_1.

- Second, assume X_1 is a categorical attribute. We also scan the entries in list $\langle X_m, X_1, C, Rids\rangle$. For each entry scanned, we check if its X_1 value is in set x. If it is the case, the entry goes to N_L, otherwise it goes to N_R.

We have split $\langle X_m, X_1, C, Rid\rangle$. Let $\langle X_m, X_1, C, Rid\rangle^L$ be the attribute list at node N_L as a result of that splitting. Note that this list is still sorted on the value for X_m in case it is a continuous attribute. Now consider how to split attribute list $\langle X_{m-1}, X_m, C, Rid\rangle$. Assume X_m is a continuous attribute. Let

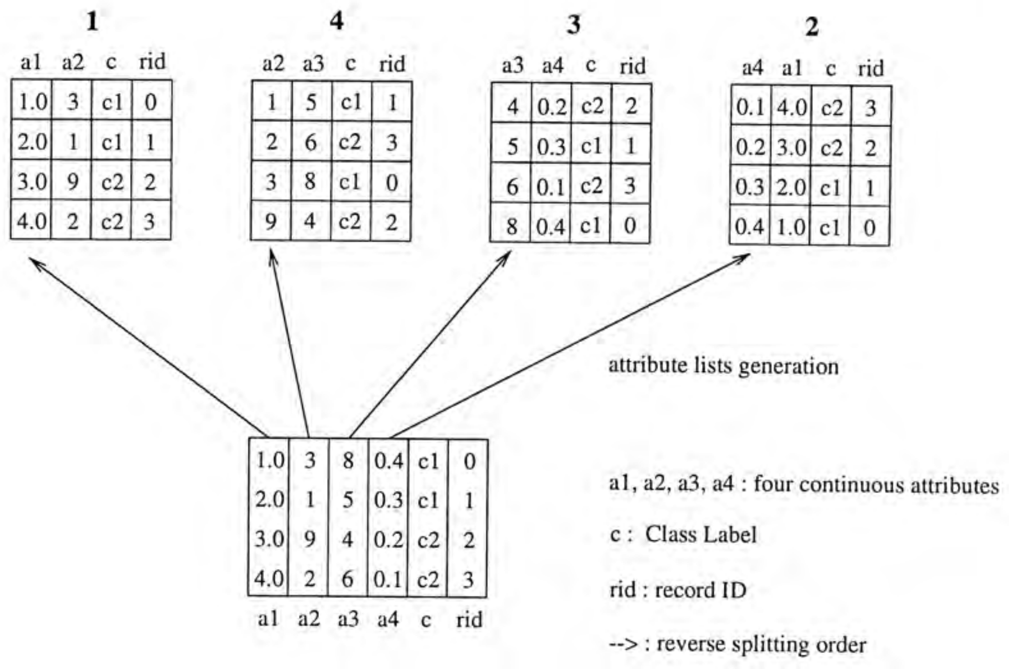


Figure 3.6: Splitting attribute by attribute pairing.

x'_m be the largest value for X_m in list $\langle X_m, X_1, C, Rid\rangle^L$. We scan the entries in list $\langle X_{m-1}, X_m, C, Rid\rangle$ top down. If an entry contains a value for X_m larger than x'_m, it will surely go to N_R, otherwise we use the hashing method similar to the previous schemes to determine the destination of the entry.

If X_m is a categorical attribute, we scan $\langle X_{m-1}, X_m, C, Rid\rangle$. For each entry scanned, we use hashing to determine which child it must go to. Once the list $\langle X_{m-1}, X_m, C, Rid\rangle$ splits, we then split $\langle X_{m-2}, X_{m-1}, C, Rid\rangle$, then $\langle X_{m-3}, X_{m-2}, C, Rids\rangle$, etc, by using the same procedure. Eventually, the list $\langle X_2, X_3, C, Rid\rangle$ will be split. We observe that these lists are split following the reverse order of their subscripts. We therefore refer to this order *reverse splitting order*. Figure 3.6 gives an example of this scheme. (The boldface numbers indicate the order the attribute lists are split, assuming the top left most list is the splitting attribute list.)

3.2.4 A Scheme using Database Replication

We notice that when we do the reverse splitting in the attribute pairing scheme, splitting the first non-splitting attribute list (i.e. the second in the reverse splitting order) is much easier than splitting other non-splitting attributes. This is because the value used for splitting that list is the splitting value of the splitting attribute list. Is it possible to use this splitting value for the splitting of all other non-splitting attribute lists? The answer is yes. We can make u copies of the entire dataset, where u is the number of continuous attributes. These copies are sorted based on the values of different continuous attributes. We use notation $\langle X_1, \cdots, X_i^*, \cdots \rangle$ to indicate that this dataset copy is sorted based on the value of X_i. (If there is no continuous attribute, a single copy must be maintained.)

At the root, all the copies are created in a preprocessing phase. As usual for any continuous attribute, the best splitting point can be determined when the copy is sorted. For categorical attributes we can choose any copy to do the evaluation when that copy is sorted in memory. Thus the splitting attribute and the splitting point are determined in a pre-evaluation of the splitting points. Now consider the children, N_L and N_R, of an internal node N, assuming the existence of u copies of the datasets, the splitting attribute X, and splitting point x at node N. Since every copy contains X, splitting is easy. We can simply compare the value for X in each entry with x. If X is continuous, a value smaller than or equal to x implies that the entry must go to N_L, otherwise to N_R. If X is categorical, a value belonging to x implies that the entry must go to N_L, otherwise it must go to N_R. Figure 3.7 describes this scheme. The asterisk indicates the attribute on which the database is sorted.

3.3 Performance Analysis

We have proposed the following new schemes:

(1) **Scheme 1**: one-to-many hashing,

(2) **Scheme 2**: many-to-one simple hashing,

(3) **Scheme 3**: many-to-one horizontal hashing,

List 3

a1	a2	a3*	a4	c	rid
3.0	9	4	0.2	c2	2
2.0	1	5	0.3	c1	1
4.0	2	6	0.1	c2	3
1.0	3	8	0.4	c1	0

List 4

a1	a2	a3	a4*	c	rid
4.0	2	6	0.1	c2	3
3.0	9	4	0.2	c2	2
2.0	1	5	0.3	c1	1
1.0	3	8	0.4	c1	0

List 1

a1*	a2	a3	a4	c	rid
1.0	3	8	0.4	c1	0
2.0	1	5	0.3	c1	1
3.0	9	4	0.2	c2	2
4.0	2	6	0.1	c2	3

List 2

a1	a2*	a3	a4	c	rid
2.0	1	5	0.3	c1	1
4.0	2	6	0.1	c2	3
1.0	3	8	0.4	c1	0
3.0	9	4	0.2	c2	2

Figure 3.7: Splitting attribute using replicated datasets.

(4) **Scheme 4**: paired attributes (one-to-many),

(5) **Scheme 5**: paired attributes (many-to-one),

(6) **Scheme 6**: dataset replication.

We also call the scheme of SPRINT **Scheme 0**. In this section, we compare the performance of these schemes. We first derive a general result for the amount of I/O operations involved in each scheme (in the worst case behavior). Then we apply these schemes to a specific database to gain some concrete idea of the performance of each of them.

We assume the available main memory space is fixed. All the data initially resides on disk, which are grouped into blocks. A block is the minimum unit to be transferred between the disk and the main memory. The original dataset is a table with $k + 1$ fields. Listed in Table 3.1 are the symbols for the parameters required by our derivation. Note that to simplify the derivation, we have assumed that all the fields (including the *rid* field) are of equal sizes.

Since all schemes need preprocessing, we will first concentrate on the number of disk blocks that must be fetched from or written to the disk when splitting takes place at any particular node for each scheme in the worst case.

Table 3.1: Notations for the parameters.

notations	unit	meaning
Z	bytes	available main memory space
f	bytes	field size of a record
b	bytes	block size
n	records	size of dataset at a node
k	attri	number of predictor attributes
u	attri	number of continuous attributes
B_s	bytes	bucket size for the splitting attribute
B_n	bytes	bucket size for non-splitting attributes

We call this number the **cost**. After that we will derive the cost for the preprocessing.

- First consider Scheme 0. Let N be an internal node of the decision tree.

  (1) The attribute list for the splitting attribute contains $\frac{3nf}{b}$ blocks. To split the attribute list, all its blocks are read and then written back. This amounts to $\frac{6nf}{b}$ block accesses.

  (2) After the splitting attribute list has been split into two lists, the smaller of the two lists, letting it be L_s, will be fetched into memory to form hash tables. Therefore the greatest possible size of this list is $\lfloor n/2 \rfloor$ entries, or bounded by $\lceil \frac{3nf}{2b} \rceil$ blocks. This list will form at most $\lceil \frac{3nf}{2B_s} \rceil$ buckets.[6]

  To split non-splitting attribute lists, for each bucket of L_s brought into the main memory, all the entries of non-splitting attribute lists must be fetched. The total size of them is $\frac{3knf}{b}$ blocks. Each of these blocks is read and subsequently written $\frac{3nf}{2B_s}$ times. Hence the total cost for splitting non-splitting attribute lists is $\frac{3nf}{2b} + \frac{3knf}{b}\frac{3nf}{B_s}$. Note that the hash table created for L_s and the non-splitting bucket B_n must fit into the main memory. If H is the hash table size, then $H + B_n \leq Z$. The number

[6] To simplify our discussion, we shall ignore the floors and ceilings in similar terms in the following analysis.

of entries in the hash table should be greater than that of L_s. However, each entry in the hash table has a small size, since it only needs to record the record id. In the implementation of the hash table, we can vary the utilization factor in the hash table, and typically $H = fB_s$, where f is a factor close to 1. We have

$$B_s + B_n \approx Z \tag{3.3}$$

(3) After both the splitting and non-splitting attributes have been split, they must be brought to memory again to evaluate the split points. This requires another $\frac{3knf}{b}$ block accesses.

Adding the above three values together, the total I/O cost for Scheme 0 at node N is

$$\frac{6nf}{b} + \frac{3nf}{2b} + \frac{k(3nf)^2}{bB_s} + \frac{3knf}{b} = (7.5 + 3k)\frac{nf}{b} + \frac{k(3nf)^2}{bB_s} \tag{3.4}$$

- For Scheme 1, we save on the third step because of the pre-evaluation. Hence, the total I/O cost is

$$\frac{6nf}{b} + \frac{3nf}{2b} + \frac{k(3nf)^2}{bB_s} = (7.5)\frac{nf}{b} + \frac{k(3nf)^2}{bB_s} \tag{3.5}$$

- For Scheme 2, we have many-to-one hashing. First, to split the splitting attribute list, we use $\frac{6nf}{b}$ block accesses. After the splitting attribute list has been split into two lists, let the the smaller of the two lists be L_s. Let the size of L_s be $n/2$ entries, or $\frac{3nf}{2b}$ blocks. The non-splitting attribute lists amounts to $\frac{3knf}{B_n}$ buckets. Then to split the non-splitting attribute lists, L_s is brought into the main memory $\frac{3knf}{B_n}$ times. Hence the cost involved in this step is $\frac{3knf}{B_n} + \frac{3nf}{2b}\frac{3knf}{B_n} = \frac{3knf}{B_n} + \frac{k(3nf)^2}{2bB_n}$. The total I/O cost is

$$\frac{6nf}{b} + \frac{3nf}{b} + \frac{k(3nf)^2}{2bB_n} = \left(6 + \frac{3kb}{B_n}\right)\frac{nf}{b} + \frac{k(3nf)^2}{2bB_n} \tag{3.6}$$

- The analysis for Scheme 3 is the same as that for Scheme 2.

- For Schemes 4 and 5, the cost to split each of the first two attribute lists in the reverse splitting order is $\frac{8nf}{b}$. which includes cost for both reading and writing. The derivation for the block accesses for each of the remaining $k-2$ attributes is similar to that for splitting a non-splitting attribute list in Scheme 1 and Scheme 2, except that now each record in the attribute list contains four fields, instead of three. Thus we omit the detail. The cost of Scheme 4 is:

$$\frac{8nf}{b} + \frac{8nf}{b} + \frac{4nf}{2b} + \frac{(k-2)(4nf)^2}{bB_s} = 18\frac{nf}{b} + \frac{(k-2)(4nf)^2}{bB_s} \tag{3.7}$$

  The cost of Scheme 5 is:

$$\frac{8nf}{b} + \frac{8nf}{b} + \frac{4knf}{b} + \frac{(k-2)(4nf)^2}{2bB_n} = 20\frac{nf}{b} + \frac{(k-2)(4nf)^2}{2bB_n} \tag{3.8}$$

- Lastly, we consider Scheme 6. Now an attribute list for an attribute (i.e, the entire dataset sorted on the value of that attribute) contains $k+2$ fields. This amounts to $(k+2)nf$ bytes, or $\frac{(k+2)nf}{b}$ blocks. Twice this value is the cost to split any replicated copy. Assuming there are u continuous attribute lists, the total cost for splitting all the attribute lists is

$$\frac{2u(k+2)nf}{b} \tag{3.9}$$

Equation 3.9 indicates that Scheme 6 would have better I/O performance for large dataset since it is linear in n, it also requires less CPU time since no hashing is necessary, but this scheme requires much more disk space to accommodate the replicated datasets.

From Equations 3.4, 3.5, and 3.7, we see that for the one-to-many schemes, it is better to use a larger B_s to minimize the I/O cost. From Equations 3.6

and 3.8, for the many-to-one schemes, it is better to use a larger B_n Since the buckets from the splitting and non-splitting attribute lists must share the limited memory space, a larger B_s (B_n) would lead to a smaller B_n (B_s). Thus if we set B_n too large, we will have a lot of overhead in bringing in small portions of records from disk and in creating a large number of small hashing tables. The CPU cost will go up. Similarly, if we set B_s too large, we will have a lot of overhead in bringing in small portions of records for the non-splitting attributes. Therefore, we expect that there is a value of B_s or B_n neither too large nor too small, where the overall performance is optimal.

The equations derived earlier also indicate that the many-to-one schemes have better I/O cost than one-to-many schemes if B_s and B_n are comparable. However, the many-to-one approach requires each bucket of the splitting attribute list to be brought into memory and the corresponding hash table to be created multiple times. This will incur more CPU costs. Therefore the choice would depend on the I/O performance and the CPU performance of the system.

The above derivation is only for the worst case behavior. For example, in Scheme 1, for each bucket of the split attribute list that has been brought into the memory, we read all buckets of the attribute list being split to probe the in-memory hash table entries. In reality, however, we may need to read only a fraction of the buckets of that list to do the probing, since the in-memory hash table is generated only by a portion (i.e., a bucket) of the split attribute list. The same thing can be said to the other hash table based schemes.

In light of this, Schemes 4 and 5 have a probabilistic advantage. In addition to hashing, attribute values are used to resolve (i.e., find the destination of) each record of the attribute list being split. With such a double resolution method, each record in a bucket of the attribute list being split may be resolved earlier, but never later than it would be in the single resolution scheme where only hashing were used. In other words, the fraction of the buckets being brought into the memory from the splitting attribute list in the double resolution scheme is likely to be smaller than it is in the single resolution scheme.

Now consider the preprocessing. For the first three schemes, each attribute list contains $\frac{3nf}{b}$ blocks. Thus it requires $\frac{3nf}{b} log \frac{3nf}{b}$ block accesses to be sorted. Thus the cost to sort all the attribute lists is $\frac{3knf}{b} log \frac{3nf}{b}$. Scheme 4 requires $\frac{4knf}{b} log \frac{4nf}{b}$ block accesses to sort all the attribute lists. Finally, Scheme 5 requires $\frac{(k+2)unf}{b} log \frac{(k+2)nf}{b}$ block accesses to sort all copies. These results show that, except for Scheme 5, the pre-sorting costs for all the schemes are dominated by the costs for splitting attribute lists.

3.4 Experimental Results

STATLOG [34] is a common benchmark for classification. However, since the largest dataset provided contains only 57,000 training records, we use the synthetic database proposed in [3] for all our experiments. There are ten classification functions proposed in [3] to produce datasets with varying complexities in distributions. In our experiments, we use two of these functions. Function 2 produces relatively small decision trees, while Function 6 results in very large trees. Both these functions divide the database into two classes: Group A and Group B. Figure 3.8 shows the predicates for Group A for each function. We use four attributes for Function 2 and six for Function 6, Table 3.2 shows the attribute descriptions.

3.4.1 Performance

We compare the total response time as well as the number of logical page access for all the schemes presented in this paper on the training sets of various sizes. Since other schemes either are shown to be less efficient than SPRINT or they make assumptions or restrictions different from ours, we choose not to include

Function 2 - Group A

$((age < 40) \wedge (50K \leq salary \leq 100K)) \vee ((40 \leq age < 60) \wedge (75K \leq salary \leq 125K)) \vee$
$((age \geq 60) \wedge (25K \leq salary \leq 75K))$

Function 6 - Group A

$((age < 40) \wedge (50K \leq (salary + commission) \leq 100K)) \vee ((40 \leq age < 60) \wedge$
$(75K \leq (salary + commission) \geq 125K)) \vee ((age \geq 60) \wedge (25K \leq (salary + commission) \leq 75K))$

Figure 3.8: Classification functions for synthetic data.

Table 3.2: Description of attributes for synthetic data.

Attribute	Description	Value
salary	salary	uniformly distributed from 20000 to 150000
commission	commission	salary $\geq$ 75000 $\Longrightarrow$ $commission = 0$ else uniformly distributed from 10000 to 75000
age	age	uniformly distributed from 20 to 80
hvalue	value of the house	uniformly distributed from $0.5 \times z \times 100000$ to $1.5 \times z \times 100000$ where $z \in \{0 \cdots 9\}$ depends on zip code of the place
hyears	years house owned	uniformly distributed from 1 to 30
loan	total loan amount	uniformly distributed from 0 to 500000

them into the comparison (see Section 3.1.1.)

Experiments are conducted under UNIX platform on a Sun Ultra 5 with 128MB main memory and 270MHz clock rate. We use a 9GB hard disk, for which the average I/O rate is 8000 blocks per second. We enforce any write or read operations bypassing the cache memory. All the schemes are implemented in C language. In our experiments, we force each page access to go to the hard disk and hence the logical page accesses correspond to physical page accesses. Table 3.3 shows various parameters setting for our experiments. We have employed both linear probing and coalesced chaining [7] for hash operations and find that the latter incurs less number of collision. As illustrated before, when we split a non-splitting attribute list, we bring into memory the buckets from the splitting attribute list to create hash tables. In the actual implementation,

[7]See Appendix B.

however, to make efficient use of memory space we do not need to put an entire bucket to the splitting attribute list in the memory. Instead, we use a working space the size of a single block and fetch into it the blocks from the splitting attribute list one after another. For each block fetched, we insert entries into the hash table. Thus, we need to put only a hash table and a bucket from non-splitting attribute list into the memory. As a result we have

$$H + B_n \approx Z \tag{3.10}$$

Although we have 128MB main memory, we restrict the usage of the main memory to be 5MB for the hash table and the buckets. The reason is that we want to simulate the case where the main memory is substantially smaller in size than the dataset. Under this circumstance the scalability for each scheme can be best observed.

3.4.2 Test 1 : Smaller Decision Tree

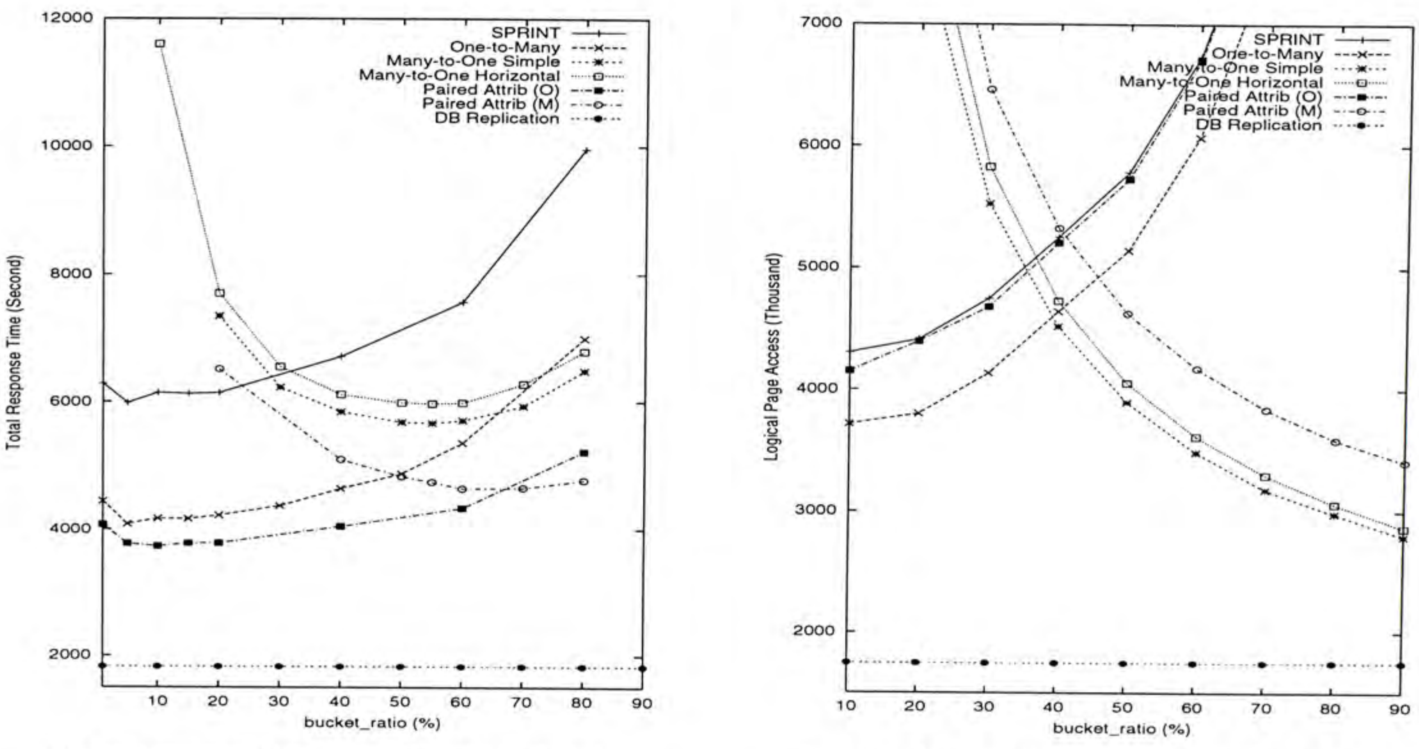
The first set of experiments is conducted with Function 2, which has 4 predictor attributes, and produces a very small decision tree. The range of total number of nodes is from 25 to 31 with different dataset size. We call this set of experiments **Test 1**. Our experiments confirm our expectations in the earlier analysis. Scheme 6 stands out as the best in response time and in page accesses for large datasets, since both I/O cost and CPU cost are low. It also has the linear scalability. However, the disk space requirement is much bigger than the other methods.

For the other schemes, given the fixed main memory allocation, we vary the bucket sizes, B_s and B_n. We discover that the overall response time follows a U-shaped curve with an optimal minimal point. This is shown in Figure 3.9 (a). If we measure only the page accesses (I/O time), the performance of the many-to-one schemes would improve with the value of B_n, while that of

Table 3.3: Parameters setting for the experiment.

Parameters	Descriptions
$page$	size of a page (4K bytes)
$data\_size$	training dataset size (number of records) = 1000K, 2000K, 3000K, ..., 10000K
$record\_size$	number of bytes for a record entry in an attribute list $= 3 \times 4$ for SPRINT, One-to-Many, Many-to-One Simple and Horizontal $= 4 \times 4$ for Paired Attrib (O), Paired Attrib (M) $= n \times 4$ for DB Replication where n is the total number of continuous attributes and field size = 4 bytes
$buffer\_size$	total main memory allocated = 5MB $\geq hashSize + otherBucketSize$
$bucket\_ratio$	ratio of $otherBucketSize$ to $hashSize$ = 0.01, 0.05, 0.1, 0.15, ..., 0.9
$hashSize$	memory size required for hash table (in bytes)
$hashRow$	number of entries in hash table $= \lfloor \frac{hashSize}{8} \rfloor$ for linear probing $= \lfloor \frac{hashSize}{12} \rfloor$ for coalesced chaining
$load\_factor, \alpha$	load factor (utilization) of hash table (see Appendix B) = 50%, 60%, 70% or 80%
$splitBucketSize$	main memory size used for the splitting attribute bucket (= B_s in Figure 3.3) $= load\_factor \times hashSize$ (bytes)
$splitBucketRow$	number of entries in splitting attribute bucket $= \lfloor \frac{splitBufferSize}{record\_size} \rfloor$
$otherBucketSize$	main memory size used for the non-splitting attribute bucket (= B_n in Figure 3.3) $= buffer\_size$ - $hashSize$ (bytes)
$otherBucketRow$	number of entries in non-splitting attribute bucket $= \lfloor \frac{otherBucketSize}{record\_size+1} \rfloor$

one-to-many schemes deteriorates. This is shown in Figure 3.9 (b).

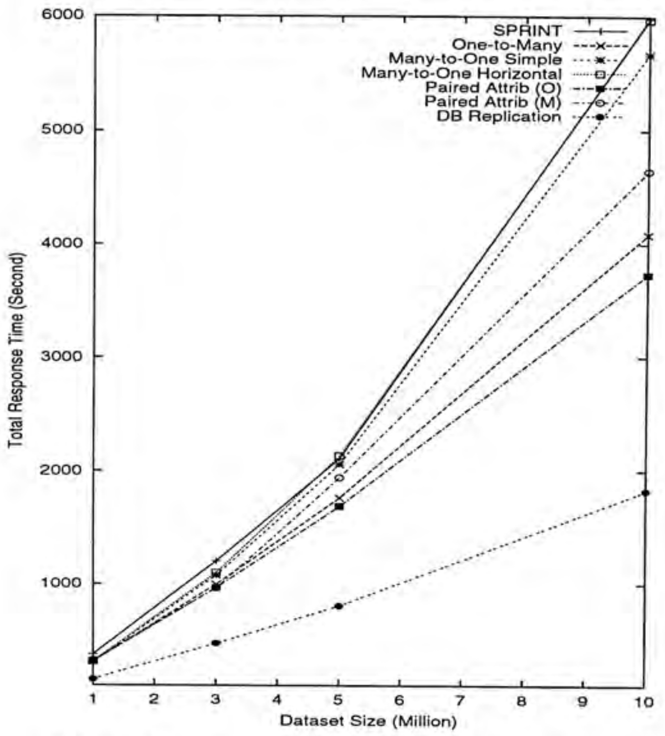


(a) Total response time by varying the bucket size

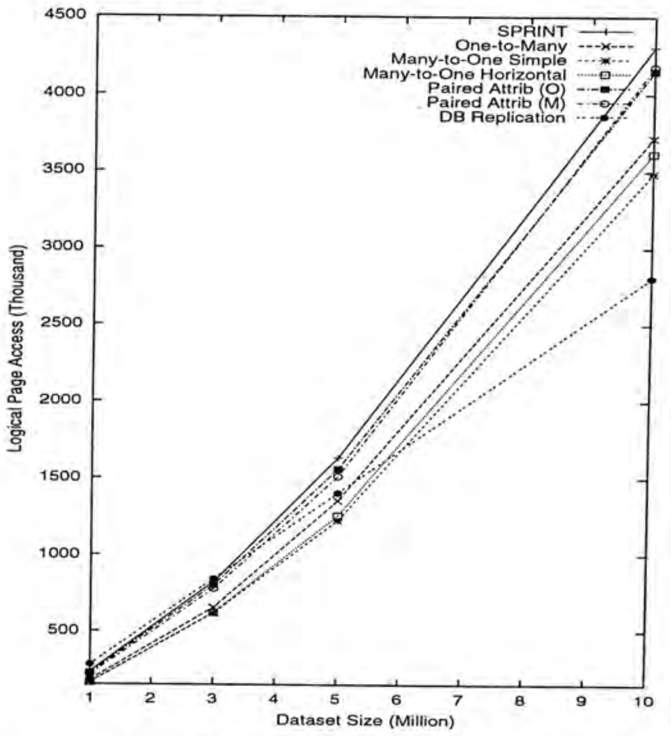
(b) Logical page access by varying the bucket size

Figure 3.9: Dataset size=10M, total buffer size=5MB, load factor=80%, using Function 2.

Next we choose the optimal bucket allocation for each scheme and perform measurement on the response time and the page accesses with varying database size. For page accesses, the many-to-one single and horizontal are the second best methods, which we can predict from Equation 3.6. The paired attribute schemes are not as good since the attribute lists are bigger in size and requires more I/O operations. The proposed methods are better than SPRINT as expected. The total response time include both the CPU time and the I/O time. Since the system we use has a very high I/O performance, the effect of the CPU time is quite dominant. The one-to-many schemes are better than many-to-one since much less hashing are needed. In particular, the one-to-many paired attribute scheme is the second best. This is because there is no hashing for the first two attribute lists at each node. On the other



(a) Total response time with optimal bucket size

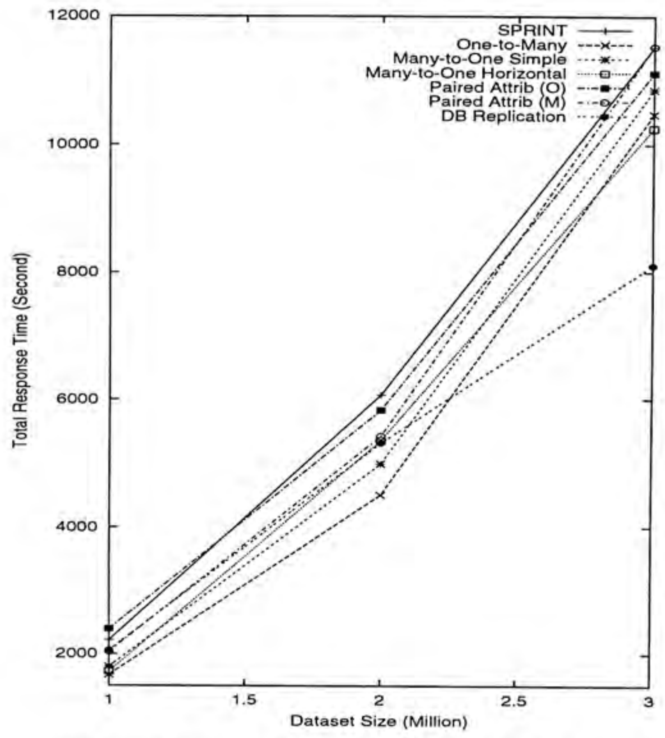


(b) Logical page access with optimal bucket size

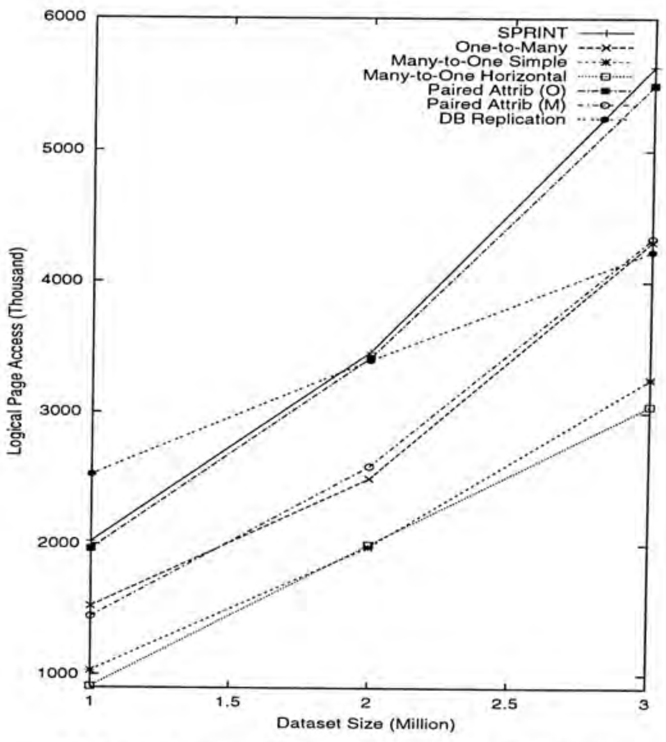
Figure 3.10: Total buffer size=5MB, load factor=80%, using Function 2.

hand, SPRINT and the horizontal scheme show very similar behavior, because in this case the SPRINT's high CPU performance is off-set by its poor I/O performance while the other way is true for horizontal scheme. We also observe that second to the database replication scheme, one-to-many scheme and one-to-many paired attribute scheme have excellent scalability in terms of the total response time. This is because the decision tree is relatively small and therefore the inferior I/O performance for one-to-many schemes is not significant. It is largely compensated by its high CPU performance.

We find that the disk usages are 1542MB and 623MB for the database replication scheme and SPRINT respectively for a 10M dataset size using Function 2. If the disk usage is feasible, the database replication scheme is superior to the other schemes.



(a) Total response time with optimal bucket size



(b) Logical page access with optimal bucket size

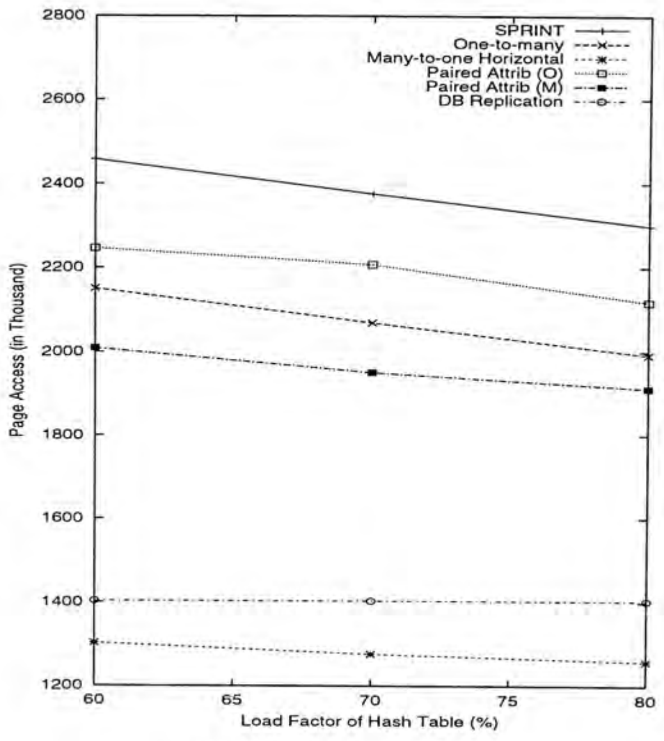
Figure 3.11: Total buffer size=5MB, load factor=80%, using Function 6.

3.4.3 Test 2: Bigger Decision Tree

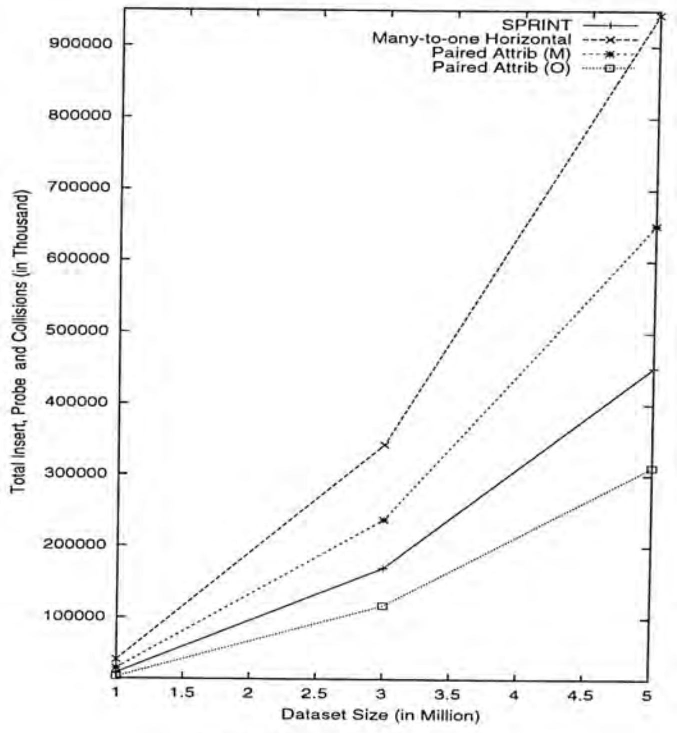
Next we experiment with Function 6, which has 6 predictor attributes, and which produces a much larger decision tree. The range of total number of nodes is from 2500 to 6000 with different dataset size. We call this **Test 2**. The results are similar to Test 1 when varying bucket sizes in that an optimal point can be found for the different schemes (except for the dataset replication scheme). Next we carry out experiments by varying the dataset size and choosing the optimal bucket sizes in each scheme. The result is shown in Figure 3.11.

From Figure 3.11, our proposed schemes again outperform SPRINT. The I/O cost of the database replication scheme has linear scalability in the dataset size. The many-to-one schemes also have good scalability. (The many-to-one simple and many-to-one horizontal are nearly linear.) For the total response time, the results are somewhat different from Test 1. In Test 1, the paired attribute schemes have pretty good performance. However, in Test 2, they are the worst among the proposed schemes. This is because the number of attributes have increased from 4 to 6. The paired attributes provides advantages for the first two attribute lists at each node, but demands slightly bigger attribute list sizes. In Test 1, we provide savings to 2 out of 4 attributes, while in Test 2, the ratio becomes 2 out of 6. The benefit becomes less significant while the overhead becomes more significant. The other difference is that the many-to-one schemes now have a better performance than they did in Test 1. We explain this as follows: since our decision trees are binary, when the decision tree becomes bigger, the number of levels increases and many nodes are found at the deeper levels of the tree. Such nodes typically correspond to smaller data sets. If data sets are large, the disk access are more contiguous, and the average disk access time per page is smaller. With smaller data sets, the disk access is fragmented, and the average disk access time per page increases. Therefore, the I/O time becomes more significant when the tree is bigger. Since the many-to-one schemes win over the one-to-many schemes in I/O, this results in better overall performance for the many-to-one schemes. Finally, we note that the horizontal scheme becomes better when the dataset size reduces to 3 millions. It can be explained by the probabilistic argument given in Appendix A. Since the data set sizes are small at many of the tree nodes, the probabilistic advantage becomes noticeable.

Next, we vary the load factor, α, of the hash table. Figure 3.12(a) shows the result. Increasing the load factor means increasing the utilization of the hash table. This allows more record entries of both splitting and non-splitting attribute lists to be loaded to the buffers in each cycle of hashing, thus decreases the total number of hash cycles needed during the distribution of record entries to the children of each node. As a result, the total number of I/O access for each node decreases. Note that, since Database Replication is independent of the load factor, the number of page access remains constant throughout the



(a) Different load factors



(b) Hashing performance

Figure 3.12: Total buffer size=5MB, dataset size=10M, using Function 2.

experiment. We also investigate the performance on hashing. Figure 3.12(b) shows the total number of insertion, probing and collisions for each of the schemes.

From the above analytical and experimental results, we conclude that the different proposed schemes are all better compared to SPRINT. The database replication scheme can be chosen when there is sufficient disk space. The other proposed schemes can be used in different scenarios. The many-to-one scheme has better I/O performance, while the one-to-many scheme has better CPU performance. The paired attribute scheme provides more savings if the number of attributes is small.

3.5 Conclusion

Pre-sorting attribute lists is an important technique for growing a decision tree. Compared with other techniques, it is less sensitive in performance to data distributions. Such a stability in performance has the advantage of, among other things, allowing us to analyze the algorithm by concentrating on its internal structures in order to improve its performance.

In this chapter, we present a family of schemes that grow decision trees based on pre-sorting. We start from the framework proposed in SPRINT. Then we show how the performance can be improved by using careful design and implementation of the procedures for splitting the attribute lists. We introduce techniques for the pre-evaluation of split points. We study several methods to split the dataset, including one-to-many and many-to-one hashing, horizontal hashing, attribute pairing and database replication, and derive results relating to their performance. We also report on experimental results, providing evidence to our expectations on the new schemes.

Part II

Mining Association Rules

Chapter 4

Background

4.1 Definition

Mining association rules is an important aspect in data mining [1, 7, 11, 16, 20, 23, 27, 29, 30, 38, 47, 49, 51]. This technique furnishes our understanding on large datasets by exploiting interesting regularities inside the data. Given a database of sales transactions, we find the relationships between different items in the database and represent such relationships in rule format, which helps us to trace the buying patterns in consumer behavior.

Here is an example of such a rule:

$$\forall x \in persons, buys(x, \text{``}bread\text{''}) \Rightarrow buys(x, \text{``}butter\text{''})$$

where x is a variable and *buy(x,y)* is a predicate that represents the fact the person x buys item y. This rule indicates that a high percentage of people who buy bread also buy butter.

A formal definition of association rules given in [7] is as follows. Let $I = \{i_1, i_2, \ldots, i_m\}$ be a set of items (**itemset**). Let D be a database of a set of transactions. Each transaction, T, consists of a set of items from I, i.e. $T \subseteq I$. Each transaction is identified by a unique identifier called TID. A transaction T is said to contain X if $X \subseteq T$, and $X \subseteq I$. An association rule of the form

$X \Longrightarrow Y$, where $X, Y \subset I$, and $X \cap Y = \emptyset$. s is the **support** of rule $X \Longrightarrow Y$ in D if $s\%$ of transactions in D contain $X \cup Y$. c is the **confidence** of rule $X \Longrightarrow Y$ if $c\%$ of transactions in D that contain X also contain Y.

Basically, the problem of mining association rules can be divided into two subproblems.

1. Find all set of items (itemsets) that have transaction supports above a predefined threshold called **minimum support**. These itemsets are called **large itemsets**.

2. Generate association rules from the large itemsets discovered in the previous step. These rules must have a confidence level above another predefined threshold called **minimum confidence**.

Recent work on mining association rules focus on the following aspects [1]:

1. Improving the computational efficiency and I/O costs in finding large itemset.

2. Introducing parallel algorithms for association rules generation.

3. Using sampling techniques for large datasets.

4. Supporting on-line generation of association rules.

5. Various extensions to the conventional problem, such as generalized associations, quantitative association rules and weighted association rules.

6. Constraints based association rules mining.

In this chapter, we give a brief survey on different kinds of association rules mining and introduce some of the state-of-the-art association rules mining algorithms.

4.2 Association Algorithms

4.2.1 Apriori-gen

Most large itemset computation algorithms are related to the Apriori algorithm [7]. It outperforms two of the earliest algorithms AIS [6] and SETM [24] by creating less candidate itemsets and avoiding the creation of candidates repeatedly for every transaction. The algorithm generates the candidate itemsets to be counted in a pass by using only the itemsets found large in the previous pass, without considering the transactions in the database. It is based on the observation that any subset of a large itemset must be large. Therefore, the candidate itemsets having k items can be generated by joining large itemsets having $k-1$ items, and deleting those that contain any subset that is not large. This is commonly known as *subset closure property.* This procedure generates a much smaller number of candidate itemsets. The number of database scan is $k_{max}+1$ where k_{max} is the maximal cardinality of a large itemset.

```
Algorithm : Apriori

$L_1$ = {large 1-itemsets};
Answer = $\emptyset$;
for ($k$=2; $L_{k-1} \neq \emptyset$; $k$++) do begin
    $C_k$ = apriori-gen($L_{k-1}$); // New candidates
    forall transaction $t \in \mathcal{D}$ do begin
        $C_t$ = subset($C_k, t$); // Candidate contained in $t$
        forall candidates $c \in C_t$ do
            $c$.count++;
    end
    $L_k$ = {$c \in C_k$ | $c$.count $\leq$ minsup }
end
Answer = $\bigcup_k L_k$;
```

Figure 4.1: The Apriori Algorithm.

Figure 4.1 outlines the Apriori algorithm. The first pass of the algorithm simply scans the database and counts the support of each item to determine the large 1-itemsets. A subsequent pass, say pass k, consists of two phases.

First, the large itemsets L_{k-1} found in the $(k-1)$-th pass are used to generate the candidate itemsets C_k. Next, the database is scanned and the support of candidates in C_k is counted using a hash tree. Itemsets in C_k of supports $\geq$ *thershold* form L_k. For the generation of candidate itemsets C_k, it consists of two steps: *join* and *prune*. First, in the *join* step, we join p and q, where (1) p, $q \in L_{k-1}$, (2) assume all items in an itemset is in lexicographic order, the first $k-2$ corresponding items in p and q are the same, and (3) the $(k-1)$th item in $p <$ that in q. We form a candidate k-itemset: $p.item_1, p.item_2, ..., p.item_{k-1}, q.item_{k-1}$. Next, in the *prune* step, we delete all itemsets $c \in C_k$ such that some $(k-1)$-subset of c is not in L_{k-1}.

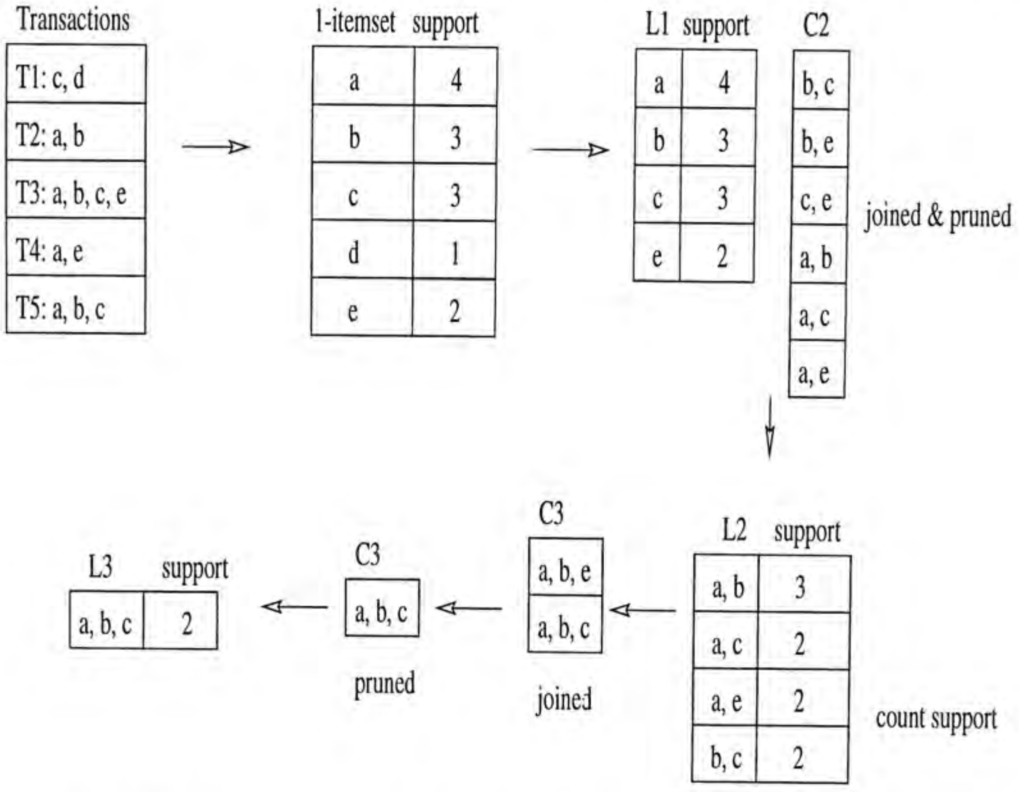


Figure 4.2: An example using Apriori-gen with threshold = 2.

Figure 4.2 shows an example of the Apriori-gen algorithm with a user specified threshold = 2. First, the database is scanned and the support of each 1-itemset is counted. Since the support of item d is 1, it is eliminated, and we get $L_1 = \{ a, b, c, e \}$. Then we form C_2 using the itemsets in L_1 by the join step. C_2 is unchanged after the prune step. Next, we scan the database and count the support for each 2-itemset in C_2. All candidate 2-itemsets of support $\geq$ threshold are considered large and put into L_2. We form C_3 using the large 2-itemsets. After the join step, we have $C_3 = \{ abe, abc \}$. However, since *be* is not in L_2, we prune *abe*. We scan the database again, and find the support of *abc* to be 2. Since L_3 consists of only one element, *abc*, we cannot form any candidate 4-itemset, and the mining process terminates.

4.2.2 Partition

Partition [47] uses a partitioning technique to reduce the I/O overhead by processing one database portion at a time in memory.

The algorithm divides the database into several non-overlapping partitions such that the size of each partition can be fit into the main memory. The mining process consists of two phases. In phase I, we consider one partition at a time. We define the local support, which is equal to the fraction of transactions containing that itemset in the partition, and generate all the large itemsets [1] for this partition. In the end of phase I, we merge the large itemsets to generate all potential large itemsets. In phase II, we count the actual supports of these itemsets so as to identify large itemsets. Since any potential large itemset must be large in at least one of the partitions, the potential set is a superset of all large itemsets which may contain false positives but no false negatives.

This method requires just two scans of the database to find the large itemsets, and is highly parallelizable. Other methods for parallelization, including the use of a shared hash-tree among multi-processors, can be found in [4, 36, 59, 60].

[1] Itemsets having local support not less than the predefined minimum support.

4.2.3 DIC

Instead of considering only k-itemsets in pass k, like that in the Apriori, Dynamic Itemset Counting (DIC) [11] reduces the number of passes made over the data by counting an itemset as soon as we suspect it may be necessary to count the itemset without waiting until the end of the previous pass, i.e. we can count the itemsets of different cardinality simultaneously.

A lattice structure is used to add and keep track of itemsets. The idea of the algorithm is that we partition the database into a number of blocks, initialize the lattice with all singleton sets, and scan the database block by block. For each block being scanned, the support of each itemset stored in the lattice is adjusted. After we process a block, we add an itemset to the lattice whenever all of its subsets are potentially large according to the support count of each of the subsets and the portion of transactions visited. When the last block has been scanned, we rewind to the first block and resume the algorithm until all the support count of each itemset in the lattice is determined. The resulting lattice is a superset of all the large itemsets.

4.2.4 FP-tree

An FP-tree (frequent pattern tree) is a variation of the *trie* data structure, which is a prefix-tree structure for storing crucial and compressed information about frequent patterns. The following is the definition of FP-tree stated in [23]: An FP-tree consists of a root labeled as "*NULL*", a set of item prefix subtrees as the children of the root, and a **frequent-item header table**. Each node in the item prefix subtree is composed of three fields: *item-name*, *count*, and *node-link*, where *item-name* indicates which item this node represents, *count* indicates the number of transactions containing items in the portion of the path reaching this node, and *node-link* points to the next node in the FP-tree carrying the same *item-name*, or null if there is none. There are two fields in each entry of the frequent-item header table: *item-name* and *head of node-link*. The latter points to the first node in the FP-tree carrying the *item-name*.

```
Algorithm : FP-tree construction
Input: A transaction database, $D$, and a user specified minimum support threshold, $\xi$.
Output: A frequent pattern tree, $T$.

build_FPtree($D$, $\xi$)
(1) Scan the transaction database, $D$. Find the support of each item.
(2) Sort the items by their supports in descending order.
      Choose all the items with support $\geq \xi$ to be the large 1-items.
(3) Build the FP-tree, $T$, by first creating its root with label being "NULL".
      For each transaction, $Trans$ in $D$:
            Select and sort the frequent items (large 1-items) in $Trans$ according to their supports.
            Let $[i|I]$ be the sorted frequent item list in $Trans$, where $i$ is the first element and
            $I$ is the remaining list. Invoke insert_FPtree($[i|I]$, $T$).

insert_FPtree($[i|I]$, $T$)
(1) If $T$ has a child $C$ such that $C$.item-name = $i$.item-name, then increment $C$'s count by 1;
      else create a new node $C$, and set its count to 1, link its parent to $T$,
            and let its node-link be linked to nodes with the same item-name via the node-link structure.
(2) If $I$ is non-empty, invoke insert_FPtree($I$, $C$).
```

Figure 4.3: FP-tree construction.

Table 4.1: A transaction database.

TID	Items	Sorted Frequent Items
001	*a,b,c,d*	*c,d,a,b*
002	*b,c,d,e*	*c,d,b,e*
003	*a,c,d*	*c,d,a*
004	*e, f*	*e*

Figure 4.3 shows the algorithm to build an FP-tree using a user specified threshold, ξ. Let us illustrate by an example the algorithm to build an FP-tree using ξ. Suppose we have a transaction database shown in Table 4.1 with $\xi = 2$. By scanning the database, we get the sorted (*item:support*) pairs, $\langle$(*c*:3), (*d*:3), (*a*:2), (*b*:2), (*e*:2), (*f*:1)$\rangle$. The frequent 1-itemsets are: *c*, *d*, *a*, *b*, *e*. We use the tree construction algorithm in [23] to build the corresponding FP-tree. We scan each transaction and insert the frequent items (according to the above

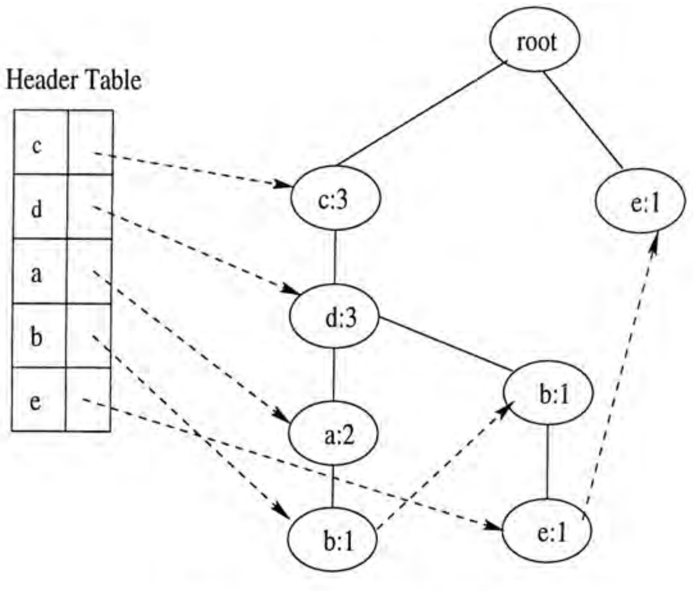


Figure 4.4: FP-tree.

sorted sequence) to the tree. First, we insert $\langle c, d, a, b \rangle$ to the empty tree. This results in a single path: root($NULL$) $\rightarrow (c : 1) \rightarrow (d : 1) \rightarrow (a : 1) \rightarrow (b : 1)$. Then, we insert $\langle c, d, b, e \rangle$. This leads to two paths with c and d being the common prefixes: root($NULL$) $\rightarrow (c : 2) \rightarrow (d : 2) \rightarrow (a : 1) \rightarrow (b : 1)$, root($NULL$) $\rightarrow (c : 2) \rightarrow (d : 2) \rightarrow (b : 1) \rightarrow (e : 1)$. Third, we insert $\langle c, d, a \rangle$. This time, no new node is created, but the counts in the first path is changed to: root($NULL$) $\rightarrow (c : 3) \rightarrow (d : 3) \rightarrow (a : 2) \rightarrow (b : 1)$. Finally, we insert $\langle e \rangle$ to the tree and we get the complete tree as shown in Figure 4.4. The header table shows the horizontal link for each frequent 1-itemset.

With the initial FP-tree, we can mine frequent itemsets of size k, where $k \geq 2$. An FP-growth algorithm [23] is used for the mining phase. We may start from the bottom of the header table and consider item e first. There are two paths: ⟨ c:3, d:3, b:1, e:1 ⟩, ⟨ e:1 ⟩. Since the second path contains only item e, we get only one prefix path for e: ⟨ c:1, d:1, b:1 ⟩, which is called e's **conditional pattern base** (the count for each item is one because the prefix path only appears once together with e). We also call e the **base item** of this conditional pattern base. Construction of an FP-tree on this conditional pattern base (**conditional FP-tree**), which acts as a transaction database

```
Algorithm : FP-growth
Input: FP-tree constructed based on the construction algorithm.
Output: The complete set of frequent itemsets.

FP-growth($Tree$, $\alpha$, $\xi$)
{
  If $Tree$ contains a single path $P$
  then for each combination, $\beta$, of the nodes in the path $P$ do
     generate pattern $\beta \cup \alpha$
        with $support$ = $minimum\ support$ of nodes in $\beta$;
  else for each $a_i$ in the header of $Tree$ do {
     generate pattern $\beta = a_i \cup \alpha$ with $support = a_i.support$;
     construct $\beta$'s conditional pattern base using $\xi$ and
        then $\beta$'s conditional FP-tree $Tree_\beta$;
     if $Tree_\beta \neq 0$
     then call FP-growth($Tree_\beta$, $\beta$, $\xi$) }
}
```

Figure 4.5: Algorithm for mining frequent patterns using FP-tree.

with respect to item e, results in an empty tree since the support for each conditional item $< \xi$ in the building phase. Next, we consider item b. We get the conditional pattern base: ⟨ c:1, d:1, a:1 ⟩, ⟨ c:1, d:1 ⟩. Construction of an FP-tree on this conditional pattern base results in an FP-tree with a single path: root($NULL$) $\rightarrow (c : 2) \rightarrow (d : 2)$. Mining this resulting FP-tree by forming all the possible combinations of items c and d with the appending of b, we get the frequent itemsets $(cb : 2)$, $(db : 2)$, and $(cdb : 2)$. Similarly, we consider items a, d and c to get the frequent itemsets. The resulting large itemsets after this mining phase are: $\langle c : 3, d : 3, a : 2, b : 2, e : 2, cd : 3, ac : 2, ad : 2, bc : 2, bd : 2, acd : 2, bcd : 2\rangle$.

The FP-tree algorithm is found to be much faster than the apriori based algorithm by experiments. It avoids the costly generation of candidate itemsets.

4.2.5 Vertical Data Mining

Many association rules algorithms use horizontal data layout in which the database can be viewed as a set of rows with each row representing a transaction of items purchased by a customer. Recently, vertical data representation has been considered. In such an alternative, we have a set of items, each associates with a column of values representing the transactions in which this item is present.

The advantages of using the vertical layout include the speed-up of support counting, dynamic reduction of database size during mining, and the support of compact storage of the database. However, most of the "vertical mining" algorithms suffer from some limitations such as special database size, special database schemas, and special characteristics of the database contents.

VIPER [49] uses a compressed bit-vector structure, called snakes, to represent itemsets. The algorithm requires multi-passes over the database, and works in a bottom up manner. It employs a DAG-based snake intersection scheme to efficiently count the supports of candidates of multiple levels in a single pass.

4.3 Taxonomies of Association Rules

4.3.1 Multi-level Association Rules

With the development of database query technique used in data warehousing, it is more meaningful and practical to arrange data at multiple levels of abstraction. The mining of multi-level association rules [21] provides a more concrete and specific information than the conventional single-level approach. However, using the Apriori like algorithm with a uniform support threshold is no longer suitable for the multi-level approach, it may generate a large amount of uninteresting itemsets at high levels of the concept taxonomies of the data.

Instead, we need to apply new technique such as using different minimum support thresholds for mining associations at different levels of abstraction. Extensions to this approach includes mining multiple-level correlations and multiple-level sequential patterns in large databases.

4.3.2 Multi-dimensional Association Rules

In classical association rule mining, each transaction in the database contains only the purchased items. Information about the items or transactions, such as purchase time and place, have been ignored. However, such information may be useful for mining interesting patterns. For example, in stock marketing, in addition to items, the time of the transactions is an important contextual information. In multi-dimensional association rule mining, we associate each record a set of attributes, known as **dimensional attributes**. These attributes form a multi-dimensional space, and a point in the space represents a transaction. Therefore, classical association rules can be viewed as a single dimensional space. Another enhancement is the **multi-dimensional inter-transaction** association rules which allows representing the associations of items among different transactions [28, 22]. In this way, we can mine more interesting patterns for a more complicated problem, such as *"If company X's stock goes down on day 1, Y's stock will go up on day 2, but go down on day 3."* [31].

4.3.3 Quantitative Association Rules

Mining quantitative association rules was proposed in [52] to handle the problem of mining association rules in large relational tables containing both quantitative and categorical attributes. This allows association rules to indicate how a given range of quantitative and categorical attributes may affect the values of other attributes in the database [1]. The problems we have to consider include the discretization of a quantitative attribute which may generate a large number of rules, etc.

4.3.4 Random Sampling

In large transaction database, it may be desirable to use sampling technique [54] to mine large itemsets in order to minimize the I/O costs. The weakness of random sampling is the presence of data skew which incurs rule inaccuracies.

4.3.5 Constraint-based Association Rules

From the view of human-interactive discovery of knowledge, conventional association rules mining, which acts as a black-box, suffers from the following serious shortcomings [37]: (i) Lack of user intervention during mining. (ii) Lack of concentration on particular types of query that the user may want. (iii) Limitation on the flexibility for users to choose the significance metrics instead of using support and confidence, and the criteria to be satisfied by the relationships to be mined.

Recent work has highlighted the significance of constraint-based mining technique [40]: users can specify their focus in mining, by means of a specific set of constraints that allow them to explore and control the interesting patterns. For example: In a supermarket, we may only want to know the relationships between items of particular types, such as soft drink and alcohol. We would like to focus on efficient techniques that allow the set of constraints to be pushed deep inside the mining process so as to efficiently prune the search space of patterns to those of interest to the users.

[22] divides constraints into five types as follows: **Knowledge type constraints** specify the type of knowledge to be mined, such as association, classification, prediction, clustering, concept description, or anomaly. **Data constraints** specify data set relevant to the mining task. **Dimension/level constraints** specify the dimension(s) or level(s) of data to be examined. **Rule**

constraints specify a set of constraints on the rules to be mined. **Interestingness constraints** measure the interestingness of discovered patterns using particular methods of measurement.

We focus on rule constraints for association rules mining. Here is the definition from [39]. A **constraint** C is a predicate on the powerset of a set of items I. A pattern or itemsets S *satisfies* a constraint C if and only if $C(S)$ is true. The complete set of patterns satisfying a constraint C, denoted as $SAT_C(I)$, is called **satisfying pattern set**. Given a transaction database, a support threshold and a set of constraints C, the problem of mining frequent patterns with constraints is to find the complete set of frequent patterns satisfying C.

Support Constraints Conventional association rules use only a single user-specified minimum support. In reality, the minimum support is not uniform.

[57] proposes *support constraints* as a way to specify general constraints on minimum support. It employs a support pushing technique that allows the highest possible minimum support to be pushed so as to tighten up the search space while preserving the essence of the Apriori.

Consider a set of items, I, partitioned into several bins $B_1, B_2, \ldots, B_m$ where each bin B_i contains a set of items in I. A **support constraint** of the form: $SC_i(B_{i_1}, \ldots, B_{i_s}) \geq \xi_i$, where $s \geq 0$ and B_{i_j} and B_{i_l} may be equal, specifies that any itemset containing at least one item from each B_{i_j} has the minimum support threshold ξ_i [57]. With such definition, we can mine different itemsets using different minimum support thresholds.

We will further investigate the constraint-based mining in Chapter 6.

Chapter 5

Mining Association Rules without Support Thresholds

In classical association rules mining, a minimum support threshold is assumed to be available for mining frequent itemsets. However, setting such a threshold is typically hard. If the threshold is set too high, nothing will be discovered; and if it is set too low, too many itemsets will be generated. In this chapter, we handle a more practical problem, roughly speaking, it is to mine the N k-itemsets with the highest support for k up to a certain k_{max} value. We call the results the N-most interesting itemsets. Generally, it is more straightforward for users to determine N and k_{max}. This approach also provides a solution for an open issue in the problem of subspace clustering. However, with the above problem definition without the support threshold, the subset closure property of the apriori-gen algorithm no longer holds. In this chapter, we propose three new algorithms, *LOOPBACK*, *BOLB*, and *BOMO*, for mining N-most interesting itemsets by variations of the FP-tree approach. A lower bound technique is introduced to determine a set of dynamic support thresholds. Experiments show that all our methods outperform the previously proposed Itemset-Loop algorithm, and the performance of BOMO can be an order of magnitude better than the original FP-tree algorithm even with the assumption of an optimally chosen support threshold.

5.1 Introduction

Classical association mining for association rules of the form $X \Longrightarrow Y$, where $X, Y \subset I$, and $X \cap Y = \emptyset$, is to first find large itemsets from transaction database D. That is, for all itemsets of cardinality ≥ 1, we would like to mine any itemset $X \cup Y$ such that its support, s, is greater than a certain value. Typically, this method requires a user specified minimum support threshold. However, without specific knowledge, users will have difficulties in setting this support threshold to obtain their required results. If the support threshold is set too large, there may be only a small number of results or even no result. In which case, the user may have to guess a smaller threshold and do the mining again, which may or may not give a better result. If the threshold is too small, there may be too many results for the users, too many results can imply an exceedingly long time in the computation. As an example of the difficulty in choosing a threshold, for the census data of United States 1990 available at the web site of IPUMS-98 [26], for two different sets of data, the thresholds for finding a reasonable number of itemsets are found to differ by an order of magnitude. See Tables 5.4 and 5.5 in Section 5.4 for an example where the reasonable support thresholds may vary from 0.3 to 21.42 for different data sets.

Another argument against the use of a uniform threshold for all itemsets is that the probability of occurrence of a larger size itemset is inherently much smaller than that of a smaller size itemset. Other objections to a uniform threshold are raised in [57], where it is believed that different items may have different characteristics that warrant different thresholds.

From our observations, it would be better for users to specify a threshold on the amount of results instead of a fixed threshold value for all itemsets of all sizes. For example, in multimedia data querying, we find that it is much more natural to allow users to specify the number of nearest neighbors rather than to specify a certain threshold on the "distance" from their query point. We have first introduced this problem definition for mining association rules in [16], in which we proposed two algorithms to mine the interesting itemsets with the constraint on the number of large itemsets instead of the minimum support threshold value. The resulting interesting itemsets are the *N-most interesting itemsets* of size k for each $1 \leq k \leq k_{max}$, given N and k_{max}.

Definition 4 A **k-itemset** is a set of items containing k items.

Definition 5 The **N-most interesting k-itemsets** : Let us sort the k-itemsets by descending support values, let S be the support of the N-th k-itemset in the sorted list. The N-most interesting k-itemsets are the set of k-itemsets having supports $\geq S$.

Definition 6 The **N-most interesting itemsets** is the union of the N-most interesting k-itemsets for each $1 \leq k \leq k_{max}$, where k_{max} is the upper bound of the itemset size we would like to find. We say that an itemset in the N-most interesting itemsets is **interesting**.

To simplify our discussion we adopt the following definition for the support of an itemset.

Definition 7 The **support** of an itemset I is the number of transactions that contain I.

Other than basket data, an important application for mining the N-most interesting itemsets is for the problem of *subspace clustering*, where we are interested to find clusters hidden in different subspaces of the given data space. With the CLIQUE [5] algorithm, each dimension of the data space is partitioned into equal number of intervals, and with this partitioning, the data space is partitioned into grid units. For any subspace of the full space, if the data density inside a grid unit is above a certain threshold, the grid unit can

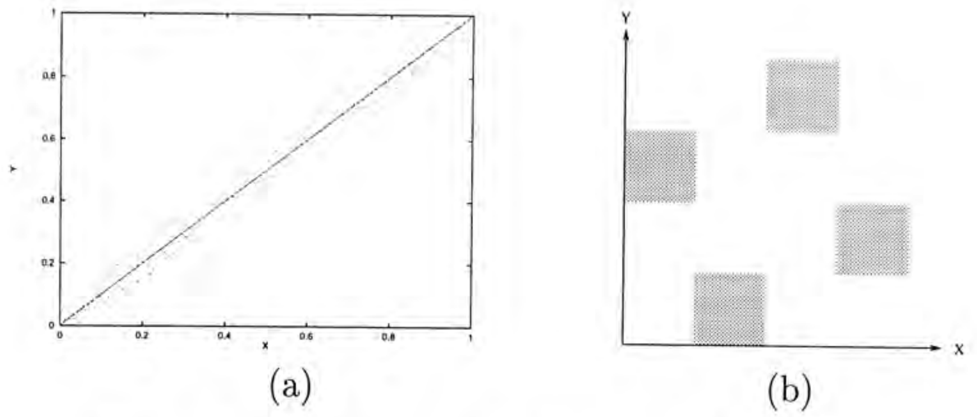


Figure 5.1: Two correlated variables X and Y.

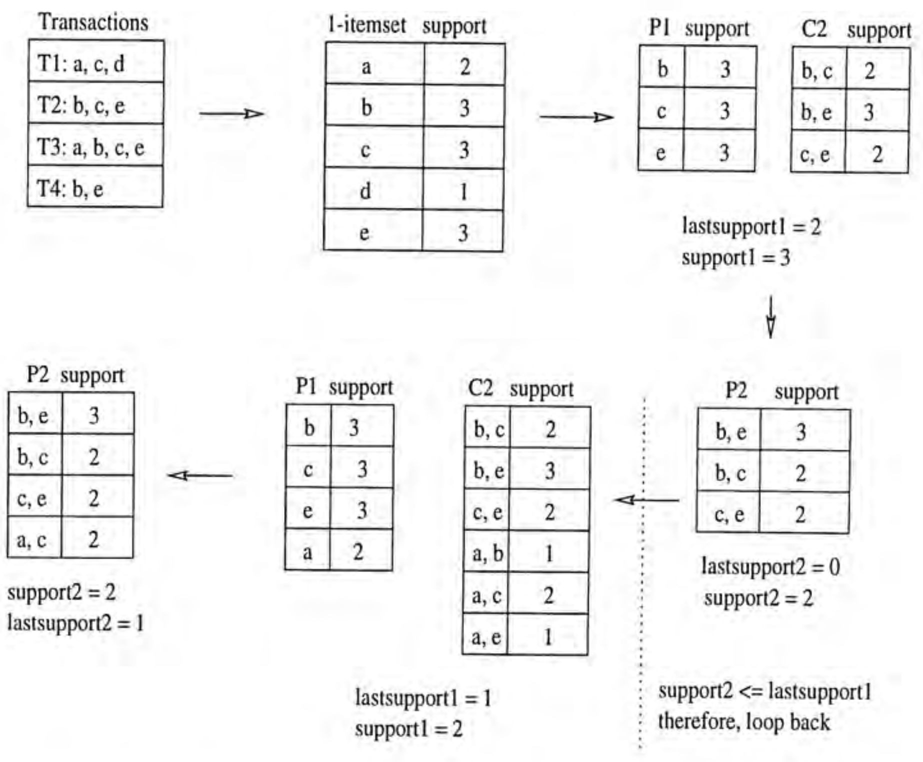
be considered part of a cluster. Hence the basic idea is very similar to finding large itemsets, and the algorithm of apriori-gen is used in CLIQUE.

However, we know that the total number of grid units in a subspace grows exponentially with the number of dimensions in the subspace, hence naturally higher dimensional grid units are much more sparsely populated compared with grid units in lower dimensional subspaces, so using a single threshold is problematic. Therefore algorithms such as CLIQUE [5] and the approach in [14] are not capable of dealing with some datasets in which some subspaces have good clustering but their projections on lower dimensional subspaces look uniform. Figure 5.1 shows two kinds of clustering which would be missed by CLIQUE unless it accepts uniform distribution as clustering. This problem can be resolved if we consider instead the N-most interesting dense units in subspaces with the same number of dimensions, which corresponds to the N-most interesting itemsets in the basket data scenario. Then in Figure 5.1, the two-dimensional subspace shown has a high chance to be considered interesting since we only compare it with other two-dimensional subspaces.

5.1.1 Itemset-Loop

To our knowledge, Itemset-Loop [16] is the first algorithm for the mining of N-most interesting itemsets. The idea is to apply a variation of the apriori-gen algorithm [7] repeatedly, using different support thresholds and a modified candidate generation mechanism. Based on the high probability that frequent itemsets of a larger size are formed by frequent itemsets of smaller size, we use a bottom up approach and form the candidate set of $(k+1)$-itemsets from the frequent k-itemsets. To prevent false dismissals, the algorithm checks if the N-th highest support of candidate $(k+1)$-itemsets (denoted as $support_{k+1}$) is greater than the $(N+1)$-th highest support of the frequent k-itemsets (denoted as $lastsupport_k$). If the condition holds, it is unnecessary for looping back. Otherwise, it means that we have not uncovered all k-itemsets of sufficient support that may generate a $(k+1)$-itemsets with supports $\geq support_{k+1}$. The system will loop back to find new potential $(k+1)$-itemsets, whose supports are not less than $support_{k+1}$. Another case when loop back is necessary is if the number of $(k+1)$-itemsets discovered is less than N. The loop back will then apply a smaller support threshold value. It has been shown that the proposed algorithm is efficient and is better than the result of the apriori-algorithm if the support threshold is set to be smaller than the optimal value which can uncover all the N-most interesting itemsets. The proposed method will be inherently much better when the guess of the support threshold is above the optimal, so that the conventional approach cannot uncover the proper results. Figure 5.2 shows an example of the loop back mechanism.

Although Itemset-Loop has good performance, it makes use of the apriori candidate generation mechanism which relies on the property of subset closure: if a k-itemset is large then all its subsets are also large. This property does not hold for mining N-most interesting itemsets. That is, if a k-itemset is among the N-most interesting k-itemsets, its subsets may not be among the



P_K : A k-itemset that can potentially form part of an interesting (k+1)-itemset.

C_K : A k-itemset that potentially has sufficient support to be interesting and is gernerataed by joining two potentially (k-1)-itemsets.

Figure 5.2: An example using Itemset-Loop.

N-most interesting itemsets. Therefore we examine other algorithms for the association rule mining problem and find that the FP-tree [23] approach does not rely on the candidate generation step. We therefore consider how to make use of the FP-tree for the N-most interesting itemsets mining.

5.2 New Approaches

In this section, we introduce three new algorithms for mining N-most interesting itemsets. We adopt ideas of the FP-tree structure [23] in our algorithms.

D	the given transaction database.
k_{max}	upper bound on the size of interesting itemsets to be found.
$result_k$	the current resulting set of N-most interesting k-itemsets.
ξ	current support threshold for all the itemsets
ξ_k	current support threshold for the k-itemsets

5.2.1 A Build-Once and Mine-Once Approach, BOMO

In the original FP-tree method [23], the FP-tree is built only with the items with sufficient support. However, in our problem setting, there is no support threshold given initially, so we cannot choose items with sufficient support. We therefore propose to build a complete FP-tree with all items in the database. Note that this is equivalent to setting the initial support threshold ξ to zero. The size of an FP-tree is bounded by the size of its corresponding database because each transaction will contribute at most one path to the FP-tree, with the length equal to the number of items in that transaction. Since there is often a lot of sharing of frequent items among transactions, the size of the tree is usually much smaller than its original database [23]. As a result, it does not require a large amount of memory to store the complete tree.

Adjusting Thresholds ξ_k and ξ

Although the initial value of ξ is zero, it will be dynamically increased as we progress with the mining step. Besides ξ, we use different thresholds for itemsets of different sizes. When nothing is known about the itemsets, we can set ξ_k to be zero for all k, $1 \leq k \leq k_{max}$. This means that we do not know any lower bound for the thresholds of interest. With $\xi_k = 0$, we would blindly include any k-itemsets in our current set of result, $result_k$. However, in the process of mining, we shall generate k-itemsets with their supports. Once we have encountered any N k-itemsets for some $k \leq k_{max}$, we know that we are interested in k-itemsets with supports at least as great as any of these N k-itemsets.

Lemma 2 Given any N k-itemsets for some $1 \leq k \leq k_{max}$, the N-most interesting k-itemsets have support $\geq$ any of these N itemsets.

Therefore during the mining phase, we adjust these ξ_k once we have found at least N itemsets of size k: we assign ξ_k to be the support of the N-th most frequent k-itemset discovered so far. In order to do this, we maintain a sorted list of the support values of the N most frequent k-itemsets discovered, for each $1 \leq k \leq k_{max}$. Figure 5.3, 5.4, and 5.5 show the main algorithm, the tree-building step and the mining step. The values of ξ_k are used to determine if an itemset should be included in the current $result_k$. Here is an example. Suppose $N = 5$ and $k_{max} = 3$, we set $\xi = 0$, $\xi_1 = \xi_2 = \xi_3 = \xi = 0$. Suppose we have found the following 1-itemsets: $\{ a : 8, b : 8, c : 6, d : 6, e : 4, f : 4 \}$. Then $\xi_1 = 4$. Considering large 2-itemsets, suppose $\langle cd : 5, ab : 8, ac : 6, bd : 4, bc : 6, ad : 6 \rangle$ have been found so far in the mining process. We can set ξ_2 to 5 since the support of the N-th most frequent 2-itemset so far, *cd*, is 5. Therefore, we need no longer consider 2-itemsets of support less than 5.

In the mining phase, we set the initial threshold value of ξ to be zero. During mining, we increase ξ by assigning to it the minimum value among the supports of the N-th most frequent k-itemset discovered so far for $1 \leq k \leq k_{max}$:

$$\xi = min(\xi_1, \xi_2, \ldots, \xi_{k_{max}}) \tag{5.1}$$

We shall see later that as the threshold becomes greater, the pruning power will also be greater.

Lemma 3 At the end of the first for loop of NFP-mine() (see Line (1) in Figure 5.5), if there are at least N different items in the set of k-itemsets discovered so far, then for $j < k$, $\xi_j \geq \xi_k$.

Proof: If $\xi_k = 0$ then the lemma is trivially true, since ξ_j is never set negative. If $\xi_k > 0$, it means that we have uncovered at least N k-itemsets and the smallest among their N highest support counts equals ξ_k. For each k-itemsets X, any subset of X has support at least that of ξ_k. If the N-most frequent k-itemsets contains N different items, let the itemsets be $I_1, I_2..., I_N$. From I_1 we can form k subsets of $(k-1)$-itemsets. In I_i, if it contains items not contained in $I_1, ..., I_{i-1}$, then if there is only one such item x, we can form $k-1$ subsets with $k-1$ elements, each subset formed by removing each of the items not equal to x. These subsets are different from all the $(k-1)$-itemsets generated from $I_1, ..., I_{i-1}$. If I_i contains more than one items which are not in $I_1, ..., I_{i-1}$, one can form k subsets with $k-1$ elements by removing any item from I_i. Therefore when we finish the subset formation from $I_1, \ldots, I_N$, we can get sufficient N $(k-1)$-itemsets, and their supports are $\geq \xi_k$. We can repeat the argument with smaller subsets until we come to the 1-itemsets. In the FP-tree mining process, the subsets are generated either earlier or at the same iteration as the generation of the k-itemset I. Therefore, for $j < k$, $\xi_j \geq \xi_k$. ∎

Corollary 1 At the end of the first for loop of NFP-mine(), if there are at least N different items in the set of k-itemsets discovered so far, then $\xi = \xi_{k_{max}}$

The above is true since $\xi = min(\xi_1, \xi_2, \ldots, \xi_{k_{max}})$ and Lemma 3 implies that for $j < k_{max}$, $\xi_j \geq \xi_{k_{max}}$. Therefore, if the condition in the Corollary is satisfied, then instead of updating ξ as $min(\xi_1, \xi_2, \ldots, \xi_{k_{max}})$, we can update ξ only when $\xi_{k_{max}}$ is updated. The pruning effect will be unchanged, while less work is spent on the update of ξ.

Pre-evaluation Step

To enhance the performance of BOMO, we use a pre-mining step to evaluate a better initial lower bound for ξ and ξ_k, $2 \leq k \leq k_{max}$. We assign an array, C_k, of size N for ξ_k, for $2 \leq k \leq k_{max}$. The FP-tree built from NFP-build()

is scanned from the root to the k-th element of each path. We store to C_k the N largest values among the counts stored in the nodes of the k-th element of all the paths. Then we assign ξ_k to the N-th largest count value stored in C_k. Using Equation (5.1) we can determine ξ. We use these initial lower bounds for our mining phase.

Pruning Conditional FP-trees

Based on the fact that an element of the header table cannot form a conditional pattern tree that consists of itemsets having supports greater than the support of this element, we have the following pruning step for BOMO: When forming the conditional FP-tree for an element, α, in the header table, we compare the support of α, which is equal to the total sum of the counts in the horizontal link of α, with ξ. If it is smaller than ξ, we can stop forming the conditional FP-trees for all the elements starting from α to the bottom element of the header table.

```
Algorithm : NFP-tree algorithm
Input: $D$
Input: $k_{max}$
Input: $N$
Output: $N$-most interesting $k$-itemsets for $1 \le k \le k_{max}$.

(1) Let $result_k$ be the resulting set of interesting k-itemsets. Set $result_k = \emptyset$.
(2) Scan the transaction database, $D$. Find the support of each item.
(3) Sort the items by their supports in descending order, denote it as sorted-list; determine $\xi_1$.
(4) $\xi_2 = \xi_3 = \cdots = \xi_{k_{max}} = \xi = 0$
(5) Create a FP-tree, $T$, with only a root node with label being "NULL".
(6) $N$FP-build($T$, $D$, sorted-list, $\xi$ ).
(7) $result_1, result_2, \ldots, result_{k_{max}} \leftarrow$ $N$FP-mine($T$, $\emptyset$, $\xi$, $\xi_1$, $\xi_2$, ..., $\xi_{k_{max}}$ )
```

Figure 5.3: NFP-tree algorithm for mining N-most interesting itemsets.

```
Algorithm : NFP-tree Building Phase

NFP-build(T, D, sorted-list, $\xi$)
(1) selected-list $\leftarrow$ sorted 1-itemsets list whose supports $\geq \xi$.
(2) Update the FP-tree as follows:
    For each transaction, $Trans$ in $D$
        If $Trans$ consists of some items which are in selected-list
            Select and sort the items, which are in selected-list, in $Trans$ according to their supports.
            Let $[i|I]$ be the sorted frequent item list in $Trans$, where $i$ is the first element and
            $I$ is the remaining list. Invoke $Insert\_NFPtree([i|I], T)$.

Insert_NFPtree([i|I], T)
(1) If $T$ has a child $C$ such that $C$.item-name = $i$.item-name,
    then increment $C$'s count by 1;
    else create a new node $C$, and let its count be 1, its parent link be linked to $T$,
        and its node-link be linked to nodes with the same item-name via the node-link structure.
(2) If $I$ is non-empty, invoke $Insert\_NFPtree(I, C)$.
```

Figure 5.4: NFP-tree construction.

```
Algorithm : NFP-tree Mining Phase
NFP-mine($Tree$, $\alpha$, $\xi$, $\xi_1$, $\xi_2$, ..., $\xi_{k_{max}}$)
{
  If $Tree$ contains a single path $P$
  (1) then for each combination, $\beta$, of the nodes in the path $P$ do
     (a) generate itemset $\beta \cup \alpha$ with $support$ = $minimum$ $support$ of nodes in $\beta$
     (b) if $\xi_{|\beta\cup\alpha|} \leq support$
            then insert $\beta \cup \alpha$ to $result_{|\beta\cup\alpha|}$; update $\xi_{|\beta\cup\alpha|}$; update $\xi$ if necessary
  (2) else for each $a_i$ in the header of $Tree$ do
       (c) generate itemset $\beta = a_i \cup \alpha$ with $support$ = $a_i.support$
       (d) construct $\beta$'s conditional pattern base using $\xi$ and
             then $\beta$'s conditional FP-tree $Tree_\beta$
       (e) if $Tree_\beta \neq \emptyset$ then NFP-mine($Tree_\beta$, $\beta$, $\xi$, $\xi_1$, $\xi_2$, ..., $\xi_{k_{max}}$ )
}
```

Figure 5.5: Algorithm for mining phase of N-most interesting itemsets.

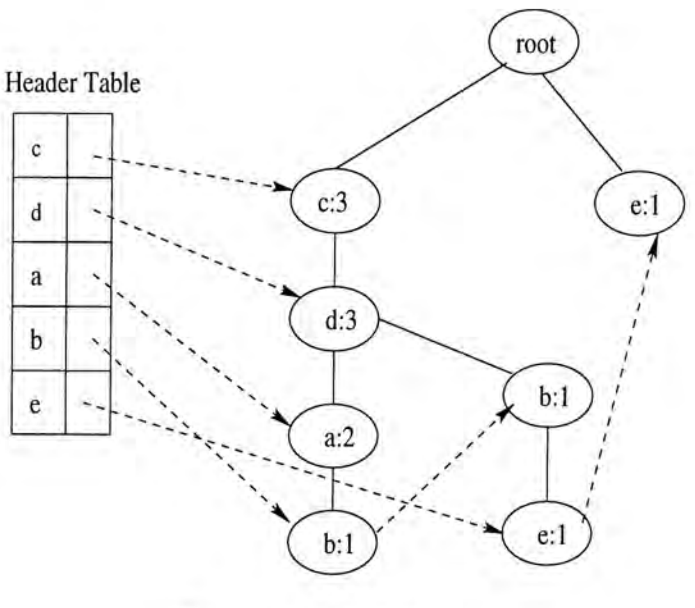


Figure 5.6: FP-tree.

Construction Order of Conditional FP-trees

We also study the starting position when processing the frequent-item header table (Line (2) of Figure 5.5). In the original FP-tree method [23] the ordering is not important since the same amount of work will be done independent of the ordering. However, in our settings, the ordering can have significant impact since the thresholds ξ, ξ_k, $1 \leq k \leq k_{max}$, are updated dynamically and if we encounter more suitable itemsets with high support counts earlier, we can set the thresholds better and increase the subsequent pruning power.

Top-down: One possible choice is to start from the top of the header table and go down the table to form conditional pattern base for each item in the table. For example, in Figure 5.6, the ordering will be {c, d, a, b, e}. This is based on the observation that frequent items are most likely located at the top levels of the FP-tree and hence we can prune the less frequent itemsets by finding the more frequent itemsets first. However, itemsets of larger sizes are usually distributed widely throughout the tree, and as a result, with this ordering, the increase rate of ξ_k for large k is slow while that for small k is fast. According to Equation (5.1), ξ is also increased slowly. This leads to a large number of large conditional FP-trees.

Bottom-up: The other extreme is to go from the bottom of the header table upwards to the top. For example, in Figure 5.6, the ordering will be {e, b, a, d, c}. However, the bottom items have the smallest supports and the corresponding itemsets discovered will also have small supports, and hence the pruning power is also small.

Starting from the middle: We have tried different starting positions and scanning orders of the header table in our experiments, and find that starting at the middle of the header table; then from the middle item going upwards to the top and then from the (middle+1) item down to the bottom of the table, is a good choice. For example, in Figure 5.6, the ordering will be {a, d, c, b, e}. The reason is that the increase rate of ξ is faster and less conditional FP-trees are formed. This ordering will be used in all of our experiments discussed in Section 5.4.

5.2.2 A Loop-back Approach, LOOPBACK

Table 5.1: A transaction database.

TID	Items	Sorted Frequent Items
001	a, b, c, d	c, d, a, b
002	b, c, d, e	c, d, b, e
003	a, c, d	c, d, a
004	e, f	e

Compared to the original FP-tree algorithm in [23], BOMO requires building an initial FP-tree with all items, while the original method may build an initial FP-tree for a subset of the items. Hence we may consider the possibility of building a smaller initial FP-tree. In order to do so, we should have an initial support threshold $\xi > 0$. Here we suggest such an approach, the initial value of ξ is determined by the smallest support of the N most frequent 1-itemsets. Let us consider the example in Table 5.1, with $N = 3$. Since the support of the third largest 1-itemset is 2, the threshold ξ is set to 2. We use the tree building algorithm to build the FP-tree. Therefore, we get c, d, a, b, and e as the large 1-itemsets. The constructed FP-tree is the same as that in Figure 5.6.

Using the previous example to illustrate the mining mechanism, we start from the top of the header table and invoke NFP-mine($Tree_{init}$, $NULL$, ξ, ξ_1, ξ_2, $\ldots$, $\xi_{k_{max}}$). Assume $k_{max} = 3$, i.e., we find large itemsets up to size $k = 3$, the resulting large itemsets once we execute NFP-mine() are: k=1: $\langle c:3, d:3, a:2, b:2, e:2\rangle$; k=2: $\langle cd:3, ac:2, ad:2, bc:2, bd:2\rangle$; k=3: $\langle acd:2, bcd:2\rangle$.

Up to this point, there may be cases that the number of k-itemsets, where $k \geq 2$, is less than N because the threshold, ξ, found in the building phase is not small enough. For the above example, there are only two large 3-itemsets found. There are not enough 3-itemsets, since $N = 3$. Therefore, a smaller ξ should be used in order to mine more itemsets in the mining phase. However, this method is exhaustive and we use a loop-back method to handle the problem. As long as there are not enough itemsets for certain level(s), we decrease ξ by a factor f, $0 < f < 1$, such that $\xi_{new} = \xi \times f$. Let us call the original ξ value ξ_{old}. We call NFP-build() to update the FP-tree in an incremental manner (see the discussion of incremental tree building below). Then we can call NFP-mine() to mine itemsets of supports $\geq \xi_{new}$. This is our basic idea for LOOPBACK.

f	a factor used to reduce ξ in the loop-back
ξ_{old}	support threshold for all itemsets in the previous round

After we have sorted the supports of items we can find the N-most interesting 1-itemsets. We initialize the support threshold value, ξ. For LOOPBACK approach, we set ξ to be the support of the N-th sorted largest 1-itemset.[1]

[1]However, if $k \geq N$, we are sure that using only the large 1-itemsets for the initial FP-tree is not enough to form itemsets of size $\geq N$. Therefore, we should choose the support of the $(k+1)$-th 1-itemset as the threshold during the building phase of the initial round.

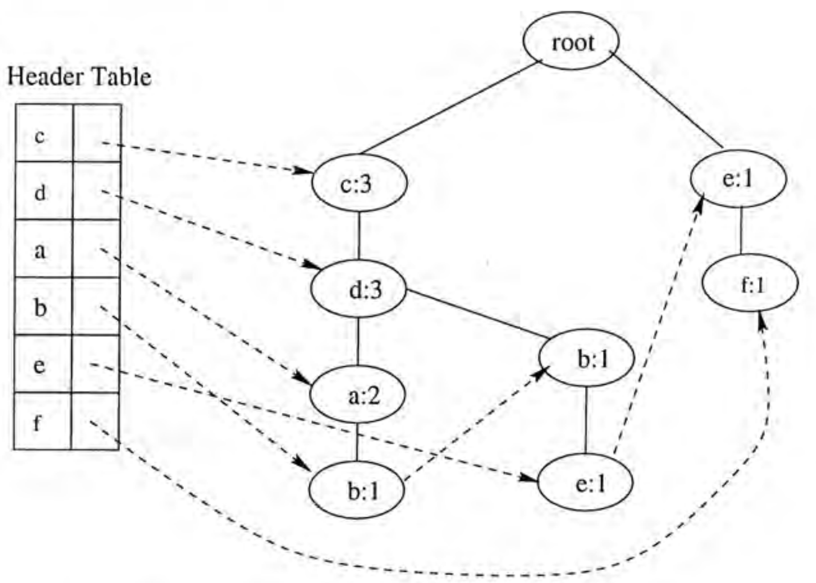


Figure 5.7: FP-tree after round 2.

Some Pruning Considerations

We use an incremental approach to build or update the FP-tree. For each loop back (round), there is no need to rebuild the whole tree using the building phase. We only need to consider transactions which contain any new added 1-itemsets (itemsets with support smaller than ξ_{old} in previous round, but larger than or equal to ξ_{new} in the current round) in the current round. We only need to insert new branches to or modify the counts of existing branches of the FP-tree built in the previous round. Figure 5.7 shows the resulting FP-tree after round 2. Only one new branch is created for item f and its corresponding horizontal link from the header table is established.

Skipping k-itemsets: In each round of loop-back, we keep itemsets which have been found so far from previous rounds, and use both ξ_{old} and ξ_{new} to filter the itemsets found during the mining phase. We do not need to generate itemsets whose supports $\geq \xi_{old}$, because they have already been found and kept in previous rounds.

Each time we loop back and redo NFP-mine(), we do not need to consider any more k-itemsets if we have already found N or more number of itemsets of size k because we have found N-most interesting k-itemsets which have supports $\geq \xi_{old}$. If we continue to consider k-itemsets we shall only discover k-itemsets of smaller support values.

Lemma 4 If N or more k-itemsets are found in a current round, then the N-most interesting k-itemsets are found.

Skipping old items: In the mining phase, we do not need to consider all the items in the header table. We only need to consider items which are newly added to the header table in the current round as well as old items which were the base items of some itemsets having supports $< \xi_{old}$ in previous round.

Using the previous example in Figure 5.7, since we cannot get enough 3-itemsets in the first round, we use a smaller threshold, $\xi_{new} = 1$, in the building phase to insert an new item, f, to the header table. Therefore, we get a header table having items c, d, a, b, e, f in the second round. However, we only need to form the conditional pattern base of each of items b, e, f. We consider item f because we have not considered any itemsets which consist of f in the previous round(s). We consider items b and e because there exist itemsets $\{cab : 1, dab : 1, dbe : 1, cbe : 1\}$ which have not been selected in the first round since the threshold was 2.

5.2.3 A Build-Once and Loop-Back Approach, BOLB

Finally we consider a hybrid approach of BOMO and LOOPBACK, in which we build the complete FP-tree only once, but in the mining we apply the technique of looping-back.

5.2.4 Discussion

In the LOOPBACK approach, we incrementally update the FP-tree in each loop back to build a new FP-tree with items of supports $\geq \xi_{new}$. This avoids building the entire tree, but will introduce overhead in the loopback. With

BOLB and BOMO, we build a complete FP-tree with all items in the transactions. This method ensures that the building phase takes place only once during the whole mining process. With BOLB and LOOPBACK, each time we loop back to redo mining, there would be redundant work in forming conditional patterns since some of the patterns should have been formed in previous rounds and have to be reformed in the current round. The BOMO algorithm eliminates this redundant work. Another problem with LOOPBACK and BOLB is that we need to determine a factor f to decrease the threshold ξ for the next round. There is no need of such a parameter in BOMO.

5.3 Generalization: Varying Thresholds N_k for k-itemsets

In the previous consideration, we fix a number N on the resulting number of itemsets for itemsets of all sizes considered. However, in general frequent itemsets will be more numerous for smaller itemsets and less so for itemsets of greater size. It would be more flexible if we allow the user to specify possibly different numbers, N_k, of resulting k-itemsets for different values of k. This is a generalization of the original problem definition. With the generalization, we need to modify the three algorithms we proposed before. However, the change is very minor. We only have to change the meaning of ξ_k, $1 \leq k \leq k_{max}$. ξ_k will be the support of the N_k-th most frequent k-itemset discovered so far, $\xi_k = 0$ if the number of k-itemsets discovered so far is less than N_k. The other parts of the algorithms remain intact.

5.4 Performance Evaluation

We compare the performance of our new approaches with the Itemset-Loop algorithm and a modified version of FP-tree algorithm for mining N-most interesting itemsets. All experiments are carried out on a SUN ULTRA 5_10 machine running SunOS 5.6 with 512MB Main Memory. Both synthetic and real datasets are used.

Real Data: Two sets of real data are from the census of United States 1990 [26]. One is a small database **(tiny.dat)** with 77 different items and 5577 tuples, another set is a large database **(small.dat)** with 77 different items and 57972 tuples.

Table 5.2: Parameter setting.

Parameter	Description	Value
$\|D\|$	Number of transactions	100K
$\|T\|$	Average size of the transactions	5, 10, 20
$\|I\|$	Average size of the maximal potentially Large itemsets	2, 4, 6, 8, 10
$\|L\|$	Number of maximal potentially large itemsets	2000
M	Number of items	1K
C	Correlation between patterns	0.25

Table 5.3: Synthetic data description.

Dataset	$\|T\|$	$\|I\|$	$\|D\|$
T5.I2.D100K	5	2	100K
T20.I6.D100K	20	6	100K
T20.I8.D100K	20	8	100K
T20.I10.D100K	20	10	100K

Synthetic Data: Several sets of synthetic data are generated from the synthetic data generator in [12]. The generator follows the data generation method in [7] with the parameter setting and datasets shown in Table 5.2 and Table 5.3.

For each dataset (real or synthetic), we perform the experiment under different values of N in the N-most interesting itemsets. The different values of N are 5, 10, 15, 20, 25, and 30. We set k_{max} to be 4 or 10, i.e. we mine itemsets up to size 4 or 10, hence k-itemsets are mined for $1 \leq k \leq 4$, or $1 \leq k \leq 10$. We compare the performance of our new approaches with the Itemset-Loop algorithm. We also evaluate the performance of the FP-tree algorithm when a known (optimal) threshold, ξ_{opt}, is given, i.e. a threshold

Table 5.4: Ideal thresholds, ξ_{opt} (%), for different datasets with $k_{max} = 4$.

Dataset	N					
	5	10	15	20	25	30
tiny.dat	9.79	7.02	6.36	6.06	5.73	5.48
small.dat	21.42	19.39	18.04	14.13	12.58	10.50
T5.I2.D100K	0.30	0.29	0.27	0.25	0.23	0.22

Table 5.5: Ideal thresholds, ξ_{opt} (%), for different datasets with $k_{max} = 10$.

Dataset	N					
	5	10	15	20	25	30
tiny.dat	0.23	0.21	0.19	0.17	0.17	0.17
small.dat	7.46	7.46	6.9	6.9	6.3	6.3
T20.I10.D100K	0.49	0.49	0.46	0.54	0.51	0.49

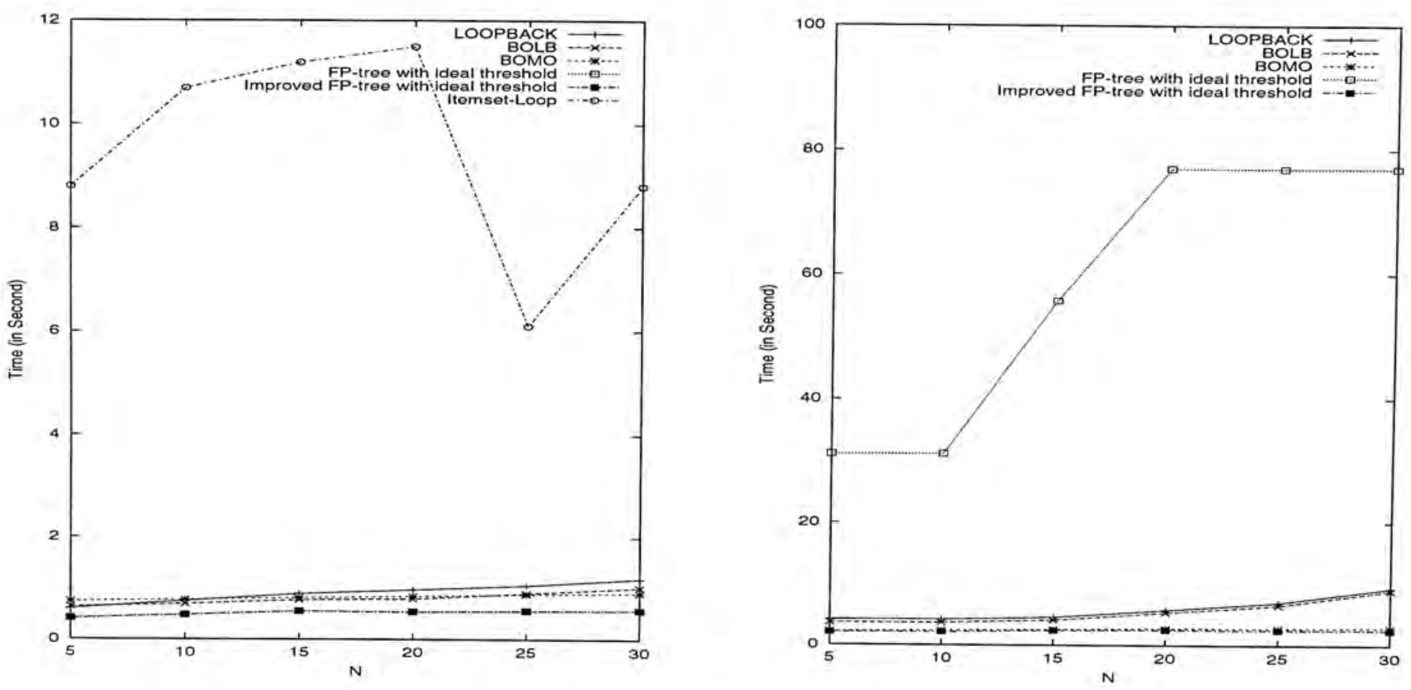
which is just small enough to make sure that all the required N-most interesting k-itemsets are of supports greater than or equal to this threshold, this method does not need any loop-back. Tables 5.4 and 5.5 show the ideal thresholds for different datasets.[2] We measure the total response time as the total CPU and I/O time used for both building and mining phases. Each data point plotted in the graphs determined by the mean value of several runs of the experiment.

First, we compare the performance using different datasets. For each level k, we find N most interesting itemsets. The threshold decrease rate, f, is set to 0.2. We find that all our approaches outperform Itemset-Loop and have similar performance as the FP-tree algorithm with ideal threshold, ξ_{opt}, for real datasets. The execution times for Itemset-Loop in Figures 5.8 (b) and 5.9 (b) are too large to be shown. For example, for tiny.dat with N = 20, k_{max} = 10, the execution time of Itemset-Loop is about 32000 sec, which is several orders of magnitude greater than our methods.

It may be expected that with ξ_{opt}, the original FP-tree algorithm [23] discussed in Section 4.2.4 should be the fastest because it does not require any loop back and it can build the smallest necessary initial FP-tree. However, Figure 5.8 (b) shows that this method (denoted by **FP-tree with ideal threshold**)

[2]The support values listed here are in terms of the percentage of the transactions that contain an itemset.

requires more total response time than our methods. The reason is that the number of k-itemsets of supports $\geq \xi_{opt}$ is too large for some of k so that a large number of itemsets are generated. From experiment, we find that if ξ_{opt} is too small and N is large, then the original FP-tree algorithm does not perform well even with an ideal threshold. An example is the dataset T5.I2.D100K with $N = 20$, $k_{max} = 10$ and $\xi_{opt} = 0.002\%$, the execution time of FP-tree algorithm (180000 sec) is more than 100 times of that of our methods (about 1400 sec for LOOPBACK).



(a) $k_{max} = 4$ (b) $k_{max} = 10$

Figure 5.8: Real dataset: tiny.dat.

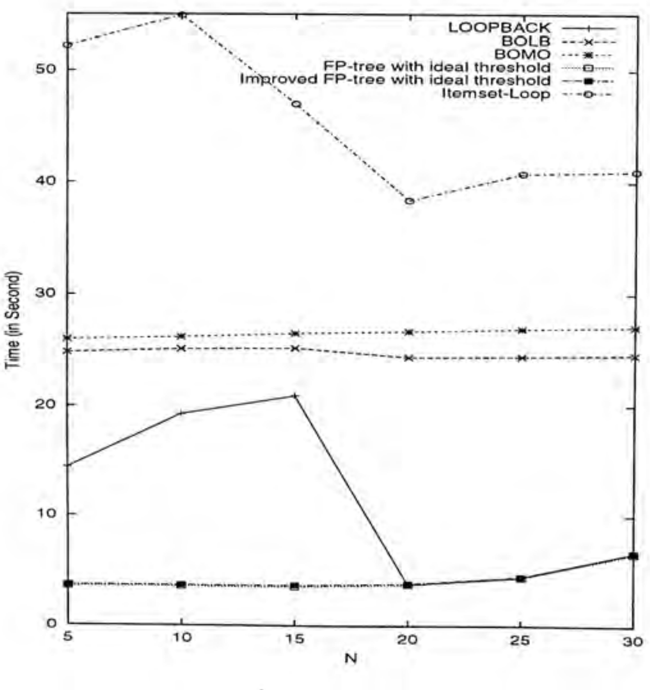
To improve the performance of the original FP-tree method, we use the set of thresholds, ξ_k, $1 \leq k \leq k_{max}$, and dynamically update ξ_k as in our NFP-tree algorithm for the pruning of small itemsets. We denoted this method as **Improved FP-tree with ideal threshold**.

Figure 5.9 shows that LOOPBACK is faster than BOLB and BOMO for the small real dataset. The whole mining process only requires a few number of loopbacks, and ξ_{opt} is large. Therefore, building a complete FP-tree in

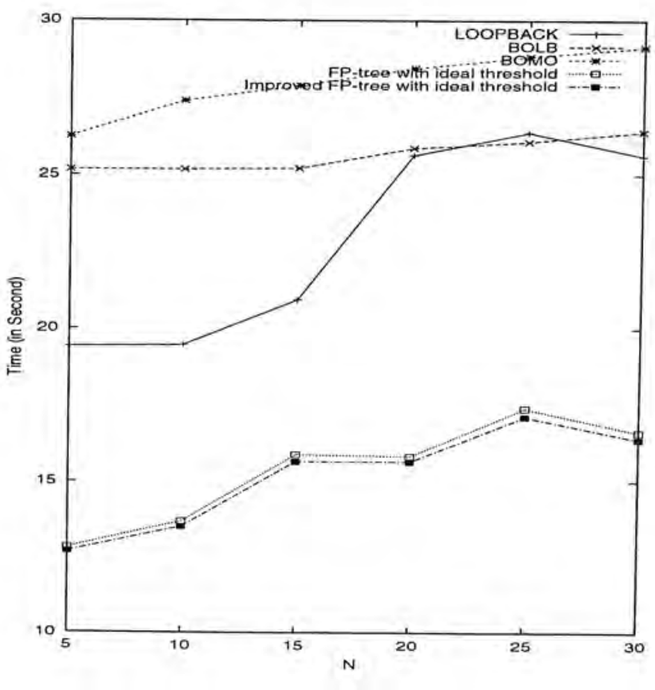
BOLB and BOMO becomes an overhead relative to the small trees built in LOOPBACK, and the enhancements in BOMO are not significant.

For synthetic data, since the datasets are much larger, it requires a longer time to scan the database in the building phase of each loop. Also, the mining phase requires much more time than the building phase. The avoidance of redundant work in BOMO becomes significant. Therefore, LOOPBACK takes the longest time to complete the mining. See Figures 5.10 to 5.11.

Among the new approaches, BOMO is the fastest and is comparable to the Improved FP-tree method. BOMO does not require any loopbacks as required in LOOPBACK and BOLB and hence eliminate any redundant work in both building and mining. All experiments show that the main memory requirement for storing the complete FP-tree is less than 100MB. This shows that BOMO is a good choice for mining N-most interesting itemsets.

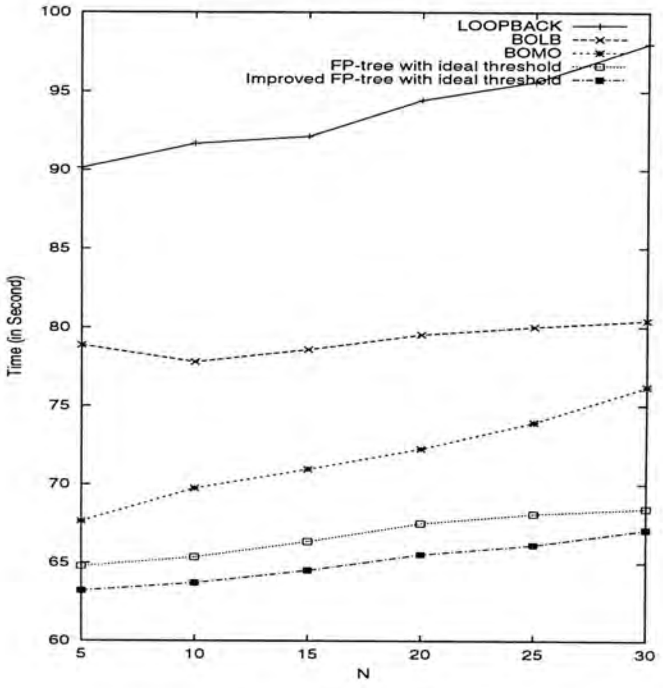


(a) $k_{max} = 4$

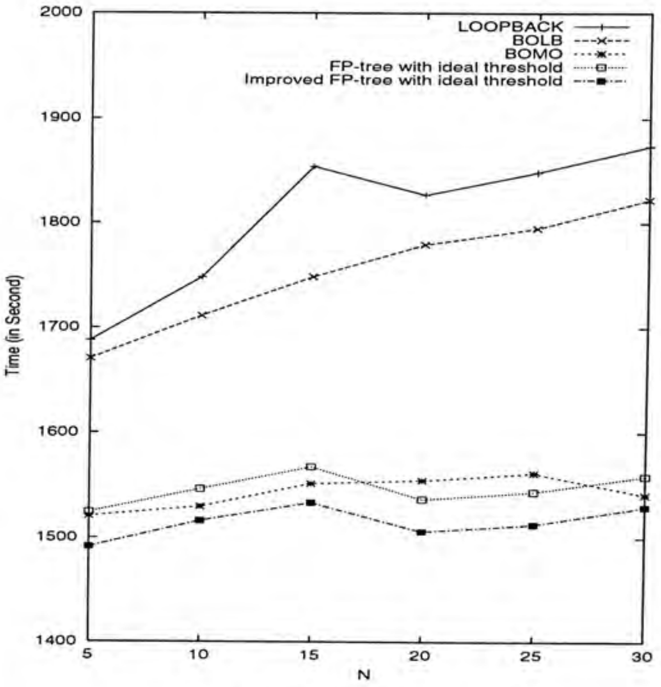


(b) $k_{max} = 10$

Figure 5.9: Real dataset: small.dat.

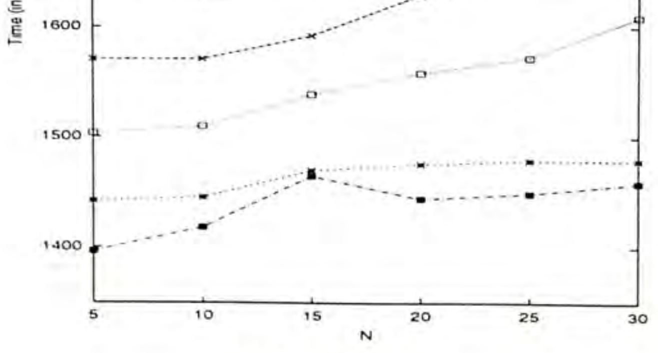


(a) T5.I2.D100K

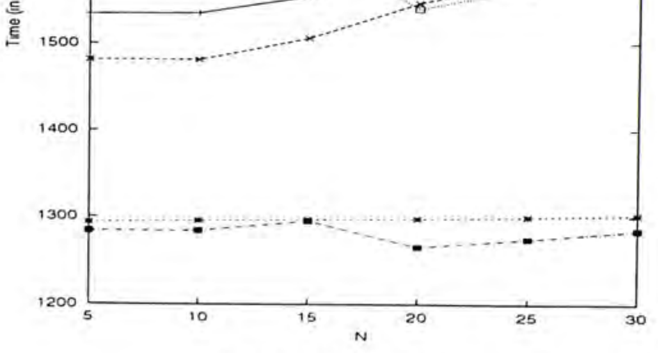


(b) T20.I6.D100K

Figure 5.10: Synthetic datasets with $k_{max} = 4$.



(a) T.20.I8.D100K



(b) T.20.I10.D100K

Figure 5.11: Synthetic datasets with $k_{max} = 10$.

5.4.1 Generalization: Varying N_k for k-itemsets

Next, we test the performance of our algorithms under different N for different values of k. We set N to be 30, 27, 24, ..., 3 for $k = 1, 2, 3, \ldots, 10$ respectively when $k_{max} = 10$. We set N to be 30, 20, 10, 5 for $k = 1, 2, 3, 4$ respectively when $k_{max} = 4$. Table 5.6 shows the execution times for different methods. Again, BOMO is comparable to the improved FP-tree algorithm.

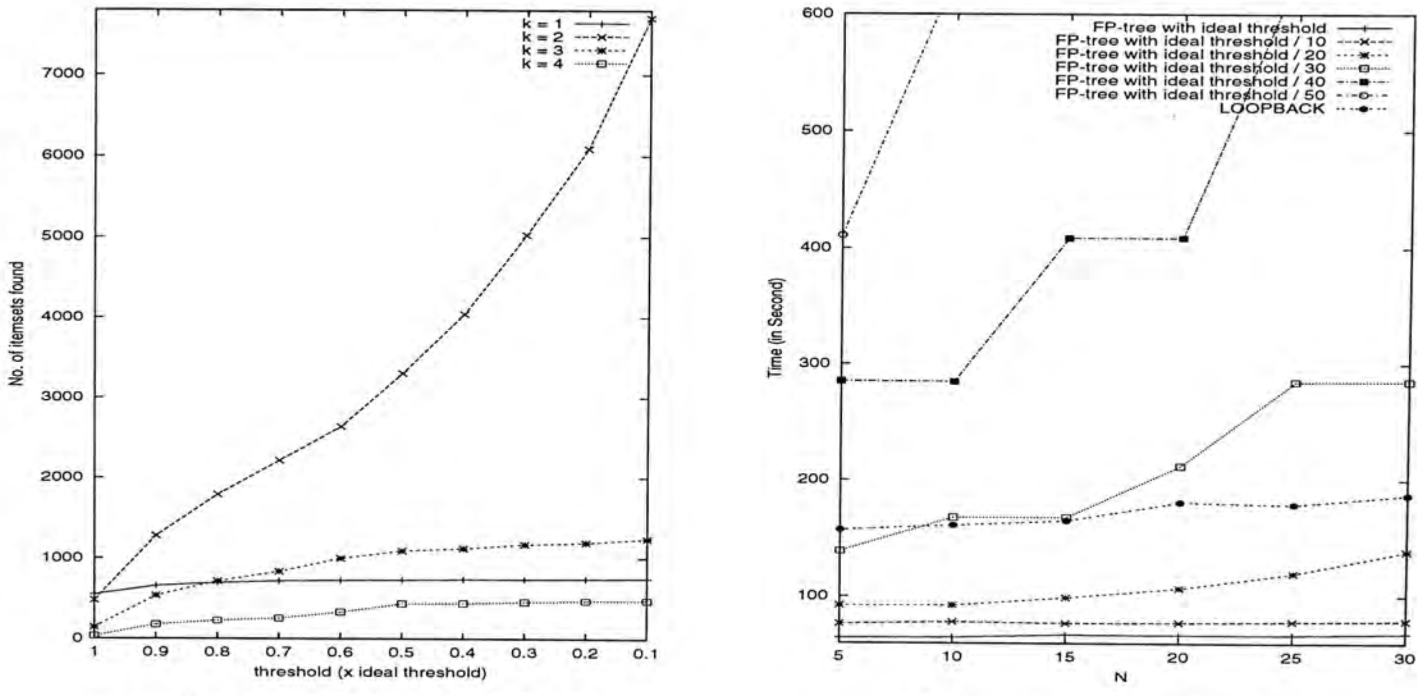
Table 5.6: Execution time (sec) with different N for different k.

Dataset	k_{max}	LOOPBACK	BOLB	BOMO	Improved FP-tree with ideal threshold
tiny.dat	4	0.5	0.6	0.8	0.5
tiny.dat	10	2.7	2.3	2.3	1.7
small.dat	4	6.6	25.0	26.8	6.6
small.dat	10	25.6	26.0	29.2	7.3
T5.I2.D100K	4	96.4	81.3	70.4	60.5
T20.I6.D100K	4	1829.0	1860.5	1505.6	1446.0
T20.I8.D100K	10	1763.5	1754.1	1480.2	1378.4
T20.I10.D100K	10	1687.5	1600.0	1332.0	1279.5

5.4.2 Non-optimal Thresholds

In real applications, it is generally very difficult to pick an optimal support threshold. If the guess is too large, then the conventional approach would not get the proper results, and therefore our approach is definitely much better. We evaluate the effect of guessing a non-optimal threshold that is too small for the FP-tree method. We decrease ξ_{opt} by different factors and use the resulting values as the non-optimal thresholds for the algorithm. Figure 5.12 (a) shows the increase in the number of itemsets for each size k and the total response time when the non-optimal threshold gets smaller. Figure 5.12 (b) shows that when user guess a non-optimal threshold, the LOOPBACK algorithm can greatly outperform the FP-tree method.

Similarly, we evaluate the effect of using a large threshold ($\geq \xi_{opt}$). We increase ξ_{opt} by 2, 4, 6, and 8, and use these values as the thresholds for the



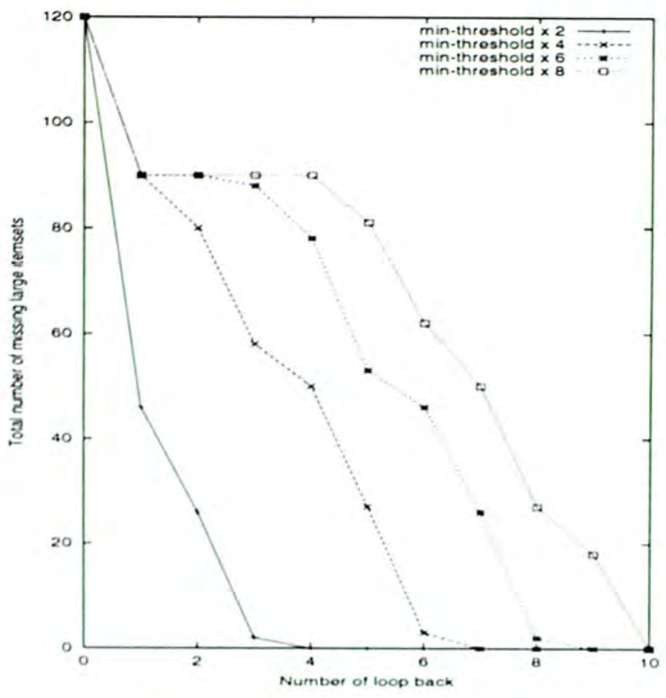
(a) number of itemsets for k_{max}=4, N=30 (b) response time for k_{max}=4

Figure 5.12: Small thresholds, synthetic dataset: T5.I2.D100K.

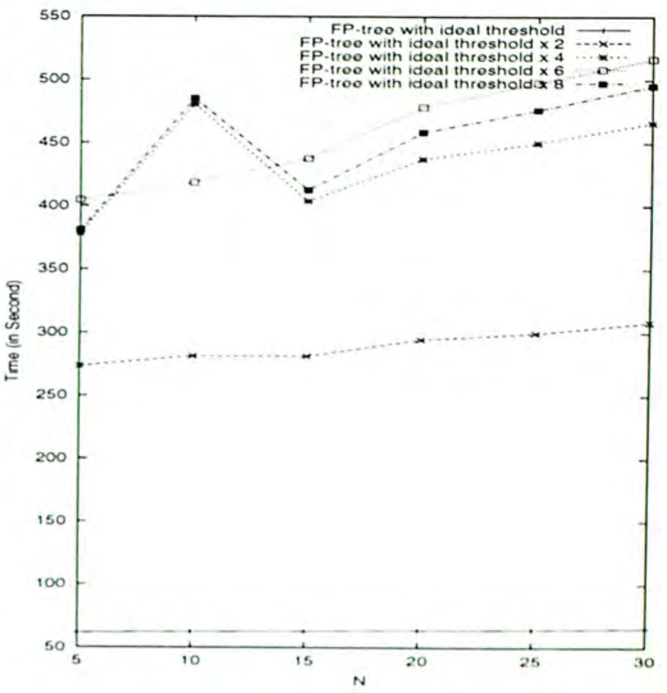
FP-tree method. Figure 5.13 (a) shows that the total number of missing large itemsets in each round in the mining phase. Figure 5.13 (b) shows the different total response time for different non-optimal thresholds.

5.4.3 Different Decrease Factors, f

We test the use of different decrease factor, f, for LOOPBACK as shown in Figure 5.14. In each loop-back, we decrease the threshold by a certain factor (decrease factor) which is ranged from 0.1 to 0.8. In general, the smaller the f, the faster is our algorithm as the number of loop-back is reduced. However, if f becomes smaller, the difference between the new and old thresholds (ξ_{new} and ξ_{old}) in each loop becomes large, the number of itemsets with supports that fall between these two thresholds increases and therefore the pruning effect of ξ becomes insignificant. This explains why there is an increase in execution time if f is too small as shown in Figure 5.14.

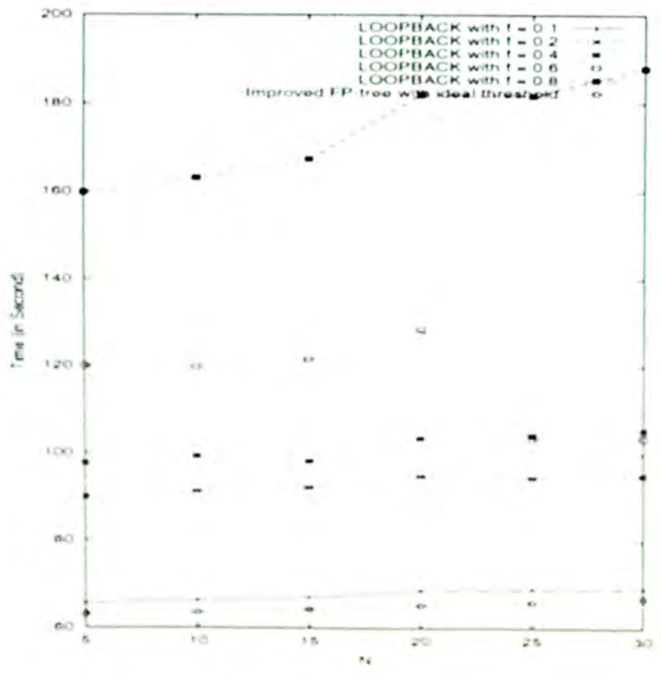


(a) number of missing itemsets for k_{max}=4, N=30

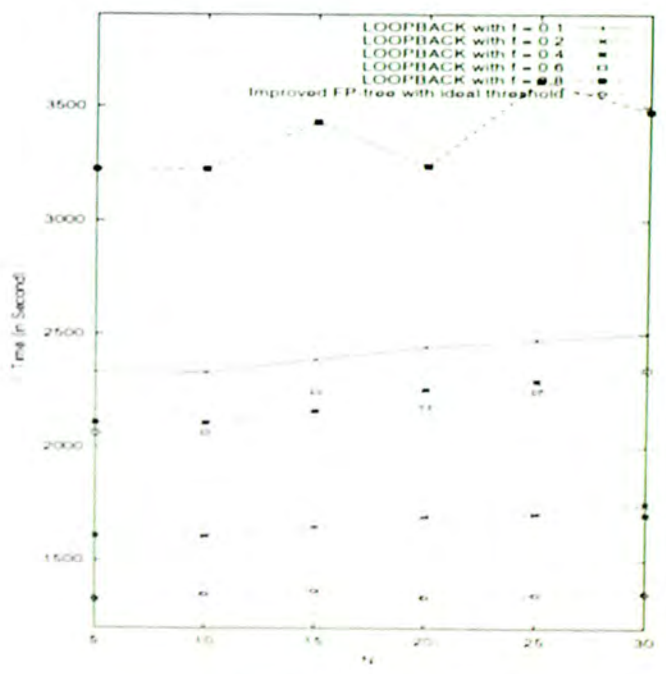


(b) response time for k_{max}=4

Figure 5.13: Large thresholds, synthetic dataset: T.5.I2.D100K.



(a) T5.I2.D100K with k_{max}=4



(b) T20.I8.D100K with k_{max}=10

Figure 5.14: Different decrease rate, synthetic datasets.

5.5 Conclusion

In this chapter, we study the problem of mining N-most interesting k-itemsets. This method can solve an important problem in the conventional mining of association rules, namely, it is difficult to specify the support threshold. In addition, experiments show that our proposed method outperforms the previous Itemset-Loop algorithm by a large margin and it is also comparable to the FP-tree algorithm, even when given an ideal threshold, and equipped with different proposed improvements. In the original problem definition of large itemsets, even an ideal threshold may induce a lot of unnecessary computation, since the support threshold is for itemsets of all sizes. For N-most interesting itemsets, we can make use of the different support thresholds for itemsets of different sizes. Therefore, we find that in some cases, the original FP-tree algorithm, even with an ideal threshold, can be an order of magnitude slower than our proposed algorithms. For thresholds that are too small for the original FP-tree algorithm, our proposed methods can have a much superior performance in both the time efficiency and the generation of useful results. For thresholds that are too large, the original FP-tree algorithm will not give a proper answer, in fact it may not return any itemsets.

In the future we may investigate other methods or data structures that can solve the problem of mining N-most interesting itemsets, or the generalized version of this problem. Finally, we are interested in the problem of mining frequent itemsets with constraints. For example, we can specify that the N most interesting itemsets among all itemsets containing item A are required. The rationale behind the having such constraints of item "A" can be found in [57], in which a support threshold version of the problem is studied. We shall investigate this problem in the next chapter.

Chapter 6

Mining Interesting Itemsets with Item Constraints

In previous chapter, we have mentioned the problem of setting a minimum support threshold. Recently, constraint-based mining is proposed for the minimum support problem for mining interesting itemsets. However, the user still need to supply the support thresholds. We propose a more practical approach, which is mining the "N-most interesting k-itemsets with item constraints", for k up to a certain k_{max} value. Generally, it is more straightforward for users to determine N and k_{max} together with a set of constraints specifying different characteristics of different items. We propose the *Double FP-trees* algorithm for the constraint-based mining without the support thresholds. Experiments show that our proposed algorithm is highly efficient in generating interesting itemsets.

6.1 Introduction

Conventional association rules mining algorithms [7, 11, 38, 47, 49, 23] use only a single user-specified minimum support. In reality, the minimum support is not uniform [16, 57]: (i) Interesting itemsets (e.g. deviations and exceptions) may have smaller supports than non-interesting itemsets (general trends). (ii)

There may be different support requirement for itemsets having items of different supports. (iii) Item absence usually has higher support than item presence. (iv) Items of higher concept levels usually have higher support than items of lower concept levels. (v) Setting the threshold value is typically hard: if the threshold is set too large, we may miss some interesting itemsets; if it is set too small, many itemsets will be returned and useless to the users.

In the case of different minimum supports for different itemsets, the Apriori-generation of a frequent itemset from its frequent subsets is lost. A naive way to handle the non-uniform supports is to apply conventional algorithms at the lowest minimum support ever specified and then filter the result using higher minimum supports. This method is inefficient since it will generate many candidates that are later discarded. [57] proposes *support constraints*, described in Section 4.3.5, as a way to specify general constraints on minimum support. It employs a support pushing technique that allows the highest possible minimum support to be pushed so as to tighten up the search space while preserving the essence of Apriori. However, the algorithm still requires a user specified support threshold.

In previous chapter, we introduce the mining of N-most frequent itemsets with our *BOMO* algorithm to handle the problem of setting support thresholds. In this chapter, we define a more practical problem of finding N-most frequent itemsets with constraints which are interesting to users. We propose our *Double FP-trees* algorithm in Section 6.2.2. Experimental results are shown in Section 5.4.

If we mine the N-most interesting itemsets, though we have removed the requirement of support threshold, the uniform threshold on the number of itemsets is still a restriction. As pointed out in [57], a more desirable set up is to allow users to set different thresholds for different items or itemsets. An example that is given in [57] is that in a supermarket scenario, the itemset {bread, milk} is usually much more frequent than the itemset {food processor, pan}. However, the latter is a valuable itemset even though the occurrence is less frequent. Here we aim at achieving this flexibility.

Consider a set of items, I, partitioned into several bins $B_1, B_2, \ldots, B_m$ where each bin B_i contains a set of items in I. We define **item constraint** in a way similar to the support constraint defined in [57], however, instead of a constraint by support, we set a constraint by the number of itemsets. Item constraint (IC_i) has the form: $IC_i(B_{i_1}, \ldots, B_{i_s}) = N_i$, where $s \geq 0$, and B_{i_j} and B_{i_l} may be equal.

We adopt the concepts of open and closed interpretations in [57]. An itemset (or pattern) I **matches** a constraint IC_i in the **open interpretation** if I contains at least one item from each bin in IC_i and these items are distinct. An itemset I matches a constraint IC_i in the **closed interpretation** if I contains exactly one item from each bin in IC_i and these items are distinct, I does not contain any other items.

With open interpretation, consider the set X of all k-itemsets containing at least one item from each B_{i_j} for a given IC_i, for $|IC_i| \leq k \leq k_{max}$, where k_{max} is a user defined parameter for the maximum size of an itemset to be mined. We sort these k-itemsets according to their supports in descending order. Let the N_i-th greatest support be ξ, then we say that all k-itemsets in X with support not less than ξ are **interesting** for the constraint. We call these itemsets the **N_i-most interesting itemsets for IC_i in the open interpretation**.

For closed interpretation, consider the set X of all k-itemsets containing exactly one item from each B_{i_j} for a given IC_i, therefore $k = |IC_i|$. We sort these k-itemsets according to their supports in descending order. Let the N_i-th greatest support be ξ, then we say that all k-itemsets in X with support not less than ξ are **interesting** for the constraint. We call these itemsets the **N_i-most interesting itemsets for IC_i in the closed interpretation**.

Given IC_i, and N_i for $1 \leq i \leq c_{max}$, where c_{max} is the number of constraints, we would like to find the N_i-most interesting itemsets for each IC_i, in either the open interpretation, or the closed interpretation. Note that k_{max} is also necessary for the open interpretation.

Table 6.1: Partition of items into different bins.

Bin	Items
B_1	a
B_2	b, c
B_3	d, e
B_4	f

Table 6.2: Item constraints.

Constraint	Bins	N_i
IC_1	B_1, B_2	N_1
IC_2	B_3, B_4	N_2
IC_3	B_2	N_3
IC_4	B_1, B_2, B_3	N_4
IC_5	B_1, B_2, B_4	N_5
IC_6	B_1, B_2, B_3, B_4	N_6

Consider that we partition a set of items into four bins as shown in Table 6.1 and specify the constraints in Table 6.2. Consider itemsets $I_1 = (acd)$, $I_2 = (abf)$, $I_3 = (bc)$, the corresponding **bin patterns** for I_1, I_2, I_3 are (B_1, B_2, B_3), (B_1, B_2, B_4), and (B_2, B_2) respectively. We say that I_1 matches IC_1, IC_3, and IC_4 in the open interpretation, and matches only IC_4 in the closed interpretation; I_2 matches IC_1, IC_3, and IC_5 in the open interpretation, and matches IC_5 in the closed interpretation; I_3 matches IC_3 in the open interpretation, but matches none of the constraints in the closed interpretation.

6.2 Proposed Algorithms

In this section we propose two algorithms for mining N-most interesting itemsets with item constraints. The first algorithm is a straightforward modification of the BOMO algorithm introduced in previous chapter. The second algorithm improves on the first one by maintaining the constraints information with a second FP-tree.

6.2.1 Single FP-tree Approach

With the closed interpretation, an itemset that matches a given constraint IC_i must be of size $|IC_i|$ since it contains exactly one element from each bin in IC_i. Therefore, we need only one dynamic support threshold ξ_i for such itemsets. The value of ξ_i is set to be the support of the itemset that matches IC_i which has the N_i-th highest support among all such itemsets that match IC_i discovered so far. We also set a global threshold,

$$\xi = min(\xi_1, \xi_2, \ldots, \xi_{c_{max}}) \tag{6.1}$$

With the open interpretation, an itemset that matches IC_i can have size equal to or greater than $|IC_i|$. Therefore the itemset size ranges from $|IC_i|$ to k_{max}. We assume that k_{max} is greater than the size of all item constraints $|IC_i|$. For each constraint IC_i and each possible itemset size k, we use a support threshold ξ_{ik} for pruning. The value of ξ_{ik} is set to be the support of the k-itemsets that matches IC_i which has the N_i-th highest support among all such k-itemsets that match IC_i discovered so far. Any newly encountered k-itemset that satisfies IC_i but is smaller than ξ_{ik} is not considered interesting. ξ_{ik} is initialized to zero, for all possible values of i and k. We also set a global threshold,

$$\xi = min(\text{values of } \xi_{ik} \text{ for all possible } i \text{ and } k) \tag{6.2}$$

We use the BOMO algorithm as the basic architecture and apply a simple constraint matching mechanism. We build an FP-tree for all the items in the transactions and mine for interesting itemsets based on the BOMO algorithm. ξ is used as the support threshold for building conditional FP-trees. With the closed interpretation, an itemset, I, having support $\geq \xi_i$ is to be matched with IC_i. If there is a match, we add I to the result set and update ξ_i. With the open interpretation, if $|IC_i| \leq k \leq k_{max}$, then an k-itemset, I, having support $\geq \xi_{ik}$ is to be matched with IC_i. If there is a match, we add I to the result set and update ξ_{ik} if necessary.

6.2.2 Double FP-trees Approaches

Using the single FP-tree approach, we require either a lot of memory space or disk space for the memory-based mining or disk-based mining respectively. In addition to the FP-tree, there is a set of user specified constraints for matching. If the number of constraints is large, the single FP-tree approach will consume a lot of computation in the matching process and storage. We try to use a compact data structure to store the constraints. We propose to employ an FP-tree for storing the set of constraints since it is a highly compact structure. We observe three advantages in doing this: (1) The FP-tree is usually substantially smaller than the original constraint set [23] since constraints may share common bins. Hence we save on the storage. (2) With this approach, we can perform the matching of itemsets and constraints in a more efficient way instead of matching the set of constraints one by one for a given itemset. (3) We can also have a better support pruning strategy instead of simply employing the lower bound technique as shown in Equations 6.1 and 6.2. We propose to employ a special order of scan on the FP-tree storing the constraints so as to speed up the pushing of the dynamic minimum support threshold as well as using a pruning strategy to tighten the threshold value.

Closed Interpretation

We build an FP-tree, **transaction FP-tree**, for the items in the transactions. We can also build another FP-tree, **constraint FP-tree**, for the bins in the

Table 6.3: A transaction database.

TID	Items	Items sorted by support
001	a, b, c, d	c, d, a, b
002	b, c, d, e	c, d, b, e
003	a, c, d	c, d, a
004	e, f	e, f

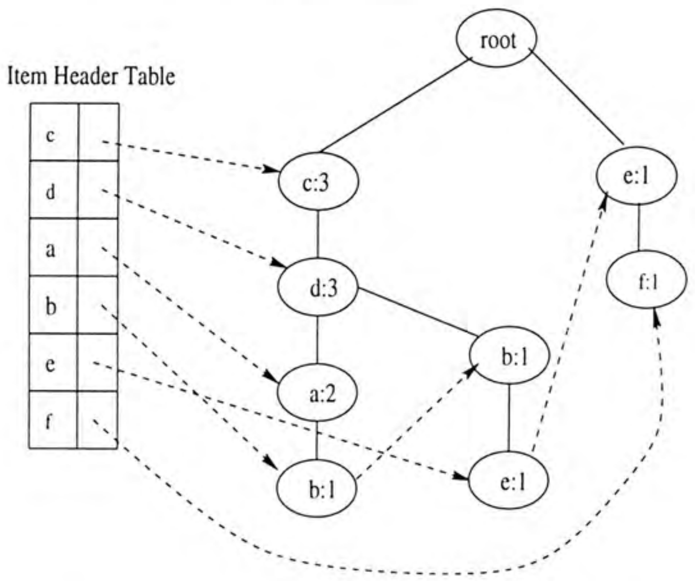


Figure 6.1: An FP-tree for a set of transactions.

constraints. The first obvious advantage is that it greatly reduces the memory size for storing the constraints by sharing common bins as much as possible. For the same reason it will also reduce the computation time for matching itemsets with the constraints.

Table 6.4: Elements in a constraint FP-tree node.

Notation	Description
$n.bin$	The bin represented by the node, n.
$n.c$	$n.c = i$ if the constraint IC_i is represented by the bins from the root to node n.
$\xi(n)$	If $n.c = i$, $\xi(n)$ is the N_i-th largest support of itemsets matching IC_i so far.
$\xi_{min}(n)$	The minimum value of ξ among the nodes of the subtree rooted by a node n.
$node\text{-}link$	The next node in the FP-tree carrying the same bin, or null if there is none.

Figures 6.1 and 6.2 shows the FP-trees constructed from the database in Table 6.3, and constraints in Section 6.1. The node structure of a constraint FP-tree is shown in Table 6.4, which is different from the node structure of a

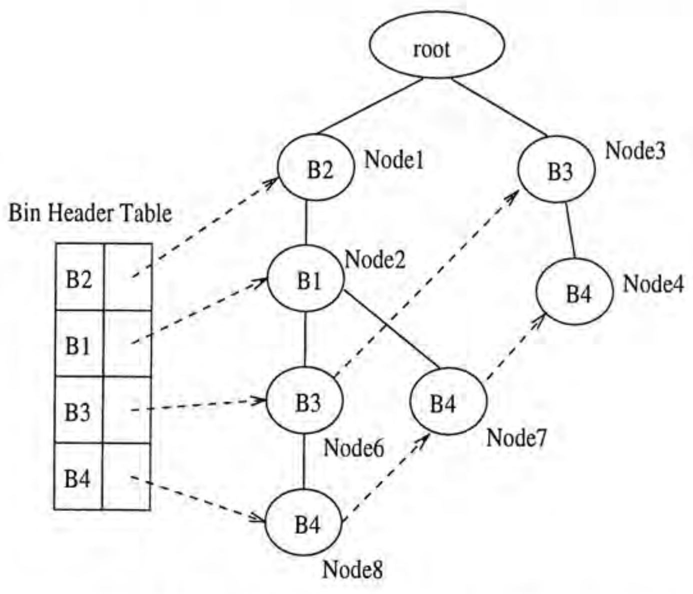


Figure 6.2: An FP-tree for a set of constraints.

transaction FP-tree described in Section 4.2.4.

In constructing the constraint tree, we first scan the constraint set and find the support for each bin. Then, we create an empty constraint FP-tree with root = "NULL". For each constraint, IC_i, the bins are sorted in descending order of their supports. We insert all the bins in the constraint as a path to the tree in a way similar to the transaction FP-tree, except we do not need to worry about the counts since there will not be any repeated constraints. Common prefixes between patterns of different constraints share the same tree path, otherwise we create a path with new nodes. For the node, *Node*, representing the last bin of the pattern IC_i, we set *Node.c* to i denoting IC_i. We say that *Node* **corresponds to** constraint IC_i. If no such constraint can be determined for *Node*, then *Node.c*=0, indicating it does not correspond to any constraint. For each node *Node*, $\xi_{min}(Node)$ is initialized to zero. If *Node.c*=0, then we set $\xi(Node) = MAXINT$, where $MAXINT$ is a large number greater than the number of transactions in the given database. This will indicate that the node does not get involved in setting the overall global threshold ξ. If $Node.c \neq 0$, then we set $\xi(Node) = 0$ as an initial value.

Lemma 5 A bin pattern formed by the bins from the root to any node of the constraint FP-tree can only match at most one constraint in the closed interpretation.

Using the previous example, there is no constraint matching the pattern represented by the bins from the root to its right child $Node_3$ representing the bin pattern (B_3), therefore the element $Node_3.c$ is undefined (zero). On the other hand, element $Node_2.c = 1$ because the pattern (B_2, B_1) represented by the root node down to $Node_2$ matches $IC_1(B_1, B_2)$.

The basic idea of our Double FP-trees approach is that we visit each item a in the header table of the conditional transaction FP-tree, T, of a base pattern α, and form the conditional transaction FP-tree for the base pattern $a \cup \alpha$ using the minimum support threshold from the constraint FP-tree, T_{IC}. Note that the initial transaction tree can be treated as a conditional transaction FP-tree with $\alpha = \phi$.

Update of ξ_{min} at the root node: This minimum support threshold can be set to ξ_{min} of the root of T_{IC} and is exactly equal to ξ in Equation 6.1. This minimum support threshold can be increased during the mining process, whenever we found an itemset interesting for a constraint IC_i, by updating ξ of the corresponding constraint FP-tree node and pushing ξ_{min} of the ancestors of this node till the root.

We perform the above steps recursively until a single path conditional FP-tree is obtained for a base pattern. Then we generate each possible itemset combination from the single path and try matching the itemset with the constraints in the constraint FP-tree in a top-down manner: *We sort the bin pattern in the itemset according to the top-down order in the constraint header table. At each tree level*[1] *l, there is at most one node matching the l-th bin in the pattern. If this is the case, we match the next bin with the subtrees of this*

[1] *Level of root is zero.*

node. We say that the itemset matches a constraint IC_i if its sorted bin pattern matches exactly a tree path and the constraint represented by the bottom matched node, n, of this path is IC_i. We include the itemset in the current resulting set if we find a match between the itemset and constraint IC_i, and the support of the itemset is not less than the N_i-th greatest support of the interesting itemsets (for IC_i) found so far, i.e. $\xi(n)$. Figures 6.3 to 6.6 show the Double FP-trees algorithm.

Lemma 6 The value of ξ_{min} at the root of T_{IC} is equal to the value of ξ of Equation 6.1 at any point of the execution.

Example: We illustrate the algorithm in more details using the previous running example. We set $N_1 = N_2 = \ldots = N_6 = 1$, and $k_{max} = 4$. We build the transaction FP-tree and the constraint FP-tree according to the supports of items and bins respectively using functions *NFP-build* (from BOMO) and *T_{IC}-build*. Assume we start from the bottom of the header table of the FP-tree, T, in Figure 6.1, we visit the item f first. The corresponding bin pattern of $f \cup \alpha$, where α $(= \emptyset)$ is the base pattern for T, is (B_4). Since ξ_{min} (=0) of the root of the constraint tree, T_{IC}, is not greater than the support of $f \cup \alpha$ (=1), we try to match the bin pattern B_4 with T_{IC} using function *Mapping*. There is no match between B_4 and the child nodes of the root of T_{IC}, so we discard $f \cup \alpha$. Next we form the conditional FP-tree of f using threshold = ξ_{min} (=0) of root of T_{IC}. We get $(fe : 1)$, which form the sorted bin pattern (B_3, B_4) and it is interesting for IC_2. So we update $\xi_{min}(Node_4) = \xi(Node_4)$ = support of $fe = 1$, and push the value of ξ_{min} upwards to the ancestors of $Node_4$ using function *Update_T_{IC}*.

Next we consider item e in T, the corresponding bin pattern of $e \cup \alpha$ is (B_3). Since support of e (=2) is greater than ξ_{min} of root of T_{IC}, we try matching $e \cup \alpha$ with T_{IC}. Although $Node_3$ matches B_3, $Node_3.c$ is undefined, therefore we discard $e \cup \alpha$. Then we consider the conditional FP-tree of e, which is a single path tree, $root(NULL) \rightarrow (c:1) \rightarrow (d:1) \rightarrow (b:1)$. We get the possible itemset combinations: $(eb:1), (ebd:1), (ebdc:1), (ed:1), (edc:1), (ec:1)$, and the corresponding bin patterns: (B_2, B_3), (B_2, B_3, B_3), (B_2, B_2, B_3, B_3), (B_3, B_3), (B_2, B_3, B_3), (B_2, B_3). Since none of them matches T_{IC}, they are discarded.

Then we consider item b from T, the corresponding bin pattern is (B_2). It matches $Node_1$ which corresponds to IC_3, so $(b:2)$ is an interesting itemset. We add it to the result set, and update $\xi(Node_1)$, $\xi_{min}(Node_1)$ and ξ_{min} of root of T_{IC}. Consider its conditional FP-tree: $root(NULL) \rightarrow (c:2) \rightarrow (d:2) \rightarrow (a:1)$. We get the itemset combinations: $(ba:1), (bad:1), (badc:1), (bd:2), (bdc:2), (bc:2)$, and the corresponding bin patterns: (B_2, B_1), (B_2, B_1, B_3), (B_2, B_2, B_1, B_3), (B_2, B_3), (B_2, B_2, B_3), (B_2, B_2). $(ba:1)$ is interesting for IC_1, we add it to the result set and update $\xi(Node_2)$, $\xi_{min}(Node_2)$ and upwards. $(bad:1)$ is also interesting for IC_4, we include it to the result set and update $\xi(Node_6)$, $\xi_{min}(Node_6)$ and upwards.

Then we consider item a corresponding to the bin pattern (B_1). It is not interesting. Consider its conditional FP-tree: $root(NULL) \rightarrow (c:2) \rightarrow (d:2)$. We get the itemset combinations: $(ad:2)$, $(adc:2)$, $(ac:2)$, and the corresponding bin patterns: (B_1, B_3), (B_2, B_1, B_3), (B_2, B_1). $(adc:2)$ is interesting and replaces (bad:1) in the result set. $(ac:2)$ is interesting and replaces (ba:1) in the result set.

Similarly we can process the items d and c in the item header table. ∎

Although there is an updward pushing of ξ_{min}, ξ_{min} of root of T_{IC} remains unchanged since $\xi_{min}(Node_7)$, and $\xi_{min}(Node_8)$ are still equal to zero. We handle this problem using the following ordering strategy.

Order of Scan: Instead of visiting each item in the item header table sequentially from head to tail or from tail to head of the table for the transaction FP-tree, we employ a new ordering scheme. Starting from the bottom of the bin header table for the constraint FP-tree, for each bin B, we build the

```
Algorithm : DFP-tree algorithm
Input: $D$
Input: $B_1, B_2, \ldots, B_{b_{max}}$
Input: $k_{max}$
Input: $IC = \{IC_1, IC_2, \ldots, IC_{c_{max}}\}$
Input: $N_1, N_2, \ldots, N_{c_{max}}$
Output: $N_i$-most interesting $k$-itemsets for each $IC_i$ where $1 \leq i \leq c_{max}$, $1 \leq k \leq k_{max}$.

(1) Let $result_{ik}$ be the resulting set of interesting $k$-itemsets matching constraint $IC_i$.
    Set $result_{ik} = \emptyset$.
(2) Scan the transaction database, $D$. Find the support of each item.
(3) Sort the items by their supports in descending order, denote it as sorted-item-list.
(4) Build a transaction FP-tree, $T$, with all the items.
(5) Scan the constraints, $IC$. Find the support of each bin.
(6) Sort the bins by their supports in descending order, denote it as sorted-bin-list.
(7) Create a constraint FP-tree, $T_{IC}$, with only a root node with label being "NULL".
(8) $T_{IC}$-build($T_{IC}$, $IC$, sorted-bin-list).
(9) $result_{11}, result_{12}, \ldots, result_{c_{max}k_{max}} \leftarrow$ DFP-mine($T$, $T_{IC}$, $\emptyset$)
```

Figure 6.3: DFP-tree algorithm for mining N-most interesting itemsets with constraints.

```
Algorithm : $T_{IC}$-tree Building Phase

$T_{IC}$-build($T_{IC}$, $IC$, sorted-list)
Update the FP-tree as follows:
    For each constraint, $IC_c$ in $IC$
        (a) Sort the bins in $IC_c$, according to the order in sorted-list
        (b) Let $[i|I]$ be the sorted item list in $IC_c$, where $i$ is the first element and
            $I$ is the remaining list. Invoke Insert_$T_{IC}$tree($[i|I]$, $T_{IC}$, $c$).

Insert_$T_{IC}$tree($[i|I]$, $T_{IC}$, constraint)
(1) Let $C$ be the child node of $T_{IC}$ such that $C.bin = i.bin$,
    (a) If no such $C$, then create a new node $C$, and let its parent link be linked to $T_{IC}$,
        and its node-link be linked to nodes with the same bin name via the node-link structure;
        $\xi(C) \leftarrow MAXINT$, $\xi_{min}(C) \leftarrow 0$
(2) If $I$ is non-empty,
    then
    (b) $C.c \leftarrow 0$
    (c) invoke Insert_$T_{IC}$tree($I$, $C$, constraint)
    else
    (d) $C.c \leftarrow constraint$
    (e) $\xi(C) \leftarrow 0$
```

Figure 6.4: Construction of constraint FP-tree.

```
Algorithm : DFP-tree Mining Phase
DFP-mine($T$, $T_{IC}$, $\alpha$)
{
  If $T$ contains a single path $P$
  (1) then for each combination, $\beta$, of the nodes in the path $P$ do
     (a) generate itemset $\beta \cup \alpha$ with $support = minimum\ support$ of nodes in $\beta$
     (b) Mapping($T_{IC}$, $\beta \cup \alpha$, $support$)
  (2) else for each $a_i$ in the header of $T$ do
     (c) generate itemset $\beta = a_i \cup \alpha$ with $support = a_i.support$
     (d) $\xi \leftarrow \xi_{min}$(root of $T_{IC}$)
     (e) construct $\beta$'s conditional pattern base using $\xi$ and
           then $\beta$'s conditional FP-tree $T_\beta$
     (f) if $T_\beta \neq \emptyset$ then DFP-mine($T_\beta$, $T_{IC}$, $\beta$)
}
```

Figure 6.5: Algorithm mining N-most interesting itemsets with constraints.

```
Algorithm : Mapping
Mapping($T_{IC}$, $\gamma$, $support$)
{
  Find the path $P$, if any, from $T_{IC}$ that matches the bin pattern of $\gamma$
       Let $P.lowest$ be the last node in $P$
       If $P.lowest.c \neq 0$ and $\xi(P.lowest) \leq support$
            (a) add $\gamma$ to the result set, $result_{ck}$,
               where $c$ is the matched constraint $IC_c = P.lowest.c$, $k$ is the size of $\gamma$
            (b) $\xi(P.lowest) \leftarrow$ the $N_c$-th largest support in $result_{ck}$
            (c) Update_$T_{IC}$($P.lowest$)
}

Update_$T_{IC}$($node_{IC}$)
{
  (1) $\xi_{min}(node_{IC}) \leftarrow$ min( $\xi(node_{IC})$, min( $\xi_{min}$ of all children of $node_{IC}$ ))
  (2) If there is change of value of $\xi_{min}(node_{IC})$
       then invoke Update_$T_{IC}$(parent of $node_{IC}$)
}
```

Figure 6.6: Matching itemset with constraints in constraint FP-tree.

conditional transaction FP-trees which consist of any item in the item header table that belongs to B first. This bottom-up approach allows the values of ξ and ξ_{min} of the leaf nodes to be updated (increased) as soon as possible, and hence we can speed up pushing the nodes in upper levels of the constraint FP-tree. Since the propagation of ξ_{min} will increase the value of ξ, we can achieve better pruning power. In other words, whenever we update ξ_{min} of a node and its ancestors after we visited its child node $Node$, we do not need to consider $\xi_{min}(Node)$ for the update.

From the running example, the order of visiting for the item header table in Figure 6.1 is $f \rightarrow e \rightarrow d \rightarrow a \rightarrow b \rightarrow c$. Therefore, after we visited bin B_4, we can set ξ_{min} of $Node_4$, $Node_7$ and $Node_8$ to $MAXINT$ and invoke function $Update\_T_{IC}$ to push ξ_{min} of $Node_2$, $Node_3$, $Node_6$ and their ancestors if necessary.

Pruning Strategy: When generating the conditional FP-tree of an item a, instead of setting the threshold to be ξ_{min} of the root of T_{IC} as shown in step 2(d) in Figure 6.5, we set ξ to be the minimum value of ξ_{min} among all the constraint FP-tree nodes belonging to the bin of a. This increases the pruning effect during mining.

Lemma 7 An itemset containing an item a with support s is not interesting if s is less than the minimum value of ξ_{min} among all the constraint FP-tree nodes belonging to the bin of a in the closed interpretation.

For example, the threshold for item f's conditional FP-tree is the minimum ξ_{min} among $Node_4$, $Node_7$ and $Node_8$ in Figure 6.2.

Corollary 2 Given an itemset I with support s, let B_{lowest} be the last bin in the sorted bin pattern for I according to the constraint header table, I is not interesting if s is less than the minimum value of ξ_{min} among all the constraint FP-tree nodes belonging to B_{lowest}.

Therefore we can further strengthen the pruning power of ξ in step 2(e) of Figure 6.5 by finding a suitable ξ_{min} from the items in β for building the conditional FP-trees of β.

Open Interpretation

We employ similar strategies for the open interpretation as for the close interpretation. There is some more complication since for each constraint IC_i, we need to consider the k-itemsets for $|IC_i| \leq k \leq k_{max}$. Therefore, at each node $Node$ in the constraint FP-tree, if it corresponds to a constraint IC_i, we keep a list of values ξ_{ik} for $|IC_i| \leq k \leq k_{max}$. Instead of at most one branch of the constraint FP-tree matching an itemset, there may be more than one branch matching an itemset in the open interpretation. As a result, the $Mapping$ function in Figure 6.6 would take more time.

6.3 Maximum Support Thresholds

In this section, we consider the case of pruning interesting itemsets which have supports greater than a user specified **maximum support threshold** ξ_{max}. In other words, we would like to find the N_i-most interesting itemsets for each constraint IC, where the supports of these itemsets are less than or equal to the maximum support threshold ξ_{max}. The rationale of employing ξ_{max} is that itemsets with very high supports can be trivial to the users and thus we do not need to mine these itemsets from the database. Suppose we have a database for maternity medical records at a local hospital, then all patients in the records will be female and almost all will be living in the same country. Hence the support of the itemset containing "female" and the country name will be very high. Such trivial frequent itemsets are typically not interesting. On the other hand, itemsets with smaller supports may be more interesting since users may not have the explicit knowledge about these itemsets.

A simple way to employ the maximum support thresholds with our algorithm is to discard itemsets of supports greater than ξ_{max} instead of inserting them into the resulting sets. The other parts of the algorithm remain unchanged. We will investigate the performance of our proposed algorithm for mining the constraint-based interesting itemsets with maximum support thresholds in Section 6.4.

6.4 Performance Evaluation

We compare the performance of the Double FP-trees and the exhaustive Single FP-tree. All experiments are carried out on a SUN ULTRA 5_10 machine running SunOS 5.6 with 512MB Main Memory. Both synthetic and real datasets are used. Note that in this section, the support values listed are in terms of the percentage of the transactions that contain the corresponding itemsets.

Real Data: We use the real census data provided in 1990 [50]. The dataset has 23 attributes, 63 items and 126229 transactions. To generate support specifications, we grouped the items from the same attribute into a bin, giving 23 bins $B_1, B_2, \ldots, B_{23}$ [57]. Let V_i be a bin variable. We specify constraint IC_i in the closed interpretation:

$$IC_i(V_1, V_2, \ldots, V_k) = N_i, \text{ where } 0 < k \leq k_{max}. \tag{6.3}$$

We specify N_i in two ways: (1) N_i is specified by the user. (2) N_i = number of itemsets matching IC_i with supports $\geq \theta_i(V_1, V_2, \ldots, V_k)$, where $\theta_i(V_1, V_2, \ldots, V_k) = min(\gamma^{k-1} \times S(V_1) \times \ldots \times S(V_k), 1)$, $\gamma > 1$, $S(V_j)$ is the smallest support of the items in the bin represented by V_j.

Synthetic Data: The synthetic datasets are obtained from the generator in [12]. We set the number of transactions = 100K, number of items = 500, average length of transactions = 10, and the default setting for all other parameters. We partitioned the support range into 4 intervals such that B_i contains the items with support in the i-th interval and the number of items in B_i is approximately equal [56]: $| B_1 |= 122$, $| B_2 |= 122$, $| B_3 |= 122$, and $| B_4 |= 124$. Table 6.5 shows the constraints in the closed interpretation, where N_i is determined by Equation (4).

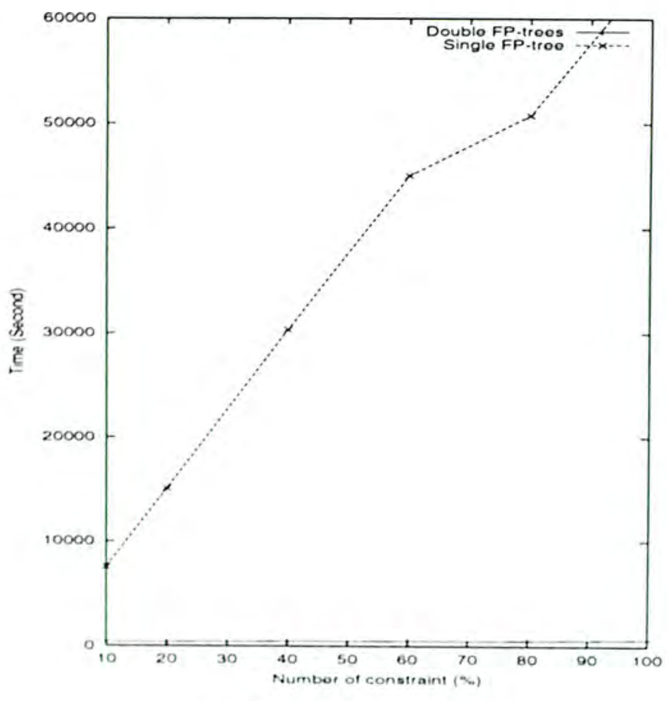
Table 6.5: Constraints for synthetic data in the closed interpretation.

C_i	N_i	C_i	N_i
$IC_1(B_1)$	122	$IC_9(B_2)$	122
$IC_2(B_1, B_2)$	14884	$IC_{10}(B_2, B_3)$	1233
$IC_3(B_1, B_2, B_3)$	453348	$IC_{11}(B_2, B_3, B_4)$	13209
$IC_4(B_1, B_2, B_3, B_4)$	36131	$IC_{12}(B_2, B_4)$	11372
$IC_5(B_1, B_2, B_4)$	120005	$IC_{13}(B_3)$	122
$IC_6(B_1, B_3)$	14884	$IC_{14}(B_3, B_4)$	8243
$IC_7(B_1, B_3, B_4)$	43492	$IC_{15}(B_4)$	124
$IC_8(B_1, B_4)$	13176		

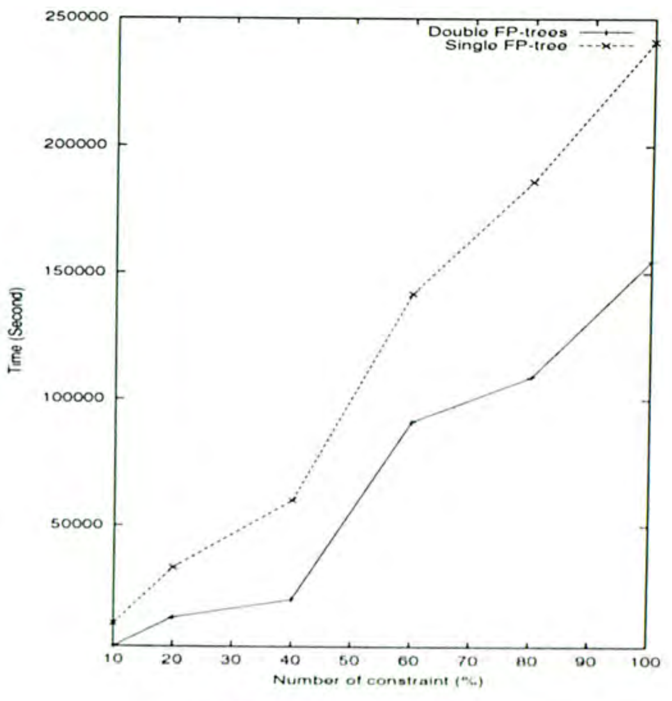
In the experiment, we also test the case for the Double FP-trees when a known (optimal) support threshold, ξ_{opt} is given, i.e. a threshold which is just small enough to make sure that all the required N-most interesting k-itemsets, which match the item constraints, are of supports greater than or equal to ξ_{opt}. We denote this method as **Double FP-trees with ideal threshold**.

Figure 6.7 shows the performance of the two approaches using the real dataset. We vary the constraint set size by randomly choosing 10%, 20%, 40%, ..., 100% constraints out of the original set. We set N using Equation 6.3 with $\gamma = 5$, and $k_{max} = 5$. The Double FP-trees approach outperforms the Single FP-tree approach in several order of magnitudes in the closed interpretation. It also gives an optimistic result in the open interpretation. Table 6.6 gives the result for the synthetic data. Since the optimal threshold is very low, the Double FP-trees with ideal threshold gives the same performance as the Double FP-trees.

As expected, the matching in the closed interpretation is faster than that in the open interpretation. An itemset can at most match one constraint in the closed interpretation while it can match several constraints in the open interpretation. Moreover, we can apply Lemma 7 for the closed interpretation, and we cannot do so for the open interpretation since any item in the base of a conditional FP-tree can both be included and excluded during matching.



(a) Closed interpretation



(b) Open interpretation

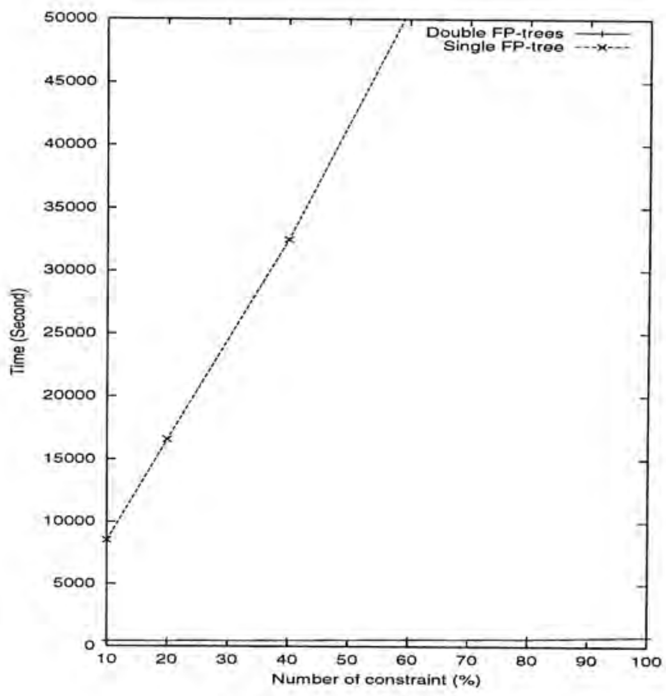
Figure 6.7: Real dataset.

Table 6.6: Synthetic data, closed interpretation.

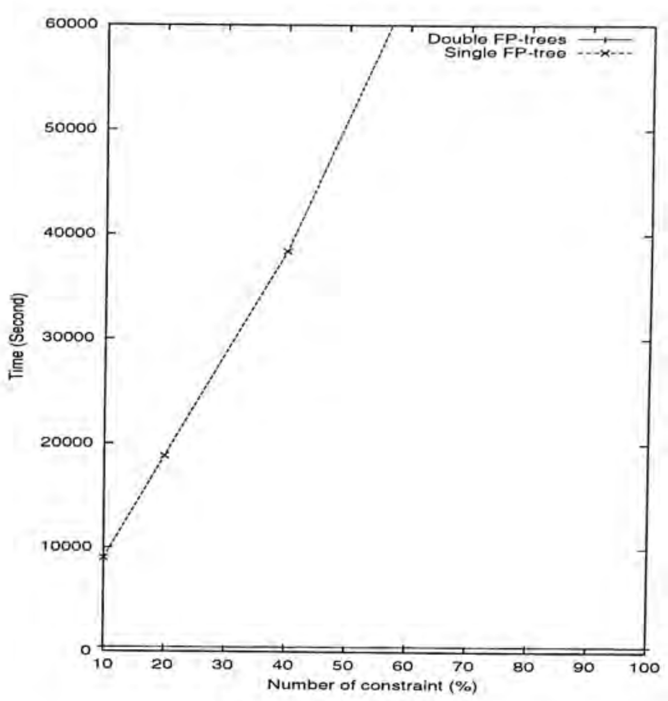
	Time (Second)
Single FP-tree	15961
Double FP-trees	12621

Next we again vary the size of the constraint set but with an uniform N for all the constraints. We set $N = 5$ for k_{max} up to 5. Again, Double FP-trees outperforms Single FP-tree. Both Figures 6.8(a) and 6.8(b) show that the Double FP-trees approach requires only 400 – 700 seconds for the mining while the Single FP-tree approach may require more than 100000 seconds.

From the result, we can see that the use of constraint FP-tree allows an efficient matching between itemsets and constraints. Moreover, we can build



(a) Closed interpretation



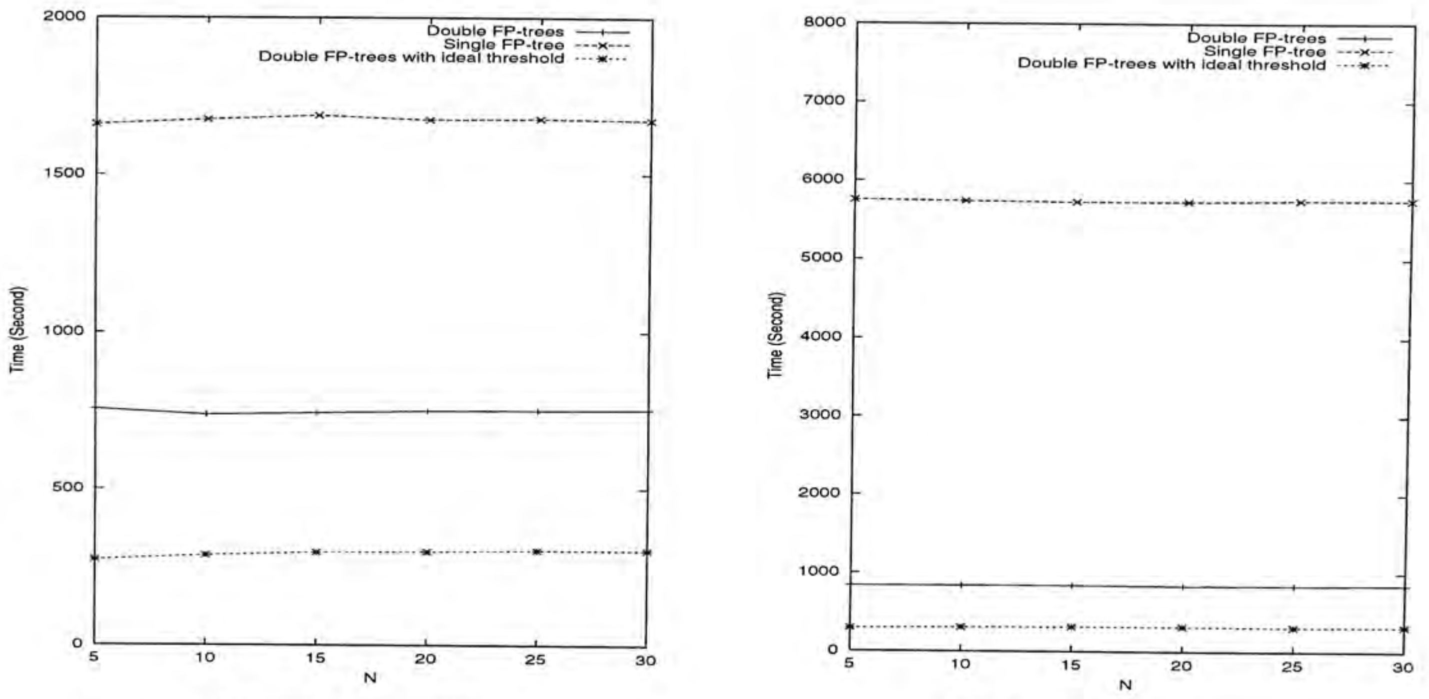
(b) Open interpretation

Figure 6.8: Real dataset, uniform $N = 5$.

less number of conditional FP-trees with smaller sizes by making use of a local and larger ξ_{min}, instead of a global minimum ξ_{min} as used in the Single FP-tree approach, for pruning when building conditional FP-trees. Table 6.7 shows the strength of the scanning order of the Double FP-trees algorithm from the total number of itemsets, which are generated and inserted to the resulting lists. The number of extraneous itemsets mined by the Single FP-tree can be an order of magnitude more than the Double FP-trees.

We also test their performance against different N and k_{max}. Figure 6.9 shows the result for $N = 5, 10, ..., 30$. Since the constraint set is much smaller than that of the real data, the mining time required for different N of the same approach does not vary much. However, using a tree structure for constraint matching benefits our Double FP-trees approach to a certain extent as shown in the figure.

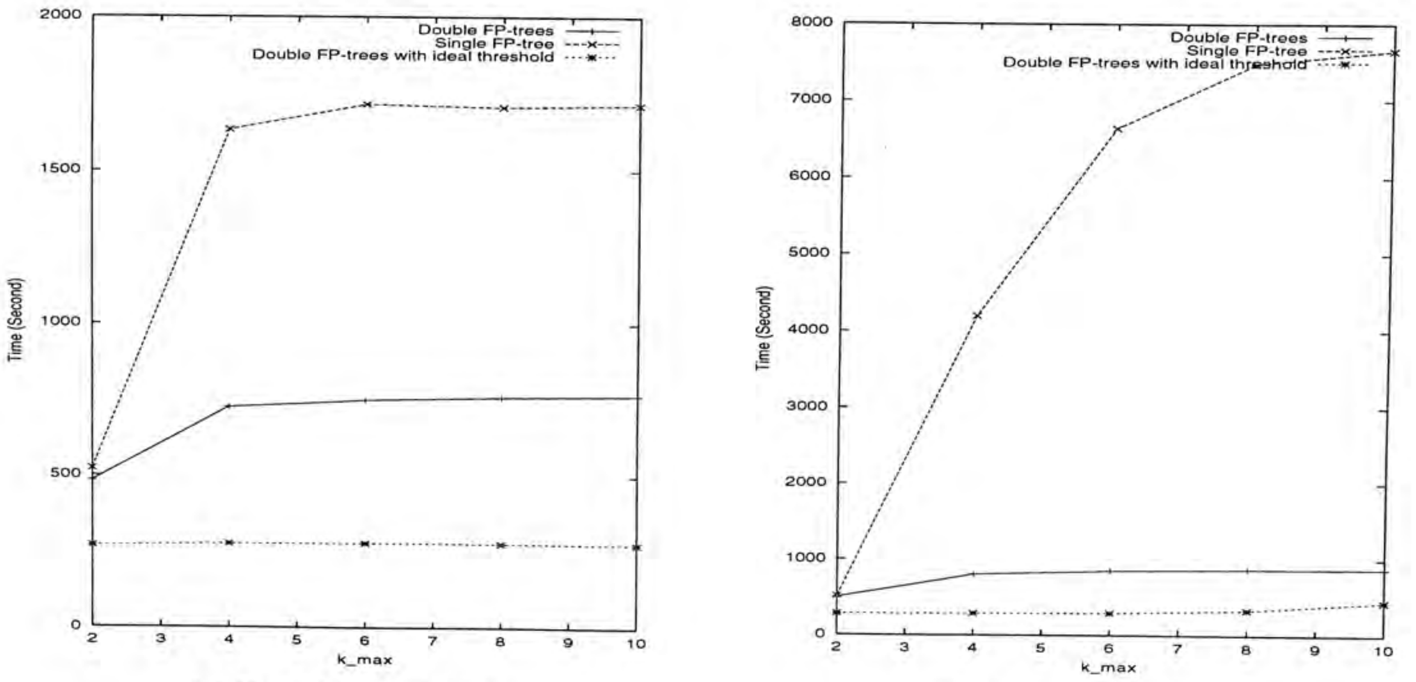
Figure 6.10 shows the result of different k_{max}. We set $k_{max} = 2, 4, ..., 10$.



(a) Closed interpretation

(b) Open interpretation

Figure 6.9: Synthetic dataset, uniform N = 5, 10, ..., 30.



(a) Closed interpretation

(b) Open interpretation

Figure 6.10: Synthetic dataset, $N = 5$, different k_{max}.

Table 6.7: Total number of itemsets mined.

Data	Method	Interpretation	N	k_{max}	Number of itemsets
Real	Double FP-trees	close	5	5	222289
	Single FP-tree	close	5	5	223023
	Double FP-trees	open	5	5	306535
	Single FP-tree	open	5	5	1220388
Synthetic	Double FP-trees	close	5	5	307
	Single FP-tree	close	5	5	1675
	Double FP-trees with ideal threshold	close	5	5	154
	Double FP-trees	close	30	5	2069
	Single FP-tree	close	30	5	4099
	Double FP-trees with ideal threshold	close	30	5	1116
	Double FP-trees	open	5	4	1663
	Single FP-tree	open	5	4	7177
	Double FP-trees with ideal threshold	open	5	4	909

The Double FP-trees method gives a linear performance in both interpretations.

In conclusion, the Double FP-trees algorithm is much more scalable than the exhaustive Single FP-tree method. It is highly effective no matter the number of constraints is large like that in the real dataset or small like that in the synthetic dataset. The algorithm is also comparable to the case when an optimal support threshold is given.

Mining with Maximum Support Thresholds: We vary the maximum support threshold ξ_{max} in the following ways: (1) For real data, $\xi_{max} = 20\%$, 30%, 40%, 50%, or 100%. (2) For synthetic data, $\xi_{max} = 5\%$, 6%, 7%, 8%, 9%, or 100%. The cases where $\xi_{max} = 100\%$ are cases where we do not have any overhead in handling maximum support thresholds, as there is essentially no threshold. From experiment, we find that the performances of the Double FP-trees algorithm with different ξ_{max} are similar. For the real dataset with N_i specified by Equation 6.3, the execution time varies from 590 to 650 seconds. For the synthetic dataset with $N = 5$, the time varies only from 750 to

770 seconds, which is less than 10%. Therefore the overhead in handling the maximum support threshold is very small, and mining interesting itemsets with maximum support thresholds can be considered as an alternative when users, who may have some special knowledge for the data, want to find interesting but non-trivial itemsets.

6.5 Conclusion

In this chapter, we study the problem of mining N-most interesting k-itemsets with a set of user specified item constraints. We employ BOMO for the item constraints problem by using a double FP-trees approach. We also provide the maximum support threshold mining mechanism for eliminating trivial large itemsets. Experiments show that our methods provide good performances.

In the future, we may investigate the mining of N-most interesting association rules in a constraint-based manner. In this case, both high support and high confidence will be required in determining the N best rules to be returned.

Chapter 7

Conclusion

Classification and association rules mining are usually considered independently for different applications. However, they are complements of each other that the former is a supervised learning process while the later is an unsupervised one. Both of them are essential to practical applications. In this thesis, we combined these two important data mining techniques for a practical problem of mining interesting patterns that contain objects belonging to the same class given only a query example.

To handle this problem in an efficient and user-controllable way, we studied several strategies in building decision trees, and investigated the mining of N-most interesting itemsets with constraints.

We proposed a pre-sorting technique for building decision trees in that the splitting criteria for each node can be evaluated during the generation of attribute lists, so as to reduce the amount of I/O operations required for the evaluation to zero while preserving the accuracy. We also considered the difference in bucket size allocation and the methods in splitting the dataset, including one-to-many and many-to-one hashing, horizontal hashing, attribute pairing and database replication. Under the limitation of main memory space compared with the very large databases, we provide absolute improvements on previous methods. We can also meet user with different requirements by the different performance characteristics of our proposed methods.

On the other hand, we investigated the constraint-based mining of interesting itemsets into two steps. We first proposed three algorithms for mining N-most interesting itemsets. We allowed users of any type to control the number of results they are interested by the elimination of specifying a minimum support threshold. With the efficient mining of our methods, we then considered the constraint-based itemsets mining. We defined item constraints which allow users to specify the particular set of items they are interested. This is always the case that particular users would only like to look for certain interesting patterns. We made use of our BOMO algorithm, and proposed the Double FP-trees approaches for mining constraint-based interesting itemsets.

With the effectiveness of the algorithms, we can handle our mining of interesting patterns for similar objects in an more efficient and practical way. In future, it is worth investigating the combination of decision tree classification and association rules mining steps into a single component.

Appendix A

Probabilistic Analysis of Hashing Schemes

Here we give some probabilistic analysis of the many-to-one simple hashing scheme and the horizontal hashing scheme introduced in Section 3.2.2. Assume equal probability of a record appearing in any slot in the attribute list. Assume that there are in total N records. Assume that we have scanned p records in the splitting list (we can neglect the hashing details for this analysis). Let us call this set of records A. The chance that the p records scanned contain a particular record is p/N. In the simple many-to-one hashing scheme, suppose we bring in k records in the non-splitting attribute list (from one single non-splitting attribute) then the probability that all the records are contained in the p records in A is $(p/N)^k$.

Next consider the horizontal hashing scheme. Assume we have n non-splitting attributes. If we use horizontal hashing, we bring in k records from the n non-splitting attribute lists. From each list we take L records where $Ln = k$. Let us call the set of k records B. Let us call the L records from an attribute P the bucket of P. For any non-splitting attribute P, in the L records for P, some of them may be repeated in B. The chance that a record for P appear in the bucket of another non-splitting attribute is L/N. The chance that it does not appear in the other bucket is $(1 - L/N)$. The chance that it does not appear in the buckets of i other non-splitting attribute is $(1 - L/N)^i$. Therefore the expected number of distinct records in the bucket for P is given by $m = L + L(1-L/N) + L(1-L/N)^2 + \ldots + L(1-L/N)^{n-1}$. The probability that all the k records from non-splitting attributes are contained in the p records in B is therefore $(p/N)^m$.

This value is greater than $(p/N)^k$. For example, if $N = 4$, $k = 2$ and $n = 2$, $L = 1$. Suppose $p/N = 0.5$, then for the first scheme the chance that B is contained in A is $0.5^2 = 0.25$. For the second scheme, the expected number of unique records in B is $1 + (1 - 0.25) = 1.75$. Hence the chance that B is contained in A is $0.5^{1.75} = 0.30$. Therefore, it is more likely with the horizontal hashing method than it is with the simple many-to-one method that we need to fetch only p records from the splitting attribute list to resolve the records in B. This may not have any effect if k is large, because $(p/N)^k$ will be very small. Also, nm is still large and $(p/N)^{nm}$ is still very small. However, if k is small, the effect will be noticeable.

Appendix B

Hash Functions

Table address	Table contents	
[0]	empty	address region
[1]	empty	
[2]	empty	
[3]	empty	
[4]	empty	
[5]	empty	cellar
[6]	empty	

(a) Initial hash table with seven buckets

Table address	Table contents
[0]	empty
[1]	empty
[2]	17
[3]	empty
[4]	19
[5]	empty
[6]	epla

(b) After inserting key 17 and 19

Figure B.1: Hash table for coalesced chaining.

Linear probing is one of the open addressing methods and is also the simplest one among all hash methods. Probing is done by

$$probe_i = (rid + i \times c) \ mod \ TableSize$$

where rid is the record ID of an entry to be probed; c is a constant representing the jump size; $TableSize$ is the size of the hash table, i.e. the number of record entry the table can contain; $probe_i$ is the hash table entry to be probed at in the i-th trial. Initially, i is zero, if the record entry fails at the $probe_0$ position

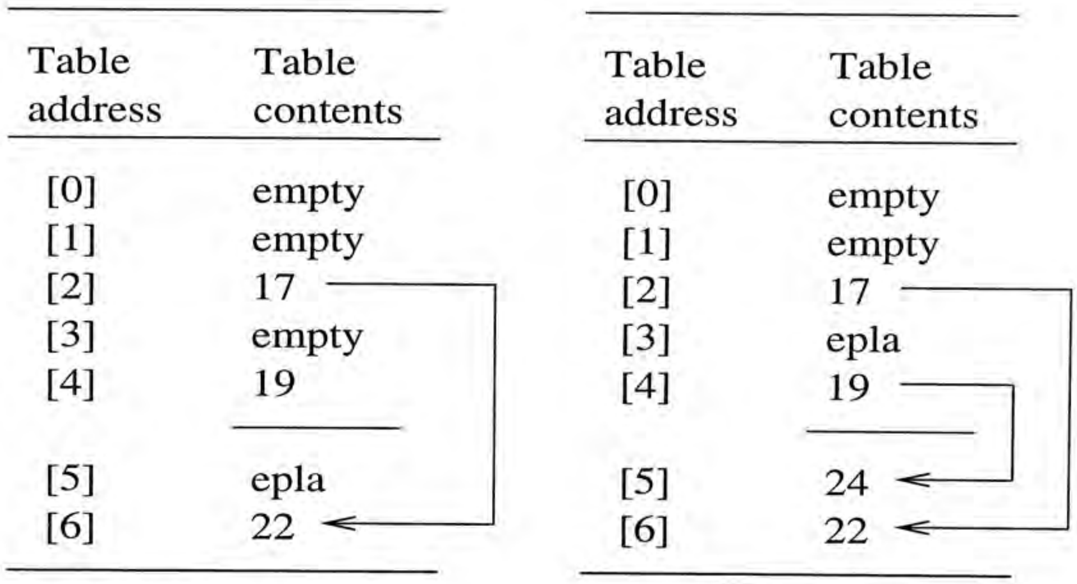


Table address	Table contents
[0]	empty
[1]	empty
[2]	17
[3]	empty
[4]	19
[5]	epla
[6]	22

(a) After inserting key 22

Table address	Table contents
[0]	empty
[1]	empty
[2]	17
[3]	epla
[4]	19
[5]	24
[6]	22

(b) After inserting key 24

Figure B.2: Resulting content of the hash table.

of the hash table, i will be incremented by one and the record will try to probe at the $probe_1$ position. Probing will continue until there is a hit for the record or certain termination condition is reached. The size of the hash table, $TableSize$, is an important parameter. It must be large enough to hold the elements we want to store, i.e. the record entries to be loaded and inserted to the hash table in each distribution cycle. In general, to give a better randomizing effect, $TableSize$ should be a prime number. Another important parameter is the **load factor**, α, which is the fraction or utilization of the table that contains records at any time.

$$\alpha = \alpha(t) = n/TableSize$$

where n is the number of records that is actually stored in the table. This kind of probing is simple and computationally efficient. However, it suffers the clustering problem. The expected collision rate for linear probing is relatively high compared to other hash methods.

Coalesced chaining [53] is shown to require less number of collision. The hash table consists of two parts: the *address region* and the *cellar*. Figure

B.1(a) shows an example of a hash table using coalesced chaining, addresses 0 to 4 make up the address region, and addresses 5 to 6 make up the cellar. We map each record into the address region using a hash function. The resulting address from the hash function is called *home address*. We use the cellar to store records only if collision occurs. In our example, we use the division hash function $H(key) = key \bmod 5$, where *key* is an integer. After inserting key values 17 and 19, we have Figure B.1(b). Then we insert 32. Since it collides with 17, we store 32 to the empty position with the largest address, i.e. the address represented by *epla*. In addition, it is added to a list that begins at its home address (address 2). The result is shown in Figure B.2(a). Next, we add key value 24. Again, collision occurs with key 19. We placed 24 in address 5 (the empty position with the largest address pointed by *epla*) and link it to a list beginning at location 4. The result is shown in Figure B.2(b).

Bibliography

[1] Charu C. Aggarwal and Philip S. Yu. Mining large itemsets for association rules. In *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, pages 23–31, March 1998.

[2] R. Agrawal, S. Ghosh, T. Imielinski, B. Iyer, and A. Swami. An interval classifier for database mining applications. In *Proceedings of VLDB*, pages 560–573, 1992.

[3] R. Agrawal, T. Imielinski, and A. Swami. Database mining: a performance perspective. *IEEE Transactions of Knowledge and Data Engineering*, 5(6):914–925, 1993.

[4] R. Agrawal and J. Shafer. Parallel mining of association rules: Design, implementation, and experience. In *Technical Report RJ10004, IBM Almaden Research Center, San Jose, CA 95120*, Jan. 1996.

[5] Rakesh Agrawal, Johannes Gehrke, Dimitrios Gunopulos, and Prabhakar Raghavan. Automatic subspace clustering of high dimensional data for data mining applications. In *Proceedings of the ACM SIGMOD Conference on Management of Data, Montreal, Canada*, pages 94–105, 1998.

[6] Rakesh Agrawal, Tomasz Imielinski, and Arun Swami. Mining association rules between sets of items in large databases. In *Proceedings of the ACM SIGMOD International Conference on the Management of Data*, pages 207–216, 1993.

[7] Rakesh Agrawal and Ramakrishnan Srikant. Fast algorithms for mining association rules in large databases. In *Proceedings of the 20th International Conference on Very Large Databases (VLDB)*, pages 487–499, 1994.

[8] K. Alsabti, S. Ranka, and V. Singh. Clouds: a decision tree classifier for large datasets. In *Proceedings of KDD*, pages 2–8, 1998.

[9] Michael J. A. Berry and Gordon Linoff. Data mining techniques. New York : Wiley Computer Pub., c2000, 1997.

[10] L. Breiman, J. Friedman, R. Olshen, and C. Stone. *Classification and regression trees.* Wadsworth, Belmont, 1984.

[11] Sergey Brin, Rajeev Motwani, Jeffrey D. Ullman, and Shalom Tsur. Dynamic itemset counting and implication rules for market basket data. In *Proceedings of the ACM SIGMOD International Conference on the Management of Data*, pages 255–264, 1997.

[12] IBM Almaden Research Center. Synthetic Data Generation Code for Associations and Sequential Patterns. http://www.almaden.ibm.com/cs/quest.

[13] P. Cheeseman, J. Kelly, and M. Self. Autoclass: a bayesian classification system. In *Proceedings of 5th Int. Conf. on Machine Learning*, pages 54–64. Morgan Kaufman, June 1988.

[14] Chun-Hung Cheng, A. Fu, and Zhang Yi. Entropy-based subspace clustering for numerical data. In *Proceedings of the ACM SIGKDD Fifth International Conference on Knowledge Discovery and Data Mining (SIGKDD)*, pages 84–93, August, 1999.

[15] U. Fayyad and K. Irani. Multi-interval discretization of continuous-valued attributes for classification learning. In *Proceedings of the 13th Int. Joint Conf. on Artificial Intelligence*, pages 1022–1027, 1993.

[16] Ada Wai-chee Fu, Renfrew Wang-wai Kwong, and Jian Tang. Mining N-most Interesting Itemsets. In *Proceedings of the International Symposium on Methodologies for Intelligent Systems (ISMIS)*, pages 59–67, 2000.

[17] J. Gehrke, V. Ganti, R. Ramakrishnan, and Wei-Yin Loh. Boat–optimistic decision tree construction. In *Proceedings of ACM SIGMOD*, pages 169–180, 1999.

[18] J. Gehrke, R. Ramakrishnan, and V. Ganti. Rainforest - a framework for fast decision tree construction. In *Proceedings of VLDB*, pages 416–427, 1998.

[19] D. Godberg. *Genetic algorithms in search, optimization and machine learning.* Morgan Kaufmann, 1989.

[20] G. Grahne, L. Lakshmanan, and X. Wang. Efficient mining of constrained correlated sets. In *Proceedings of the International Conference on Data Engineering (ICDE)*, pages 512–521, 2000.

[21] Jiawei Han and Yongjian Fu. Discovery of multiple-level association rules from large databases. In *Proceedings of the 21st International Conference on Very Large Databases (VLDB)*, pages 420–431, 1995.

[22] Jiawei Han, L. V. S. Lakshmanan, and R.T. Ng. Constraint-based, multidimensional data mining. In *COMPUTER (special issues on Data Mining), 32(8)*, pages 46–50, 1999.

[23] Jiawei Han, J. Pei, and Yiwen Yin. Mining Frequent Patterns without Candidate Generation. In *Proceedings of the ACM SIGMOD International Conference on the Management of Data*, pages 1–12, 2000.

[24] M. Houtsma and Arun Swami. Set-oriented mining of association rules. In *Technical Report RJ 9567, IBM Research Report*, Oct. 1993.

[25] Laurent Hyafil and Ronald L. Rivest. Constructing optimal binary decision trees is np-complete. In *Information Processing Letters, 5(1)*, pages 15–17, 1976.

[26] Minnesota Population Center in University of Minnesota. IPUMS-98. http://www.ipums.umn.edu/usa/samples.html.

[27] M. Kamber, J. Han, and J. Y. Chiang. Mining partial periodicity using frequent pattern trees. In *Proceedings of Knowledge Discovery and Data Mining (KDD)*, pages 207–210, 1997.

[28] M. Kamber, Jiawei Han, and Jenny Y. Chiang. Metarule-guided mining of multi-dimensional association rules using data cubes. In *Proceedings of the 3rd International Conference on Knowledge Discovery and Data Mining*, pages 207–210, 1997.

[29] M. Klemettinen, H. Mannila, P. Ronkainen, Toivonen H., and A.I. Verkamo. Finding interesting rules from large sets of discovered associated rules. In *Proceedings of the International Conference on Information and Knowledge Management (CIKM)*, pages 401–408, 1994.

[30] B. Lent, A. Swami, and J. Widom. Clustering association rules. In *Proceedings of the International Conference on Data Engineering (ICDE)*, pages 220–231, 1997.

[31] Hongjun Lu, Ling Feng, and Jiawei Han. Beyond intra-transaction association analysis: Mining multi-deimensional inter-transaction association rules. In *ACM Transactions on Information Systems, 18(4)*, pages 423–454, 2000.

[32] M. Mehta, R. Agrawal, and J. Rissanen. Sliq: a fast scalable classifier for data mining. In *Proceedings of fifth Int. Conf. on EDBT*, pages 18–32, March 1996.

[33] M. Mehta, J. Rissanen, and R. Agrawal. Mdl - based decision tree pruning. In *Proceedings of KDD*, pages 216–221, 1995.

[34] D. Michie, D. J. Spiegelhalter, and C. C. Taylor. Machine learning, neural and statistical classification. Ellis Horwood, 1994.

[35] Y. Morimoto, T. Fukuda, Matsuzawa H., Tokuyama T., and Yoda K. Algorithms for mining association rules for binary segmentations of huge categorical databases. In *Proceedings of the VLDB*, pages 380–391, 1998.

[36] A. Mueller. Fast sequential and parallel methods for association rule mining: A comparison. In *Technical Report CS-TR-3515, Department of Computer Science, University of Maryland, College Park, MD*, 1995.

[37] Raymond T. Ng, Laks V. S. Lakshmanan, and Jiawei Han. Exploratory mining and pruning optimizations of constrained associations rules. In *Proceedings of the ACM SIGMOD International Conference on the Management of Data*, pages 13–24, 1998.

[38] J.S. Park, M.S. Chen, and P.S. Yu. An effective hash-based algorithm for mining association rules. In *Proceedings of the ACM SIGMOD International Conference on the Management of Data*, pages 175–186, 1995.

[39] J. Pei and Jiawei Han. Can we push more constraints into frequent pattern mining. In *Proceedings of the International Conference on Knowledge Discovery and Data Mining (KDD)*, pages 1–5, 2000.

[40] J. Pei, Jiawei Han, and L. V. S. Lakshmanan. Mining frequent itemsets with convertible constraints. In *Proceedings of the International Conference on Data Engineering (ICDE)*, pages 433–442, 2001.

[41] J. Quinlan. Discovering rules by induction from large collections of examples. In *Expert Systems in the micro Electronic Age*, pages 168–201, 1979.

[42] J. Quinlan. Induction of decision trees. In *Machine Learning*, pages 81–106, 1986.

[43] J. Quinlan. *C4.5: programs for machine learning.* Morgan Kaufmann, 1993.

[44] J. Quinlan and R. Rivest. Inferring decision trees using minimum description length principle. In *Information and Computation*, volume 80, pages 227–248, 1989.

[45] R. Rastogi and K. Shim. Public: A decision tree classifier that integrates pruning and building. In *Proceedings of VLDB*, pages 404–415, 1998.

[46] B. Ripley. *Pattern recognition and neural networks.* Cambridge university Press, Cambridge, 1996.

[47] A. Savasere, E. Omiecinski, and S. Navathe. An efficient algorithm for mining association rules in large database. In *Proceedings of the 21st International Conference on Very Large Databases (VLDB)*, pages 432–443, 1995.

[48] J. Shafer, R. Agrawal, and M. Mehta. Sprint: a scalable parallel classifier for data mining. In *Proceedings of the 22nd VLDB*, pages 544–555, 1996.

[49] Pradeep Shenoy, Jayant R. Haritsa, S. Sudarshan, Gaurav Bhalotia, Mayank Bawa, and Devavrat Shah. Turbo-charging Vertical Mining of Large Databases. In *Proceedings of the ACM SIGMOD International Conference on the Management of Data*, pages 22–33, 2000.

[50] C. Silverstein, S. Brin, R. Motwani, and J Ullman. Scalable techniques for mining causal structures. In *Proceedings of the 24th International Conference on Very Large Databases (VLDB)*, pages 594–605, 1998.

[51] R. Srikant, Q. Vu, and R. Agrawal. Mining association rules with item constraints. In *Proceedings of Knowledge Discovery and Data Mining (KDD)*, pages 67–73, 1997.

[52] Ramakrishnan Srikant and R. Agrawal. Mining Quantitative Association Rules in Large Relational Tables. In *Proceedings of the ACM SIGMOD International Conference on the Management of Data*, pages 1–12, 1996.

[53] F. Daniel Stubbs and W. Neil Webre. In *Data Structures with Abstract Data Types and Modula-2*, pages 340–363. Brooks/Cole Publishing Company, 1987.

[54] H. Toivonen. Sampling large databases for association rules. In *Proceedings of VLDB*, pages 134–145, 1996.

[55] H. Wang and C. Zaniolo. Cmp: a fast decision tree classifier using multivariate predictions. In *Proceedings of the 16th ICDE*, pages 449–460, 2000.

[56] K. Wang, Y. He, and J. Han. Pushing support constraints into frequent itemset mining. In *School of Computing, National University of Singapore*, 2000.

[57] Ke Wang, Yu He, and Jiawei Han. Mining frequent itemsets using support constraints. In *Proceedings of the 26th VLDB Conference*, pages 43–52, 2000.

[58] S. Weiss and C. Kulikowski. *Computer systems that learn: Classification and prediction methods from statistics, neural nets, machine learning and expert systems.* Morgan Kaufmann, 1991.

[59] M. J. Zaki, M. Ogihara, S. Parthasarathy, and W. Li. Parallel data mining for association rules on shared-memory multi-processors. In *Supercomputing' 96, Pittsburg, PA*, pages 17–22, 1996.

[60] M. J. Zaki, S. Parthasarathy, and W. Li. A localized algorithm for parallel association mining. In *9th Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, pages 321–330, 1997.

[61] Mohammed Javeed Zaki and Mitsunori Ogihara. Theoretical foundations of association rules. In *Proceedings of 3rd SIGMOD'98 Workshop on Research Issues in Data Mining and Knowledge Discovery (DMKD'98)*, pages 7:1–7:8, 1998.