Strategies for Partitioning Data in Association Rule Mining

Thesis submitted in accordance with the requirements of the

University of Liverpool for the degree of Doctor of Philosophy

in

Computer Science

by

Shakil Ahmed

August 2004

Strategies for Partitioning Data in Association Rule Mining

by

Shakil Ahmed

© Copyright 2004

Acknowledgements

I am very grateful to my supervisors, Professor Paul Leng and Dr Frans Coenen, for their invaluable assistance, encouragement, advice and guidance throughout my research and in the preparation of this thesis.

I am grateful to the head of the department of Computer Science, Professor Michael Wooldridge, for the financial support without which this research would surely have been extremely difficult.

I am also appreciative to other members of the department. Thanks to Kenneth Chan, Patrick Colleran, David Nixon, Dave Shield and Andrew Craig for their essential technical assistance.

I will always remember my friends Marie Devlin, Carmen Pardavila, Christian Setzkorn, David Kennedy, Chakkrit Snae, David Huang, Ian Blacoe, Aiman Badri, Katie Atkinson, Alison Chorley, Qin Xin, Justin Wang and others for their generous support and making my life in Liverpool memorable.

Finally, I owe a lot to my wife, who I wish to thank for her patience, tolerance, support, and constant encouragement during the hardest period of my life. I am grateful to my parents for always encouraging me to do my best. All thanks and praise to almighty God for blessing me to accomplish the research.

Abstract

The problem of extracting association rules from databases is well known. The most demanding part of the problem is the determination of the *support* for all those sets of items which occur often enough to be of possible interest. All methods of association rule mining require the *frequent sets* of items to be first computed. The cost of this increases in proportion to the database size, and also with its density. Densely-populated databases can give rise to very large numbers of candidates that must be counted. Both these factors cause performance problems, especially when the data structures involved become too large for primary memory. The research described within this thesis investigated strategies for partitioning the data in these cases. New methods are presented that partition data using a tree structure within which candidates are enumerated. It is shown that the new methods scale well for increasing dimensions of data, and perform significantly better than alternatives, especially when dealing with dense data and low support thresholds.

Contents

Acknowledgements ... iii

Abstract ... iv

List of Figures ... ix

1 Introduction ... 1

1.1 Introduction ... 1

1.2 Problem Statements ... 3

1.3 Thesis Contributions ... 5

1.4 Thesis Outline ... 7

2 Background on Knowledge Discovery in Databases ... 9

2.1 Introduction ... 9

2.2 The KDD Process ... 9

2.2.1 Problem Specification ... 10

2.2.2 Resourcing ... 10

2.2.3 Data Cleansing ... 10

2.2.4 Pre-processing ... 11

2.2.5 Data Mining ... 11

2.2.6 Evaluation of Results ... 11

2.2.7 Interpretation of Results ... 12

2.3 Data Mining ... 12

2.3.1 Classification ... 13

2.3.2 Clustering....13

2.3.3 Association Rule Mining....14

2.4 Association Rule Mining Problem....14

2.5 Sequential Algorithms for Finding Frequent Sets....18

2.5.1 AIS....19

2.5.2 Apriori....21

2.5.3 Apriori-TID and –Hybrid....24

2.5.4 Partitioning....25

2.5.5 Sampling....27

2.5.6 Dynamic Itemset Counting....30

2.5.7 Methods that find Maximal Frequent Sets....31

2.5.8 Depth-Project....34

2.5.9 FP-Growth....35

2.6 Parallel Algorithms....38

2.6.1 Data Parallelism....39

2.6.2 Task Parallelism....40

2.7 Summary....42

3 Review of Computing Support via Partial Totals....43

3.1 Introduction....43

3.2 Partial Support....43

3.3 Computing Total Supports....51

3.4 Summary....58

4 **Strategies for Partitioning Data** 60

4.1 Introduction 60

4.2 Data Partitioning (DP) 61

4.3 Negative Border (NB) 67

4.4 Tree Partitioning (TP) 73

4.5 Data and Tree Partitioning (DTP) 84

4.6 Summary 87

5 **Experiments and Results** 89

5.1 Introduction 89

5.2 Size of Database 90

5.3 Degree of Segmentation 95

5.4 Number of Attributes 99

5.5 Degree of Partitioning 103

5.6 Density of Database 106

5.7 Performance Comparison 110

5.8 Summary 113

6 **Distributed Association Rule Mining** 116

6.1 Introduction 116

6.2 The Apriori-T algorithm 117

6.3 Architecture and network configuration 118

6.4 Data Distribution 119

6.5 Task Distribution 120

6.6 Tree Distribution....121

6.7 Evaluation....124

6.7.1 Number of *T*-tree Messages....125

6.7.2 Amount of Data Sent and Received....125

6.7.3 Number of Updates....126

6.7.4 Execution Time....127

6.8 Summary....127

7 Overall Conclusion....129

7.1 Introduction....129

7.2 P-tree and T-tree data structures....129

7.3 Strategies for Partitioning Data....130

7.4 Distributed ARM....132

7.5 Summary....133

References....136

Appendix A....151

Appendix B....164

Appendix C....166

List of Figures

Figure 1.1: Example of a P-tree....6

Figure 2.1: Lattice of subsets of {A, B, C, D}....19

Figure 2.2: Example of a negative border....28

Figure 2.3. FP-tree....37

Figure 3.1: Relationship between subsets of {A, B, C, D}....44

Figure 3.2: Complete P-tree for all the subsets of {A,B,C,D}....45

Figure 3.3: A sample dataset....49

Figure 3.4: P-tree generation example....50

Figure 3.5: P-tree in its final form....51

Figure 3.6: Complete T-tree for all the subsets of {A,B,C,D}....53

Figure 3.7: t-tree generation example....56

Figure 3.8: T-tree in its final form....57

Figure 4.1: Original and Horizontally Segmented dataset....61

Figure 4.2: P-trees form segmented dataset....64

Figure 4.3: Building T-tree level 1....65

Figure 4.4: Building T-tree level 2....66

Figure 4.5: Building T-tree level 3....67

Figure 4.6: Building T-tree with Negative Border....72

Figure 4.7: Update the initial T-tree with P-tree2....73

Figure 4.8: Update the initial T-tree with P-tree3......73

Figure 4.9: Example of a P-tree......74

Figure 4.10: Partition-P-trees from figure 4.9......76

Figure 4.11: PP-trees after reordering of attributes......77

Figure 4.12: Original and Vertically Partitioned Dataset......80

Figure 4.13: PP-trees from vertically partitioned dataset......80

Figure 4.14: Building PT-tree2 from PP-tree2......81

Figure 4.15: Building PT-tree3 from PP-tree3......82

Figure 4.16: Final PT-trees......83

Figure 4.17: Original and HSVP dataset......85

Figure 4.18: PP-trees from HSVP dataset......86

Figure 4.19: PT-trees from PP-trees......86

Figure 5.1: Time to construct P/PP-trees for increasing size of databases......91

Figure 5.2: Memory requirements for largest P-tree/ Partition (T10.I5.N500)......91

Figure 5.3: Time to find frequent sets for increasing size of databases (0.01% support)......92

Figure 5.4: Memory requirements for T-tree/ largest PT-tree (T10.I5.N500)......93

Figure 5.5: Execution times for T10.I5.N500 (0.01% support)......93

Figure 5.6: Memory requirements for T10.I5.N500......94

Figure 5.7: Time to construct P/PP-trees for increasing number of segments......96

Figure 5.8: Memory requirements of largest P-tree/ Partition (T10.I5.N500.D50000)......96

Figure 5.9: Time to find frequent sets for increasing number of segments (0.01% support)....97

Figure 5.10: Memory requirements for T-tree/ largest PT-tree (T10.I5.N500.D50000)....98

Figure 5.11: Effect of increasing segmentation (0.01% support)....98

Figure 5.12: Effect on memory for increasing number of segments....99

Figure 5.13: Time to construct P/PP-trees for increasing number of attributes....100

Figure 5.14: Memory requirements for largest P-tree/ Partition (T10.I5.D50000)...100

Figure 5.15: Time to find frequent sets for increasing number of attributes (0.01% support)....101

Figure 5.16: Memory requirements for T-tree/ largest PT-tree (T10.I5.D50000)....101

Figure 5.17: Execution times for T10.I5.D50000 (0.01% support)....102

Figure 5.18: Memory requirements for T10.I5.D50000....102

Figure 5.19: Time to construct PP-trees for increasing items/partition....103

Figure 5.20: Memory requirements for largest partition (T10.I5.N500.D50000)....104

Figure 5.21: Times to find frequent sets for increasing items/partition (0.01% support)....104

Figure 5.22: Memory requirements for largest partition (T10.I5.N500.D50000)....105

Figure 5.23: Execution times for T10.I5.N500.D50000 (0.01% support)....105

Figure 5.24: Memory requirements for T10.I5.N500.D50000....106

Figure 5.25: Time to construct P/PP-trees for increasing density....107

Figure 5.26: Memory requirements for largest P-tree/ Partition (N500.D50000)....107

Figure 5.27: Time to find frequent sets for increasing density (0.1% support)....108

Figure 5.28: Memory requirements for T-tree/ largest PT-tree (N500.D50000)......108

Figure 5.29: Execution times for N500.D50000 (0.1% support)..........................109

Figure 5.30: Memory requirements for N500.D50000......................................110

Figure 5.31: Time to find frequent sets for decreasing support thresholds............111

Figure 5.32: Memory requirements for largest T-tree/Partition (T10.I5.N500.D50000)..111

Figure 5.33: Performance of the methods for decreasing support thresholds........112

Figure 5.34: Memory requirements for T10.I5.N500.D50000.............................113

Figure 6.1. Distributed T-tree representing the vertical partitioning presented in the example..123

Figure 6.2. Average Total size (Kbytes) of messages sent and read/taken per process..126

Figure 6.3. Average number of updates (x 10^6) to generate a final T-tree per process..126

Figure 6.4. Average execution time (seconds) per process..............................127

Chapter 1

Introduction

1.1 Introduction

The information revolution is generating huge volumes of data in diverse areas including sales, financial transactions, services, and scientific and engineering activities. For example the earth observation satellites of the National Aeronautics and Space Administration (NASA) generate a terabyte (10^{15} bytes) of remote sensing data every day [Frawely *et al.*, 1992] [Zomaya *et al.*, 1999] [Bramer, 1999]. Very large databases may now contain billions of records and thousands of features [Fayyad *et al.*, 1996b].

However, the very large databases themselves in raw form have limited value. What is of value is the knowledge that can be gathered from the data and put to use. But, as the technology for storing and retrieving large volumes of data has developed, and increasingly larger databases are built, the technology for analysing the data has not kept pace [Agrawal *et al.*, 1993b]. Most traditional data analysis methods are designed to analyse smaller data sets and often do not scale up well to analyse these larger database [Fayyad *et al.*, 1996c]. There is an urgent need for a new generation of computational techniques and tools to assist humans in extracting useful information (knowledge) from the rapidly growing volumes of data [Fayyad *et al.*, 1996a]. Collectively techniques that intelligently and automatically assist in transforming these large volumes of data into useful knowledge are part of the process of *Knowledge Discovery in Databases* (*KDD*).

KDD can be defined as *"the automatic and non-trivial process of identifying valid, novel, potentially useful and ultimately understandable patterns in data"* [Fayyad *et al.*, 1996a] [Fayyad *et al.*, 1996b]. It is the nontrivial extraction of implicit, previously unknown and

potentially useful information from data [Frawely *et al.*, 1992]. By knowledge discovery in databases, "interesting" knowledge, regularities, or high-level information can be extracted from the relevant sets of data in databases and be investigated from different angles, and large databases thereby serve as rich and reliable sources for knowledge generation and verification [Chen *et al.*, 1996]. Here "interesting" means an overall measure of pattern value, combining validity, novelty, usefulness and simplicity [Fayyad *et al.*, 1996a]. The key challenge in KDD is the extraction of interesting knowledge and insight from massive databases.

Data mining, viewed as a major step in the KDD process, is the discovery of valuable information from large data volumes using computationally "efficient" techniques [Fayyad *et al.*, 1996d] [Chen *et al.*, 1996] [Zomaya *et al.*, 1999]. It consists of applying data analysis and discovery algorithms that produce a particular enumeration of patterns over the data [Fayyad *et al.*, 1996a]. The choice of data mining algorithm will depend on the type of data that is being analysed and the nature of the data-mining task. Almost always, what is being sought is some relationship which can be observed between categories of information in the data. A particular way to describe such a relationship is in the form of an *association rule* which relates attributes within the database.

Association rules are statements of the form "*90 percent of customers who purchase bread and butter also purchase milk*". These are probabilistic relationships, of the form $A \Rightarrow B$, between disjoint sets of database attributes, which is obtained from examination of records in the database. In the simplest case, the attributes of a database are Boolean, and the database takes the form of a set of records (transactions) each of which reports the presence or absence of each of the attributes (items) in the record. A rule is usually only of potential interest if the items it associates (the member of the itemset $A \cup B$) occur together relatively frequently in the data being examined. These *frequent sets* are defined to be those which exceed some defined threshold level of *support*. The support of an itemset is the number (or proportion) of database records in which that itemset occurs as a subset.

Association Rule Mining (ARM) obtains, from a binary valued dataset, a set of rules which indicate that the *consequent* of a rule is likely (with a certain degree of probability) to apply if the *antecedent* applies [Agrawal *et al.*, 1993b]. Mining association rules is one of the core data mining tasks [Park *et al.*, 1995]] [Agrawal *et al.*, 1996]. The challenge is to identify a set of relations in a binary valued attribute set which describe the likely coexistence of groups of attributes. The ARM problem can be decomposed into two sub-problems: (1) to find the set of all subsets of attributes that frequently occur in database records, and additionally, (2) to extract rules on how a subset of attributes influences the presence of another subset [Agrawal *et al.*, 1993a][Zaki, 1999]. Having obtained the frequent sets of attributes that occur together sufficiently often, the solution of extracting rules that are likely to be of interest is rather straightforward [Agrawal *et al.*, 1993a]. Identifying the frequent sets is thus the most computationally demanding aspect of ARM.

1.2 Problem Statements

The biggest problem in deriving association rules is the task of computing support counts for all possible combinations of items into itemsets. The problem arises because the number of possible itemsets is exponential in the number of possible attribute-values of the binary dataset. For most real data, the number *n* of such items is likely to be such that counting the support of all 2^n sets of items (*itemsets*) is infeasible. Most ARM algorithms, including Apriori [Agrawal and Srikant, 1994], thus make use of the “downward closure property of the set of all itemsets” to find the frequent sets. The observation is that if an itemset is adequately supported, then all the subsets of the itemset will also be adequately supported. Consequently if an itemset is not supported, any effort to calculate the support for its supersets will be wasted. These algorithms proceed in general by attempting to count the support only for those *candidate* itemsets which the algorithm identifies as possible frequent sets, rather than all combinations of items. The performance of these methods, clearly,

depends both on the size of the original database, typically millions or billions of records, and on the number of candidate itemsets being considered.

Of course, it cannot be known in advance whether a candidate itemset will be frequent, and it will therefore be necessary to consider many itemsets that are not in fact frequent. The number of possible candidates increases with increasing density (greater number of items present in a record) of data and with decreasing support thresholds. The task, therefore, increases in difficulty with the scale of the data including density and also with low support threshold. The greatest challenge is posed by data that is too large to be contained in primary memory, especially when high data density and/or a low support threshold give rise to very large numbers of candidates that must be counted. In applications such as medical epidemiology, we may be searching for rules that associate rather rare items within quite densely-populated data, and in these cases the low support-thresholds required may lead to very large candidate sets. These factors motivate a continuing search for efficient algorithms.

Performance will be affected, especially, if the magnitudes involved make it impossible for the algorithm to proceed entirely within primary memory. In this case, assuming that parallelisation or distributed data mining is not an option, some strategy for *partitioning* the data will be required to enable algorithmic stages to be carried out on primary-memory-resident data. In demanding cases local candidate sets of partitioned data will also be required to store in secondary memory. Effective partitioning and storing will be required to reduce the number of accesses to secondary memory. Not all methods can be readily adapted to deal with non-store-resident data, and performance in these cases may not scale linearly with the database size and density.

The final concern, when implementing ARM algorithms using partitioning, is the nature of data structures used to store itemsets as the algorithm progresses; this must be concise and suitable for non-store-resident data while at the same time offering fast access times. Whatever hardware is being used, there comes a point where the data to be mined can

no longer be stored in primary memory. In this event, part of the data/structure must be "dumped" to secondary storage in applications involving a single-processor implementation or must be "distributed" over a given number of processors in applications involving multiple-processor implementation. Not all data structures are suitable or readily adaptable for this non-store-resident data. The challenge is to develop methods that will find all the frequent sets using suitable data structures and effective partitioning in cases where the data is much too large to be contained in primary memory.

1.3 Thesis Contributions

In this thesis various methods of partitioning are examined to limit the total primary memory requirement in Association Rule Mining, including that required both for the source data and for the candidate sets. Both 'horizontal' partitioning, which divides the source data into sets of records, and 'vertical' partitioning, which partitions records into sets of items are considered. The goal is to research and evaluate methods that will find all the frequent sets efficiently in cases where the data is much too large to be contained in primary memory and demonstrate solutions that will scale effectively as the database size and density increases.

The most well known algorithm, Apriori [Agrawal and Srikant, 1994], originally used a Hash Tree to store sets of items. But, of course, Apriori can be implemented using other data structures. More recently researchers have focused on the use of Set Enumeration Trees [Rymon, 1992] for association rule mining. The set enumeration tree imposes an ordering on the set of items and then enumerates the sets according to this ordering. Not all methods and data structures can be readily adapted to deal with partitioning source data and candidate sets when the data is much too large to be contained in primary memory. For example, the otherwise very efficient *DepthProject* [Agarwal *et al.*, 2000] algorithm is explicitly targeted at memory-resident data, while the multiply-linked structure of the *FP*-tree [Han *et al.*, 2000]

causes problems for non-store-resident implementation. The *CATS* tree [Cheung and Zaiane, 2003], an extension of the *FP*-tree, also assumes no limitation on main memory capacity.

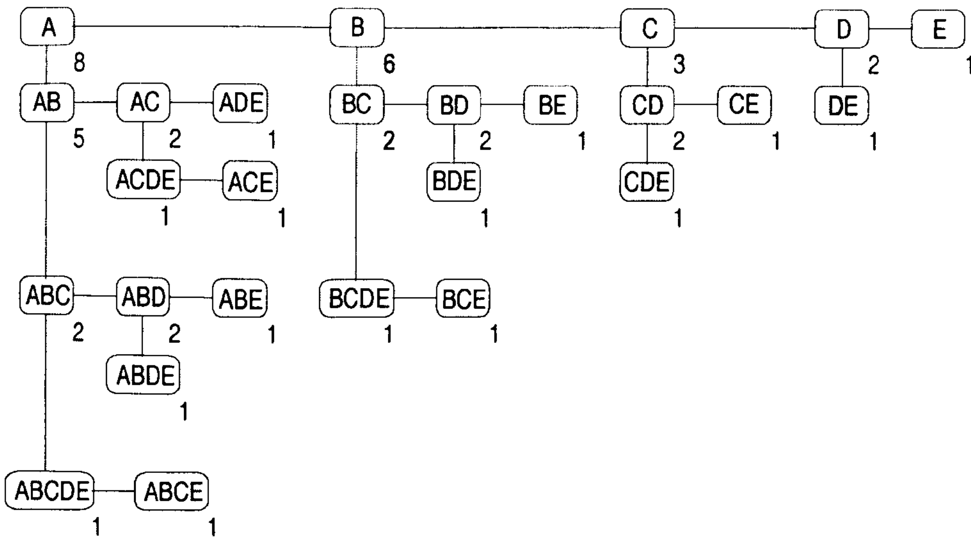


Figure 1.1: Example of a P-tree

Two forms of set enumeration tree have been developed by Coenen and Leng and used in the Apriori-TFP algorithm [Coenen *et al.*, 2001]: (1) the *P*-tree that enumerates sets in a partial ordering, in which sibling nodes are placed in lexicographic order and subtrees are lexicographically-following supersets of their root node, (2) the *T*-tree has a similar form to the *P*-tree, differing only in that subtrees are in this case supersets that lexicographically precede their root node. After its construction, the *P*-tree [Goulbourne *et al.*, 2000] can be easily converted into a pointer-free form. In this form it is readily adaptable to implementations in which the data is divided into store-resident concise partitions that are then stored in secondary memory. Figure 1.1 illustrates a *P*-tree for the attribute-set {A,B,C,D,E}. The *T*-tree, in turn, is a very versatile structure which can be used effectively in conjunction with many established methods to store candidate sets. These structures, which are central to the methods discussed in this thesis, will be described in detail in Chapter 3. The Apriori-TFP algorithm, essentially a form of Apriori, demonstrates significant performance gains (reported in [Coenen *et al.*, 2001]) in comparison with Apriori, and also some improvements over the

FP-growth [Han *et al.*, 2000] algorithm, which uses somewhat similar structures and has some similar properties. The Apriori-TFP method begins by performing a single pass of the database to perform a partial summation of the support totals. These partial counts are stored in a *P*-tree which contains all the sets of items present as distinct records in the database, plus some additional sets that are lexicographically preceding subsets of these. This tree structure is then used in the Apriori-TFP algorithm to complete the summation of the final support counts, storing the results in the *T*-tree. On completion of the algorithm, the *T*-tree will contain all frequent sets with their complete support-counts.

In this thesis implementations of Apriori-TFP are considered in cases when it will be impossible to contain all the data required in main memory thus requiring some strategy for partitioning the data. From these approaches, different methods of partitioning are examined to limit the total primary memory requirement, including that required both for the source data and for the candidate sets, for carrying out the counting of support totals needed for applications involving a single processor system. For applications involving multiple processors, different methods are examined where the input data is distributed among processors. Our hypothesis is that the *P*-tree and *T*-tree structures form a basis for an effective partitioning of the data. Our aim is to demonstrate performance that will scale effectively for large, dense databases.

1.4 Thesis Outline

The thesis begins by presenting major algorithms for mining association rules and examining their dealing with non-store-resident data in Chapter 2. The *P*-tree and *T*-tree structures used for the new methods are described with associated algorithms in Chapter 3. Chapter 4 presents the strategies for partitioning source data and candidate sets when the data is much too large to be contained in primary memory for applications involving single processor implementations. Experiments and results of these are presented in Chapter 5. For

applications involving multiple processors, the strategies for distributing data among processors are presented in Chapter 6. And finally, Chapter 7 summarizes the main contribution of the thesis and suggests avenues for future work.

Chapter 2

Background on Knowledge Discovery in Databases

2.1 Introduction

In this chapter a summary is presented to review the literature on Knowledge Discovery in Databases (KDD). The organisation of the chapter is as follows: major stages of the KDD process are outlined in section 2.2, and techniques of data mining are classified in section 2.3. The problems of mining association rules are described in detail in section 2.4. Sections 2.5 then presents major sequential algorithms for mining association rules and observes their dealing with non-store-resident data. Distributed/parallel algorithms for mining association rules are outlined in section 2.6, and finally a summary in section 2.7.

2.2 The KDD Process

KDD (Knowledge Discovery in Databases) refers to the overall process of discovering useful knowledge in data. The KDD process has been analysed by many researchers [Debuse *et al.*, 2000] [Weiss and Indurkhya, 1998] [Fayyad *et al.*, 1996a] [Brachman and Anand, 1996]. The problem of knowledge extraction from large databases involves many stages and a unique scheme has not yet been agreed upon. However, the stages of the KDD process can be categorized as: *Problem Specification, Resourcing, Data Cleansing, Pre-processing, Data Mining, Evaluation of Results, and Interpretation of Results.*

2.2.1 Problem Specification

This first stage of the KDD process aims to develop an understanding of the application domain and identify the goal of the application. The resources necessary to carry out the project should be determined at this stage. It is necessary to undertake some preliminary database examinations in order to determine factors such as the size of the database and the number of attributes. At this stage the data itself may not be available so a description of the data might be used. However, the user requires a solid understanding of the domain in order to select the correct subsets of data, suitable classes of patterns and good criteria for interestingness [Bayardo and Agrawal, 1999] [Silberschatz and Tuzhillin, 1995] [Klemettinen *et al.*, 1994] of the pattern.

2.2.2 Resourcing

The purpose of this stage is to create a target dataset on which discovery is to be performed. This stage comprises the collection of the resources required to carry out the project. Often the most time consuming resource required to collect is the data itself. The data may not be readily available. For example, it may come from different sources and require consistent combining and formatting.

2.2.3 Data Cleansing

The purpose of this stage is to ensure that the data is correct, i.e. errors or noise in the data are removed. The problem of outliers and missing values might also be handled at this stage although they could also be dealt with in the pre-processing or data mining stages. The distinction between data cleansing and pre-processing is that no learning or knowledge discovery takes place when data cleansing take place. Data cleansing is usually carried out only once for each database.

2.2.4 Pre-processing

Pre-processing is undertaken to prepare the data so that it is ready for input into the data mining stage. A number of operations may be undertaken at this stage that can improve the performance of data mining, including deciding on strategies for handling missing data fields and reordering attributes of the databases. Pre-processing may be repeated as the process continues.

2.2.5 Data Mining

Data Mining is the discovery of valuable information from large data volumes, using computationally efficient techniques [Zomaya *et al.*, 1999] [Fayyad *et al.*, 1996d] [Chen *et al.*, 1996]. It consists of applying data analysis and discovery algorithms that, under acceptable computational efficiency limitations, produce a particular enumeration of patterns over the data [Fayyad *et al.*, 1996a]. The data-mining component of the KDD process often involves repeated iterative application of particular data mining methods. The explosive growth of databases makes the scalability of data mining techniques increasingly important. Data mining gives organizations the tools to search through these large data sets to find the trends, patterns, and correlations that can guide strategic decision-making. This research focuses on the data mining stage of the KDD process, the data mining process is therefore considered in more detail in section 2.3.

2.2.6 Evaluation of Results

This stage is used to determine if the discovered patterns are valid. Apparently interesting patterns may simply be discovered due to random variation in the data rather than some real world phenomenon. A common method used to validate results is to use separate training and test sets of data. The training data is used to discover patterns, which are then tested on the 'unseen' test data. The performance of the patterns on the test data gives an indication of

the strength of the patterns on other 'unseen' data. Another validation method, that is often used when limited volumes of data are available, is cross validation [Fayyad *et al.*, 1996c].

2.2.7 Interpretation of Results

Presenting the data mining results to the users is extremely important because the usefulness of the results is dependent on it. As in the evaluation stage, it would be expected that new knowledge would tend to match existing knowledge and would be explainable by the domain experts. Obviously, some difference between known and discovered knowledge is required for patterns to be novel. However, if the differences are very great it may be because errors have been made in the KDD process.

2.3 Data Mining

Finding useful patterns in data has been given a variety of names including data mining, knowledge extraction, information discovery, information harvesting, data archaeology, and data pattern processing [Fayyad *et al.*, 1996a]. Two major goals of data mining are: *prediction* and *description*. Prediction involves using some variables or fields in the database to predict unknown or future values of other variables of interest, and description focuses on finding human-interpretable patterns describing the data [Fayyad *et al.*, 1996c]. The distinction between prediction and description is useful for understanding the overall discovery goal although the boundaries between them are not sharp. Some of the predictive models can be descriptive and vice versa. The task of data mining can be classified according to the kind of knowledge to be mined. Three major data mining problems are: *Classification, Clustering, and Association Rule Mining (ARM).*

2.3.1 Classification

Classification is learning a function that classifies a data item into one of several predefined classes [Fayyad *et al.*, 1996a] [Fayyad *et al.*, 1996c] [Weiss and Kulikowski, 1991] [Hand, 1981]. It is a predictive model and often referred to as supervised learning because the classes are determined before examining the data [Dunham, 2003]. Classification often describes these classes by looking at the characteristics of data already known to belong to the classes.

To construct the classification model a sample dataset is treated as the training set, and the remainder is used as the test set. One designated attribute in the training set is called the dependent attribute and the others the predictor attributes. The goal is to build a model that takes the predictor attributes as inputs and produces a value for the dependent attribute. If the dependent attribute is numerical the problem is a regression problem otherwise it is called a classification problem [Ganti *et al.*, 1999]. Researchers have proposed many classification models such as: neural networks [Bishop, 1995], genetic algorithms [Goldberg, 1989], decision tables [Kirk, 1965], and classification trees [Breiman, 1984].

2.3.2 Clustering

Clustering is a descriptive model that identifies a finite set of categories or clusters to describe the data [Fayyad *et al.*, 1996a] [Fayyad *et al.*, 1996c] [Jain and Dubes, 1988] [Titterington *et al.*, 1985]. It distributes data into several groups so that similar objects fall into the same group or cluster. Clustering is similar to classification except that the groups are not predefined, but rather defined by the data alone [Dunham, 2003]. Clusters are defined by finding natural groupings of data items based on similarity metrics or probability density models [Fayyad *et al.*, 1996b]. Clustering is alternatively referred to as unsupervised learning.

Clustering analysis helps to construct meaningful partitioning of large data sets based on a "divide and conquer methodology". The clustering problem has been studied in many

fields, including statistics, machine learning, and biology. However, scalability was not a design goal in these applications; researchers always assumed the complete data set would fit in main memory, and the focus was on improving the clustering quality [Ganti *et al.*, 1999]. With the application of clustering to very large data sets involved in data mining the issue of scalability has assumed increasing importance.

2.3.3 Association Rule Mining

Association Rule Mining (ARM) is one of the core data mining tasks [Agrawal *et al.*, 1996] [Park *et al.*, 1995]. ARM, a descriptive model, refers to the data-mining task of uncovering relationships among data. An association rule is an implication of the form $A \Rightarrow B$, relating disjoint sets of database attributes, which is interpreted to mean "if the set of attribute-values A is found together in a database record, then it is likely that the set B will be present also". These rules are often used in the retail sales community to identify items that are frequently purchased together. The research objective which is the focus of this thesis is concerned with association rule mining; a much more detailed description is therefore given in the following sections.

2.4 Association Rule Mining Problem

ARM (Association Rule Mining) involves the extraction of association rules from a binary database [Agrawal *et al.*, 1993a]. In a database of this kind, each attribute simply records the presence or absence of some property in the record. In other kinds of database, continuously-valued attributes can be dealt with by discretization, and, for convenience of processing, most methods convert multiple-valued attributes into a number of binary attributes, or *items*, each of which can be said to be present or absent in each record. The paradigmatic example is in supermarket shopping-basket analysis. In this case, each record in the database is a representation of a single shopping transaction, recording the set of all items purchased in

that transaction. With respect to binary valued data sets an *association rule* is a relationship between disjoint sets of these items, obtained from examination of the data. The task is to discover important associations among items such that the presence of some items in a transaction will imply the likely presence of other items in the same transaction; that is, rules that associate one set of attributes of a relation to another. A formal statement of the problem can be found in [Agrawal *et al.*, 1993a] [Cheung *et al.*, 1996a]:

Definition 2.1: Let $I = \{I_1, I_2, \ldots, I_n\}$ be a set of n distinct attributes, also called *items*. Let D be a database, where each record or transaction T is a set of items such that $T \subseteq I$. An *association rule* is a relationship between disjoint sets of the form $A \Rightarrow B$, where $A \subset I$, $B \subset I$, and $A \cap B = \varnothing$. Here, A is called antecedent, and B consequent.

A rule is usually only of potential interest if the set of items (the *itemset*) it associates occurs together relatively frequently in the data being examined. These frequent sets are defined to be those which exceed some defined level of *support* ($\dot{s}$). The support of an itemset A, denoted as $\dot{s}(A)$, is the number of records in which that itemset occurs as a subset. An itemset is *frequent* or *large* if its support is more than a user specified minimum support. A frequent itemset is *maximal* if it is not a subset of any other frequent itemset. Support of an association rule can be defined as follows.

Definition 2.2: The *support* ($\dot{s}$) for the rule $A \Rightarrow B$ is the number of database records which contain $A \cup B$.

Therefore, if we say that the support for the rule $A \Rightarrow B$ is 10% then it means that 10% of the total records contain $A \cup B$. The support represents the statistical significance of an association rule. A high support is often desirable for association rule mining although this is not always the case. In applications such as medical epidemiology, we may be searching for

rules that associate rather rare items, and thus a low support-threshold is required. Another important parameter which can be related to each association rule is its *confidence* ($\dot{c}$). It can be defined as follows.

Definition 2.3: For a given number of records, the *confidence* ($\dot{c}$) of the rule $A \Rightarrow B$ is the ratio of the number of records that contain $A \cup B$ to the number of records that contain A; i.e.

$$\dot{c}\ (A \Rightarrow B) = \dot{s}\ (A \cup B)\ /\ \dot{s}\ (A)$$

Thus, if we say that the rule $A \Rightarrow B$ has a confidence of 80%, it means that 80% of the records containing A also contain B. The confidence of the rule indicates the degree of correlation in the dataset between A and B. Confidence is a measure of a rule's strength. In many applications, a high confidence (80-90%) will be required for a rule to be regarded as interesting. However, this will depend very much on the nature of the data.

Mining of association rules from a database consists of finding all rules that meet the user-specified thresholds of support and confidence. A rule is *frequent* if its support is greater than minimum support and *strong* if its confidence is more than the user-specified minimum confidence. The problem of mining association rules can be decomposed into two sub-problems [Agrawal *et al.*, 1993a] as stated in Algorithm 2.1.

Algorithm 2.1. Basic

Input:

Itemset (I), Database (D), Support threshold ($\dot{s}$) and Confidence threshold ($\dot{c}$)

Output:

Association rules satisfying $\dot{s}$ and $\dot{c}$,

Algorithm:

1) Find all sets of items which occur with a frequency that is greater than or equal to the user-specified support threshold $\dot{s}$.

2) Using the frequent itemsets, generate rules which meet the threshold of confidence ç.

The first step in Algorithm 2.1 finds all *frequent* or *large* itemsets. Itemsets other than frequent itemsets are referred as *infrequent* or *small* itemsets. The second step in Algorithm 2.1 generates strong and interesting rules from the frequent itemsets obtained in the first step.

The identification of the frequent itemsets is computationally expensive [Agrawal and Srikant, 1994]. The problem arises because the number of possible sets is exponential in the number of possible attribute-values. For most real data, the number n of such items is likely to be such that counting the support of all 2^n itemsets is infeasible. For this reason, almost all methods attempt to count the support only of *candidate* itemsets that are identified as possible frequent sets. It is, of course, not possible to completely determine the candidate itemsets in advance, and it will therefore be necessary to consider many itemsets that are not in fact frequent. Once these frequent sets have been found, it is relatively straightforward to identify the rules that are likely to be of interest [Agrawal *et al.*, 1993a]. The algorithm for finding association rules is stated in Algorithm 2.2.

Algorithm 2.2. Find Association Rules from Frequent Itemsets

Input:

Frequent Itemsets (F) and Confidence threshold (ç)

Output:

Association rules satisfying ç

Algorithm:

1) Find all nonempty subsets, x, of each frequent itemset, f

3) For every subset, obtain a rule of the form $x \Rightarrow (f - x)$ if ṣ (f) / ṣ (x) $\geq$ ç

Since finding frequent itemsets in a substantial database is very expensive and dominates the overall cost of mining association rules, most research has been focused on

developing efficient algorithms to solve step 1 in Algorithm 2.1 [Agrawal and Srikant, 1994] [Cheung *et al.*, 1996a] [Klemettinen *et al.*, 1994]. In general, algorithms for finding frequent sets involve one or (usually) several passes of the source data, in each of which the support for some set of candidate itemsets is counted. The performance of these methods, clearly, depends both on the size of the original database, typically millions or billions of records, and on the number of candidate itemsets being considered. The number of possible candidates increases with increasing *density* of data (greater number of items present in a record) and with decreasing support thresholds. For this reason, efficient algorithms for finding frequent sets remain a major area of research. The following section provides an overview of the major sequential algorithms.

2.5 Sequential Algorithms for Finding Frequent Sets

Most algorithms developed for finding frequent sets are sequential, or derived from sequential algorithms. Sequential algorithms use a single processor system of limited memory. In most cases, they attempt to find all frequent sets by reducing the number of database passes and/or the search space. Most sequential ARM algorithms, including Apriori [Agrawal and Srikant, 1994], make use of the "downward closure" property to reduce the search space. The observation is that if an itemset is adequately supported, then all the subsets of the itemset will also be adequately supported. Consequently if an itemset is not supported, any effort to calculate the support for its supersets will be wasted. These algorithms proceed in general by attempting to count the support only for those *candidate* itemsets which are identified as possible frequent sets, rather than all possible combinations of items. To assist in identifying these candidate sets it is helpful to observe that the subsets of a set of items may be represented as a lattice. A lattice of this form is shown in Figure 2.1. The notation *ABC* will be used to represent the set {A, B, C}.

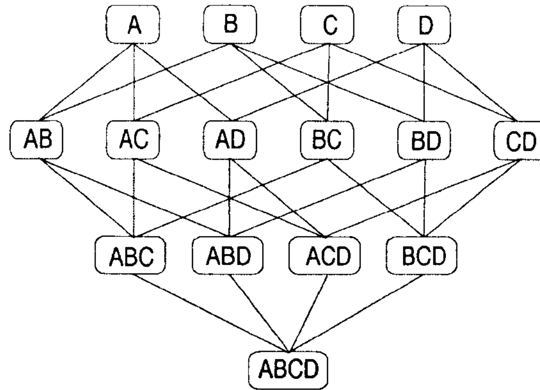


Figure 2.1: Lattice of subsets of {A, B, C, D}

For any set of items to be frequent, it is required that all its subsets to be frequent. For example, an essential condition for the set *ABC* to be considered as interesting is that *AB*, *AC*, and *BC* are all frequent which in turn requires that each of *A*, *B*, and *C* exceeds the required threshold of support. This observation provides a basis for pruning the lattice of subsets to reduce the search space. If it is known that *D* is not supported then it is no longer necessary to consider *AD, BD, CD, ABD, ACD, BCD* or *ABCD*. Algorithms that proceed on this basis reduce their requirement for storage and computation by eliminating infrequent candidates as soon as they are able to do so. Major sequential algorithms discussed in the following subsections are: (1) AIS, (2) Apriori, (3) Apriori-TID and Apriori-Hybrid, (4) Partitioning, (5) Sampling, (6) Dynamic Itemset Counting (DIC), (7) Methods that find Maximal Frequent Sets, (8) Depth-Project, and (9) *FP*-Growth.

2.5.1 AIS

AIS is the first published algorithm developed in 1993 to find all frequent itemsets in a transaction database [Agrawal *et al.*, 1993a]. It focused on processing databases for decision support by discovering qualitative association rules. This technique is limited to only one item in the consequent. In the notation used in [Agrawal *et al.*, 1993a], an association rule is

expressed in the form $A \Rightarrow I_j \mid ç$, where A is a set of some items in a domain I, I_j is a single item in I that is not present in A, and $ç$ is the confidence of the rule.

The AIS algorithm makes multiple passes over the database. In the first pass, it counts the support of individual items and determines which of them are frequent in the database. Subsequent passes examine itemsets of size 2,3,4,... successively. Frequent itemsets of each pass are extended to generate candidate itemsets in the following pass. Candidate itemsets are generated "on the fly" during the pass as data is being read. After reading a transaction, the common itemsets between frequent itemsets of the previous pass and items of this transaction are determined. These common itemsets are extended with other individual items in the transaction to generate new candidate itemsets. A frequent itemset A is extended with only those items in the transaction that are frequent and occur in the lexicographic ordering of items later than any of the items in A [Agrawal and Srikant, 1994]. Items are kept in lexicographic order to avoid duplication of an itemset. The candidates generated from a transaction are added to the set of candidate itemsets maintained for the pass, or the counts of the corresponding entries are increased if the candidate already exists in the set. This process terminates when no more frequent itemsets are found in a pass.

The major shortcoming of this algorithm is that it generates too many candidate itemsets which cannot be frequent, wasting too much effort. For example, if XYZ is frequent and P, an item of a transaction, is frequent then the itemset $XYZP$ will be included in the set of candidates even if, for example, XP is not frequent. However, $XYZP$ cannot be frequent if XP is infrequent. Generation of a huge number of candidate sets might cause the memory buffer to overflow. Therefore, it may become necessary to store the candidate sets in secondary memory.

A similar scheme was proposed in [Houtsma and Swami, 1995]. The algorithm, SETM, used standard SQL to calculate frequent itemsets. Like AIS, it also generates candidates based on transactions read from the database. However, it separates candidate

generation from counting. Like AIS, the major shortcoming of this algorithm is that it generates too many candidate itemsets that later turn out to be infrequent [Agrawal and Srikant, 1994]. A further disadvantage in that it used TIDs (Transaction IDs) to generate candidates, which required more space to store as many entries as the number of transactions in which the candidate itemset is present. Moreover, at the end of each pass repeated sorting of itemsets is needed.

2.5.2 Apriori

The Apriori algorithm [Agrawal and Srikant, 1994] has been the basis of very many subsequent association rule algorithms. This algorithm introduces the "downward closure" property that any subset of a frequent itemset must be frequent. Also, it is again assumed that items within an itemset are kept in lexicographic order. The fundamental difference of this algorithm from the AIS and SETM algorithms is the way of generating candidate itemsets. The Apriori algorithm generates the candidate itemsets using only the itemsets found frequent in the previous pass – without considering the transactions in the database [Agrawal and Srikant, 1994]. It generates candidate itemsets only when all the subsets of these are found frequent. By only considering frequent itemsets of the previous pass, the number of candidate itemsets is significantly reduced.

Apriori performs repeated passes of the database, successively computing support-counts for sets of single items, pairs, triplets, and so on. At the end of each pass, sets that fail to reach the required support threshold are eliminated, and candidates for the next pass are constructed as supersets of the remaining (frequent) sets. The iterative process terminates when no new frequent itemsets are found. Since no set can be frequent which has an infrequent subset, this procedure guarantees that all frequent sets will be found.

The first pass of the algorithm simply counts item occurrences to determine the frequent 1-itemsets (F_1). A subsequent pass, say pass k, consists of two phases. First, the

frequent itemsets of size (k-1), F_{k-1}, is joined with itself to obtain candidates of size k, C_k. Next it scans all transactions to obtain supports of candidates in C_k. An overview of the algorithm is shown below:

Algorithm 2.3. Apriori [Agrawal and Srikant, 1994]

Input:

Itemset (I), Database (D), and Support threshold ($\dot{s}$)

Output:

Frequent Itemsets (F)

Algorithm:

1) F_1 = {frequent 1-itemsets};

2) **for** (k = 2; $F_{k-1} \neq \emptyset$; k++) **do** **begin** // k represents the pass number

3) C_k = apriori-gen(F_{k-1}); // New candidates of size k generated from F_{k-1}

4) **forall** transactions $t \in D$ **do begin**

5) C_t = subset(C_k, t); // Candidates contained in t

6) **forall** candidates $c \in C_t$ **do**

7) c.count++; // Increment the count of all candidates

8) **end**

9) F_k = { $c \in C_k$ | c.count ≥ $\dot{s}$ } // Candidates in C_k with minimum support $\dot{s}$.

10) **end**

11) F := $\cup_k F_k$;

Candidate generation

The Apriori candidate generation function, apriori-gen() [Agrawal and Srikant, 1994], has two steps. In the first step, F_{k-1} is joined with itself to obtain C_k. In the second step (prune), it deletes all itemsets from the join result, which have some (k-1)–subset that is not in F_{k-1}. Then, it returns the remaining frequent k-itemsets.

Consider the following example for candidate generation: Let F_3 be {*ABC*, *ABD*, *ACD*, *ACE*, *BCD*}. The set of candidate itemsets, C_4, is generated from F_3 as follows. From F_3, two itemsets with common initial items, such as *ABC* and *ABD*, are identified first and then joined using the last two disjoint items, producing *ABCD*. After the join step, C_4 will be {*ABCD*, *ACDE*}. The prune step will delete the itemset *ACDE* because not all of its subsets, e.g. *ADE*, are in F_3. It will then be left with only *ABCD* in C_4 to be counted. Apriori stores these candidate itemsets in a *hash tree* [Agrawal and Srikant, 1994] structure.

Evaluation of Apriori

Two aspects of the performance of this algorithm are of concern. The first is the number of database passes that are required and the second is the number of candidates which may be generated, especially in the early cycles of the algorithm. One of the inherent performance weaknesses of Apriori is that it requires the source data to be scanned repeatedly; in principle, the number of passes required is one greater than the size of the largest frequent set. This is computationally expensive if, as is likely to be the case in many applications, the source data cannot be contained in primary memory and many passes of the data may also be required. Apriori also generates very many candidates in the early cycles [Agrawal and Srikant, 1994]. This is a problem, especially for low support thresholds and when the database is very densely populated, as in these cases the number of candidates to be considered may become too large to be contained in primary memory. A large number of candidates also leads to slow counting (steps 5-7 of Algorithm 2.3), including the time to locate candidates in the hash tree.

2.5.3 Apriori-TID and -Hybrid

As mentioned earlier, Apriori scans the entire database in each pass to count support. Scanning of the entire database may not be needed in all passes. Based on this conjecture, [Agrawal and Srikant, 1994] proposed another algorithm called Apriori-TID (TID for transaction identifier). Apriori-TID uses the Apriori's candidate generating function to determine candidate itemsets before a pass. The main difference from Apriori is that it does not use the original database for counting support after the first pass. Rather, it uses an encoding of the candidate itemsets used in the previous pass denoted by $\overline{C_k}$. As with SETM, each member of the set $\overline{C_k}$ is of the form <TID, {X_k}> where each X_k is a potentially frequent *k*-itemset present in the transaction with the identifier TID. $\overline{C_1}$ is essentially the original set of database transactions, each in the form of a TID with a list of items contained in the transaction. Subsequently, the member of $\overline{C_k}$ corresponding to transaction *t* is <*t*.TID, {$c \in C_k \mid c$ contained in *t*}>. If a transaction does not contain any candidate *k*-itemset, then $\overline{C_k}$ will not have an entry for this transaction. Thus the number of entries in $\overline{C_k}$ may be smaller than the number of transactions in the database, especially for larger values of *k*. In addition, for large values of *k*, each entry may be smaller than the corresponding transaction because very few candidates may be contained in the transaction.

Since Apriori-TID uses $\overline{C_k}$ rather than the entire database after the first pass, it is effective in later passes when $\overline{C_k}$ becomes smaller. However, for small values of *k*, each entry may be larger than the corresponding transaction because an entry in C_k includes all candidate *k*-itemsets contained in the transaction. This means that the size of $\overline{C_k}$ generated in the early passes of the algorithm may generally exceed the size of the original dataset. This problem is at its worst if a large number of 2-itemsets are supported.

As mentioned on page 496 [Agrawal and Srikant, 1994], Apriori has better performance in early passes, and Apriori-TID outperforms Apriori in later passes in cases when its $\overline{C_k}$ sets can fit in the memory. Based on this observations, the Apriori-Hybrid

technique was developed. In the early stages of the algorithm Apriori is used when the number of candidates is high. Apriori-TID is activated when the number of candidate sets is deemed small enough to allow the $\overline{C_k}$ to fit into main memory. Therefore, an estimation of $\overline{C_k}$ at the end of each pass is necessary. Also, there is a cost involvement of switching from Apriori to Apriori-TID. The switching point between the two algorithms is critical to the performance because once the switch is made all the subsequent $\overline{C_k}$ sets generated in each pass must fit in main memory which may not always be possible.

2.5.4 Partitioning

The major drawback of Apriori (and Apriori-TID) is that in dealing with databases that are much too large to hold in main memory, repeated passes of the database are required to compute the support for single items, pairs etc., in turn. The Partition algorithm [Savasere *et al.*, 1995] reduces the number of database passes from backing store to two. It divides the database into a number of non-overlapping segments of equal size that are small enough to be handled in main memory. The algorithm applies the Apriori level-wise procedure to each data segment in turn; retaining the segment in primary storage throughout its repeated passes. Hence the passes through the in-memory data segments are quicker. After loading a segment into the main memory there is no additional disk I/O for the segment. For each segment, thus, a set of *locally* frequent itemsets is determined, each of which reaches the proportionate threshold of support in that segment. This method uses the property that a frequent itemset in the whole database must be locally frequent in at least one partition of the database. The union of the local frequent itemsets are used to generate *global* candidates. A second pass of the complete database is required to establish which of the locally frequent sets are (globally) frequent. To efficiently generate the candidates, the Partition method stores the itemsets in 'sorted' order. It also stores references to the itemsets in a hash table data structure. An overview of the Partition algorithm is given below:

Algorithm 2.4. Partition [Savasere *et al.*, 1995]

Input:

Itemset (I), Database (D), Number of partitions (N) and Support threshold ($\dot{s}$)

Output:

Frequent Itemsets (F)

Algorithm:

1) **for** i = 1 to N **do**

2) read_in_partition($D_i \in D$);

3) F^i= gen_frequent_itemsets(D_i); // Locally frequent itemsets

4) C := $\cup_i F^i$; // Global candidates

5) **for** i = 1 to N **do**

6) read_in_partition($D_i \in D$);

7) **for all** candidates $c \in C$ gen_count(c, D_i);

8) $F = \{ c \in C \mid c.\text{count} \geq \dot{s} \}$

The advantage gained by this method is that by working with a subset of the database which can be contained in main memory, repeated access of database records becomes acceptable. The Partition algorithm, however, favours a homogeneous (have similar frequent itemset distribution) data. That is, if the count of an itemset is evenly distributed in each partition, then most of the itemsets to be counted in the second pass will be frequent. However, for a skewed data distribution, many of the itemsets in the second pass may turn out to be infrequent, thus wasting a lot of CPU time counting false itemsets. AS-CPA (Anti-Skew Counting Partition Algorithm) [Lin and Dunham, 1998] is a family of anti-skew algorithms, which were proposed to improve the Partition algorithm when data distribution is skewed. In the first pass, the counts of the itemsets found in the previous partitions will be accumulated and incremented in the later partitions.

The drawback of these approaches, however, highlights the second weakness of Apriori that the number of candidates whose support is to be counted may become very large, especially when the data is such that the frequent sets may contain many items (the "long pattern" problem [Bayardo *et al.*, 1999] [Agarwal *et al.*, 2000]). If, for example, there is just one set of 20 items that reaches the threshold of support, then the methods inescapably require the support for all the 2^{20} subsets of this set to be counted. In the Partition algorithm, this is exacerbated because there may be many more sets that are locally frequent in some partition, even though they are not globally frequent. These methods require all candidates to be retained in primary memory (for efficient processing) during the final database pass which may not always be possible. The Partitioning method also chooses partitions such that all data structures can be accommodated in the main memory although in some situations it may be necessary to store the temporary data on disk [Savasere *et al.*, 1995].

2.5.5 Sampling

The aim of reducing database passes also motivated the strategy introduced in the *sampling* algorithm of [Toivonen, 1996]. Here, the idea is to pick a random sample of the data, use it to determine all frequent sets that probably hold in the whole database, and then verify the results with the rest of the database. A random sample of the source data, small enough to fit in primary memory, is first processed using the Apriori procedure, with a modified support threshold and other modifications designed to make it likely that all the globally frequent sets will be identified in the sample. The sets thus found become candidates for a single full pass of the source data to verify this.

It is necessary for the initial processing of the sample to identify an enlarged candidate set, to give a reasonable probability that all the actual frequent sets will be included. To do this, first, the support threshold is lowered so much that it is very unlikely that any frequent sets are missed. Then, after finding all the *locally frequent* sets in the sample

being examined, this collection is augmented by adding its `negative border' [Toivonen, 1996] to produce global candidates. The intuition behind the concept of negative border is that given a collection of sets that are frequent in the sample, the negative border contains the "closest" itemsets that could be frequent, too. The *negative border* (B_d^-) is the collection of all itemsets that are not frequent in the sample but all of whose subsets are.

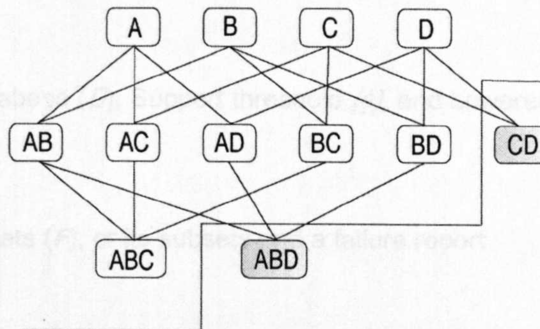


Figure 2.2: Example of a negative border

Figure 2.2 shows an example of a negative border. Itemsets *CD* and *ABD* are not frequent in the sample but all their subsets are. These itemsets on the negative border are marked after the initial processing of the sample. The significance of the negative border is that it defines a boundary between the frequent and non-frequent sets in the sample. A single full database pass is then carried out to count final supports of all itemsets including that of negative border. If no set on the negative border is finally found to be frequent, then no sets `outside' the border can be frequent either. The approach of Toivonen's method is probabilistic and in some cases it may happen that all frequent itemsets are not found in one pass; that means a *failure* [Toivonen, 1996]. The approach of Toivonen's method is probabilistic and in some cases it may happen that all frequent itemsets are not found in one pass; that means a *failure* [Toivonen, 1996]. A failure is identified by checking the negative border. If no set on the negative border meets the support threshold, then no frequent set has been missed. If, however, it is possible to construct a superset from frequent sets in the negative border, then

this superset has been missed and may be a frequent set. In this case a further pass is required to check this. For example, if the itemset *CD* in Figure 2.2 is found to be frequent, then its supersets *ACD* and *BCD* may also be frequent (because *AC, AD, BC,* and *BD* are frequent). An overview of the algorithm is shown below:

Algorithm 2.5. Sampling [Toivonen, 1996]

Input:

Itemset (I), Database (D), Support threshold ($\dot{s}$), and Lowered support threshold ($l\dot{s}$)

Output:

Frequent Itemsets (F), or its subsets and a failure report

Algorithm:

1) D_s = a random sample drawn from D;

2) $F^s = Apriori(I, D_s, l\dot{s})$; // Frequent sets in the sample

3) $C = F^s \cup Bd^-(F^s)$; // Candidate set

4) **forall** transactions $t \in D$ **do begin**

5) C_t = subset(C, t); // Candidates contained in t

6) **forall** candidates $c \in C_t$ **do**

7) c.count++; // Increment the count of all candidates

8) **end**

9) $F = \{ c \in C \mid c.\text{count} \geq \dot{s} \}$

10) Report if *failure* // if any $c \in Bd^-(F^s) \mid c.\text{count} > \dot{s}$

The candidate set generated from the sample includes all sets that are frequent, using the lowered support threshold, in the sample together with their negative border. This candidate set is likely to be much larger than the actual frequent itemsets. This method also requires all the enlarged candidate sets to be retained in primary memory (for efficient processing) during the final database pass, which may not always be possible.

2.5.6 Dynamic Itemset Counting

The DIC (Dynamic Itemset Counting) algorithm [Brin *et al.*, 1997] tries to generate and count the itemsets earlier so that the number of database passes can be reduced. DIC is also a variant of Apriori that differs in how the candidate itemsets are generated. It starts checking itemsets after an interval, rather than waiting until the database pass is completed. The intuition behind DIC is that it works like a train running over the data with stops at several intervals. When the train reaches the end of the transaction file, it has made one pass over the data and it starts over at the beginning for the next pass. The "passengers" on the train are itemsets and their occurrence on the train is counted in the transactions.

While scanning the first interval, the 1-itemsets are generated and counted. At the end of the first interval frequent 1-itemsets are determined and candidate 2-itemsets are generated as supersets of the frequent sets. These 2-itemsets are counted together with 1-itemsets while scanning the second interval. At the end of the second interval, candidate 3-itemsets are generated and counted during the third interval with the 1-itemsets and 2-itemsets. In general, at the end of the kth interval, the candidate $(k+1)$-itemsets are generated as supersets of the frequent k-itemsets and counted in the later interval together with the previous candidate itemsets. After reaching the end of the database, it starts over at the beginning for the next pass and counts the itemsets which are not fully counted. The DIC algorithm works as follows:

Algorithm 2.6. DIC [Brin *et al.*, 1997]

Input:

Itemset (I), Database (D), Range (R), and Support threshold (s)

Output:

Frequent Itemsets (F)

Algorithm:

1) The empty itemset is marked with a solid box. All the 1-itemsets are marked with dashed circles. All other itemsets are unmarked.
2) Read *R* transactions. For each transaction, increment the respective counters for the itemsets marked with dashes.
3) If a dashed circle has a count that exceeds $\dot{s}$, turn it into a dashed square. If any immediate superset of it has all of its subsets as solid or dashed squares, add a new counter for it and make it a dashed circle.
4) If a dashed itemset has been counted through all the transactions, make it solid and stop countin it.
5) If we are at the end of the transaction file, rewind to the beginning.
6) If any dashed itemsets remain, go to step 2.

This method favours homogeneous data, as does the Partition algorithm. If data is not homogeneous, DIC might generate many itemsets that are locally frequent but not globally (false positives) [Zaki, 1999]. If the data is very correlated, it may not realize that an itemset is actually frequent until most of the intervals have been counted. The data structure used in DIC is similar to a hash tree and it needs to store extra information at each node.

2.5.7 Methods that find Maximal Frequent Sets

Apriori-like algorithms, such as those described above, employ a bottom-up search that enumerates every single frequent itemset. This implies that in order to produce a frequent itemset of size n; it must produce all 2^n subsets since they must be frequent too. This exponential complexity fundamentally restricts Apriori-like algorithms to discovering only short patterns [Bayardo, 1998]. The long pattern problem is so difficult to solve computationally, that even for databases of relatively smaller sizes, it is very difficult to find long patterns. For problems in which the patterns are longer than 15-20 items, and the database is too large to

fit into primary memory, most of the algorithms which require the generation of all subsets of frequent itemsets are impractical anyway [Agarwal *et al.*, 2000]. It is tempting, consequently, to look for methods which seek to identify *maximal* interesting sets without first examining all their smaller subsets.

Instead of generating all subsets of frequent itemsets the Max-Miner algorithm [Bayardo, 1998] extracts only the 'maximal' frequent itemsets. An itemset is maximal frequent if it is frequent and has no frequent superset. This algorithm searches for maximal sets, using Rymon's generic set-enumeration tree [Rymon, 1992]. The intention of using this framework was to expand sets over an ordered and finite search space as a tree. Note that the tree could be traversed depth-first, breadth first, or even best first as directed by some heuristic. This tree is central to the methods which are the basis of the thesis, and will be described fully in the next chapter. Max-Miner employs a purely breath-first search of the set-enumeration tree in order to limit the number of passes made over the data. However, like Apriori, the number of passes over the data made by Max-Miner is bounded by the size of the longest frequent itemset [Bayardo, 1998]. An overview of the Max-Miner algorithm is shown below:

Algorithm 2.7. Max-Miner [Bayardo, 1998]

Input:

Itemset (I), Database (D), and Support threshold (s)

Output:

Maximal Frequent Itemsets (F^m)

Algorithm:

1) $C = \varnothing$; // Candidate set

2) F = gen_initial_groups(D, C);

3) $C = F$;

4) **while** $C \neq \varnothing$ **do**

5) Scan D to count the support of all candidates in C;

6) **for each** $c \in C$ such that $h(c) \cup t(c)$ is frequent **do**

7) $F = F \cup \{h(c) \cup t(c)\}$; // h = head or parent, t = tail or child

8) $C_{new} = \varnothing$; // New candidate set

9) **for each** $c \in C$ such that $h(c) \cup t(c)$ is infrequent **do**

10) $F = F \cup \{gen_sub_nodes(c, C_{new})\}$;

11) $C = C_{new}$;

12) Remove from F any itemset with a proper superset in F;

13) Remove from C any candidate c such that $h(c) \cup t(c)$ has a superset in F;

14) $F^m = F$;

Max-Miner always attempts to "look ahead" in order to quickly identify maximal frequent itemsets. For this, this algorithm not only uses subset infrequency based pruning, as does Apriori, but also uses pruning based on superset frequency. These strategies are used to prune entire branches of the data structure from consideration. Itemset (re)ordering is also used to increase the effectiveness of superset frequency pruning. In a development from Max-Miner, the Dense-Miner algorithm [Bayardo *et al.,* 1999] imposes additional constraints on the rules being sought to reduce further the search space.

MaxEclat and MaxClique algorithms [Zaki *et al.,* 1997] also try to identify maximal frequent itemsets. These algorithms are similar to Max-Miner in that they also attempt to look ahead and identify maximal frequent itemsets early on to help prune the candidate itemsets considered. The important difference is that Max-Miner attempts to look ahead throughout the search, whereas MaxEclat and MaxClique look ahead only during an initialization phase prior to a purely bottom-up Apriori-like search with exponential scaling. The initialization phase of MaxClique is also prone to problems with maximal frequent itemsets since it uses a dynamic programming algorithm for finding maximal cliques in a graph whose largest clique is at least the size of the longest frequent itemset [Bayardo, 1998]. However, the method is of interest, as it introduces a different kind of partitioning. In this, candidate sets are partitioned into

clusters which can be processed independently. The problem with the method is that, especially when dealing with dense data and low support thresholds, expensive pre-processing is required before effective clustering can be identified. The partitioning by *equivalence class*, however, is relevant to the methods which will be described later.

The described long pattern mining algorithms may cope better with dense and store-resident data than the other algorithms described, but again usually involve multiple database passes. However, all association rules cannot be efficiently extracted from maximal frequent itemsets alone, as this would require performing the intractable task of enumerating and computing support of all their subsets [Bayardo *et al.,* 1999]. Therefore, these algorithms may not be convenient for finding all frequent sets with their support counts as required for mining all association rules.

2.5.8 Depth-Project

The DepthProject algorithm [Agarwal *et al.,* 2000] also generates 'maximal' frequent itemsets. The main difference from Max-Miner is that it uses depth-first search instead of breath-first search. A similar lexicographic set enumeration tree is also used to store maximal frequent itemsets. The root of the tree corresponds to the null itemset and points to the entire transaction database.

The DepthProject algorithm is recursive in nature, so that the call from each node is an independent itemset generation problem, which finds all frequent itemsets that are descendants of a node. The first call of the algorithm is from the root (null) node. The first step of the algorithm generates all the candidate extensions of the itemset node *N*. For the case of the null node, this call returns all the single items in the database. For other nodes, the procedure returns frequent extensions of *N* which are lexicographically larger than any item in *N*. This set of items occurs lexicographically after the node and are considered to be the possible lexicographic frequent extension of the node. Then it counts the support of each of

these candidate extensions. This method also uses the 'look ahead' [Bayardo, 1998] pruning technique. An overview of the algorithm is shown below:

The DepthProject algorithm is explicitly targeted for finding maximal frequent sets in cases when the database patterns are very wide in memory-resident data. This technique may not be useful for applications where the number of records in the database is large. However, like other maximal frequent mining methods (see sub-section 2.5.7), this algorithm also does not perform the intractable task of enumerating and computing support of all the subsets of maximal sets that are required for mining all association rules, which cannot be efficiently extracted from maximal frequent itemsets alone. Therefore, this algorithm may also not be convenient for counting the required supports of all frequent itemsets.

2.5.9 FP-Growth

Apriori-like algorithms count supports for maximal frequent itemsets as well as all their subsets by generating and testing candidates. The generation of candidates can, however, be costly in terms of efficiency. It is also tedious to repeatedly pass the entire database and check a large set of candidates. The *FP*-growth method [Han *et al.*, 2000] is an alternative method that aims to avoid these problems.

A *frequent pattern* tree (*FP*-tree) structure is used for storing the database in a compressed form. The *FP*-tree is an extended prefix-tree structure for storing quantitative information about frequent patterns. The *FP*-tree stores only a frequent single item at each node, and includes additional links to facilitate processing. Additional links are included in the *FP*-tree so that the tree can be processed from the leaf nodes as well as from the root node. These links start from a header table and link together all nodes in the *FP*-tree which store the same 'label', i.e. item. To construct a *FP*-tree it needs two passes of the database. The construction process begins with an initial pass of the database to count support for the single items. Those that fail to meet the threshold support are eliminated, and the others ordered by

decreasing frequency. The database is then passed through a second time to construct an initial *FP*-tree. If multiple transactions within the database share an identical frequent set they are merged. If two transactions share a common prefix, the shared parts are merged. The *FP*-tree contained in primary memory is then used, instead of the original data, to find the complete set of frequent sets. The *FP*-tree algorithm works as follows:

Algorithm 2.8. FP-tree [Han *et al.*, 2000]

Input:

Itemset (*I*), Database (*D*), and Support threshold (*s*)

Output:

FP-tree (*T*)

Algorithm:

1) Scan *D* and collect the set of frequent 1-itemsest, F_1.

2) Sort F_1 in descending order.

3) $C = F_1$; // Candidate set

4) **forall** transactions $t \in D$ **do begin**

5) $c = t \cap C$; // Frequent itemset in *t*

6) Sort *c* according to the order of *C*.

7) T = insert_tree(*c*, *T*);

8) **end**

For example, given the data set {{A,C,D}, {B,D,E}, {B,D,F}}, the *FP*-tree construction process begins by identifying the frequent 1-itemsets in the data. It is assumed, for the illustration, that all single items are frequent. Note that the new ordering of the items, according to decreasing frequency, will be {D,B,A,C,E,F}. The data set is required to be passed again to construct an initial *FP*-tree. Record 1 in the data set ({A,C,D}) is read first and placed in the *FP*-tree, creating from the root node, as shown in Figure 2.3 (a) (note the links for the header table).

The second record ({B,D,E}), the first element of which is common with an existing node, is then added in the *FP*-tree as shown in Figure 2.3 (b). The last record ({B,D,F}) is then added to complete the initial *FP*-tree as shown in Figure 2.3 (c).

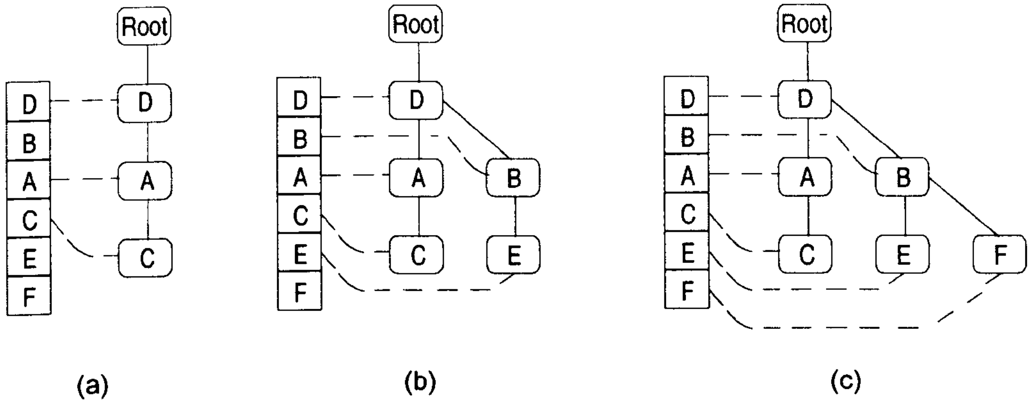


Figure 2.3: FP-tree

The algorithm, *FP*-growth, for mining the *FP*-tree structure is a recursive procedure during which many sub *FP*-trees and header tables are created. The process commences by examining each item in the header table, starting with the least frequent. For each entry the support value for the item is produced by following the links connecting all occurrences of the current item in the *FP*-tree. If the item is adequately supported, then for each leaf node a set of *ancestor labels* is produced (stored in a prefix tree), each of which has a support equivalent of that of the leaf node from which it is generated. If the set of ancestor labels is not null, a new tree is generated with the set of ancestor labels as the dataset, and the process repeated.

The advantage offered by the *FP*-growth algorithm is partly gained from the ordering process, which reduces the overall size of the input dataset (because unsupported single items are eliminated), and reduces processing time by allowing the most common items to be processed most efficiently. However, although an *FP*-tree is rather compact, its construction needs two passes of the data. Also, one cannot assume that an *FP*-tree can always fit in main

memory for any large databases [Han *et al.*, 2000]. The multiply linked structure of the *FP*-tree causes problems for non-store-resident implementation. The CATS tree [Cheung and Zaiane, 2003], an extension of the *FP*-tree, also assumes no limitation on main memory capacity.

2.6 Parallel Algorithms

The space and time complexity of ARM has led some researchers to propose parallel processing [Cheung *et al.*, 1996a]. Parallel algorithms, to confront the problem, use multiple processors instead of a single processor. Most parallel/distributed ARM algorithms are adaptations of existing sequential (serial) algorithms. These algorithms focus on how to parallelize the task of finding frequent itemsets between multiprocessor systems.

Two dominant approaches for using multiple processors are *distributed memory* and *shared memory* systems. In a distributed memory system, the approach to multiprocessing is to build a system from many units, each containing a processor and memory. Here each processor has its own private or local memory, which only that processor can access directly. For a processor to access data in the local memory of another processor, a copy of the desired data elements must be passed from one processor to another. A different approach to multiprocessing is to build a system from many units where all processors share common memory. In a shared memory system, each processor has direct and equal access to all the system's memory.

The multiple-processor paradigm can be categorised into *data parallelism* and *task parallelism* [Chattratichat *et al.*, 1997]. In data parallelism, the database is partitioned among processors. Each processor works on its local partition of the database and performs the computation of counting the local support for the global candidate itemsets, exchanging counts as required with other processors. In the task parallelism paradigm, the candidate set is partitioned and distributed across the processors. Each processor has access to the entire

database but is responsible for some disjoint subset of the candidates. Every processor performs its different computation independently and exchanges its local candidate counts with all the other processors.

Neither method necessarily provides a complete solution to the problems of dealing with very large databases. Data parallelism requires that all the candidates fit into the main memory of each processor. Conversely, task parallelism requires that each processor has complete access to the full database. Both methods, details of which are given below, require processes to exchange information to maintain global counts, involving a significant communication overhead. Thus, strategies for effective data partitioning are very relevant for parallel implementations also. However, both methods are discussed in more detail in the following subsections.

2.6.1 Data Parallelism

In data parallelism architecture, each processor has its own local memory, which only that processor can access directly. For a processor to access data in the local memory of another processor, "message passing" must take place between processors. Thus, the data parallelism corresponds to the case where the data is apportioned among the processes, typically by "horizontally" segmenting the dataset into sets of records. Each process then mines its allocated segment (exchanging information on-route as necessary).

The "count distribution algorithm" [Agrawal and Shafer, 1996] is an example of the data parallelisation approach. The algorithm attempted to minimize communication by duplicating the candidate sets with counts in each processor's memory. The algorithm operates as follows:

1. Divide the dataset among the available processors so that each processor is responsible for a particular horizontal segment.

2. Determine the local support counts for the candidate 1-itemsets.
3. Exchange the local counts with other processors so that each processor obtains the global support count for all 1-itemsets (a process which Agrawal refers to as *global reduction*).
4. Each processor then prunes the 1-itemsets and generates a set of candidate 2-itemsets from the supported 1-itemsets and then determines the local support for each of these candidate sets, and so on.

A disadvantage of the count distribution algorithm is that each processor generates the same number of candidates as the serial corresponding algorithm. Both [Park *et al.,* 1995] and [Cheung *et al.,* 1996b] suggest modifications to the above algorithm. [Park *et al.* 1995] make use of their "direct hashing technique" to prune candidate sets. [Cheung *et al.,* 1996b] also suggest more advanced candidate pruning and global reduction techniques. The Data Allocation Algorithm (DAA) [Manning and Keane, 2001] uses a data preprocessing tool, Principal Component Analysis (PCA) [Joliffe, 1994], to improve the distribution of data and then uses the algorithm Fast Parallel Mining (FPM) [Cheung and Xiao, 1998]. DAA improves the performance of FPM for data of high sparsity though may not work well for very large and dense data. The first parallel algorithm considered in this thesis in chapter 6, Data Distribution (*DD*), is an adaptation of the count distribution algorithm.

2.6.2 Task Parallelism

Task Parallelism corresponds to the case where the processors perform different computations independently, such as counting disjoint sets of candidates, but have or need access to the entire data. Each process may have a direct and equal access to all system's memory or the process of accessing the database can involve explicit communication of the local portions.

The "data distribution algorithm" [Agrawal and Shafer, 1996] is an example of the task parallelisation approach. The algorithm takes the approach where each processor works with the entire data but only a portion of the candidate set. Each process thus requires access to the set of candidate itemsets generated at each level. Broadly the algorithm operates as follows (Agrawal and Shafer actually also horizontally segment and distribute the dataset):

1. Equally distribute the candidate 1-itemsets among the processors.
2. Each processor generates support counts for its local candidate sets using both local data and data received from other processors. Then the resulting frequent 1-itemsets are pruned.
3. Each processor exchanges its frequent-1 set with those associated with other processors.
4. Generate the next level candidate sets, distribute these sets equally among the processors and repeat until there are no more candidate item sets.

A disadvantage associated with the above task parallel algorithm is the amount of messaging that will be involved, especially in step 2. During this step, every processor sends its local data to all other processors as well as receiving the local data of other processors. Consequently a number of authors (for example Han *et al.* 1997, Shitani *et al.* 1996, Zaki *et al.* 1997) have described alternatives to improve on this. The second parallel algorithm considered in this thesis (in chapter 6), Task Distribution (*TKD*), is a variation of the task parallelism concept using different mechanism for dividing the candidate itemsets between the available processes.

2.7 Summary

The problem of finding association rules falls within the purview of data mining or knowledge discovery in databases. Mining all frequent itemsets in a large and dense (greater number of items present in a record) dataset is the most demanding ARM problem. Most algorithms developed for finding frequent itemsets are sequential, or based on sequential algorithms. These basic algorithms have already been discussed in this chapter. As mentioned earlier, most algorithms use the frequent itemsets found on the previous pass(es) to generate candidate sets. The algorithms assume that candidate itemsets are held in the main memory to obtain their support counts. All algorithms also require that all or part of the original data to be held in the main memory. In general, performance of all methods suffers if extensive disk access is required during processing. Thus most algorithms assume that there is enough memory to handle these problems.

Databases continue to increase in dimensions; consequently even a single scan of this type of data is considered expensive. Performance is even more badly affected if multiple passes of disk resident data are required. Most sequential algorithms scan data multiple times hence they are not really scalable. All current methods require, for efficiency, the candidate sets to be contained in primary memory. Data mining is really targeted at extremely large databases, far too large to be contained in main memory. This thesis seeks to address these disadvantages by considering various strategies for partitioning data (including candidate sets) so that the current "partition" can be stored in primary memory. This approach will be expanded upon in the following chapters.

Chapter 3

Review of Computing Support via Partial Totals

3.1 Introduction

For reason of computational efficiency ARM requires efficient mechanisms for storing the candidate sets under consideration together with their support. When the density of data is high, and/or the support threshold is low, the number of candidate sets may become very large. In these cases, especially, a data structure that enables effective linking between candidate nodes is an important aspect of performance. This chapter describes two suitable data structures, and associated algorithms, both developed at Liverpool University for this purpose. The *P*-tree structure, which computes partial supports, is described in section 3.2. The concept of computing total support, using the *T*-tree structure, from the partial supports is described in section 3.3.

3.2 Partial Support

Most of the methods described in the previous chapter proceed essentially by defining some candidate set and then examining each record to identify all the members of the candidate set that are subsets of the record. The computational cost of this increases with the density of the database which leads to an exponential increase in the number of subsets or candidates to be counted. In principle, however, it is possible to reduce this cost of subset-counting by exploiting the relationships between sets of attributes illustrated in Figure 3.1. For example, in the simplest case, a record containing the attribute set *BCD* will increase the support-counts for each of its subsets *BCD, BC, BD, CD, B, C* and *D*. Strictly, however, only

the first of these is necessary, since minimum support values of all the subsets of *BCD* can be inferred subsequently from the support-count of *BCD*.

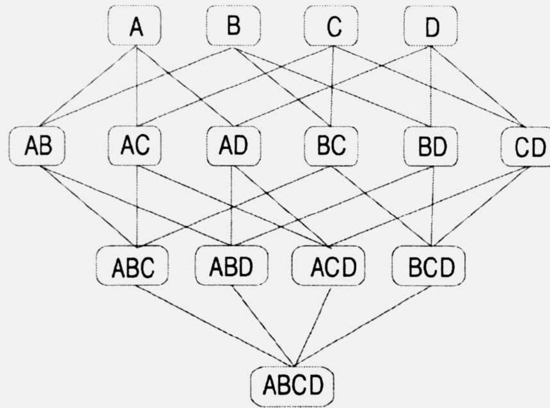


Figure 3.1: Relationship between subsets of {A,B,C,D}

Let us see, however, how the total support of a set can be computed from the support counted for its supersets. Let i be a subset of the set I (where I is the set of n attributes represented by the database). The *partial support* for the set i, P_i, is defined to be the number of records whose contents are identical with the set i. Then T_i, the *total support* for the set i, can be determined as [Goulbourne *et al.*, 2000]:

$$T_i = \sum P_j \qquad (\forall j, j \supseteq i)$$

Thus, the total support of any set can easily be counted from the partial support of its supersets. The algorithms to count the support of sets of attributes can thus be defined in two phases.

Phase 1: Count the partial support of all the sets that appear as distinct records in the data. If there are m records, there will be at most m sets to be counted (fewer if some records are identical).

Phase 2: Use these partial counts to determine the total support of the candidate sets whose support we need to determine.

Computing partial supports, in phase 1, allows capturing all the relevant information of the database in a single pass and in a form that enables efficient computation of the support totals. The initial counting will also improve overall performance if there are duplicated records in the data. There are other advantages, however, which will become apparent. A form of Rymon's set enumeration framework [Rymon, 1992] is used for storing the partial supports. In a set enumeration tree, each subtree contains all the supersets of the root node which follow the root node in lexicographic order. Figure 3.2 shows the *complete* P-tree (Partial support Tree) for all the subsets of *I* with their partial support counts, for *I* = {A,B,C,D}. The dataset used to build the tree is: {{A}, {A,B}, {A,C}, {A,D}, {A,B,C}, {A,B,D}, {A,C,D}, {A,B,C,D}, {B}, {B,C}, {B,D}, {B,C,D}, {C}, {C,D}, {D}}.

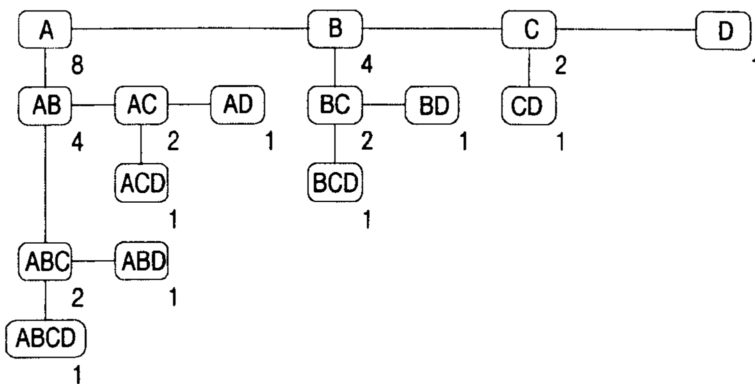


Figure 3.2: Complete P-tree for all the subsets of {A,B,C,D}

As can be seen in the Figure, the tree consists of subtrees rooted at attributes *A, B, C* and *D*. All these subtrees contain lexicographically-following supersets of their root node. For example, the subtree rooted at *B* includes only sets *BC, BD* and *BCD*. Note that all these sets are headed by the root node *B* and follow *B* in lexicographic order. Using this tree as a storage structure for partial support-counts is straightforward and computationally efficient: locating the required position on the tree for any set of attributes starts directly from the root of a subtree and requires at most *n* steps. Also, when locating a node on the tree, the traversal may pass through a number of nodes which are subsets of the target node. This advantage of

the structural relationship is taken to accumulate *interim* support-counts at these nodes. The interim support counts will include the partial support of the sets and of the succeeding supersets. The algorithm for constructing the *P*-tree proceeds as follows:

1. For $K = 1, 2, \ldots.$
2. Read record K of the data.
3. Traverse the tree to find the location of the record, starting at the node representing the first attribute of K.
4. Create a new node in the tree for the record if it is not already present.
5. Increment the count for all parent nodes traversed in the course of finding the location.
6. Repeat steps (2), (3), (4) and (5); until K is the final record of the data.

The algorithm proceeds by comparing the record R to be inserted with node B on the tree built so far, starting with the root node. The actions that follow (step 3-5 of the algorithm above) apply 5 rules [Goulbourne *et al.*, 2000]. Note the $<$ and $>$ operators are used to define lexicographic not numeric ordering, thus:

Rule 1 ($R = B$, i.e. an identical node is found):

Simply increment the support associated with the current node and return.

Rule 2 ($R < B$ and $R \subset B$, i.e. new record is a parent of current node):

1. Create a new node for R and place the existing node associated with B on the new node's child branch.
2. Place the new node either as a new root node, or add it to the child or sibling branch of the previously investigated node.
3. If necessary move one or more of the younger siblings of the previously existent node up to become younger siblings of the newly created node.

Rule 3 ($R < B$ and $R \not\subset B$, i.e. new record is an elder sibling of existing node):

1. If R and B have a leading sub string S and S is not equal to the code associated with the parent node of the current node representing B then:
   - Create a dummy node for S with support equivalent to that associated with the node for B.
   - Place the new dummy node either as a new root node, or add it to the child or sibling branch of the previously investigated node.
   - Then create a new node for R and place this so that it is a child of the newly created dummy node.
   - Finally place the previously existent node for B as a younger sibling of the node for R.
2. Else create a new node for R and place the existing node associated with B on the new node's sibling branch. The new node is then placed either as a new root node, or is added to the child or sibling branch of the previously investigated node.

Rule 4 ($R > B$ and $R \subset B$, i.e. new record is a child of current node):

Increment the support for the current node (B) by one and:

1. If node associated with B has no child node, create a new node for R and add this to the existing node's child branch.
2. Otherwise proceed down child branch and apply the rules to the next node encountered.

Rule 5 ($R > B$ and $R \not\subset B$, i.e. new record is a younger sibling of current node):

1. If node associated with B has no sibling node, and:
   - If current node (node for B) does not have a parent node and R and B have a leading sub string S then:
     - Create a dummy node for S with support equivalent to that associated with the node for B.

- Place the new dummy node either as a new root node, or add it to the child or sibling branch of the previously investigated node.
- Create a new node for *R* and place this so that it is a younger sibling of the newly created dummy node.
- Place the previously existent node for *B* as a child of the node for *R*.

- Otherwise proceed down sibling branch and apply the rules to the next node encountered.

To find the proper location of a record, the algorithm passes through a number of nodes which are subsets of the target node, incrementing the support count of each en route. For example, to locate the set *BCD* in the tree, the algorithm passes through its subset *B* and then *BC*, incrementing the count of each of these.

This algorithm will compute interim support-counts Q_i for each subset i of I, where Q_i is defined thus:

$$Q_i = \sum P_j \qquad (\forall j, j \supseteq i, j \text{ follows } i \text{ in lexicographic order})$$

It then becomes possible to compute total support using the equation:

$$T_i = Q_i + \sum P_j \qquad (\forall j, j \supset i, j \text{ precedes } i \text{ in lexicographic order})$$

The counts stored at each node of Figure 3.2 are *interim* support-totals, representing support derived from the set and its succeeding supersets in the tree. The numbers associated with the nodes in this illustration would arise if every combination of the attributes were present exactly once in the database; thus, for example, $Q(BC) = 2$, derived from one instance of *BC* and one of *BCD*. The total support of *BC* can then be computed by adding this interim support with the partial supports of its supersets which precedes *BC* in lexicographic order. Thus:

$$\begin{aligned} T(BC) &= Q(BC) + P(ABC) + P(ABCD) \\ &= Q(BC) + Q(ABC) \\ &= 2 + 2 = 4 \end{aligned}$$

The term *P*-tree (Partial support Tree) is used to refer to this *incomplete* set-enumeration tree of interim support-counts. To store counts for a complete *P*-tree, as in Figure 3.2, of course implies a storage requirement of order 2^n. This can be avoided, however, from the observation that for large n it is likely that most of the subsets i will be unrepresented in the database and will therefore not contribute to the partial-count summation. A version of the algorithm to exploit this builds the tree dynamically as records are processed, storing partial totals only for records which appear in the database. Nodes are created only when a new subset i is encountered in the database, or when two siblings i and j share a leading subset which is not already represented. The latter provision of creating "dummy" nodes is necessary to maintain the structure of the tree as it grows. To maintain the lexicographic order of the tree it may also be necessary to "move up" siblings from a current node to become siblings of a new parent node. The formulae for computing total supports still apply, and it needs only to sum interim supports that are present in the tree. Building the tree dynamically implies a storage requirement of order m (the number of records in the database) rather than 2^n. The *P*-tree, thus, contains all the sets of items present as distinct records in the database, plus some additional sets that are leading subsets of these. A detailed description of the algorithm for building the *P*-tree can be found in [Goulbourne *et al.*, 2000]. An example illustrating the generation of a *P*-tree is given below.

Example 3.1: *P*-tree generation

Consider the sample dataset as shown in Figure 3.3. The number of attributes of the database, $n = 6$ and number of records, $m = 6$.

```
A C E
A D
A C E F
A C D
D
A C E
```

Figure 3.3: A sample dataset

The *P*-tree generation algorithm begins by reading the first record and creating the node *ACE* with support 1 (fig 3.4 a), then reads the second record *AD*. Since *AD* and the node *ACE* have a common leading subset, *A*, a "dummy" parent node is created with the nodes *ACE* and *AD* as children (fig 3.4 b). A "dummy" node when created gets the sum of the support of its children, but subsequently is treated as a normal node. Record *ACEF* is added as a child of *ACE* incrementing the support for *A* and *ACE* by one en route (fig 3.4 c). *ACD* is added by creating the "dummy" node *AC*, with *ACD* and *ACE* as its children, incrementing the support for *A* en route (fig 3.4 d).

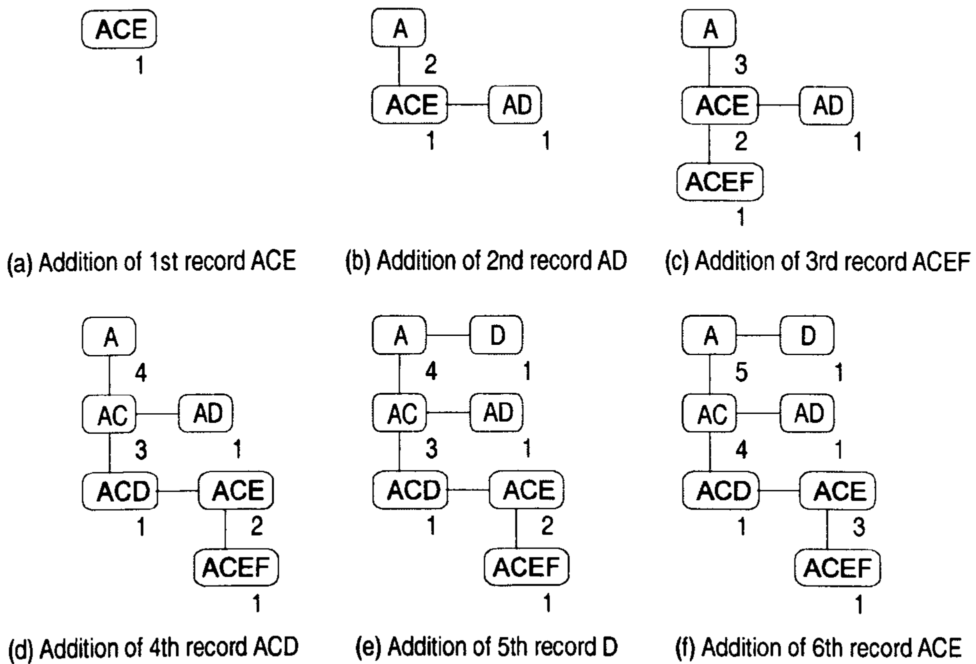


Figure 3.4: P-tree generation example

ACEF still remains as a child of *ACE*. Record *D* is added creating a new node as a sibling of *A* with support 1 (fig 3.4 e). The last record *ACE* is then included as shown in fig 3.4 (f). Note that because this last record already exists in the tree (duplicate record) its support is incremented by one, incrementing the supports of its parent nodes *A* and *AC* en route.

From inspection of the final *P*-tree of the above example (fig 3.4 f), it is apparent that it is unnecessary to store any particular itemset which is duplicated in its child, say for

example *ACE* is duplicated in its child *ACEF*, so the actual implementation omits this. Thus the itemset *ACEF* is stored as *F*. The tree in its final form can be represented as shown in Figure 3.5 and this form of *P*-tree will be used in the remaining chapters.

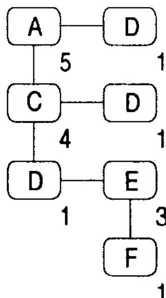


Figure 3.5: P-tree in its final form

3.3 Computing Total Supports

The construction of the *P*-tree essentially performs, in a single database pass, a reorganisation of the data into a structured set of attribute sets which appear as distinct records in the database (plus the dummy nodes) with their partial summation. For any candidate set *S* of subsets of *I*, the calculation of total supports can be completed by walking this tree, adding interim supports as required according to the formulae above.

Consider again the set *BC* with reference to Figure 3.2. The phase 1 computation has already added into the interim support-count, the contribution from the supersets which follow *BC* in the tree-ordering, i.e. the contribution from *BCD*. To complete the summation of total support, any contribution from preceding supersets, i.e. *ABC* and *ABCD*, must be added. But any contribution from *ABCD* has already been accumulated in *ABC*, so addition of the latter is only needed. Now consider the converse of this. The partial total accumulated at *ABCD* makes a contribution to the total support for all the subsets of *ABCD*. However, the contribution in respect of the subset *ABC* is already included in the interim total for *ABC*, so, when considering the node *ABCD*, only those subsets which include the attribute *D* need to

be considered. In general, for any node j, it is necessary to consider only those subsets of j which are not also subsets of its parent.

Thus, the following algorithm completes the summation of total supports for a candidate set S [Coenen *et al.*, 2004a]:

Algorithm 3.1.

Input:

P-tree (P), Candidate set (S)

Output:

Counts T_i for all sets i in S

Algorithm:

$\forall$ sets i in S do $T_i = 0$

$\forall$ nodes j in P do

begin $k = j$ - parent (j);

$\forall$ i in S, $i \subseteq j$, $i \cap k$ not empty, do

begin add Q_j to T_i

end

end

This algorithm is essentially generic and can be applied in different ways depending on how the candidate set is defined. In general, methods like Apriori which involve multiple database passes should gain, from using the *P*-tree to replace the original database, for two reasons: firstly, because any duplicated records are merged in the *P*-tree structure, and secondly, because the partial computation incorporated in the *P*-tree is carried out only once, when the tree is constructed; thus reducing the computation required on each subsequent pass. Therefore, in the second pass of Apriori, using the *P*-tree would in the best case require only $r - 1$ subsets of a record of r attributes to be considered (i.e. those not covered by its parent)

rather than the $r(r-1)/2$ required by the original Apriori. For example, consider the case of a set *ABCD* that is present in the *P*-tree as a child of *ABC*. When the parent node *ABC* is processed in the course of algorithm 3.1, its interim support will be added to the total support of all its subsets. But, this interim support incorporates the support for its child *ABCD*. So, when *ABCD* is processed, its support count is only needed to determine the total support for the sets *AD*, *BD* and *CD*. This advantage becomes greater, of course, the larger the sets being considered. It is this property that gives the major performance advantage gained from using the *P*-tree: although the candidate set being considered is not reduced, the number of candidates within the set that need to be examined at each step is reduced, reducing the overall time required to search the candidate set.

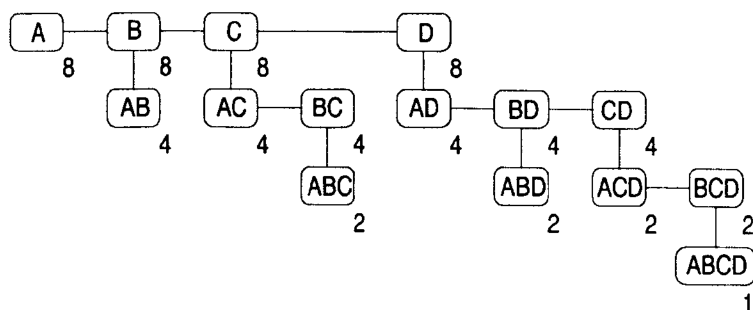


Figure 3.6: Complete T-tree for all the subsets of {A,B,C,D}

Figure 3.6 illustrates a *complete* T-tree (Total support Tree) for all the subsets of *I* with their final support counts. As can be seen, the T-tree also consists of subtrees rooted at attributes *A*, *B*, *C* and *D*. But, unlike the *P*-tree, each sub-tree includes only supersets of its root node which contain attributes that lexicographically precede or are equal to the root node. For example, the subtree rooted at *C* includes its supersets *AC*, *BC* and *ABC*. Note also that all these supersets terminate with the subtree root *C*. Here node *C* is the parent of nods *AC* and *BC*. This structure is called a *T*-tree, representing the target sets for which the total support is to be calculated. Here, for any node *t* in the *T*-tree, all the subsets of *t* which include an attribute *i* will be located in that segment of the tree found by a depth-first traversal

starting at node i and finishing at node t. This allows the *T*-tree to be used as a structure to effect an implementation of algorithm 3.1 [Coenen *et al.*, 2001]:

Algorithm 3.2. TFP (Compute Total- from Partial- support)

Input:

P-tree (P), *T*-tree (T)

Output:

Total support counts for T

Algorithm:

```
∀ nodes j in P do
    begin k = j - parent (j);
        i = first attribute in K;
        starting at node i of T do
        begin if i ⊆ j then add Q_j to T_i;
                    if i = j then exit
                    else recurse to child node;
            proceed to sibling node;
        end
    end
```

In any practical method, however, it is necessary to create a subset of the *T*-tree using only the current frequent sets and to remove all the infrequent sets. Thus, a version of the Apriori algorithm using this structure is used to construct candidates of singletons, pairs, triplets, etc., in successive passes. This algorithm is known as the *Apriori-TFP* (Total-from-Partial) algorithm and essentially uses the *P*-tree rather than the original database. The algorithm has the following form [Coenen *et al.*, 2001] [Coenen *et al.*, 2004a]:

1. *K* = 1
2. Build level *K* in the *T*-tree.
3. "Walk" the *P*-tree, applying algorithm *TFP* to add interim support associated with individual *P*-tree nodes to the level *K* nodes established in (2).
4. Remove any level *K* *T*-tree nodes that do not have an adequate level of support.
5. Increase *K* by 1.
6. Repeat steps (2), (3), (4) and (5); until a level *K* is reached where no nodes are adequately supported.

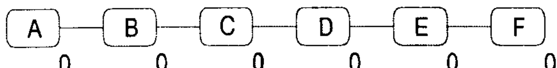
The detailed Apriori-TFP algorithm is described in [Coenen *et al.*, 2001] [Coenen *et al.*, 2004a]. On completion of the algorithm, the *T*-tree finally contains all frequent sets with their complete support-counts. An example illustrating the generation of a *T*-tree using the algorithm is given below.

Example 3.2: *T*-tree generation

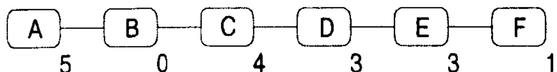
To build a *T*-tree for a user specified support threshold, the *P*-tree of Figure 3.5 is used instead of the original data set. Let us see the level wise building of a *T*-tree for a support threshold of 50% (equivalent to 3 records in this case). The generation process of the *T*-tree is shown in Figure 3.7.

***T*-tree Level 1**

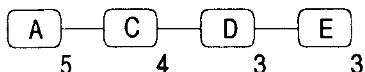
Initially the top level of the *T*-tree is created for all 6 candidates (fig 3.7 a). The *P*-tree is then traversed to add interim supports associated with individual *P*-tree nodes to the candidate nodes. After adding supports from the *P*-tree, the set of singletons with their final support totals are obtained (fig 3.7 b). As can be seen in fig 3.7 (b), support of attributes *B* and *F* are less than the support threshold (50% or 3). So these singletons are pruned to get the final top level or the frequent singletons (fig 3.7 c).



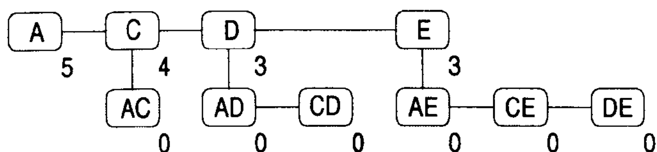
(a) Top level candidates



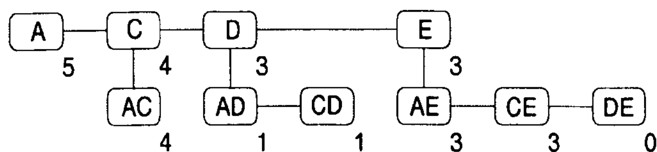
(b) After adding supports of P-tree



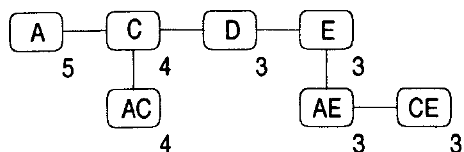
(c) Top level, after pruning



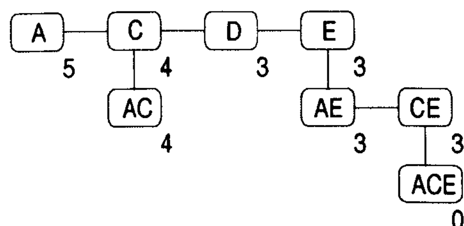
(d) Generation of second level candidates



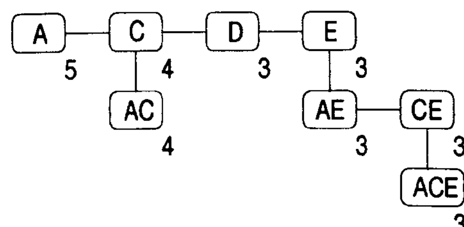
(e) After adding supports of P-tree



(f) After pruning level two



(g) Generation of third level candidate



(h) After adding supports of P-tree

Figure 3.7: T-tree generation example

T-tree Level 2

The supported singletons are used to generate candidate set pairs or *T*-tree level-2. As can be seen in fig 3.7 (d), all the subsets of *T*-tree level-2 candidates are contained in the tree built so far, i.e. they have the necessary threshold of support. To determine the total supports for these level 2 *T*-tree nodes the *P*-tree is traversed a second time, and for each *P*-tree node, interim supports associated with individual *P*-tree nodes are added to the appropriate candidate pairs (fig 3.7 e). As can be seen in fig 3.7 (e), pairs *AD*, *CD* and *DE* are infrequent so they are pruned to get frequent pairs. Figure 3.7 (f) shows pairs *AC*, *AE* and *CE* as frequent with their support totals.

T-tree Level 3

These pairs are use to generate candidate triplets. Only one candidate triplet, *ACE* is generated (fig 3.7 g) in level 3, as *ACE* is the only triplet all of whose subsets (*AC*, *AE*, *CE*) are supported. After another *P*-tree pass (fig 3.7 h), we see triplet *ACE* is supported and we get the final *T*-tree since there is no way to grow the tree.

Again, from inspection of the final *T*-tree of the above example (fig 3.7 h), it is apparent that duplication of parent to child is unnecessary and can easily be avoided; say for example *CE* is duplicated in its child *ACE*, so the actual implementation omits this. Thus the itemset *ACE* is stored in level 3 as *A*. The tree in its final form can be represented as shown in Figure 3.8 and we will use this form of *T*-tree.

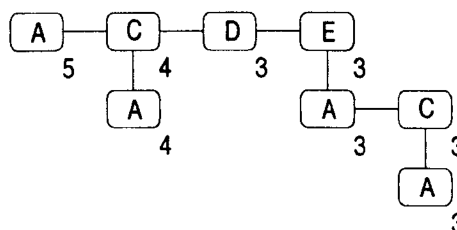


Figure 3.8: T-tree in its final form

3.4 Summary

The *P*-tree and *T*-tree structures and Apriori-TFP algorithm described in this chapter offer significant advantages, in terms of storage and execution time with respect to existing ARM techniques. Results demonstrating these advantages are given in [Coenen *et al.*, 2001]. More detailed algorithms of the *P*-tree and *T*-tree generation can be found in our paper published in [Coenen *et al.*, 2004b]. A rather similar structure to the *P*-tree, the *FP*-tree [Han *et al.*, 2000] has been described in the previous chapter. This structure has a different form but quite similar properties to the *P*-tree, but is built in two database passes, the first of which eliminates attributes that fail to reach the support threshold, and orders the others by frequency of occurrence. Each node in the *FP*-tree stores a single attribute, so that each path in the tree represents and counts one or more records in the database. The *FP*-tree also includes more structural information, including all the nodes representing any one attribute being linked into a list. This structure facilitates the implementation of the "FP-Growth" algorithm which generates subtrees from the *FP*-tree corresponding to each frequent attribute, to represent all sets in which the attribute is associated with its predecessor in the tree ordering. Recursive application of the algorithm generates all frequent sets.

However, unlike the *P*-tree which is essentially a generic structure that can subsequently be used as a basis for many algorithms to complete the count-summation, the *FP*-tree is closely tied to the FP-Growth algorithm. Results presented for FP-Growth demonstrate its effectiveness in cases when the tree is memory-resident, but the linkage between nodes of the tree makes it difficult to effect a comparable implementation when this is not the case. The problem is worsened because of the additional storage overheads of the *FP*-tree, which in general store many more nodes than *P*-tree, and needs more references/pointers to be stored at each node.

The simple *P*-tree structure, however, requires only that each node be linked to its parent. *P*-tree, thus, can only be processed in a "top down" manner. In fact, on completion the

P-tree is converted to a table form (array of arrays) which can be processed in any order. This is possible because the Apriori-TFP algorithm does not require the linkage between nodes in the tree to be maintained. Thus, it is possible to store the "tree" in secondary storage, and consequently the implementation may efficiently scale to deal with non-store-resident data. The major advantages of the *P*-tree in respect of a large and dense database can be summarized thus:

(1) It merges duplicated records and records with leading sub-strings, thus reducing the storage and processing requirements for these.

(2) It allows partial counts of the support for individual nodes within the tree to be accumulated effectively as the tree is constructed.

(3) It uses minimum and effective referencing which will also be capable of dealing with non-store resident data.

(3) Finally, and most importantly, the tree can easily be partitioned into subtrees and stored into secondary storage as a composite structure if required.

The *T*-tree, with its computationally efficient referencing mechanism, is a very versatile structure that can be used in conjunction with many established ARM methods. A major advantage offered by the *T*-tree structure is that branches of it can also be partitioned and processed independently if the candidate set is very large. Also, the structure can readily be adapted for non store-resident data. In the remaining chapters we will examine ways of using these structures in implementations that assume data is much too large to reside in primary memory.

Chapter 4

Strategies for Partitioning Data

4.1 Introduction

The challenge of ARM is to find all frequent sets in a computationally effective manner. This in turn will require use of a suitable data structure. Where the data is too large to be contained in primary memory some effective partitioning mechanism will also be required. Many of the ARM methods described in the literature cannot readily be adapted to deal with non-store-resident data. Where algorithms can be adapted to handle non-store-resident data performance often does not scale linearly with the database size and density. This thesis investigates ways in which Apriori-TFP can be adapted to deal with non-store-resident data given an appropriate division of the data, especially when high data density and/or a low support threshold give rise to very large numbers of candidates. The hypothesis presented here is that the *P*-tree and *T*-tree structures, and consequently Apriori-TFP, can support an effective partitioning of the data and/or candidate sets. In this chapter a number of strategies, most of which are already published in our papers [Ahmed *et al.*, 2003] [Ahmed *et al.*, 2004], are presented for partitioning source data and candidate sets, when the data is much too large to be contained in primary memory.

The organisation of this chapter is as follows: 'horizontal' partitioning, which divides the source data into sets of records, is presented in section 4.2 (Data Partitioning). Section 4.3 describes an approach based on the "Negative Border" concept of [Toivonen, 1996], to find all frequent sets in a single (horizontal) partition of the data. 'Vertical' partitioning, which partitions records into sets of items, is presented in section 4.4 (Tree Partitioning). Finally, an

approach that combines horizontal and vertical partitioning is investigated in section 4.5 (Data and Tree Partitioning).

4.2 Data Partitioning (DP)

When source data cannot be handled as a whole in primary memory one may think of solving the problem by slicing the data into pieces of a reasonable size. The 'natural' implementation of Apriori-TFP, in this context, requires a partitioning of the data into segments of manageable size and to construct a *P*-tree for each. This form of partitioning, in which each segment contains a number of complete database records, will be referred as *data partitioning* (*DP*), or *segmentation.* Here, the database is divided 'horizontally' into a number of non-overlapping segments each of which is small enough to be handled in primary memory.

Each segment of the data is considered in isolation and a *P*-tree created for it that is then stored in secondary memory. The algorithm starts with the first record in the first segment and the construction of the *P*-tree commenced. *P*-tree creation is continued until the end of the segment. The tree is then stored into secondary storage, and reading of records is continued to create a *P*-tree for the next segment, and so on. More formally the algorithm can be presented as follows.

Algorithm 4.1: Data partitioning to create and store *P*-trees

Input:

Database (*D*), Num segments (*M*)

Output:

M *P*-trees in secondary storage.

Algorithm:

1) Divide the data set into *M* segments.

2) For each segment of the data set

i) Create a *P*-tree in primary memory

ii) Store the *P*-tree into secondary storage

In this and the other methods described, each *P*-tree created enumerates all the sets present as distinct itemsets in the corresponding segment, recording for each an incomplete summation of their support-count. The stored *P*-trees are then treated as a composite structure from which the final support totals for all the frequent sets are computed, storing these in a *T*-tree.

For storing, the *P*-tree is converted to a table form (array of arrays). The rows of the *P*-tree table consist of corresponding levels of the *P*-tree; so that all 1-itemsets of the *P*-tree are included in level 1 of the table, 2-itemsets in level 2, and so on. When this is done, of course, we lose the access speed advantages of the tree structure, but the algorithm that we employ require all nodes to be traversed so this aspect is no longer relevant. The Apriori-TFP algorithm does not require the linkage between nodes in the tree to be maintained, and can traverse the *P*-tree table in any order.

The *P*-trees stored in secondary storage are used, in the second stage, to build a single *T*-tree. Instead of passing the database every time, the segmentation process needs to pass the *P*-trees to build each level of the *T*-tree. To build a single *T*-tree from the *P*-trees stored in backing store, a form of the Apriori-TFP algorithm described in the previous chapter is used (see section 3.3). Here, each pass of the algorithm requires each of the *P*-trees to be read in turn from secondary memory. The algorithm is:

Algorithm 4.2: Build a single *T*-tree from stored *P*-trees

Input:

P-trees (P), Num segments (M), Support threshold ($\dot{s}$),

Output:

T-tree satisfying $\dot{s}$

Algorithm:

1) For $K = 1, 2....$

2) Build level K in the T-tree.

3) Read all the stored P-trees in turn, adding interim supports associated with individual P-tree nodes to the level K nodes established in (2).

4) Remove any level K T-tree nodes that do not have an adequate level of support s.

5) Repeat steps (2), (3), and (4); until a level K is reached where no nodes are adequately supported.

The method creates a final single T-tree in primary memory that contains all the frequent sets and their support-counts. Note that the primary memory requirement is for storage to contain the P-tree for a single segment of the data, plus the whole of the T-tree. Here is an example of the method.

Example 4.1

Suppose that a data set of 6 attributes and 6 records, as shown in fig 4.1 (a), cannot be handled as a whole in primary memory but a smaller set of, say, two records can be handled. So the data set is divided into three segments of equal size as shown in fig 4.1 (b).

(a) Dataset	(b) Segmented Dataset	
A C E	A C E	
A D	A D	Segment 1
A C E F	-------------	
A C D	A C E F	
D	A C D	Segment 2
A C E	-------------	
	D	
	A C E	Segment 3

Figure 4.1: Original and Horizontally Segmented Dataset

The algorithm starts reading from the first record (*ACE*) of the data set and creates *P*-trees independently for one segment after another. After creating a *P*-tree, the tree is stored into secondary storage and the memory is cleaned for reuse. Creation of *P*-tree 1 (fig 4.2 a) is completed after reading the last record (*AD*) of segment 1. Then continue reading from the first record (*ACEF*) of the second segment. Creation of *P*-tree 2 (fig 4.2 b) is completed after reading the last record (*ACD*) of segment 2. Reading is continued and *P*-tree 3 is created with records *D* and *ACE*, as shown in fig 4.2 (c), for the last segment of data. These stored *P*-trees are used later to build a *T*-tree for a user specified support threshold.

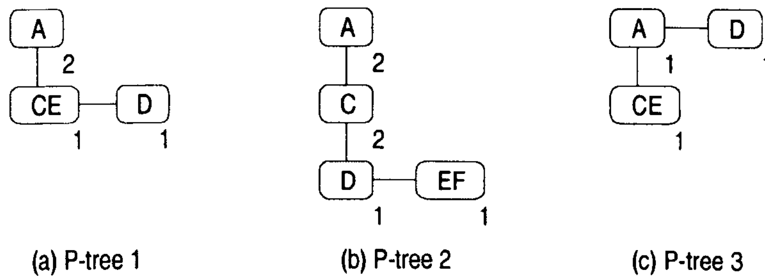


Figure 4.2: P-trees from segmented dataset

_T_-tree Level 1

Let us see the level wise building of a *T*-tree for a support threshold of 50% i.e. 3 records in the above example. The generation process of level 1 is shown in Figure 4.3. First the top level of *T*-tree (fig 4.3 a) is created for all 6 attributes. Then the *P*-trees are read from secondary storage in turn to add interim supports associated with individual *P*-tree nodes to the candidate nodes. The *P*-trees are read in succession to update top-level nodes (shown in fig 4.3 b-d). After adding supports for all the *P*-trees, nodes whose supports are less than the user specified threshold are pruned. It can be seen in fig 4.3 (d) that attributes *B* and *F* are not supported (less than 50% or 3), so they are pruned leaving only supported attributes in the top level (fig 4.3 e).

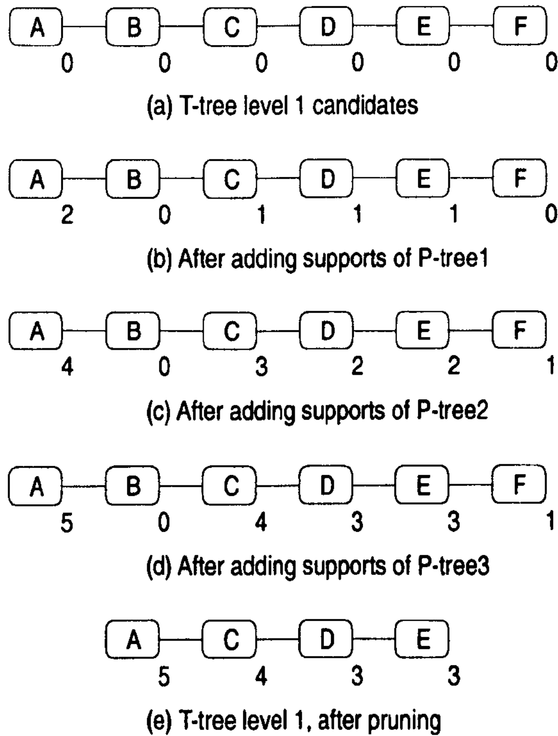
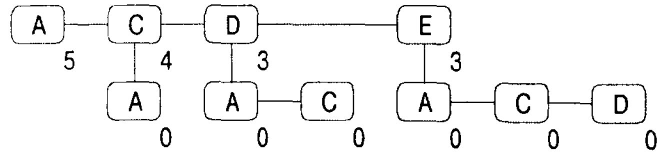


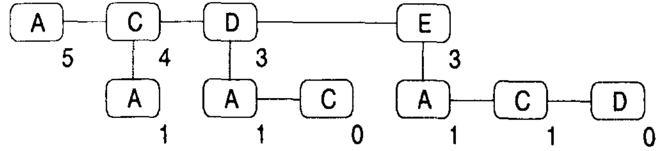
Figure 4.3: Building T-tree level 1

T-tree Level 2

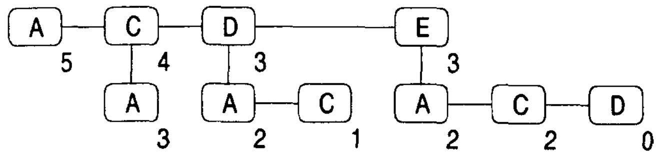
The supported top-level attributes are used to generate candidate set for pairs or *T*-tree level-2. *T*-tree level-2 candidates are shown in fig 4.4 (a). The first *P*-tree is read to update supports for pairs (fig 4.4 b), then the second *P*-tree (fig 4.4 c), and then the third *P*-tree (fig 4.4 d). After pruning level 2 for supports less than the user specified threshold we get *CA*, *EA* and *EC* as supported pairs (fig 4.4 e).



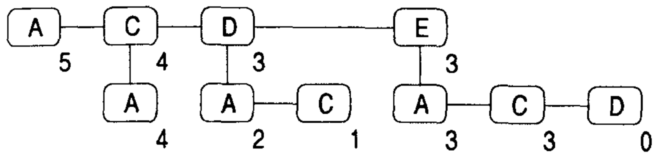
(a) T-tree level 2 candidates



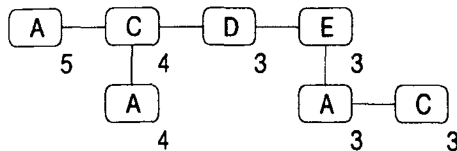
(b) After adding supports of P-tree1



(c) After adding supports of P-tree2



(d) After adding supports of P-tree3

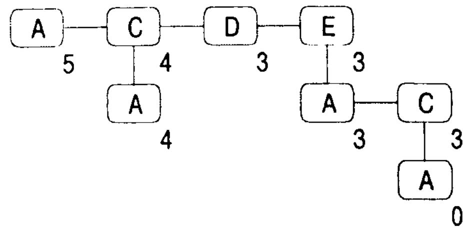


(e) T-tree level 2, after pruning

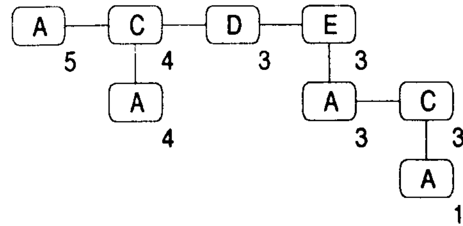
Figure 4.4: Building T-tree level 2

T-tree Level 3

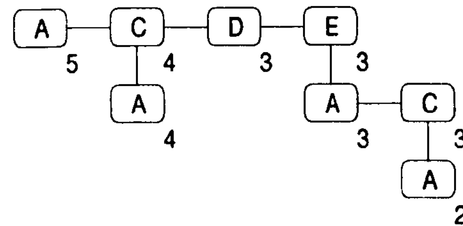
The pairs are used to generate candidate triplets. Only one candidate triplet, *ECA* is generated (fig 4.5 a) in level 3, as *ECA* is the only triplet all of whose subsets are supported. The first *P*-tree is read (fig 4.5 b), then the second *P*-tree (fig 4.5 c) and then the third (fig 4.5 d). It can now be seen that triplet *ECA* is supported. There are no more candidate sets and thus the complete *T*-tree has been obtained.



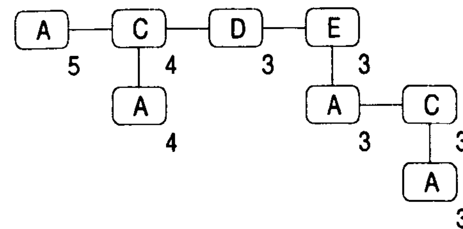
(a) T-tree level 3 candidate



(b) After adding supports of P-tree1



(c) After adding supports of P-tree2



(d) After adding supports of P-tree3

Figure 4.5: Building T-tree level 3

4.3 Negative Border (NB)

The segmentation method described above is the most obvious and simplest way of transforming the Apriori-TFP algorithm into a form applicable when data is not resident in primary memory. The drawback, of course, is that the basic Apriori methodology requires repeated passes to be made of the database to compute the support for single items, pairs

etc., in turn. For Apriori-TFP, the *P*-tree is used instead of the database, but it is still necessary to read each *P*-tree segment repeatedly from backing store. The only, relatively small advantage to be gained from using the *P*-tree rather than the original data is that, in its tabular form, levels of the tree can be discarded at the end of each pass. So, for example, after pass 1 is completed, it is no longer necessary to read the rows of tabulated *P*-tree that represent single items, since these have now been fully counted.

The concept of the *negative border* [Toivonen, 1996] tries to avoid repeated passes of the *P*-tree (dataset) by obtaining an early estimate of the possible frequent sets, which can then be counted in one pass. To do this, it needs to take a single sample or segment of data, find all the sets that are frequent in this segment, then verify the results with the rest of the database. Because there may be sets that are frequent in the entire database but not in our chosen segment, the set of candidates to be considered must be enlarged. First, when examining the sample, the support threshold is lowered to decrease the likelihood that frequent sets are missed. Then, after finding all the *locally frequent* sets in the segment being examined, this collection is augmented by adding its `negative border'. The *negative border* is the collection of all itemsets that are not frequent in the segment but all of whose subsets are. The significance of the negative border is that it defines a boundary between the frequent and non-frequent sets. If no set on the negative border is finally found to be frequent, then no sets `outside' the border can be frequent either.

The application of this method in our case can be divided into three stages:

1. First, build and store *P*-trees for each segment of data, as for the segmentation method.
2. Build a *T*-tree with its negative border for the first segment of data (for simplicity, it will be assumed that the data is randomly distributed, so that the first segment can be taken as an accurate sample of the whole).
3. Update the *T*-tree for the other segments of data.

An initial *T*-tree with negative border is built, using a lowered support threshold, from the first stored *P*-tree segment, using a slightly modified Apriori-TFP algorithm. Although this will again require repeated passes of the *P*-tree segment, if this can be contained in primary memory then it will be read once only. The negative border is easily obtained. In Apriori-TFP, a candidate set is added to the tree when its subsets are all frequent. If after counting its support, the set is found to be infrequent, it is deleted from the tree. In this variant, these sets on the tree are retained but not used to construct further (superset) candidates. Thus, these sets are not themselves frequent, but have only frequent subsets, and have no supersets in the final tree. The algorithm for building the initial *T*-tree, with negative border from the first stored *P*-tree segment, is as follows:

Algorithm 4.3: Build initial *T*-tree with its negative border

Input:

P-tree 1 (P_1), Support threshold ($\underset{\cdot}{s}$)

Output:

Initial *T*-tree with its negative border satisfying lower support threshold

Algorithm:

1) Read the first *P*-tree into memory.
2) For $K = 1, 2, \ldots$
3) Build level K in the *T*-tree.
4) Pass the *P*-tree (in memory), adding interim supports associated with individual *P*-tree nodes to the level K nodes established in (3).
5) Mark any level K *T*-tree nodes, not to be considered to build next level, that do not have an adequate level of lower support threshold.
6) Repeat steps (3), (4), and (5); until a level K is reached where no nodes are adequately supported.

For each of the other segments of data, the *P*-tree is read into primary memory, and a single traversal is performed to count the support of all the sets in the *T*-tree, adding these to the totals already stored. The final *T*-tree contains the final support-counts for all the candidate sets identified in the first section, including those on their negative border. Sometimes, unfortunately, it may happen that all necessary sets have not been evaluated. There has been a *failure* in the generation process if all frequent sets are not found in one pass. If there are no misses, then the method is guaranteed to have found all frequent sets. Misses indicate a potential failure; if there is a miss of any set, then some superset of the missed set might be frequent but not evaluated. A miss is identified by checking the negative border. If any superset can be constructed from a set, all of whose subsets are frequent, then it will be necessary to perform another pass to count its support.

Algorithm 4.4: Update the *T*-tree with its negative border

Input:

P-trees (P), Num segments (M), Support threshold ($\underset{\cdot}{s}$),

Output:

T-tree with negative border satisfying $\underset{\cdot}{s}$

Algorithm:

1) For all *P*-trees other than the first
    a. Read the *P*-tree into memory.
    b. Pass the *P*-tree (in memory), adding interim supports associated with individual *P*-tree nodes to all the *T*-tree nodes.
2) Check the negative border to confirm that there is no failure.

Note that the primary memory requirement for storage, in this method, is to contain the *P*-tree for a single segment of the data, plus the whole of the *T*-tree with its negative border. Here is an example of the method.

Example 4.2

Let us consider the same data set used in Example 4.1. Assume again that the data set is divided into three equal segments and each segment can be processed in main memory. In the first stage *P*-trees are created and stored independently for each segment as described in Example 4.1.

In the second stage the first *P*-tree is read once from secondary storage and an initial *T*-tree with negative border is built, using a lowered support threshold. Let us assume the support threshold for the dataset is 50% (i.e. 3 records, which translate to 1 record per segment). This threshold will be lowered by 1/3, to 33%, for the algorithm. Thus, the required threshold for the first segment is 0.66 records. Generation of the initial *T*-tree with negative border is shown in Figure 4.6. As usual, the top level of the *T*-tree (fig 4.6 a) is created for all 6 items. Then the *P*-tree (in memory) is passed to update the top-level nodes (fig 4.6 b). After adding supports for the *P*-tree unlike Apriori-TFP we do not prune nodes whose supports are less than the lower threshold but mark those not to be considered to build the next level. As can be seen attributes *B* and *F* are not supported (less than 0.66) and they are not considered when generating candidate set for pairs. These nodes, which are part of the negative border, are shadowed in Figure 4.6. The candidate set of pairs (level-2) is then built, as shown in fig 4.6 (c), using only the supported attributes. The *P*-tree is passed to update supports for the pairs (fig 4.6 d). It can be seen that the support of pairs *CA, DA, EA,* and *EC* exceeds the threshold thus the next level (the set of triples) can be generated from these pairs. Only one candidate triple, *ECA* is generated (fig 4.6 e) in level 3, as *ECA* is the only triple all of whose subsets (*EC, EA, CA*) are supported. The *P*-tree is passed again to update supports for triples (fig 4.6 f). And at the end of this pass, it can be seen that triplet *ECA* is supported and the final *T*-tree with *B, F, DC* and *ED* as negative border is produced.

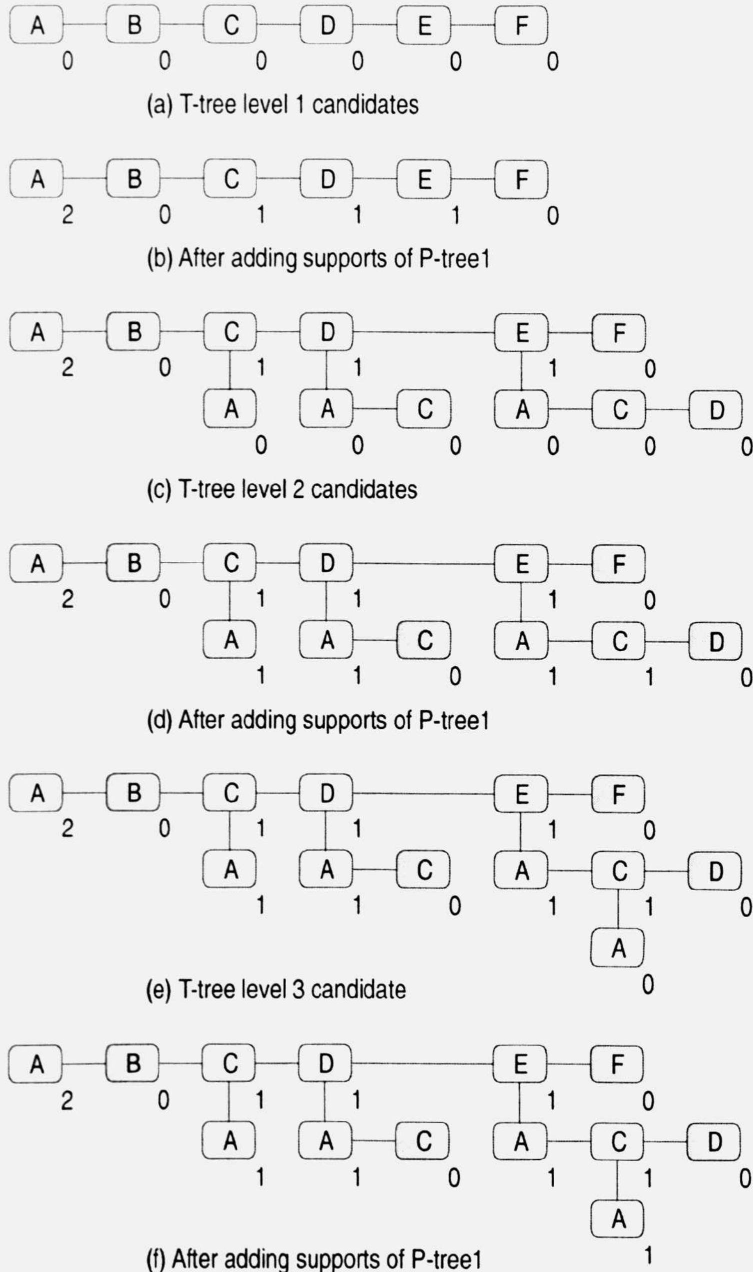


Figure 4.6: Building T-tree with Negative Border

In stage three, the other *P*-trees are read in turn into primary memory and passes are performed to update supports of all the *T*-tree nodes including those in the negative border. Figure 4.7 shows the updated *T*-tree with *P*-tree 2. After passing *P*-tree 3, the final *T*-tree (Figure 4.8) is obtained that contains the final support-counts for all the candidate sets identified in the first section, including those on their negative border. It can be seen that negative border candidates *B, F, DC* and *ED* are still not supported (less than 3) in the whole

database. So all frequent sets with their final support count have been obtained. Note, however, that the tree of Figure 4.8 is larger than that of fig 4.5 (d) because of the inclusion of the negative border.

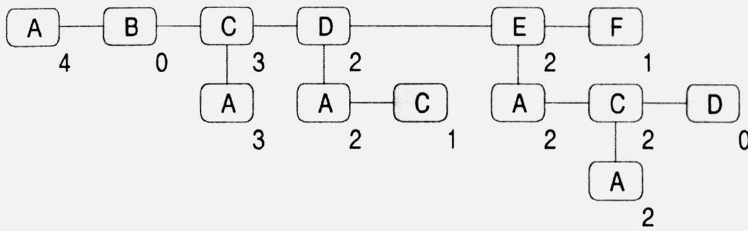


Figure 4.7: Update the initial T-tree with P-tree2

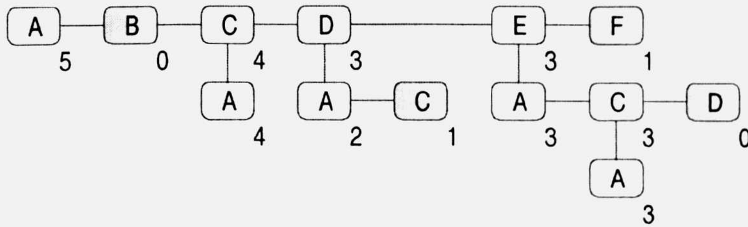


Figure 4.8: Update the initial T-tree with P-tree3

4.4 Tree Partitioning (TP)

Both methods described above require primary memory sufficient to contain the *P*-tree for a single segment of data together with the whole of the *T*-tree. Because there is no partitioning of the *T*-tree, which finally contains all the frequent sets, there may be problems in dealing with very large candidate sets. This is especially the case for the negative border method because of the larger *T*-tree to be produced. To address this problem, a method will be considered that partitions both the *P*-tree and the *T*-tree into subtrees that can be processed separately, i.e. *tree partitioning* (*TP*).

A possible way of partitioning the data is to divide the set of attributes under consideration into subsets, each of which defines a *vertical partitioning* of the data set. The

problem with this is that the sets for which support is to be counted contain attributes from several partitions so that the linkage between attributes of a long pattern will be lost while building individual *P*-trees (as well as *T*-trees). The *P*-tree structure, however, offers another form of partitioning, into subtrees that represent lexicographically-following supersets of their root node. The problem with this vertical partitioning strategy is that, although subtrees of the *T*-tree properly represent subsets of the candidate set, the subtrees of the *P*-tree do not map directly on to this. In this case, again, it is still not possible to compute the support for a set by considering only the subtree of the *P*-tree in which it is located. Although *succeeding* supersets of a set *S* are located in the subtree rooted at *S*, *predecessor* supersets are scattered throughout the preceding part of the *P*-tree, which must therefore be traversed in order to compute the support for *S*.

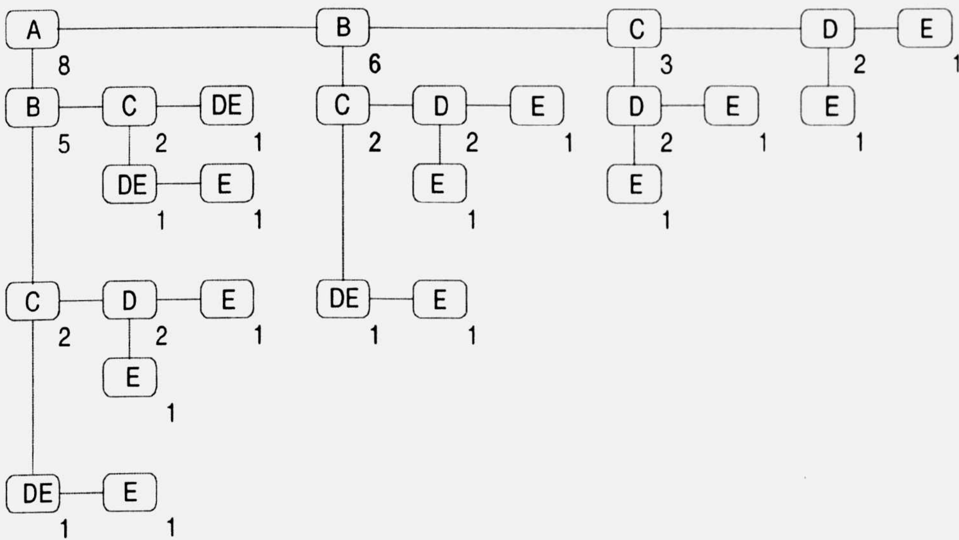


Figure 4.9: Example of a P-tree

To illustrate this, let us consider a dataset the records of which are {A,D,E}, {A,C,D,E}, {A,C,E}, {A,B,D}, {A,B,E}, {A,B,D,E}, {A,B,C,D,E}, {A,B,C,E}, {B}, {B,D}, {B,E}, {B,D,E}, {B,C,D,E}, {B,C,E}, {C,D}, {C,E}, {C,D,E}, {D}, {D,E}, and {E}. Figure 4.9 shows the *P*-tree that would be constructed. Now, consider the support for the set *BD* in the data used for Figure 4.9. In the subtree rooted at *B*, we find a partial support total for *BD*, which includes the

total for its superset *BDE*. To complete the support count for *BD*, however, we must add in the counts recorded for its preceding supersets *BCDE*, *ABD* (incorporating *ABDE*) and *ABCDE*, the latter two of which are in the subtree headed by *A*.

The problem can be overcome by a different partitioning of the *P*-tree structure where subtrees properly represent subsets of the candidate set. The *Tree Partitioning* (*TP*) method begins by dividing the ordered attribute-set into *n* subsequences. Then it proceeds to construct separate *Partition-P*-trees (*PP*-trees), which will be labelled as *PP1, PP2,...,PPn*, for these attribute-sets. For example, for the data used in Figure 4.9, the attribute-set might be divided into 3 sequences of attributes, {A,B}, {C,D} and {E}, labelled 1,2,3 respectively. A *PP-tree* is defined for each sequence, labelled *PP1, PP2* and *PP3*. The construction of these is a slight modification of the original *P*-tree construction method. The first partition-tree, *PP1*, is a proper *P*-tree that counts the partial support for the power set of {A,B}. *PP2*, however, counts all those sets that include at least a member of {C,D} in a tree that includes just these attributes and their predecessors. The third tree, *PP3*, will count all sets that include {E}. The three *PP*-trees obtained are illustrated in Figure 4.10. The *PP*-trees are, in effect, overlapping partitions of the *P*-tree of Figure 4.9, with some restructuring resulting from the omission of nodes when they are not needed.

The effect of this is that the total support for any set *S* can now be obtained from the *PP*-tree corresponding to the last attribute within *S*; for example, all the counts contributing to the support of *BD* can now be found in *PP2*. The apparent drawback is that the later trees in the sequence are of increasing size; in particular considering the total number of nodes, *PP3* is almost as large as the original *P*-tree. This can be overcome, however, by a suitable reordering of the attributes. In descending order of their frequency in the data, the ordering of the attributes in the example will be E,D,B,C,A. Using the same data as for Figures 4.9 and 4.10, *PP*-trees will be constructed using this descending ordering, for the sets of attributes {E,D}, {B,C} and {A} respectively.

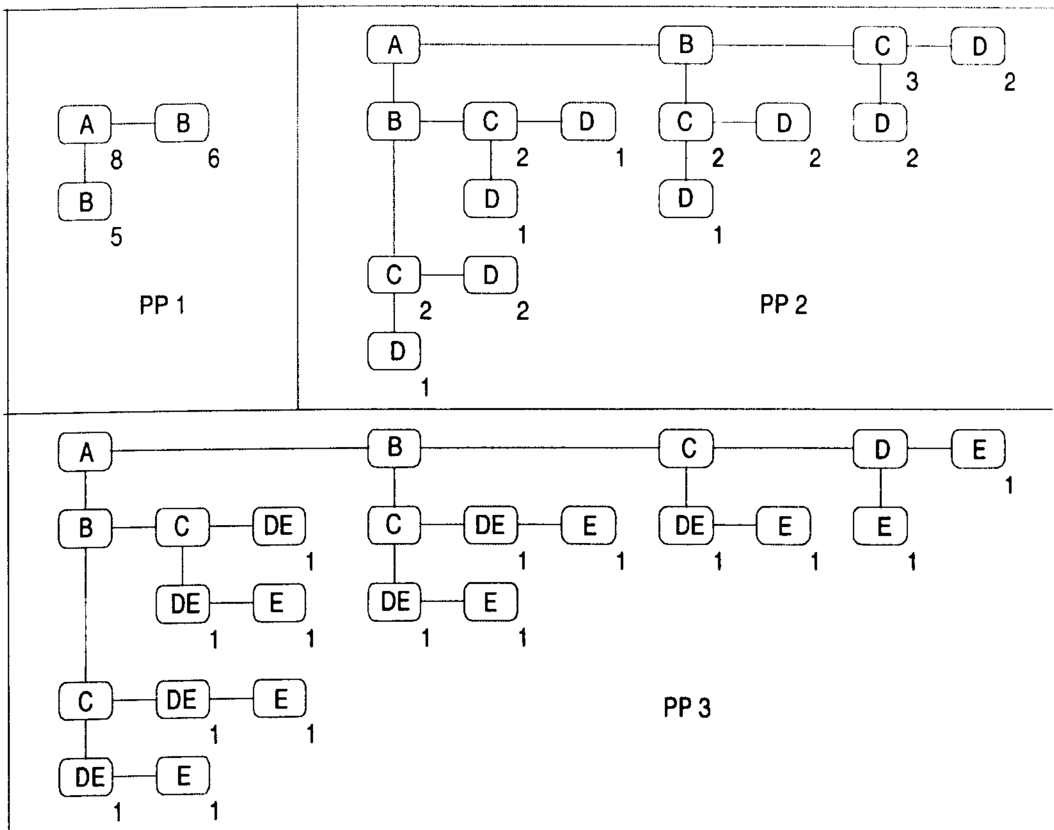


Figure 4.10: Partition-P-trees from figure 4.9

The results are shown in Figure 4.11. Now, because the less frequent attributes appear later in the sequence, the trees become successively more sparse, so that *PP3* now has only 13 nodes, compared with the 23 of *PP3* in Figure 4.10. In fact, results presented in [Coenen and Leng, 2001] demonstrate that ordering attributes in this way leads to a smaller *P*-tree and faster operation of Apriori-TFP. The additional advantage for partitioning is that the *PP*-trees become more compact and more equal in size. The total support-count for *BD* (now ordered as DB) is again to be found within *PP2*, but now requires the addition of only 2 counts (DB+EDB).

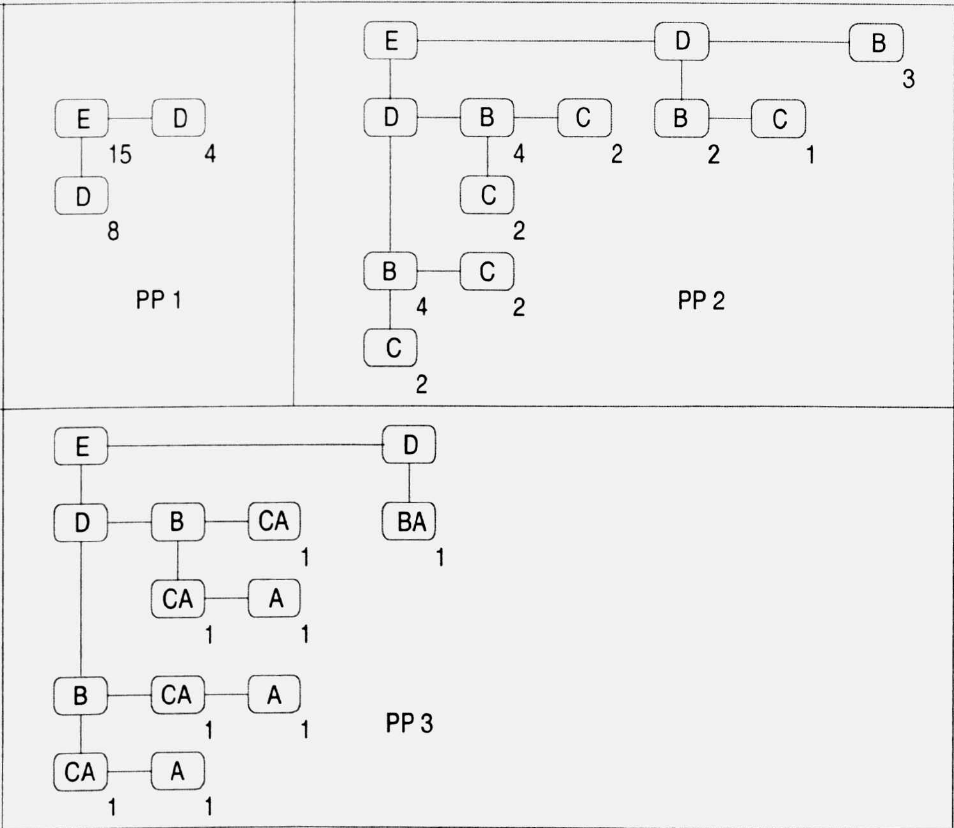


Figure 4.11: PP-trees after reordering of attributes

The form of vertical partitioning described here offers a way of dividing the source data into a number of *PP*-trees each of which may then be processed independently. A single database pass is required to build each *PP*-tree. The algorithm to build and store these *PP*-trees is as follows:

Algorithm 4.5: Vertical partitioning to create and store *PP*-trees

Input:

Database (*D*), Itemset (*S*), Num partitions (*N*)

Output:

N *PP*-trees in secondary storage.

Algorithm:

1) Divide the itemset into *N* partitions.

2) For each partition

   a) Pass the database to create a *PP*-tree in primary memory

      - For each record of the database

         - If it contains an item of the partition then include just the set of these items and their predecessors (if any) to the tree.

   b) Store the *PP*-tree into secondary storage

The stored trees are than processed independently to build a *T*-tree for each. This tree will be referred as the *Partition-T*-tree (*PT*-tree). A form of the Apriori-TFP algorithm is used to build a *PT*-tree from the *PP*-tree stored on backing store. To generate candidate sets (other than for the top level) any subsets outside the partition are assumed, if necessary, to be frequent. The algorithm to build these *T*-trees is as follows:

Algorithm 4.6: Build *PT*-trees from stored *PP*-trees

Input:

PP-trees (*P*), Num partitions (*N*), Support threshold ($\dot{s}$)

Output:

N *PT*-trees satisfying $\dot{s}$

Algorithm:

1) For each partition of the attribute set

   a. Read a *PP*-tree.

   b. For *K* = 1, 2, ...

   c. Build level *K* in the *T*-tree with the attribute-set of the partition and assuming, where necessary, that any (*K-1*)-itemsets outside the partition are frequent.

   d. Pass the *PP*-tree (in memory), adding interim supports associated with individual tree nodes to the level *K* nodes established in (c).

e. Remove any level *K* *T*-tree nodes that do not have an adequate level of support *s*.

f. Repeat steps (c), (d), and (e); until a level *K* is reached where no nodes are adequately supported.

This form of tree partitioning offers a way of dividing the source data into a number of *PP*-trees each of which is then processed independently and the trees stored for each partition need be read once only. The method may be summarized thus:

1. Choose an appropriate partitioning of the attribute-set into sequences 1, 2,...n.
2. Perform *n* passes of the database to build *Partition-P*-trees *PP1, PP2...PPn.*
3. Read *PP1* into memory, and build a *T*-tree to count the total support for all frequent sets formed from members of set 1 only.
4. Read *PP2* into memory, and build a *T*-tree counting the support for frequent sets formed from members of set 2 with its predecessors.
5. Repeat step 4 for *PP3, PP4*, etc.

The method, therefore, creates *T*-trees, which finally contain all the frequent itemsets for a partition of the attribute-set. Thus, the candidate sets are distributed into several *T*-trees. The primary memory requirement for storage, thus, is to contain the *PP*-tree for a single partition of the data, plus the corresponding *PT*-tree. Here is an example of the method.

Example 4.3

Consider the same data set as used in Example 4.1 (fig 4.12 a). The set of attributes {A,B,C,D,E,F} is divided into three subsets {A,B}, {C,D} and {E,F} as shown in fig 4.12 (b). In the first partition, subsets of records containing at least one item from the subset {A,B} are included. In the second partition all those subsets of records that contain at least one item

from the subset {C,D} with its predecessors are included. Similarly records that contain at least one item from the subset {E,F} are included in the last partition. A record comprising (say) the first record, *ACE*, would thus be included in subsets {A,B} as *A*, {C,D} as *AC* and {E,F} as *ACE*.

```
                   A B|   C D|    E F
                   -------------------------
A C E              A   | A C   | A C E
A D                A   | A D   |   -
A C E F            A   | A C   | A C E F
A C D              A   | A C D |   -
D                  -   | D     |   -
A C E              A   | A C   | A C E
```

(a) Dataset (b) Partitioned Dataset

Figure 4.12: Original and Vertically Partitioned Dataset

Stage 1:

In stage 1 *PP*-trees, as shown in Figure 4.13, are created and stored for one partition after another. After storing a *PP*-tree into secondary storage, the memory is cleaned for reuse. The first partition-tree, PP1 (fig 4.13 a), is a proper *P*-tree that counts the partial support for the power set of {A,B}. PP2 (fig 4.13 b), counts all those sets that include a member of {C,D} in a tree that includes just these items and their predecessors. The third tree, PP3 (fig 4.13 c), counts all sets that include any member of {E,F}.

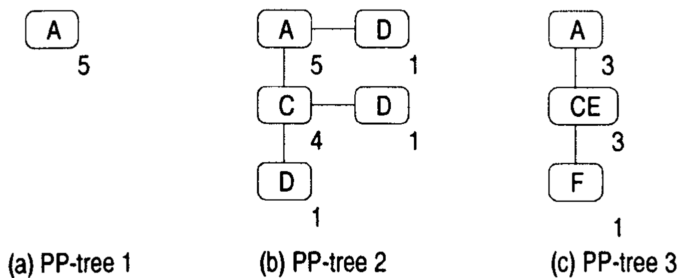


Figure 4.13: PP-trees from vertically partitionrd dataset

Stage 2:

In stage 2 *PP*-trees are read in turn and used to build *T*-trees for the corresponding partitions. Let us see the level wise building of *Partition T*-trees for a threshold of 50% (3 records). *PP*-tree 1 (fig 4.13 a) has only one attribute so an identical *T*-tree of attribute *A* with support 5 is

built for the first partition. Figure 4.14 shows the building of the *T*-tree for subset {C,D}. Only attributes *C* and *D* are considered to build the top level of tree (fig 4.14 a). After passing the *PP*-tree 2 (in memory) to add interim supports for singles, as can be seen in fig 4.14 (b) all the candidate nodes are supported. These supported top-level attributes with their predecessors (*A, B*) are used to generate the candidate set for pairs as shown in fig 4.14 (c). Note that here it is assumed that *A* and *B* both reach the required support threshold (although in fact this is not the case for *B*). After passing the *PP*-tree 2 to add interim supports for pairs, it can be seen only pair *CA* is supported (fig 4.14 d) and all other pairs are pruned to get the final tree (fig 4.14 e) for the second partition.

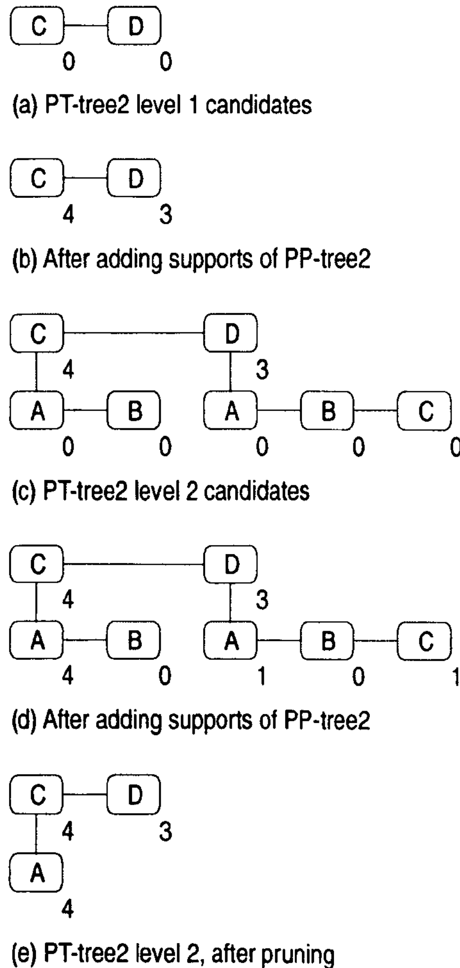
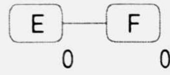
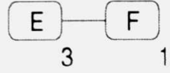


Figure 4.14: Building PT-tree 2 from PP-tree 2

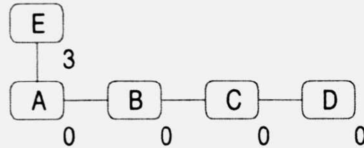


(a) PT-tree3 level 1 candidates

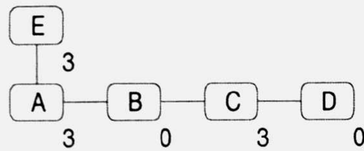


(b) After adding supports of PP-tree3

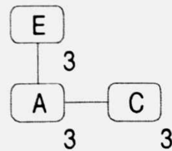
(c) PT-tree3 level 1, after pruning



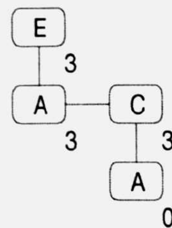
(d) PT-tree3 level 2 candidates



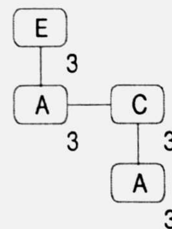
(e) After adding supports of PP-tree3



(f) PT-tree3 level 2, after pruning



(g) PT-tree3 level 3 candidate



(h) After adding supports of PP-tree3

Figure 4.15: Building PT-tree 3 from PP-tree 3

Figure 4.15 shows the building of the *T*-tree for subset {E,F}. The top level of the tree (fig 4.15 a) is built for attributes *E* and *F*. After passing the *PP*-tree 3 (in memory) to add interim supports, it can be seen that only candidate node *E* is supported (fig 4.15 b), the other node is pruned (fig 4.15 c). The supported top-level attribute *E*, with its predecessors, is used to generate candidate pairs as shown in fig 4.15 (d). After passing the *PP*-tree 3 to add interim supports for pairs, it can be seen from fig 4.15 (e) that pairs *EA* and *EC* are supported so that the other pairs are pruned (fig 4.15 f). These pairs are use to generate candidate triples. Only one candidate triple, *ECA* is generated (fig 4.15 g), as *ECA* is the only triple all of whose subsets within the partition (*EC* and *EA*) are supported - the predecessor subset *CA* which is not contained in the current partition is assumed to be frequent. At the end of another pass of *PP*-tree 3, to add interim supports for the triples, the triple *ECA* is supported (fig 4.15 h) and the final *T*-tree for the last partition is produced (there is no way to grow the tree further).

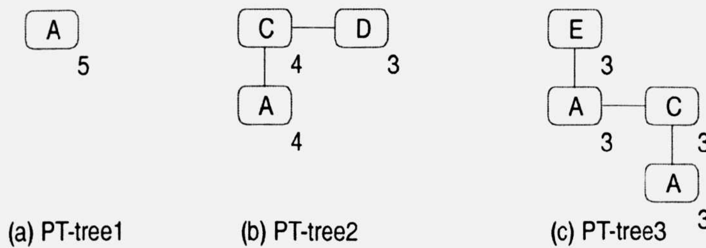


Figure 4.16: Final PT-trees

Figure 4.16 shows the three final *PT*-trees that have been constructed. Notice that the nodes and their counts are exactly the same as for the *T*-tree of fig 4.5 (d). The tree partitioning method has produced the same result as the original *DP* method, but has reduced the primary memory requirement by partitioning the *T*-tree.

4.5 Data and Tree Partitioning (DTP)

The tree partitioning, described above, reduces the maximum primary memory requirement, as only a *PP*-tree together with the subtree of the *T*-tree corresponding to that partition needs to be contained in memory. Further advantages of the method are that more of the counting is done by the relatively efficient procedures used when constructing the *P*-tree, and *T*-tree traversal works are reduced because of the use of smaller *T*-tree partitions. The chief drawback, however, is that repeated passes of the database are now required to construct the *PP*-trees. With a sufficiently large data set it will of course not be possible to construct all the *PP*-trees within primary memory in a single database pass. To overcome this, the *TP* approach can be combined with a (horizontal) segmentation of the original data, as described in *DP,* into segments small enough to allow the corresponding *PP*-trees to be contained in primary store.

In this case (*DTP*), *PP*-trees will be built first, in a single database pass, for each segment of the database (choosing a segment size which allows this to be done entirely in primary memory). Then, the stored *PP*-trees are processed in the order of the vertical partitioning, to construct the complete *T*-trees for partition 1, 2 ...in turn. The overall method is as follows. For clarity, the term *segment* will be used to refer to the horizontal division of the data into sets of records, and *partition* will be used to refer to the vertical division into sets of attributes and the corresponding tree structures:

1. Choose an appropriate partitioning of the items into *n* sequences 1, 2, 3,..etc.
2. Divide the source data into *m* segments.
3. For each segment of data, construct *n* *PP*-trees in primary memory, storing each tree finally to disk. This construction phase involves just one pass of the source data.
4. For partition 1, read the *PP1* trees for all segments into memory, and apply the Apriori-TFP algorithm to build a *T*-tree that finds the final frequent sets in the

partition. This stage requires the *PP1* trees for each segment of data to be read once only. The *T*-tree remains in memory throughout, finally being stored to disk.

5. Repeat step 4 for partitions 2, 3, ..*n*.

The method offers two speed advantages over simple horizontal segmentation. First, the number of disk passes has now been effectively reduced to 2: one (step 3) to construct the *PP*-trees, and a second pass (of the stored trees) to complete the counting (steps 4 and 5) since the *PP*-trees stored for each segment need be read once only. The second advantage is that now it needs, at each stage, to deal with smaller tree structures, leading to faster traversal and counting. In this method the maximum memory requirement is for the *PP*-trees for one partition, plus the corresponding *T*-tree partition. Below is an example of the method.

Example 4.4: Again consider the same data set used previously (fig 4.17 a). Here the data set is first divided into three equal segments and then each segment is further divided into three subsets {A,B}, {C,D} and {E,F} as shown in fig 4.17 (b).

```
                A B|   C D|    E F
                ---------------------
A C E           A  | A C  | A C E
A D             A  | A D  |  -
A C E F         ---------------------
A C D           A  | A C  | A C E F
D               A  | A C D|  -
A C E           ---------------------
                -  | D    |  -
                A  | A C  | A C E
```

(a) Dataset (b) HSVP dataset

Figure 4.17: Original and HSVP dataset

In stage 1 each segment of data set is taken in turn to create and store all the *PP*-trees for the segment as shown in Figure 4.18. After storing the trees into secondary storage,

the memory is cleaned for reuse. *PP*-trees for partition 1, 2, and 3 of segment 1 are shown in fig 4.18 (a), (b) and (c) respectively. Similarly *PP*-trees for the second segment are shown in fig 4.18 (d), (e) and (f) and third segment are in fig 4.18 (g), (h) and (i).

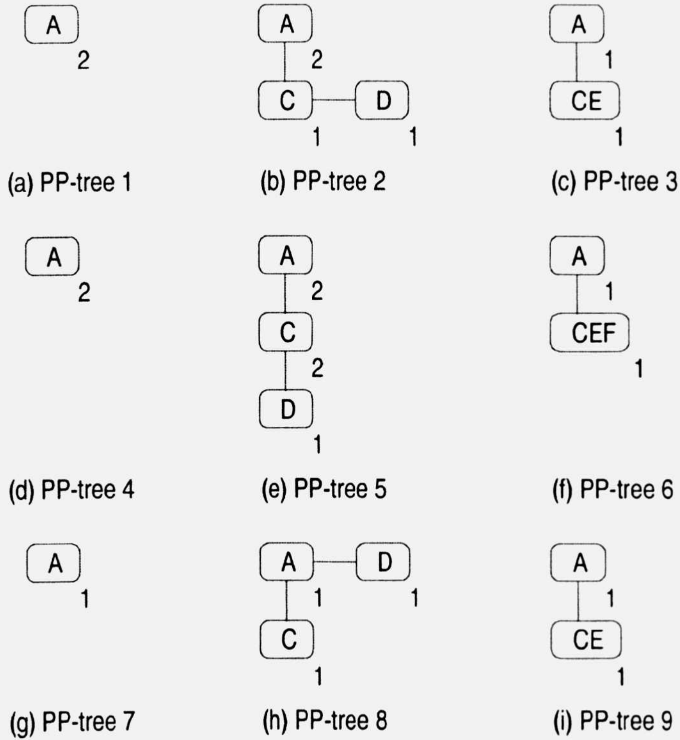


Figure 4.18: PP-trees from HSVP dataset

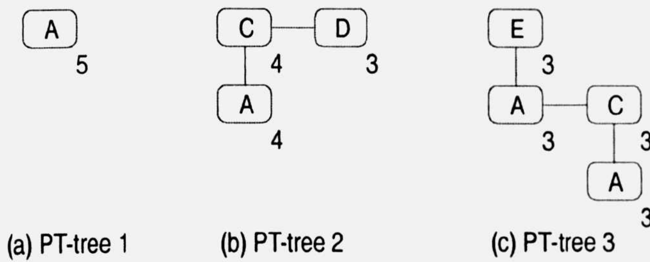


Figure 4.19: PT-trees from PP-trees

In stage 2, *PP*-trees are read for a partition and the *T*-tree for the corresponding partition is built. The algorithm needs to read these *PP*-trees from backing store only once instead of reading for each level. Figure 4.19 shows the *PT*-trees for the support threshold of

50%. *PT*-tree 1 (fig 4.19 a) is built using the first partition *P*-trees shown in fig 4.18 (a), (d), and (g). *PT*-tree 2 (fig 4.19 b) is built using the second partition *P*-trees in fig 4.18 (b), (e), and (h). And *PT*-tree 3 (fig 4.19 c) is built using the last partition *P*-trees in fig 4.18 (c), (f), and (i).

4.6 Summary

Different strategies for partitioning the data and data-structure have been considered in cases when it will be impossible to contain all the data required in primary memory for implementing Apriori-TFP. Different methods of partitioning to limit the total primary memory requirement are examined, including that required both for the source data and for the candidate sets. The methods are: (1) *DP* (Data Partitioning), (2) *NB* (Negative Border), (3) *TP* (Tree Partitioning), and (4) *DTP* (Data and Tree Partitioning).

In the *DP* and *NB* methods, it is necessary to divide the database horizontally and read the entire database only once to create *P*-trees. These methods produce a single *T*-tree. The *DP* method, however, requires repeated passes of the disk-resident data, and the *NB* method involves an enlarged *T*-tee. The *TP* method uses a form of vertical partitioning to build partitioned *P*-trees that can be processed independently to build *T*-trees. The *TP* method leads to a partitioning that is essentially similar to that obtained by the construction of *conditional* databases described in [Han *et al.*, 2000] and [Pei *et al.*, 2000], and the COFI-trees proposed in [El-Hajj and Zaiane, 2003]. The latter method also creates subtrees that can be processed independently, but requires an initial construction of an *FP*-tree that must be retained in primary memory for efficient processing. The partitioning strategy proposed in [Pei *et al.*, 2000] for dealing with an *FP*-tree too large for primary storage, would first construct the *a-conditional* database corresponding to the *PP*-tree, and after building the *FP*-tree for this, would copy relevant transactions (e.g. *EDBCA*), into the next (*c-conditional*) database, as

EDBC. The *DTP* method, however, avoids this multiple copying by constructing all the trees in memory in a single database pass, during which partial support totals are also counted.

Chapter 5

Experiments and Results

5.1 Introduction

To investigate the performance of different methods (strategies) described in the previous chapter, a number of experiments for extracting association rules from databases in cases where the data is too large to be contained in main memory have been carried out. These experiments for finding the performance characteristics in various circumstances are described in this chapter. Synthetic data sets used in the experiments were constructed using the QUEST generator described in [Agrawal and Srikant, 1994]. This uses parameters: *T*, which defines the average number of attributes found in a set; and *I*, the average size of the maximal supported set. Higher values of *T* and *I* in relation to the number of items (attributes) *N* correspond to a more densely-populated database. All the programs were written in standard C++ and run under the Linux operating system. Time and memory were measured using the C++ standard library functions and operators. The experiments were performed on an AMD Athlon workstation with a clock rate of 1.3 GHz, 256 Kb of cache, and 512 Mb of RAM. The data was stored on an NFS server (1 Gb filestore). Detailed results of the experiments, some of which are already presented in our papers published in [Ahmed *et al.*, 2003] and [Ahmed *et al.*, 2004], are presented in this chapter.

The organisation of the chapter is as follows:

(1) Section 5.2 (Size of Database) examines scalability of the methods for increasing size of databases.

(2) The effect of applying a different degree of 'horizontal' segmentation, which is used to evaluate Data Partitioning (*DP*), Negative Border (*NB*), and Data and Tree Partitioning (*DTP*) methods, is examined in section 5.3 - Degree of Segmentation.

(3) Next the methods are examined for databases of increasing number of attributes (section 5.4 - Number of Attributes).

(4) Different degree of 'vertical' partitioning, used in Tree Partitioning (*TP*) and Data and Tree Partitioning (*DTP*) methods, are investigated in section 5.5 - Degree of Partitioning. In most of these experiments, the partitioning divides the attribute-set equally.

(5) The performance of the methods is then compared for increasing density of databases in section 5.6 - Density of Database.

(6) Finally, performance comparisons of the methods for different support thresholds are presented in section 5.7 - Performance Comparison.

(7) Section 5.8 summarises the results of the experiments.

5.2 Size of Database

First it is essential to establish that the methods scale acceptably with increasing size of databases; that is, the different partitioning strategies successfully constrain the maximum requirement for primary memory, without leading to unacceptable execution times. For this purpose data sets were generated with parameters T10.I5.N500: i.e. 500 items, with an average record-length of 10 items and an expected maximal frequent pattern size of 5. The data sets were increased in size by 50,000 records. For the (horizontal) segmentation used in the *DP* (or *NB*) and *DTP* methods, the dataset was divided into segments of 50,000 records. For the (vertical) partitioning used in the *TP* and *DTP* methods, the dataset was divided into 500 partitions, i.e. a partition *P*-tree for each item. In fact, it will be apparent from the experiments that increasing the degree of (vertical) partitioning always reduces the primary

memory requirement (as would be expected). The experiments are set up to measure (1) time to construct P/PP-trees for increasing size of databases, (2) memory requirements for largest P-tree/ Partition (T10.I5.N500), (3) time to find frequent sets for increasing size of databases (0.01% support), and (4) memory requirements for T-tree/ largest PT-tree (T10.I5.N500).

The results for these experiments, and for all other experiments described in this chapter, are tabulated in Appendix A. In this chapter selected results will be presented in graphical form to illustrate the significant features.

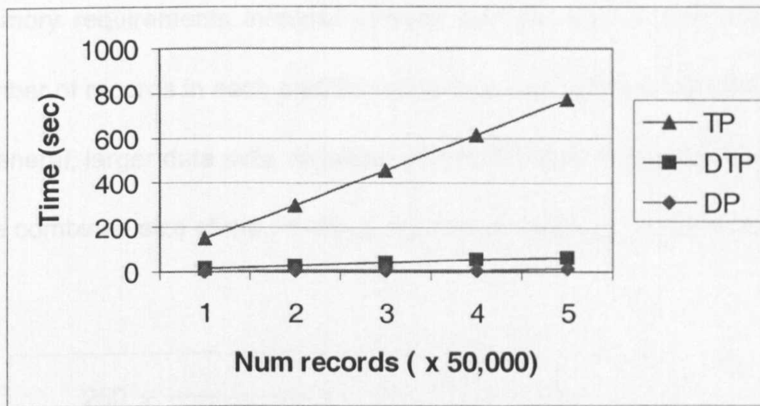


Figure 5.1: Time to construct P/PP-trees for increasing size of databases

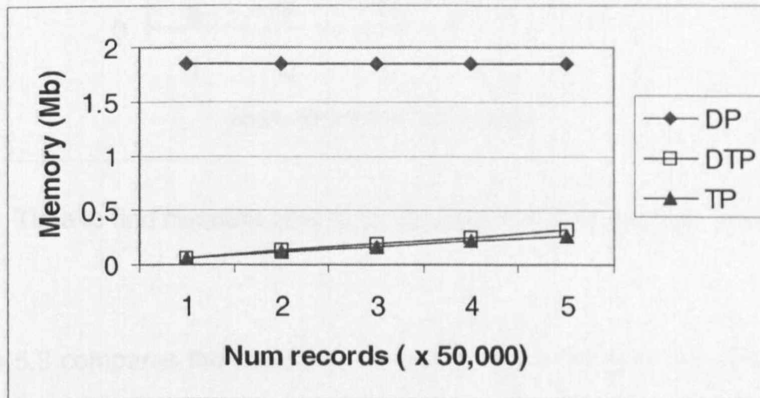


Figure 5.2: Memory requirements for largest P-tree/ Partition (T10.I5.N500)

Figure 5.1 shows the time in seconds to construct and store *P*-trees in method *DP* (or *NB*) and partition *P*-trees in methods *TP* and *DTP* for the databases of increasing size; i.e.

1,2,3,4 and 5 segments. It can be seen that the time to construct and store these trees increases linearly with the size of databases. Figure 5.2 shows the maximum memory requirements, in megabytes, for the methods. This will be the size of the largest *P*-tree (in method *DP* or *NB*), or the largest vertical partition in methods *TP* and *DTP*. As mentioned in the previous chapter, the vertical partition contains a single *PP*-tree in method *TP*, but the number of *PP*-trees increases, in method *DTP*, with the number of segments. As can be seen, the memory requirement for the method *DP* (or *NB*) is constant for the databases. This is what we would expect in the case of a database of homogeneous density. For methods *TP* and *DTP*, memory requirements increase linearly with the size of the database since the increased number of records in each partition adds new sets with a corresponding increase in tree size. In general, larger data sets, requiring greater horizontal segmentation, lead to some increase in the combined size of the *PP*-trees, but this is relatively slight for the *DTP* method.

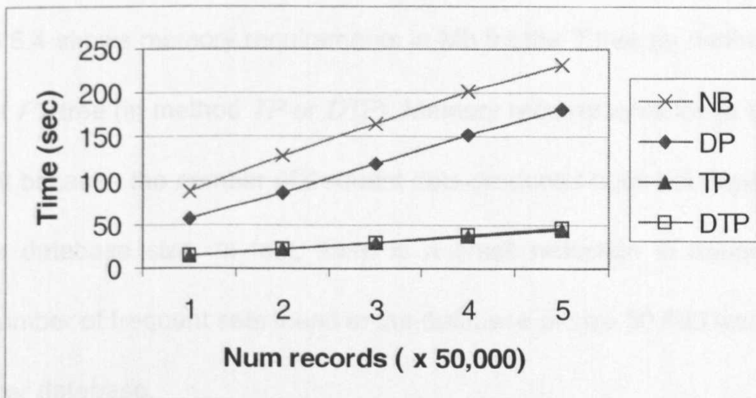


Figure 5.3: Time to find frequent sets for increasing size of databases (0.01% support)

Figure 5.3 compares the performance of the methods (times in seconds) in using the stored trees to compute final support totals or frequent sets, for the support threshold of 0.01%. For the 'Negative Border' method, the actual support thresholds were reduced in each case to 2/3 of the chosen Figure when constructing the initial *T*-tree with the aim of avoiding failure to find all frequent sets in one pass. To find frequent sets a *T*-tree was built in method

DP (or *NB*) and partition *T*-trees were built in methods *TP* and *DTP*. It can be seen that the time to build these trees increases linearly with the size of databases. The reason for this is that the number of *P*-trees or *PP*-trees required to be read from secondary storage is higher for larger databases.

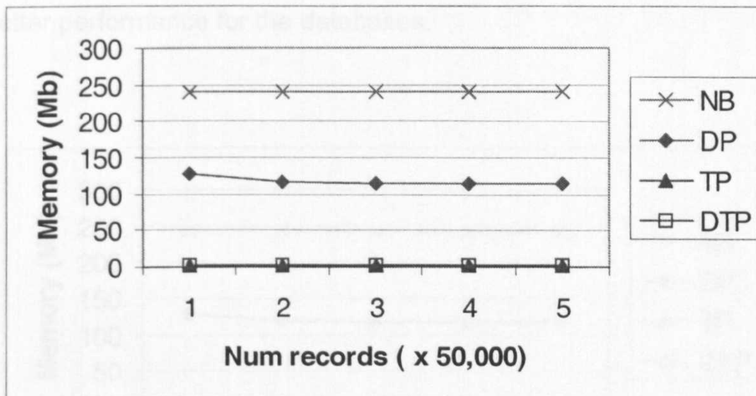


Figure 5.4: Memory requirements for T-tree/ largest PT-tree (T10.I5.N500)

Figure 5.4 shows memory requirements in Mb for the *T*-tree (in methods *DP* and *NB*) and the largest *PT*-tree (in method *TP* or *DTP*). Memory requirements for all the methods are nearly constant because the number of frequent sets produced does not depend to any great degree on the database size. In fact, there is a small reduction in memory requirement because the number of frequent sets found in the database of size 50,000 was slightly greater than in the larger database.

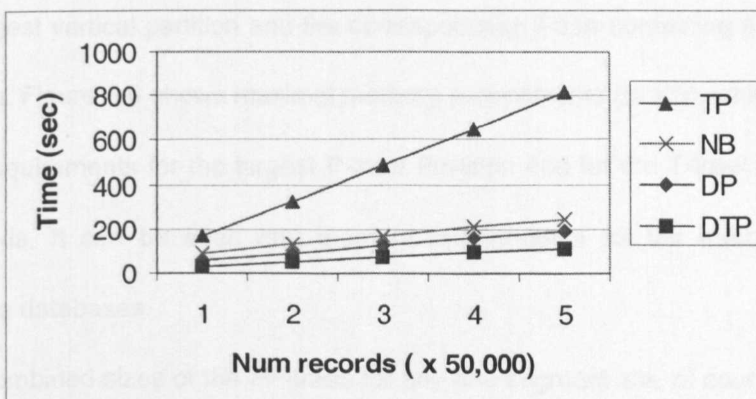


Figure 5.5: Execution times for T10.I5.N500 (0.01% support)

Figure 5.5 shows the overall time, which is the time to construct P/PP-trees plus time to find frequent sets, with a support threshold of 0.01%, for datasets of increasing size. The times illustrated include both the time to construct the *P*-trees (*DP* or *NB* method) or *PP*-trees (*TP* and *DTP*) and to execute the Apriori-TFP algorithm. It can be seen that the performance of all methods scales linearly with the size of the dataset, and that the *DTP* method offers substantially better performance for the databases.

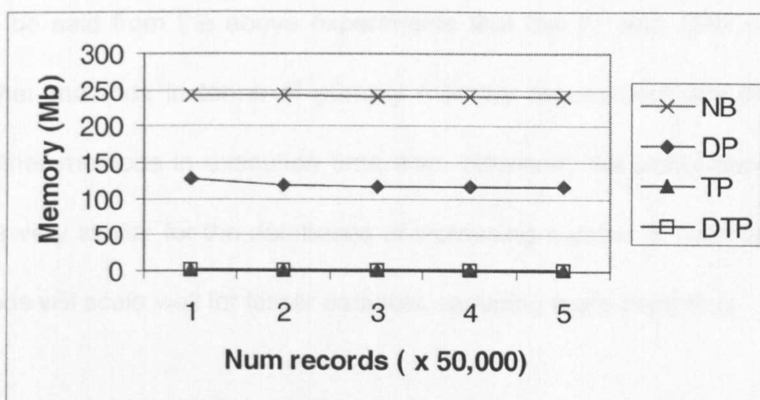


Figure 5.6: Memory requirements for T10.I5.N500

Importantly, this performance is achieved within conservative requirements for primary storage. The maximum memory requirement for the *DP* (as well as *NB*) method is the sum of the memory requirement for the largest *P*-tree segment and that for the whole of the *T*-tree. For the *TP* and *DTP* methods, the maximum memory requirement is defined by the size of the largest vertical partition and the corresponding *T*-tree containing the frequent sets in that partition. Figure 5.6 shows maximal memory requirements (in Mb), which is the sum of the memory requirements for the largest P-tree/ Partition and for the T-tree/ largest PT-tree, for the methods. It can be seen that memory requirements for the methods are nearly constant for the databases.

The combined sizes of the *PP*-trees for any one segment are, of course, greater than the size of a corresponding *P*-tree. While constructing the P/PP-trees, the sum of the sizes of

the 500 *PP*-trees for any one segment is about 15.56 Mb (varying little between segments), compared to a *P*-tree size of about 1.85 Mb (Figure 5.2). This size of the *PP*-trees was not the dominant store requirement in the illustrated case (Figure 5.6), but in other cases could be a constraint during the construction of the *PP*-trees. If this is so, the problem can easily be overcome by imposing a greater degree of horizontal segmentation. It will be apparent that increasing the number of segments has little effect on execution times, while reducing memory requirements during the *PP*-tree construction.

It can be said from the above experiments that the *TP* and *DTP* methods strongly outperform other methods in terms of primary memory requirement and the *DTP* method outperforms other methods in execution time also. However, the performance difference of the methods is very similar for the databases of increasing number of records. This suggests that the methods will scale well for larger datasets, requiring more segments.

5.3 Degree of Segmentation

The effect of horizontal segmentation, used in methods *DP*, *NB* and *DTP*, was further examined on a database divided into an increasing number of segments. The data set was generated with parameters *T* =10, *I* =5, *N* =500 and *D* =50,000. In this case a vertical partitioning of 50 items per partition (10 partitions in all) was imposed for the *DTP* method, while varying the number of segments for the methods. The dataset was investigated for divisions into 1, 2, 5, 10 and 50 equal segments; i.e. 50,000, 25,000, 10,000, 5,000 and 1,000 records/segment respectively. The small database of 50,000 records, in this and the remaining experiments, was chosen for convenience, but in all cases a requirement has been imposed that only one segment can be retained in primary store at one time. The experiments was designed to measure: (1) time to construct P/PP-trees for increasing number of segments, (2) memory requirements of largest P-tree/ Partition (T10.I5.N500.D50000), (3)

time to find frequent sets for increasing number of segments (0.01% support), and (4) memory requirements for T-tree/ largest PT-tree (T10.I5.N500.D50000).

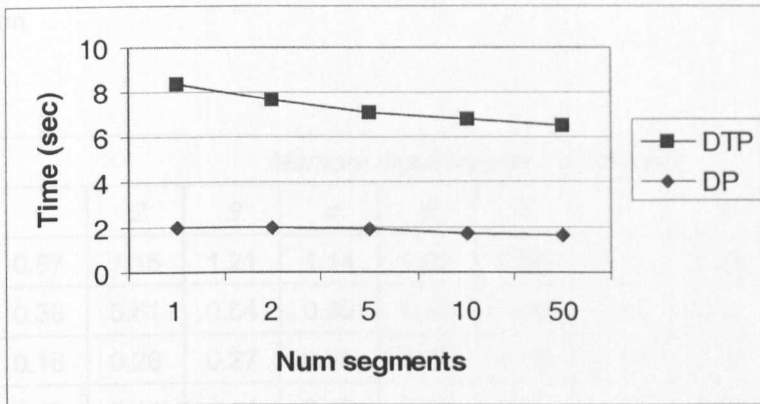


Figure 5.7: Time to construct P/PP-trees for increasing number of segments

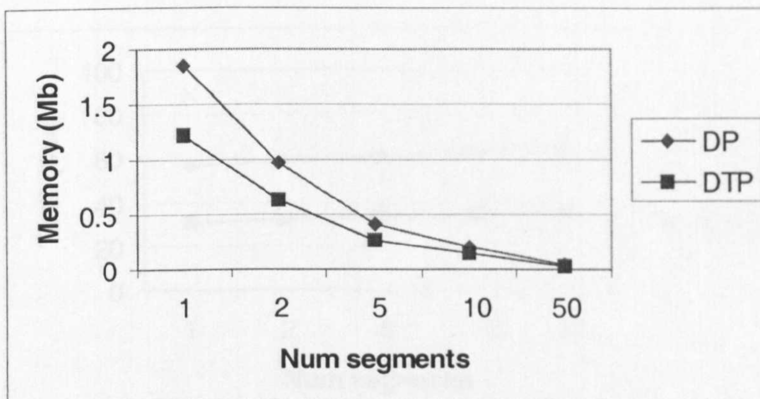


Figure 5.8: Memory requirements of largest P-tree/ Partition (T10.I5.N500.D50000)

Figure 5.7 shows the time in seconds to construct and store *P*-trees in method *DP* (or *NB*) and *PP*-trees in method *DTP* for increasing numbers of segments. It should be noted that the time to build these trees decreases slightly as the number of segments is increased. The reason for this is that, although the number of trees increases with the number of segments, the size of each tree becomes successively less, reducing tree-traversal time as each is constructed. It can be seen in Figure 5.8, which shows the memory requirement for the largest *P*-tree or partition, that this decreases significantly for increasing number of segments.

The total memory requirement to contain all the *PP*-trees for any one segment also decreases, as one would expect, from 7.8 Mb (1 segment) to a maximum of 0.22 Mb (50 segments). Table 5.1 summarize the memory requirement in Mb for a segment during *PP*-tree construction.

	Memory requirements for PP-trees									
Partition	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*
1 segment	0.67	1.15	1.21	1.14	1.02	0.88	0.73	0.57	0.34	0.09
2 segments	0.38	0.61	0.64	0.59	0.53	0.46	0.38	0.29	0.17	0.05
5 segments	0.18	0.26	0.27	0.25	0.22	0.19	0.16	0.12	0.07	0.02
10 segments	0.10	0.14	0.14	0.13	0.11	0.10	0.08	0.06	0.04	0.01
50 segments	0.02	0.03	0.03	0.03	0.03	0.02	0.02	0.02	0.01	0.01

Table 5.1: Memory requirements for different partitions of a segment.

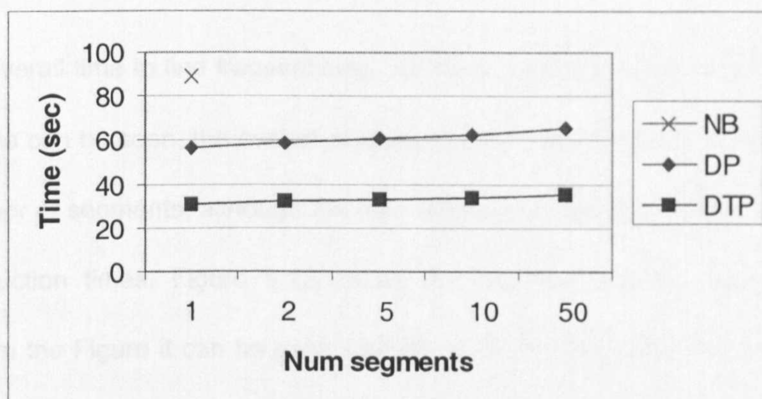


Figure 5.9: Time to find frequent sets for increasing number of segments (0.01% support)

Figure 5.9 shows the time in seconds to compute final support totals or find frequent sets for a support threshold of 0.01%. As can be seen, this time increases slightly with the number of segments. This is to be expected since a higher number of segments requires more trees to be read from secondary storage. Figure 5.10 shows the memory requirement for the *T*-tree or the largest *PT*-tree. It can be seen that the memory requirement for methods *DP* and *DTP* are constant, which is expected; but not for method *NB*. In method *NB*, trees could not be built for smaller segments of the dataset because of memory overflow. Recall

that in this method, with the aim of avoiding failure to find all frequent sets in one pass, the actual support thresholds is reduced to 2/3 of the chosen Figure when constructing the initial *T*-tree and thus the tree is likely to become larger as segmentation is increased.

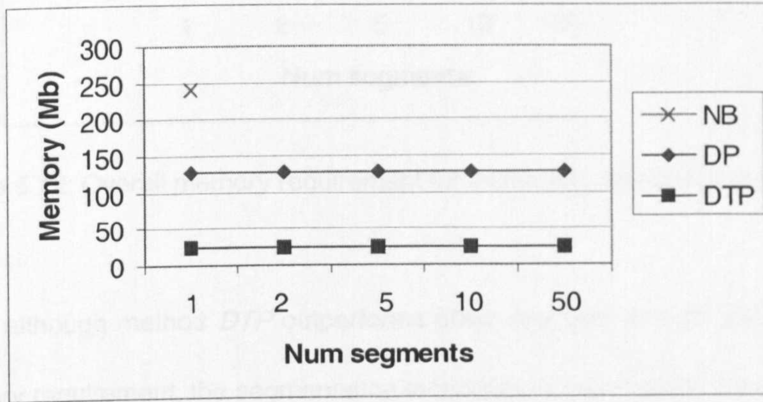


Figure 5.10: Memory requirements for T-tree/ largest PT-tree (T10.I5.N500.D50000)

The overall time to find frequent sets, for the support threshold of 0.01%, is shown in Figure 5.11. As can be seen, the overall execution time to find frequent sets increase slightly with the number of segments, although the rate of increase is reduced due to the decreasing *P*-tree construction times. Figure 5.12 shows the maximal memory requirement for the methods. From the Figure it can be seen that the memory requirement for methods *DP* and *DTP* is almost constant since the memory requirement for the largest *P*-tree or partition is negligible in comparison with the memory requirement of the *T*-tree or the *PT*-tree.

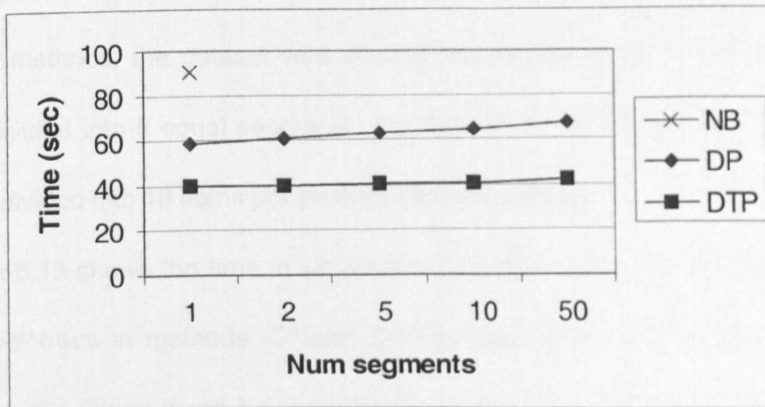


Figure 5.11: Effect of increasing segmentation on overall execution time (0.01% support)

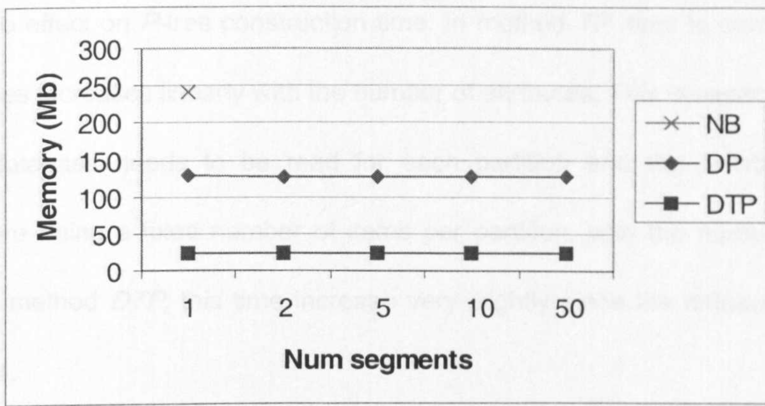


Figure 5.12: Overall memory requirement for increasing number of segments

Thus, although method *DTP* outperforms other methods in both execution time and primary memory requirement, the segmentation technique, in most cases, has very little effect on their performance. This suggests that the methods will scale well for larger datasets, requiring more partitions.

5.4 Number of Attributes

In this section a number of experiments are described to establish whether the methods scale acceptably with increasing number of attributes. For this purpose data sets were generated with parameters T10.I5.D50000: i.e. 50,000 records, with an average record-length of 10 items and an expected maximal frequent pattern size of 5. The number of attributes used in the data sets was increased in steps of 500. For the horizontal segmentation used in the *DP*, *NB*, and *DTP* methods, the dataset was divided into segments of 10,000 records, i.e. the dataset was divided into 5 equal segments. For the *TP* and *DTP* methods, the dataset was also vertically divided into 10 items per partition (items/partition).

Figure 5.13 shows the time in seconds to construct and store *P*-trees in method *DP* (or *NB*) and *PP*-trees in methods *TP* and *DTP* for databases with increasing numbers of attributes. From the Figure it can be seen that for method *DP*, times to construct *P*-trees are nearly constant. So it can be said that, in this case the number of attributes in the database

has virtually no effect on *P*-tree construction time. In method *TP*, time to construct and store partition *P*- trees increases linearly with the number of attributes. This is expected since in this method the database needs to be read for each partition and the number of partitions increases, there being a fixed number of items per partition, with the number of attributes. Conversely in method *DTP*, this time increase very slightly since the database needs to be read only once.

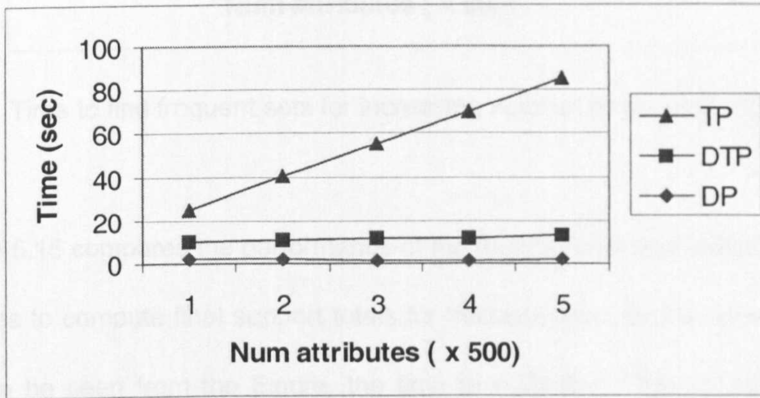


Figure 5.13: Time to construct P/PP-trees for increasing number of attributes

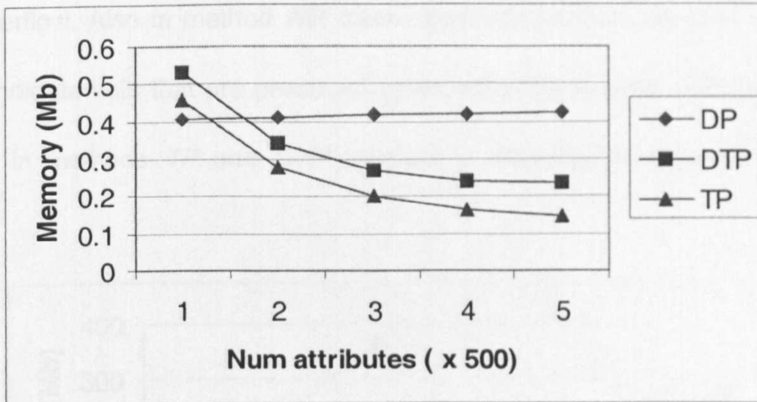


Figure 5.14: Memory requirements for largest P-tree/ Partition (T10.I5.D50000)

Figure 5.14 shows the memory requirement, in megabytes, for the largest *P*-tree in method *DP* (or *NB*), and for the largest vertical partition in methods *TP* and *DTP*. Memory requirement for method *DP* increases very slightly with the number of attributes. For methods

TP and *DTP* it decreases with increasing number of attributes, since for sparser databases each vertical partition contains a smaller number of attributes resulting in smaller trees.

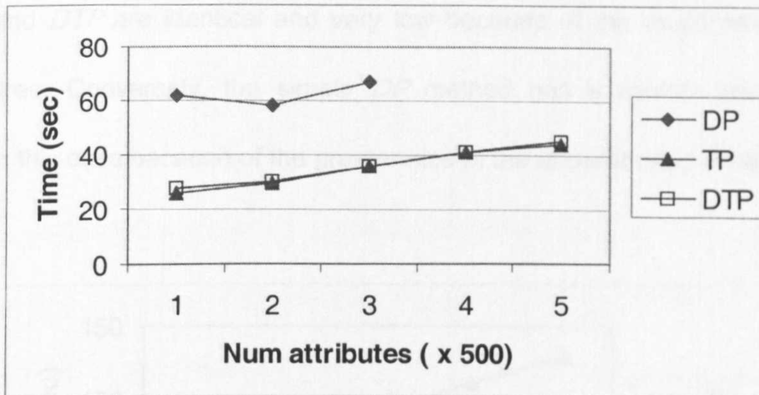


Figure 5.15: Time to find frequent sets for increasing number of attributes (0.01% support)

Figure 5.15 compares the performance of the methods (times in seconds) when using the stored trees to compute final support totals for frequent sets, for the support threshold of 0.01%. As can be seen from the Figure, the time to build the *T*-tree in method *DP* varies slightly with the number of attributes. This is a result of the variation in the number of frequent sets found in each case. For higher numbers of attributes the tree could not be built because of memory overflow. Also in method *NB*, trees could not be built because of the excessive size of the candidate sets that are produced by lowering the support threshold. The time for building trees in methods *TP* and *DTP* appears to increase linearly with the number of attributes.

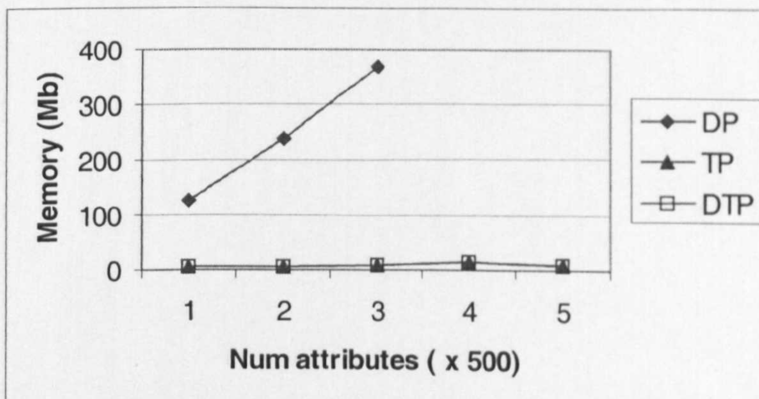


Figure 5.16: Memory requirements for T-tree/ largest PT-tree (T10.I5.D50000)

Figure 5.16 shows memory requirements in Mb for the *T*-tree (in method *DP*) and the largest *PT*-tree (in method *TP* or *DTP*). As can be seen, the memory requirements for methods *TP* and *DTP* are identical and very low because of the much smaller size of the partitioned *T*-tree. Conversely, the simple *DP* method has a rapidly increasing memory requirement, in this case because of the greater size of the unpartitioned *T*-tree.

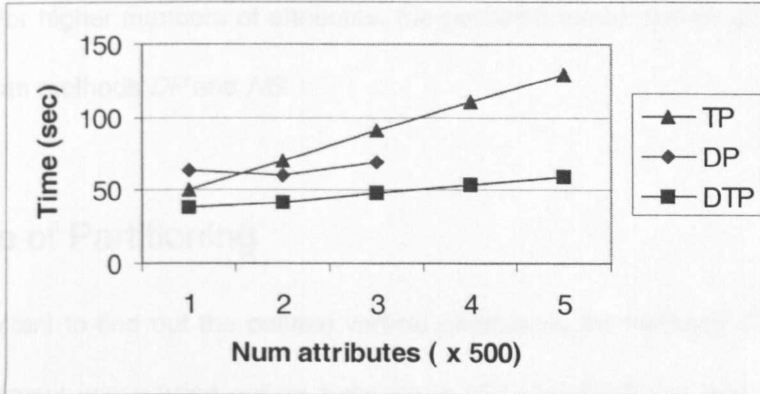


Figure 5.17: Execution times for T10.I5.D50000 (0.01% support)

Figure 5.17 shows the overall execution times, with a support threshold of 0.01%, for datasets of increasing number of attributes. The times illustrated include both the time to construct the *P*-trees (*DP* or *NB* method) or *PP*-trees (*TP* and *DTP*) and to execute the Apriori-TFP algorithm. As can be seen, the overall time to find frequent sets increases linearly for methods *TP* and *DTP*, although the rate of increase for method *DTP* is slow.

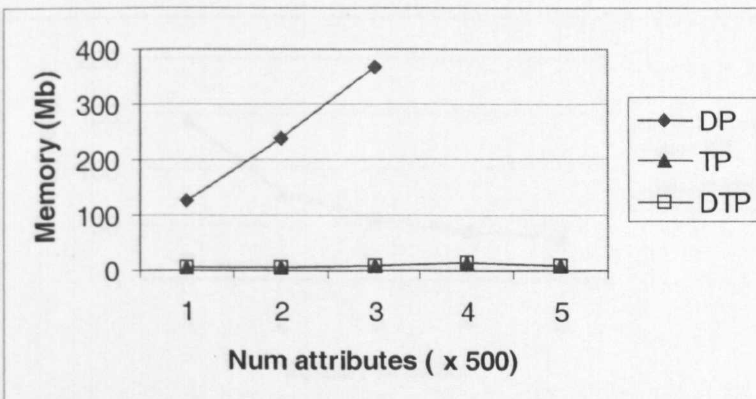


Figure 5.18: Memory requirements for T10.I5.D50000

Figure 5.18 shows maximal memory requirements (in Mb) for the datasets with increasing numbers of attributes. As can be seen, the memory requirement in method *DP* increases linearly and the rate of increase is very high, but for methods *TP* and *DTP* it is identical, very low, and nearly constant.

From these experiments for increasing number of attributes it is possible to conclude that *DTP* scales better than other methods in both execution time and primary memory requirement. For higher numbers of attributes, the performance of method *DTP* seems to be much better than methods *DP* and *NB*.

5.5 Degree of Partitioning

It is also important to find out the optimal vertical partitioning for methods *TP* and *DTP*. For this, the experiment was carried out on a database of T10.I5.N500.D50000. For the vertical partitioning the database was investigated with 1, 2, 3, 4, and 5 items/partition i.e. the database was divided vertically into 500, 250, 166, 125, and 100 partitions respectively. In the case of 3 items/partition, the last item was added to the last partition but in other cases items were distributed equally. For the horizontal segmentation of method *DTP*, it was also assumed that the memory was big enough to handle 10,000 records at a time, so the database was divided into 5 equal segments.

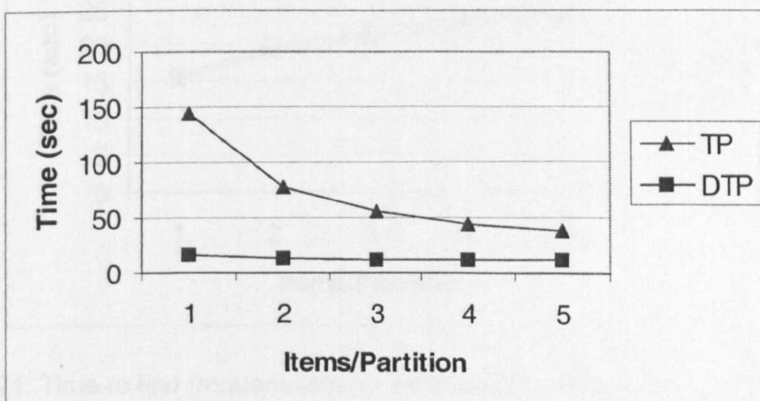


Figure 5.19: Time to construct PP-trees for increasing items/partition

Figure 5.19 shows time in seconds to create and store *PP*-trees for increasing items/partition of the database. As can be seen, the time to build these trees in method *TP* decreases with the number of partitions, but the time in method *DTP* is nearly constant. This is expected since in method *TP* the database needs to be read for each partition; whereas in method *DTP*, although the number of *PP*-trees increases with the number of partitions, the database needs to be read only once. Figure 5.20 shows the memory requirement for the largest *PP*-tree in method *TP*, and partition in method *DTP*. Memory requirements for the methods increase linearly with items/partition. This is to be expected since more items leads to a larger tree.

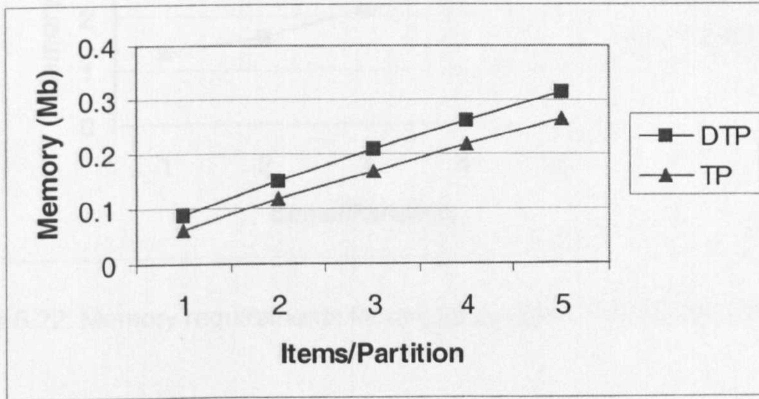


Figure 5.20: Memory requirements for largest partition (T10.I5.N500.D50000)

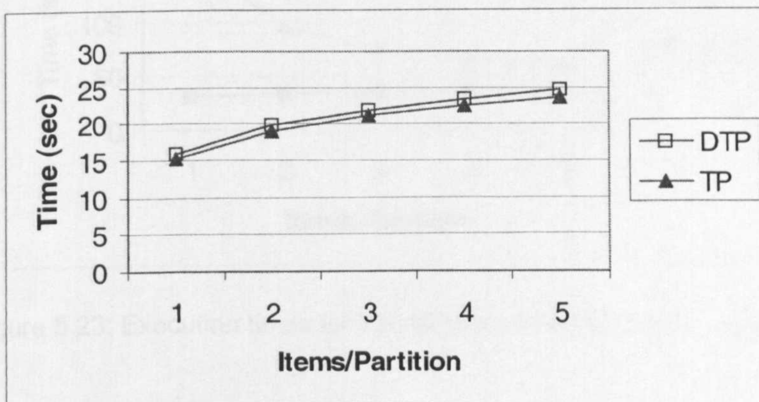


Figure 5.21: Time to find frequent sets for increasing items/partition (0.01% support)

Figure 5.21 shows time in seconds to find frequent sets (build *PT*-trees) for the support threshold of 0.01%. It is interesting to see that the time to build trees, for these methods, increases with items/partition. The reason for this is likely to be that for a smaller number of items/partition it builds smaller *PT*-trees, thus reducing "tree-walk" time. This can be seen in Figure 5.22 which shows the memory requirement for the largest *PT*-tree, which increase with items/partition. Note that the memory requirement for methods *TP* and *DTP* are identical since they build identical *PT*-trees.

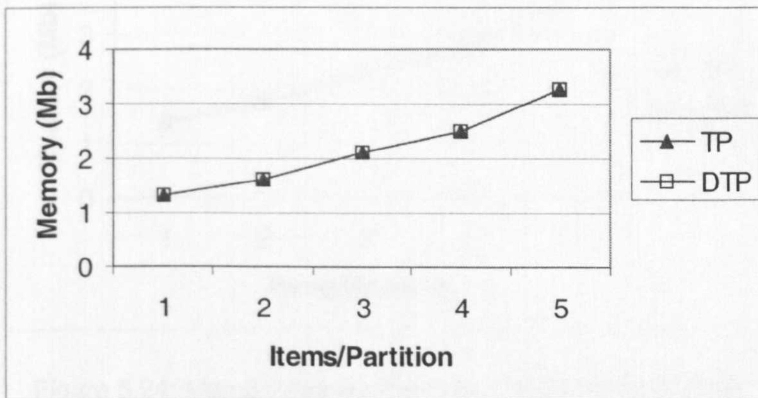


Figure 5.22: Memory requirements for largest partition (T10.I5.N500.D50000)

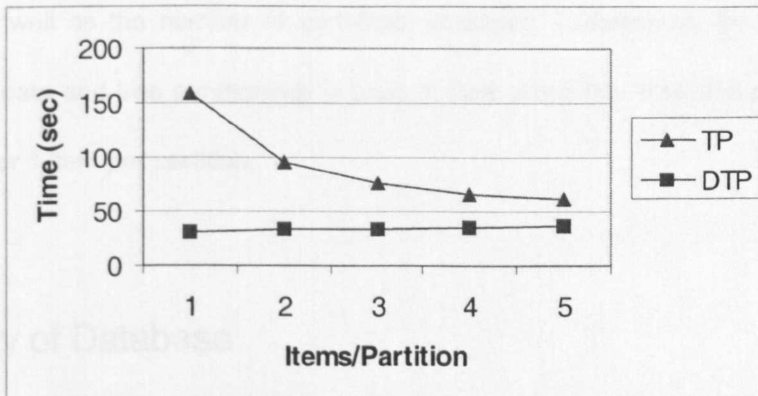


Figure 5.23: Execution times for T10.I5.N500.D50000 (0.01% support)

The total time to find frequent sets, for the support threshold of 0.01%, is shown in Figure 5.23. From the Figure it can be seen that the overall time in method *TP* decreases

significantly with the number of partitions although in method *DTP* it increase slightly with items/partition. The latter, less obvious result, arises because the increased time taken to construct a greater number of *PP*-trees is usually more than compensated for by the faster processing of smaller *PT*-trees. Figure 5.24 shows maximal memory requirements, which increase with items/partition for the dataset. As can be seen, the maximal memory requirement is very similar for both methods *TP* and *DTP*.

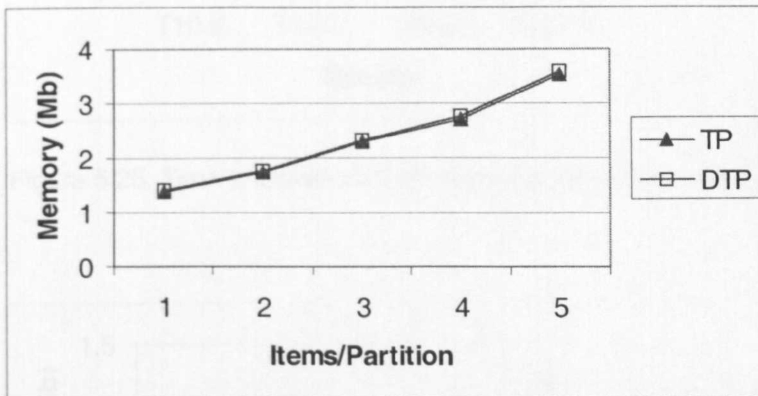


Figure 5.24: Memory requirements for T10.I5.N500.D50000

It can be seen from the above experiments that the *TP* (tree partitioning) method performs less well as the number of partitions increases. Conversely, the performance of method *DTP* (data and tree partitioning) is best, in both execution time and primary memory requirement, for 1 item per partition.

5.6 Density of Database

In this section the performance of the methods is examined with respect to databases of increasing density i.e., for increasing values of *T* and *I*. The fixed parameters used for this experiment were *N*=500 and *D*=50,000 and data sets were generated with increasing density *T*10.*I*5, *T*14.*I*7, *T*18.*I*9, and *T*22.*I*11. For the horizontal segmentation used in the *DP* (or *NB*)

and *DTP* methods, the dataset was divided into 5 equal segments. For the *TP* and *DTP* methods, the dataset was vertically divided into 50 equal partitions (10 items/partition).

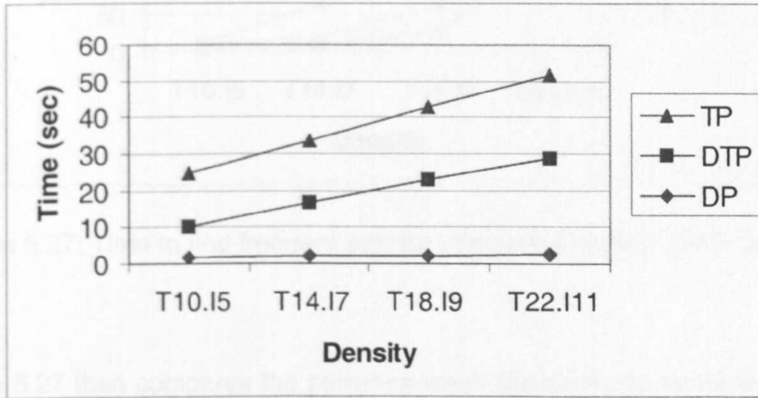


Figure 5.25: Time to construct P/PP-trees for increasing density

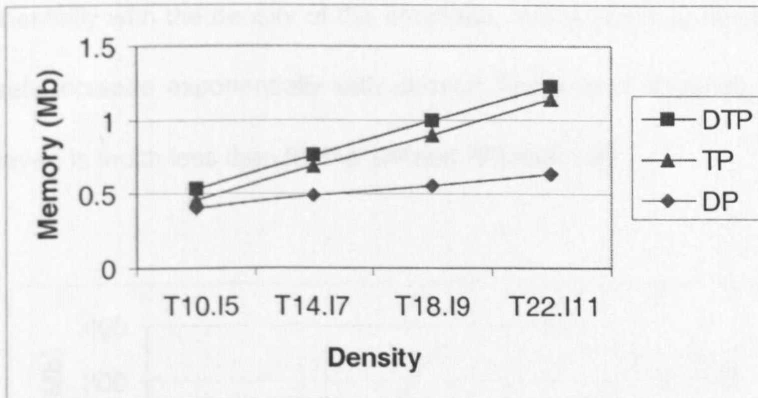


Figure 5.26: Memory requirements for largest P-tree/ Partition (N500.D50000)

Figure 5.25 shows the time in seconds to construct and store *P*-trees, in method *DP* (or *NB*), and *PP*-trees, in methods *TP* and *DTP*, for databases of increasing density. As can be seen, the time to build these trees increases linearly with the density although the rate of increase in method *DP* is very low. Figure 5.26 shows the memory requirement, in Mb, for the largest *P*-tree in method *DP*, and the vertical partition in methods *TP* and *DTP*. Memory requirements, in all methods, also increase linearly with density, as would be expected.

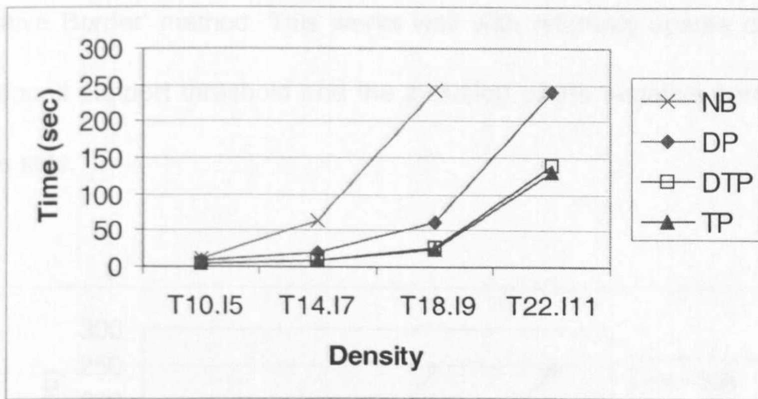


Figure 5.27: Time to find frequent sets for increasing density (0.1% support)

Figure 5.27 then compares the performance of the methods (times in seconds) when using the stored trees to compute final support totals for frequent sets, using a support threshold of 0.1%. From the Figure it is clear that the time to build trees, in all methods, increase exponentially with the density of the database, this is primarily because the number of candidate sets increase exponentially with density. The rate of increase for methods *TP* and *DTP*, however, is much less than for the *DP* and *NB* methods.

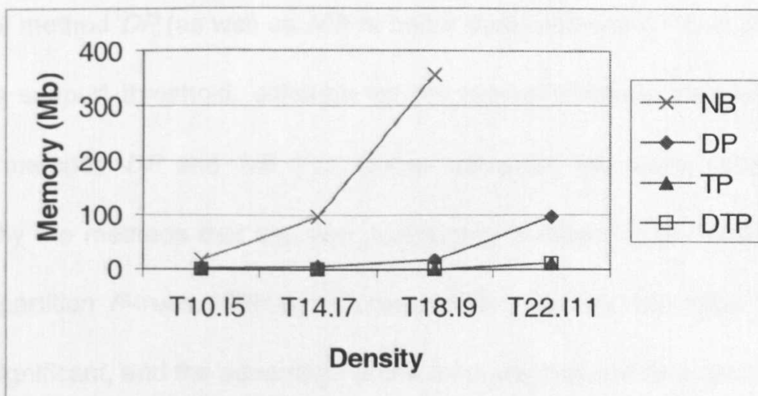


Figure 5.28: Memory requirements for T-tree/ largest PT-tree (N500.D50000)

Figure 5.28 shows memory requirements in Mb for the *T*-tree (in methods *DP* and *NB*) and the largest *PT*-tree (in method *TP* or *DTP*). As can be seen, memory requirements for the methods also increase exponentially with the density. The problem is particularly acute

with the 'Negative Border' method. This works well with relatively sparse data, but at high density the reduced support threshold and the inclusion of the negative border lead to very large candidate sets.

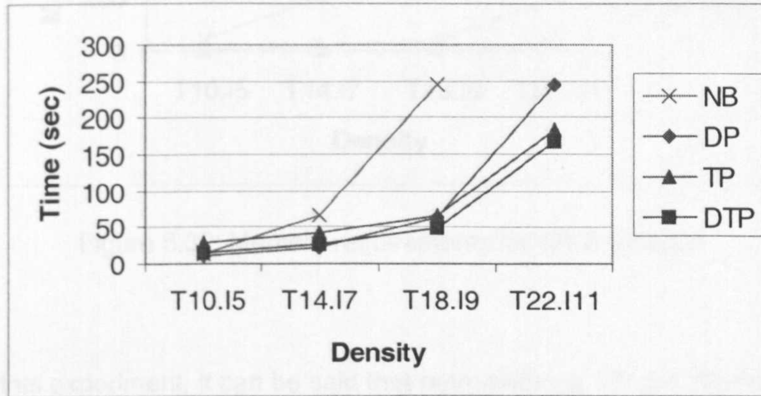


Figure 5.29: Execution times for N500.D50000 (0.1% support)

The overall time to generate frequent sets, with a support threshold of 0.1%, for the databases is shown in Figure 5.29. The Figure demonstrates that the overall time to find frequent sets increases with the increasing density of databases. Interestingly, the performance of method *DP* (as well as *NB*) is better then methods *TP* and *DTP* for a sparse dataset at this support threshold, although for the denser datasets methods *TP* and *DTP* outperformed methods *DP* and *NB*. For sparse datasets, the faster computation of the frequent sets by the methods that use tree partitioning is offset by the longer time taken to construct the partition *P*-trees. With more dense data, however, the latter factor becomes decreasingly significant, and the advantage of the methods that use tree partitioning becomes increasingly apparent. This is principally because of the much smaller candidate sets that are involved. This becomes more apparent from the comparison of maximal memory requirements, shown in Figure 5.30. This reflects the growing size of the candidate sets (and hence the *T*-tree) as the data density increases, leading to larger memory requirements and to longer times to find candidates.

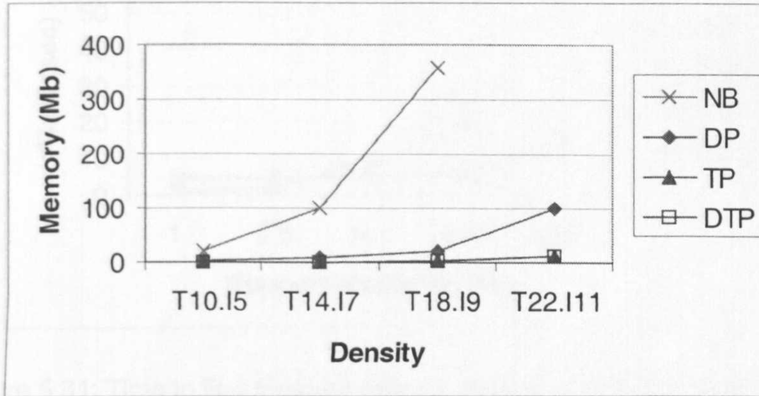


Figure 5.30: Memory requirements for N500.D50000

From this experiment, it can be said that both methods *TP* and *DTP* outperform other methods in both execution time and primary memory requirement, as the density of the dataset increases.

5.7 Performance Comparison

The final set of experiments compare the performance of the four methods for different support thresholds. The experiments again relate to the T10.I5.N500.D50000 data set, for support thresholds decreasing from 1.0 through to 0.01. In these experiments, the data was again divided into 5 segments for the *DP* (or *NB*) and *DTP* methods. In addition, for the vertical partitioning of methods *TP* and *DTP*, the database was also divided into 500 partitions (1 item/partition).

For methods *DP* (or *NB*), *TP* and *DTP*, times to construct and store *P/PP*-trees are 1.97, 143.69, and 15.48 seconds and memory requirement for the largest *P*-tree/ *PP*-tree/ Partition are 0.408174, 0.06315, and 0.088026 Mb respectively.

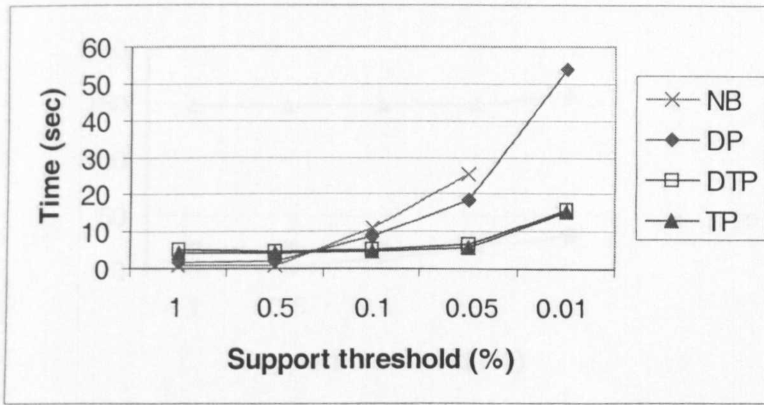


Figure 5.31: Time to find frequent sets for decreasing support thresholds

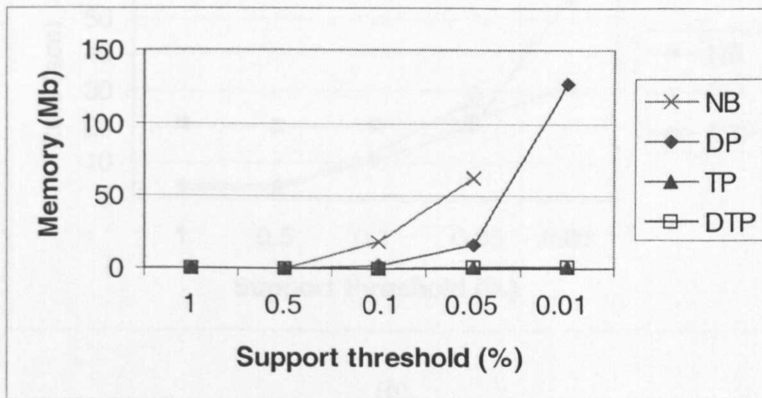
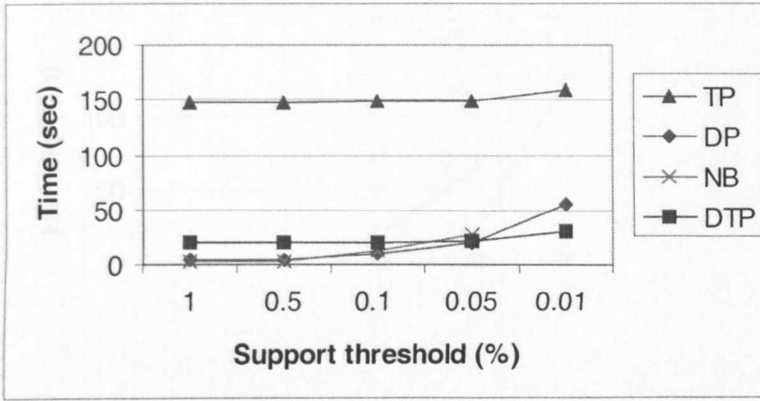
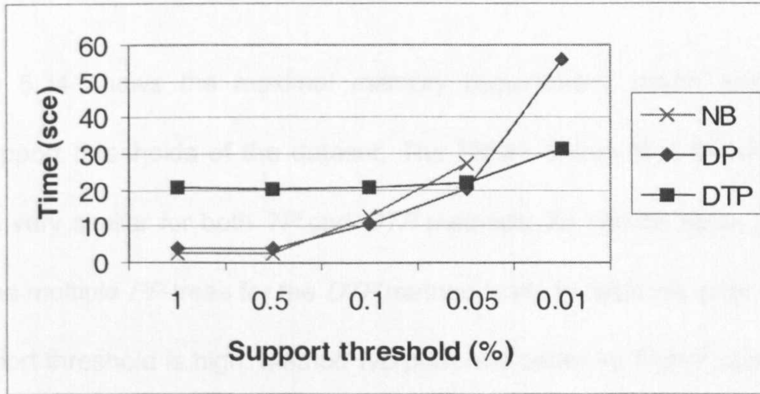


Figure 5.32: Memory requirements for largest T-tree/Partition (T10.I5.N500.D50000)

Figure 5.31 shows time in seconds to compute final support totals for frequent sets for varying support thresholds. As expected, the time to find these frequent sets, in all methods, increases for decreasing support thresholds although the rate of increase is low for methods *TP* and *DTP*. In method *NB*, the initial tree could not be built for the lowest support thresholds because of memory overflow. This can be seen in Figure 5.32, which shows the memory requirement for the largest *T*-tree or partition. This shows a significant increase for methods, *NB* and *DP*, for decreasing support thresholds but is nearly constant for methods *TP* and *DTP*. Note that the memory requirements for methods *TP* and *DTP* are identical and negligible at lower thresholds.



(a)



(b)

Figure 5.33: Performance of the methods for decreasing support thresholds

The total time to find frequent sets, for decreasing support thresholds, is shown in Figure 5.33. Here the time taken to construct the *P/PP*-trees is taken into account. As can be seen, the overall time to find frequent sets in all methods increases for decreasing support thresholds. Here again, as can be seen from Figure 5.33 (a), the performance of method *TP* is worst because of the *PP*-trees building time, which is very high for 1 item/partition. For higher support thresholds the *NB* method is fastest, this can clearly be seen in Figure 5.33 (b). As the support threshold is reduced, however, method *DTP* scales better and for thresholds below 0.05 is the best method.

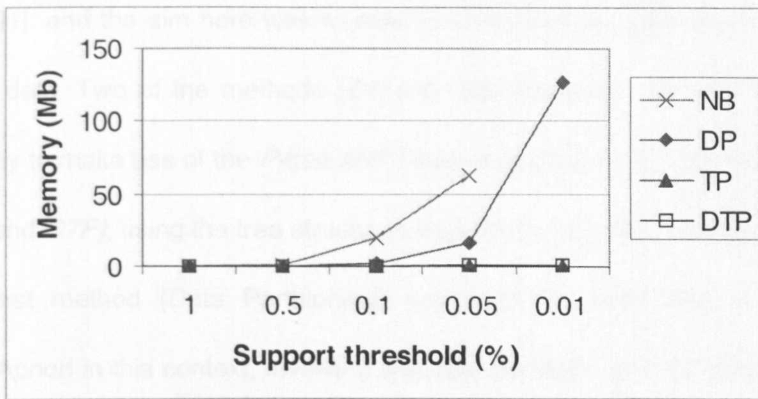


Figure 5.34: Memory requirements for T10.I5.N500.D50000

Figure 5.34 shows the maximal memory requirement, which also increases for decreasing support thresholds of the dataset. The Figure shows that the maximal memory requirement is very similar for both *TP* and *DTP* methods. As can be seen, the overhead of constructing the multiple *PP*-trees for the *DTP* method leads to relatively poor execution times when the support threshold is high. Method *NB* performs better for higher support thresholds, but declines rapidly for lower thresholds. Performance of method *DP* was also failing for lower thresholds. As the support threshold is reduced, the increasing cost of servicing a growing set of candidates in the *NB* (as well as *DP*) methods lead to rapidly increasing memory requirements and execution times. At the lower support thresholds, however, method *DTP* strongly outperforms all other methods and is clearly the best in both execution time and primary memory requirement for this data set.

5.8 Summary

In this chapter the results of a number of experiments, carried out to investigate the performance of the methods for extracting association rules from databases where the data is too large to be contained in main memory, has been described. For the experiments the databases were reordered so that attributes appeared in decreasing frequency. This reordering offers significant performance advantages to Apriori-based algorithms [Coenen

and Leng, 2001], and the aim here was to examine how best to apply the approach to non-store-resident data. Two of the methods (*DP* and *NB*) examined are established methods, modified slightly to make use of the *P*-tree and *T*-tree data structures, and two others are new methods (*TP* and *DTP)*, using the tree structures to produce a vertical partitioning of the data.

The first method (Data Partitioning) examined is, essentially, a straightforward adaptation of Apriori in this context, involving a simple partitioning of the data into segments. It has been shown that the method scales well for increasing numbers of segments, but, like the original Apriori, its performance drawback is the repeated passes of disk-resident data that it involves, especially when low support thresholds and/or high-density data is involved. The sampling method, *NB* (Negative Border), of [Toivonen, 1996] was developed specifically to avoid the cost of multiple database passes when data is non-store-resident. The results confirm its effectiveness for relatively high support thresholds; but, at very low support thresholds, as candidate sets become very large, the additional memory requirement of the method becomes an increasing overhead, and in the extreme the method requires an additional database pass to find all the frequent sets.

The new, *TP* and *DTP*, methods have been introduced with the aim of reducing the memory requirement by a partitioning of the attribute set, and a corresponding construction of trees so that each contain some subset of the candidates to be counted. The results show these methods to be extremely effective in limiting the maximal primary memory requirement, even at very low support thresholds, because they enable both the original data (as represented by *PP*-trees) and the candidate set to be partitioned for memory management. For the most computationally demanding cases, at low support thresholds, a high degree of partitioning appears to work best, and in these cases these methods are significantly faster than the others we have considered. With a very high degree of partitioning, the increased cost of preprocessing the data to produce *PP*-trees may become a problem. This is evident from the results for method *TP*. It has been shown, however, that this problem can be overcome by applying a horizontal segmentation of the data together with the vertical

partitioning (method *DTP*). In the experiments, this substantially reduced the preprocessing time with little effect on performance in generating the frequent sets. At low support thresholds, this method significantly outperforms all others in both time and space requirements.

Chapter 6

Distributed Association Rule Mining

6.1 Introduction

The work described so far has examined ways of partitioning data for applications involving a single processor implementation. The tree partition (vertical partition) strategy has been shown to be extremely effective in limiting the maximal primary memory requirement while finding frequent sets. As a result of increasing the degree of partitioning, the cost of preprocessing the data to produce partition *P*-trees increases for a single processor. A solution to the problem has been demonstrated using a single processor and applying a horizontal segmentation of the data together with the vertical partitioning. Although this method has been shown to improve performance significantly, it is clear that a substantial cost is still incurred as the size of the database to be processed is increased. The problem may, however, be overcome using multiple processors by distributing the input data among processors. Using multiple processors has another notable advantage: all frequent sets may easily be found without costly data preprocessing for producing partition *P*-trees.

The *TP* (Tree Partitioning) technique can be applied to "distributed" and "parallel" Association Rule Mining to distribute the input data among multiple processors. Using this approach each partition can be mined in isolation while at the same time taking into account the possibility of the existence of frequent itemsets distributed across two or more partitions. The partitioning is facilitated by the novel *T*-tree data structure (no *P*-tree), and association rule mining algorithm (Apriori-T), that allows for computationally effective distributed/parallel ARM. The partitioning approach offers significant advantages with respect to computational efficiency when compared to alternative mechanisms for (a) dividing the input data between

processors and/or (b) achieving distributed/parallel ARM. Distributed ARM that makes use of the *TP* technique was introduced in our paper published in [Coenen *et al.,* 2003]. A substantial portion of this chapter has been taken from the text of that paper.

The organisation of the chapter is as follows: in section 6.2 the Apriori-T algorithm which uses only the *T*-tree data structure is described. In section 6.3 some technical aspects of the architecture/network configuration that is assumed are considered, then three distinct distributed/parallel algorithms are compared (sections 6.4, 6.5 and 6.6) that make use of the *T*-tree structure, namely: (1) *Data Distribution* (*DD*), (2) *Task Distribution* (*TKD*) and *Tree Distribution* (*TD*). Some performance comparisons are then presented in section 6.7, and a summary in section 6.8.

6.2 The Apriori-T algorithm

The *T*-tree data structure described in Chapter 3: (a) readily lends itself to distribution/parallelisation, and (b) facilitates vertical distribution of the input dataset. Moreover, the cost of producing the partition *P*-trees, which is significant for single processor system, is avoided. An algorithm Apriori-T has thus been developed, for parallel ARM, combining the classic Apriori ARM algorithm [Agrawal and Srikant, 1994] with only the *T*-tree data structure (Apriori-TFP described in the previous chapters uses both the *T*-tree and P-tree structures). Apriori-T uses the original data set (not *P*-trees) to find all frequent itemsets. As each level is processed, candidates are added as a new level of the *T*-tree, their support is counted, and those that do not reach the required support threshold pruned. When the algorithm terminates, the *T*-tree contains only frequent itemsets. At each level, new candidate itemsets of size *K* are generated from identified *K-1* itemsets, using the downward closure property of itemsets, which in turn may necessitate the examination of neighbouring branches in the *T*-tree to determine if a particular *K-1* subset is supported. We refer to this process as X-checking. Note that X-checking adds a computational overhead; offset against the

additional effort required to establish whether a candidate *K* itemset, all of whose *K-1* itemsets may not necessarily be supported, is or is not a frequent itemset. In a distributed implementation X-checking may have a greater cost.

The number of candidate nodes generated during the construction of a *T*-tree, and consequently the computational effort required, is very much dependent on the distribution of items within the input data. Best results are produced by reordering the dataset, according to the support counts for the 1-itemserts, so that the most frequent 1-itemsets occur first ([Coenen and Leng, 2001]).

Unlike Apriori-TFP, Apriori-T uses only the *T*-tree data structure, and so does not exploit the performance advantages offered by the *P*-tree structure. The use of only the *T*-tree is focused here, because this allows the original data to be distributed freely between processors, and there will be no additional communication costs involved in preprocessing the data. In some contexts, of course, a distributed implementation of the *P*-tree structure would offer advantages. However, making the simplifying assumption that the data will not be processed into *P*-tree allows us to consider only the cost effective means of partitioning the *T*-tree, and to compare three methods on this basis.

6.3 Architecture and network configuration

The *DD, TKD* and *TD* algorithms described here assume the availability of at least two processors (preferably more), one *Master* and one or more *Workers* operating across a network. Naturally, the approaches described here perform better as the number of available processors is increased. Most of the experiments have used five processors, one master and four workers. The significant distinction between the master and the worker processors is that synchronisation, where required, is the responsibility of the master. Processors are identified by a unique ID number: 0 for the Master, and 1 to *N* for the Workers. The algorithms also assume that all processors have access, across a network, to a central data warehouse.

The parallel/distributed ARM algorithms described here have all been implemented using JavaSpaces [Arnold *et al.*, 1999], which in turn was inspired by Linda [Carreiro and Galernter, 1989]. The philosophical base of both JavaSpaces and Linda is the existence of a central store of objects (called a *tuple space*) which can be accessed using a small number of operations (three in the case of JavaSpaces - write, read and take), this in turn greatly simplifies the implementation of both parallel and distributed applications. The exchange of information (*messaging*) using JavaSpaces takes the form of sending serialized objects (i.e. converted into a stream of bytes) to and from the space.

Although JavaSpaces have been used, the ARM algorithms/techniques described here could equally have been implemented using many other appropriate platforms, including agent platforms such as Java agents and the JADE message passing environment.

6.4 Data Distribution

The *Data Distribution* (*DD*) algorithm uses horizontal segmentation to divide the dataset into segments each comprising an equal number of records. ARM in this case comprises the generation of a number of *T*-trees, one for each segment, which must then be accumulated on completion of each level. The approach is therefore similar to the "count distribution algorithm" described in section 2.6.

The algorithm comprises the following steps:

1. Start all processes, Master plus a number of Workers.
2. Master determines horizontal segmentation according to the total number of available processes and transmits this information, via the JavaSpace, to the Worker processes.
3. Each process generates a level 1 *local* *T*-tree for its allocated segment.

4. Each process serialises and sends its level 1 local *T*-tree to each other process so that all processes can collect their own and the other level 1 local *T*-trees into a single *global* *T*-tree.
5. Each process then prunes its level 1 global *T*-tree according to the support threshold, and generates and counts the next level local candidates for its allocated segment.
6. Steps 4 and 5 are repeated, for levels 2,3... until there are no more candidate sets to be counted.

Note that the *DD* algorithm requires the transmission of a local (component) *T*-tree at each level on behalf of each process. This is necessary because pruning of the current level and construction of the next level requires knowledge of the *global* support counts. Messaging in both parallel and distributed systems represents a significant computational overhead (in some cases be more important than any other advantage gained). By serializing a single level of nodes in a *T*-tree and wrapping the serialisation up as a single message this overhead is significantly reduced, but remains a significant factor in the performance of this method.

6.5 Task Distribution

In the *Task Distribution* (*TKD*) algorithm each processor interacts with the entire data set (as opposed to a horizontal segment or a vertical partition). However, the candidate sets generated at each level (as the Apriori-T algorithm proceeds) are equally distributed among the available processors so that each process determines the support counts only for its allocated candidates. The approach is therefore similar to the "data distribution algorithm" (described in chapter 2 section 2.6). As mentioned earlier *TKD,* unlike "data distribution algorithm", has access to a central data warehouse.

The algorithm comprises the following steps:

1. Start all processes, Master plus a number of Workers.
2. Each process has an identification number and is aware of the number of processes (Workers plus Master) that are running.
3. Each process determines the set of 1-itemset candidates, and then (using knowledge of the number of available processes) identifies its allocation of candidate items.
4. Processes then generate and count local "top-level" *T*-trees for their allocation, serialise these trees and transmit them to each other process.
5. Each process accumulates its local top-level *T*-tree with those received from the other processes and produces a global top-level *T*-tree.
6. Processes then generate the next level of candidate item sets, determine their own allocation of candidates, and then each generates and counts a new level in their copy of the global *T*-tree "so far" with respect to their allocation.
7. Each process serialises and transmits the newly counted *T*-tree level for its allocation to the other processes; after which it will collect the serialisation received from the other processors into its copy of the global *T*-tree "so far".
8. Steps 6 and 7 are repeated for levels 2,3,... until there are no more candidate sets.

Again, as with the *DD* approach, *TKD* includes a significant amount of messaging at the end of each level, though in this case the quantity of information exchange is less.

6.6 Tree Distribution

The Tree Distribution (*TD*) algorithm commences with distributing the input dataset over the available number of processors using only a tree partition (vertical partition) strategy. Initially the set of single items (columns) is split equally between the available processes so that an *allocationItemSet* (a sequence of single items) is defined for each process in terms of a *startColNum* and *endColNum*.

$$allocationItemSet = \{ n \mid startColNum < n \leq endColNum \}$$

Each process will have its own *allocationItemSet* which is then used to determine the subset of the input dataset to be considered by the process. Using its *allocationItemSet* each process will proceed as follows:

1. Remove all records in the input dataset that do not intersect with the *allocationItemSet*.
2. From the remaining records remove those items whose column number is greater than *endColNum*. We cannot remove those items whose identifiers are less than *startColNum* because these may represent the "leading sub-string" of frequent itemset to be included in the sub *T*-tree counted by the process.

The input dataset distribution procedure, given an *allocationItemSet*, can be summarised as follows:

$\forall$ *records* $\in$ *input data*

if (*record* $\cap$ *allocationItemSet* $\equiv$ *true*)

record = { $n \mid n \in$ *record* $n \leq$ *endColNum* }

else delete record.

For example, given the data set {{A,C,F}, {B}, {A,C,E}, {B,D}, {A,E}, {A,B,C}, {D}, {A,B}, {C}, {A,B,D}}, and assuming three processes, the above partitioning process will result in three dataset partitions:

Process 1 (a to b): {{A}, {B}, {A}, {B}, {A}, {A,B}, { }, {A,B}, { }, {A,B}}

Process 2 (c to d): {{A,C}, { }, {A,C}, {B,D}, { }, {A,B,C}, {D}, { }, {C}, {A,B,D}}

Process 3 (e to f): {{A,C,F}, { }, {A,C,E}, { }, {A,E}, { }, { }, { }, { }, { }}

Figure 6.1 shows the resulting sub *T*-trees assuming all combinations represented by each partition are supported.

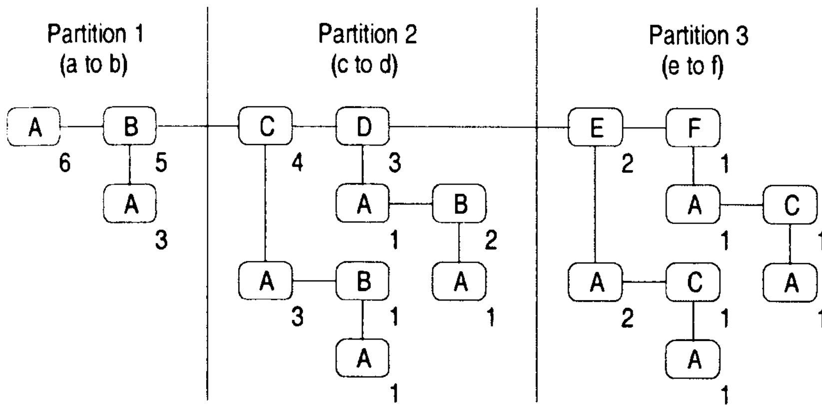


Figure 6.1. Distributed T-tree representing the vertical partitioning presented in the example

Note that because the input dataset is ordered according to the frequency of 1-itemsets the size of the individual partitioned sets does not necessarily increase as the *endColNum* approaches *N* (the number of items in the input dataset); in the later partitions, the lower frequency leads to more records being eliminated. Thus the overall result of the vertical partitioning is that the overall size of the dataset (applicable to the process in question) is reduced. Once partitioning is complete each partition can be mined, using the Apriori-T algorithm, in isolation.

The *TD* algorithm can thus be summarised as follows:

1. Start all processes, Master plus a number of Workers.
2. Master determines the division of *allocationItemSet* according to the total number of available processes and transmits this information to the Workers.
3. Each process then generates a *T*-tree for its allocated partition (a sub tree of the final *T*-tree).

4. On completion each process transmits its partition of the *T*-tree to all other processes which are then merged into a single *T*-tree (so that each process has a copy of he final *T*-tree ready for the next stage in the ARM process – rule generation).

The process begins with a top-level "tree" comprising only those 1-itemsets included in its *allocationItemSet*. The process will then generate the candidate 2-itemsets that belong in its sub *T*-tree. These will comprise all the possible pairings between each element in the *allocationItemSet* and the lexicographically preceding attributes of those elements (see Figure 6.1). The support values for the candidate 2-itemsets are then determined and the sets pruned to leave only frequent 2-itemsets. Candidate sets for the third level are then generated. Again, no items from succeeding *allocationItemSet* are considered, but, the possible candidates will, in general, have subsets which are contained in preceding *allocationItemSet* and which, therefore, are being counted by some other process. Checking to find whether subsets in a different partition are frequent would involve a significant message-passing overhead, so in this case, this X-checking is not considered. Instead, the process will generate its candidates assuming, where necessary, that any subsets outside its local *T*-tree are frequent.

6.7 Evaluation

The evaluation presented here uses five processes and the data set T20.I10.N500.D500K (generated using the IBM Quest generator used in [Agrawal and Srikant, 1994]), although similar results have been obtained using other data sets. In all cases the dataset has been preprocessed so that it is in descending order. The experiments were run in a network of 1.2 GHz Intel Celeron CPUs each with 512 Mb of RAM and running under Red Hat Linux 7.3.

6.7.1 Number of *T*-tree Messages

The most significant overhead of any distributed/parallel ARM algorithm is the number of messages sent and received between processes. For *DD* (data distribution) and *TKD* (task distribution) processes are required to exchange information as each level of the *T*-tree is constructed; the number of levels will equal the size of the largest supported set. For *TD* (tree distribution) the number of messages sent is independent of the number of levels in the *T*-tree; communication takes place only at the end of the tree construction. *TD* therefore has a clear advantage in terms of the number of messages sent.

6.7.2 Amount of Data Sent and Received

Figure 6.2 shows the average amount of data sent and received by each process for each of the Apriori-T algorithms under consideration assuming five processes. Note that:

- With respect to *DD*, for each generated *T*-tree level, un-pruned levels of the *T*-tree are passed from one process to another (and then pruned).
- In the case of *TKD* pruned sections of levels in the *T*-tree are passed from one process to another.
- *TD* passes entire pruned sub *T*-trees (pruned *T*-tree branches, not entire levels). Consequently the amount of data passed between processes when using *TD* is significantly less than that associated with the other approaches.
- In the case of *DD* adding more processes probably increases the amount of communication, because all processes send data to all others. In the case of *TKD* and *TD*, however, each process sends only the set of candidates it is counting or has counted, which becomes proportionately smaller as the number of processes is increased; i.e. for these methods the messaging overhead remains approximately constant.

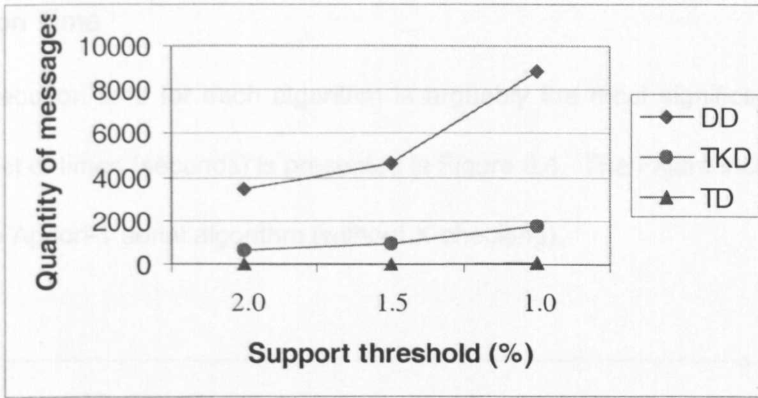


Figure 6.2. Average Total size (Kbytes) of messages sent and read/taken per process

6.7.3 Number of Updates

The number of support value updates/incrementations per process is a good indication of the amount of work done by each process. Figure 6.3 gives the number of updates for each of the algorithms under consideration for a range of support thresholds and using five processes. Note that:

- Figure 6.3 includes, for comparison, values for the serial form of the Apriori-T algorithm without X-checking.
- *DD*, *TKD* and *TD* all have the same average number of updates (as would be expected).

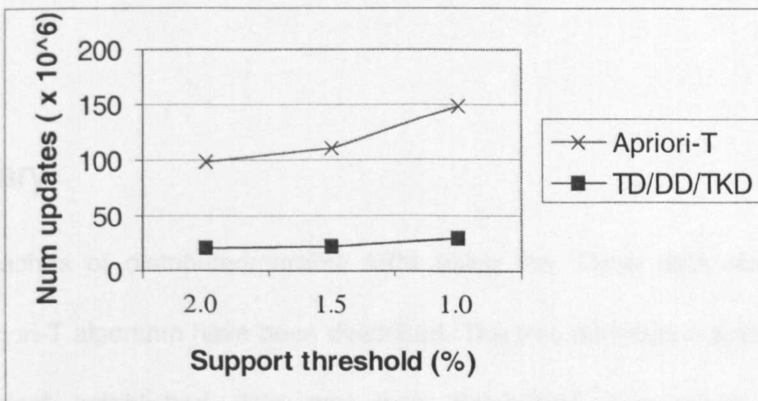


Figure 6.3. Average number of updates (x 10^6) to generate a final T-tree per process

6.7.4 Execution Time

The overall execution time for each algorithm is arguably the most significant performance parameter. A set of times (seconds) is presented is Figure 6.4. The Figure includes execution times using the Apriori-T serial algorithm (without X-checking).

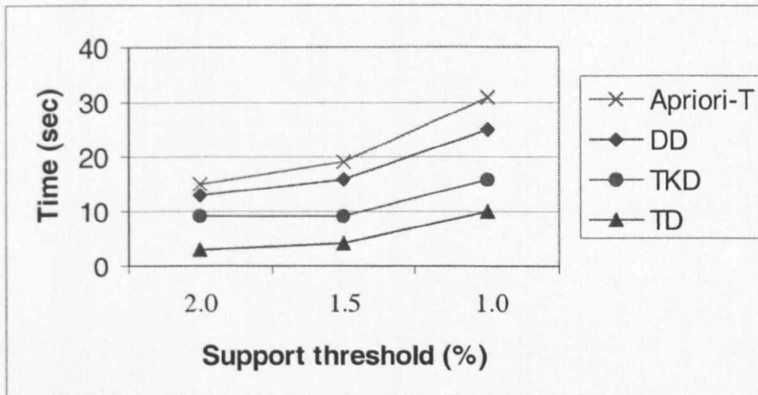


Figure 6.4. Average execution time (seconds) per process

In terms of execution time the task distribution algorithms, *TKD*, and (especially) the tree distribution algorithm *TD*, perform much better than the data distribution algorithm (*DD*) because of the messaging overhead. Note also that, for *DD*, as further processes are added, the increasing overhead of messaging more than out-weighs any gain from using additional processes, so that distribution/parallelisation becomes counter productive. *TKD* shows some gain from the addition of further processes, however *TD* gives the best results and the best scaling.

6.8 Summary

Various approaches of distributed/parallel ARM using the *T*-tree data structure and the associated Apriori-T algorithm have been described. The tree distribution approach has been evaluated against established data and task distribution approaches. The principal advantages offered by *TD* are: (1) minimal amount of message passing compared to *DD* and

TKD, (2) minimal message size, especially with respect to *DD* but also when compared to *TKD*, and (3) enhanced efficiency as the number of processes increases, unlike *DD*.

The experimental evaluation of tree distribution clearly demonstrates that the approach performs much better than those methods that use data and task distributed approaches. This is largely due to the vertical partition technique of the *T*-tree data structure.

Chapter 7

Overall Conclusion

7.1 Introduction

Detailed conclusions have been presented at the end of individual chapters. This chapter is presented to summarise the overall conclusions drawn from the research carried out for this thesis.

The problems of ARM and the goal of the research have been described in Chapter 1. The greatest challenge of ARM is posed by data that is too large to be contained in primary memory, especially when high data density and/or a low support threshold give rise to very large numbers of candidates that must be counted. The number of the candidate sets for which support counts are required may make it impossible for the algorithm to proceed entirely within primary memory. In this thesis, Association Rule Mining Algorithms for very large datasets which cannot be contained in main memory (thus requiring some strategy for partitioning the data) has been considered.

7.2 P-tree and T-tree data structures

For effective partitioning of a large and dense database, it is important to use suitable data structures. Two data structures, *P*-tree and *T*-tree, were used for the experiments. Detailed descriptions of these structures have been given in Chapter 3.

The major advantages offered by the *P*-tree structure in respect to a large and dense database are: (1) it allows partial counts of the support for individual nodes within the tree to be accumulated effectively as the tree is constructed, (2) it merges duplicated records and

records with leading sub-strings, thus reducing the storage and processing requirements for these, (3) it uses minimum and effective referencing which is thus capable of dealing with non-store resident data, and most importantly, (4) it can easily be partitioned into subtrees and stored into secondary storage as a composite structure.

The *T*-tree offers great advantages in respect to a large and dense database: (1) it contains all frequent sets with their complete support-counts, (2) it is a very versatile data structure that can easily be used in conjunction with many established ARM methods, (3) it also uses effective referencing which can readily be adaptable for non store-residence data, and most importantly, (4) branches of it can easily be partitioned into subtrees and processed independently.

Thus, the *P*-tree and *T*-tree structures can easily form the basis for an effective partitioning of the data.

7.3 Strategies for Partitioning Data

In the research, various partitioning strategies have been examined to limit the total primary memory requirement, including that required both for the source data and for the candidate sets, for carrying out the counting of the required support totals.

Partitioning of the source data has been examined in the both *DP* (Data Partitioning) and *NB* (Negative Border) methods. In these methods, the database was horizontally divided into segments of equal number of records and each segment was then considered in isolation to create a *P*-tree for each. The entire database was read only once while creating *P*-trees. These trees were stored into backing store to reduce the primary memory requirement for a single processor system. Using the disk-resident data (as represented by *P*-trees), these methods finally create a single *T*-tree.

A range of experiments have been carried out to investigate the performance of the methods for finding frequent sets in cases where the data is too large to be contained in main

memory. The *DP* method examined is an adaptation of Apriori [Agrawal and Srikant, 1994], involving horizontal partitioning of the data into segments. It has been shown that the method scales well for increasing number of segments, but, like the original Apriori, its performance drawback is the repeated passes of disk-resident data, especially when low support thresholds and/or high-density data is involved. The sampling method of [Toivonen, 1996], *NB* in this thesis, was developed specifically to avoid the cost of multiple database passes when data is non-store-resident. The results confirm its effectiveness for relatively high support thresholds, but, at very low support thresholds, as candidate sets become very large, the additional memory requirement of the negative border method becomes an increasing overhead.

The tree partitioning, a form of vertical partitioning of the data, has been examined in the *TP* (Tree Partitioning) method. The *TP* method builds partitioned *P*-trees that can be processed independently to build *T*-trees. The tree partitioning has been introduced to reduce the memory requirement by a partitioning of the attribute set, and a corresponding construction of trees such that each contains some subset of the candidates to be counted. It has been shown that this method is extremely effective in limiting the maximal primary memory requirement, even at very low support thresholds, because it enables both the original data (as represented by *PP*-trees) and the candidate set to be partitioned for memory management. With a very high degree of partitioning, the increased cost of preprocessing the data to produce *PP*-trees becomes a problem since repeated passes of the database are required for single processor implementation.

It has been shown, however, that this problem can be overcome by applying a horizontal segmentation of the data together with the vertical partitioning. The *DTP* (Data and Tree Partitioning) method constructs all the *PP*-trees in a single database pass, during which partial supports are also counted. It has been shown that this substantially reduced the preprocessing time with little effect on performance in generating the frequent sets. It has also been shown that this method is extremely effective in limiting the maximal primary memory

requirement. For the most computationally demanding cases, at low support thresholds, a high degree of partitioning appears to work best, and in these cases this method is significantly faster than the other considered methods.

7.4 Distributed ARM

The work described has concentrated on the examination of ways of partitioning data for applications involving a single processor implementation. Even in this type of implementation, the tree partition strategy has become extremely effective in limiting the maximal primary memory requirement while finding frequent sets. It has been shown that the tree partition is also effective in applications involving multiple processors where the input data is distributed among processors.

The *TD* (tree distribution) approach has been evaluated against established *DD* (data distribution) and *TKD* (task distribution) approaches. In the *DD* method, the dataset is horizontally divided into segments each comprising an equal number of records and each process then generates a *T*-tree using its allocated segment (exchanging information on-route as necessary). The major drawback of the *DD* method is that it requires the transmission of a local *T*-tree at each level on behalf of each process. In the *TKD* method, each processor interacts with the entire data set. However, the candidate sets generated at each level are equally distributed among the available processors so that each process determines the support counts only for its allocated candidates. Again, as with the *DD* approach, *TKD* includes a significant amount of messaging at the end of each level. In the Tree Distribution (*TD*) method, the input dataset is divided using only the tree partition (vertical partition) strategy and then distributed over the available number of processes. Each process then generates a *T*-tree for its allocated partition. The exchanging information is therefore not required at the end of each level. On completion each process transmits its partition of the *T*-tree to all other processes which are then merged into a single *T*-tree.

It has been shown that the tree distribution approach performs much better than those methods that use data and task distribution approaches.

7.5 Summary

This thesis has examined ways of partitioning data for Association Rule Mining. The aim has been to identify methods that will enable efficient counting of frequent sets in cases where the data is too large to be contained in primary memory, and also where the density of the data means that the number of candidates to be considered becomes very large. The starting point was a method which makes use of an initial preprocessing of the data into a tree structure (the *P-tree*) which incorporates a partial counting of support totals and offer significant performance advantages. Here, ways of applying the approach in cases that require the data to be partitioned for primary memory use have been investigated. Methods have been described, in particular, that involve a partitioning of the tree structures to enable separate subtrees to be processed independently. The advantage of this approach is that it allows both the original data to be partitioned into more manageable subsets, and also partitions the candidate sets to be counted. The latter results in both lower memory requirements and also faster counting in a single processor application.

The experimental results reported in the thesis show that the *DTP* (Data and Tree Partitioning) method described is extremely effective in limiting the maximal memory requirements of the algorithm, while its execution time scales only slowly and linearly with increasing data dimensions. Its overall performance, both in execution time and especially in memory requirements, is significantly better than that obtained from either simple data segmentation or a method that aims to find frequent sets from a sample of the data. The advantage increases with increasing density of data and with reduced thresholds of support – i.e. for the cases that are in general most challenging for association rule mining. Furthermore, a relatively high proportion of the time required by the method is taken up in the

preprocessing stage during which the *P*-trees are constructed. Because this stage is independent of the later stages, in many single processor applications it could be accepted as a one-off data preparation cost. In this case, the gain over other methods becomes even more marked.

The evaluations of different distribution approaches demonstrate that the approach that distributes the *T*-tree (*TD*) performs much better than the approach that distributes data (*DD*) or tasks (*TDK*). This is because of the high message-passing overheads associated with the latter approaches. In the experiments, the best results were obtained using the *TD* method (i.e. vertical partitioning), which exploits the structure of the *T*-tree most effectively. The advantages offered by the *TD* approach result from the limited number of messages sent and the relatively small content of the messages. These advantages result, in turn, entirely from the vertical partitioning of the *T*-tree data structure described.

For future work, there are a few avenues that can be pursued:

- A variety of variables were required to be set for various experiments of this thesis. Future work can investigate finding the optimal degree of segmentation and partitioning in different cases.
- Performance comparisons of the methods have been shown using synthetic datasets. Performance of different methods can also be compared using real very large and dense datasets.
- It would be interesting to compare *DP* (Data Partitioning) method with Partition [Savasere *et al.,* 1995] algorithm using secondary storage for storing frequent sets of each partition.
- The distribution approaches evaluated in this thesis have been implemented without using the *P*-tree data structure. Further research can be carried out implementing parallel methods using the *P*-tree with *T*-tree, in order to investigate the most effective strategies for distributed implementation.

- Comparisons of methods using *P*-tree and *T*-tree with those using *T*-tree only (serial and parallel) can be performed.
- Furthermore, the distribution approaches are evaluated only for a fixed number of processors. Experiments can be carried out to investigate how things scale-up for increasing number of processors.

References

[Agrawal *et al.*, 1993a]

Agrawal, R., Imielinski, T. and Swami, A. "Mining Association Rules between Sets of Items in Large Databases." *In Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 207-216, Washington, D.C., May 1993.

[Agrawal *et al.*, 1993b]

Agrawal, R. Imielinski, T. and Swami, A. "Database Mining: A performance Perspective." *In IEEE Transactions on Knowledge and Data Engineering,* Vol. 5, No. 6, pages 914-925, December 1993.

[Agrawal and Srikant, 1994]

Agrawal, R. and Srikant, R. "Fast Algorithms for Mining Association Rules." *In Proceedings of the 20th International Conference on Very Large Databases*, pages 487-499, Santiago, Chile, September 1994.

[Agrawal *et al.*, 1996]

Agrawal, R., Mannila, H., Srikant, R., Toivonen, H. and Verkamo, A.I. "Fast Discovery of Association Rules." *In Advances in Knowledge Discovery and Data Mining*, eds Fayyad, U., Piatetsky-Shapiro, G., Smyth, P. and Uthurusammy, R., pages 307-328, Menlo Park, California, AAAI Press, 1996.

[Agrawal and Shafer, 1996]

Agrawal, R. and Shafer, J.C. "Parallel Mining of Association Rules." *In IEEE Transactions on Knowledge and Data Engineering,* Vol. 8, No. 6, pages 962-969, December 1996.

[Agarwal *et al.*, 2000]

Agarwal, R.C., Aggarwal, C.C. and Prasad, V.V.V. "Depth First Generation of Long Patterns." *In Proceedings of the sixth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining,* pages 108-118, Boston, USA, 2000.

[Ahmed *et al.*, 2003]

Ahmed, S., Coenen, F. and Leng, P. "Strategies for Partitioning Data in Association Rule Mining." *In Research and Development in Intelligent Systems XX (Proceedings of AI2003, the Twenty-third SGAI International Conference on Innovative Techniques and Applications of Artificial Intelligence, Cambridge, 15-17 December 2003),* eds F Coenen, A Preece and A Macintosh, Springer-Verlag, London, pages 127-139, 2004.

[Ahmed *et al.*, 2004]

Ahmed, S., Coenen, F. and Leng, P. "A Tree Partitioning Method for Memory Management in Association Rule Mining." *In Proceedings of the Sixth International Conference on Data Warehousing and Knowledge Discovery (DaWaK 2004),* LNCS 3181, pages 331-340, Zaragoza, Spain, 1-3 September 2004.

[Arnold *et al.*, 1999]

Arnold, K., Freeman, E. and Hupfer, S. *JavaSpaces: Principles, Patterms and Practice,* Addision Wesley, 1999.

[Bayardo, 1998]

Bayardo, R.J. "Efficiently Mining Long Patterns from Databases." *In Proceedings of the ACM SIGMOD Conference on Management of Data*, pages 85-93, Seattle, Washington, June 1998.

[Bayardo *et al.*, 1999]

Bayardo, R.J. Agrawal, R. and Gunopulos, D. "Constraint-Based Rule Mining in Large, Dense Databases". *In Proceedings of the 15th International Conference on Data Engineering*, pages 188-197, Sydney, Australia, March 1999.

[Bayardo and Agrawal, 1999]

Bayardo, R.J. and Agrawal, R. "Mining the most interesting rules." *In Proceedings of the 5th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 145-154, San Diego, California, August 1999.

[Berry and Linoff, 1997]

Berry, M.J. and Linoff, G.S. *Data Mining Techniques for Marketing, Sales and Customer Support*, John Wiley and sons, 1997.

[Bishop, 1995]

Bishop, C.M., *Neural networks for pattern recognition,* Oxford: New York: Clarendon Press, Oxford University Press, 1995.

[Brachman and Anand, 1996]

Brachman, R.J. and Anand, T. "The process of knowledge discovery in databases: A human centred approach." *In Advance in Knowledge Discovery and Data Mining*, eds

Fayyad, U., Piatetsky-Shapiro, G. and Smyth, P. and Uthurusammy, R. AAAI/MIT Press, 1996.

[Breiman, 1984]

Breiman. L., *Classification and regression trees,* Belmont, Calif.: Wadsworth International Group, 1984.

[Brin *et al.,* 1997]

Brin, S., Motwani, R., Ullman, J.D. and Tsur, S. "Dynamic Itemset Counting and Implication Rules for Market Basket Data." *In Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 255-264, Arizona, USA, May 1997.

[Bramer, 1999]

Bramer, M., editor. *Knowledge Discovery and Data Mining.*. Institute of Electrical Engineers, 1999.

[Carreiro and Galernter, 1989]

Carreiro, N. and Galernter, D. "Linda in Context." *In Communications of the ACM*, Vol. 32, No. 4, 1989.

[Chattratichat *et al.,* 1997]

Chattratichat, J., Darlington, J., Ghanem, M. and Guo, Y. ""Large Scale Data Mining: Challenges and Responses." *In Proceedings of the Third International Conference on Knowledge Discovery and Data Mining (KDD'97)*, pages 143-146. Newport, California, August 1997.

[Chen *et al.*, 1996]

Chen, M. Han, J. and Yu, P. S. "Data Mining: An Overview from a Database Perspective." *In IEEE Transactions on Knowledge and Data Engineering*, Vol. 8, No. 6, pages 866-883, 1996.

[Cheung *et al.*, 1996a]

Cheung, D.W., Ng, V.T., Fu, A.W. and Fu, Y. "Efficient Mining of Association Rules in Distributed Databases," *In IEEE Transactions on Knowledge and Data Engineering,* Vol. 8, No. 6, pages 911-922, December 1996.

[Cheung *et al.*, 1996b]

Cheung, D.W., Han, J., Ng, V.T., Fu, A.W. and Fu, Y. "A Fast Distributed Algorithm for Mining Association Rules," *In Proceedings of International Conference on Parallel and Distributed Information Systems (PDIS'96)*, pages 31-42, 1996.

[Cheung and Xiao, 1998]

Cheung, D.W., and Xiao, Y. "Effect of data skewness in parallel mining of association rules." *In Proceedings of the 2nd Pacific-Asia Conference on Knowledge Discovery and Data Mining (PAKDD-98),* pages 48-60, Melbourne, Australia, April 1998.

[Cheung and Zaiane, 2003]

Cheung, W. and Zaiane, O.R. "Incremental Mining of Frequent Patterns without Candidate Generation or Support Constraint." *In Proceedings of the Seventh International Database Engineering and Applications Symposium (IDEAS'03),* pages 111-116, Hong Kong, July 2003.

[Coenen *et al.*, 2001]

Coenen, F., Goulbourne, G. and Leng, P. "Computing Association Rules using Partial Totals." *In Principles of Data Mining and Knowledge Discovery (Proceedings of PKDD 2001 Conference)*, eds L de Raedt and A Siebes: Lecture Notes in AI 2168, pages 54-66, Springer-Verlag, 2001.

[Coenen and Leng, 2001]

Coenen, F. and Leng, P. "Optimising Association rule Algorithms using Itemset Ordering." *In Research and Development in Intelligent Systems XVIII (Proceedings of ES2001 Conference, Cambridge, Dec 2001)*, eds M Bramer, F Coenen and A Preece, Springer-Verlag, London, pages 53-66, 2002.

[Coenen and Leng, 2002]

Coenen, F. and Leng, P. "Finding Association Rules with some very frequent attributes." *In Principles of Data Mining and Knowledge Discovery (Proceedings of PKDD 2002 Conference, Helsinki)*, eds T Elomaa, H Mannila and H Toivonen: Lecture Notes in AI 2431, pages 99-111, Springer-Verlag, 2002.

[Coenen *et al.*, 2003]

Coenen, F., Leng, P. and Ahmed, S. "T-Trees, Vertical Partitioning and Distributed Association Rule Mining." *In the Third IEEE International Conference on Data Mining (ICDM 2003)*, pages 513-516, Melbourne, Florida, 19-22 November 2003.

[Coenen *et al.*, 2004a]

Coenen, F., Goulbourne, G. and Leng, P. "Tree structures for mining association rules." *In Journal of Data Mining and Knowledge Discovery*, Vol. 15, No. 7, pages 391-398, 2004.

[Coenen *et al.*, 2004b]

Coenen, F., Leng, P. and Ahmed, S. "Data Structure for Association Rule Mining: T-Trees and P-Trees." *In IEEE Transactions on Knowledge and Data Engineering,* Vol. 16, No. 6, pages 774-778, June 2004.

[Dunham, 2003]

Dunham, MH. *Data Mining: Introductory and Advanced Topics.* Prentice Hall, Person Education Inc., New Jersey, 2003.

[Debuse *et al.*, 2000]

Debuse, J.C.W. de la Iglesia, B., Howard, C.M. and Rayward-Smith, V.J. "Building the KDD Roadmap: A methodology for Knowledge Discovery." *Industrial Knowledge Management*, ed Roy R., Springer-Verlag, Lindon, 2000.

[El-Hajj and Zaiane, 2003]

El-Hajj, M. and Zaiane, O.R. "Non Recursive Generation of Frequent K-itemsets from Frequent Pattern Tree Representations." *In Proceedings of the 5th International Conference on Data Warehousing and Knowledge Discovery (DaWaK 2003),* pages 371-380, Prague, Czech Republic, September 2003.

[Fayyad *et al.*, 1996a]

Fayyad, U., Piatetsky-Shapiro, G. and Smyth, P. "Knowledge Discovery and Data Mining: Towards a Unifying Framework". *In Proceedings of the Second International Conference on Knowledge Discovery and Data Mining (KDD-96),* pages 82-95, Portland, Oregon, August 1996.

[Fayyad *et al.*, 1996b]

Fayyad, U., Piatetsky-Shapiro, G. and Smyth, P. "The KDD Process for Extracting Useful Knowledge from Volumes of Data." *In Communications of the ACM,* Vol. 39, No. 11, pages 27-34, November 1996.

[Fayyad *et al.*, 1996c]

Fayyad, U., Piatetsky-Shapiro, G. and Smyth, P. (1996). "From Data Mining to Knowledge Discovery in Databases." *In AI Magazine*, Vol. 17, No. 3, pages 37-54, AAAI Press, 1996.

[Fayyad *et al.*, 1996d]

Fayyad, U., Piatetsky-Shapiro, G. and Smyth, P. "From Data Mining to Knowledge Discovery: An Overview." *In Advance in Knowledge Discovery and Data Mining*, eds Fayyad, U., Piatetsky-Shapiro, G. and Smyth, P. and Uthurusammy, R. pages 1-34, AAAI Press, 1996.

[Frawely *et al.*, 1992]

Frawely, W.J, Piatetsky-Shapiro, G. and Matheus, C.J. "Knowledge Discovery in Databases: An Overview." *In AI Magazine*, pages 57-70, AAAI Press, 1992.

[Ganti *et al.*, 1999]

Ganti, V. Gehrke, J. Ramakrishnan, R. "Mining Very Large Databases." *In Computer,* Cover Feature, pages 38-45, IEEE Press, 1999.

[Goldberg, 1989]

Goldberg. D.G., *Genetic algorithms in search, optimization, and machine learning,* Reading, Mass. : Addison-Wesley Pub. Co., 1989.

[Gonnet and Baeza-Yates, 1991]

Gonnet. G.H. and Baeza-Yates, R. *Handbook of Algorithms and Data Structures: In Pascal and C,* second edition, Addision Wesley, 1991]

[Goulbourne *et al.*, 2000]

Goulbourne, G., Coenen, F. and Leng, P. "Algorithms for Computing Association Rules using a Partial-Support Tree." *In Journal of Knowledge-Based Systems,* Vol. 13, pages 141-149, 2000. (also Proc ES'99.)

[Han *et al.*, 1997]

Han, E.H, Karypis, G. and Kumar, V. "Scalable Parallel Data Mining for Association Rules." *In Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 277-288, Arizona, USA, May 1997.

[Han *et al.*, 2000]

Han, J., Pei, J. and Yin, Y. "Mining Frequent Patterns without Candidate Generation." *In Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 1-12, Dallas, USA, May 2000.

[Hand, 1981]

Hand, D.J. *Discrimination and Classification.* Wiley, Chester, UK, 1981.

[Hipp *et al.*, 2000]

Hipp, J., Guntzer, U. and Nakhaeizadeh, G. "Algorithms for Association Rule Mining: A General Survey and Comparison." *In SIGKDD Explorations,* Vol. 2, No. 1, pages 58-64, July 2000.

[Jain and Dubes, 1988]

Jain, A.K. and Dubes, R.C. *Algorithms for Clustering Data.* Prentice-Hall, Englewood, Callif, NJ, 1988.

[Joliffe, 1994]

Joliffe, I. *Principal Component Analysis.* New York: Springer Verlag, 1986.

[Kirk, 1965]

Kirk. H.W., "Use of decision tables in computer programming", *Communications of the ACM,* Vol. 8, Issue 1, pages 41-43, January 1965.

[Klemettinen *et al.,* 1994]

Klemettinen, M., Mannila, H., Ronkainen, P., Toivonen, H. and Verkamo, A.I. "Finding Interesting Rules From Large Sets of Discovered Association Rules." *In Proceedings of the 3rd International Conference on Information and Knowledge Management (CIKM'94),* pages 401-407, November, 1994.

[Lin and Dunham, 1998]

Lin, J.L. and Dunham, M.H. "Mining Association Rules: Anti-Skew Algorithms." *In Proceedings of the IEEE International Conference on Data Engineering,* pages 486-493, February 1998.

[Mannila *et al.,* 1994]

Mannila, H., Toivonen, H. and Verkamo, A.I. "Efficient Algorithms for Discovering Association Rules." *In Proceedings of the AAAI Workshop on Knowledge Discovery in Databases (KDD-94),* pages 181-192, Seattle, Washington, July 1994.

[Manning and Keane, 2001]

Manning, A.M., Keane, J.A. "Data Allocation Algorithm for Parallel Association Rule Discovery." *In Proceedings of the 5th Pacific-Asia Conference on Knowledge Discovery and Data Mining (PAKDD 2001),* pages 413-420, Hong Kong, April 2001.

[Park *et al.,* 1995]

Park, J.S., Chen, M. Yu, P.S. "An Effective Hash Based Algorithm for Mining Association Rules." *In Proceedings of the ACM SIGMOD International Conference on Management of Data,* pages 175-186, San Jose, California, May 1995.

[Pasquier *et al.,* 1999]

Pasquier, N., Bastide, Y., Taouil, R. and Lakhal, L. "Discovering Frequent Closed Itemsets for Association Rules." *In International Conference on Database Theory (ICDT'99),* Number 1540 of LNCS, pages 398-416, 1999.

[Pei *et al.,* 2000]

Pei, J., Han, J. and Mao, R. "CLOSET: An Efficient Algorithm for Mining Frequent Closed Itemsets." *in Proceedings of ACM SIGKDD Workshop on Knowledge Discovery and Data Mining,* pages 11-20, Boston, USA, August 2000.

[Pothering and Naps, 1995]

Pothering, G.J. and Naps, T.L. *Introduction to Data Structures and Algorithm Analysis with C++,* West, 1995.

[Richards *et al.*, 2001]

Richards, G. Rayward-Smith, V.J. Sonksen, P.H. Caey, S. and Weng, C. " Data mining for indicators of early mortality in a database of clinical records." *In Artificial Intelligence in Medicine*, Vol. 22, No. 3, pages 215-231, Elsevier, 2001.

[Rymon, 1992]

Rymon, R. "Search through systematic set enumeration". *In Proceedings of the 3rd International Conference on Principles of Knowledge Representation and Reasoning,* pages 539-550, Morgan Kaufman, 1992.

[Savasere *et al.*, 1995]

Savasere, A., Omiecinski, E. and Navathe, S. "An Efficient Algorithm for Mining Association Rules in Large Databases." *In Proceedings of the 21st International Conference on Very Large Databases,* pages 432-444, Zurich, Swizerland, September 1995.

[Seno M. and Karypis G.]

Seno, M. and Karypis, G. "LPMiner: An Algorithm for Finding Frequent Itemsets Using Length-Decreasing Support Constraint." *In Proceedings of the First IEEE Conference on Data Mining (ICDM 2001),* pages 505-512, San Jose, California, USA, 2001.

[Silberschatz and Tuzhillin, 1995]

Silberschatz, A. and Tuzhillin, A. "On Subjective Measures of Interestingness in Knowledge Discovery". *In Proceedings of the First International Conference on Knowledge Discovery and Data Mining (KDD-95),* pages 275-281, Montreal, Canada, August 1995.

[Skillicorn, 1999]

Skillicorn, D. "Strategies for Parallel Data Mining." In IEEE Concurrency, pages 26-35, October-December 1999.

[Srikant and Agrawal, 1995]

Srikant, R. and Agrawal, R. "Mining Generalized Association Rules." *In Proceedings of the 21st International Conference on Very Large Databases*, pages 407-419, Zurich, Swizerland, September 1995.

[Srikant and Agrawal, 1996]

Srikant, R. and Agrawal, R. "Mining Quantitative Association Rules in Large Relational Tables." *In Proceedings of the ACM SIGMOD Conference on Management of Data*, Montreal, Canada, June 1996.

[Srikant *et al.*, 1997]

Srikant, R., Vu, Q. and Agrawal, R. "Mining Association Rules with Item Constraints." *In Proceedings of the Third International Conference on Knowledge Discovery and Data Mining (KDD'97)*, Newport, California, August 1997: AAAI Press.

[Titterington *et al.*, 1985]

Titterington, D.M. Smith, A.F.M. and Makov, U.E. *Statistical Analysis of Finite-Mixture Distributions*. Wiley, Chichester, UK, 1985.

[Toivonen, 1996]

Toivonen, H. "Sampling Large Databases for Association Rules." *In Proceedings of the 22nd* International Conference on Very Large Databases, Mumbai, India, pages 1-12, 1996.

[Wyatt and Altman, 1995]

Wyatt, J.C. and Altman, D.G. "Prognostic Models: Clinically useful or Quickly Forgotten?" *BMJ 311*, pages 1539-1541, 1995.

[Weiss and Kulikowski, 1991]

Weiss, S. and Kulikowski, C. *Computer System That Learn: Classification and Prediction Methods from Statistics, Neural Networks, Machine Learning and Expert Systems.* San Francisco, Calif.: Morgan Kaufmann, 1991.

[Weiss and Indurkhya, 1998]

Weiss, S. and Indurkhya. *Predictive Data Mining, a practical guide.* Morgan Kaufmann, 1998.

[Xiao and Dunham, 1999]

Xiao, Y. and Dunham, M.H. "Considering Main Memory in Mining Association Rules." *In Proceedings of the First International Conference on Data Warehousing and Knowledge Discovery (DaWak 1999)*, pages 209-218, September 1999.

[Zaki *et al.*, 1996]

Zaki, M.J., Parthasarathy, S., Li, W. and Ogihara, M. "Evaluation of Sampling for Data Mining of Association Rules." *Technical Report: TR617*, University of Rochester, NY, USA, 1996.

[Zaki *et al.*, 1997]

Zaki, M.J. Parthasarathy, S., Ogihara, M. and Li, W. "New Algorithms for Fast Discovery of Association Rules." *In Proceedings of the Third International Conference*

on Knowledge Discovery and Data Mining (KDD'97), pages 283-286. Newport, California, August 1997.

[Zaki, 1999]

Zaki, M. J. "Parallel and Distributed Association Mining: A Survey." *In IEEE Concurrency*, Vol. 7, No. 4, pages 14-25, December 1999.

[Zaki, 2000]

Zaki, M.J. "Scalable Algorithms for Association Mining." *In IEEE Transactions on Knowledge and Data Engineering*, Vol. 12, No. 3, pages 372-390, May/June 2000.

[Zomaya *et al.*, 1999]

Zomaya, A.Y., El-Ghazawi, T. and Frieder, O. "Parallel and Distributed Computing for Data Mining." *In IEEE Concurrency*, pages 11-13, December 1999.

Appendix A

The following tables detail the results for the experiments described in the Chapter 5.

A.1 Results for increasing Size of Databases

	Number of records (x 50,000)				
	1	*2*	*3*	*4*	*5*
DP/NB	2.02	4.35	6.56	8.89	10.95
TP	151.77	302.85	457.95	613.18	770.4
DTP	13.65	28.05	41.26	54.44	67.46

Table A.1: Time (seconds) to construct P/PP-trees for increasing size of databases

	Number of records (x 50,000)				
	1	*2*	*3*	*4*	*5*
DP/NB	1.85666	1.85712	1.85695	1.85703	1.85714
TP	0.06315	0.11657	0.16871	0.22073	0.27107
DTP	0.06315	0.12749	0.19276	0.25681	0.32078

Table A.2: Memory requirements (Mb) for largest P-tree/ Partition (T10.I5.N500)

	Number of records (x 50,000)				
	1	*2*	*3*	*4*	*5*
DP	56.98	87.12	120.14	153.43	182.41
NB	88.61	129.2	166.09	202.33	230.98
TP	14.98	22.14	29.07	36.62	43.46
DTP	15.03	22.35	30.16	38.56	46.38

Table A.3: Time (seconds) to find frequent sets for increasing size of databases (0.01% support)

	Number of records (x 50,000)				
	1	*2*	*3*	*4*	*5*
DP	126.965	117.143	115.029	114.606	114.151
NB	239.871	239.828	239.601	239.653	239.578
TP	1.31474	1.3105	1.30422	1.27952	1.27618
DTP	1.31474	1.3105	1.30422	1.27952	1.27618

Table A.4: Memory requirements (Mb) for T-tree/ largest PT-tree (T10.I5.N500)

	Number of records (x 50,000)				
	1	*2*	*3*	*4*	*5*
DP	59	91.47	126.7	162.32	193.36
NB	90.63	133.55	172.65	211.22	241.93
TP	166.75	324.99	487.02	649.8	813.86
DTP	28.68	50.4	71.42	93	113.84

Table A.5: Execution times (seconds) for T10.I5.N500 (0.01% support)

	Number of records (x 50,000)				
	1	*2*	*3*	*4*	*5*
DP	128.822	119	116.886	116.463	116.008
NB	241.728	241.685	241.458	241.51	241.435
TP	1.37789	1.42707	1.47293	1.50025	1.54725
DTP	1.37789	1.43799	1.49698	1.53633	1.59696

Table A.6: Memory requirements (Mb) for T10.I5.N500

A.2 Results for increasing Number of Segments

	Number of horizontal segments				
	1	*2*	*5*	*10*	*50*
DP/NB	2.02	1.99	1.91	1.7	1.61
DTP	8.39	7.66	7.05	6.8	6.5

Table A.7: Time (seconds) to construct P/PP-trees for increasing number of segments

	Number of horizontal segments				
	1	*2*	*5*	*10*	*50*
DP/NB	1.85666	0.96884	0.40817	0.21013	0.04673
DTP	1.21409	0.63561	0.26907	0.14139	0.03271

Table A.8: Memory requirements (Mb) of largest P-tree/ Partition (T10.I5.N500.D50000)

	Number of horizontal segments				
	1	*2*	*5*	*10*	*50*
DP	56.98	58.71	61.33	62.55	65.7
NB	88.61	overflow	overflow	overflow	overflow
DTP	31.25	32.12	33.34	33.99	35.2

Table A.9: Time (seconds) to find frequent sets for increasing number of segments

(0.01% support)

	Number of horizontal segments				
	1	*2*	*5*	*10*	*50*
DP	126.965	126.965	126.965	126.965	126.96
NB	239.871	overflow	overflow	overflow	overflow
DTP	23.4591	23.4591	23.4591	23.4591	23.4591

Table A.10: Memory requirements (Mb) for T-tree/ largest PT-tree (T10.I5.N500.D50000)

	Number of horizontal segments				
	1	*2*	*5*	*10*	*50*
DP	59	60.7	63.24	64.25	67.31
NB	90.63	overflow	overflow	overflow	overflow
DTP	39.64	39.78	40.39	40.79	41.7

Table A.11: Effect on time (seconds) for increasing segmentation (0.01% support)

	Number of horizontal segments				
	1	*2*	*5*	*10*	*50*
DP	128.822	127.934	127.373	127.175	127.011
NB	241.728	overflow	overflow	overflow	overflow
DTP	24.6732	24.0947	23.7282	23.6005	23.4918

Table A.12: Effect on memory (Mb) for increasing number of segments

A.3 Results for increasing Number of Attributes

	Number of attributes (x 500)				
	1	*2*	*3*	*4*	*5*
DP/NB	1.85	1.93	1.72	1.82	1.88
TP	24.73	40.33	55.15	69.94	85.25
DTP	10.15	11	11.65	12.13	13.51

Table A.13: Time (seconds) to construct P/PP-trees for increasing number of attributes

	Number of attributes (x 500)				
	1	*2*	*3*	*4*	*5*
DP/NB	0.40817	0.41311	0.41902	0.42149	0.42521
TP	0.45856	0.27934	0.20347	0.16151	0.14419
DTP	0.53255	0.34443	0.26976	0.24164	0.23413

Table A.14: Memory requirements (Mb) for largest P-tree/ Partition (T10.I5.D50000)

	Number of attributes (x 500)				
	1	*2*	*3*	*4*	*5*
DP	61.59	58.37	66.87	overflow	overflow
TP	26.09	29.21	35.62	40.93	44.31
DTP	27.79	30.29	36.11	41.28	45.09

Table A.15: Time (seconds) to find frequent sets for increasing number of attributes

(0.01% support)

	Number of attributes (x 500)				
	1	*2*	*3*	*4*	*5*
DP	126.965	238.729	366.677	overflow	overflow
TP	6.03513	6.17467	7.58429	13.7864	8.32238
DTP	6.03513	6.17467	7.58429	13.7864	8.32238

Table A.16: Memory requirements (Mb) for T-tree/ largest PT-tree (T10.I5.D50000)

	Number of attributes (x 500)				
	1	*2*	*3*	*4*	*5*
DP	63.44	60.3	68.59	overflow	overflow
TP	50.82	69.54	90.77	110.87	129.56
DTP	37.94	41.29	47.76	53.41	58.6

Table A.17: Execution times (seconds) for T10.I5.D50000 (0.01% support)

	Number of attributes (x 500)				
	1	*2*	*3*	*4*	*5*
DP	127.373	239.142	367.096	overflow	overflow
TP	6.49369	6.45401	7.78776	13.9479	8.46657
DTP	6.56768	6.5191	7.85405	14.028	8.55651

Table A.18: Memory requirements (Mb) for T10.I5.D50000

A.4 Results for increasing Items/Partition

	Items/Partition				
	1	*2*	*3*	*4*	*5*
TP	143.69	77.28	55.51	44.2	37.87
DTP	15.48	13.39	12.2	12.16	11.6

Table A.19: Time (seconds) to construct PP-trees for increasing items/partition

	Items/Partition				
	1	*2*	*3*	*4*	*5*
TP	0.06315	0.117376	0.167728	0.218036	0.263818
DTP	0.088026	0.14973	0.208206	0.262288	0.31481

Table A.20: Memory requirements (Mb) for largest partition (T10.I5.N500.D50000)

	Items/Partition				
	1	*2*	*3*	*4*	*5*
TP	15.14	18.99	21.2	22.41	23.54
DTP	15.8	19.83	21.86	23.36	24.65

Table A.21: Time (seconds) to find frequent sets for increasing items/partition (0.01% support)

	Items/Partition				
	1	*2*	*3*	*4*	*5*
TP	1.31474	1.61194	2.11278	2.50216	3.26811
DTP	1.31474	1.61194	2.11278	2.50216	3.26811

Table A.22: Memory requirements (Mb) for largest partition (T10.I5.N500.D50000)

	Items/Partition				
	1	*2*	*3*	*4*	*5*
TP	158.83	96.27	76.71	66.61	61.41
DTP	31.28	33.22	34.06	35.52	36.25

Table A.23: Execution times (seconds) for T10.I5.N500.D50000 (0.01% support)

	Items/Partition				
	1	*2*	*3*	*4*	*5*
TP	1.37789	1.729316	2.280508	2.720196	3.531928
DTP	1.402766	1.76167	2.320986	2.764448	3.58292

Table A.24: Memory requirements (Mb) for T10.I5.N500.D50000

A.5 Results for increasing Density of Database

	Density of database			
	T10.I5	*T14.I7*	*T18.I9*	*T22.I11*
DP/NB	1.85	2.23	2.38	2.65
TP	24.73	33.42	42.87	51.73
DTP	10.15	16.86	23.1	28.72

Table A.25: Time (seconds) to construct P/PP-trees for increasing density

	Density of database			
	T10.I5	*T14.I7*	*T18.I9*	*T22.I11*
DP/NB	0.40817	0.48732	0.56254	0.6367
TP	0.45856	0.69009	0.90295	1.14112
DTP	0.53255	0.76749	0.9969	1.23048

Table A.26: Memory requirements (Mb) for largest P-tree/ Partition (N500.D50000)

	Density of database			
	T10.I5	*T14.I7*	*T18.I9*	*T22.I11*
DP	8.73	20.71	61.75	241.32
NB	10.87	63.36	241.99	overflow
TP	4.16	8.55	24.28	130.9
DTP	4.55	8.95	25.85	139.3

Table A.27: Time (seconds) to find frequent sets for increasing density (0.1% support)

	Density of database			
	T10.I5	*T14.I7*	*T18.I9*	*T22.I11*
DP	1.72479	5.13111	18.0669	96.7207
NB	19.0834	98.1602	357.289	overflow
TP	0.11697	0.28505	1.15173	10.6531
DTP	0.11697	0.28505	1.15173	10.6531

Table A.28: Memory requirements (Mb) for T-tree/ largest PT-tree (N500.D50000)

	Density of database			
	T10.I5	*T14.I7*	*T18.I9*	*T22.I11*
DP	10.58	22.94	64.13	243.97
NB	12.72	65.59	244.37	overflow
TP	28.89	41.97	67.15	182.63
DTP	14.7	25.81	48.95	168.02

Table A.29: Execution times (seconds) for N500.D50000 (0.1% support)

	Density of database			
	T10.I5	*T14.I7*	*T18.I9*	*T22.I11*
DP	2.13296	7.36111	20.4469	99.3707
NB	19.4916	100.39	359.669	overflow
TP	0.57553	0.97514	2.05468	11.7942
DTP	0.64952	1.05254	2.14863	11.8836

Table A.30: Memory requirements (Mb) for N500.D50000

A.6 Results for decreasing Support Thresholds

	Support threshold (%)				
	1.0	*0.5*	*0.1*	*0.05*	*0.01*
DP	1.91	2.04	8.62	18.19	54.03
NB	0.71	0.76	10.79	25.25	
TP	4.25	4.17	4.61	5.77	15.14
DTP	5.05	4.85	5.23	6.59	15.8

Table A.31: Time (seconds) to find frequent sets for decreasing support thresholds

	Support threshold (%)				
	1.0	*0.5*	*0.1*	*0.05*	*0.01*
DP	0.004104	0.032704	1.72479	16.5055	126.965
NB	0.019392	0.218112	19.0834	62.1182	overflow
TP	0.004104	0.005064	0.088144	0.601584	1.31474
DTP	0.004104	0.005064	0.088144	0.601584	1.31474

Table A.32: Memory requirements (Mb) for largest T-tree/Partition (T10.I5.N500.D50000)

	Support threshold (%)				
	1.0	*0.5*	*0.1*	*0.05*	*0.01*
DP	3.88	4.01	10.59	20.16	56
NB	2.68	2.73	12.76	27.22	overflow
TP	147.94	147.86	148.3	149.46	158.83
DTP	20.53	20.33	20.71	22.07	31.28

Table A.33: Performance of the methods (seconds) for decreasing support thresholds

	Support threshold (%)				
	1.0	*0.5*	*0.1*	*0.05*	*0.01*
DP	0.412278	0.440878	2.132964	16.91367	127.3732
NB	0.427566	0.626286	19.49157	62.52637	overflow
TP	0.067254	0.068214	0.151294	0.664734	1.37789
DTP	0.09213	0.09309	0.17617	0.68961	1.402766

Table A.34: Memory requirements (Mb) for T10.I5.N500.D50000

Appendix B

The following tables detail the results for the experiments described in the Chapter 6.

B.1 Amount of Data Sent and Received

	Support threshold (%)		
	2.0	*1.5*	*1.0*
DD	3472	4621	8855
TKD	695	924	1771
TD	11	21	60

Table B.1: Average Total size (Kbytes) of messages sent and read/taken per process

B.2 Number of Updates

	Support threshold (%)		
	2.0	*1.5*	*1.0*
Apriori-T	99	111	148
TD/DD/TKD	20	22	30

Table B.2: Average number of updates (x 10^6) to generate a final T-tree per process

B.3 Execution Time

	Support threshold (%)		
	2.0	*1.5*	*1.0*
Apriori-T	15	19	31
DD	13	16	25
TKD	9	9	16
TD	3	4	10

Table B.3: Average execution time (seconds) per process

Appendix C

IBM Quest Synthetic Data Generation Code

IBM Quest synthetic Data Generation Code can be used to find frequent itemsets with/without taxonomies. It can also be used to obtain sequential patterns. There are two possible output formats for the data file: 1) **Binary** <CustID, TransID, NumItems, List-Of-Items.> and 2) **Ascii** < CustID, TransID, and Item>.

Functionality: Finding Frequent Itemsets as well as sequential pattern

Vendor: IBM Almaden Research Center

Cost: Free to download

Language: C++

Platform: UNIX

URL: http://www.almaden.ibm.com/software/quest/Resources/index.shtml

Contact Info: srikant@almaden.ibm.com