NOTE TO USERS

This reproduction is the best copy available.

UMI®

EFFICIENTLY MINING FREQUENT ITEMSETS FROM VERY LARGE DATABASES

JIANFEI ZHU

A THESIS
IN
THE DEPARTMENT
OF
COMPUTER SCIENCE

PRESENTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF DOCTOR OF PHILOSOPHY
CONCORDIA UNIVERSITY
MONTRÉAL, QUÉBEC, CANADA

SEPTEMBER 2004
© JIANFEI ZHU, 2004

Library and
Archives Canada

Published Heritage
Branch

395 Wellington Street
Ottawa ON K1A 0N4
Canada

Bibliothèque et
Archives Canada

Direction du
Patrimoine de l'édition

395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence
ISBN: 0-612-96957-6
Our file Notre référence
ISBN: 0-612-96957-6

The author has granted a non-exclusive license allowing the Library and Archives Canada to reproduce, loan, distribute or sell copies of this thesis in microform, paper or electronic formats.

L'auteur a accordé une licence non exclusive permettant à la Bibliothèque et Archives Canada de reproduire, prêter, distribuer ou vendre des copies de cette thèse sous la forme de microfiche/film, de reproduction sur papier ou sur format électronique.

The author retains ownership of the copyright in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

L'auteur conserve la propriété du droit d'auteur qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou aturement reproduits sans son autorisation.

In compliance with the Canadian Privacy Act some supporting forms may have been removed from this thesis.

Conformément à la loi canadienne sur la protection de la vie privée, quelques formulaires secondaires ont été enlevés de cette thèse.

While these forms may be included in the document page count, their removal does not represent any loss of content from the thesis.

Bien que ces formulaires aient inclus dans la pagination, il n'y aura aucun contenu manquant.

Canada

Abstract

Efficiently Mining Frequent Itemsets from Very Large Databases

Jianfei Zhu, Ph.D.

Concordia University, 2004

Efficient algorithms for mining frequent itemsets are crucial for mining association rules and for other data mining tasks. Methods for mining frequent itemsets and for iceberg data cube computation have been implemented using a prefix-tree structure, known as a FP-tree, for storing compressed frequency information. Numerous experimental results have demonstrated that these algorithms perform extremely well. In this thesis we present a novel FP-array technique that greatly reduces the need to traverse FP-trees, thus obtaining significantly improved performance for FP-tree based algorithms. The technique works especially well for sparse datasets. We then present new algorithms for mining all frequent itemsets, maximal frequent itemsets, and closed frequent itemsets. The algorithms use the FP-tree data structure in combination with the FP-array technique efficiently, and incorporate various optimization techniques. In the algorithm for mining maximal frequent itemsets, a variant FP-tree data structure, called a MFI-tree, and an efficient maximality-checking approach are used. Another variant FP-tree data structure, called a CFI-tree, and an efficient closedness-testing approach are also given in the algorithm for mining closed frequent itemsets. Experimental results show that our methods outperform the existing methods in not only the speed of the algorithms, but also their memory consumption and their scalability. We also notice that most algorithms for mining frequent itemsets assume that the main memory is large enough for the data structures used in the mining, and very few efficient algorithms deal with the cases when the database is *very* large or the minimum support is very low. We thus investigate approaches to mining frequent itemsets when data structures are too large to fit in main memory. Several divide-and-conquer algorithms are presented for mining from disks. Many novel techniques are introduced. Experimental results show that the techniques reduce the required disk accesses by orders of magnitude, and enable truly scalable data mining.

Acknowledgments

I would like to express my gratitude to all those who made it possible for me to complete this thesis. I want to thank the Computer Science Department for giving me permission to commence this thesis in the first instance, and to do the necessary research work. I thank Prof. Dr. L. V. S. Lakshmanan, my former co-supervisor and now the professor of University of British Columbia, who gave me financial support for two years, supervised me and helped me so much. I learned a lot from him and he was very kind to me.

I am deeply indebted to Prof. Dr. G. Grahne who is both my supervisor and my friend. As a supervisor, he and Dr. Lakshmanan brought me from China, which totally changed my life. His help, stimulating suggestions and encouragement helped me during the time of my research and writing of this thesis. As a friend, he has suffered due to my English patiently without any complaints and has always been nice to me.

My officemates and friends, Ritesh Mukherjee, Victoria Kiricenko and Alina Andreevskaia supported me in my research work, being especially helpful with my English style and grammar. I am very grateful for all their assistance. I also want to thank all my friends. Especially Dr. Jianlong Lin, who encouraged and helped me to apply for the Ph.D. program in the Department of Computer Science in Concordia University.

Finally, I would like to give my special thanks to my parents who gave birth to me and raised me for a long and difficult time, my wife Haixin Xia whose patient love enabled me to complete this work, and my daughter Michelle who gave me much happiness and encouraged me to go ahead.

Contents

List of Figures viii

List of Tables xii

1 Introduction 1

1.1 Motivation . . . 2

1.2 Contributions . . . 4

1.3 Organization of this thesis . . . 5

2 Problem Definition and Related Work 7

2.1 Problem Definition . . . 7

2.2 Related Work . . . 10

2.2.1 Mining All Frequent Itemsets . . . 10

2.2.2 Mining MFI's . . . 15

2.2.3 Mining CFI's . . . 18

2.3 Mining frequent itemsets in very large databases . . . 19

3 Discovering All Frequent Itemsets 21

3.1 FP-array technique . . . 21

3.2 Discussion . . . 23

3.3 FPgrowth* : an improved FP-growth method . . . 24

3.4 Experimental evaluation and performance study . . . 25

3.4.1 FPgrowth* versus FPgrowth . . . 25

3.4.2 FPgrowth* versus other algorithms . . . 27

4 **Discovering Maximal Frequent Itemsets: First Attempt** **37**
4.1 The MFI-Tree . . . 38
4.2 FPmax: Discovering maximal frequent itemsets . . . 40
4.3 Maximality checking . . . 42
4.4 Data characteristics and performance . . . 43
4.4.1 Experiments on synthetic data . . . 45
4.4.2 Experiments on real datasets . . . 47
4.4.3 Scalability of the algorithms . . . 49

5 **Discovering Maximal Frequent Itemsets: Second Attempt** **51**
5.1 FPmax*: Mining MFI's . . . 51
5.2 Maximality checking . . . 53
5.3 An optimization . . . 54
5.4 Discussion . . . 54
5.5 Experimental evaluation . . . 56
5.5.1 FPmax* versus FPmax, MAFIA and GenMax . . . 56
5.5.2 FPmax* versus other algorithms . . . 58

6 **Discovering Closed Frequent Itemsets** **65**
6.1 The CFI-tree and algorithm FPclose . . . 65
6.2 Performance study . . . 68

7 **Mining Frequent Itemsets from Secondary Memory** **76**
7.1 Strategies for mining frequent itemsets from disk . . . 77
7.2 Algorithm Diskmine . . . 84
7.2.1 Divide-and-conquer by aggressive projection . . . 84
7.2.2 Memory management . . . 87
7.2.3 Applying the FP-array technique . . . 88
7.2.4 Statistics . . . 89
7.2.5 Grouping items . . . 90
7.2.6 Database projection . . . 93
7.2.7 The disk I/O's . . . 93

7.3 Experimental Results 94

8 Conclusions and Future Work **99**

8.1 Summary of this thesis 99

8.2 Future work 101

Bibliography **103**

List of Figures

2.1 An itemset lattice . . . 8

2.2 A database example . . . 9

2.3 Algorithm Apriori . . . 11

2.4 An Example FP-tree . . . 13

2.5 Algorithm FP-growth . . . 14

2.6 The bitmap representation and the depth-first search . . . 16

3.1 Two FP-array examples . . . 22

3.2 Algorithm FPgrowth* . . . 25

3.3 FPgrowth* vs. FPgrowth on sparse datasets . . . 26

3.4 FPgrowth* vs. FPgrowth on dense datasets . . . 27

3.5 Runtime of Mining All FI's on *T20I10N1KP5KC0.25D200K* . . . 29

3.6 Runtime of Mining All FI's on *T100I20N1KP5KC0.25D200K* . . . 29

3.7 Runtime of Mining All FI's on *chess* . . . 30

3.8 Runtime of Mining All FI's on *connect* . . . 30

3.9 Runtime of Mining All FI's on *mushroom* . . . 31

3.10 Runtime of Mining All FI's on *kosarak* . . . 31

3.11 Runtime of Mining All FI's on *accidents* . . . 31

3.12 Runtime of Mining All FI's on *pumsb** . . . 31

3.13 Memory Usage of Mining All FI's on *T20I10N1KP5KC0.25D200K* . . . 33

3.14 Memory Usage of Mining All FI's on *T100I20N1KP5KC0.25D200K* . . . 33

3.15 Memory Usage of Mining All FI's on *chess* . . . 34

3.16 Memory Usage of Mining All FI's on *connect* . . . 34

3.17 Memory Usage of Mining All FI's on *mushroom* 34
3.18 Memory Usage of Mining All FI's on *kosarak* 34
3.19 Memory Usage of Mining All FI's on *accidents* 34
3.20 Memory Usage of Mining All FI's on *pumsb** 34
3.21 Scalability of runtime of Mining All FI's 36
3.22 Scalability of Memory Usage of Mining All FI's 36
4.1 Construction of a MFI-tree 38
4.2 Construction of Maximal Frequent Itemset Tree 40
4.3 Algorithm FPmax 41
4.4 ATL=20, APL=20 45
4.5 ATL=20, APL=100 45
4.6 ATL=100, APL=20 46
4.7 ATL=100, APL=100 46
4.8 ATL=20, minimum support=1% 47
4.9 APL=20, minimum support=1% 47
4.10 Best algorithms for different type of data. 47
4.11 Running FPmax, GenMax and MAFIA on dataset *mushroom* 48
4.12 Running FPmax, GenMax and MAFIA on dataset *chess* 48
4.13 Running FPmax, GenMax and MAFIA on dataset *connect* 48
4.14 Running FPmax, GenMax and MAFIA on dataset *pumsb** 48
4.15 Scalability of FPmax, GenMax and MAFIA Running on Synthetic Datasets 49
4.16 Scalability of FPmax, GenMax and MAFIA on Duplicated Real Datasets 49
5.1 Algorithm FPmax* 52
5.2 Size-reduced Maximal Frequent Itemset Tree 56
5.3 dataset *T40I10D100K* 57
5.4 dataset *T100I20D100K* 57
5.5 dataset *pumsb** 57
5.6 dataset *connect* 57
5.7 Runtime of Mining Maximal FI's on *T20I10N1KP5KC0.25D200K* 59
5.8 Runtime of Mining Maximal FI's on *T100I20N1KP5KC0.25D200K* 59

5.9 Runtime of Mining Maximal FI's on *Chess* 60
5.10 Runtime of Mining Maximal FI's on *Connect* 60
5.11 Runtime of Mining Maximal FI's on *Mushroom* 60
5.12 Runtime of Mining Maximal FI's on *Kosarak* 60
5.13 Runtime of Mining Maximal FI's on *Accidents* 60
5.14 Runtime of Mining Maximal FI's on *Pumsb** 60
5.15 Memory Consumption of Mining Maximal FI's on *T20I10N1KP5KC0.25D200K* . . . 61
5.16 Memory Consumption of Mining Maximal FI's on *T100I20N1KP5KC0.25D200K* . . 61
5.17 Memory Consumption of Mining Maximal FI's on *Chess* 62
5.18 Memory Consumption of Mining Maximal FI's on *Connect* 62
5.19 Memory Consumption of Mining Maximal FI's on *Mushroom* 62
5.20 Memory Consumption of Mining Maximal FI's on *Kosarak* 62
5.21 Memory Consumption of Mining Maximal FI's on *Accidents* 62
5.22 Memory Consumption of Mining Maximal FI's on *Pumsb** 62
5.23 Scalability of runtime of Mining Maximal FI's 64
5.24 Scalability of Memory Consumption of Mining Maximal FI's 64
6.1 Construction of Closed Frequent Itemset Tree 66
6.2 Algorithm FPclose 67
6.3 Size-reduced Closed Frequent Itemset Tree 69
6.4 Runtime of Mining Closed FI's on *T20I10N1KP5KC0.25D200K* 70
6.5 Runtime of Mining Closed FI's on *T100I20N1KP5KC0.25D200K* 70
6.6 Runtime of Mining Closed FI's on *Chess* 71
6.7 Runtime of Mining Closed FI's on *Connect* 71
6.8 Runtime of Mining Closed FI's on *Mushroom* 71
6.9 Runtime of Mining Closed FI's on *Kosarak* 71
6.10 Runtime of Mining Closed FI's on *Accidents* 72
6.11 Runtime of Mining Closed FI's on *Pumsb** 72
6.12 Memory Consumption of Mining Closed FI's on *T20I10N1KP5KC0.25D200K* 73
6.13 Memory Consumption of Mining Closed FI's on *T100I20N1KP5KC0.25D200K* . . . 73
6.14 Memory Consumption of Mining Closed FI's on *Chess* 73

6.15 Memory Consumption of Mining Closed FI's on *Connect* . . . 73
6.16 Memory Consumption of Mining Closed FI's on *Mushroom* . . . 74
6.17 Memory Consumption of Mining Closed FI's on *Kosarak* . . . 74
6.18 Memory Consumption of Mining Closed FI's on *Accidents* . . . 74
6.19 Memory Consumption of Mining Closed FI's on *Pumsb** . . . 74
6.20 Scalability of runtime of Mining Closed FI's . . . 75
6.21 Scalability of Memory Consumption of Mining Closed FI's . . . 75
7.1 General divide-and-conquer algorithm for mining frequent itemsets from disk. . . . 77
7.2 A simple divide-and-conquer algorithm for mining frequent itemsets from disk . . . 81
7.3 Recurrence structure of Basic Projection . . . 82
7.4 A more aggressive divide-and-conquer algorithm for mining frequent itemsets from disk . . . 83
7.5 Recurrence structure of Aggressive Projection . . . 84
7.6 Algorithm Diskmine . . . 85
7.7 Trial main memory mining algorithm . . . 85
7.8 Main memory mining algorithm . . . 86
7.9 The FP-array A_α . . . 89
7.10 Cutpoint . . . 92
7.11 Performance of algorithms running on synthetic dataset: Runtime . . . 95
7.12 Performance of algorithms running on synthetic dataset: Time for Disk I/O's . . . 96
7.13 Performance of algorithms running on synthetic dataset: CPU time . . . 96
7.14 Performance of algorithms running on real dataset: Runtime . . . 96
7.15 Performance of algorithms running on real dataset: Time for Disk I/O's . . . 97
7.16 Performance of algorithms running on real dataset: CPU time . . . 97
7.17 Estimation Accuracy . . . 98
7.18 Scalability of Diskmine . . . 98

List of Tables

3.1 Dataset Characteristics 28

7.1 Statistics from the partial FP-tree T_α 90

Chapter 1

Introduction

Today we all suffer from information overload. The amount of information available to us is so overwhelming that it hinders rather than helps us. Computers were supposed to alleviate our lives by providing and handling information, but they are only throwing massive amounts of data "in our face". The Internet, that was supposed to be panacea, has caused the "information glut". Not only do we face the problem of finding relevant information, but thanks to the information glut we do not even know what to look for. On the other hand, the information glut is also a potential goldmine, since terabytes of raw data are ubiquitously being recorded in commerce, science, and government. For instance, a medium sized business or a huge company can easily collect a few gigabytes of data each year. However, this data risks becoming a "data graveyard", unless computerized methods are developed to extract the knowledge inherent in the morass.

Computer Science, Statistics, Machine Learning, and many other fields already had some equipment for knowledge extraction in their scientific toolboxes. However, these tools were not developed for the data morass, for instance most algorithms assume that data sets are small enough for main memory processing. The systematic, scientific adaptation and development of the toolbox constitute the emerging discipline of Data Mining.

There are many successful applications of data mining. Here are some examples.

1. Market basket data. A rule in basket data could be that 98% of customers that purchase tires and auto accessories also get automotive services done. Finding all such rules is valuable for cross-marketing and attached mailing applications.

2. In a web access database at a popular site, an object is a web user and an attribute is a web page. Finding sequences of most frequently accessed pages of that site is useful for restructuring the web-site, or dynamically inserting relevant links in web pages based on user access patterns.

3. Decision trees can be constructed from bank-loan histories to produce algorithms to decide whether to grant a loan.

4. Comparison of the genotype of people with/without a condition allowed the discovery of a set of genes that together account for many cases of diabetes. This sort of mining will become much more important now that the human genome has been constructed.

Besides database researchers, data mining is also noticed by researchers of many other subareas. The researchers of statistics, machine learning, clustering algorithms, and visualization all have laid claim to this subject. As database researchers, we concentrate on the challenges that appear when the data is large and the computations are complex. That means, data mining can be thought of as algorithms for extracting interesting patterns and discovering valuable rules from very large databases.

What are the interesting patterns in large databases? Over the years, researchers have studied various problems related to mining interesting patterns from large databases. For example, association rules [6, 7, 37, 14, 58], causality [49], multidimensional patterns [31, 36], correlation patterns [11, 17, 18], sequential patterns [8], episodes [38], classification [47], clustering [62], patterns with constraints [19, 39, 34] and so on. In this thesis, we focus on mining frequent itemsets from large databases, which is the core of mining association rules. Novel techniques and new algorithms will be proposed.

1.1 Motivation

The association rule mining is motivated by problems such as market basket analysis. A tuple in a market basket database is a set of items purchased by a customer in a transaction. An association rule mined from market basket database states that if some items are purchased in a transaction, then it is likely that some other items are purchased as well. Finding all such rules is valuable for guiding future sales promotions and store layout.

An itemset is a set of items in the database. The support of an itemset is the percentage of the transactions in the database that contain the itemset. An itemset is called frequent if the support of the itemset is greater than a user defined threshold.

The core of mining association rule is to efficiently mine frequent itemsets [6, 7, 37, 14, 58]. Once we get all frequent itemsets, the generation of all association rule is straightforward. Efficiently mining frequent itemsets also plays an important role in some other data mining tasks such as sequential patterns, episodes, multi-dimensional patterns and so on [8, 38, 31]. In addition, frequent itemsets are one of the key abstractions in data mining.

An important property of frequent itemset is "anti-monotonicity" [34], which means any subset of a frequent itemset is frequent. When a transaction database is very dense, i.e. when the database contains large number of long frequent itemsets, mining *all* frequent itemsets might not be a good idea. For example, if there is a frequent itemset with size ℓ, then all 2^ℓ nonempty subsets of the itemset have to be generated. Sometimes we only want to know whether an itemset is frequent or not. Then it is sufficient to discover only all the *maximal frequent itemsets* (MFI's). A frequent itemset is maximal if all its supersets are not frequent. Therefore, many of the existing algorithms only mine maximal frequent itemsets.

The deficiency of mining only MFI's, from a MFI and its support δ, is that we only know that all its subsets are frequent and the support of any of its subset is not less than δ. We do not know the exact value of the support. While for generating association rules, we do need the support of all frequent itemsets. To solve this problem, another type of a frequent itemset, the *Closed Frequent Itemset* (CFI), was proposed. A frequent itemset is closed if all its supersets have less support. In most cases, though, the number of CFI's is greater than the number of MFI's, but still far less than the number of all FI's.

There are many factors that determine the efficiency of an algorithm for mining frequent itemsets. In this thesis, the following factors are considered:

- The number of database scans. Since the database is very large, one database scan needs a large number of disk I/O's. Thus, the number of database scans should be as small as possible.
- The data structures. We need data structures for representing a database, and data structures for keeping frequent itemsets and candidate frequent itemsets. The efficiency of the data structures has great influence on the efficiency of the algorithm.

- Memory management. This is an issue for any algorithm, especially now as the input of the algorithm is a very large database.
- The approaches for maximality checking and closedness testing also determine the efficiency of the algorithm for mining maximal and closed frequent itemsets.

Most of the known algorithms, such as Apriori [6, 7], DepthProject [5], and dEclat [60], work well when the main memory is big enough to fit the whole database or/and the data structures (candidate sets, FP-trees, etc). When a database is very large or when the minimum support is very low, either the data structures used by the algorithms may not be accommodated in main memory, or the algorithms spend too much time on multiple passes over the database. This was, for example, demonstrated by the *First IEEE ICDM Workshop on Frequent Itemset Mining Implementations, FIMI '03* [2], where a number of well known algorithms were implemented and independently tested. The results showed that "*none* of the algorithms is able to gracefully scale-up to very large datasets, with millions of transactions" [15].

At the same time very large databases do exist in real life. In a medium size business or in a company as big as Walmart, it is very easy to collect a few gigabytes of data. Commerce, science, and government can store terabytes of raw data. The question of how to handle these databases is still one of the most difficult problems in data mining.

1.2 Contributions

In this thesis, we use the FP-tree structure, the data structure that was first introduced in [28], to mine frequent itemsets in databases. The following is a list of our contributions:

1. A novel technique that uses the FP-array to greatly improve the performance of the algorithms operating on FP-trees.
2. By giving algorithm FPgrowth*, we first demonstrate that the use of our FP-array technique drastically speeds up the FP-growth method [28], since it needs to scan each FP-tree only once for each recursive call emanating from it. We then use this technique and give a new algorithm FPmax*, which extends our previous algorithm FPmax, for mining maximal frequent itemsets. In FPmax*, we use a variant of the FP-tree structure for maximality testing, and give

number of optimizations that further reduce runtime. We also design an algorithm, FPclose, for mining closed frequent itemsets. FPclose uses yet another variation of the FP-tree structure for checking the closedness of frequent itemsets. The closedness checking is quite different from CLOSET+ [53].

3. We also consider the problem of mining frequent itemsets from *very* large databases. We adopt a divide-and-conquer approach. First we give three algorithms, the general divide-and-conquer algorithm, then an algorithm using simple projection, and an algorithm using aggressive projection. We analyze the number of steps and disk I/O's required by these algorithms. In a detailed divide-and-conquer algorithm, called *Diskmine*, we use the highly efficient FPgrowth* method [21, 22] to mine frequent itemsets from a FP-tree for the main memory part of data mining. We describe several novel techniques useful in mining frequent itemsets from disks, such as the *FP-array technique*, the *item-grouping technique*, and *memory management techniques*.

1.3 Organization of this thesis

The structure of this thesis is as follows,

- In Chapter 2 we define the problem of mining frequent itemsets in large and very large databases and identify the related work done on the problem.
- Chapter 3 describes our solutions for the problems of mining all frequent itemsets. In this chapter, we first give our FP-array technique that results in the greatly improved method FPgrowth*. The FP-array technique is also applied to save disk I/O's for mining frequency pattern from very large datasets. Experimental results are given at the end of the chapter.
- Chapter 4 describes our first attempt to solve the problems of mining maximal frequent itemsets. Algorithm FPmax and a data structure, MFI-tree, are introduced. The likely behavior of some existing algorithms are analyzed and validated based on experimental results.
- Chapter 5 describes an improved solution for the problem of mining maximal frequent itemsets. Algorithm FPmax* is given and some new techniques used in the algorithm are explained. Experimental results are given at the end of this chapter.

- Chapter 6 describes our solutions to the problem of mining closed frequent itemsets. A data structure, CFI-tree, is introduced. Experimental results are given at the end of the chapter.
- Since most algorithms for mining frequent itemsets do not perform very well for very large datasets, in Chapter 7, we first introduce and analyze three algorithms for mining frequent itemsets from disks. Then we give a detailed divide-and-conquer algorithm *Diskmine*, in which many novel optimization techniques are used. Experimental results are also given.
- Chapter 8 summarizes the work done in this thesis. We also discuss some open problems and future work.

Chapter 2

Problem Definition and Related Work

In this chapter, we first define the problem of mining frequent itemset, then give a survey of the related work done on the problem, in which some existing algorithms are identified and analyzed.

2.1 Problem Definition

Definition 2.1 *Let* $I = \{i_1, i_2, \ldots, i_n\}$, *be a finite set of* items. *An* I-transaction τ *is a subset of* I. *An I-transactional* database $\mathcal{D}$ *is a finite bag of I-transactions. An itemset S is a subset of I. Association rules are statements of the form* $X \Rightarrow Y$, *where* $X, Y \subseteq I$. ■

In this thesis, items will sometimes also be denoted by $a, b, c, \ldots$.

The association rule $X \Rightarrow Y$ means that if we find all items in X, then we have a good chance of finding all items in Y.

To find all association rules, we use two important measures, *support* and *confidence.*

Definition 2.2 *The count of an itemset* X, *count(X), is the number of transactions in* $\mathcal{D}$*that contain* X. *The support in* $\mathcal{D}$*for* X *is the percentage of transactions in* $\mathcal{D}$*that contains* X. *The support for a rule* $X \Rightarrow Y$ *is the probability that a transaction of* $\mathcal{D}$ *contains both* X *and* Y. *In other words, the support is the count of transactions that contain both* X *and* Y *divided by the number of transactions in* $\mathcal{D}$. ■

Since we always know the number of transactions, in this thesis, sometimes we also use count to represent support.

For a rule whose support is low, the rule may have arisen purely by chance.

Definition 2.3 *The confidence of a rule is the probability that a transaction of $\mathcal{D}$ contains Y given that the transaction contains X (i.e., $Prob(Y \mid X) = count(X \cup Y)/count(X)$).* ■

The confidence of a rule indicates the conditional probability (in the database) between the itemsets in the rule.

The goal of mining association rules is to find rules with a support of at least a user specified *minimum support* and a confidence of at least a user specified *minimum confidence.*

The central notion for the rest of this thesis is that of a *frequent itemset.*

Definition 2.4 *An itemset X is called frequent if its support is greater than the minimum support.* ■

From the definition of confidence, we can see that if we know the support of every itemset, it is straightforward to compute the confidence of a rule. Thus, the problem of mining association rules can be converted to the problem of mining frequent itemsets.

The itemsets in a transactional database gives rise to a subset lattice. Figure 2.1 is an example of a lattice on the space $I = \{a, b, c, d\}$ of four items. In the figure, Φ represents the empty set.

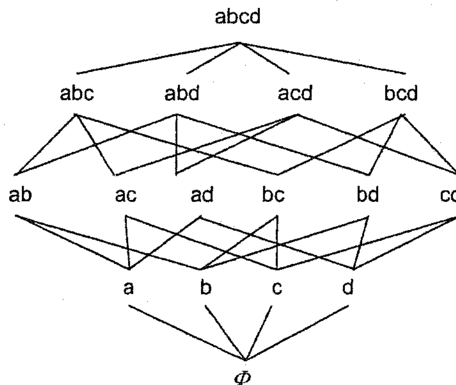

Figure 2.1: An itemset lattice

When a transaction database is very dense, i.e. when the database contains a large number of large frequent itemsets, mining *all* frequent itemsets might not be a good idea. For example, if there is a frequent itemset with size ℓ, then all 2^{ℓ} nonempty subsets of the itemset are also frequent

and have to be generated. However, since frequent itemsets are *downward closed*, meaning that any subset of a frequent itemset is frequent, it is sufficient to discover only all the *maximal* frequent itemsets (MFI's).

Definition 2.5 *A frequent itemset X is called maximal if there does not a exist frequent itemset Y such that $X \subset Y$.* ■

Many existing algorithms only mine maximal frequent itemsets (MFI). However, mining only MFI's has the following deficiency. From a MFI and its support δ, we know that all its subsets are frequent and the support of any of its subset is not less than δ, but we do not know the exact value of the support. For generating association rules, we do need support of all frequent itemsets. To solve this problem, another type of a frequent itemset, the *Closed* Frequent Itemset (CFI), was proposed.

Definition 2.6 *A frequent itemset X is called closed if the support of all its supersets is less than the support of X.* ■

$a\ d\ e$
$b\ a\ f\ g\ h$
$b\ a\ d\ f$
$b\ a\ c$
$a\ d\ g\ k$
$b\ d\ g\ c\ i$
$b\ d\ g\ e\ j$

Figure 2.2: A database example

Definition 2.7 *A CFI Y is called minimal CFI of X if either $X = Y$ or $X \subset Y$ and Y has the greatest support in all supersets of X. The support of a frequent itemset X is the same as the support of its minimal closed superset Y.* ■

In Figure 2.2, if the minimum support is 2, then $\{f, a\}$ and $\{f, a, b\}$ are closed itemsets, while $\{f, b\}$ is not. The support of $\{f, b\}$ is 6, which is the same as the support of its minimal closed itemset $\{f, a, b\}$.

In most cases, though, the number of CFI's is greater than the number of MFI's, but still far less than the number of all frequent itemsets (FI's). As an example, in the small database Figure 2.2, if the minimum support is 2, the number of all FI's is 20, the number of MFI's is 6, and the number of CFI's is 15.

To mine frequent itemsets from a transactional database, transactions must be read from secondary memory, processed and perhaps stored into some data structures. Some frequent itemsets may be found and sometimes candidate frequent itemsets are generated. In order to make the mining process fast, one must use data structures to organize the transactions, frequent patterns and candidate frequent patterns intelligently. Some data structures may also be needed for maximality testing. Pruning techniques also play an important role in the mining process. The following gives a brief overview of the work that has been done on mining all FI's, MFI's and CFI's.

2.2 Related Work

2.2.1 Mining All Frequent Itemsets

The itemset lattice is a conceptualization of the search space when mining frequent itemsets. There are then basically two types of algorithms to mine frequent itemsets, *breadth-first* algorithms and *depth-first* algorithms. The breadth-first algorithms, such as Apriori [6, 7], scan the database testing all candidate itemsets level by level for frequency. Other depth-first algorithms, such as the FP-growth method [28], search the lattice bottom-up in "depth-first" way (one should perhaps say "height-first" way). From a singleton itemset $\{i\}$, successively larger candidate sets are generated by adding one element at a time. The following two sections give brief introduction of algorithm Apriori and the FP-growth method.

Apriori: A breadth-first Algorithm

The problem of mining frequent itemsets was first introduced by Agrawal *et al.* [6, 7], who proposed algorithm Apriori. Many algorithms, such as partitioning [48, 33], sampling [51], mining of generalized and multi-level rules [26, 50], mining of multi-dimensional rules [36], mining association rules with constraints [39, 34], and so on, are variants of the Apriori algorithm.

Apriori discovers all frequent itemsets in a large database of transaction. It takes advantages of the most important property of frequency pattern, *anti-monotonicity.* Anti-monotonicity means that every subset of a frequent itemset is also frequent. A candidate frequent itemset is generated only if all its subsets are all frequent. By this way, Apriori successfully prunes the candidate frequent itemsets.

Apriori uses hash-trees to store frequent itemsets and candidate frequent itemsets. Each leaf node in a hash-tree contains a list of itemsets, and each interior node contains a hash table. Candidate frequent itemset generation and subset testing are all based on the hash-trees.

Figure 2.3 gives the Apriori algorithm. In the algorithm, a k-itemset is an itemset that contains k items, $\mathcal{F}_k$ is set of frequent k-itemsets. C_k is set of candidate k-itemsets (potentially frequent itemsets). Each member of $\mathcal{F}_k$ and C_k has two fields: i) *itemset* and ii) *support count.*

Procedure *Apriori*($\mathcal{D}$)
Input: I-transactional data-set $\mathcal{D}$
Output: The complete set of all FI's in $\mathcal{D}$.
Method:
$\mathcal{F}_1$={frequent 1-itemsets};
for ($k = 2; \mathcal{F}_{k-1} \neq \emptyset; k++$) **do begin**
  //joins $\mathcal{F}_{k-1}$ with $\mathcal{F}_{k-1}$ to get k-itemsets
  insert into C_k
  select $p.item_1, p.item_2, \ldots, p.item_{k-1}, q.itemk - 1$
  from $\mathcal{F}_{k-1}\ p, \mathcal{F}_{k-1}\ q$
  where $p.item_1 = q.item_1, \ldots, p.item_{k-2} = q.itemk - 2, p.item_{k-1} < q.item_{k-1}$;
  for each itemsets $c \in C_k$ **do** //prune
    for each $(k-1)$-subsets s of c **do**
      if($s \notin \mathcal{F}_{k-1}$) **then**
        delete c from C_k;
  for each transactions $T \in \mathcal{D}$ **do begin**
    $C_T = \{c \mid c \in C_k \ and \ c \subseteq T\}$; //Candidates contained in T
    for each candidates $c \in C_T$ **do**
      $c.count++$;
  end
  $\mathcal{F}_k$ =$\{c \in C_k \mid c.count \geq minsup\}$
end
return $\cup_k \mathcal{F}_k$;

Figure 2.3: Algorithm Apriori

In Figure 2.3, the first pass of the algorithm simply counts item occurrences to determine the frequent 1-itemsets. A subsequent pass, say pass k, consists of two phases. First, the algorithm joins $\mathcal{F}_{k-1}$ with $\mathcal{F}_{k-1}$ to generate all possible k-itemsets. Since any subset of a frequent itemset must be frequent, the algorithm prunes all itemsets $c \in C_k$ such that some $(k-1)$-subset of c is not in $\mathcal{F}_{k-1}$. Next, the database is scanned and the support of candidates in C_k is counted. The output of the algorithm is the union of the frequent itemsets of all passes.

The disadvantage of using Apriori for large database is in the fact that since transactions are not stored in the main memory, Apriori needs ℓ database scans if the size of the largest frequent itemset is ℓ. Thus, for large database, the database scans may take much time.

Apriori is the first algorithm for mining frequent itemsets. Later, many variants [41, 51, 48,

39, 34, 10, 40] were proposed and implemented. These algorithms use some data structures to reduce the database scans or count the support of frequent itemsets fast. For example, in [41], hash tables are used for candidate set generation. The algorithm is especially fast for candidate 2-itemsets generation. Sampling [51] is another way to reduce the database scans. It picks a random sample from the very large database, finds from the sample all association rules that probably hold in the whole database, then verifies the results with the rest of the database. In [10], the Apriori is implemented by using a prefix tree representation for counters and a doubly recursive scheme to count the transactions. The kDCI method [40] applies a novel counting inference strategy to efficiently determine the itemset supports. Because of all these improvements, it is shown in [15] that Apriori is still the fastest algorithm for some datasets in some cases.

FP-growth method: A depth-first Algorithm

Breadth-first algorithms always repeatedly scan the database and check a set of candidates for frequency by pattern-matching. This is costly especially when there exist prolific frequent patterns, long patterns or quite low minimum support thresholds.

On the other hand, depth-first algorithms such as the FP-growth method save disk I/O's by scanning the database once or twice. In the FP-growth method, Han *et al.* proposed a data structure tree called an a FP-tree (Frequent Pattern tree). The FP-tree is a compact representation of all relevant frequency information in a database. Every branch of the FP-tree represents a frequent itemset, and the nodes along the branches are stored in decreasing order of frequency of the corresponding items, with leaves representing the least frequent items. Compression is achieved by building the tree in such a way that overlapping itemsets share prefixes of the corresponding branches.

A FP-tree T has a header table, $T.header$, associated with it. Single items and their counts are stored in the header table in decreasing order of their frequency. The entry for an item also contains the head of a list that links all the corresponding nodes of the FP-tree.

Compared with Apriori [7] and its variants, which need as many database scans as the length of the longest pattern, the FP-growth method only needs two database scans when mining all frequent itemsets. The first scan counts the number of occurrences of each item. The second scan constructs the initial FP-tree which contains all frequency information of the original dataset. Mining the

database then becomes mining the FP-tree.

To construct the FP-tree, it is necessary to find all frequent items by an initial scan of the database. Then these items are inserted in the header table, in decreasing order of their count. In the next (and last) scan, as each transaction is scanned, the set of frequent items in it is inserted into the FP-tree as a branch. If an itemset shares a prefix with an itemset already in the tree, the new itemset will share a prefix of the branch representing that itemset. In addition, a counter is associated with each node in the tree. The counter stores the number of transactions containing the itemset represented by the path from the root to the node in question. This counter is updated during the second scan, when a transaction causes the insertion of a new branch. Figure 2.4 (a) shows an example of a dataset and Figure 2.4 (b) the FP-tree for that dataset. Note that there may be more than one node corresponding to an item in the FP-tree. The frequency of any one item i is the sum of the count associated with all nodes representing i, and the frequency of an itemset equals the sum of the counts of the least frequent item in it, restricted to those branches that contain the itemset. For instance, from Figure 2.4 (b) we can see that the frequency of the itemset $\{a, d\}$ is 3.

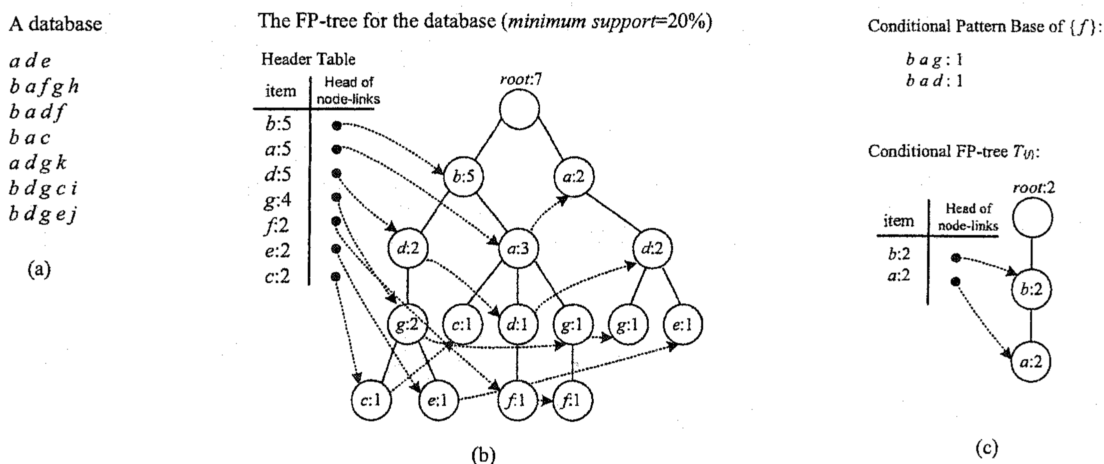

Figure 2.4: An Example FP-tree

Thus the constructed FP-tree contains all frequency information of the database. Mining the database becomes mining the FP-tree. Figure 2.5 gives the FP-growth method. In Figure 2.5, FP-tree T has two attributes: *base* and *header*. *T.base* contains the itemset X, for which T is a conditional FP-tree, and the attribute *header* contains the head table.

The FP-growth method relies on the following principle: if X and Y are two itemsets, the count of itemset $X \cup Y$ in the database is exactly that of Y in the restriction of the database to those

transactions containing X. This restriction of the database is called the *conditional pattern base* of X, and the FP-tree constructed from the conditional pattern base is called X's *conditional FP-tree*, which we denote by T_X. We can view the FP-tree constructed from the initial database as $T_\emptyset$, the conditional FP-tree for $\emptyset$. Note that for any itemset Y that is frequent in the conditional pattern base of X, the set $X \cup Y$ is a frequent itemset for the original database.

```
Procedure FP-growth(T)
Input:  A conditional FP-tree T
Output: The complete set of all FI's corresponding to T.
Method:
  if T only contains a single path P
  then for each subpath Y of P
     output pattern Y ∪ T.base with count = smallest count of nodes in Y
  else for each i in T.header do begin
     output Y = T.base ∪ {i} with i.count
     traverse T to construct a new header table for Y's FP-tree
     traverse T to construct Y's conditional FP-tree T_Y;
     if T_Y ≠ ∅
        call FP-growth(T_Y);
  end
```

Figure 2.5: Algorithm FP-growth

Given an item i in the header table of a FP-tree T_X, by following the linked list starting at i in the header table of T_X, all branches that contain item i are visited. These branches form the conditional pattern base of $X \cup \{i\}$, so the traversal obtains all frequent items in this conditional pattern base. The FP-growth method then constructs the conditional FP-tree $T_{X\cup\{i\}}$, by first initializing its header table based on the found frequent items, and then visiting the branches of T_X along the linked list of i one more time and inserting the corresponding itemsets in $T_{X\cup\{i\}}$. Note that the order of items can be different in T_X and $T_{X\cup\{i\}}$. As an example, let us construct the conditional FP-tree $T_{\{f\}}$ for the database in Figure 2.4 (a). By following the linked list for f in the original tree $T_\emptyset$, we discover that the items and their counts in the conditional pattern base of $\{f\}$ are b:2, a:2, g:1, d:1. Then we follow the linked list for f in $T_\emptyset$ one more time, now inserting into $T_{\{f\}}$ the branch $(b{:}1, a{:}1)$ (g is not frequent in the base) from the first node in the linked list, and then the branch $(b{:}1, a{:}1)$ from the second node in the linked list. The resulting $T_{\{f\}}$ is the conditional FP-tree for $\{f\}$, shown in Figure 2.4(c). The above procedure is applied recursively, and it stops when the resulting new FP-tree contains only one single path. The complete set of frequent itemsets is generated from all single-path FP-trees.

The FP-growth method has great performance when compared with Apriori algorithm. Experimental results in [28] show that it is about an order of magnitude faster than the Apriori algorithm. It has been widely used in many database research tasks such as the data cube computations [27].

There are many improved variants of the FP-growth method. Top-down FP-growth [54] searches the FP-tree in top-down order, which is different from the bottom-up order in the original FP-growth method. By top-down search, Top-down FP-growth does not generate conditional pattern bases and sub-FP-trees, thus saving time and space. In [45], Pei *et al.* extended the FP-tree data structure to Hyper-structure for mining frequent patterns in large databases. The experimental results show that the algorithm H-mine has high performance in various kinds of data.

FPgrowth*[21, 22] extends the FP-growth method by using optimizations such as a FP-array technique for reducing FP-tree traversing time. FPgrowth* is up to two times faster than the FP-growth method especially when the dataset is sparse. Details of FPgrowth* will be explained in Chapter 3.

PatriciaMine [46] uses the data structure Patricia Tries, which is similar to FP-tree, to represent frequency information of the original dataset. In the algorithm, a number of optimizations are used for reducing time and space of database projections.

Other Depth-first Algorithms

In [58], Zaki and Gouda also proposed a depth-first search algorithm, Eclat, in which the database is "vertically" represented. Eclat uses a linked list to organize frequent patterns, however, each itemset now corresponds to an array of transaction IDs (the "TID-array"). Each element in the array corresponds to a transaction that contains the itemset. Frequent itemset mining and candidate frequent itemset generation are done by TID-array intersections. Later, Zaki and Gouda [60], introduced a technique, called *diffset*, for reducing the memory requirement of TID-arrays. The diffset technique only keeps track of differences in the TID's of a candidate itemset when it is generating frequent itemsets. The Eclat algorithm incorporating the diffset technique, is called dEclat [60].

2.2.2 Mining MFI's

Since frequent itemsets are downward closed, mining frequent itemsets can be reduced to mining a "border" in the itemset lattice. All itemsets above the border are infrequent, the others that are

below the border are all frequent. For instance, if a database is $\mathcal{D} = \{abd, bcd, abc, bcd, abcd, ac\}$ and the minimum support is 50%, then the border in the lattice of Figure 2.6(b), should be at the dashed line. Since the sets *abcd*, *abc*, *abd*, *acd*, and *ad* are above the border, they are all infrequent itemsets. The rest of the itemsets, *bcd*, *ab*, *ac*, *bc*, *bd*, *cd*, *a*, *b*, *c*, *d* and $\emptyset$ are all frequent itemsets, and they can be compactly represented by listing only the maximal elements $\{ab, ac, bcd\}$.

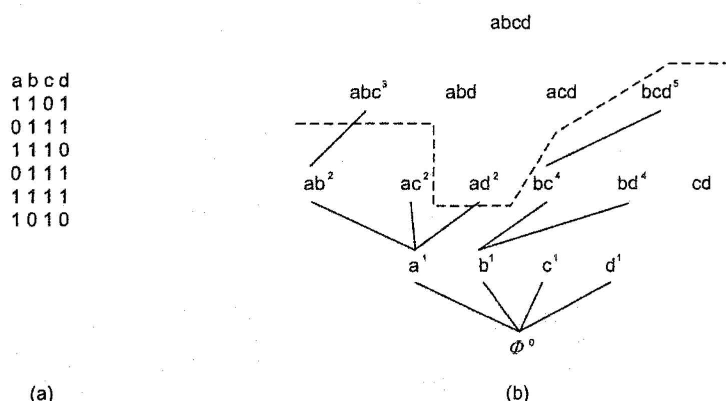

Figure 2.6: The bitmap representation and the depth-first search

The problem of mining maximal frequent itemsets can be divided into two subproblems. First, to mine for frequent itemsets. Second, for each discovered frequent itemset, to test if it is maximal. There are many ways to test if a frequent itemset is maximal. The naive method is to do a pairwise comparison between all pairs of frequent itemsets. However, we can take advantage of the order in which the frequent itemsets are generated to avoid unnecessary maximality checking.

Bayardo [9] introduced MaxMiner which extends Apriori to mine only "long" patterns (large maximal frequent itemsets). Since the MaxMiner only looks for the maximal FI's, the search space can be reduced. MaxMiner performs not only *subset* infrequency pruning, where a candidate itemset with an infrequent subset will not be considered, but also a "lookahead" to do *superset* frequency pruning. MaxMiner still needs many passes of the database to find maximal frequent itemsets.

There are also some depth-first algorithms for mining MFI's. DepthProject by Agarwal, Aggrawal, and Prasad [5] mines only long patterns. It performs a mixed depth-first/breadth-first traversal of the itemset lattice. In the algorithm, both subset infrequency pruning and superset frequency pruning are used. The database is represented as a bitmap. Each row in the bitmap is a bitvector corresponding to a transaction, each column corresponds to an item. The number of rows is equal to the number of transactions, and the number of columns is equal to the number of

items. A row has a 1 in the ith position if corresponding transaction contains the item i, and a 0 otherwise. Figure 2.6 (a) shows an example for bitmap representation of the transaction database $\mathcal{D}$. The count of an itemset is the number of rows that have 1's in all corresponding positions. For example, the count of bcd is 3 since row 2, 4 and 5 have 1's in position b, c and d. By carefully designed counting methods, the algorithm significantly reduces the cost for finding support counts. Experimental results in [5] show that DepthProject outperforms MaxMiner by at least an order of magnitude.

In [12], Burdick, Calimlim, and Gehrke extend the idea in DepthProject and give an algorithm called MAFIA to mine maximal frequent itemsets. Similar to DepthProject, their method also uses a bitmap representation, where the count of an itemset is based on the column in the bitmap (the bitmap is called "vertical bitmap"). As an example, in Figure 2.6 (a), the bitvectors for items b, c, and d are 111110, 011111, and 110110, respectively. To get the bitvectors for any itemset, we only need to apply the bitvector *and*-operation $\otimes$ on the bitvectors of the items in the itemset. For the above example, the bitvector for itemset bc is $111110 \otimes 011111$, which equals 011110, while the bitmap for itemset bcd can be calculated from the bitmaps of bc and d, i.e., $011110 \otimes 110110$, which is 010110. The count of an itemset is the number of 1's in its bitvector. MAFIA is a depth-first algorithm. The following example demonstrates its mining sequence. In the dataset Figure 2.6 (a), with a minimum support of 50%, Figure 2.6 (b) shows the the sequence of itemsets tested for frequency. The sets are tested in the order indicated by the number on the top-right side of the itemsets.

MAFIA also uses some pruning techniques. For example, the support of an itemset XY equals the support of X, if and only if the *and* of the bitvectors for X and Y equals the bitvector for X. This is the case if the bitvector for Y has a 1 in every position that the bitvector for X has 1. The last condition is easy to test. This allows us to conclude without counting that XY also is frequent. The technique is called Parent Equivalence Pruning in [12].

In general, MAFIA is regarded as one of the best method for mining all MFI's.

Another method, GenMax, was proposed by Gouda and Zaki [16]. It takes a novel approach to maximality testing. Most methods, including MaxMiner use a variant of the algorithm in [55] and find the maximal elements among n sets in time $O(\sqrt{n} \log n)$. Gouda and Zaki use a technique called *progressive focusing*. This technique, instead of comparing a newly found frequent itemset

with all maximal frequent itemsets found so far, maintains a set of *local maximal frequent itemsets*, LMFI's. The frequent itemset is firstly compared with itemsets in LMFI. Most non-maximal frequent itemsets can be found by this step, thus reducing the number of subset tests. GenMax like other methods uses a vertical representation of the database. However, for each itemset, GenMax stores a *transaction identifier set*, or TIS, rather than bitvector. The cardinality of an itemset's TIS equals its support. The TIS of itemset XY can be calculated from the intersection of the TIS's of X and Y. Experiment results show that GenMax outperforms other existing algorithms on some types of datasets.

In [20], G. Grahne and J. Zhu presented the FPmax algorithm for mining MFI's using the FP-tree structure. FPmax is also a depth-first algorithm. It takes advantage of the FP-tree structure so that only two database scans are needed, thus allowing for the database to be much larger than the size of the main memory. In FPmax, a tree structure similar to the FP-tree is used for maximality testing. Experimental results show that FPmax outperforms GenMax and MAFIA for many, although not for all, cases. Details of FPmax will be introduced in Chapter 4. In [21, 22], FPmax is extended to FPmax*, in which the FP-array technique is used to reduce the time spent on traversing FP-trees, and a more efficient maximality checking approach is also used. In [15], FPmax* is recommended as "one of the best for maximal itemset mining". Details of FPmax* will be introduced in Chapter 5.

One more method that uses the FP-tree structure is AFOPT [35]. In the algorithm, item searching order, conditional database representation, conditional database construction strategy, and tree traversal strategy are considered which makes the algorithm adaptive to general situations.

SmartMiner [61], also a depth-first algorithm, uses a technique to prune candidate frequent itemsets in the itemset lattice fast. The technique gathers "tail" information for a node in the lattice, the tail information will be passed to determine the next node to explore during the depth-first mining. Items are dynamic reordered based on the tail information. The algorithm was compared with MAFIA and GenMax on two datasets, and it shows that it's an order of magnitude faster than MAFIA and GenMax.

2.2.3 Mining CFI's

In [43], Pasquier *et al.* introduced closed frequent itemsets. The algorithm proposed in the paper, A-close, extends Apriori to mine all CFI's. Zaki and Hsiao [59] developed a depth first algorithm,

CHARM, for CFI mining. As in their earlier work [16], in CHARM, each itemset corresponds to a TID-array, and the main operation of the mining is again TID-array intersections. CHARM also uses the diffset technique to reduce the memory requirement for TID-array intersections.

In [44], Pei, Han, and Mao extend the FP-growth method to a method called CLOSET for mining CFI's. The FP-tree structure is used, and some optimizations for reducing the search space are proposed. Experimental results show that CLOSET is faster than CHARM and A-close. In [30], the FP-growth method is extended to mining top-K frequent closed patterns without minimum support.

CLOSET is extended to CLOSET+ by Wang *et al.* in [53] to find the best strategies for mining frequent closed itemsets. In CLOSET+, Wang *et al.* use data structures and data traversal strategies that depend on the characteristics of the dataset to be mined. In the algorithm, a global prefix-tree is used to keep track of all closed itemsets. A candidate closed frequent itemset will be compared with some already found closed itemset kept in the tree. According to the experimental results, CLOSET+ outperforms all previous algorithms.

FPclose [21, 22] is also an extension of the FP-growth method. In FPclose, like in CLOSET+, a prefix-tree is introduced. However, instead of maintaining one global tree for all candidate closed frequent itemsets, a prefix-tree is constructed for each conditional FP-tree. With other optimizations, FPclose outperforms almost all other algorithms in [14]. Details of FPclose will be discussed in Chapter 6.

LCM [52] is another depth-first algorithm. In LCM, tree-shaped transversal routes are constructed. These routes consist of only frequent closed itemsets, and allow to find all frequent closed itemsets in polynomial time per itemset. Thus, storing previously found closed itemsets becomes unnecessary.

2.3 Mining frequent itemsets in very large databases

Many researchers have tried to mine frequent itemsets from very large databases. A number of approaches have been proposed. One of them is *sampling*. For instance, [51] picks a random sample of the database, finds all frequent itemsets from the sample, and then verifies the results with the rest of the database. This approach needs only one pass of the database. However, the results are probabilistic, meaning that some critical frequent itemsets could be missing.

Partitioning [48, 45] is another approach for mining very large databases. This approach first partitions the database into many small databases, and mines candidate frequent itemsets from each small database. One more pass over the original database is then done to verify the candidate frequent itemsets. The approach thus needs only two database scans. However, when the data structures used for storing candidate frequent itemsets are too big to fit in main memory, a significant amount of disk I/O's are needed for the disk resident data structures.

In [28, 29], Han *et al.* introduced the FP-growth method by using the FP-tree structure. Two approaches were suggested for the case that the FP-tree is too large to fit into main memory.

The first approach writes the FP-tree to disk, then mines all frequent sets by reading the frequency information from the FP-tree. However, the size of the FP-tree could be the same as the size of the database, and for each item in the FP-tree, we need at least two FP-tree traversals. Thus the I/O's for writing and reading the disk-resident FP-tree could be prohibitive.

The second approach *projects* the original database on each frequent item [1], then mines frequent itemsets from the small projected databases. The advantage of this approach is that any frequent itemset mined from a projected database is a frequent itemset in the original database. To get *all* frequent itemsets, we only need to take the union of the frequent itemsets from the small projected databases. This is in contrast to the partitioning approach, where all candidate frequent itemsets have to be stored and later verified by another pass of database. The biggest problem of the projection approach is that the total size of the projected databases could be too large, and there will be too many disk I/O's for the projected databases.

To reduce the disk I/O's for the projected databases, G. Grahne and J. Zhu [23] introduced an *aggressive* projection approach. In the approach, a projected database is for a group of items. In addition, the approach uses many techniques such as the FP-array technique, which make it possible for the approach to fully use the limited main memory and save numerous number of disk I/O's.

[1] in a way to be explained in Chapter 7

Chapter 3

Discovering All Frequent Itemsets

In this chapter, we address the problem of mining all frequent itemsets. First, we give our FP-array technique for reducing the traversal of FP-trees. By applying FP-array on the FP-growth method, the method is extended to FP-growth*. Then we give our first attempt to apply the FP-array technique on mining frequent itemsets from very large databases. Experimental results are given at the end of this chapter.

3.1 FP-array technique

The main work done in the FP-growth method is traversing FP-trees and constructing new conditional FP-trees after the first FP-tree is constructed from the original database. From numerous experiments we found out that about 80% of the CPU time was used for traversing FP-trees. Thus, the question is, can we reduce the traversal time so that the method can be sped up?

The answer is yes, by using a simple additional data structure. Recall that for each item i in the header of a conditional FP-tree T_X, two traversals of T_X are needed for constructing the new conditional FP-tree $T_{X \cup \{i\}}$. The first traversal finds all frequent items in the conditional pattern base of $X \cup \{i\}$, and initializes the FP-tree $T_{X \cup \{i\}}$ by constructing its header table. The second traversal constructs the new tree $T_{X \cup \{i\}}$. We can omit the first scan of T_X by constructing a frequent pair array A_X while building T_X. T_X is initialized with an attribute A_X.

Definition 3.1 *Let T be a conditional FP-tree, $I = \{i_1, i_2, \ldots, i_m\}$ be the set of items in the header table of T. A frequent pair array (FP-array) of T is a $(m-1) \times (m-1)$ matrix, where each element*

of the matrix corresponds the counter of an ordered pair of items in I. ■

Obviously, there is no need to set a counter for both item pair (i_j, j_k) and item pair (i_k, i_j). Therefore we only set the counters for all pairs (i_k, i_j) such that $k < j$.

We use an example to explain the construction of the FP-array. In Figure 2.4 (a), supposing that the minimum support is 20%, after the first scan of the original database, we sort the frequent items as b:5, a:5, d:5, g:4, f:2, e:2, c:2. This order is also the order of items in the header table of $T_\emptyset$. During the second scan of the database we will construct $T_\emptyset$, and a FP-array $A_\emptyset$. This FP-array will store the counts of all pairs of items. All cells in the FP-array are initialized as 0.

a	3					
d	3	3				
g	3	2	3			
f	2	2	1	1		
e	1	1	2	1	0	
c	2	1	1	1	0	0
	b	a	d	g	f	e

(a) $A_\emptyset$

d	2	
a	1	1
	b	d

(b) $A_{\{g\}}$

Figure 3.1: Two FP-array examples

In $A_\emptyset$, each cell is a counter of a pair of items, cell $A_\emptyset[c, b]$ is the counter for itemset $\{c, b\}$, cell $A_\emptyset[c, a]$ is the counter for itemset $\{c, a\}$, and so forth. During the second scan for constructing $T_\emptyset$, for each transaction, first all frequent items in the transaction are extracted. Suppose these items form itemset I. To insert I into $T_\emptyset$, the items in I are sorted according to the order in header table of $T_\emptyset$. When we insert I into $T_\emptyset$, at the same time $A_\emptyset[i, j]$ is incremented by 1 if $\{i, j\}$ is contained in I. For example, for the second transaction, $\{b, a, f, g\}$ is extracted (item h is infrequent) and sorted as b, a, g, f. This itemset is inserted into $T_\emptyset$ as usual, and at the same time, $A_\emptyset[f, b], A_\emptyset[f, a]$, $A_\emptyset[f, g], A_\emptyset[g, b]$, $A_\emptyset[g, a], A_\emptyset[a, b]$ are all incremented by 1. After the second scan, FP-array $A_\emptyset$ contains the counts of all pairs of frequent items, as shown in table (a) of Figure 3.1.

Next, the FP-growth method is recursively called to mine frequent itemsets for each item in header table of $T_\emptyset$. However, now for each item i, instead of traversing $T_\emptyset$ along the linked list starting at i to get all frequent items in i's conditional pattern base, $A_\emptyset$ gives all frequent items for i. For example, by checking the third line in the table for $A_\emptyset$, frequent items b, a, d for the conditional pattern base of g can be obtained. Sorting them according to their counts, we get b, d, a. Therefore,

for each item i in $T_\emptyset$ the FP-array $A_\emptyset$ makes the first traversal of $T_\emptyset$ unnecessary, and $T_{\{i\}}$ can be initialized directly from $A_\emptyset$.

For the same reason, from a conditional FP-tree T_X, when we construct a new conditional FP-tree for $X \cup \{i\}$, for an item i, a new FP-array $A_{X\cup\{i\}}$ is calculated. During the construction of the new FP-tree $T_{X\cup\{i\}}$, the FP-array $A_{X\cup\{i\}}$ is filled. For instance, from the FP-tree in Figure 2.4(b), if the conditional FP-tree $T_{\{g\}}$ is constructed, the cells of its FP-array $A_{\{g\}}$ will be the table (b) in Figure 3.1. This FP-array is constructed as follows. From the FP-array $A_\emptyset$, we know that the frequent items in the conditional pattern base of $\{g\}$ are, in order, b, d, a. By following the linked list of g, from the first node we get $\{d, a\} : 1$, so it is inserted as $(d : 1, a : 1)$ into the new FP-tree $T_{\{g\}}$. At the same time, $A_{\{g\}}[d, a]$ is incremented by 1. From the second node in the linked list, $\{b, a\} : 1$ is extracted, and it is inserted as $(b : 1, a : 1)$ into $T_{\{g\}}$. At the same time, $A_{\{g\}}[b, a]$ is incremented by 1. From the third node in the linked list, $\{b, d\} : 2$ is extracted, and it is inserted as $(b : 2, a : 2)$ into $T_{\{g\}}$. At the same time, $A_{\{g\}}[b, a]$ is incremented by 2. Since there are no other nodes in the linked list, the construction of $T_{\{g\}}$ is finished, and FP-array $A_{\{g\}}$ is ready to be used for construction of FP-trees in next level of recursion. The construction of FP-arrays and FP-trees continues until the FP-growth method terminates.

Based on above discussion, we define a variation of the FP-tree structure in which besides all attributes given in [28], a FP-tree also has an attribute, *FP-array*, which contains the corresponding FP-array.

Now let us analyze the size of a FP-array. Suppose the number of frequent items in the first FP-tree is n. Then the size of the associated FP-array is $\sum_{i=1}^{n-1} i = n(n-1)/2$. We can expect that FP-trees constructed from the first FP-tree have fewer frequent items, so the sizes of the associated FP-arrays decrease. At any time, since a FP-array is an attribute of a FP-tree, when the space for the FP-tree is freed, the space for the FP-array is also freed.

3.2 Discussion

The FP-array technique works very well especially when the dataset is sparse and very large. The FP-tree for a sparse dataset and the recursively constructed FP-trees will be big and bushy, due to the fact that they do not have many shared common prefixes. The FP-arrays save traversal time for all items and the next level FP-trees can be initialized directly. In this case, the time saved

by omitting the first traversals is far greater than the time needed for accumulating counts in the associated FP-arrays.

However, when a dataset is dense, the FP-trees are more compact. For each item in a compact FP-tree, the traversal is fairly rapid, while accumulating counts in the associated FP-array may take more time. In this case, accumulating counts may not be a good idea.

Even for the FP-trees of sparse datasets, the first levels of recursively constructed FP-trees are always conditional FP-trees for *the most common prefixes.* We can therefore expect the traversal times for the first items in a header table to be fairly short, so the cells for these first items are unnecessary in the FP-array. As an example, in Figure 3.1 table (a), since b, a, and d are the first 3 items in the header table, the first two lines do not have to be calculated, thus saving counting time.

Note that the datasets (the conditional pattern bases) change during the different depths of the recursion. In order to estimate whether a dataset is sparse or dense, during the construction of each FP-tree we count the number of nodes in each level of the tree. Based on experiments, we found that if the upper quarter of the tree contains less than 15% of the total number of nodes, we are most likely dealing with a dense dataset. Otherwise the dataset is likely to be sparse.

If the dataset appears to be dense, we do not calculate the FP-array for the next level of the FP-tree. Otherwise, we calculate FP-array for each FP-tree in the next level, but the cells for the first several (say 15) items in its header table are not set.

In Chapter 7, we will see that the FP-array technique works also very well for very large datasets.

3.3 FPgrowth* : an improved FP-growth method

Figure 3.2 contains the pseudo code for our new method FPgrowth*. The procedure has an FP-tree T as parameter. The tree has attributes: *base*, *header* and *FP-array*. *T.base* contains the itemset X, for which T is a conditional FP-tree, the attribute *header* contains the head table, and *T.FP-array* contains the FP-array A_X.

In *FPgrowth**, line 6 tests if the FP-array of the current FP-tree is *NULL*. If the FP-tree corresponds to a sparse dataset, its FP-array is not *NULL*, and line 7 will be used to construct the header table of the new conditional FP-tree from the FP-array directly. One FP-tree traversal is saved for this item compared with the FP-growth method in [28]. In line 9, during the construction,

Procedure *FPgrowth*(T)*
Input: A conditional FP-tree T
Output: The complete set of all FI's corresponding to T.
Method:
1. **if** T only contains a single path P
2. **then for each** subpath Y of P
3. output itemset $Y \cup T.base$ with count = smallest count of nodes in Y
4. **else for each** i in *T.header* **do begin**
5. output $Y = T.base \cup \{i\}$ with $i.count$
6. **if** *T.FP-array* is not NULL
7. construct a new header table for Y's FP-tree from *T.FP-array*
8. **else** construct a new header table from T;
9. construct Y's conditional FP-tree T_Y and its FP-array A_Y;
10. **if** $T_Y \neq \emptyset$
11. call *FPgrowth**(T_Y);
12. **end**

Figure 3.2: Algorithm FPgrowth*

we also count the nodes in the different levels of the tree, in order to estimate whether we shall really calculate the FP-array, or just set $T_Y.FP\text{-}array = NULL$.

From our experimental results we found that a FP-tree could have millions of nodes, thus, allocating and deallocating those nodes takes plenty of time. In our implementation, we used our own main memory management for allocating and deallocating nodes. Since all memory for nodes in a FP-tree is deallocated after the current recursion ends, a chunk of memory is allocated for each FP-tree when we create the tree. The chunk size is changeable. After generating all frequent itemsets from the FP-tree, the chunk is discarded. Thus we successfully avoid freeing nodes in the FP-tree one by one, which is more time-consuming.

3.4 Experimental evaluation and performance study

In this section, we present a performance comparison of algorithm FPgrowth* with other algorithms. We first compare the performance of FPgrowth* with FPgrowth, then compare FPgrowth* with some of the best algorithms in FIMI'03 [21].

3.4.1 FPgrowth* versus FPgrowth

We implemented FPgrowth* and FPgrowth and compared them by running them on many datasets, synthetic and real. Here we give the results on two representative datasets and two real datasets. The two synthetic datasets, *T40I10D100K* and *T100I20D100K*, were generated from the benchmark

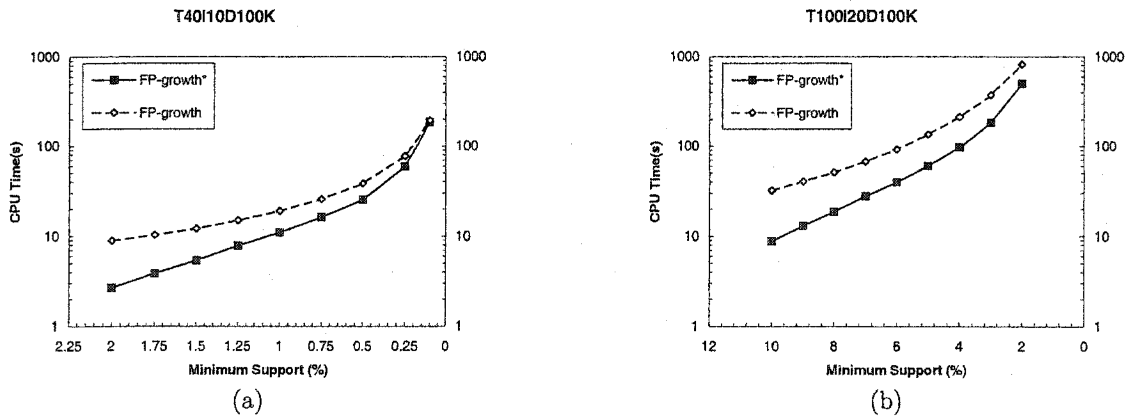

Figure 3.3: FPgrowth* vs. FPgrowth on sparse datasets

application of [1]. They both use 100,000 transactions and 1000 items. In *T40I10D100K*, the average transaction length was 40, and the average pattern length was 10. In *T100I20D100K*, the average transaction length was 100, and the average pattern length was 20. These two synthetic datasets are both sparse datasets. The two real datasets, *pumsb** and *connect*, were downloaded from [2]. Dataset *connect* is compiled from game state information. Dataset *pumsb** is produced from census data of Public Use Microdata Sample (PUMS). These two real datasets are both quite dense, so a large number of frequent itemsets can be mined even for very high values of minimum support.

All experiments were performed on a 1GHz Pentium III with 512 MB of memory running RedHat Linux 7.3. All time units in the figures refer to CPU time.

Figure 3.3 (a) and (b) show the CPU time of the two algorithms running on dataset *T40I10D100K* and *T100I20D100K* respectively. Due to the use of the FP-array technique, and the fact that *T40I10D100K* and *T100I20D100K* are sparse datasets, FPgrowth* turns out to be faster than FPgrowth. Since the FP-tree constructed from dataset *T100I20D100K* is taller and wider than *T40I10D100K*, the speedup from FPgrowth to FPgrowth* is increased. As another fact, when the minimum support is very low, we can expect the FP-tree to achieve a good compactification, starting at the initial recursion level. Thus the FP-array technique does not offer a big gain. Consequently, as verified in Figure 3.3, for very low levels minimum support, FPgrowth* and FPgrowth have almost the same runtime.

Figure 3.4 (a) and (b) show the CPU time of the two algorithms running on dataset *pumsb** and *connect* respectively. Since *pumsb** and *connect* are both very dense datasets, by some statistics we found that the FP-trees have very good compactness, and we can not expect that the FP-array

technique achieves a significant speedup for dense datasets, therefore the technique was not widely used during the mining, and FPgrowth* and FPgrowth have almost the same runtime.

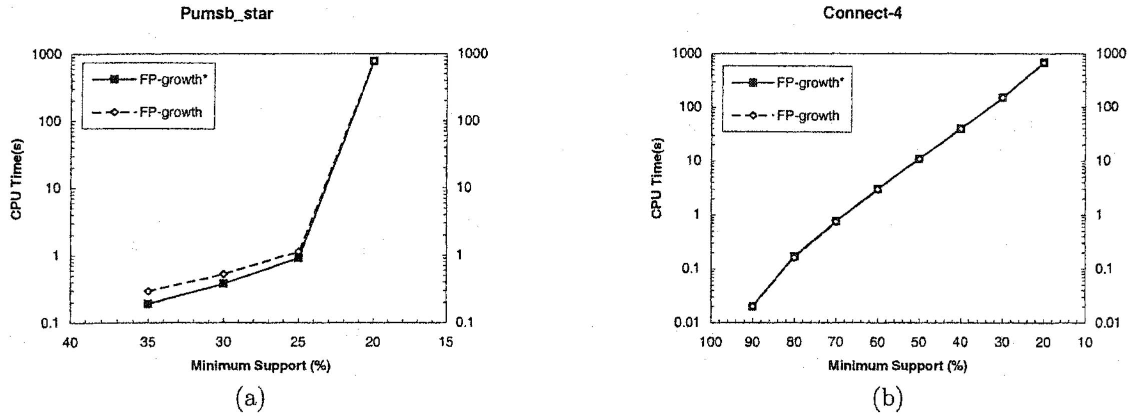

Figure 3.4: FPgrowth* vs. FPgrowth on dense datasets

3.4.2 FPgrowth* versus other algorithms

We conducted many performance tests to compare FPgrowth* with FP-growth, dEclat, and MAFIA (with "all" option). Part of the experimental results from both synthetic and real datasets were shown in [20, 21].

In 2003, *the First IEEE ICDM Workshop on Frequent Itemset Mining Implementations (FIMI'03)* [2] was organized. The goal of workshop was to find the most efficient algorithms for mining frequent itemset. The independent experiments conducted by the organizers compared the performance of algorithms such as optimized version of Apriori, FP-growth and Eclat. All in all 18 algorithms for mining all frequent itemsets were compared. The results demonstrate that some algorithms have obviously bad performance on all datasets. Unfortunately, the experiments in the workshop did not compare the scalability of algorithms, and the experimental results of the memory consumption of algorithms were not published. Therefore, we repeated part of the experiments of FIMI'03. In our experiments, we compared FPgrowth* with five existing algorithms, kDCI [40], dEclat [60], Apriori [10, 6, 7], PatriciaMine [46] and LCM [52], and FP-growth method [28, 29] implemented by us. All of these algorithms are the best or the second best algorithms in the workshop for some cases.

In FIMI'03, algorithms were run on 17 datasets, including synthetic and real, sparse and dense datasets. We use 7 the most representative datasets of the 17 datasets here.

The seven datasets include one synthetic dataset and six real datasets. The synthetic dataset

is *T20I10N1KP5KC0.25D200K*. It is generated from the benchmark synthetic dataset application from the website of IBM research center [1]. In *T20I10N1KP5KC0.25D200K*, the average transaction length is 20, average pattern length is 10, number of items is 1000, 5000 frequent patterns are used for seeds, the correlation between consecutive patterns is 0.25 and the transaction number is 200,000. In FIMI'03, algorithms were also run on another synthetic dataset, *T30I15N1KP5KC0.25D200K*, which is different from *T20I10N1KP5KC0.25D200K* only in the average transaction length and the average pattern length. In *T30I15N1KP5KC0.25D200K*, the average transaction length is 30 and the average pattern length is 15. Instead of it, in our experiments we worked with another synthetic dataset, *T100I20N1KP5KC0.25D200K*, generated by using the benchmark synthetic dataset application. Its average transaction length is 100 and average pattern length is 20. Obviously, the average transaction length and average pattern length differences between *T100I20N1KP5KC0.25D200K* and *T20I10N1KP5KC0.25D200K* are bigger than those between *T30I15N1KP5KC0.25D200K* and *T20I10N1KP5KC0.25D200K*.

The six real datasets are: *accidents, kosarak, chess, connect, mushroom,* and *pumsb**. Dataset *accidents* was donated by Karolien Geurts and contains (anonymous) traffic accident data. *Kosarak* was provided by Ferenc Bodon and contains (anonymous) click-stream data of a Hungarian on-line news portal. The *connect* and *pumsb** datasets are the same as the datasets in Section 3.4.1. The *chess* datasets are derived from their respective game steps. The *mushroom* dataset contains characteristics of various species of mushrooms. *Connect, chess*, and *mushroom* were originally taken from the UC Irvine Machine Learning Database Repository. Table 3.1 lists the number of items, average transaction length, size, and number of transactions in each dataset.

Dataset	#Items	Avg. Length	Size (bytes)	#Transactions
T20I10N1KP5KC0.25D200K	1000	20	15,821,880	200,000
T100I20N1KP5KC0.25D200K	1000	100	76,146,703	200,000
chess	75	37	342,294	3,196
connect	129	43	9,255,309	67,557
mushroom	119	23	570,408	8,124
kosarak	41,270	8.1	32,972,514	990,002
accidents	468	33.8	35,509,823	340,183
*pumsb**	2088	50.5	11,291,914	49,046

Table 3.1: Dataset Characteristics

The experiments were performed on a DELL Inspiron 8600 laptop with Pentium M, 1.6 GHz Processor, and 1GB of memory. The operating system was RedHat Linux 2.4.20 and gcc 3.2.2 was

used for the compilation. Both time and memory consumption of each algorithm running on each dataset were recorded. Runtime was recorded by "time" command, and memory consumption was recorded by "memusage". In the following figures, if there is no data for time of algorithm A running on dataset B for some minimum support δ, it is either because A needs too much time (longer than 30 minutes) to run on B for minimum support δ, or the implementation of the A has some bugs. There will be no memory consumption data either for A running on B for minimum support δ.

The runtime

Figure 3.5 and Figure 3.6 show the time of all algorithms running on *T20I10N1KP5KC0.25D200K* and *T100I20N1KP5KC0.25D200K*, respectively. In Figure 3.5, FPgrowth* is slower than kDCI, Apriori and dEclat for high minimum support. While for low minimum support, FPgrowth* becomes the fastest, and dEclat, kDCI and Apriori become an order of magnitude slower than FPgrowth*. The algorithm which was the fastest, dEclat, now becomes the slowest. At the same time, FPgrowth* and FPgrowth have almost the same runtime, their lines almost overlap, which means the FP-tree constructed from *T20I10N1KP5KC0.25D200K* is not tall and wide enough for using the FP-array techniques to make big difference.

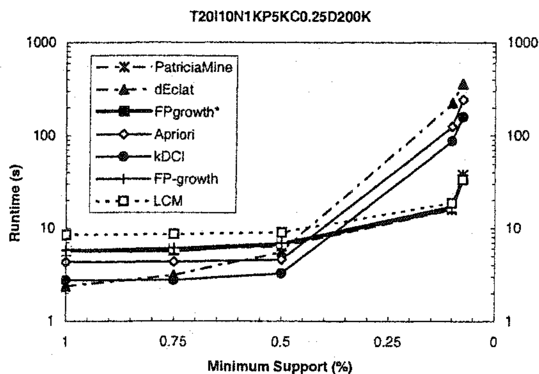

Figure 3.5: Runtime of Mining All FI's on *T20I10N1KP5KC0.25D200K*

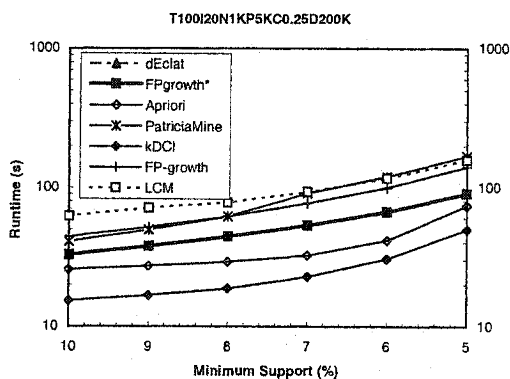

Figure 3.6: Runtime of Mining All FI's on *T100I20N1KP5KC0.25D200K*

In Figure 3.6, almost all algorithms have consistent performance on the dataset. The kDCI method is the fastest algorithm for all minimum supports. FPgrowth* ranks three in seven algorithms. Notice that FPgrowth* is almost 2 times faster than FPgrowth for minimum support 5%, which means that the FP-array technique saved much time for traversing the FP-trees.

The results shown in Figure 3.5 and Figure 3.6 can be explained as follows. When the datasets

such as the synthetic datasets are sparse, FPgrowth* has to construct bushy FP-trees. However, when the minimum support is high, not many frequent itemsets can be mined from the FP-trees. On the contrary, if the minimum support is low, the dataset could have many frequent itemsets. Then the time used for constructing FP-trees offers a big gain. For Apriori, kDCI, and dEclat, when the minimum support is high, there are fewer frequent itemsets and candidate frequent itemsets to be produced. Thus they only need to build small data structures for keeping frequent itemsets and candidate frequent itemsets, which does not take much time. But for low minimum support, considerable time has to be spent on keeping and pruning candidate frequent itemsets. The operations take more time than mining frequent itemsets from the compact FP-trees. This is why in the present experiment FPgrowth* is slow for high minimum support and fast for low minimum support. We can expect that in Figure 3.6, FPgrowth* would outperform Apriori, kDCI and dEclat for lower minimum support.

Figure 3.7 to Figure 3.12 show the experimental results of running 7 algorithms on all real datasets. In all figures, the line for the FP-growth method overlaps the line for FPgrowth*, which means the two methods have very similar performance on dense datasets.

In Figure 3.7, all the algorithms once again have consistent performances on dataset *chess* which is a very dense dataset. FPgrowth* has similar performance to PatriciaMine and LCM when the minimum support is high, but is slower than PatriciaMine and LCM when the minimum support is low.

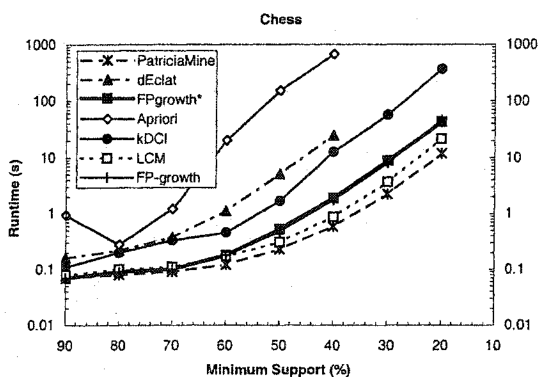

Figure 3.7: Runtime of Mining All FI's on *chess*

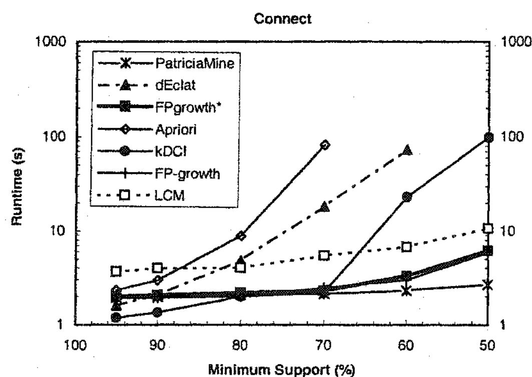

Figure 3.8: Runtime of Mining All FI's on *connect*

Shown in Figure 3.8 and 3.9, algorithms have similar performances on datasets *connect* and *mushroom*, both are very dense datasets. FPgrowth* is still slower than PatriciaMine for low minimum support. However, it is always at least two times faster than LCM.

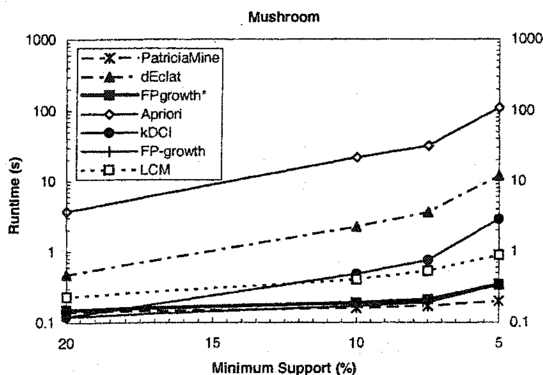

Figure 3.9: Runtime of Mining All FI's on *mushroom*

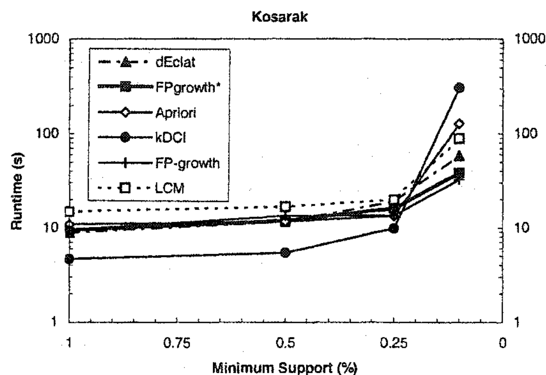

Figure 3.10: Runtime of Mining All FI's on *kosarak*

In Figure 3.10, FPgrowth* and FPgrowth are the fastest algorithms for very low minimum support. We also can see that FPgrowth* is even slower than FPgrowth for minimum minimum support 0.1%. That means the FP-tree constructed from dataset *kosarak* is so compact that the time saved for FP-tree traversing is less than the time for applying the FP-array technique.

Figure 3.11 shows the performance of all algorithms on *accidents*. In the figure, the line for the PatriciaMine almost totally overlaps the line for FPgrowth*, and FPgrowth* is one of the fastest algorithms when the minimum support is low.

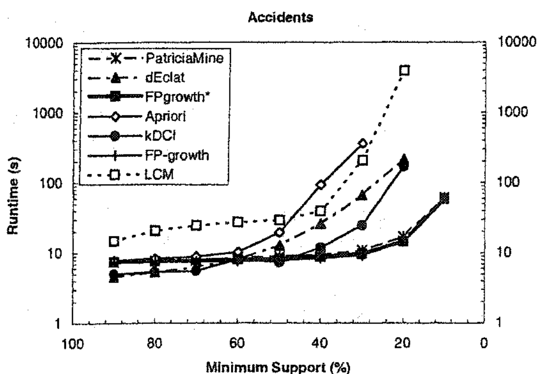

Figure 3.11: Runtime of Mining All FI's on *accidents*

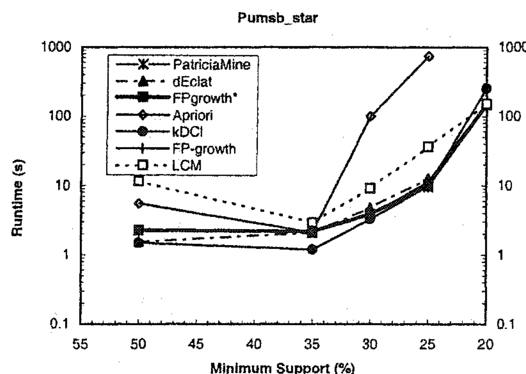

Figure 3.12: Runtime of Mining All FI's on *pumsb**

Similar to Figure 3.11, the PatriciaMine has almost the same speed as FPgrowth* (their lines

once again overlap), and FPgrowth* is still one of the fastest algorithms for low minimum support in Figure 3.12.

In figures 3.5 to 3.12, FPgrowth* shows stable performance. It is always among the fastest algorithms. On the contrary, Apriori, dEclat, kDCI, and LCM demonstrate their worst performances. For example, in Figure 3.5, dEclat is the fastest algorithm when the minimum support is low, but it is also the slowest one when the minimum support is high. The kDCI method has many cases as a fast algorithm, but it is the slowest one in Figure 3.10 when the minimum support is low. Apriori does not have a fastest case and it is always among the slowest algorithms.

There is no algorithm that is always the fastest. We believe that the data distributions in the datasets have some influence on the performances of the algorithms. Unfortunately, the exact influence of data distribution for each algorithm is still unknown. We also found that it is time-consuming to determine the data distribution before mining the frequent itemsets. Thus, it would be wise to choose the stable algorithms such as FPgrowth* to mine frequent itemsets.

Another observation is that PatriciaMine has similar performance as FPgrowth* method. This is because both algorithms are based on prefix tree data structure. FPgrowth* has many optimizations that are similar to those of PatriciaMine. Though FPgrowth* also uses the FP-array technique, all real datasets in Table 3.1 are dense datasets and the FP-array technique does not work very well on the dense datasets.

Memory Consumption

Since memory consumption also demonstrates the quality of an algorithm, we record the peak main memory consumption of the algorithms when running them on the datasets.

Figure 3.13 and Figure 3.14 show the peak main memory consumption of the algorithms on two synthetic datasets. FPgrowth* and the FP-growth method consume almost the same main memory, their lines overlap again. We can see that FPgrowth* and the FP-growth method unfortunately use the maximum amount of main memory. Its main memory consumption is almost 4 times bigger than the dataset size. Algorithm kDCI uses the lowest amount of main memory when the minimum support is low while dEclat uses the least main memory when the minimum support is high. From the figures, we also can see that memory consumption of all algorithms is bigger than the dataset size.

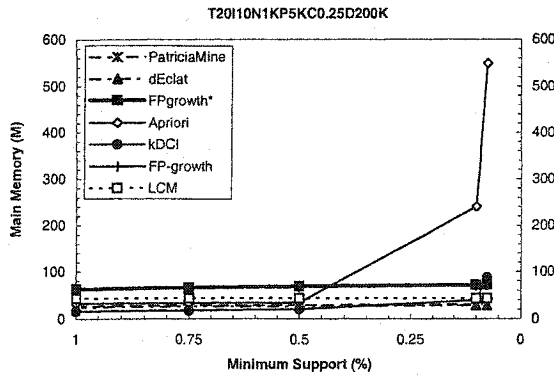

Figure 3.13: Memory Usage of Mining All FI's on *T20I10N1KP5KC0.25D200K*

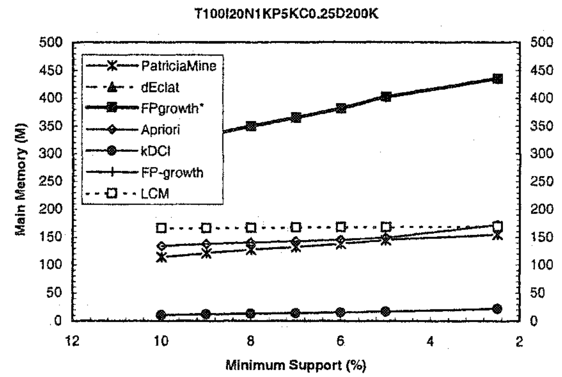

Figure 3.14: Memory Usage of Mining All FI's on *T100I20N1KP5KC0.25D200K*

The question of why FPgrowth* and the FP-growth method consume so much main memory when running on synthetic dataset can be answered as follows. In both figures the minimum support is pretty low, so there are many frequent single items in the datasets. Therefore wide and bushy trees have to be constructed for mining all frequent itemsets. Since the number of frequent single items almost stays the same as when the minimum support changes, the sizes of FP-trees remain almost the same, as we can see from the figures.

Comparing Figure 3.5 with Figure 3.13, we also can see that FPgrowth* still has good speed even it has to construct big FP-trees. However, from Figure 3.6 and Figure 3.14, we can see that the construction of FP-trees does influence the speed of FPgrowth* much.

When we implemented FPgrowth*, the FP-tree structure was implemented as a standard trie. In PatriciaMine, the FP-tree structure was implemented as a PatriciaMine trie. That's why it uses less main memory than FPgrowth*.

From Figure 3.15 to Figure 3.20, the FP-tree structure shows great compactness. Now, the main memory consumption of FPgrowth* is almost the same as that of the PatriciaMine and the FP-growth method. In the figures, we almost can not see the lines for the PatriciaMine and the FP-growth method because they were overlapped by the lines for FPgrowth*. From the experimental data, we see that for high minimum support, the memory used by FPgrowth* is even smaller than the memory used by PatriciaMine, which means Patricia trie needs more memory than standard trie when the compactness of FP-tree is very high.

From the figures, we also notice that when minimum support is low, the main memory used by algorithms such as Apriori and kDCI increases rapidly, while the main memory used by FPgrowth*

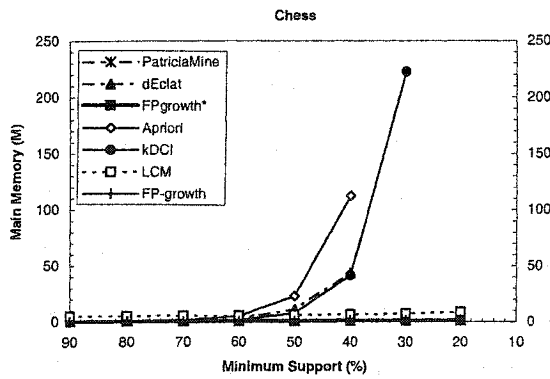

Figure 3.15: Memory Usage of Mining All FI's on *chess*

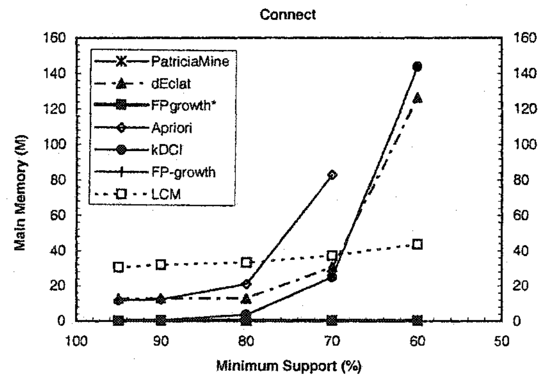

Figure 3.16: Memory Usage of Mining All FI's on *connect*

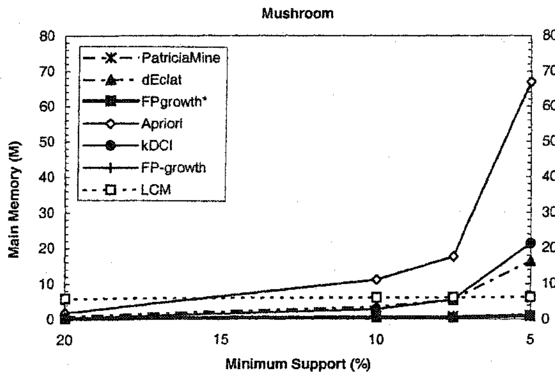

Figure 3.17: Memory Usage of Mining All FI's on *mushroom*

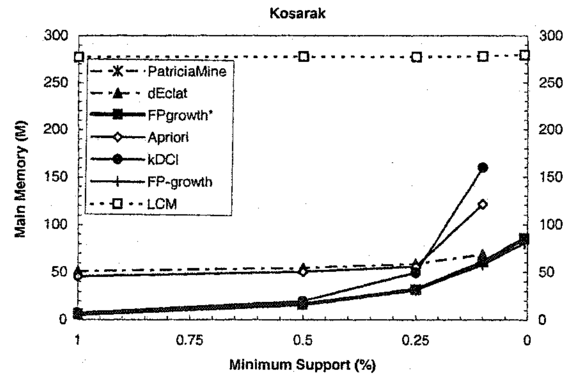

Figure 3.18: Memory Usage of Mining All FI's on *kosarak*

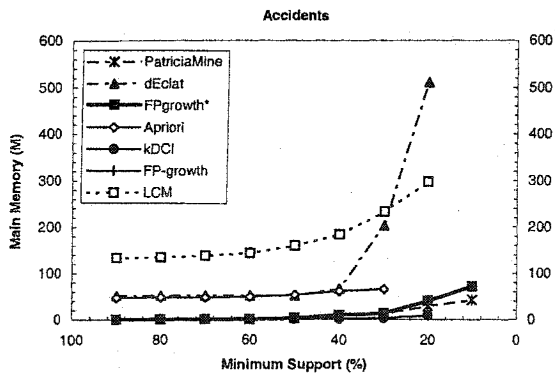

Figure 3.19: Memory Usage of Mining All FI's on *accidents*

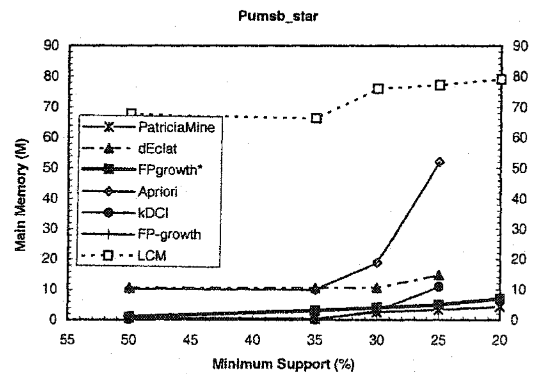

Figure 3.20: Memory Usage of Mining All FI's on *pumsb**

and PatriciaMine does not have a big change. This is because algorithms such as Apriori and kDCI have to keep large number of frequent itemsets and candidate frequent itemsets. The number of itemsets that needs to be kept increases exponentially when the minimum support becomes lower. For FPgrowth* and PatriciaMine, if the number of frequent single items does not change much when minimum support becomes lower, the size of FP-trees does not change much, either.

In figures 3.5 to 3.20, we see that the performance of almost all algorithms show big differences between synthetic datasets and real datasets. Similar situations can be seen later from the experimental results in Chapter 4, Chapter 5, and Chapter 6. People may ask these questions: why are there so big differences? and what are the synthetic datasets good for? The differences stem from the distribution differences between synthetic datasets and real datasets. When the synthetic datasets were generated, some distributions such as Poisson distribution and binomial distribution were used for generating a transaction. However, the real datasets used here do not follow those distributions. All of these datasets are skewed. In some dataset such as *connect*, many itemsets occur in more than 80% of the transactions. Though the real datasets are totally different from the synthetic datasets in this thesis, we can not say that there are no real datasets that are similar to the synthetic datasets. In other words, in real life, we still can find some real datasets that have similar distributions to the synthetic datasets. It still makes sense to run the algorithms on both synthetic datasets and real datasets.

Scalability

We also tested the scalability of all algorithms by running them on synthetic datasets. The number of transactions in the datasets for Figure 3.21 and Figure 3.22 ranges from 200K to 1 million. In all datasets, the number of items is 1000, the average transaction length is 20, the number of patterns used as generation seeds is 5000, average pattern length is 10, and the correlation between patterns was set as 0.25.

All algorithms ran on all datasets for minimum support 0.1%. Both runtime and memory consumption were recorded.

Figure 3.21 shows the speed scalability of all algorithms. In the figure, the lines for PatriciaMine, FPgrowth*, FPgrowth, and LCM overlap each other, which means that the four algorithms have almost the same scalability, which is also the best scalability. Runtime increases about 4 times when

the size of dataset increases 5 times, while runtime of algorithms such as kDCI increases about 10 times when the transaction number increases from 200K to 1 million.

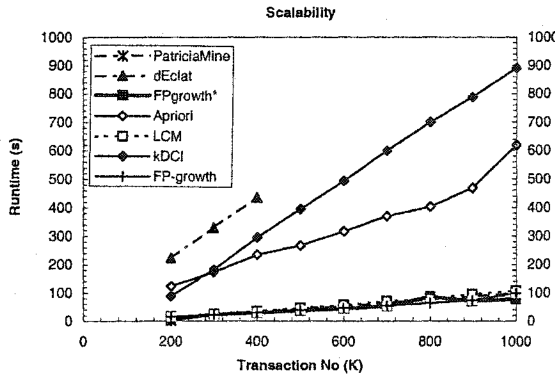

Figure 3.21: Scalability of runtime of Mining All FI's

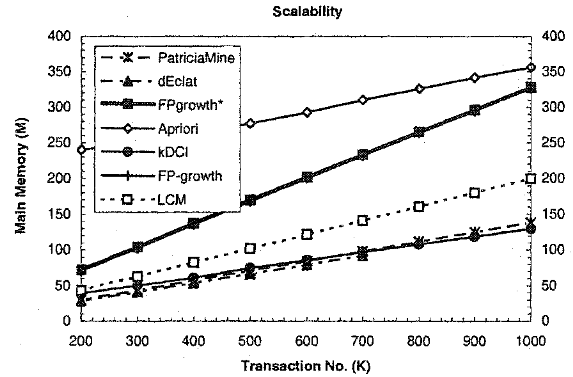

Figure 3.22: Scalability of Memory Usage of Mining All FI's

Figure 3.22 gives the main memory scalability of all algorithms. In the figure, the line for the FP-growth method overlaps the line for FPgrowth*, and the line for dEclat overlaps the line for PatriciaMine. The figure shows that main memory consumption of FPgrowth* and FP-growth increases linearly when size of datasets changes, which is not as good as the main memory consumption of other algorithms.

From all figures of runtime, main memory consumption and scalability of all algorithms, we can easily draw the conclusion that PatriciaMine is the best algorithm for mining all frequent itemsets since it is one of the fastest algorithms and it is always among the algorithms that use least main memory. FPgrowth* is also a stable algorithm and always among the fastest ones.

Chapter 4

Discovering Maximal Frequent Itemsets: First Attempt

The FP-growth method uses the FP-tree structure to mine all frequent itemsets efficiently. Since any subset of a frequent set is also frequent, it is sufficient to mine only the set of *maximal* frequent itemsets, instead of mining all frequent itemsets. In this Chapter, we study the performance of two existing approaches, GenMax and MAFIA, for mining maximal frequent itemsets. We also introduce a data structure, MFI-tree, for testing the maximality of a frequent itemset. Then we develop an extension, called FPmax, of the FP-growth method. Since one cannot expect that a single approach will be suitable for all types of data, we analyze the behavior of the three approaches GenMax, MAFIA, and FPmax, under various types of data. We validate our conclusions through careful experimentation with synthetic data, in which the parameters influencing the data characteristics are easily tunable.

On the basis of these conclusions, we then predict of the performance of each of the three methods for specific data characteristics. We test these predictions on real datasets, and find that they are valid in most cases.

4.1 The MFI-Tree

Obviously, compared with FPgrowth*, extra work needs to be done by FPmax is to check if a frequent itemset is maximal. The naive way to do this is during a post-processing step, as for example described in [55]. In [55], an algorithm is introduced for extracting all maximal elements in set of sets. The sets come in random order. If there are n sets, then getting all maximal sets takes at least $O(\sqrt{n}\log n)$ time. Post-processing usually is very costly. Instead, we introduce the *Maximal Frequent Itemset tree* (MFI-tree), to keep the track of MFI's. A newly discovered frequent itemset is inserted into the MFI-tree, unless it is a subset of an itemset already in the tree.

The MFI-tree resembles a FP-tree. Each MFI-tree has a root labeled "*root*". Children of the root are item prefix subtrees. In a FP-tree, each node in the subtree has three fields: item-name, count and node-link. In the MFI-tree, the count is replaced by the *level* of the node. The level-field will be useful for maximality testing. All nodes with the same item-name are linked together, as in a FP-tree. The node-link points to the next node with the same item-name. The MFI-tree also has a header table. The header table of a MFI-tree is constructed based on the item order in the header table of the FP-tree $T_\emptyset$ constructed from the original database. Each entry in the header table consists of two fields, item-name and head of a linked list. The head points to the first node with the same item-name in the MFI-tree.

The construction of the MFI-tree is described in Figure 4.1.

```
Procedure MFI_tree_construction(MFIs)
Input:  MFIs: set of MFI's, generated by FPmax
Output: A MFI-tree M
Method:
  create the root node of M
  create a header table from T∅.header
  for each itemset L in MFIs do begin
    sort the items in L according to the order of items in M.header
    let i= the first item in L, N = M.root
    repeat
       if i is not the item-name of any child of N
          create a new child node C for N, set C.item = i, set C.level as current level
          link C to the end of the link list for i starting from M.header
       let N = C, i=the next item in L
    until all items in L are processed
  end
```

Figure 4.1: Construction of a MFI-tree

To see how a MFI-tree is constructed, we take the FP-tree in Figure 2.4 as an example.

After the construction of the first FP-tree as in the FP-growth method, the method is called recursively for each frequent item in the header table in Figure 2.4 (b). The FP-trees corresponding to $\{c\}$, $\{e\}$, $\{f\}$ and $\{a\}$-conditional pattern bases each contain only a single branch, and therefore the recursion stops. But the FP-trees corresponding to $\{g\}$ and $\{d\}$-conditional pattern bases each contain multiple branches, and therefore the recursion should continue. In the following table we summarize the $\{i\}$-conditional pattern bases for each frequent item i in Figure 2.4(b) and its corresponding conditional FP-tree.

item	conditional pattern base	conditional FP-tree
c	$\{(gdb:1),(ab:1)\}$	$\{(b:2)\}$
e	$\{(gdb:1),(da:1)\}$	$\{(d:2)\}$
f	$\{(dab:1),(gab:1)\}$	$\{(a:2,b:2)\}$
g	$\{(db:2),(ab:1),(da:1)\}$	$\{(b:3,(d:2,a:1)),(d:1,a:1)\}$
d	$\{(b:2),(ab:1),(a:2)\}$	$\{((a:3,b:1),b:2)\}$
a	$\{(b:3)\}$	$\{(b:3)\}$
b	$\emptyset$	$\emptyset$

Figure 4.2 illustrates the construction of the MFI-tree for the example of Figure 2.4. In Figure 4.2, a node $x:\ell$ means that the node is for item x and its level is ℓ. Start from the first FP-tree T in Figure 2.4(b), by calling FPmax(T). Since T contains more than one path, a bottom-up search has to be done. For item c, its conditional FP-tree only has one single path, so we get the first frequent itemset $\{b,c\}$. Obviously this set is maximal, so it is inserted into the MFI-tree directly (Figure 4.2 (a)). Note in Figure 4.2 the header table in the MFI-tree is the same as that of the FP-tree in Figure 2.4, constructed from the database. Similarly, for item e and f, we can get maximal frequent itemsets $\{d,e\}$ and $\{b,a,f\}$. Figure 4.2(a) shows the tree after inserting (b,c), (d,e) and (b,a,f). When inserting (b,a,f), since (b,a,f) shares prefix b with (b,c), only two new nodes for a and f are inserted. Now for item g, from its conditional pattern base, another conditional FP-tree for g, T_g can be constructed. T_g is not a single path tree, so FPmax will be recursively called on T_g, it returns the g-conditional frequent itemsets $\{b,d,g\}$ and $\{a,g\}$, and since there is no link-chain for g, they are also maximal. We then insert $\{g,d,b\}$ into the MFI-tree (Figure 4.2 (b)). Similar item d, the $\{d\}$-conditional itemset is $\{b,d\}$ and $\{a,d\}$, and by calling *maximality_checking*, we determine that $\{b,d\}$ is a subset of an existing MFI, so it will not be inserted into the MFI-tree, only $\{a,d\}$ is

inserted into the MFI-tree. Finally, no itemsets in the conditional bases for $\{a\}$, or $\{b\}$ are maximal, no new MFI's will be inserted into the MFI-tree.

Every branch of the MFI-tree forms a MFI. Thus the MFI's in Figure 4.2 (b) are $\{c,b\}$, $\{b,a,f\}$, $\{b,d,g\}$, $\{e,d\}$, $\{d,a\}$ and $\{a,g\}$.

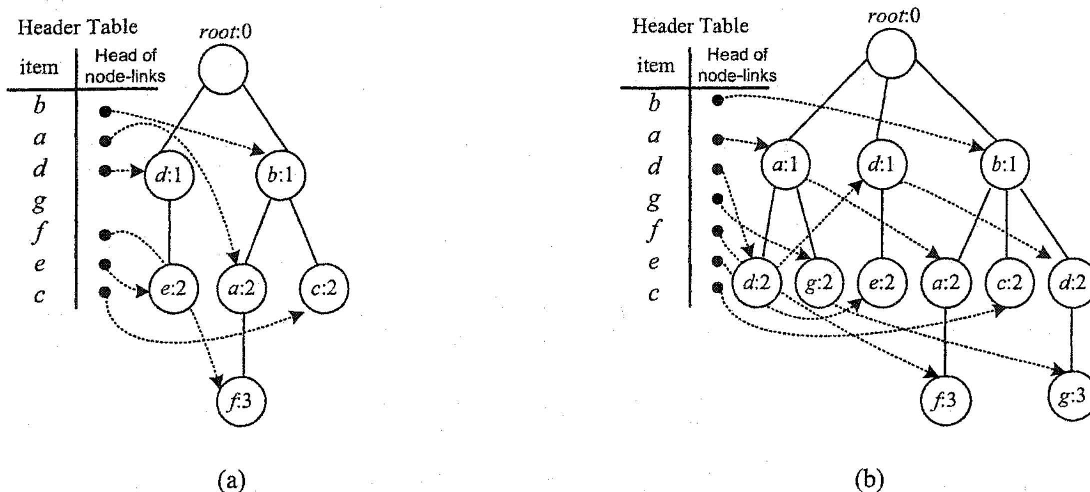

Figure 4.2: Construction of Maximal Frequent Itemset Tree

4.2 FPmax: Discovering maximal frequent itemsets

Extending the FP-growth method, we can get algorithm FPmax described in Figure 4.3. The procedure has a FP-tree T as parameter. The tree has attributes: *base*, *header*. $T.base$ contains the itemset X, for which T is a conditional FP-tree, the attribute *header* contains the head table.

FPmax is also a recursive algorithm. In it, a global MFI-tree is used to keep track of all MFI's. In the initial call, a FP-tree is constructed from the first scan of the database. Before recursively calling FPmax, we already know that the set containing all items in $T.base$ and the items in the FP-tree is not a subset of existing MFI's. If there is only one single path in the FP-tree, this single path together with $T.base$ is a MFI of the database. If the FP-tree is not a single-path tree, its least frequent item is appended to $T.base$, and line 6 calls function *maximality_checking* to check if the new base Y together with all frequent items in the $Y \cup Tail$ is subset of existing MFI's in the global MFI-tree. If not, FPmax will be called recursively.

Theorem 4.1 *FPmax returns all and only the maximal frequent itemsets in the given dataset.*

Procedure FPmax(T)
Input: T: a FP-tree
Global:
    M: a MFI-tree.
Output: The M that contains all MFI's
Method:
1. **if** T only contains a single path P
2.   insert $T.base \cup P$ into M
3. **else for each** i in the $T.header$ **do begin**
4.   set $Y = T.base \cup \{i\}$;
5.   Tail = {all frequent items in the i's conditional pattern base}
6.   **if not** *maximality_checking*($Y \cup Tail$, M);
7.   construct Y's conditional FP-tree T_Y;
8.   call FPmax(T_Y);
9. **end**

Figure 4.3: Algorithm FPmax

Proof. First of all, we prove that FPmax generates only MFI's. Before constructing any FP-tree, the FP-growth method [28] sorts all the items by their counts. Suppose the sorted frequent items are $i_1, i_2, \ldots, i_n$, frequent itemsets will be mined from i_n with minimal count to i_1 with the maximal count.

For every item i_k, to get all its frequent sets, all its conditional pattern bases can be extracted from the first FP-tree constructed from the database. These bases only contain items in $\{i_1, i_2, \ldots, i_{k-1}\}$, i.e., items that precedes i_k. Thus, any frequent itemsets in the bases only consists of items in $\{i_1, i_2, \ldots, i_{k-1}\}$. From these bases, another FP-tree corresponding to i_k can be constructed. We call all MFI's mined from this FP-tree i_k-MFI's. Any i_k-MFI cannot be subset of any MFI of items $i_1, i_2, \ldots, i_{k-1}$ that will be generated latter because any i_k-MFI contains i_k, while no MFI of items $i_1, i_2, \ldots, i_{k-1}$ can contain i_k. Thus, a frequent itemset generated from a single-path FP-tree cannot be a subset of any frequent itemset generated later, i.e., it is either maximal frequent set or a subset of some existing MFI's. Now we prove that FPmax generates all maximal frequent itemsets. In FPmax, if in line 2 it gives all subsets of $T.base \cup P$, and in line 6 *maximality_checking* is not called, all frequent itemsets will be generated, as in the FP-growth method. While line 2 will not lose any MFI, and the effect of function *maximality_checking* is to expunge non-maximal frequent itemsets, all MFI's will be preserved. Thus, FPmax will generate all MFI's. ∎

4.3 Maximality checking

In FPmax, function *maximality_checking* is called to check if $Y \cup Tail$ is a subset of some MFI's in the MFI-tree. If $Y \cup Tail$ is a subset of some MFI, then any frequent itemset generated from the FP-tree corresponding to Y could not be maximal, and thus we can stop mining MFI's for Y. By calling *maximality_checking*, we do superset frequency pruning.

Note that before and after calling *maximality_checking*, if $Y \cup Tail$ is not subset of any MFI, we still do not know if $Y \cup Tail$ is frequent or not. By constructing the FP-tree for Y from the conditional pattern bases of i, if the FP-tree only has a single path, we can conclude that $Y \cup Tail$ is frequent. Since $Y \cup Tail$ was not a subset of any previously discovered MFI, it is a new MFI and will be inserted to the MFI-tree.

To do maximality testing, one possibility is to always compare a set with the MFI's in the MFI-tree. However, we can do better. We found that most frequent sets are subsets of the latest MFI inserted into the MFI-tree. Therefore, each time we insert a new MFI into the MFI-tree, we keep a copy of this most resent MFI, any new frequent set will be compared with this set first. Only if the new set is not subset of the most recent MFI, the new set will be compared with the MFI's in the MFI-tree.

By using the header-table in the MFI-tree, a set S is not necessarily compared with all MFI's in the MFI-tree. First, S is sorted according to the order of items in header table. Suppose the sorted S is $\{i_1, i_2, \ldots, i_j\}$. From the header table, we find the node list for i_j. For each node in the the list, we test if S is a subset of the ancestors of the node. Note that both sets are ordered according to the header table, so this subset test can be done in linear time. Second, the level of node i_j can be used for saving comparison time. We test if the level of i_j is smaller than j. If it is, the comparison stops because there are not enough ancestors of i_j for matching the rest of S. This pruning technique is also applied as we move up the branch and toward the front of S. The function *maximality_checking* returns *false* if $\{i_1, i_2, \ldots, i_{j-1}\}$ is not a subset of any set in the MFI-tree.

4.4 Data characteristics and performance

In [5], Agarwal *et al.* have shown that DepthProject achieves more than one order of magnitude speedup over MaxMiner[9]. In [12], the performance numbers of MAFIA show that MAFIA outperforms DepthProject by a factor of three to five. In [16], Gouda and Zaki claimed that MAFIA is one of the best methods for mining a *superset* of all MFI's, and that GenMax is one of the best methods for enumerating the *exact* set of MFI's.

In the present section, we wish to reach an understanding of how the data characteristics influence the performance of MAFIA, GenMax, and our new algorithm FPmax. We first analyze the mining time used by the three algorithms.

We can divide the mining task in two parts. The first part consists of mining a superset of maximal frequent itemsets, and the second part is for pruning out non-maximal frequent itemsets. For FPmax, the time resources in the first part are invested in the construction of a FP-tree to concisely represent the database, and then extracting frequent itemsets from the FP-tree using the FP-growth method.

In the second part of the mining task, in order to extract the maximal frequent itemsets, the FPmax algorithm has to perform a large number of maximality tests. Suppose there are n items in header table. Then we know there are at most $C_n^{\lfloor n/2 \rfloor}$ maximal frequent itemsets. If we construct a MFI-tree for all these MFI's, the tree has height $\lfloor n/2 \rfloor$. In the first level, there are $C^1_{\lceil n/2 \rceil+1}$ nodes, in the second level, there are $C^2_{\lceil n/2 \rceil+2}$ nodes, in the ith level, there are $C^i_{\lceil n/2 \rceil+i}$ nodes, and in the last level, the $\lfloor n/2 \rfloor$th level, there are $C_n^{\lfloor n/2 \rfloor}$ nodes. Thus, the total number of nodes in the tree is

$$\sum_{i=1}^{\lfloor n/2 \rfloor} C^i_{\lceil n/2 \rceil+i} \qquad (1)$$

This is also an upper bound on the number of maximality tests needed in constructing the MFI-tree. Similar observations apply to the size of the FP-tree.

In the first part of the mining task, both GenMax and MAFIA construct a column-wise representation of the bitmap view of the database. To extract the frequent itemsets from the columns, MAFIA has to compute a number of bitvector *and*-operations, and *GenMax* does TIS intersections. If there are n items in the dataset, in the worst case, if the length of all MFI's is $n/2$, by a similar analysis as above, the total number of bitvector operations or TIS intersections could be equal to (1). However, a dense and a sparse dataset having the same number of maximal frequent itemsets,

will both require the same number of bitvector operations or TIS intersections.

For the second part of the mining task, GenMax and MAFIA differ in their approaches. GenMax extracts the maximal frequent itemsets during the TIS intersections, while MAFIA performs a post-processing step for the extraction.

Now let's see how the parameters of the benchmark synthetic data generator at [1] influence the performance of the three algorithms. The adjustable parameters include

- the average size of the transactions, also called *average transaction length*, ATL, and
- the average size of the maximal potentially large itemsets, also called *average pattern length*, APL.

We can think of the ATL as influencing the *density* of the dataset. The APL, on the other hand, determines the *average length* of MFI's that the dataset will contain. Thus, a long ATL generates a dense dataset, and a long APL gives long average MFI's. This gives us four categories of data.

1. **Short ATL, short APL.** In this dataset we can expect that each transaction will be fairly short, and that the MFI's will also be short. For FPmax, this could result in a costly bushy FP-tree.

   Now, if the minimum support is high, there might be only relatively few maximal frequent itemsets. This means that FPmax spent a considerable time effort to construct an FP-tree, from which only a small set of MFI's will be extracted. In this case, we can expect GenMax and MAFIA to be more efficient, since the ATL will not influence the time computing bitvector operations and set intersections.

   However, in the case where the minimum support is low, there might be numerous maximal frequent itemsets. Then the time FPmax invested in the FP-tree will pay off, since the size of the output (the MFI's represented as a MFI-tree) will also be large Now we expect FPmax to outperform the other two algorithms.

2. **Short ATL, long APL.** Here we expect that the transactions in the dataset are short, while the average length of the MFI's is close to ATL. For FPmax, this will result in a small FP-tree. Contrary to the first case, we can now efficiently extract the MFI-tree from the small FP-tree. We can also extract a relatively large MFI-tree from the small FP-tree. GenMax and MAFIA

on the other hand, will still need to do all the bitvector operations and set intersections they did in a dataset with short APL. Thus, we can expect that FPmax is the algorithm of choice for data with short ATL and long APL.

3. **Long ATL, short APL.** Now the transactions in the dataset are long. For FPmax, it will result in a very bushy and tall FP-tree. This will require more space and time than in the case of short ATL. Now, if the minimum support is high and we get small size MFI-tree, FPmax will not be efficient, GenMax or MAFIA will perform better. On the other hand, if the minimum support is low and we have a large output, FPmax may outperform GenMax and MAFIA.

4. **Long ATL, long APL.** For FPmax, both the FP-tree and the MFI-tree could be very large, which means we need more time to construct the FP-tree, and we need more comparisons to construct the MFI-tree. For this type of data, if it needs less time to post-process the superset of MFI's, MAFIA could be the best, otherwise, GenMax is the best.

4.4.1 Experiments on synthetic data

To test the accuracy of the analysis above, we ran three algorithms on synthetic datasets. The source codes for MAFIA and GenMax were provided by their authors. We used the application from [1] to generate synthetic datasets. For all datasets in this section, the number of transactions was fixed at 100,000, and the number of items was fixed at 1000. All experiments were performed on a 1GHz Pentium III with 512 MB of memory running RedHat Linux 7.3. All timings in the figures are CPU time.

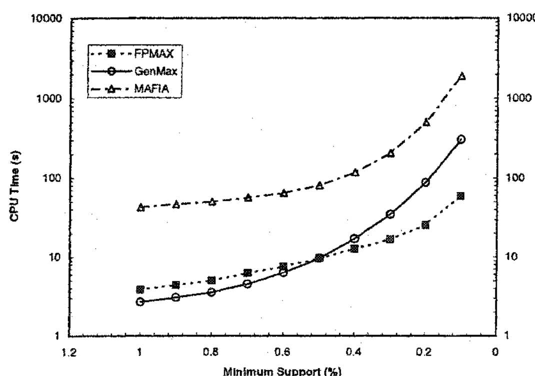

Figure 4.4: ATL=20, APL=20

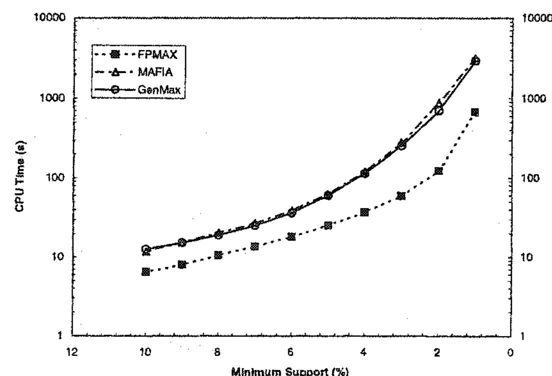

Figure 4.5: ATL=20, APL=100

The results of the first set of experiments are shown in Figure 4.4. We ran the algorithms on a

short ATL of 20, and short APL also of 20. We see that FPmax and GenMax outperform MAFIA 5-10 times. GenMax is faster than FPmax when the minimum support is high, and when minimum support is low, FPmax is faster. For minimum support 0.1%, FPmax is 5 times faster than GenMax, while for minimum support 1%, GenMax is only about 1.5 times faster than FPmax.

The synthetic data for the second set of experiments, displayed in Figure 4.5, has a short (20) ATL and a long (100) APL. In this case, the performance of MAFIA and GenMax is almost the same. FPmax is clearly the most efficient on this dataset. FPmax outperforms the other two by factor of at least two, both for high and low minimum support.

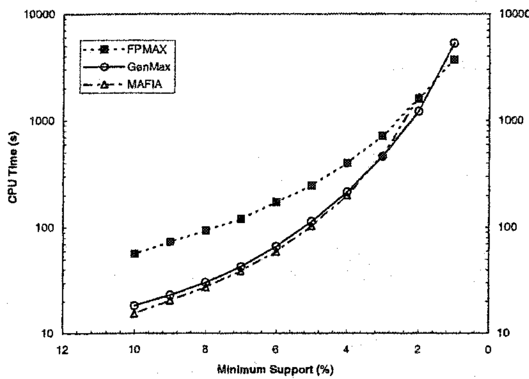

Figure 4.6: ATL=100, APL=20

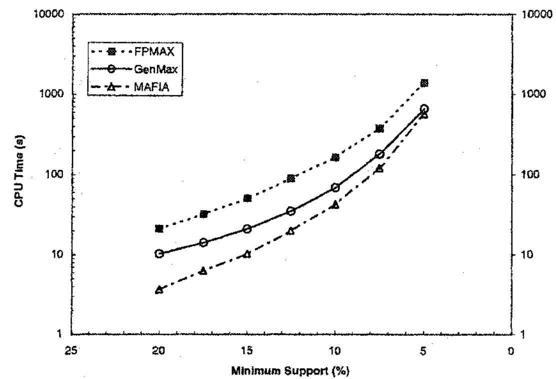

Figure 4.7: ATL=100, APL=100

Figure 4.6 shows a totally different figure for algorithms on dataset with a long ATL (100) and a short APL (20). We can see that FPmax is slower than other algorithms for most time. On this type of data, MAFIA or GenMax is the best for high minimum support, while FPmax tends to be faster than other two for low minimum support.

We also ran the algorithms on the dataset with long (100) ATL and long (100) APL. On this type of data, from Figure 4.7 we can see that MAFIA seems to be the best, although GenMax performs well too. FPmax seems to be slow at all time even when the minimum support is low.

For the next two experiments we fixed a low minimum support of 1%.

Figure 4.8 shows the result for the datasets generated by fixing ATL to 20 and varying APL from 20 to 100. In these experiments, FPmax has better performance than the other two algorithms. MAFIA and GenMax have the same tendency, although GenMax is faster than MAFIA.

Then we generated the second datasets by fixing APL to 20 and varying APL from 20 to 100. Figure 4.9 show the result. We can see that FPmax is slightly faster than GenMax, while MAFIA

is distinctly slower than the other two algorithms.

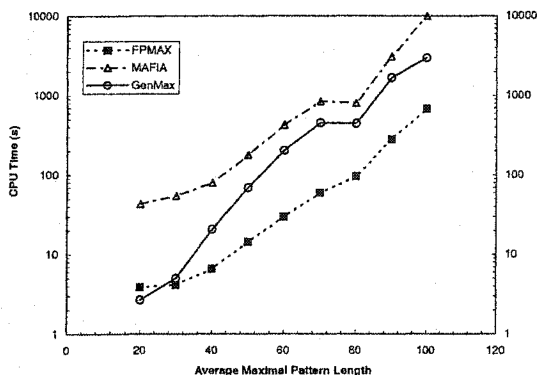

Figure 4.8: ATL=20, minimum support=1%

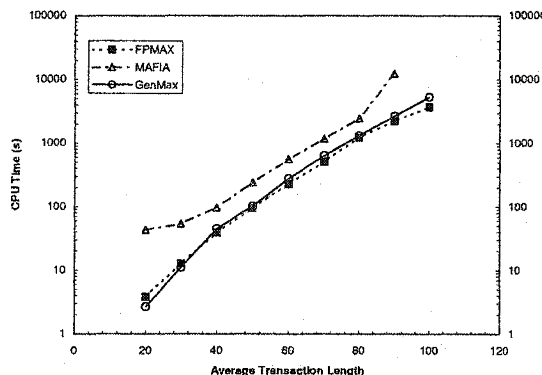

Figure 4.9: APL=20, minimum support=1%

The results of our experiments can be summarized in Figure 4.10, which gives the best algorithm for the various types of data.

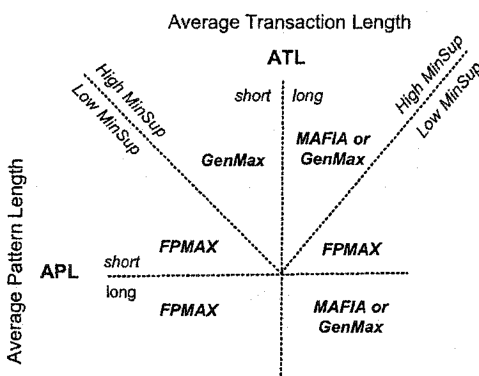

Figure 4.10: Best algorithms for different type of data.

4.4.2 Experiments on real datasets

Next, we ran the programs on four real datasets, *chess*, *connect*, *mushroom*, and *pumsb**, which were used in Chapter 3. These real datasets are all very dense, so many MFI's can be mined even for very high values of minimum support.

Figure 4.11 to Figure 4.14 show the performance of the three algorithms on these real datasets. Figure 4.11 shows the experimental results on *mushroom*. Here FPmax outperforms the other algorithms, for all levels of minimum support. In dataset mushroom, the average transaction length is 23, and the average MFI's length ranges from 8 to 19 for minimum support 10% to 0.1%. We can categorize this dataset as having short ATL, and long APL.

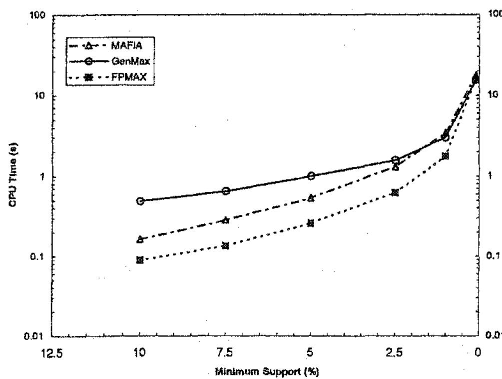

Figure 4.11: Running FPmax, GenMax and MAFIA on dataset *mushroom*

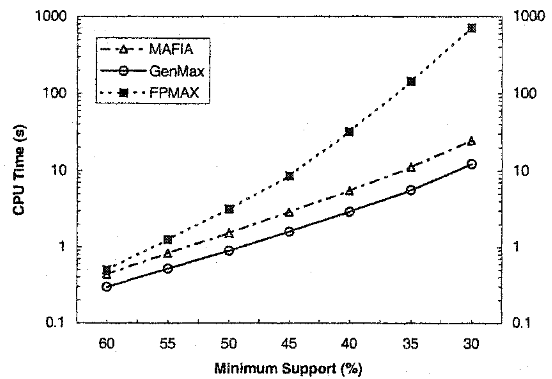

Figure 4.12: Running FPmax, GenMax and MAFIA on dataset *chess*

The results for the *chess* dataset is shown in Figure 4.12. The ATL of the dataset is 37 while the average length of MFI's is up to 12, which means both ATL and APL are long, so FPmax is not expected to perform well with this dataset. Also, here MAFIA needs more work in post-processing, so GenMax is the best algorithm for *mushroom* dataset.

In the dataset *connect*, though the ATL of this dataset is 43, which is fairly long, and the average length of the MFI's is 9 to 21 for minimum support 90% to 10%, which also is fairly long, nevertheless FPmax is the best algorithm for high minimum support, and GenMax is the best for low minimum support. This result does not fit the rule in Figure 4.10. We conjecture that *connect* has a skewed distribution, different from binomial or exponential distributions that are used to generate synthetic datasets [7]. By checking the size of the FP-tree and the number of maximality tests needed in constructing the MFI-tree, we found that they are both far smaller than those for synthetic dataset with ATL equal to 40.

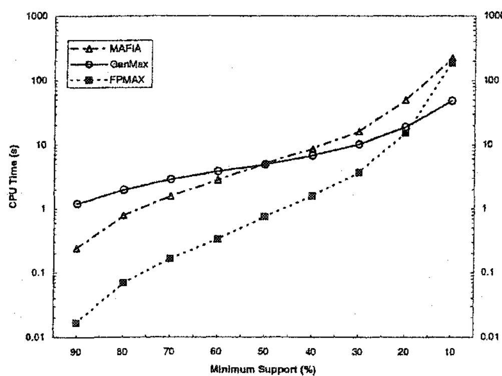

Figure 4.13: Running FPmax, GenMax and MAFIA on dataset *connect*

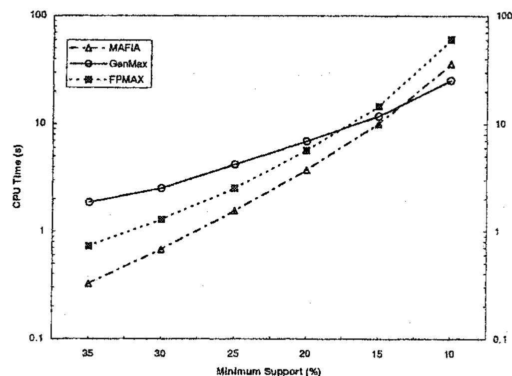

Figure 4.14: Running FPmax, GenMax and MAFIA on dataset *pumsb**

The *pumsb** dataset is also skewed, long (50) ATL and long APL (average length of MFI's is 7 to 14 for minimum support 35% to 10%). As we can see from Figure 4.14, for high minimum support, the MAFIA is the most efficient, followed by FPmax, and then GenMax.

Based on the experiments with datasets *connect* and *pumsb**, it appears that the predictions in Figure 4.10 do not hold. This is because the two datasets are very skewed. Some itemsets occur in 80% of the transactions in the datasets. Though for each dataset, its average transaction length and its average pattern length match one case in Figure 4.10, the data distribution in the dataset is still very different from the data distribution in the corresponding synthetic dataset. However, when the data distribution of a real dataset is similar to the data distribution of a synthetic dataset, we can predict the fastest algorithm for the real dataset by Figure 4.10.

4.4.3 Scalability of the algorithms

To test the scalability of three algorithms, we also ran the programs on both synthetic and real datasets, while varying the number of transactions in datasets.

For the synthetic datasets, we set ATL to 10, and APL to 4, and vary the number of transactions from 100,000 to 1,000,000. We chose a minimum support of 0.05%, because for this level FPmax and GenMax use almost the same amount of CPU time. Figure 4.15 shows that the mining time increases almost linearly for all three algorithms, while MAFIA and GenMax show a steeper increase than FPmax.

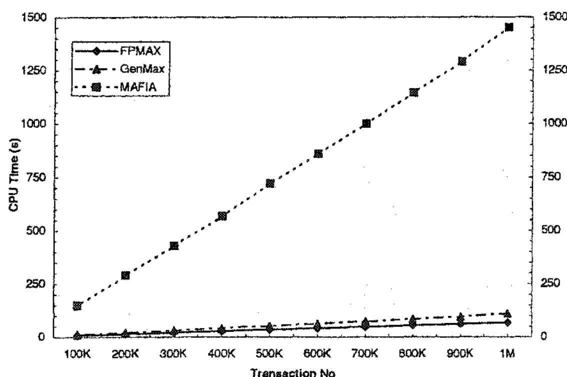

Figure 4.15: Scalability of FPmax, GenMax and MAFIA Running on Synthetic Datasets

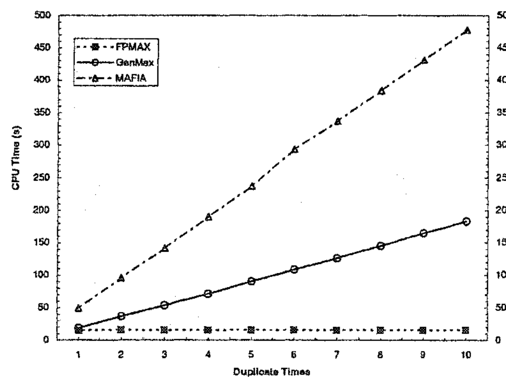

Figure 4.16: Scalability of FPmax, GenMax and MAFIA on Duplicated Real Datasets

The steeper increase for MAFIA and GenMax in Figure 4.15 is not accidental. For synthetic datasets, if we increase the number of transactions and keep other parameters unchanged, we can

expect more similar transactions, while the number of MFI's will not increase much. For FPmax, adding transactions similar to the existing ones will not increase the sizes of the FP-tree and MFI-tree much, while it does increase the cost of set intersections because the sets now become long. In the extreme case, if we increase the dataset by adding transactions equal to those that are already in the dataset, we can expect that the CPU time for FPmax will remain unchanged, while it will increase for GenMax and MAFIA. Figure 4.16 shows the result on dataset which is generated by duplicating the real dataset *connect* two to ten times. From the figure, we can see that the line for FPmax is flat while the CPU time for the other two algorithms increase rapidly. From Figure 4.16 we can also see that leaving the pruning of non-maximal frequent itemsets to a post processing step, as MAFIA does, increases the amount of work.

Chapter 5

Discovering Maximal Frequent Itemsets: Second Attempt

In Chapter 4 we developed FPmax, another method that mines maximal frequent itemsets using the FP-tree structure. In FPmax, a data structure, MFI-tree, was introduced. A global MFI-tree was set to keep all maximal frequent itemsets. Any candidate maximal frequent itemset was compared with the itemsets stored in the MFI-tree. The experimental results in Chapter 4 showed that FPmax does not always outperform GenMax [16] and MAFIA [12]. In this chapter, we extend FPmax to FPmax* by applying FP-array technique introduced in Chapter 3. Furthermore, we present a more efficient method to test if a frequent itemset is maximal. The performance of FPmax*, namely its speed, main memory consumption, and scalability, is studied.

5.1 FPmax*: Mining MFI's

In FPmax, only one global MFI-tree is constructed, a newly discovered frequent itemset is inserted into the MFI-tree, unless it is a subset of an itemset already in the tree. However, for large datasets, the MFI-tree can be quite large, and sometimes one itemset needs thousands of comparisons for maximality testing. Inspired by the method maximality checking is done in [16], we developed FPmax* method, in which for each conditional FP-tree T_X, a small MFI-tree M_X is created. The tree M_X will contain all maximal itemsets in the conditional pattern base of X. To see if a local

MFI Y generated from a conditional FP-tree T_X is maximal, we only need to compare Y with the itemsets in M_X. This achieves a significant speedup of FPmax.

We called the new method FPmax* because we can expect that it outperforms FPmax while using the FP-array technique. But in addition to it, FPmax* has a more efficient maximality test, as well as a number of other optimizations.

FPmax* has two parameters, one is a FP-tree T, another one is M. Besides the two attributes *base* and *header*, the FP-tree now has another attribute *FP-array*. *T.FP-array* contains the FP-array A_X. *T.base* contains the itemset X, for which T is a conditional FP-tree, and the attribute *header* contains the head table. Note that In FPmax*, the header table of M_X is constructed based on the item order in the table of T_X.

Procedure *FPmax*(T, M)*
Input: T, a FP-tree
    M, the MFI-tree for *T.base*
Output: Updated M
Method:
1. **if** T only contains a single path P
2. insert P into M
3. **else for each** i in *T.header* **do begin**
4. set $Y = T.base \cup \{i\}$;
5. **if** *T.FP-array* is not NULL
6. *Tail*={frequent items for i in *T.FP-array*}
7. **else**
8. *Tail*={frequent items in i's conditional pattern base}
9. sort *Tail* in decreasing order of the items' counts
10. **if not** *maximality_checking*($Y \cup Tail, M$)
11. construct Y's conditional FP-tree T_Y and its FP-array A_Y;
12. initialize Y's conditional MFI-tree M_Y;
13. call *FPmax**(T_Y, M_Y);
14. merge M_Y with M
15. **end**

Figure 5.1: Algorithm FPmax*

Figure 5.1 gives algorithm FPmax*. In the figure, *T.base* contains the items that form the conditional basis of the current call. The first call will be for the FP-tree constructed from the original database, and it will have an empty MFI-tree. Before a recursive call *FPmax*(T,M)*, we already know from line 10 that the set containing *T.base* and the items in the current FP-tree is not a subset of any existing MFI. During the recursion, if there is only one single path in T, this single path together with *T.base* is a MFI of the database. In line 2, the MFI is inserted into M. If the FP-tree is not a single-path tree, then for each item i in the header table, we start preparing for

the recursive call *FPmax**(T_Y, M_Y), for $Y = T.base \cup \{i\}$. The items in the header table of T are processed in increasing order of frequency, so that maximal frequent itemsets will be found before any of their frequent subsets. Lines 5 to 8 use the FP-array technique, and line 10 calls function *maximality_checking* to check if Y together with all frequent items in Y's conditional pattern base is a subset of any existing MFI in M (thus we do superset pruning here). If *maximality_checking* return false, FPmax* will be called recursively, with (T_Y, M_Y). The implementation of function *maximality_checking* will be explained shortly.

Note that before and after calling *maximality_checking*, if $Y \cup Tail$ is not subset of any MFI, we still do not know whether $Y \cup Tail$ is frequent. If, by constructing Y's conditional FP-tree T_Y, we find out that T_Y only has a single path, we can conclude that $Y \cup Tail$ is frequent. Since $Y \cup Tail$ was not a subset of any previously discovered MFI, it is a new MFI and will be inserted into M_Y.

5.2 Maximality checking

The function *maximality_checking* works similarly as in FPmax. Except now $Y \cup Tail$ will be compared with a smaller MFI-tree.

Unlike a FP-tree, which is not changed during the execution of the algorithm, a MFI-tree is dynamic. At line 12, for each Y, a new MFI-tree M_Y is initialized from the predecessor MFI-tree M. Then after the recursive call, M is updated on line 14 to contain all newly found frequent itemsets. In the actual implementation, we however found that it was more efficient to update all MFI-trees along the recursive path, instead of merging only at the current level. In other words, we omitted line 14, and instead on line 2, P is inserted into the current M, and also into all predecessor MFI-trees that the implementation of the recursion needs to keep in main memory in any case.

For more details, on line 12, when a MFI-tree M_{Y_j} for $Y_j = i_1 i_2 \ldots i_j$ is created for next call of FPmax*, we know that conditional FP-trees and conditional MFI-trees for $Y_{j-1} = i_1 i_2 \ldots i_{j-1}$, $Y_{j-2} = i_1 i_2 \ldots i_{j-2}, \ldots, Y_1 = i_1$, and $Y_0 = \emptyset$ are all in main memory. To make M_{Y_j} store all already found MFI's that contain Y_j, M_{Y_j} is initialized by extracting MFI's from $M_{Y_{j-1}}$. The initialization can be done by following the linked list for i_j from the header table of $M_{Y_{j-1}}$, and extracting the maximal frequent itemsets containing i_j. Each such found itemset I is sorted according to the order of items in M_{Y_j}'s header table (the same order as T_{Y_j}'s header table), and then inserted into M_{Y_j}.

On line 2 we have found a new MFI P in T_{Y_j}, so P is inserted into M_{Y_j}. Since $Y_j \cup P$ also

contains $Y_{j-1}, \ldots, Y_1$, Y_0 and the trees $M_{Y_{j-1}}, \ldots, M_{Y_1}, M_{Y_0}$ are all in main memory, to make these MFI-trees consistently store all already discovered MFI's that contain their corresponding itemset, for each $k = 0, 1, \ldots, j$, the MFI $P \cup (Y_j - Y_k)$ is inserted into the corresponding MFI-tree M_{Y_k}.

At the end of the execution of FPmax*, the MFI-tree M_{Y_0} (i.e. $M_\emptyset$) contains all MFI's mined from the original database.

Since FPmax* is a depth-first algorithm, it is straightforward to show that the above maximality checking is correct. Based on the correctness of the FPmax method, we can conclude that FPmax* returns all and only the maximal frequent itemsets in a given dataset.

5.3 An optimization

In the method FPmax*, one more optimization is used. Suppose, that at some level of the recursion, the header table of the current FP-tree is $i_1, i_2, \ldots, i_m$. Then starting from i_m, for each item in the header table, we may need to do the work from line 4 to line 14. If for any item, say i_k, where $k \leq m$, its maximal frequent itemset contains items $i_1, i_2, \ldots, i_{k-1}$, i.e., all the items that have not yet called FPmax* recursively, these recursive calls can be omitted. This is because for those items, their tails must be subsets of $\{i_1, i_2, \ldots, i_{k-1}\}$, so $maximality_checking(Y \cup Tail)$ would always return true.

FPmax* also uses the memory management described in Section 3.3, for allocating and deallocating space for FP-trees and MFI-trees.

5.4 Discussion

One may wonder if the space required for all the MFI-trees of a recursive branch is too large. Actually, before the first call of FPmax*, the first FP-tree has to fit in main memory. This is also required by the FP-growth method. The corresponding MFI-tree is initialized as empty. During recursive calls of FPmax*, new conditional FP-trees are constructed from the first FP-tree or from an ancestor FP-tree. From the experience of [28], we know the recursively constructed FP-trees are relatively small. If we keep all the MFI's in main memory, we can expect that the total size of those FP-trees is not greater than the final size of the MFI-tree for $\emptyset$. For the same reason, we can expect that the MFI-trees constructed from ancestors are also small. During the mining, the MFI-tree for

$\emptyset$ grows gradually, other small descendant MFI-trees will be constructed for recursive call and then discarded after the call. Thus we can conclude that the total main memory requirement for running FPmax* on a dataset is proportional to the sum of the size of the FP-tree and the MFI-tree for $\emptyset$.

The above discussion is relevant for the case when we keep all MFI's in main memory throughout the mining process. We call the approach the *basic* approach. Note that FPmax* is a depth-first algorithm. This means that the FP-trees in main memory can be trimmed to give space for newly constructed FP-trees and MFI-trees. For each item in the header table of a FP-tree, after a recursive call of FPmax*, since no information for this item will be used in the remaining mining, all nodes for the item can be deleted from the FP-tree. This can be done by following the linked list for the item in the header table. At the same time, the corresponding cells for the item in the FP-array described in Section 3.1 can also be deleted. For the same reason, in a MFI tree, after a recursive call of FPmax* for an item, all nodes for the item can be deleted from the tree. Thus, the size of a FP-tree and the FP-array of the FP-tree becomes smaller and smaller; in the end of the recursion, the size of a FP-tree is 0. The MFI-trees grow at the beginning of the recursion. Gradually, at some point, the MFI-trees become smaller and smaller since less items are stored in the MFI-trees, and finally, they become 0 as well. In the best case, the total main memory used by FPmax* is approximately the size of the FP-tree for $\emptyset$, which is the same as the main memory requirement for FPgrowth*. This happens when the biggest MFI-tree constructed during the call is not greater than the space saved by trimming the FP-trees. We call this approach the *trimming* approach.

Unfortunately, if we want to save main memory by the trimming approach, we can not apply the main memory management discussed in Section 3.3. There, the main memory chunk for an entire FP- or MFI-tree is discarded at a time, so a FP-tree, a FP-array or a MFI-tree can not be trimmed during recursive call. Therefore, there is a trade-off between main memory and speed. A compromise could be as follows. During a call, we do not trim the FP-trees and MFI-trees, we let the MFI-trees grow slowly. When a MFI is inserted, only part of the MFI is inserted into the MFI-trees. Suppose i_k is the current item, and from current call $\{i_k\} \cup Z$ is found to be a MFI, then only Z is inserted into the MFI-tree for $\emptyset$. We call this approach the *compromise* approach. By compromise approach, the size of MFI-trees is reduced. As an example, Figure 5.2 shows the size-reduced MFI-tree for $\emptyset$ associated with the dataset in Figure 2.4. Shown in Figure 4.2(b) is the complete MFI-tree for $\emptyset$ which has 12 nodes, while the size-reduced MFI-tree has only 6 nodes.

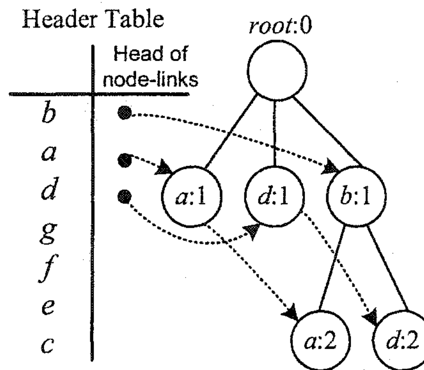

Figure 5.2: Size-reduced Maximal Frequent Itemset Tree

In the basic approach, the MFI-tree for $\emptyset$ stores all MFI's in D. Compared with the trimming approach and the compromise approach, it needs the highest amount of main memory. The trimming approach trims FI- and MFI-trees. It needs the lowest amount of main memory, but it takes the longest CPU time. The compromise approach keeps only part of a MFI into MFI-trees. Compared with the basic approach, the compromise approach inserts fewer nodes into MFI-trees, so it needs less main memory and it is faster than the basic approach. Though it is still needs more main memory than the trimming approach, since we can apply main memory management as described in Section 3.3, we believe that it is faster than the trimming approach.

In Section 5.5, in the experiments, FPmax* is implemented by using the compromise approach. The experimental results show that main memory requirement is drastically low for some datasets compared with the basic approach.

5.5 Experimental evaluation

In this section, we present a performance comparison of algorithm FPmax* with other algorithms. We first compare the performance of FPmax*, FPmax, MAFIA and GenMax, as they were compared in Chapter 4. We then compare FPmax* with some of the best algorithms presented in FIMI'03 [21].

5.5.1 FPmax* versus FPmax, MAFIA and GenMax

We ran the four algorithms on two synthetic datasets and two real datasets. All datasets were used in Section 3.4.1. The two synthetic datasets are *T40I10D100K* and *T100I20D100K*. The two real

datasets are *pumsb** and *connect.*

Figure 5.3 gives the result for running these algorithms on the sparse dataset *T40I10D100K.* We can see that FPmax is slower than GenMax for all levels of minimum support, while FPmax* outperforms GenMax by a factor of at least two. Figure 5.4 shows the results for the very sparse dataset *T100I20D100K.* In it, FPmax is the slowest algorithm, while FPmax* is the fastest algorithm.

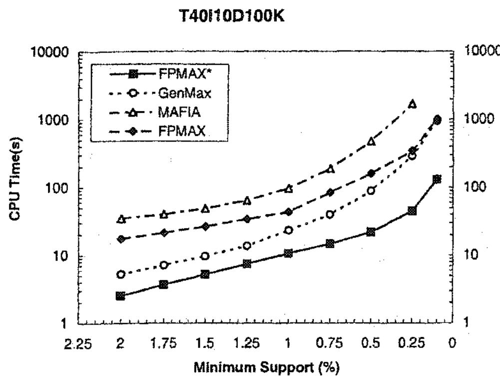

Figure 5.3: dataset *T40I10D100K*

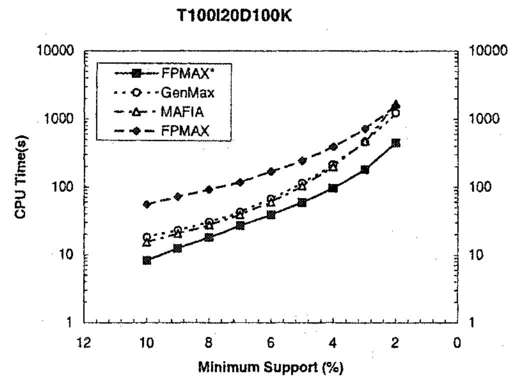

Figure 5.4: dataset *T100I20D100K*

Figure 5.5 shows that FPmax* is the fastest algorithm for the dense dataset *pumsb**, even though FPmax is the slowest algorithm on this dataset for very low levels of minimum support. In Figure 5.6, FPmax outperforms GenMax and MAFIA for high levels of minimum support, but it is slow for very low levels. FPmax*, on the other hand is about one to two orders of magnitude faster than GenMax and MAFIA for all levels of minimum support.

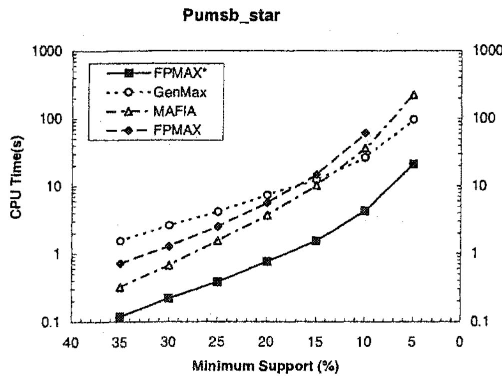

Figure 5.5: dataset *pumsb**

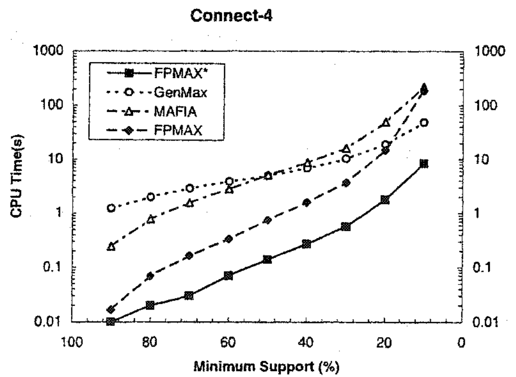

Figure 5.6: dataset *connect*

5.5.2 FPmax* versus other algorithms

The FIMI'2003 [2] workshop also included a contest to find the most efficient algorithms for mining maximal frequent itemset. In the independent experiments conducted by the organizers, the performance of 8 algorithms for this category, including optimized version of Apriori, MAFIA, FP-growth and Eclat, were compared. From the experimental results [14, 15], we can see that FPmax* outperformed all the other algorithms submitted to the workshop for most datasets.

Since the organizers in FIMI'03 did not compare the scalability of the algorithms, and their experimental results of memory consumption of algorithms were not published, we repeated part of the experiments in FIMI'03. Some algorithms in FIMI'03 had apparently very bad performance. For this reason only 4 algorithms, MAFIA [12], LCM [52], AFOPT [35], and GenMax [16], together with FPmax* [21], were chosen and compared. FPmax* was implemented by keeping only part of a MFI in MFI-trees, as explained in Section 5.4, to save main memory and CPU time.

In addition to these 5 algorithms, SmartMiner algorithm was included in the experiment. Introduced in [61], it was claimed to be an order of magnitude faster than MAFIA and GenMax, two of the fastest algorithms for mining maximal frequent itemsets. Since the authors implemented the algorithm in Java, we implemented SmartMiner ourselves in C++. The performance of all 6 algorithms was compared in the following experiments.

We ran the six algorithms on the datasets used in Chapter 3, i.e., two synthetic datasets: *T20I10N1KP5KC0.25D200K* and *T100I20N1KP5KC0.25D200K*, and six real datasets: *accidents, kosarak, chess, connect, mushroom*, and *pumsb**.

All experiments were performed on a DELL Inspiron 8600 laptop with Pentium M, 1.6 GHz Processor, and 1GB of memory. Both time and memory consumption of each algorithm running on each dataset were recorded. Runtime was recorded by "time" command, and memory consumption was recorded by "memusage". In the figures, if there is no data for time of algorithm A running on dataset B for some minimum support δ, it is either because A needs too much time (longer than 30 minutes) to run on B for minimum support δ, or the implementation of the A has some bugs. There will be no memory consumption data either for A running on B for minimum support δ in this case.

The runtime

Figure 5.7 gives the result for running 6 algorithms on the dataset *T20I10N1KP5KC0.25D200K*. In the figure, FPmax* is the fastest algorithm, for both low and high minimum support. SmartMiner is as fast as FPmax* for high minimum support, but its runtime is almost two times of the runtime of FPmax* for low minimum support. Nevertheless, SmartMiner is faster than GenMax and MAFIA an order of magnitude for low minimum support.

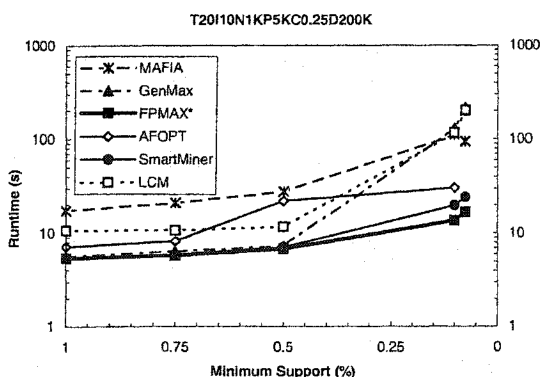

Figure 5.7: Runtime of Mining Maximal FI's on *T20I10N1KP5KC0.25D200K*

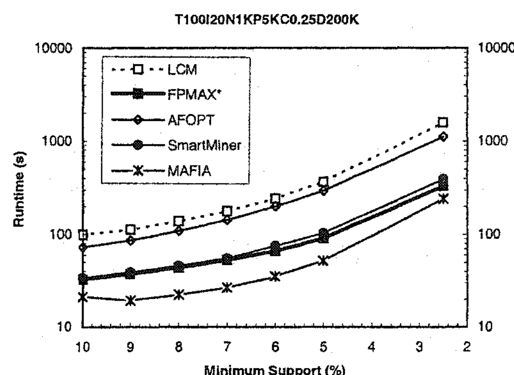

Figure 5.8: Runtime of Mining Maximal FI's on *T100I20N1KP5KC0.25D200K*

In Figure 5.8, MAFIA is the fastest algorithm for dataset *T100I20N1KP5KC0.25D200K*. SmartMiner is still as fast as FPmax* for high minimum support, but slower than FPmax* for low minimum support.

Since in *T100I20N1KP5KC0.25D200K*, the average transaction length and average pattern length are pretty long, FPmax* has to construct bushy FP-trees from the dataset, which can be seen from Figure 5.16. The constructing time and the time for traversing the FP-trees dominates the whole mining time, especially for high minimum support. On the contrary, MAFIA does not have much workload for high minimum support, fewer maximal frequent itemsets and candidate maximal frequent itemsets will be generated. However, as shown in Figure 5.8, when the minimum support is low, we can expect that FPmax* outperforms MAFIA because now the construction of the big FP-tree offers a big gain by large number of maximal frequent itemsets.

Figure 5.9 to Figure 5.14 show the experimental results of running the 6 algorithms on real datasets. In the figures, FPmax* shows the best performance on almost all datasets, for both high and low minimum support, benefiting from the great compactness of the FP-tree structure on dense datasets. SmartMiner has similar performance to FPmax* when minimum support is high, and it

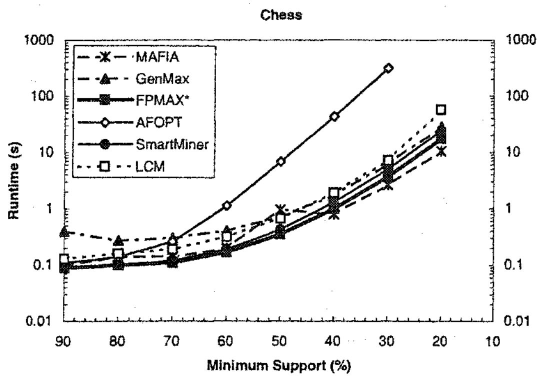

Figure 5.9: Runtime of Mining Maximal FI's on *Chess*

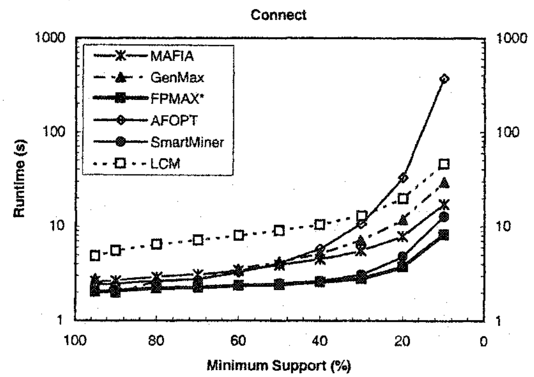

Figure 5.10: Runtime of Mining Maximal FI's on *Connect*

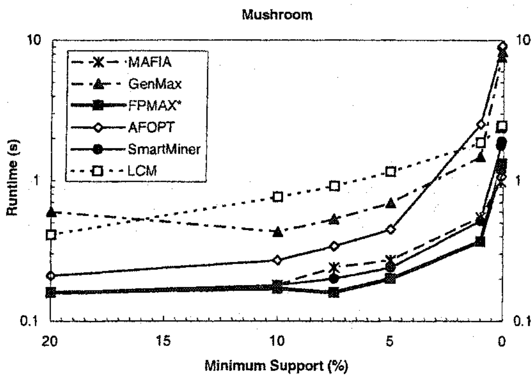

Figure 5.11: Runtime of Mining Maximal FI's on *Mushroom*

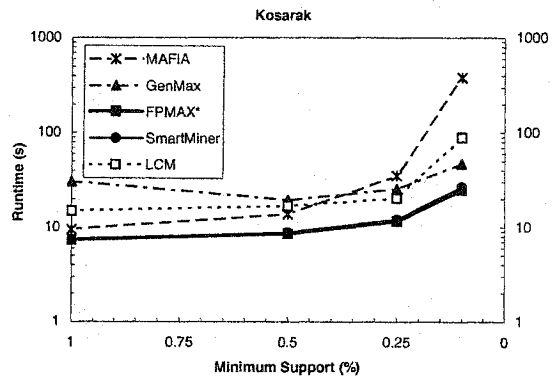

Figure 5.12: Runtime of Mining Maximal FI's on *Kosarak*

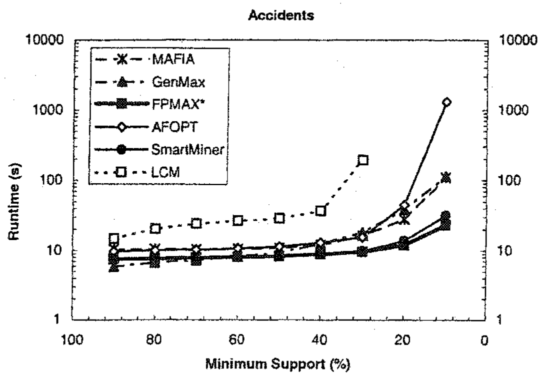

Figure 5.13: Runtime of Mining Maximal FI's on *Accidents*

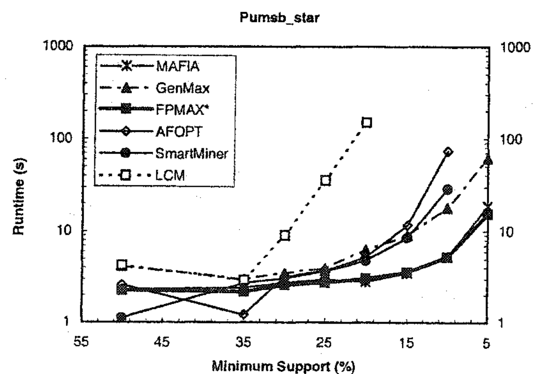

Figure 5.14: Runtime of Mining Maximal FI's on *Pumsb**

is slower than FPmax* when minimum support is low. In Figure 5.12, there is a overlap between the lines for the two algorithms, which means the two algorithms have almost the same speed on the dataset. The overlap in Figure 5.14 between the lines for MAFIA and FPmax* also shows the similar CPU time of the two algorithms on dataset *Pumsb**.

All experiments on both synthetic and real datasets show that the FP-array technique and the *maximality_checking* function are indeed very effective.

Memory Consumption

Another criterion of our comparison is the memory consumption of all algorithms for mining maximal frequent itemsets.

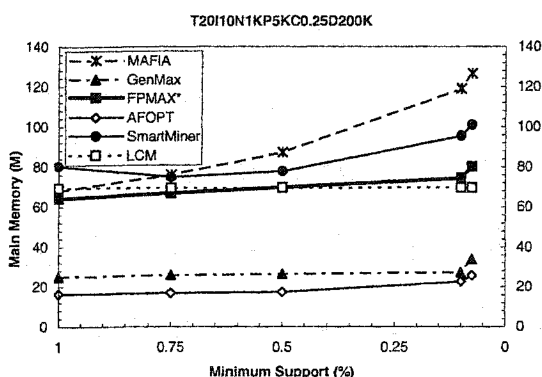

Figure 5.15: Memory Consumption of Mining Maximal FI's on *T20I10N1KP5KC0.25D200K*

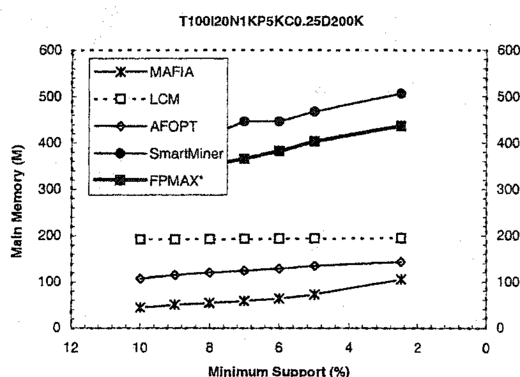

Figure 5.16: Memory Consumption of Mining Maximal FI's on *T100I20N1KP5KC0.25D200K*

Similar to the memory consumption for mining all frequent itemsets on synthetic datasets, here FPmax* still uses much main memory for mining maximal frequent itemsets. The reason was explained in Section 3.4.2 for figures 3.13 and 3.14.

By comparing Figure 3.13 and Figure 5.15, Figure 3.14 and Figure 5.16, we also can see that MFI-trees containing all maximal frequent itemsets in *T20I10N1KP5KC0.25D200K* and *T100I20N1KP5KC0.25D200K* are pretty small. This is consistent with the explanation of why FPmax* is slower than MAFIA in Figure 5.8.

AFOPT and GenMax [1] use the lowest amount of main memory in Figure 5.15 and Figure 5.16, but they are far slower than FPmax* as shown in Figure 5.7 and Figure 5.8.

[1]For some unknown reason, we can not run GenMax on dataset *T100I20N1KP5KC0.25D200K*

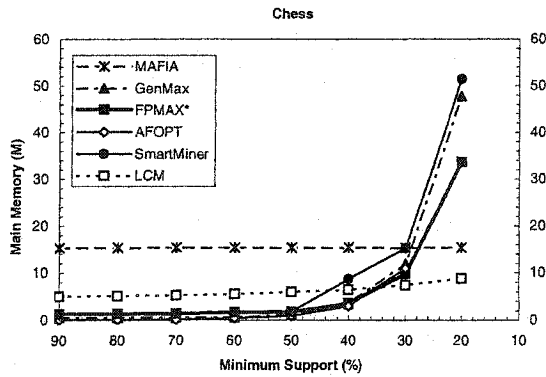

Figure 5.17: Memory Consumption of Mining Maximal FI's on *Chess*

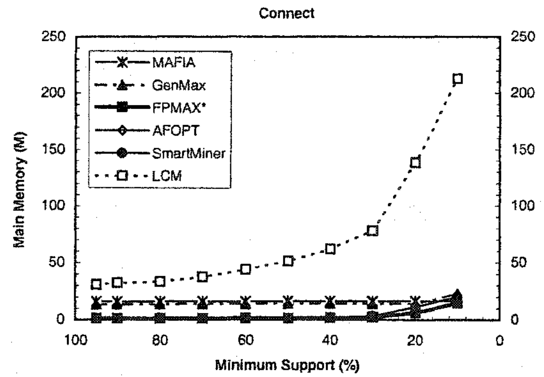

Figure 5.18: Memory Consumption of Mining Maximal FI's on *Connect*

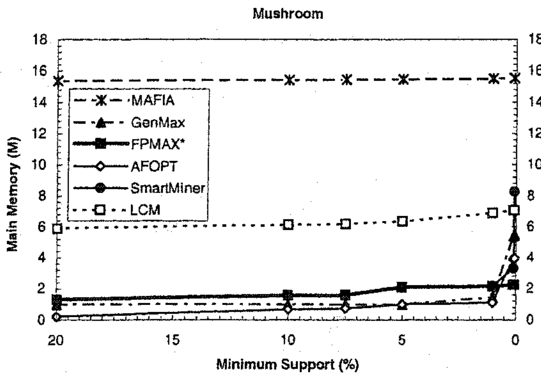

Figure 5.19: Memory Consumption of Mining Maximal FI's on *Mushroom*

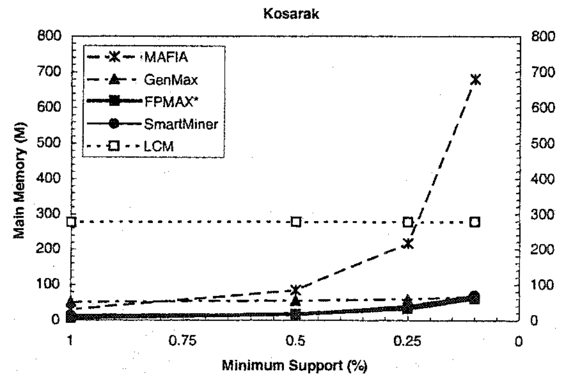

Figure 5.20: Memory Consumption of Mining Maximal FI's on *Kosarak*

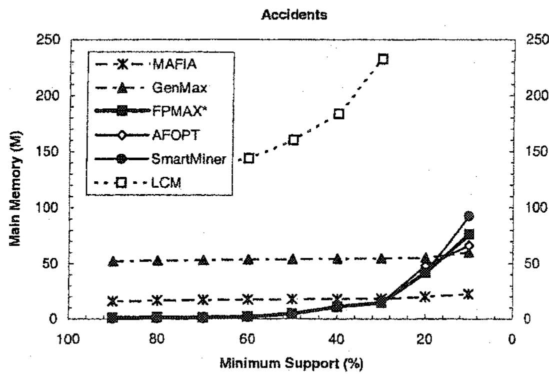

Figure 5.21: Memory Consumption of Mining Maximal FI's on *Accidents*

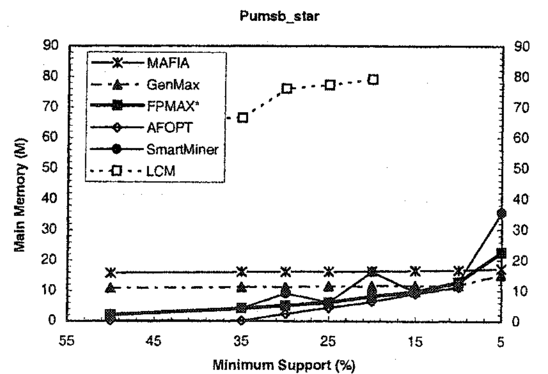

Figure 5.22: Memory Consumption of Mining Maximal FI's on *Pumsb**

Figures 5.17 to 5.22 show the main memory consumption of all algorithms running on all datasets. In all the figures, the lines for SmartMiner overlap the lines for FPmax* for high minimum support. However, when the minimum support is low, SmartMiner consumes much more main memory than FPmax*. From the figures, we can see that FPmax* uses the lowest amount of main memory in Figure 5.18, Figure 5.20, and in Figure 5.21 for high minimum support. For other cases, FPmax* uses slightly more main memory than AFOPT. However, for all cases, FPmax* is faster than AFOPT. We also notice that MAFIA always uses large amounts of main memory, with its best case is in Figure 5.16, when both the average transaction length and the average pattern length are long. Its runtime is also the shortest according to Figure 5.8. For most cases, GenMax consume more main memory than FPmax*.

Another fact shown in the figures is that the memory consumption of FPmax* and GenMax increases exponentially when the minimum support becomes very low. This can be observed, for example, in Figure 5.17. The increase happens because the algorithms have to keep a large number of maximal frequent itemsets in main memory, and the data structures such as MFI-tree become very large. We can also see that when the main memory needed by the algorithms increases rapidly, their runtime also increases very fast. The pair of Figure 5.11 and Figure 5.19 is a typical example.

Scalability

Figure 5.23 shows the speed scalability of all algorithms on synthetic datasets. All datasets were used in Chapter 3 for testing the scalability of all algorithms for mining all frequent itemsets. The number of transactions in the datasets ranges from 200,000 to 1,000,000. The minimum support was fixed as 0.1%.

Figure 5.23 shows that FPmax* is also a scalable algorithm. Runtime increases almost 5 times when the data size increases 5 times. The figures also demonstrate that other algorithms have good scalability. No algorithms have exponential runtime increasing when the dataset size increases.

Figure 5.24 shows that FPmax* possesses good scalability of memory consumption as well. Memory consumption changes from 76 megabytes to 332 megabytes when data size changes from 16 megabytes to 80 megabytes. All algorithms have similar scalability on synthetic datasets.

In conclusion, from the experimental results shown in all figures of runtime, main memory consumption, and scalability of all algorithms, though FPmax* is not the fastest algorithm for some

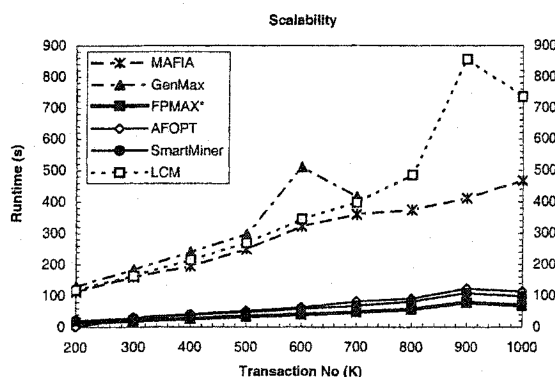

Figure 5.23: Scalability of runtime of Mining Maximal FI's

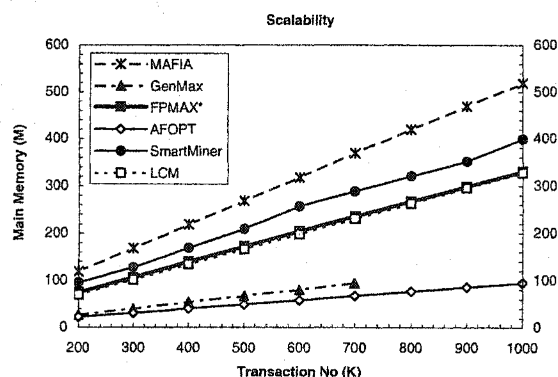

Figure 5.24: Scalability of Memory Consumption of Mining Maximal FI's

datasets and sometimes it consumes more main memory than other algorithms, overall it still has superior performance as compared with five other excellent algorithms. This result is consistent with the result of FIMI'03 and it proves that FPmax* is the best algorithm for mining maximal frequent itemsets.

Chapter 6

Discovering Closed Frequent Itemsets

Closed frequent itemset is the most important category of frequent itemset. The complete set of closed frequent itemsets in a database is smaller than the complete set of all frequent itemsets, and it keeps the support information of all frequent itemsets. In this chapter, we present the FPclose algorithm for mining frequent closed itemsets. It works similarly to FPmax*. Both mine frequent patterns from FP-trees. Whereas FPmax* needs to check that a newly found frequent itemset is maximal, FPclose needs to verify that the new frequent itemset is closed. For this we use a CFI-tree, which is another variation of a FP-tree.

6.1 The CFI-tree and algorithm FPclose

For the same reason as in FPmax*, a newly discovered frequent itemset can be a subset only of a previously discovered CFI. Similar to a MFI-tree, a CFI-tree is related to a FP-tree and an itemset X, and can be presented as C_X, X is called the *base* of the trees. The CFI-tree C_X always stores all already found CFI's containing itemset X, and their counts. A newly found frequent itemset Y that contains X only needs to be compared with the CFI's in C_X. If in C_X, there is no superset of Y with the same count as Y, Y is closed.

In a CFI-tree, each node in the subtree has four fields: item-name, count, node-link and level.

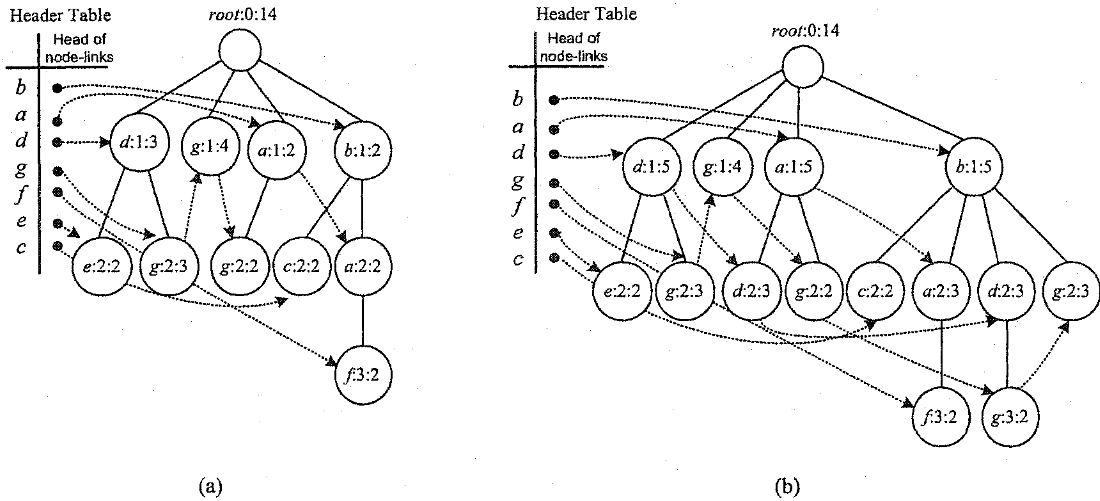

Figure 6.1: Construction of Closed Frequent Itemset Tree

Here, level is still used for subset testing, as in MFI-trees. The count field is needed because when comparing a Y with a set Z in the tree, we are trying to verify that it is not the case that $Y \subset Z$, and Y and Z have the same count. The order of the items in a CFI-tree's header table is the same as the order of items in header table of its corresponding FP-tree.

The insertion of a CFI into a CFI-tree is similar to the insertion of a transaction into a FP-tree, except now the count of a node is not incremented, it is always replaced by the maximal count up-to-date. Figure 6.1 shows some snapshots of the construction of a CFI-tree with respect to the FP-tree in Figure 2.4(b). The item orders in two trees are the same because they are both for base $\emptyset$. Note that insertions of CFI's into the top level CFI-tree will occur only after recursive calls have been made. In the following example, the insertions would in actuality be performed during various stages of the execution, not in bulk as the example might suggest. In Figure 6.1, a node $x : \ell : c$ means that the node is for item x, its level is ℓ and its count is c. In Figure 6.1(a), after inserting the first 6 CFI's into the CFI-tree, then we insert (d, g) with count 3. Since (d, g) shares the prefix d with (d, e), only node g is appended, and at the same time, the count for node d is changed from 2 to 3. The tree in part (b) of Figure 6.1 contains all CFI's for the dataset in Figure 2.4(a).

Figure 6.2 gives algorithm FPclose. Before calling FPclose with some (T, C), we already know from line 8 that there is no existing CFI X such that $T.base \subset X$, and $T.base$ and X have the same count (this corresponds to the optimization 4 in [53]). If there is only one single path in T, the nodes and their counts in this single path can be easily used to list the $T.base$-local closed frequent

Procedure *FPclose*(T, C)
Input: T, a FP-tree
C, the CFI-tree for $T.base$
Output: Updated C
Method:
1. **if** T only contains a single path P
2. generate all CFI's from P
3. **for each** CFI X generated
4. **if not** *closed_checking*(X, C)
5. insert X into C
6. **else for each** i in *T.header* **do begin**
7. set $Y = T.base \cup \{i\}$;
8. **if not** *closed_checking*(Y, C)
9. **if** *T.FP-array* is not NULL
10. *Tail* = {frequent items for i in *T.FP-array*}
11. **else**
12. *Tail*={frequent items in i's conditional pattern base}
13. sort *Tail* in decreasing order of items' counts
14. construct the FP-tree T_Y and its FP-array A_Y;
15. initialize Y's conditional CFI-tree C_Y;
16. call *FPclose*(T_Y, C_Y);
17. merge C_Y with C
18. **end**

Figure 6.2: Algorithm FPclose

itemsets. These itemsets will be compared with the CFI's in C. If an itemset is closed, it is inserted into C. If the FP-tree T is not a single-path tree, we execute line 6. Lines 9 to 12 use the FP-array technique. Lines 4 and 8 call function *closed_checking*(Y, C) to check if a frequent itemset Y is closed. If it is, the function returns true, otherwise, false is returned. Lines 14 and 15 construct Y's conditional FP-tree and CFI-tree. Then FPclose is called recursively for T_Y and C_Y.

Note that line 17 is not implemented as such. As in algorithm FPmax*, we found it more efficient to do the insertion of lines 3–5 into all CFI-trees currently in main memory.

To list all candidate closed frequent itemsets from a FP-tree with only a single path, suppose the path with counts is $(i_1 : c_1, i_2 : c_2, \ldots, i_p : c_p)$, where $i_j : c_j$ means item i_j has the count c_j. Starting from i_1, comparing the counts of every two adjacent items $i_j : c_j$ and $i_{j+1} : c_{j+1}$, in the path, if $c_j \neq c_{j+1}$, we list $i_1, i_2, \ldots, i_j$ as a candidate closed frequent itemset with count c_j.

CFI-trees are initialized similarly to MFI-trees, described in Section 5.2. The implementation of function *closed_checking* is almost the same as the implementation of function *maximality_checking*, except now we also consider the count of an itemset. Given an itemset $Y = \{i_1, i_2, \ldots, i_k\}$ with count c, suppose the order of the items in header table of the current CFI-tree is $i_1, i_2, \ldots, i_k$. Following the linked list of i_k, for each node in the list, first we check if its count is equal to or greater than

c. If it is, we then test if Y is a subset of the ancestors of that node. Here, the level of a node can also be used for saving comparison time, as in section 5.2. The function *closed_checking* returns true only when there is no existing CFI Z in the CFI-tree such that Z is a superset of Y and the count of Y is equal to or greater than the count of Z.

At the end of the execution of FPclose, the CFI-tree $C_\emptyset$ contains all CFI's mined from the original database. The proof of correctness of FPclose is straightforward.

In FPclose, we use an optimization which was not used in CLOSET+. Suppose a FP-tree for itemset Y has multiple paths, the count of Y is c, and the order of items in its header table is $i_1, i_2, \ldots, i_m$. Starting from i_m, for each item in the header table, we may need to do the work from line 7 to line 17. If for any item, say i_k, where $k \leq m$, $Y \cup \{i_1, i_2, \ldots, i_k\}$ is a closed itemset with count c, FPclose can terminate the current call. This is because for those items, *closed_checking* on line 8 would always return true.

Memory management allocating and deallocating space for FP-trees and CFI-trees is similar to the memory management of FPgrowth* and FPmax*.

By a similar analysis as in Section 5.4, we estimate the total main memory requirement for running FPclose on a dataset. If the tree that contains all CFI's needs to be kept in main memory, the algorithm needs space approximately sum of the size of the first FP-tree and its CFI-tree. Otherwise, if we care more about the main memory of FPclose, we trim FP-trees and CFI-trees, then the main memory requirement is approximately the size of the first FP-tree for $\emptyset$; if we care more about the speed of FPclose, we also take the compromise, i.e., instead of trimming FP-trees and CFI-trees, for each CFI, only part of it will be inserted into CFI-trees. By this way, less main memory will be used and the algorithm is faster than the one that keeps all complete CFI's. Figure 6.3 shows the size-reduced CFI-tree for $\emptyset$ corresponding to the dataset in Figure 2.4. In the CFI-tree, only 6 nodes are inserted, while in the complete CFI-tree in Figure 6.1 (b), 15 nodes are inserted.

6.2 Performance study

One of the first attempts to use FP-trees in CFI mining was the algorithm CLOSET+ [53]. CLOSET+ searches for the best strategies for mining frequent itemsets. Data structures and data traversal strategies are used depending on the characteristics of the dataset to be mined. The algorithm is

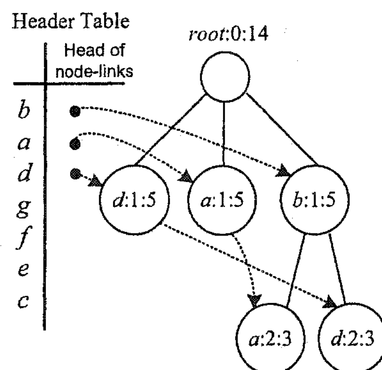

Figure 6.3: Size-reduced Closed Frequent Itemset Tree

viewed as the best algorithm for mining closed frequent itemsets.

Unfortunately, CLOSET+ was not implemented in FIMI'03 where mining closed frequent itemsets was one of the three categories of the contest in FIMI'03. Six other algorithms were compared in the independent experiments conducted by the organizers. In [15], the conclusion was drawn that "If one were to pick an overall best algorithm, it would arguably be FPclose, since it either performs the best or shows up in the runner-up spot, more times than any other algorithm". Thus CLOSET+ and FPclose, while both being recognized as the best, were not compared.

We got executable of CLOSET+ from [4]. We also selected 5 of the 6 algorithms compared in FIMI'03, Charm [59], LCM [52], AFOPT [35], Apriori [10, 6, 7] and FPclose [21]. All 6 algorithms were run on the same 8 datasets that were used in Chapter 3 and Chapter 5, i.e., two synthetic datasets *T20I10N1KP5KC0.25D200K* and *T100I20N1KP5KC0.25D200K*, and six real datasets: *accidents, kosarak, chess, connect, mushroom*, and *pumsb**.

The computer used for running the algorithms is a DELL Inspiron 8600 laptop with Pentium M, 1.6 GHz Processor, and 1GB of memory. Both time and memory consumption of each algorithm running on each dataset were recorded. Command "time" recorded runtime and command "memusage" recorded the memory consumption. In the figures, the runtime and memory consumption of an algorithm A running on dataset B for some minimum support δ is not recorded if A needs too much time (longer than 30 minutes) to run on B for minimum support δ, or the implementation of the A has some bugs.

In the present experiments, FPclose is implemented by keeping only part of a CFI in CFI-trees, as explained in Section 5.4.

The runtime

Figure 6.4 and Figure 6.5 show the runtime of all algorithms on synthetic datasets. In Figure 6.4, Charm, CLOSET+, and AFOPT have almost the same speed. Their lines overlap. For the two synthetic datasets, FPclose is only slower than Apriori for high minimum support. When the minimum support is low, FPclose becomes the fastest algorithm and Apriori becomes the slowest algorithm.

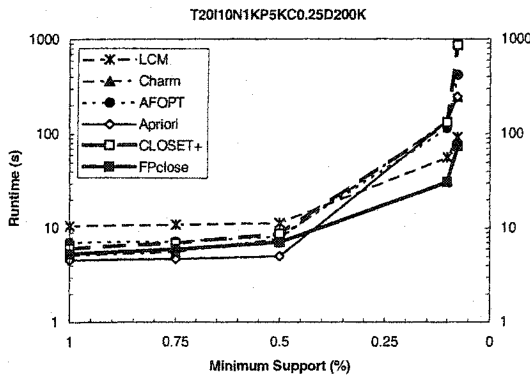

Figure 6.4: Runtime of Mining Closed FI's on *T20I10N1KP5KC0.25D200K*

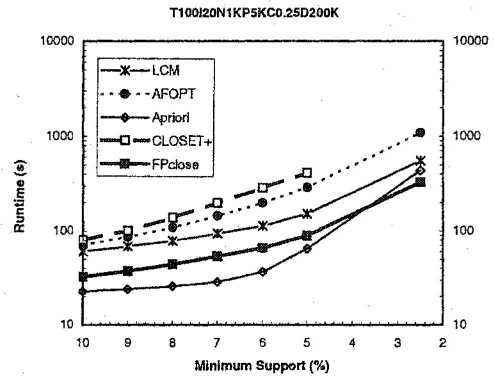

Figure 6.5: Runtime of Mining Closed FI's on *T100I20N1KP5KC0.25D200K*

FPclose is slower than Apriori for high minimum support because FPclose spends much time to construct a bushy and wide FP-tree while from the FP-tree only small number of closed frequent itemsets are generated. On the contrary, Apriori only needs small size data structures to keep and generate candidate frequent itemsets and closed frequent itemsets. When the minimum support is low, FPclose is fast because its work on the first stage offers a big gain, many closed frequent itemsets will be generated. While for Apriori, a lot of work has to be done to deal with large number of candidate frequent itemsets. Figure 6.12 shows that the main memory needs by Apriori changes drastically when minimum support becomes lower.

For real datasets, as shown in Figure 6.6 to Figure 6.11, FPclose has a very good performance. It is the fastest algorithm in Figure 6.9, Figure 6.10 and Figure 6.11.

In Figure 6.6, FPclose is faster than LCM for high minimum support, but it is slower than LCM when the minimum support is low. CLOSET+ is the fastest algorithm when the minimum support is very high, but the runtime increases rapidly when the minimum support becomes low.

In Figure 6.7, CLOSET+ is the fastest algorithm. However, we also can see that its runtime still increases very fast and we can expect that it will be slower than FPclose when the minimum

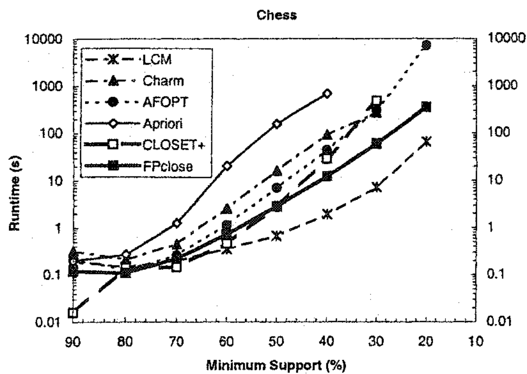

Figure 6.6: Runtime of Mining Closed FI's on *Chess*

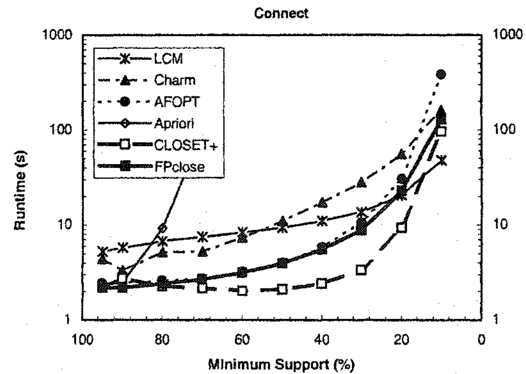

Figure 6.7: Runtime of Mining Closed FI's on *Connect*

support becomes lower.

In Figure 6.8, CLOSET+ is still the fastest. Both Figure 6.7 and Figure 6.8 show that the strategies introduced in [53] work well for datasets *mushroom* and *connect*.

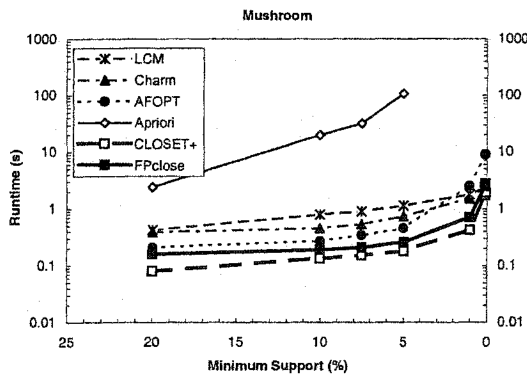

Figure 6.8: Runtime of Mining Closed FI's on *Mushroom*

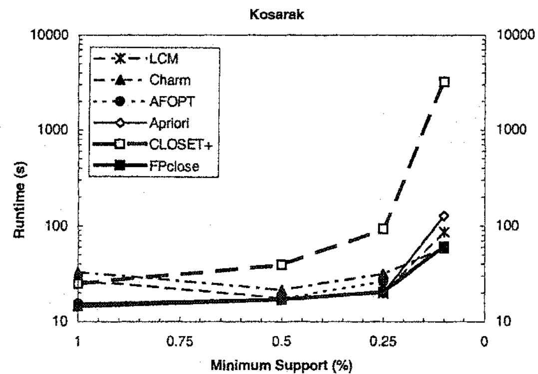

Figure 6.9: Runtime of Mining Closed FI's on *Kosarak*

In Figure 6.9, however, CLOSET+ demonstrates the worst performance. Its runtime shows an exponential increase when the minimum support becomes low. In the same test, FPclose is the fastest algorithm. It is an order of magnitude faster than CLOSET+ for the lowest minimum support in Figure 6.9. The lines for AFOPT and Apriori overlap the line for FPclose for high minimum support, but the two algorithms are all slower than FPclose when the minimum support is low.

FPclose and CLOSET+ show similar speed when running on *accidents*, as indicated in Figure 6.10. However, in Figure 6.11, CLOSET+ is almost two order of magnitude slower than FPclose for minimum support 5%. The other algorithms show similar performance on dataset *accidents* and *pumsb**.

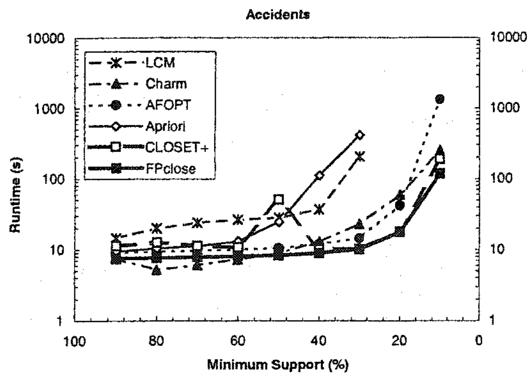

Figure 6.10: Runtime of Mining Closed FI's on *Accidents*

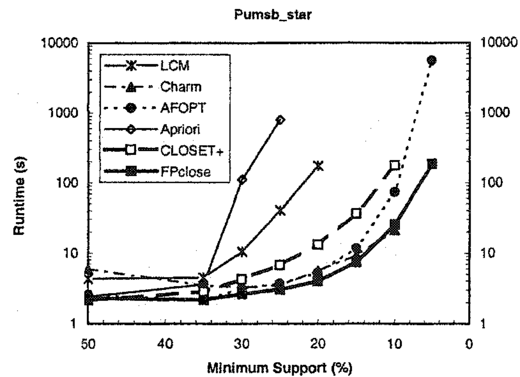

Figure 6.11: Runtime of Mining Closed FI's on *Pumsb**

One of the reasons why FPclose is usually faster than CLOSET+ for low minimum support is the fact that CLOSET+ uses a global tree to store already found closed frequent itemsets. When there are large amounts of frequent itemsets, each candidate closed frequent itemset has to be compared with many existing itemsets. In FPclose, on the contrary, multiple CFI-trees are constructed. Consequently, a candidate closed frequent itemset only needs to be compared with small set of itemsets, saving lots of time.

Memory consumption

Figure 6.12 and Figure 6.13 show the peak main memory consumption of the algorithms when running them on synthetic datasets. In Chapter 3 and Chapter 5, the main memory consumption of FPgrowth* and FPmax* for synthetic data is high. Here, FPclose consumes a lot of main memory, too. The reason is similar to the reason explained in both Section 3.4.2 and Section 5.5.

CLOSET+ uses the maximum amount of main memory in both figures. This is not surprising. Besides the main memory consumed for the bushy and wide FP-trees for the synthetic datasets, it has to keep a big tree for closedness testing as well. At the same time in FPclose, only part of each closed frequent itemset was kept in a CFI-tree, as explained in Section 5.4.

When running all algorithms on real dataset *chess*, FPclose consumes the least amount of main memory for high support, as shown in Figure 6.14. However, when the minimum support becomes low, the main memory consumption increases fast. By comparing Figure 3.15 and Figure 6.14, we know that most space was spent on keeping the CFI-trees.

In both Figure 6.15 and Figure 6.16, FPclose uses the least amount of memory. Both figures show

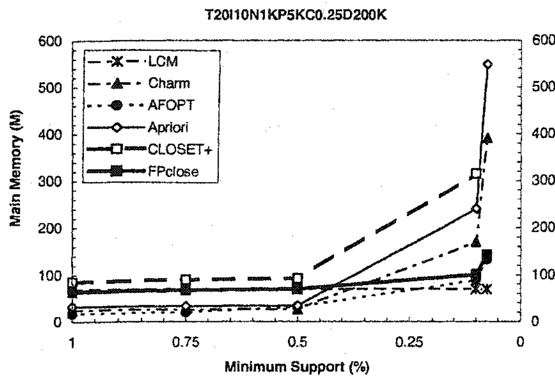

Figure 6.12: Memory Consumption of Mining Closed FI's on *T20I10N1KP5KC0.25D200K*

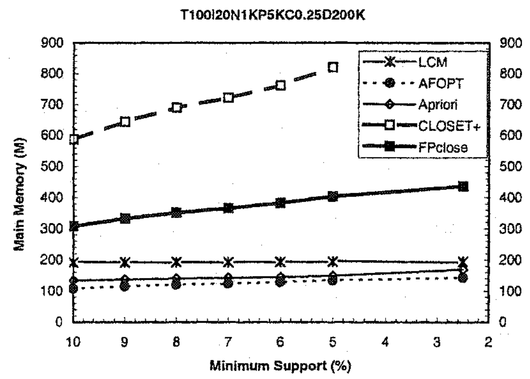

Figure 6.13: Memory Consumption of Mining Closed FI's on *T100I20N1KP5KC0.25D200K*

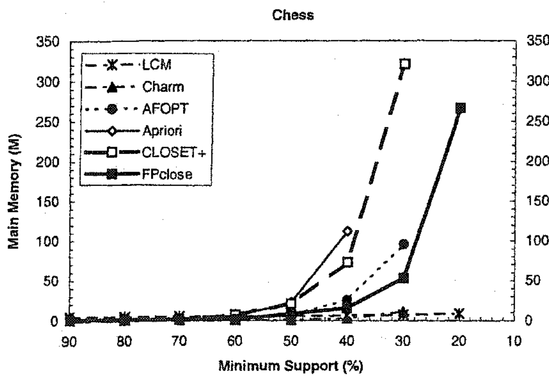

Figure 6.14: Memory Consumption of Mining Closed FI's on *Chess*

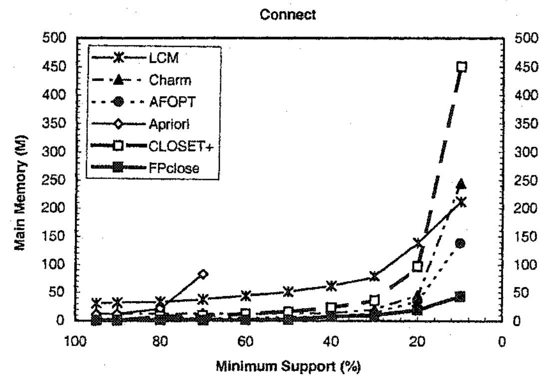

Figure 6.15: Memory Consumption of Mining Closed FI's on *Connect*

that FP-trees and their CFI-trees have very good compactness for dataset *connect* and *mushroom*.

In Figure 6.17 and Figure 6.18, the main memory consumption of FPclose increases slowly compared with the increase of other algorithms. Figure 6.19 shows that FPclose consumes the least amount of main memory and the dataset produces a large number of closed frequent itemsets when the minimum support is low.

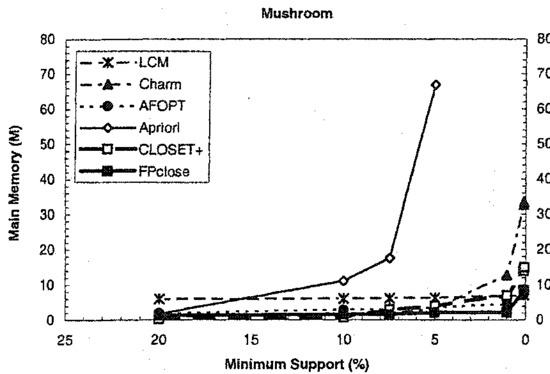

Figure 6.16: Memory Consumption of Mining Closed FI's on *Mushroom*

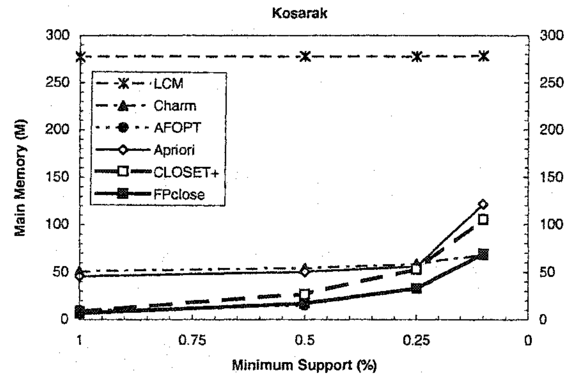

Figure 6.17: Memory Consumption of Mining Closed FI's on *Kosarak*

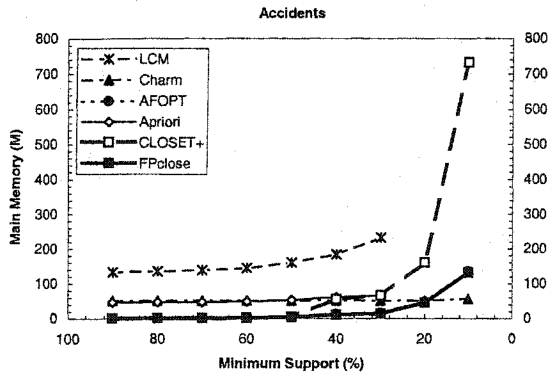

Figure 6.18: Memory Consumption of Mining Closed FI's on *Accidents*

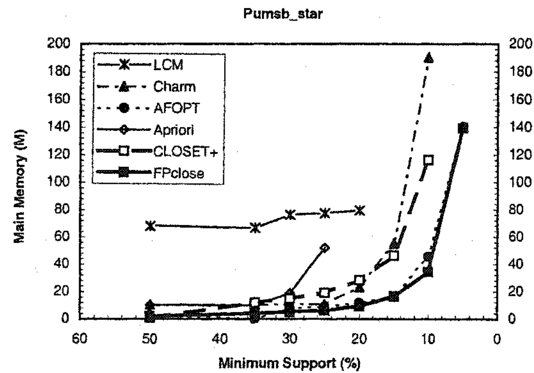

Figure 6.19: Memory Consumption of Mining Closed FI's on *Pumsb**

From figures 6.12 to 6.19, we see that FPclose almost always uses the lowest main memory. We also see that the main memory consumption of CLOSET+ is always very high. In Figure 6.7, CLOSET+ is faster than FPclose, while in Figure 6.15, for the minimum support 10%, CLOSET+ consumes main memory an order of magnitude higher than FPclose.

Scalability

Figure 6.20 and Figure 6.21 show the scalability of all algorithms when running them on synthetic datasets. These datasets were used for testing the scalability of the algorithms for mining all and maximal frequent itemsets in Chapter 3 and Chapter 5. The number of transactions in the datasets ranges from 200,000 to 1,000,000. The minimum support was fixed as 0.1%.

All the algorithms are scalable algorithm for runtime, according to Figure 6.20. The difference is their increasing ratio. In the figure, we can see that FPclose has the smallest ratio.

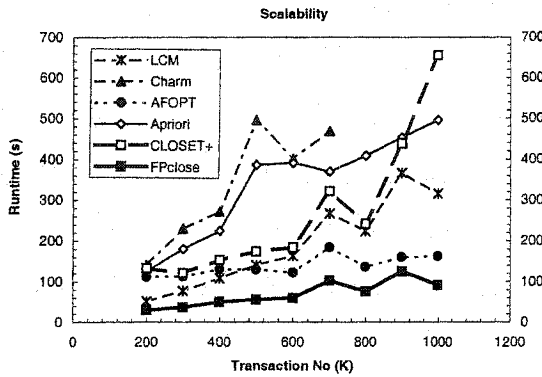

Figure 6.20: Scalability of runtime of Mining Closed FI's

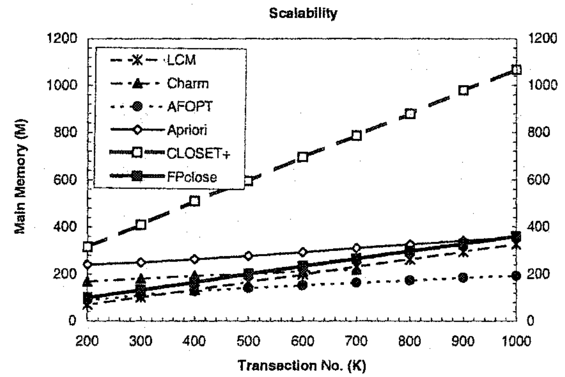

Figure 6.21: Scalability of Memory Consumption of Mining Closed FI's

In Figure 6.21, CLOSET+ shows the worst scalability of its main memory consumption, while FPclose has a reasonable scalability compared with other algorithms. Main memory size increases 4 times when the size of synthetic datasets increases 5 times. AFOPT shows the best scalability.

All the experimental results in this section allows us to conclude that overall FPclose is the best algorithm for mining closed frequent itemsets. This conclusion is consistent with the result of FIMI'03 [14].

Chapter 7

Mining Frequent Itemsets from Secondary Memory

In Table 3.1, we showed size of all datasets used in the experiments of Chapter 3 to 6. By comparing the dataset size in the table with the size of memory consumption in figures from 3.13 to 3.20, we can see that the memory consumption of any algorithm running on any dataset could be greater than the size of the dataset, especially when the minimum support is low. The situation is even worse when we mine maximal or closed frequent itemsets.

Furthermore, most algorithms, such as Apriori [6, 7] and dEclat [60] work well when the main memory is big enough to fit the whole database or/and the data structures (hash trees, FP-trees, etc). When a database is very large or when the minimum support is very low, those algorithms do not perform very well because they run "out of memory".

What is the best way to mine frequent itemsets from very large databases residing in a secondary memory storage, such as disks? Here "very large" means that the data structures constructed from the database for mining frequent itemsets can not fit in the available main memory. In this chapter we address this problem. We limit the problem to mine *all* frequent itemsets. We first analyze the approaches for mining frequent itemsets from disks, and introduce three algorithms. Then we present detailed divide-and-conquer algorithm *Diskmine*, in which many novel optimization techniques are used.

7.1 Strategies for mining frequent itemsets from disk

Basically, there are two strategies for mining frequent itemsets, the data-structures approach, and the divide-and-conquer approach.

The *data-structures* approach consists of reading the database buffer by buffer, and generate data-structures (i.e. candidate sets or FP-trees). Since the data-structures do not fit into main memory, additional disk I/O's are required. The number of passes and disk I/O's required by the approach depend on the algorithm and its data-structures. For example, if the algorithm is Apriori [7] using a hash-tree for candidate itemsets, disk based hash-trees have to be used. Then the number of passes for the algorithm is the same as the length of the longest frequent itemset, and the number of disk I/O's for the hash-trees depends on the size of the hash-trees on disk.

The basic strategy for the *divide-and-conquer* approach is shown in Figure 7.1. In the approach, $|\mathcal{D}|$ denotes the size of the data structures used by the mining algorithm, and M is the size of available main memory. Function *mainmine* is called if candidate frequent itemsets (not necessary all) can be mined without writing the data structures used by a mining algorithm to disks. In Figure 7.1, a very large database is decomposed into a number of smaller databases. If a "small" database is still too large, i.e, the data structures are still too big to fit in main memory, the decomposition is recursively continued until the data structures fit in main memory. After all small databases are processed, all candidate frequent itemsets are combined in some way (obviously depending on the way the decomposition was done) to get all frequent itemsets for the original database.

Procedure *diskmine*($\mathcal{D}, M$)
if $|\mathcal{D}| \leq M$ **then**
    return *mainmine*($\mathcal{D}$)
else
    decompose $\mathcal{D}$ into $\mathcal{D}_1, \ldots \mathcal{D}_k$.
    return *combine diskmine*($\mathcal{D}_1, M$),
                                        .... ,
                                diskmine($\mathcal{D}_k, M$).

Figure 7.1: General divide-and-conquer algorithm for mining frequent itemsets from disk.

The efficiency of *diskmine* depends on the method used for mining frequent itemsets in main memory and on the number of disk I/O's needed in the decomposition and combination phases. Sometimes the disk I/O is the main factor. Since the decomposition step involves I/O, ideally the number of recursive calls should be kept small. The faster we can obtain small decomposed

databases, the fewer recursive call we will need. On the other hand, if a decomposition cuts down the size of the projected databases drastically, the trade-off might be that the combination step becomes more complicated and might involve heavy disk I/O.

In the following we discuss two decomposition strategies, namely decomposition by partition, and decomposition by projection.

Partitioning is an approach in which a large database is decomposed into cells of small non-overlapping databases. The cell-size is chosen so that all frequent itemsets in a cell can be mined without having to store any data structures in secondary memory. However, since a cell only contains partial frequency information of the original database, all frequent itemsets from the cell are local to that cell of the partition, and could only be *candidate* frequent itemsets for the whole database. Thus the candidate frequent itemsets mined from a cell have to be verified later to filter out false hits. Consequently, those candidate sets have to be written to disk in order to leave space for processing the next cell of the partition. After generating candidate frequent itemsets from all cells, another database scan is needed to filter out all infrequent itemsets. The partition approach therefore needs only two passes over the database, but writing and reading candidate frequent itemsets will involve a significant number of disk I/O's, depending on the size of the set of candidate frequent itemsets.

We can conclude that the partition approach to decomposition keeps the recursive levels down to one, but the penalty is that the combination phase becomes expensive.

To get an easier combination phase, we adopt another decomposition strategy, which we call *projection*. This approach projects the original database on several databases, each determined by one or more frequent item(s). One advantage of this approach is that any frequent itemset mined from a projected database is a frequent itemset in the original database. To get *all* frequent itemsets, we only need to take the union of the frequent itemsets discovered in the small projected databases. The biggest problem of the projection approach is that the total size of the projected databases could be too large, and there could be too many disk I/O's for the projected databases. Thus, there is a trade-off between the easier combination phase and possible too many disk I/O's.

To analyze the recurrence and required disk I/O's of the general divide-and-conquer algorithm when the decomposition strategy is projection, let us suppose that:

- The original database size is D bytes.

- The data structure is a FP-tree.

- The FP-tree constructed from original database $\mathcal{D}$ is T, and its size is $|T|$ bytes.

- If a conditional FP-tree T' is constructed from a FP-tree T, then $|T'| \leq c \cdot |T|$, for some constant $c < 1$.

- The main memory mining method is the FP-growth method [28, 29]. Two database scans are needed for constructing a FP-tree from a database.

- The block size is B bytes.

- The main memory available for the FP-tree is M bytes.

In the first line of the algorithm in Figure 7.1, if T can not fit in memory, then projected databases will be generated. We assumed that the size of the FP-tree for a projected database is $c \cdot |T|$. If $c \cdot |T| \leq M$, function *mainmine* can be called for the projected database, otherwise, the decomposition goes on. At pass m, the size of the FP-tree constructed from a projected database is $c^m \cdot |T|$. Thus, the number of passes needed by the divide-and-conquer projection algorithm is $1 + \lceil \log_c M/T \rceil$. Based on our experience and the analysis in [28, 29], we can say that for all practical purposes the number of passes will be at most two. For example, Let $D = 100$ Gigabytes, $T = 10$ Gigabytes, $M = 1$ Gigabyte, and $c = 10\%$. Then the number of passes is $1 + \lceil \log_{0.1} 2^{30}/(10 \times 2^{30}) \rceil = 2$. In five passes we can handle databases up to 100 Terabytes. Namely, we get $1 + \lceil \log_{0.1} 2^{30}/(10 \times 2^{40}) \rceil = 5$.

Assume that there are two passes, and that the sum of the sizes of all projected databases is D'. After the first database scan for finding all frequent single items, the second database scan attempts to construct a FP-tree from the database. If the main memory is not big enough, the scan will be aborted. We assume on average half of $\mathcal{D}$ is read at this stage, which means $1/2 \cdot D/B$ disk I/O's. The third scan is for decomposition. Totally, there are $5/2 \times D/B$ disk I/O's. The projected databases have to be written to the disks first, then later each scanned twice for building the FP-tree. This step needs $3 \times D'/B$ disk I/O's. Thus, the total disk number of disk I/O's for the general divide-and-conquer projection algorithm is at least

$$5/2 \cdot D/B + 3 \cdot D'/B. \qquad (2)$$

Obviously, the smaller D', the better the performance.

One of the simplest projection strategies is to project the database on each frequent item, which we call *basic projection.* First we need some formal definitions.

Definition 7.1 *Let I be a set of items. By I^* we will denote* strings *over I, such that each symbol occurs at most once in the string. If α, β are strings, then $\alpha.\beta$ denotes the concatenation of the string α with the string β.*

Let $\mathcal{D}$ be an I-database. Then $freqstring(\mathcal{D})$ is the string over I, such that each frequent item in $\mathcal{D}$ occurs in it exactly once, and the items are in decreasing order of frequency in $\mathcal{D}$. ■

As an example, consider the $\{a, b, c, d, e\}$-database $\mathcal{D} = \{\{a, c, d\}, \{b, c, d, e\}, \{a, b\}, \{a, c\}\}$. If the minimum support $\delta = 50\%$, then $freqstring(\mathcal{D}) = acbd$.

Definition 7.2 *Let $\mathcal{D}$ be an I-database, and let $freqstring(\mathcal{D}) = i_1 i_2 \cdots i_k$. For $j \in \{1, \ldots, k\}$ we define $\mathcal{D}_{i_j} = \{\tau \cap \{i_1, \ldots, i_j\} : i_j \in \tau, \tau \in \mathcal{D}\}$.*

Let $\alpha \in I^*$. We define $\mathcal{D}_\alpha$ inductively: $\mathcal{D}_\epsilon = \mathcal{D}$, and let $freqstring(\mathcal{D}_\alpha) = i_1 i_2 \cdots i_k$. Then, for $j \in \{1, \ldots, k\}$, $\mathcal{D}_{\alpha.i_j} = \{\tau \cap \{i_1, \ldots, i_j\} : i_j \in \tau, \tau \in \mathcal{D}_\alpha\}$. ■

Obviously, $\mathcal{D}_{\alpha.i_j}$ is an $\{i_1, \ldots, i_j\}$-database. The decomposition of $\mathcal{D}_\alpha$ into $\mathcal{D}_{\alpha.i_1}$, $\ldots$, $\mathcal{D}_{\alpha.i_k}$ is called the *basic projection*.

To illustrate the basic projection, let's consider the above example, starting from the least frequent item in the *freqstring*, we obtain $\mathcal{D}_d = \{\{a, c, d\}, \{b, c, d\}\}$, $\mathcal{D}_b = \{\{c, b\}, \{a, b\}\}$, $\mathcal{D}_c = \{\{a, c\}, \{c\}, \{a, c\}\}$, and $\mathcal{D}_a = \{\{a\}, \{a\}, \{a\}\}$.

Definition 7.3 *Let $\alpha \in I^*$, $i_j \in I$, and let $\mathcal{D}_{\alpha.i_j}$ be an I-database. Then $freqsets(\delta, \mathcal{D}_{\alpha.i_j})$ denotes the subsets of I that contain i_j and are frequent in $\mathcal{D}_{\alpha.i_j}$ when the minimum support is δ. Usually, we shall abstract δ away, and write just $freqsets(\mathcal{D}_{\alpha.i_j})$.* ■

Lemma 7.1 *Let $\mathcal{D}_\alpha$ be an I-database, and $freqstring(\mathcal{D}_\alpha) = i_1 i_2 \cdots i_k$. Then*

$$freqsets(\mathcal{D}_\alpha) = \bigcup_{j \in \{1, \ldots, k\}} freqsets(\mathcal{D}_{\alpha.i_j})$$

Proof. ($\subseteq$-*direction*). Let $S \in freqsets(\mathcal{D}_\alpha)$, and suppose i_n is the item in S that is least frequent in $\mathcal{D}_\alpha$. Since $\mathcal{D}_{\alpha.i_n}$ is an $\{i_1, \ldots, i_n\}$-database, and transactions in $\mathcal{D}_\alpha$ that contain item i_j are all in $\mathcal{D}_{\alpha.i_j}$, if S is frequent in $\mathcal{D}_\alpha$, then S must be frequent in $\mathcal{D}_{\alpha.i_j}$.
($\supseteq$-*direction*). For any frequent itemset $S \in freqsets(\mathcal{D}_{\alpha.i_j})$, according to the definition, the support of any itemset in $\mathcal{D}_{\alpha.i_j}$ is not greater than the support of it in $\mathcal{D}_\alpha$. Therefore, S must be frequent in $\mathcal{D}_\alpha$. ■

In the previous example, for $\mathcal{D}_d$, *freqsets*($\mathcal{D}_d$)=$\{\{d\},\{c,d\}\}$. Note though $\{c\}$ is also frequent in $\mathcal{D}_d$, it is not listed since it does not contain d. It will be listed in *freqsets*($\mathcal{D}_c$). Similarly, *freqsets*($\mathcal{D}_b$)=$\{\{b\}\}$, *freqsets*($\mathcal{D}_c$)=$\{\{c\},\{a,c\}\}$ and *freqsets*($\mathcal{D}_a$)=$\{\{a\}\}$. We also can notice that $\mathcal{D}_d$ and $\mathcal{D}_c$ are not that much smaller than the original database. The upside is though that the set of all frequent itemsets in $\mathcal{D}$ now simply is the union of *freqsets*($\mathcal{D}_d$), *freqsets*($\mathcal{D}_b$), *freqsets*($\mathcal{D}_c$) and *freqsets*($\mathcal{D}_d$). This means that the combination phase is a simple union.

Figure 7.2 gives a divide-and-conquer algorithm that uses basic projection. A transaction τ in $\mathcal{D}_\alpha$ will be partly inserted into $\mathcal{D}_{\alpha.i_j}$ if and only if τ contains i_j. The parallel projection algorithm introduced in [29] is an algorithm of this kind.

Procedure *basicdiskmine*($\mathcal{D}_\alpha, M$)
if $|\mathcal{D}_\alpha| \leq M$ **then**
  return *mainmine(*$\mathcal{D}_\alpha$*)*
else
  let $freqstring(\mathcal{D}_\alpha) = i_1 i_2 \cdots i_n$,
  return *basicdiskmine*($\mathcal{D}_{\alpha.i_1}, M$) $\cup$
    $\ldots \cup$
    basicdiskmine($\mathcal{D}_{\alpha.i_n}, M$).

Figure 7.2: A simple divide-and-conquer algorithm for mining frequent itemsets from disk

Let's analyze the disk I/O's of the algorithm in Figure 7.2. As before, we assume that there are two passes, that the data structure is a FP-tree, and that the main memory mining method is FP-growth. If in $\mathcal{D}_\epsilon$, each transaction contains on the average n frequent items, each transaction will be written to n projected databases. Thus the total length of the associated transactions in the projected databases is $n + (n-1) + \cdots + 1 = n(n+1)/2$, the total size of all projected databases is $(n+1)/2 \cdot D \approx n/2 \cdot D$.

Still there are two full database scans and a incomplete database scan for $\mathcal{D}_\epsilon$, as explained for formula (1). The number of total disk I/O's is $5/2 \cdot D/B$. The projected databases have to be written to the disks first, then later scanned twice each for building a FP-tree. This step needs at least $3 \cdot n/2 \times D/B$. Thus, the total disk I/O's for the divide-and-conquer algorithm with basic projection is

$$5/2 \cdot D/B + n \cdot 3/2 \cdot D/B \quad (3)$$

The recurrence structure of algorithm *basicdiskmine* is shown in Figure 7.3. The reader should ignore nodes in the shaded area at this point, they represent processing in main memory.

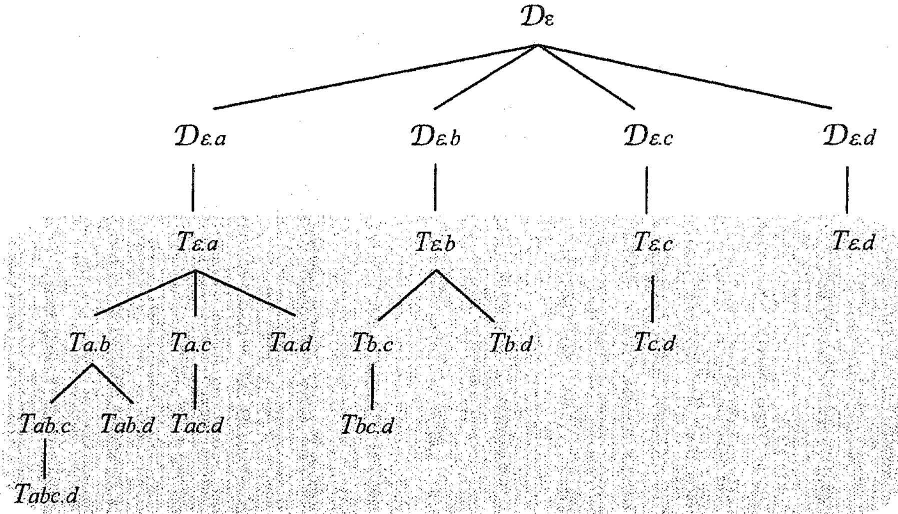

Figure 7.3: Recurrence structure of Basic Projection

In a typical application n, the average number of frequent items could be hundreds, or thousands. It therefore makes sense to devise a smarter projection strategy. Before we go further, we introduce some definitions and a lemma.

Definition 7.4 *Let $\mathcal{D}_\alpha$ be an I-database, and let $freqstring(\mathcal{D}_\alpha) = \beta_1.\beta_2.\cdots.\beta_k$, where each β_j is a string in I^*. We call $\beta_1.\beta_2.\cdots.\beta_k$ a* grouping *of $freqstring(\mathcal{D}_\alpha)$. Let $\beta_j = i_{j_1}.\dots.i_{j_m}$, for $j \in \{1,\dots,k\}$. We now define $\mathcal{D}_{\alpha.\beta_j} =$*

$$\{\tau \cap \{i_{1_1},\dots,i_{j_m}\}, \tau \in \mathcal{D}_\alpha, \tau \cap \{i_{j_1},\dots,i_{j_m}\} \neq \emptyset\}.$$

In $\mathcal{D}_{\alpha.\beta_j}$, items in β_j are called master items, *items in $\beta_1,\dots,\beta_{j-1}$ are called* slave items. ∎

For the previous example, $freqstring(\mathcal{D}_\alpha) = acbd$, $\beta_1 = ac$, $\beta_2 = bd$ gives the grouping $ac.bd$ of $acbd$. Now $\mathcal{D}_{bd} = \{\{a,c,d\},\{b,c,d\},\{a,b\}\}$ and $\mathcal{D}_{ac} = \{\{a,c\},\{c\},\{a\},\{a,c\}\}$.

Definition 7.5 *Let $\{\alpha,\beta\} \subset I^*$, and let $\mathcal{D}_{\alpha.\beta}$ be an I-database. Then $freqsets(\mathcal{D}_{\alpha.\beta})$ denotes the subsets of I that contain at least one item in β and are frequent in $\mathcal{D}_{\alpha.\beta}$.* ∎

Lemma 7.2 *Let $\alpha \in I^*$, $\mathcal{D}_\alpha$ be an I-database, and $freqstring(\mathcal{D}_\alpha) = \beta_1\beta_2\cdots\beta_k$. Then*

$$freqsets(\mathcal{D}_\alpha) = \bigcup_{j\in\{1,\dots,k\}} freqsets(\mathcal{D}_{\alpha.\beta_j})$$

Proof. Straightforward from Lemma 7.1 and the definition of $\mathcal{D}_{\alpha.\beta}$. ∎

By following the above example, we can get *freqsets*($\mathcal{D}_{bd}$)=$\{\{d\},\{b\},\{c,d\}\}$, and *freqsets*($\mathcal{D}_{ac}$)= $\{\{c\},\{a\},\{a,c\}\}$.

Based on Lemma 7.2, we can obtain a more aggressive divide-and-conquer algorithm for mining from disks. Figure 7.4 shows the algorithm *aggressivediskmine.* Here, $freqstring(\mathcal{D}_\alpha)$ is decomposed into several substrings β_j, each of which could have more than one item. Each substring corresponds to a projected database. A transaction τ in $\mathcal{D}_\alpha$ will be partly inserted into $\mathcal{D}_{\alpha.\beta_j}$ if and only if τ contains at least one item a such that $a \in \beta_j$. Since there will be fewer projected databases, there will be fewer disk I/O's. Compared with the algorithm in Figure 7.2, we can expect that a large amount of disk I/O will be saved by the algorithm in Figure 7.4.

Procedure *aggressivediskmine*($\mathcal{D}_\alpha, M$)
if $|\mathcal{D}_\alpha| \leq M$ **then**
  return *mainmine*($\mathcal{D}_\alpha$)
else
  let $freqstring(\mathcal{D}_\alpha) = \beta_1\beta_2\cdots\beta_k$,
  return *aggressivediskmine*($\mathcal{D}_{\alpha.\beta_1}, M$) $\cup$
      $\ldots \cup$
    aggressivediskmine($\mathcal{D}_{\alpha.\beta_k}, M$).

Figure 7.4: A more aggressive divide-and-conquer algorithm for mining frequent itemsets from disk

Let's analyze the recurrence and disk I/O's of the aggressive divide-and-conquer algorithm. The number of passes needed by the algorithm is still $1 + \lceil \log_c M/T \rceil \approx 2$, since grouping items does not change the size of a FP-tree for a projected database. However, for disk I/O, suppose in $\mathcal{D}_\epsilon$, each transaction contains on average n frequent items, and that we can group them into k groups of equal size. Then the n items will be written to the projected databases with total length $n/k+2\cdot n/k+\ldots+k\cdot n/k = (k+1)/2\cdot n$. Total size of all projected databases is $(k+1)/2\cdot D \approx k/2\cdot D$. The total disk I/O's for the aggressive divide-and-conquer algorithm is then

$$5/2 \cdot D/B + k \cdot 3/2 \cdot D/B \quad (4)$$

The recurrence structure of algorithm *aggressivediskmine* is shown in Figure 7.5. Compared to Figure 7.3, we can see that the part of the tree that corresponds to decomposition (the non-shaded part) is much smaller in Figure 7.5. Although the example is very small, it exhibits the general structure of the two trees.

If $k \ll n$, we can expect that the aggressive divide and conquer algorithm will significantly

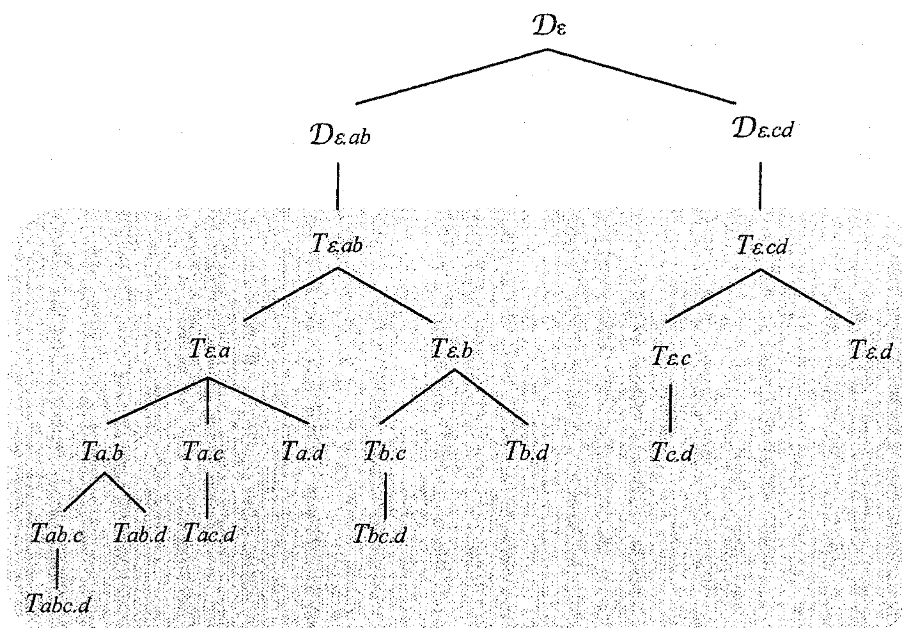

Figure 7.5: Recurrence structure of Aggressive Projection

outperform the basic one.

7.2 Algorithm Diskmine

In this section we give the details of our divide-and-conquer algorithm for mining frequent itemsets from secondary memory. We call the algorithm *Diskmine.* In the algorithm, the FP-tree is used as data structure and the extension of the FP-growth method, FPgrowth* [21], as method for mining frequent itemsets from a FP-tree.

7.2.1 Divide-and-conquer by aggressive projection

The algorithm *Diskmine* is shown in Figure 7.6. In the algorithm, $\mathcal{D}_\alpha$ is the original database or a projected database, and M is the maximal size of main memory that can be used by *Diskmine.*

Diskmine uses the FP-tree as data structure and FPgrowth* [21] as main memory mining algorithm. Since the FP-tree encodes all frequency information of the database, we can shift into main memory mining as soon as the FP-tree fits into main memory.

Since a FP-tree usually is a significant compression of the database, our *Diskmine* algorithm begins optimistically, by calling *trialmainmine*, which starts scanning the database and constructing

Procedure *Diskmine*($\mathcal{D}_\alpha, M$)
scan $\mathcal{D}_\alpha$ and compute *freqstring*($\mathcal{D}_\alpha$)
call *trialmainmine*($\mathcal{D}_\alpha, M$)
if *trialmainmine*($\mathcal{D}_\alpha, M$) aborted **then**
  compute a grouping $\beta_1\beta_2\cdots\beta_k$ of $freqstring(\mathcal{D}_\alpha)$
  Decompose $\mathcal{D}_\alpha$ into $\mathcal{D}_{\alpha.\beta_1}, \ldots, \mathcal{D}_{\alpha.\beta_k}$
  for j = 1 **to** k **do begin**
    if β_j is a singleton **then**
      $Diskmine(\mathcal{D}_{\alpha.\beta_j}, M)$
    else
      $mainmine(\mathcal{D}_{\alpha.\beta_j})$
  end
else return *freqsets*($\mathcal{D}_\alpha$)

Figure 7.6: Algorithm Diskmine

the FP-tree. If the tree can be successfully completed and stored in main memory, we have reached the bottom level of the recursion, and can obtain the frequent itemsets of the database by running FPgrowth* on the FP-tree in main memory.

Procedure *trialmainmine*($\mathcal{D}_\alpha, M$)
start scanning $\mathcal{D}_\alpha$ and building the FP-tree T_α in main memory.
if $|T_\alpha|$ exceeds M **then**
  return the incomplete T_α
else
  call *FPgrowth**(T_α) and **return** *freqsets*($\mathcal{D}_\alpha$).

Figure 7.7: Trial main memory mining algorithm

If, at any time during *trialmainmine* we run out of main memory, we abort and return the partially constructed FP-tree, and a pointer to where we stopped scanning the database. We then resume processing *Diskmine*($\mathcal{D}_\alpha, M$) by computing a grouping $\beta_1, \ldots, \beta_k$ of *freqstring*($\mathcal{D}_\alpha$), and then decomposing $\mathcal{D}_\alpha$ into $\mathcal{D}_{\alpha.\beta_1}, \ldots, \mathcal{D}_{\alpha.\beta_k}$. We recursively process each decomposed database $\mathcal{D}_{\alpha.\beta_j}$. During the first level of the recursion, some groups β_j will consist of a single item only. If β_j is a singleton, we call *Diskmine*, otherwise we call *mainmine* directly, since we put several items in a group only when we estimate that the corresponding FP-tree will fit into main memory.

In computing the grouping $\beta_1, \ldots, \beta_k$ we assume that transactions in a very large database are evenly distributed, i.e., if a FP-tree is constructed from part of a database, then this FP-tree represents the FP-tree for the whole database. In other words, if the size of the FP-tree is n for p% of the database, then the size of the FP-tree for whole database is $n/p \cdot 100$. Most of the time, this gives an overestimation, since a FP-tree increases fast only at the beginning stage, when items are encountered for the first time and inserted into the tree. In the later stages, the changes to the

FP-tree will be mostly counter updates.

Procedure *mainmine*($\mathcal{D}_{\alpha.\beta}$)
build a modified FP-tree $T_{\alpha.\beta}$ for $\mathcal{D}_{\alpha.\beta}$
for each $i \in \{\beta\}$ **do begin**
  construct the FP-tree $T_{\alpha.i}$ for $\mathcal{D}_{\alpha.i}$ from $T_{\alpha.\beta}$
  call *FPgrowth**($T_{\alpha.i}$) and **return** *freqsets*($\mathcal{D}_{\alpha.i}$).
end

Figure 7.8: Main memory mining algorithm

In *basicdiskmine*, there is only one master item in each projected database (for $\mathcal{D}_\epsilon$, no master item at all), a FP-tree can be constructed without considering the master item. In Figure 7.8, since $\mathcal{D}_{\alpha.\beta}$ is for multiple master items, the FP-tree constructed from $\mathcal{D}_{\alpha.\beta}$ has to contain those master items. However, the item order is a problem for the FP-tree, because we only want to mine all frequent itemsets that contain master items. To solve this problem, we simply use the item order in the partial FP-tree returned by the aborted *trialmainmine*($\mathcal{D}_\alpha$). This is what we mean by a "modified FP-tree" on the first line in the algorithm in Figure 7.8.

The entire recurrence structure of *Diskmine* can be seen in Figure 7.5. Compared to the basic projection in Figure 7.3 we see that since the aggressive projection uses main memory more effective, the decomposition phase is shorter, resulting in less I/O.

In Figure 7.5, the shaded area shows the recursive structure of FP-growth*. Comparing with the shaded area in Figure 7.3 which shows the recursive structure of the FP-growth method, we can see that the main difference is the extra shaded level in Figure 7.5. This level is for the FP-trees of groups. For each group, since the total size of all FP-trees for its master items may be greater than the size of main memory, a "modified FP-tree" is constructed. This FP-tree will fit in main memory. From the FP-tree, smaller FP-trees can be constructed one by one, as shown in both figures. As an example, in Figure 7.3, *basicdiskmine* enters the main memory phase for instance for the conditional database $\mathcal{D}_{\epsilon.a}$. Then FP-growth first constructs the FP-tree $T_{\epsilon.a}$ from $\mathcal{D}_{\epsilon.a}$ (in Figure 7.5, $T_{\epsilon.a}$ is constructed from $T_{\epsilon.ab}$). The tree rooted at $T_{\epsilon.a}$ shows the recursive structure of FP-growth, assuming for simplicity that the relative frequency remains the same in all conditional pattern bases.

Theorem 7.1 *Diskmine*($\mathcal{D}$) *returns freqsets*($\mathcal{D}$).

Proof. The correctness of *Diskmine* can be derived from the correctness of the FPgrowth* method in [21] and Lemma 7.2 in Section 7.2. In *Diskmine*, each item acts as master item in exactly one

projected database. If a projected database is only for one master item i_j, the result of FPgrowth* method or a recursive call of *Diskmine* will be $freqsets(\mathcal{D}_{i_j})$. If a projected database is for a set $\{\beta\}$ of master items, it contains all frequency information associated with the master items. Since in the FPgrowth* method, the order of the items in a FP-tree does not influence the correctness of the FPgrowth* method, *mainmine* indeed returns only frequent itemsets that contain master item(s), i.e. *mainmine* gives the exact value of $freqsets(\mathcal{D}_{\alpha.\beta})$. According to Lemma 7.2, algorithm *Diskmine* then correctly outputs all itemsets in frequent the original database. ■

7.2.2 Memory management

Given a database $\mathcal{D}_\alpha$, to successfully apply the FPgrowth* method, the basic main memory requirement is that the size of the FP-tree T_α constructed from $\mathcal{D}_\alpha$, is less than the available amount M of main memory. In addition, we need space for the descendant conditional FP-trees that will be constructed during the recursive calls of FPgrowth*.

Suppose the main memory requirement for T_α plus its descendant FP-trees is m. If $M < m$, but the difference $m - M$ is not very big, the FPgrowth* method could still be run because the operating system uses virtual memory. However, there could be too many page swappings which take too much time and make FPgrowth* very slow. Therefore, given M, for a very large database $\mathcal{D}_\alpha$, we have to stop the construction of the FP-tree T_α and the execution of FPgrowth* method before all physical main memory is used up.

Another problem is that we will construct a large number of FP-trees. Since there can be millions of nodes in those FP-trees, inserting and deleting nodes is time-consuming.

In the implementation of the algorithm, we use our own main memory management for allocating and deallocating nodes, and calculating the main memory we have already used. We first use the method introduced in Section 3.3 to avoid freeing nodes in FP-trees one by one. Then we assume that the main memory needed by a FP-tree is proportional to the number of nodes in the FP-trees. We also assume that the workspace needed for calling *FPgrowth*(T)* method on a FP-tree is roughly 10% of the size of the FP-tree T. Here, 10% is a liberal assumption according to the experimental result in [28]. Later in this section, a more accurate value will be given. If the size of FP-tree is more than $0.9 \cdot M$, we conclude that M is not big enough to store whole FP-tree T_α.

7.2.3 Applying the FP-array technique

In *Diskmine*, the FP-array technique is also applied to save FP-tree traversals. Furthermore, when projected databases are generated, the FP-array technique can save a great number of disk I/O's.

Recall that in *trialmainmine*, if a FP-tree can not be accommodated in main memory, the construction stops. Suppose now we decided to stop scanning the database. Then later, after generating all projected databases, for a projected database with only one master item, two database scans are required to construct a FP-tree for the master item. The first scan gets all frequent items for the master item, the second scan constructs the FP-tree. For a projected database with several master items, though the FP-tree constructed from the database uses the modified item order (the order from the header of the FP-tree in the previous level of the recursion), to construct new FP-trees for the master items, two FP-tree traversals are needed. To avoid the extra scan, in *Diskmine* we calculate an array for each FP-tree. When constructing the FP-tree from $\mathcal{D}_\alpha$, if it is found that the tree can not fit in main memory, the construction of the FP-tree T_α stops, but the scan of the database $\mathcal{D}_\alpha$ continues so that we finish filling the cells of the FP-array A_α. Here, some extra disk I/O's are spent, but the payback will be that we save one database scan for each projected database. Furthermore, finishing the scanning of $\mathcal{D}_\alpha$ does not require any more main memory, since the array A_α is already there.

From the FP-array, for each projected database, the count of each pair of master items and the count of each pair of master item and slave item can be known. As an example, suppose a projected database is only for one master item i_j and slave items $i_1, \ldots, i_{j-1}$. To mine all frequent itemsets, from the line for i_j in the FP-array, accurate counts for $[i_j, i_{j-1}], [i_j, i_{j-2}], \ldots, [i_j, i_1]$ can be easily found. If there were no FP-array we would need an extra database scan.

With the FP-array, we can also make a projected database drastically smaller. In the definition of $\mathcal{D}_{\alpha.\beta_j}$, we see that $\mathcal{D}_{\alpha.\beta_j}$ is a database of all items in $\beta_1, \ldots, \beta_j$ Actually, by checking the array A_α, if a slave item is found not frequently co-occurring with any master item in β_j, it is useless to include the slave item in $\mathcal{D}_{\alpha.\beta_j}$, because no frequent itemsets mined from $\mathcal{D}_{\alpha.\beta_j}$ will contain that slave item. For the same reason, if we also find that a master item a is not frequent with any other master item or slave item, it will not be written to $\mathcal{D}_{\alpha.\beta_j}$, either. However, the frequent itemset $\alpha.a$ is outputted. Furthermore, if from the FP-array, we see that a master item a is only frequent with one item (master or slave) b, frequent itemsets $\alpha.a$ and $\alpha.a.b$ are outputted directly, and item a

will not appear in $\mathcal{D}_{\alpha.\beta_j}$. Therefore, by looking through the array, we find all slave items, such that they are not frequent with any master item in β_j, and all master items, such that their number of frequent items in $\{\beta_1, \ldots, \beta_j\}$ is 0 or 1. When generating $\mathcal{D}_{\alpha.\beta_j}$, all those items are removed from the transactions we put in $\mathcal{D}_{\alpha.\beta_j}$.

As an example, suppose in a projected database $\mathcal{D}_\alpha$ the *freqstring*($\mathcal{D}_\alpha$) = $abcdefgh$. We also suppose that $\mathcal{D}_\alpha$ will be deeply decomposed as two projected databases. One projected database is for group *abcd*, while the other is for group $efgh$. The FP-array A_α is shown in Figure 7.9. If now the minimum support is 10001, then from the FP-array, none of the item pairs are frequent. Therefore, we don't need to generate any projected database. We only need to output 8 frequent itemsets, i.e., $\alpha.a$ to $\alpha.g$ with their support. If now the minimum support is 5000, then in $\mathcal{D}_{\alpha.efgh}$, according to the definition, it contains all transactions that contain e, f, g, or h. Items included in $\mathcal{D}_{\alpha.efgh}$ are a, b, c, d, e, f, g, h. However, with the existence of A_α, we can see that f is not frequent with any other items, so we just output $\alpha.f$ and its support directly, and in $\mathcal{D}_{\alpha.efgh}$, item f is not included. Furthermore, items g and h are only frequent with each other, so we can just output frequent itemsets $\alpha.g$, $\alpha.h$, and $\alpha.gh$ with their support directly. In $\mathcal{D}_{\alpha.efgh}$, we remove g and h as well. Since e is frequent with a, b, c and d, we have to generate the projected database. But now in $\mathcal{D}_{\alpha.efgh}$, there are only items a, b, c, d, and e. By FP-array technique, now the size of $\mathcal{D}_{\alpha.efgh}$ becomes very small.

b	10000						
c	9900	9100					
d	9000	7500	8800				
e	8900	8700	8800	8900			
f	4000	3500	3000	2500	2000		
g	1000	1000	1000	2000	2000	2500	
h	1000	2000	1000	2000	3000	2000	9000
	a	b	c	d	e	f	g

Figure 7.9: The FP-array A_α

7.2.4 Statistics

Algorithm *Diskmine* collects some statistics on the partial FP-tree T_α and the rest of database $\mathcal{D}_\alpha$, for the purpose of grouping items together. Table 7.1 shows the statistics information. In the table, $\mathcal{D}_\alpha$

is the original database or the current projected database, and *freqstring*($\mathcal{D}_\alpha$)= $i_1 \ldots i_j \ldots i_k \ldots i_n$. The partial FP-tree is T_α and δ the absolute value of the minimum support.

$t(\mathcal{D}_\alpha)$	Number of transactions in $\mathcal{D}_\alpha$
$A_\alpha[j,k]$	Count of frequent item pair $\{i_j, i_k\}$ in $\mathcal{D}_\alpha$
$t(T_\alpha)$	Number of transactions used for constructing T_α
$\nu(T_\alpha)$	Number of nodes in T_α
$\nu[j](T_\alpha)$	Number of nodes in T_α if we retain only nodes for items $i_1, \ldots, i_j$
$\mu[j](T_\alpha)$	Number of nodes in T, where a node P for item i_k is counted if it satisfies the following conditions: 1) P is in a branch that contains i_j 2) $i_k \in \{i_1, \ldots, i_j\}$ 3) $A_\alpha[j,k] > \delta$

Table 7.1: Statistics from the partial FP-tree T_α

In the table, the FP-array discussed in Section 7.2.3 is also listed as statistics. Values for the cells of the FP-array are accumulated during the construction of the partial T_α. If *trialmainmine* is aborted, the rest of the statistics is collected by scanning the remaining part of $\mathcal{D}_\alpha$. Values in $\nu[j](T_\alpha)$ can also be obtained during the construction of T_α. Here $\nu[j](T_\alpha)$ records the size of the FP-tree after T_α is trimmed and only contains items $i_1, \ldots, i_j$. Notice that $\nu(T_\alpha)$ is equal to $\nu[n](T_\alpha)$. This is also the size of a tree that can fit in main memory. The value for $\mu[j](T_\alpha)$ can be obtained by traversing T_α once, it gives the size of the FP-tree $T_{\alpha.i_j}$.

It might seem that collecting all this statistics is a large overhead, however, since all work is done in main memory, it does not take much time. And the time saved for disk I/O's is far more than the time spent on gathering statistics.

7.2.5 Grouping items

In Figure 7.6, the fourth line computes a grouping $\beta_1\beta_2 \cdots \beta_k$ of $freqstring(\mathcal{D}_\alpha)$. Each string β corresponds to a group and each β consists of at least one item. For each β, a new projected database $\mathcal{D}_{\alpha.\beta}$ will be computed from $\mathcal{D}_\alpha$, then written to disk and read from disk later. Therefore, the more groups, the more disk I/O's. In other words, there should be as many items in each β as possible. To group items, two questions have to be answered.

1. If β currently only has one item i_j, after projection, is the main memory big enough for accommodating $T_{\alpha.i_j}$ constructed from $\mathcal{D}_{\alpha.i_j}$ and running the FPgrowth* method on $T_{\alpha.i_j}$?

2. If more items are put in β, after projection, is the main memory big enough for accommodating $T_{\alpha.\beta}$ constructed from $\mathcal{D}_{\alpha.\beta}$ and running FPgrowth* on $T_{\alpha.\beta}$ only for items in β?

Answering the first question is pretty easy, since for each item i_j, the number $\mu[j](T_\alpha)$ gives the size of a FP-tree if the tree is constructed from the partial FP-tree T_α. Therefore $\mu[j](T_\alpha)$ can be used to estimate the size of FP-tree $T_{\alpha.i_j}$. By the assumption that the transactions in $\mathcal{D}_\alpha$ are evenly distributed and that the partial T_α represents the whole FP-tree for $\mathcal{D}_\alpha$, the estimated size of FP-tree $T_{\alpha.i_j}$ is $\mu[j](T_\alpha) \cdot t(\mathcal{D}_\alpha)/t(T_\alpha)$.

Before answering the second question, we introduce the *cut point* from which the first group can be easily found.

Finding the cut point. Recall the order that FPgrowth* uses in mining frequent itemsets. Starting from the least frequent item i_n, all frequent itemsets that contains i_n are mined first. Then the process is repeated for i_{n-1}, and so on. Notice that when mining frequent itemsets for i_k, all frequency information about $i_{k+1}, \ldots, i_n$ is useless. Thus, though a complete FP-tree T_α constructed from $\mathcal{D}_\alpha$ could not fit in main memory, we can find many k's such that the trimmed FP-tree containing only nodes for items $i_k, \ldots, i_1$ will fit into main memory. All frequent itemsets for $i_k, \ldots, i_1$ can be then mined from one trimmed tree. We call the biggest of such k's the *cut point*. At this point, main memory is big enough for storing the FP-tree containing only $i_k, \ldots, i_1$, and there is also enough main memory for running FPgrowth* on the tree. Obviously, if the cut point k can be found, items $i_k, \ldots, i_1$ can be grouped together. Only one projected database is needed for $i_k, \ldots, i_1$.

There are two ways to estimate the cut point. One way is to get cut point from the value of $t(\mathcal{D}_\alpha)$ and $t(T_\alpha)$ in Table 7.1. Figure 7.10 illustrates the intuition behind the cut point. In the figure, $l = t(T_\alpha)$, and $m = t(\mathcal{D}_\alpha)$. Since the partial FP-tree for $t(T_\alpha)$ of $t(\mathcal{D}_\alpha)$ transactions can be accommodate in main memory, we can expect that the FP-tree containing $i_k, \ldots, i_1$, where $k = \lfloor n \cdot t(T_\alpha)/t(\mathcal{D}_\alpha) \rfloor$, also will fit in main memory.

The above method works well for many databases, especially for those databases whose corresponding FP-trees have plenty of sharing of prefixes for items from i_1 to the cut point. However, if the FP-tree constructed from a database does not share prefixes that much, the estimation could fail, since now the FP-tree for items from i_1 to the cut point could be too big. Thus, we have to consider another method. In Table 7.1, $\nu[j](T_\alpha)$ records the size of the FP-tree after the partial FP-tree T_α is trimmed and only contains items $i_1, \ldots, i_j$. Based on $\nu[j](T_\alpha)$ the number of nodes in the complete FP-tree for item i_j can be estimated as $\nu[j](T_\alpha) \cdot t(\mathcal{D}_\alpha)/t(T_\alpha)$. Now, finding the cut point becomes

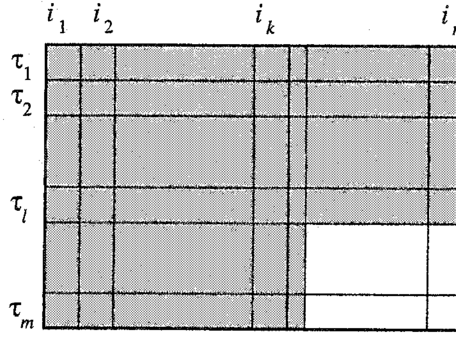

Figure 7.10: Cutpoint

finding the biggest k such that $\nu[k](T_\alpha) \cdot t(\mathcal{D}_\alpha)/t(T_\alpha) \leq \nu(T_\alpha)$, and $\nu[k+1](T_\alpha) \cdot t(\mathcal{D}_\alpha)/t(T_\alpha) > \nu(T_\alpha)$.

Sometimes the above estimation only guarantees that the main memory is big enough for the FP-tree which contains all items between i_1 and the cut point, while it does not guarantee that the descendant trees from that FP-tree can fit in main memory. This is because the estimation does not consider the size of descendant trees correctly (in Section 7.2.2, we assumed that the size of a conditional tree is 10% of its nearest ancestor tree). Actually, from $\mu[j](T_\alpha)$ we can get a more accurate estimation of the size of the biggest descendant tree. To find the cut point, we need to find the biggest k, such that $(\nu[k](T_\alpha) + \mu[j](T_\alpha)) \cdot t(\mathcal{D}_\alpha)/t(T_\alpha) \leq \nu(T_\alpha)$, and $(\nu[k+1](T_\alpha) + \mu[m](T_\alpha)) > \nu(T_\alpha)$, where $j \leq k$, $\mu[j](T_\alpha) = max_{j \in \{1,\ldots,k\}} \mu[j](T_\alpha)$, and $m \leq k+1$, $\mu[m](T_\alpha) = max_{m \in \{1,\ldots,k+1\}} \mu[m](T_\alpha)$.

Grouping the rest of the items. Now we answer the second question, how to put more items into a group? Here we still need $\mu[j](T_\alpha)$. Starting with $\mu[cutpoint+1](T_\alpha)$, we test if $\mu[cutpoint+1](T_\alpha) \cdot t(\mathcal{D}_\alpha)/t(T_\alpha) > \nu(T_\alpha)$. If not, we put next item cutpoint+2 into the group, and test if $(\mu[cutpoint+1](T_\alpha) + \mu[cutpoint+2](T_\alpha)) \cdot t(\mathcal{D}_\alpha)/t(T_\alpha) > \nu(T_\alpha)$. We repeatedly put next item in $freqstring(\mathcal{D})$ into the group until we reach an item i_j, such that

$$\sum_{m=cutpoint+1}^{j} \mu[m](T_\alpha) \cdot t(\mathcal{D}_\alpha)/t(T_\alpha) > \nu(T_\alpha).$$

Then starting from i_j, we put items into next group, until all items find its group.

Why can we put items $i_j, \ldots, i_k$ together into β? This is because even if we construct $T_{\alpha.i_j}, \ldots, T_{\alpha.i_k}$ from the projected databases $\mathcal{D}_{\alpha.i_j}, \ldots, \mathcal{D}_{\alpha.i_k}$ and put all of them into main memory, the main memory is big enough according to the grouping condition. At this stage, $T_{\alpha.i_j}, \ldots, T_{\alpha.i_k}$ all can be constructed by scanning $\mathcal{D}_\alpha$ once. Then we mine frequent itemsets from the FP-trees. However, we

can do better. Obviously $T_{\alpha.i_j}, \ldots, T_{\alpha.i_k}$ overlap a lot, and the total size of the trees is definitely greater than the size of $T_{\alpha.\beta}$. It also means that we can put more items into each β, only if the size of $T_{\alpha.\beta}$ is estimated to fit in main memory. To estimate the size of $T_{\alpha.\beta}$, part of T_α has to be traversed by following the links for the master items in T_α.

7.2.6 Database projection

After all items have found their groups, the original database will be projected to small databases according to Definition 7.4. To save disk I/O's, three techniques can be used:

1. In a group β, if the number of master items is greater than half of the number of frequent items (this often happens in the group that contains cut point), then $\mathcal{D}_{\alpha.\beta}$ is not necessary computed. To mine all frequent itemsets, $T_{\alpha.\beta}$ can be directly constructed from $\mathcal{D}_\alpha$ by reading it once. This is because $\mathcal{D}_{\alpha.\beta}$ is not much smaller than $\mathcal{D}_\alpha$, while the disk I/O's for reading from $\mathcal{D}_\alpha$ once is less than the disk I/O's for writing and reading $\mathcal{D}_{\alpha.\beta}$ once.

2. Since the partial tree T_α is now in main memory, it records all frequency information of those transactions that have been read so far, when computing projected databases, the frequency information of those transactions can be gotten from T_α. Thus disk I/O's are only spent on reading from those transactions that did not contribute to T_α.

3. As discussed in Section 7.2.3, by using the FP-array technique, in group β_j, we find all slave items, such that they are not frequent with any master item in β_j, and all master items, such that their number of frequent items in $\{\beta_1, \ldots, \beta_j\}$ is 0 or 1. When computing $\mathcal{D}_{\alpha.\beta_j}$, all those items are removed from new transactions in $\mathcal{D}_{\alpha.\beta_j}$.

7.2.7 The disk I/O's

Let's re-count the disk I/O's used in *Diskmine*. The first scan is still for obtaining all frequent items in $\mathcal{D}_\epsilon$, and it needs D/B disk I/O's. In the second scan we construct a partial FP-tree T_ϵ, then continue scanning the rest database for statistics. The second scan is a full scan, which needs another D/B disk I/O's. Suppose then that k projected databases have to be computed. According to Section 7.2, the total size of the projected databases is approximately $k/2 \cdot D$. For computing the projected databases, the frequency information in T_ϵ is reused, so only part of $\mathcal{D}_\epsilon$

is read. We assume on average half of $\mathcal{D}_\epsilon$ is read at this stage, which means $1/2 \cdot D/B$ disk I/O's. By using of the FP-array technique, writing and later reading k projected databases now only take $2 \cdot k/2 \cdot D/B = k \cdot D/B$ disk I/O's. Suppose all frequent itemsets can be mined from the projected databases without going to the third level. Then the total disk I/O's is

$$5/2 \cdot D/B + k \cdot D/B \tag{5}$$

Compared with formula 4, *Diskmine* saves at least $k/2 \cdot D/B$ disk I/O's, thanks to the various techniques used in the algorithm.

7.3 Experimental Results

In this section, we present the results from a performance comparison of *Diskmine* with the *Parallel Projection Algorithm* in [29] and the *Partitioning Algorithm* introduced in [48]. The scalability of *Diskmine* is also analyzed, and the accurateness of our memory size estimations are validated.

As mentioned in Section 7.2, the Parallel Projection Algorithm is a basic divide-and-conquer algorithm, since for each item a projected database is created. For performance comparison, we implemented Parallel Projection Algorithm, by using FP-growth as main memory method, as introduced in [29]. The Partitioning Algorithm is also a divide-and-conquer algorithm. We implemented the partitioning algorithm by using the Apriori implementation in [3]. We chose this implementation, since it was well written and easy to adapt for our purposes.

We ran the three algorithms on both synthetic datasets and real datasets. Some synthetic datasets have millions of transactions, and the size of the datasets ranges from several megabytes to several hundreds gigabytes. Without loss of generality, only the results for some synthetic datasets and a real dataset are shown here.

All experiments were performed on a 2.0GHz Pentium 4 with 256 MB of memory under Windows XP. For *Diskmine* and the Parallel Projection Algorithm, the size of the main memory is given as an input. For the Partitioning Algorithm, since it only has two database scans and each main-memory-sized partition and all data structures for Apriori are stored into main memory, the size of main memory is not controlled, and only the runtime is recorded.

We first compared the performance of three algorithms on synthetic dataset. Dataset *T100I20D100K* was generated from the application of IBM research center [1]. The dataset has 100,000 transactions

and 1000 items, and occupies about 40 megabytes of memory. The average transaction length is 100, and the average pattern length is 20. The dataset is very sparse and the FP-tree constructed from the dataset is bushy. For Apriori, a large number of candidate frequent itemsets will be generated from the dataset.

When running the algorithms, the main memory size was set as 128 megabytes. Figures 7.11, 7.12, 7.13 shows the experimental results. In the figures, "Basic Algorithm" represents the Parallel Projection Algorithm, and "Aggressive Algorithm" represents the *Diskmine* algorithm.

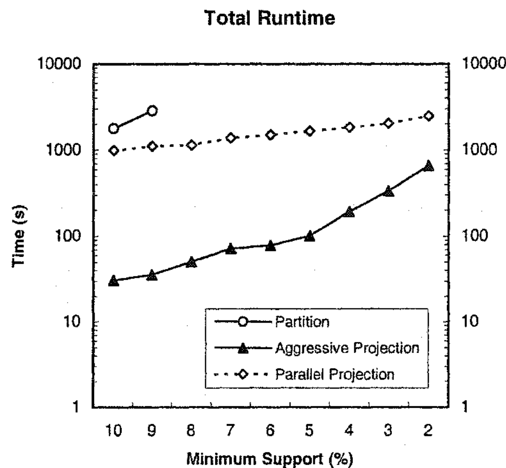

Figure 7.11: Performance of algorithms running on synthetic dataset: Runtime

In Figure 7.11 we compare the total runtime, we can see that the Partitioning Algorithm is the slowest in the group. In the figure, there are no points for the algorithm when minimum support is very low, since the algorithm is extremely slow and we gave up recording its runtime. The Aggressive algorithm has the best performance. It's more than an order of magnitude faster than the Basic algorithm when the minimum support is high.

To see the differences of the CPU time and the time used for disk I/O's between the Aggressive algorithm and the Basic algorithm, we recorded the CPU time and the time for disk I/O's of each algorithm separately. Figure 7.12 and Figure 7.13 shows the CPU time and the time for disk I/O's used by each algorithm, respectively. In Figure 7.12, as expected, we can see that the disk I/O time of the Aggressive algorithm is orders of magnitude smaller than that of the Basic algorithm. On the other hand, in Figure 7.13 we can see that the Basic algorithm, however, is not slower than the Aggressive algorithm if we only compare their CPU time. In [21], where we were concerned with main memory mining, we found that if a dataset is sparse the boosted FPgrowth* method

has a much better performance than the original FP-growth. The reason here the CPU time of the Aggressive algorithm is not always less than that of Basic algorithm is that the Aggressive algorithm has to spend CPU time on calculating statistics. However, from Figure 7.11, we also can see that the CPU overhead used by the Aggressive algorithm now become insignificant compared to the savings in disk I/O.

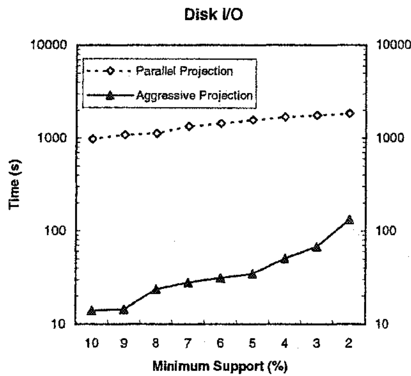

Figure 7.12: Performance of algorithms running on synthetic dataset: Time for Disk I/O's

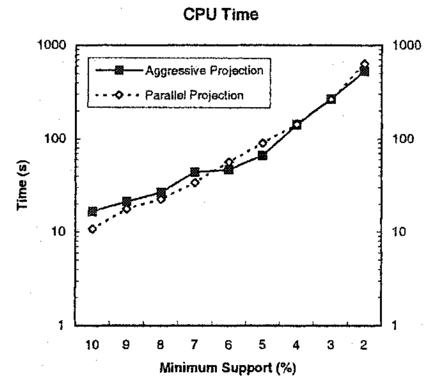

Figure 7.13: Performance of algorithms running on synthetic dataset: CPU time

We then ran the algorithms on a real dataset *Kosarak*, which is used as a test dataset in [2]. The dataset is about 40 megabytes. Since it is a dense dataset and its FP-tree is pretty small, we set the main memory size as 16 megabytes for the experiments. Results are shown in Figures 7.14, 7.15, 7.16.

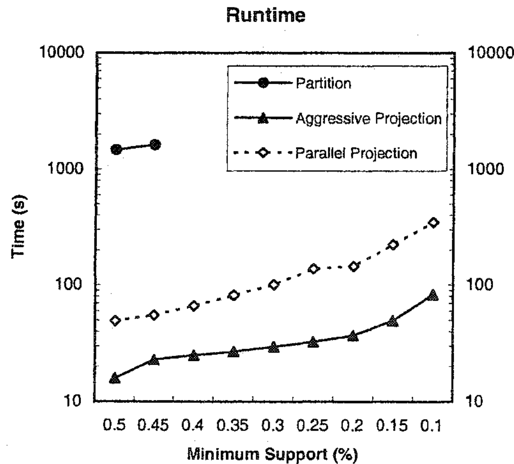

Figure 7.14: Performance of algorithms running on real dataset: Runtime

In Figure 7.11, the Partitioning Algorithm is still the slowest. We only recorded the runtime for

high minimum support. The algorithm is extremely slow when the minimum support is low. This is because it generates too many candidate frequent itemsets. Together with the data structures, these candidate sets use up main memory and virtual memory was used. By separating the CPU time and the time for disk I/O's needed by the Aggressive algorithm and the Basic algorithm, we get Figure 7.15 and Figure 7.16. In Figure 7.15, the time used for disk I/O's of the Aggressive algorithm is still remarkably less than the time used for disk I/O's of the Basic Algorithm. We can again notice that in Figure 7.16 the CPU time of the Basic Algorithm is less than that of the Aggressive algorithm. This is because *Kosarak* is a dense dataset so the FP-array technique does not help a lot. In addition, calculating the statistics takes an amount of time.

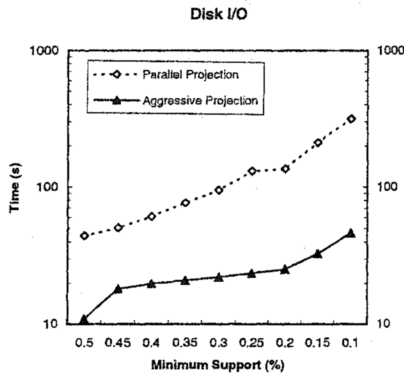

Figure 7.15: Performance of algorithms running on real dataset: Time for Disk I/O's

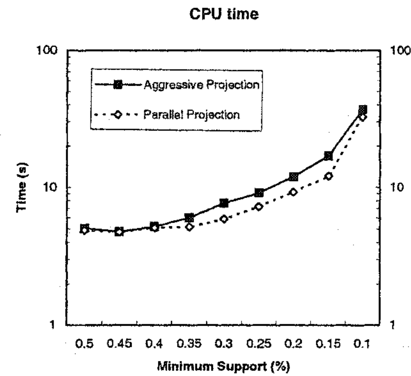

Figure 7.16: Performance of algorithms running on real dataset: CPU time

To test the effectiveness of the techniques for grouping items, we run *Diskmine* on *T100I20D100K* and see how close the estimation of the FP-tree size for each group is to its real size. We still set the main memory size as 128 megabytes, the minimum support $\delta = 2\%$. When generating the projected databases, items were grouped into 7 groups (the total number of frequent items is 826). As we can see from Figure 7.17, in all groups, the estimated size is always slightly larger than the real size. Compared with the Basic Algorithm, which constructs a FP-tree for each item from its projected database, the Aggressive Algorithm almost fully uses the main memory for each group to construct a FP-tree.

As a divide-and-conquer algorithm, one of the most important properties of *Diskmine* is its good scalability. We ran *Diskmine* on a set of synthetic datasets. In all datasets, the item number was set as 10000 items, the average transaction length as 100, and the average pattern length as 20. The number of the transactions in the datasets varied from 200,000 to 2,000,000. Datasets size

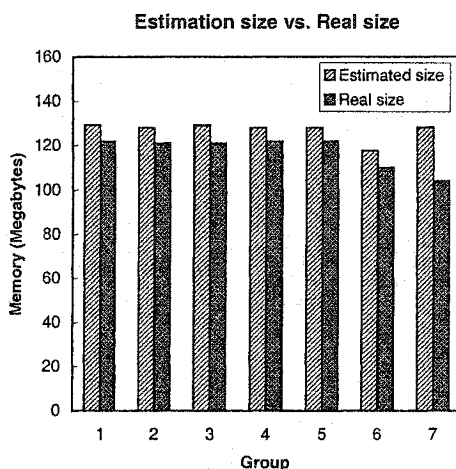

Figure 7.17: Estimation Accuracy

ranges from 100 megabytes to 1 gigabyte. Minimum support was set as 1.5%, and the available main memory was 128 megabytes. Figure 7.18 shows the results. In the figure, the CPU and the disk I/O time is always kept in a small range of acceptable values. Even for the datasets with 2 million transactions, the total runtime is less than 1000 seconds. Extrapolating from these figures using formula (4), we can conclude that a dataset the size of the Library of Congress collection (25 Terabytes) could be mined in around 18 hours with current technology.

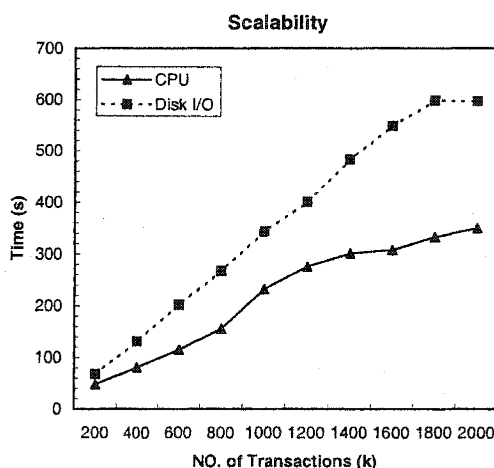

Figure 7.18: Scalability of Diskmine

Chapter 8

Conclusions and Future Work

Data mining is known as "discovery of useful summaries of data" or "knowledge discovery". Mining association rules is one of the most important data mining tasks. In this thesis, we focus on the efficient algorithms for mining frequent itemsets which is the core of mining association rules.

This chapter summarizes the work done in this thesis first, then discussions for the future work are given.

8.1 Summary of this thesis

Mining association rule is one of the most important tasks of data mining. The key to mining association rule is to solve the problem of mining frequent itemsets. Mining all frequent itemsets, maximal frequent itemsets, and closed frequent itemsets are three categories of the problem.

Since a database can be very large and at the same time the main memory size is always limited, we can not expect that the data structures for representing database and/or the data structures for keeping itemsets always fit in memory. Thus, algorithms for mining from secondary memory are also needed.

In this thesis, we considered the following factors for algorithms for mining frequent itemsets: a) The number of database scans required by the algorithms; b) The data structures used in the algorithms; c) Memory management of the algorithms; d) For mining maximal and closed frequent itemsets, the efficiency of the algorithms for maximality checking and closedness testing. We selected the FP-tree structure, a compact and efficient data structure originally proposed by J. Han *et al.*

in [28], as the main data structure used in our algorithms. Using a FP-tree requires two database scans to mine frequent itemsets. Our contributions are as follows:

1. We introduced a novel FP-array technique that allows using FP-trees more efficiently when mining frequent itemsets. The technique greatly reduces the time spent traversing FP-trees, and works especially well for sparse datasets.

2. By using the FP-array technique in the FP-growth method [28], the FPgrowth* algorithm is introduced to mine all frequent itemsets. Both our experimental results and the results of the independent experiments conducted by the organizers of FIMI'03 [15] show that FP-growth* is one of the best known algorithms for mining all frequent itemsets. Our experimental results also show that FP-growth* has small main memory consumption and good scalability.

3. Similarly we developed an algorithm FPmax for mining maximal frequent itemsets. In the algorithm, FPmax uses the FP-tree structure, as well as an effective maximality checking approach. For the maximality testing, a variation of the FP-tree, called a MFI-tree, is introduced to keep track of all MFI's. In FPmax, a newly found frequent itemset is always compared with MFI's kept in a global MFI-tree. In order to understand the performance on datasets with different characteristics, we analyzed the probable behavior of GenMax, MAFIA and FPmax. Numerous experiments on synthetic datasets were done to validate our analysis. Experimental results show that FPmax outperforms GenMax and MAFIA in many cases, but not all cases.

4. Extending FPmax, we elaborated the FPmax* algorithm, which applies the FP-array technique and an efficient approach for maximality checking. In FPmax*, a newly found frequent itemset is always compared with a small set of MFI's that are kept in a MFI-tree. FPmax* is the best algorithm for mining maximal frequent itemsets as suggested by the organizers of FIMI'03. Our experimental results also show that FPmax* is a scalable algorithm with very low memory consumption.

5. For mining closed frequent itemsets we give the FPclose algorithm. In the algorithm, a CFI-tree, another variation of a FP-tree, is used for testing the closedness of frequent itemsets. The compactness of its data structure, the efficiency of the approach for closedness testing, and careful implementation made FPclose the fastest algorithm in FIMI'03 [15].

6. We introduced several divide-and-conquer algorithms for mining frequent itemsets from secondary memory. The recurrences and disk I/O's of all algorithms were analyzed. We then gave a detailed divide-and-conquer algorithm which almost fully uses the limited main memory and saves numerous number of disk I/O's. Many novel techniques are used in the algorithm Diskmine. Experimental results show that Diskmine successfully reduces the number of disk accesses, sometimes by orders of magnitude, and that our algorithm scales up to terabytes of data. The experiments also validate that the estimation techniques used in Diskmine are accurate.

8.2 Future work

Though the experimental results given in this thesis show the success of our algorithms, we still have a lot of work to do.

- Experimental results in Chapter 3, 5 and 6 show that FPgrowth*, FPmax* and FPclose consume lots of main memory, even though they are among the algorithms that have the lowest memory consumption. Consuming too much main memory reduces the scalability of the algorithms and makes it difficult to apply those algorithms for parallel data mining. We notice from the experimental result in Chapter 3 that using a Patricia Trie to implement the FP-tree data structure could be a good solution for the problem.
- Currently, there are very few efficient algorithms for mining *maximal* frequent itemsets and *closed* frequent itemsets from very large databases. Unlike in *Diskmine*, where the frequent itemsets mined from all projected databases are globally frequent, a maximal frequent itemset or a closed frequent itemset mined from a projected database is only locally maximal or closed. As a challenge, a data structure, whose size may be very big, must be set for keeping all already discovered maximal or closed frequent itemsets.
- For the work done in Chapter 7, we notice that our implementation of the partitioning algorithm is based on an existing Apriori implementation, which is not necessarily highly optimized. Furthermore, there are situations when there are not too many candidate itemsets in a database, but the FP-tree constructed from the database is very big. In this situation the Partitioning Algorithm only needs two database scans and all frequent items can be nicely mined

in main memory, or with very little I/O for keeping the candidate sets in virtual memory. In this situation *Diskmine* also needs two database scans, and it additionally needs to decompose the database. Therefore, exploring whether some clever disk-based data structure would make the partition approach scale, is another interesting direction for further research.

- Data mining may involve a huge quantity of data and amount of computation. In this thesis, we only considered about sequential mining of frequent itemsets. Parallel computing is also a crucial component for successful large-scale mining of frequent itemsets [41, 25, 57, 42, 56, 32]. Because of the excellent properties of FPgrowth*, FPmax* and FPclose, we believe that the combination of memory-consumption-improved FPgrowth*, FPmax* and FPclose with parallel computing is also a good research direction. Some work has been done in [13].

- The work done in [24] shows that Apriori algorithms can also be applied to XML documents. How to continue our work in [24] and apply the algorithms in this thesis to XML data is another direction for our future research.

Bibliography

[1] http://www.almaden.ibm.com/software/quest/Resources/index.shtml.

[2] http://fimi.cs.helsinki.fi.

[3] www.cs.helsinki.fi/u/goethals/software.

[4] http://www-sal.cs.uiuc.edu/ hanj/pubs/software.htm.

[5] R. C. Agarwal, C. C. Aggarwal, and V. V. V. Prasad. Depth first generation of long patterns, In *Proceedings of the sixth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 108-118, 2000.

[6] R. Agrawal, T. Imielinski, and A. Swami. Mining association rules between sets of items in large databases. In *1993 ACM-SIGMOD Proc. Special Interest Group on Management of Data (ACM-SIGMOD'93)*, pages 207–216.

[7] R. Agrawal and R. Srikant. Fast algorithms for mining association rules. In *Proceeding of International Conference on Very Large Data Bases*, pages 487–499, Santiago, Chile, Sept. 1994.

[8] R. Agrawal and R. Srikant. Mining sequential patterns. In *International Conference on Data Engineering (ICDE 1995)*, pages 3–14.

[9] R. J. Bayardo. Efficiently mining long patterns from databases. In *Proceeding of Special Interest Group on Management of Data*, pages 85–93, Seattle, WA, June 1998.

[10] C. Borgelt. Efficient implementations of Apriori and Eclat. In *Proceedings of the 1st Workshop on Frequent Itemset Mining Implementations (FIMI'03)*, Melbourne, FL, Nov. 2003.

[11] S. Brin, R. Motwani, and C. Silverstein. Beyond market basket: Generalizing association rules to correlations. In *Proceeding of Special Interest Group on Management of Data*, pages 265–276, Tucson, Arizona, May 1997.

[12] D. Burdick, M. Calimlim, and J. Gehrke. MAFIA: a maximal frequent itemset algorithm for transactional databases. In *Proceedings of the 17th International Conference on Data Engineering*, pages 443–452, Heidelberg, Germany, April 2001.

[13] T. Eavis, G. Grahne, A. Rau-Chaplin, and J. Zhu. Parallel mining of frequent itemsets in very large databases, in preparation.

[14] B. Goethals and M. J. Zaki (Eds.). *Proceedings of the First IEEE ICDM Workshop on Frequent Itemset Mining Implementations (FIMI '03)*. CEUR Workshop Proceedings, Vol 80 http://CEUR-WS.org/Vol-90.

[15] B. Goethals and M. J. Zaki. Advances in frequent itemset mining implementations: Introduction to FIMI03. In *Proceedings of the 1st Workshop on Frequent Itemset Mining Implementations (FIMI'03)*, Melbourne, FL, Nov. 2003.

[16] K. Gouda and M.J. Zaki. Efficiently mining maximal frequent itemsets. In *1st IEEE International Conference on Data Mining (ICDM)*, pages 163–170, San Jose, November 2001.

[17] G. Grahne, L. V. S. Lakshmanan, and X. Wang. Interactive mining of correlations - A constraints perspective. ACM SIGMOD workshop on research issues in data mining and knowledge discovery, pages 7-1, Philadelphia, 1999.

[18] G. Grahne, L. V. S. Lakshmanan, and X. Wang. Efficient mining of constrained correlated Sets. In *International Conference on Data Engineering (ICDE 2000)*, pages 512-521 San Diego, CA, 2000.

[19] G. Grahne, L. V. S. Lakshmanan, X. Wang, and M. Xie. On dual mining: From patterns to circumstances, and back. In *International Conference on Data Engineering (ICDE 2001)*, pages.195-204, April 2001.

[20] G. Grahne and J. Zhu. High performance mining of maximal frequent itemsets. In *Proceedings of Workshop on High Performance Data Mining: Pervasive and Data Stream Mining*, San Francisco, CA, May 2003.

[21] G. Grahne and J. Zhu. Efficiently using prefix-trees in mining frequent itemsets. In *Proceedings of the 1st IEEE ICDM Workshop on Frequent Itemset Mining Implementations (FIMI'03)*, Melbourne, FL, Nov. 2003.

[22] G. Grahne and J. Zhu. Fast algorithms for frequent itemset mining using prefix-trees. Submitted to *IEEE Transactions on Knowledge and Data Mining.*

[23] G. Grahne and J. Zhu. Mining frequent itemsets from secondary memory. To appear in *IEEE International Conference on Data Mining (ICDM'04).*

[24] G. Grahne and J. Zhu. Discovering approximate keys in XML data. In *Proceedings of the 2002 ACM CIKM International Conference on Information and Knowledge Management*, pages 453-460, McLean, VA, Nov. 2002.

[25] E.-H. Han, G. Karypis, and V. Kumar. Scalable parallel data mining for association rules. In *1997 ACM-SIGMOD Proc. Special Interest Group on Management of Data (ACM-SIGMOD'97)*, pages 277–288.

[26] J. Han and Y. Fu. Discovery of multiple-level association rules from large databases. In *Proc. 1995 Int. Conf. Very Large Data Bases (VLDB'95)*, pages 420–431.

[27] J. Han, J. Pei, G. Dong, and K. Wang. Efficient computation of iceberg cubes with complex measures *Proc. 2001 ACM-SIGMOD Int. Conf. on Management of Data (SIGMOD'01)*, Santa Barbara, CA, May 2001.

[28] J. Han, J. Pei, and Y. Yin. Mining frequent patterns without candidate generation. In *Proceeding of Special Interest Group on Management of Data* , pages 1-12, Dallas, TX, May 2000.

[29] J. Han, J. Pei, Y. Yin, and R. Mao. Mining frequent patterns without candidate generation: A Frequent-Pattern tree approach. In *Data Mining and Knowledge Discovery*, Vol. 8, pages 53-87, 2004.

[30] J. Han, J. Wang, Y. Lu, and P. Tzvetkov. Mining Top-K frequent closed patterns without minimum support. In *Proc. 2002 Int. Conf. on Data Mining (ICDM'02)*, Maebashi, Japan, Dec. 2002.

[31] M. Kamber, J. Han, and J.Y. Chiang. Metarule-guided mining of multi-dimensional association rules using data cubes. In *Proc. 1997 Int. Conf. Knowledge Discovery and Data Mining (KDD 97)*, Newport Beach, CA, pp. 207–210.

[32] A. Javed and A. Khokhar. Frequent pattern mining on message passing multiprocessor systems. To appear in *Distributed and Parallel Databases*, 16 (3): 321-334, November 2004.

[33] V.S. Lakshmanan, C. Leung, and R. Ng. The segment support map: Scalable mining of frequent itemsets. In *SIGKDD Explorations Special Issue on Scalable Data Mining*, Volume 2, Issue 2, pages 21-27. December 2000.

[34] L. V. S. Lakshmanan, R. Ng, J. Han, and A. Pang. Optimization of constrained frequent set queries with 2-variable constraints. In *1999 ACM-SIGMOD Proc. Special Interest Group on Management of Data (ACM-SIGMOD'99)*, pages 157–168, Philadelphia, PA, June 1999.

[35] G. Liu, H. Lu, J. X. Yu, W. Wei, and X. Xiao. AFOPT: An efficient implementation of pattern growth approach. In *Proceedings of the 1st Workshop on Frequent Itemset Mining Implementations (FIMI'03)*, Melbourne, FL, Nov. 2003.

[36] B. Lent, A. Swami, and J. Widom. Clustering association rules. In *Proc. 1997 Int. Conf. Data Engineering (ICDE'97)*, pages 220–231, Birmingham, England, Apr. 1997.

[37] H. Mannila, H. Toivonen, and A. I. Verkamo. Efficient algorithms for discovering association rules. In *Proc. AAAI'94 Workshop Knowledge Discovery in Databases (KDD'94)*, pages 181–192, Seattle, WA, July 1994.

[38] H. Mannila, H. Toivonen, and I. Verkamo. Discovery of frequent episodes in event sequences. In *Data Mining and Knowledge Discovery.* Volume 1, 3(1997), pages 259–289.

[39] R. Ng, L. V. S. Lakshmanan, J. Han, and A. Pang. Exploratory mining and pruning optimizations of constrained associations rules. In *1998 ACM-SIGMOD Proc. Special Interest Group on Management of Data (ACM-SIGMOD'98)*, pages 13–24, Seattle, WA, June 1998.

[40] S. Orlando, C. Lucchese, P. Palmerini, R. Perego, and F. Silvestri. kDCI: a multi-strategy algorithm for mining frequent sets. In *Proceedings of the 1st Workshop on Frequent Itemset Mining Implementations (FIMI'03)*, Melbourne, FL, Nov. 2003.

[41] J. S. Park, M. S. Chen, and P. S. Yu. An effective hash-based algorithm for mining association rules. In *1995 ACM-SIGMOD Proc. Special Interest Group on Management of Data (ACM-SIGMOD'95)*, pages 175–186.

[42] S. Pathasarathy, M. J. Zaki, M. Ogihara, and W. Li. Parallel data mining for association rules on shared-memory systems. In *Knowledge and Information Systems*, Volume 3, Number 1, pp 1-29, Feb 2001.

[43] N. Pasquier, Y. Bastide, R. Taouil, and L. Lakhal. Discovering frequent closed itemsets for association rules. In *ICDT'99*, Jan. 1999.

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

[45] J. Pei, J. Han, H. Lu, S. Nishio, S. Tang, and D. Yang. H-mine: Hyper-structure mining of frequent patterns in large databases. In *Proc. of IEEE Intl. Conference on Data Mining*, Pages 441-448, 2001.

[46] A. Pietracaprina and D. Zandolin. Mining frequent itemsets using patricia tries. In *Proceedings of the 1st Workshop on Frequent Itemset Mining Implementations (FIMI'03)*, Melbourne, FL, Nov. 2003.

[47] R. Rastogi and K. Shim. PUBLIC: A decision tree classifier that integrates building and pruning. In *Proceedings of the Very Large Database Conference (VLDB)*, New York, 1998.

[48] A. Savasere, E. Omiecinski, and S. Navathe. An efficient algorithm for mining association rules in large databases. In *Proceeding of Int. Conf. Very Large Data Bases* , pages 432–443, 1995.

[49] C. Silverstein, S. Brin, R. Motwani, and J. Ullman. Scalable techniques for mining causal structures. In *Proc. 1998 Int. Conf. Very Large Data Bases (VLDB'98)*, pages 594–605.

[50] R. Srikant and R. Agrawal. Mining generalized association rules. In *Proc. 1995 Very Large Data Base*, pp 407–419.

[51] H. Toivonen. Sampling large databases for association rules. In *Proceeding of Int. Conf. Very Large Data Bases*, pages 134–145, 1996

[52] T. Uno, T. Asai, Y. Uchida, and H. Arimura. LCM: An efficient algorithm for enumerating frequent closed item sets. In *Proceedings of the 1st Workshop on Frequent Itemset Mining Implementations (FIMI'03)*, Melbourne, FL, Nov. 2003.

[53] J. Wang, J. Han, and J. Pei. CLOSET+: Searching for the best strategies for mining frequent closed itemsets. In *Proc. 2003 ACM SIGKDD Int. Conf. on Knowledge Discovery and Data Mining (KDD'03)*, Washington, D.C., Aug. 2003.

[54] K. Wang, L. Tang, J. Han, and J. Liu. Top down FP-Growth for association rule mining. In *Proc. of the 6th Pacific Area Conference on Knowledge Discovery and Data Mining (PAKDD)*, 2002.

[55] D. Yellin. An algorithm for dynamic subset and intersection testing. In *Theoretical Computer Science*, Vol. 129: 397-406, 1994.

[56] O. R. Zaïane, M. El-Hajj, and P. Lu, Fast parallel association rule mining without candidacy generation, In *IEEE International Conference on Data Mining (ICDM'01)*, pages 665–668, Nov. 2001.

[57] M. J. Zaki. Parallel and distributed association mining: A survey. IEEE Concurrency 1999, Vol.7, No. 4, pp 14-24.

[58] M. J. Zaki. Scalable algorithms for association mining. In *IEEE Transactions on Knowledge and Data Mining*, 12(3):372-390, May-June 2000.

[59] M. J. Zaki and C. Hsiao. CHARM: An efficient algorithm for closed itemset mining. In *Proceeding of The 2nd SIAM International Conference on Data Mining*, Arlington, April 2002.

[60] M. J. Zaki and Karam Gouda. Fast vertical mining using diffsets. In *9th International Conference on Knowledge Discovery and Data Mining*, Washington, DC, August 2003.

[61] Q. Zou, W. W. Chu, and B. Lu. SmartMiner: A depth first algorithm guided by tail information for mining maximal frequent itemsets. In *Proceeding of IEEE International Conference on Data Mining (ICDM'02)*, Maebashi City, Japan, December, 2002.

[62] T. Zhang, R. Ramakrishnan, and M. Livny. BIRCH: A new data clustering algorithm and its applications. In *Data Mining and Knowledge Discovery.* Volume 1(2): 141-182 (1997).