SUPPORTING EFFICIENT AND SCALABLE FREQUENT PATTERN MINING

by

GUIMEI LIU

A Thesis Submitted to
The Hong Kong University of Science and Technology
in Partial Fulfillment of the Requirements for
the Degree of Doctor of Philosophy
in Computer Science

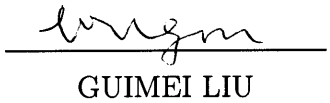
May 2005, Hong Kong

Authorization

I hereby declare that I am the sole author of the thesis.

I authorize the Hong Kong University of Science and Technology to lend this thesis to other institutions or individuals for the purpose of scholarly research.

I further authorize the Hong Kong University of Science and Technology to reproduce the thesis by photocopying or by other means, in total or in part, at the request of other institutions or individuals for the purpose of scholarly research.



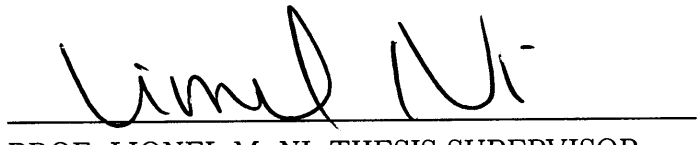
GUIMEI LIU

SUPPORTING EFFICIENT AND SCALABLE FREQUENT PATTERN MINING

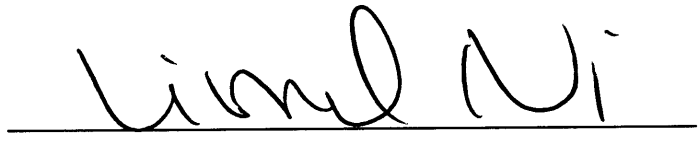
by

GUIMEI LIU

This is to certify that I have examined the above Ph.D. thesis
and have found that it is complete and satisfactory in all respects,
and that any and all revisions required by
the thesis examination committee have been made.



PROF. LIONEL M. NI, THESIS SUPERVISOR



PROF. LIONEL M. NI, HEAD OF DEPARTMENT

Department of Computer Science

5 May 2005

ACKNOWLEDGMENTS

I am very grateful to Professor Hongjun Lu, my thesis supervisor, for many things. This work would not have been possible without his continuous support, encouragement and guidance. He taught me how to conduct original research and write research papers, and gave me insightful suggestions on my research work. He shared with me his knowledge and experiences, and gave me valuable advice on research and career development. I benefited a lot from him. He was the best supervisor one can have, and I am so lucky to be his student. I wish I have learned more from him. He always lives in my heart and memories.

I would like to thank Professor Lionel M. Ni, who is willing to be my supervisor after Professor Lu passed away. I would also like to express my gratitude to Professor Frederick H. Lochovsky, Professor Fugee Tsung, Professor Qiang Yang and Professor Jeffrey Xu Yu for severing as my committee members. My gratitude also goes to Dr. Qing Luo who was a committee member for my thesis proposal defence, and Professor Dimitris Papadias and Professor Dit-Yan Yeung who severed as committee members for my PhD qualifying examination.

I would like to thank my research group members, Professor Jeffrey Xu Yu, Wei Wang, Haifeng Jiang, Wenwu Lou and Xiangye Xiao, for their kind help during the course of my PhD study. They gave me many valuable suggestions on my research. I learned a lot from them during research group meetings. Many discussions with them have been of great help to my research. I would also like to thank my friends in HKUST, in particular the friends in the database laboratory, for their help on many things.

I would like to thank the Department of Computer Science of Hong Kong University of Science and Technology for giving me the opportunity to pursue a PhD degree in computer science. I am also thankful to the support staff in computer science department for always being helpful.

Last but not least, I would like to thank my parents and my brother for their constant love and support.

TABLE OF CONTENTS

Title Page i

Authorization Page ii

Signature Page iii

Acknowledgments iv

Table of Contents v

List of Figures viii

List of Tables ix

Abstract x

Chapter 1 Introduction 1

1.1 Problem Definition 1

1.2 Motivations and Contributions 4

1.3 Organization of The Thesis 8

Chapter 2 Related Work 9

2.1 Frequent Itemset Mining Algorithms 9

2.1.1 The candidate generate-and-test approach 10

2.1.2 The filter-and-refine approach 11

2.1.3 The vertical mining approach 12

2.1.4 The pattern growth approach 13

2.2 Removing Redundancy from Mining Results 16

2.2.1 Mining frequent closed itemsets 16

2.2.2 Mining non-redundant association rules 17

2.2.3 Mining maximal frequent itemsets 18

2.3 Incremental Frequent Itemset Mining 19

2.4 Online Interactive Association Rule Mining 20

2.5 Mining Frequent Itemsets with Constraints 21

Chapter 3 AFOPT: An Adaptive Algorithm for Frequent Itemset Mining 22

3.1 Mining All Frequent Itemsets 23

3.1.1 Conditional database representation 25

3.1.2 A running example 26

3.1.3 Structure size comparison 28

3.1.4 The ascending frequency order 29

3.2 Time Complexity Analysis 30

3.3 Mining Maximal Frequent Itemsets 36

3.3.1 Incorporating pruning techniques 36

3.3.2 Subset checking 38

3.4 A Performance Study 40

3.4.1 Mining all frequent itemsets 41

3.4.2 Mining maximal frequent itemsets 44

3.5 Summary 45

Chapter 4 SSP: Efficient Out-of-core mining of Frequent Itemsets 47

4.1 The Framework 48

4.2 The SSP-naive Algorithm 49

4.3 The SSP-static Algorithm 54

4.4 The SSP-dynamic Algorithm 59

4.5 A Performance Study 63

4.5.1 Varying the minimum support threshold 64

4.5.2 Varying the dataset generating parameters 67

4.5.3 Varying the size of the memory 69

4.6 Summary 70

Chapter 5 CFP-tree: Storing and Querying Frequent Itemsets 71

5.1 The CFP-tree Structure 71

5.1.1 Storing all frequent itemsets 71

5.1.2 Storing frequent closed itemsets 74

5.2 CFP-tree Construction 76

5.2.1 Computing and storing all frequent itemsets 76

5.2.2 Computing and storing frequent closed itemsets 77

5.3 Query Processing 80

5.3.1 The properties of the CFP-tree structure 80

5.3.2 Queries with minimum support constraints 81
5.3.3 Queries with item constraints 83
5.3.4 Queries with both constraints 84
5.3.5 Other types of queries 85
5.4 Incremental Maintenance 86
5.4.1 Maintaining the negative border of a CFP+-tree 87
5.4.2 Maintaining a complete set of frequent itemsets 88
5.4.3 Maintaining frequent closed itemsets 90
5.5 A Performance Study 92
5.5.1 Compactness of the CFP-tree structure 93
5.5.2 Construction time 94
5.5.3 Query processing cost 96
5.5.4 Incremental update cost 99
5.6 Summary 104

Chapter 6 Frequent Itemset Class: Facilitating Online Association Rule Mining 105
6.1 Frequent Itemset Class 106
6.2 Mining Frequent Itemset Classes 110
6.2.1 Computing frequent itemset classes 110
6.2.2 Maintaining frequent itemset classes 111
6.3 Generating Class Association Rules 111
6.3.1 Storing and querying frequent itemset classes 112
6.3.2 Nested loop join 113
6.3.3 Block nested loop join 114
6.4 A Performance Study 115
6.4.1 The reduction in the number of rules 116
6.4.2 Computation time of frequent itemset classes 117
6.4.3 Rule generation time 118
6.5 Summary 120

Chapter 7 Conclusion 122
7.1 Summary of The Thesis 122
7.2 Future Research Directions 124

References 125

LIST OF FIGURES

1.1	Transaction database and frequent itemsets	2
2.1	Search space tree of the frequent itemset mining problem	9
3.1	Conditional database representation	26
3.2	The mining process of the AFOPT algorithm	27
3.3	Pattern length distribution	41
3.4	Mining all frequent itemsets	42
3.5	Mining maximal frequent itemsets	44
4.1	The mining process of SSP-naive	52
4.2	Differential matrix	55
4.3	The mining process of SSP-static	59
4.4	The mining process of SSP-dynamic	61
4.5	Varying the minimum support threshold: running time	65
4.6	Varying the minimum support threshold: I/O cost	66
4.7	Varying dataset generating parameters	69
4.8	Verying memory size	69
5.1	CFP-tree: storing all frequent itemsets	72
5.2	CFP-tree: storing frequent closed itemsets	74
5.3	Two-layer hash map	79
5.4	CFP+-tree with negative border	88
5.5	Compactness of the CFP-tree structure	93
5.6	CFP-tree construction time	95
5.7	Queries with minimum support constraints	99
5.8	Queries with item constraints	100
5.9	Inserting new transactions	101
5.10	Deleting old transactions	103
6.1	Cover equivalent classes	107
6.2	Class association rules (min_sup=40%)	109
6.3	Frequent itemset classes computation time	118
6.4	Rule generation time	119

LIST OF TABLES

2.1	Pattern growth algorithms	15
3.1	Comparison of structure size	28
3.2	Comparison of three item search orders (bucket size=0)	30
3.3	Number of recursive calls	35
3.4	Datasets	41
4.1	Conditional database size	49
4.2	Prefix-trie compression ratio	60
4.3	Datasets	64
5.1	Size of CFP-tree w.r.t closed itemsets and all itemsets	75
5.2	Datasets	92
5.3	Number of non-closed itemsets removed by suffix sharing	96
5.4	Pruning power of the two-layer hash map	96
5.5	Size of CFP-tree, pattern set and inverted files	98
6.1	Datasets	116
6.2	Number of class rules and itemset rules	116
6.3	Number of class rules and non-redundant itemset rules	117
6.4	Fixed values for four parameters	119

SUPPORTING EFFICIENT AND SCALABLE FREQUENT PATTERN MINING

by

GUIMEI LIU

Department of Computer Science

The Hong Kong University of Science and Technology

ABSTRACT

With the large amount of data collected in various applications, data mining has become an essential tool for data analysis and decision support. Frequent itemset mining is an important problem in the data mining area with a wide range of applications. In this dissertation, we investigate several techniques to support efficient and scalable frequent itemset mining.

We first identify the key factors of a frequent itemset mining algorithm, and propose an algorithm AFOPT for efficient mining of frequent itemsets. The AFOPT algorithm adopts the pattern growth approach. It uses a new structure, called Ascending Frequency Ordered Prefix-Trie (or AFOPT for short), and two other structures to store conditional databases. An adaptive strategy is employed to choose proper structures according to the density of the conditional databases, which makes the AFOPT algorithm perform well on both sparse and dense databases.

To deal with very large databases with millions of transactions, we propose a Search Space based Partitioning approach SSP for scalable out-of-core mining. The novel idea of SSP is to partition the database based on the search space. Different partitions share data but not frequent itemsets. As such, frequent itemsets

can be mined from each partition without the need to scan the whole database to verify their support. Furthermore, techniques are developed so that the available memory can be utilized during the mining process to minimize disk I/O. Our experiment results show that the proposed algorithms achieve significant performance improvement over existing out-of-core mining algorithms.

Many decision support systems need to support online interactive frequent itemset mining, which is very challenging because frequent itemset mining is a computation intensive repetitive process. We propose a compact disk-based data structure—CFP-tree (Condensed Frequent Pattern tree) for storing precomputed frequent itemsets to support online mining requests. Efficient algorithms for retrieving frequent itemsets and association rules from a CFP-tree, as well as efficient algorithms to construct and update a CFP-tree, are developed. One obstacle to online association rule mining is the undesirably large result size. We introduce the concept of frequent itemset class and the class association rule to reduce the result size without information loss. The benefits of organizing itemsets into classes are two-fold. On the one hand, the output size is reduced and it is easier for users to browse the results. On the other hand, the computation cost is saved because the rule generation cost of the itemsets in the same class is shared. Our performance study demonstrates that with CFP-tree and the concept of frequent itemset class, frequent itemset and association rule mining requests can be responded to promptly.

CHAPTER 1

INTRODUCTION

The essence of data mining is the non-trivial extraction of implicit, previously unknown and potentially useful information from data [78]. Association rule mining is to discover meaningful rules with strong statistical significance from data. It is an important problem in the data mining area and has a wide range of applications. Common examples include customer buying pattern analysis [11], inclusion dependency mining from relational databases [58], Web log mining [17, 87, 98], text mining [40, 59, 27] and biological data mining [38, 22]. Association rule mining can also be integrated with traditional classification and clustering techniques to improve their performance [25, 50, 27].

In this dissertation, we propose several techniques to support efficient and salable association rule mining. This chapter gives some background knowledge on association rule mining and briefly summarizes our research contributions.

1.1 Problem Definition

The association rule mining problem is first introduced by Agrawal et al. [4] in the context of transaction databases. A transaction database is a database containing a set of transactions and each transaction is associated with a transaction id. An example transaction database is shown in Figure 1.1(a) and the example database contains 7 transactions. Let $D = \{t_1, t_2, \cdots, t_N\}$ be a transaction database and $I = \{a_1, a_2, \cdots, a_n\}$ be the set of items appearing in D, where t_i ($i \in [1, N]$) is a transaction and $t_i \subseteq I$. Each subset of I is called an *itemset*. If an itemset contains k items, then the transaction is called a k-itemset. Given a transaction t and an itemset l, if $l \subseteq t$ then we say t *contains* l and l *covers* t.

Definition 1 (Support of an Itemset) *The support of an itemset l in a database D is denoted by $support_D(l)$, and is defined as $support_D(l) = \|\{t|t \in D \text{ and } l \subseteq t\}\|/\|D\|$.*

TID	Transactions
1	a, c, e, f, m, p
2	a, b, f, m, p
3	a, b, d, f, g
4	d, e, f, h, p
5	a, c, d, m, v
6	a, c, h, m, s
7	a, f, m, p, u

(a) database

All Patterns (min_sup = 40%)
c:3, d:3, p:4, f:5, m:5, a:6
cm:3, ca:3, pf:4, pm:3, pa:3, fm:3, fa:4, ma:5
cma:3, pfm:3, pfa:3, pma:3, fma:3
pfma: 3

(b) Frequent Itemsets

Closed Patterns (min_sup=40%)
d:3, f:5, a:6
pf:4, fa:4, ma:5
cma:3
pfma:3

(c) Closed

Maximal Patterns (min_sup=40%)
d:3
cma:3
pfma:3

(d) Maximal

Figure 1.1: Transaction database and frequent itemsets

Definition 2 (Frequent Itemset) *Given a transaction database D and a user specified minimum support threshold min_sup ($min\_sup \in (0, 1]$), if $support_D(l) \geq min\_sup$, then l is called a frequent itemset in D.*

Similarly, we can define the count of an itemset l in a database D as $count_D(l) = \|\{t|t \in D \text{ and } l \subseteq t\}\|$ and a natural number is used as the minimum count threshold. Figure 1.1(b) shows the set of frequent itemsets mined from the transaction database shown in Figure 1.1(a) with minimum support of 40% or minimum count of 3. For brevity, a frequent itemset $\{i_1, i_2, \cdots, i_m\}$ with count n is represented as $i_1 i_2 \cdots i_m : n$.

Definition 3 (Association Rule) *An association rule is of the form r: $l_1 \rightarrow l_2$, where l_1 and l_2 are two itemsets and $l_1 \bigcap l_2 = \phi$. The support of a rule r in a database D is defined as $support_D(r) = support_D(l_1 \bigcup l_2)$. The confidence of rule r is defined as $confidence_D(r) = support_D(l_1 \bigcup l_2)/support_D(l_1)$. Given a transaction database D, a minimum support threshold min_sup and a minimum confidence threshold min_conf, if $support_D(r) \geq min\_sup$ and $confidence_D(r) \geq min\_conf$, then r is called a valid rule in D. Itemset l_1 is called the left hand side (LHS) of r, and l_2 is called the right hand side (RHS) of r.*

Given a transaction database D, a minimum support threshold *min_sup* and a minimum confidence threshold *min_conf*, the task of association rule mining is to find all valid association rules in D. Most of the existing association rule mining algorithms follow a two phase approach:

- frequent itemset mining: find all frequent itemsets with respect to *min_sup* in D.

- association rule generation: generate association rules with confidence $\geq$ min_conf from precomputed frequent itemsets.

It is widely recognized that the first step of association rule mining is the bottleneck. The main challenge is that any subset of I can be frequent in D, and the total number of subsets of I is exponential to the size of I. Existing mining algorithms mainly concentrate on studying the first step. They utilize the *apriori* property (also called the anti-monotone property) to prune the search space, which is stated as follows: *if an itemset is not frequent, then none of its supersets can be frequent.* The apriori property has been widely used in mining other types of patterns, e.g. sequential patterns [6, 85], episodes [55, 57], periodic patterns [33, 31], frequent tree patterns [100] and frequent subgraph patterns [39, 96].

Data collected from some domains, e.g. the biological domain, can be very dense and contain very long frequent itemsets. Any algorithm that produces all frequent itemsets suffers from generating an enormous number of frequent itemsets on such datasets. The reason being that given a frequent k-itemset, all of its subsets are frequent and the number of them is $2^k - 1$. Some researchers have noticed the long pattern problem and suggest mining frequent closed itemsets [69, 102, 74] or maximal frequent itemsets [41, 2, 14, 29]. A frequent itemset is closed if all of its proper supersets are less frequent than it. A frequent itemset is maximal if none of its proper supersets is frequent. They are formally defined as follows.

Definition 4 (Frequent Closed Itemset) *An itemset l is a frequent closed itemset if l is frequent, and there does not exist l' such that $l \subset l'$ and $support(l) = support(l')$.*

Definition 5 (Maximal Frequent Itemset) *An itemset l is a maximal frequent itemset if l is frequent, and there does not exist l' such that $l \subset l'$ and l' is frequent.*

Figure 1.1(c) and Figure 1.1(d) show respectively the set of frequent closed itemsets and the set of maximal frequent itemsets mined from the database shown in Figure 1.1(a) with minimum support of 40%. A maximal frequent itemset must be a frequent closed itemset, but not vice versa. For example, f:5, a:6, pf:4,

fa:4 and ma:5 are frequent closed itemsets, but they are not maximal frequent itemsets because their superset $pfma : 3$ is frequent. If we use FI to denote the complete set of frequent itemsets discovered from a database, FCI to denote the set of frequent closed itemsets and MFI to denote the set of maximal frequent itemsets, we have $MFI \subseteq FCI \subseteq FI$. Both MFI and FCI can be orders of magnitude smaller than FI. FCI is a concise representation of FI without information loss. We can derive the complete set of frequent itemsets from FCI with support information. MFI provides a concise view of the frequent itemsets. From MFI we can also derive the complete set of frequent patterns, but the support information is lost.

Many variations and new types of association rules have been proposed based on the basic form of the association rule. Multi-level association rules [32] and generalized association rules [83] aim to find association rules at multiple concept levels where a concept taxonomy is available. Quantitative association rules [84, 7] not only deal with categorical attributes, but also quantitative attributes. Optimized association rules [26] try to compute optimal ranges for quantitative attributes to maximize support, confidence or other interesting measurements. Cyclic association rules [66] and calendric association rules [76] display regular variation over time. Negative association rules [80] capture the negative connections between itemsets. Fault-tolerant association rules [97, 75] allow a small fraction of mismatch between itemsets and transactions. Inter-transaction association rules [91] break the boundary of transactions, and the itemsets can contain items from different transactions. Top-k frequent itemset mining [36] reports the k most frequent itemsets, therefore mining top-k frequent itemsets does not rely on a minimum support threshold. Besides minimum support and minimum confidence, many other measurements have been used to measure the interestingness of association rules [88]. In this dissertation, we focus on the basic form of the association rule mining problem.

1.2 Motivations and Contributions

The association rule mining problem has drawn much attention over the past decade. Tens of or even hundreds of algorithms have been proposed for mining frequent itemsets. However, few of them perform well on both sparse and

dense databases. The candidate generate-and-test algorithms, e.g. the Apriori algorithm [5], repeatedly scan the database to check a large number of candidate itemsets by subset matching. In the worst case, the number of database scans is equal to the maximal length of the frequent itemsets. Furthermore, subset matching is a costly operation. Therefore, the candidate generate-and-test approach suffers from both enormous I/O cost and CPU cost on very large databases with prolific frequent itemsets [35]. The pattern growth algorithms generally work better than the candidate generate-and-test algorithms, but they also have some shortcomings. The FP-growth algorithm [35] does not work well on sparse databases due to the high construction and traversal cost of the FP-tree structure. The H-Mine algorithm [73] tries to overcome the drawbacks of the FP-growth algorithm. It uses a hyper-structure to store conditional databases, in which two pointers are maintained for each item. Therefore, the size of a hyper-structure is roughly 3 times as large as the original database. In this thesis, we propose an adaptive algorithm—AFOPT for efficient mining of frequent itemsets on both sparse and dense databases [51]. The AFOPT algorithm has a new structure—Ascending Frequency Ordered Prefix-Trie (or AFOPT for short) and two other structures to store conditional databases. It adaptively chooses proper structures according to the density of the conditional databases, which makes it consume much less space than other pattern growth algorithms and perform well on both sparse and dense databases.

A recent comparative study on frequent itemset mining implementations (FIMI'03 [28]) concludes that few existing algorithms can scale up to very large databases with millions of transactions. The candidate generate-and-test algorithms need to scan the database multiple times. The pattern growth approach can reduce the number of database scans to two if all the conditional databases constructed from the original database can fit into the main memory. When the conditional databases are too large to fit into the main memory, writing and reading conditional databases can incur tremendous I/O cost if handled improperly. Existing pattern growth algorithms concentrate mainly on optimizing in-memory performance. They either focus on compressing the conditional databases as much as possible such that they can fit into the memory, or try to reduce the construction and traversal cost of the conditional databases. None of them really deals with the situation when the conditional databases are too large to be held in the

memory. In this thesis, we propose a *search space based partitioning* approach SSP that achieves scalable frequent itemset mining over millions of transactions. SSP is specially designed for out-of-core mining. It assumes that the memory is not sufficient to accommodate even compressed information such as frequent pattern trees. The novel idea of SSP is to partition the database based on the search space. A transaction can belong to multiple partitions, but a frequent itemset can belong to only one partition. As such, a local frequent itemset is also a globally frequent itemset. Three algorithms that exploit data overlap among partitions to reduce I/O cost are proposed. Our experiment results show that with careful management, I/O cost can be reduced significantly, which makes SSP a true scalable approach for mining frequent itemsets from very large databases with millions of transactions.

Frequent itemset mining is a time-consuming process due to its intrinsic characteristics: both I/O intensive and CPU intensive. Furthermore, association rule mining requires predefined thresholds on support and confidence. Different databases and applications often require different thresholds. Even for the same database, different applications may have different requirements. In many real applications, it is hard for a user to guess a-priori how many rules might satisfy a given minimum support and confidence, and there are often no guidelines for choosing the proper thresholds, which makes association rule mining a repetitive process to find appropriate thresholds for a given database and application. Decision support systems that rely on association rule mining often need to support online interactive mining. As in data warehouses where summaries of data along various dimensions and their combinations are precomputed and stored (materialized) so that OLAP queries can be answered quickly without the need for scanning the original data, precomputing frequent itemsets seems an attractive solution to support online association rule based applications. For this purpose, we propose a disk-based data structure, called Condensed Frequent Pattern tree (or CFP-tree for short), to store precomputed frequent itemsets on a disk [52]. Online mining requests are supported by retrieving frequent itemsets and association rules from a CFP-tree.

The main contributions of the thesis can be summarized as follows:

- We propose an algorithm AFOPT for efficient mining of frequent itemsets,

which uses a new structure AFOPT and an adaptive strategy to store conditional databases. We compare various structures for storing conditional databases and investigate several orders for exploring the search space. The results show that the AFOPT structure combined with the ascending frequency order is capable of minimizing both the total number of conditional databases and the size of the conditional databases, thus requiring less space and time than other structures.

- We propose a salable search space based partitioning approach SSP for mining frequent itemsets from very large databases with millions of transactions. To the best of our knowledge, SSP is the first pattern growth algorithm specially designed for out-of-core mining. It achieves significant performance improvement over existing out-of-core algorithms.

- As frequent itemset mining is a time consuming process, algorithms that adopt the "mining on demand" strategy can hardly support online interactive mining. We propose a compact disk-based data structure—CFP-tree for storing precomputed frequent itemsets to support online interactive mining. With a CFP-tree, the time required for answering mining requests is independent of the size of the base database and the number of precomputed frequent itemsets, and it is in fact dependent of the size of the output, which is a favorable feature for OLAP applications. We have also developed efficient algorithms to incrementally update the precomputed frequent (closed) itemsets stored in a CFP-tree.

- A complete set of frequent itemsets often contain lots of redundant information, which sometimes makes the association rule mining task impossible because of the large result size. We proposed the concept of frequent itemset class and class association rules to reduce result size. The number of class association rules can be even significantly smaller than the number of non-redundant itemset association rules. The set of traditional itemset association rules can be fully recovered from class association rules at little cost.

1.3 Organization of The Thesis

The rest of the thesis is organized as follows.

Chapter 2 surveys previous work on frequent itemset mining and association rule mining.

In Chapter 3, we describe an adaptive algorithm AFOPT for efficient mining of frequent itemsets. The efficiency of the algorithm is validated by both analysis of time complexity of the algorithm and the experiment results. At the end of this chapter, we discuss how to incorporate several pruning techniques into the AFOPT algorithm to mine maximal frequent itemsets.

In Chapter 4, we present the SSP approach for out-of-core mining of frequent itemsets from very large databases. The central idea of the SSP approach is to partition the database according to the search space, and utilize data sharing among partitions to reduce I/O cost. Three algorithms that implement the SSP approach are presented.

The CFP-tree structure is proposed in Chapter 5. A set of CFP-tree related algorithms, including very fast tree construction algorithms, efficient algorithms for retrieving frequent itemsets satisfying user specified constraints from a CFP-tree and an incremental update algorithm, are developed and evaluated.

In Chapter 6, we introduce the concept of frequent itemset class and class association rules to facilitate online association rule mining. We present an efficient algorithm to compute frequent itemset classes, and develop online algorithms to generate class association rules satisfying user-specified constraints from precomputed frequent itemset classes.

Chapter 7 concludes the thesis and discusses several directions for future research.

CHAPTER 2

RELATED WORK

The association rule mining problem has been extensively studied from various aspect over the past decade. In this chapter, we survey previous work on frequent itemset mining and association rule mining, with an emphasis on frequent itemset mining algorithms.

2.1 Frequent Itemset Mining Algorithms

Most association rule mining algorithms adopt the two-phase approach and focus on the frequent itemset mining phase. A few algorithms mine association rules directly [20, 94]. They put some constraints on association rules to make direct search feasible. Cohen et al. [20] propose an algorithm to mine rules with confidence extremely close to 100%. The algorithm proposed by G. Webb [94] requires that the dataset must be resident in the memory. In the rest of this section, we concentrate on discussing frequent itemset mining algorithms.

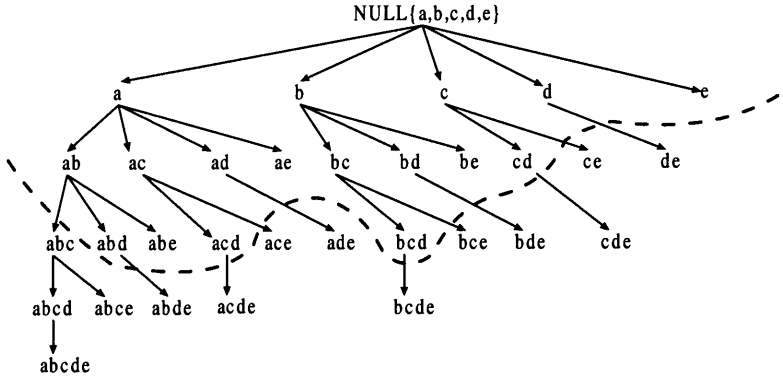


Figure 2.1: Search space tree of the frequent itemset mining problem

Given a set of items I and a transaction database D, any subsets of I can be frequent in D. They form the search space of the frequent itemset mining problem. The search space can be represented using set enumeration tree [77, 41, 2, 14, 29]. An example search space tree for $I = \{a, b, c, d, e\}$ is shown in

Figure 2.1. The items in I are sorted into lexicographic order. The root of the tree represents the empty set. Each node at level k represents a k-itemset. For every itemset l in the tree, only items after the last item of l can be appended to l to form a longer itemset. Those set of items are called *candidate extensions* of l. For example, item d is a candidate extension of ac, but b is not a candidate extension of ac because b is before c. If item i is a candidate extension of l and $l \bigcup \{i\}$ is frequent, then i is called a *frequent extension* of l.

The size of the search space is exponential to the number of items in I. One main goal of an efficient frequent itemset mining algorithm is to count support for as few itemsets as possible in the search space. Most existing algorithms use the *apriori* property to prune the search space, which is stated as follows: *if an itemset is not frequent, then none of its supersets can be frequent.* Existing algorithms can be classified into four categories: the candidate generate-and-test approach [4, 56, 5, 68, 12, 47], the filter-and-refine approach [79, 90, 49, 45], the vertical mining approach [103, 81, 101] and the pattern growth approach [35, 34, 1, 73, 54, 53].

2.1.1 The candidate generate-and-test approach

The candidate generate-and-test approach uses the breadth-first-order to explore the search space. Frequent itemset mining can be viewed as a set containment join between the transaction database and the search space of the frequent itemset mining problem. The candidate generate-and-test approach essentially uses block nested loop join, i.e. the search space is the inner relation and it is divided into blocks according to the itemset length. Different from a simple block nested loop join, in the candidate generate-and-test approach the output of the previous pass is used as seeds to generate the next block. For example, in the k-th pass of the Apriori algorithm [5], the transaction database and the candidate k-itemsets are joined to generate frequent k-itemsets. The frequent k-itemsets are then used to generate the next block—candidate $(k+1)$-itemsets. The DHP algorithm [68] makes improvements based on the Apriori algorithm. It trims transactions using frequent itemsets after each iteration. It also prunes candidate itemsets using hashing. The OSSM algorithm [47] prunes candidate itemsets using a segmentation technique. The PASCAL algorithm [10] utilizes the redundancy

in a complete set of frequent itemsets. It first mines representative itemsets, called key patterns, from the database, then uses a counting inference technique to compute all frequent itemsets.

Given the large amount of main memory available nowadays, it is a waste of memory to put only itemsets of a single length into the memory. It is desirable to fully utilize the available memory by putting some longer and possibly frequent itemsets into the memory at an earlier stage to reduce the number of database scans. The first frequent itemset mining algorithm AIS [4] tries to count support for longer itemsets earlier by estimating the frequency of longer itemsets using the output of the current pass. However, AIS does not fully utilize the pruning power of the apriori property, thus many unnecessary candidate itemsets are generated and tested. The DIC algorithm [12] starts counting the support of an itemset shortly after all the subsets of that itemset are determined to be frequent rather than wait until the next pass. However, DIC algorithm still cannot guarantee the full utilization of the main memory. In fact, all the candidate generate-and-test algorithms face a trade-off: On the one hand, the memory is not fully utilized and it is desirable to put as many candidate itemsets as possible into the memory to reduce the number of database scans. On the other hand, set containment testing is a costly operation. Putting itemsets into the memory at an earlier stage increases the risk of counting support for many unnecessary candidate itemsets.

2.1.2 The filter-and-refine approach

The filter-and-refine approach reduces the number of database scans to two. The basic idea is to first generate an approximate set of frequent itemsets by scanning the database once (the filtering step), and then produce the exact set of frequent itemsets in a second database scan (the refining step).

Existing filter-and-refine algorithms differ mainly in the filtering step. The *Partition* algorithm [79] is based on the observation that if we divide a database into several disjoint partitions, then a frequent itemset must be frequent in at least one partition. The Partition algorithm divides a database into disjoint partitions such that each partition can be mined in the memory. The itemsets frequent in at least one partition form the set of candidate itemsets on the whole database. This algorithm has several drawbacks: (1) The frequent itemset mining task is not

only I/O intensive, but also CPU intensive. Mining the partitions independently means duplicating the computation cost several times. (2) When partitioning a database, it is very hard to accurately estimate the amount of space needed for mining one partition. (3) Data skew can cause the algorithm to generate many false candidate itemsets. Lin et al. [49] propose an algorithm AS_CPA to handle skewed data by using the cumulative counts of the candidate itemsets. The sampling algorithm proposed by Toivonen [90] picks a random sample to compute frequent itemsets along with a negative border in the filtering phase. Lan et al. [45] use an indexing structure BBS (Bit-sliced Bloom-filtered Signature file) to facilitate mining. A BBS structure is constructed from the original database, and the candidate itemsets are obtained by scanning the BBS structure instead of the database. One drawback of the algorithm is that the size of the BBS structure is critical for its performance.

The filter-and-refine algorithms usually need to generate and test more candidate itemsets than the candidate generate-and-test algorithms. They all assume that the number of candidate itemsets is small enough to fit into the main memory so that the refining phase requires only one database scan, which may not be true on very large and/or dense datasets.

2.1.3 The vertical mining approach

The vertical mining approach maintains a tid list or a tid bitmap for each frequent itemset. Candidate itemset testing is performed by tid list/bitmap intersection, which is more efficient than subset matching. The main drawback of the vertical mining approach is that it needs to maintain a large number of tid lists/bitmaps, which prevents it from scaling well with respect to the number of transactions. Several techniques have been proposed to reduce the size of the tid lists/bitmaps. For example, the VIPER algorithm [81] uses a compression technique, and the dEclat algorithm [101] uses diffsets to reduce the size of tid lists/bitmaps. It is difficult to handle the case when all tid lists/bitmaps of the frequent items cannot be held in the memory. The reason being that to mine all the frequent itemsets containing a frequent item, the tid list/bitmap of that item has to be intersected with all the other tid lists/bitmaps. Existing vertical mining algorithms usually change to horizontal mining if all the tid lists/bitmaps cannot be held in the

memory.

2.1.4 The pattern growth approach

The pattern growth approach uses depth-first order to probe the search space. It adopts the divide-and-conquer methodology. The search space is divided into disjoint sub search spaces. The database is divided into partitions according to the sub search spaces. For example, suppose there are 5 items $\{a, b, c, d, e\}$ frequent in a transaction database D, and the search space is shown in Figure 2.1. The items are sorted lexicographically. The search space is divided into 5 disjoint sub search spaces: (1) itemsets starting with a; (2) itemsets starting with b, i.e. itemsets containing b but not containing a; (3) itemsets starting with c; (4) itemsets starting with d; (5) itemsets containing only e. Based on these five sub search spaces, the database is divided into 5 partitions. Each partition is called a *conditional database*. The conditional database of an item i, denoted as D_i, contains all the transactions containing item i. Items before i are eliminated from each transaction in D_i. All the itemsets starting with item a_i can be mined from D_{a_i} without accessing any other information. Each conditional database is divided recursively following the same procedure. The pattern growth approach not only reduces the number of database scans, but also avoids the costly subset matching operation.

Two basic operations in the pattern growth approach are *support counting* over conditional databases and *new conditional database construction*. Therefore, the number of conditional databases constructed during the mining process and the mining cost of individual conditional databases are key factors that affect the performance of a pattern growth algorithm. They mainly depend on four factors: item search order, conditional database representation (tree-based or array-based structures), conditional database construction strategy (physical or pseudo) and conditional database traversal strategy (top-down or bottom-up).

Item Search Order. When we divide the search space, all frequent items are sorted according to some order. This order is called *item search order*. The sub search space of an item contains all the items after the item in the item search order but no items before the item. Two typical item search orders are proposed in the literature: static lexicographic order and dynamic ascending frequency

order. The static lexicographic order is to sort items lexicographically. It is a fixed order—all the sub search spaces use the same order. The tree-projection algorithm [1] and the H-Mine algorithm[73] adopt this order. The dynamic ascending frequency order reorders frequent items in every conditional database into ascending frequency order. The most infrequent item is the first item, and all the other items are its candidate extensions. The most frequent item is the last item and it has no candidate extensions. The FP-growth algorithm [35], the AFOPT algorithm [53] and most of the maximal frequent itemset mining algorithms [41, 2, 14, 29] adopt this order. The rationale behind the ascending frequency order is that the size of conditional databases shrinks quickly in subsequent levels. The drawback is that it incurs additional sorting cost, e.g. every transaction in a conditional database has to be sorted according to the ascending frequency order.

Conditional Database Representation. The traversal cost and construction cost of a conditional database depends heavily on the representation format of the conditional database. Different data structures have been proposed to store conditional databases, e.g. tree-based structures such as the FP-tree structure [35] and the AFOPT structure [53], and array-based structures such as the hyper-structure [73]. Tree-based structures are capable of reducing traversal cost because duplicated transactions are merged and different transactions share the storage of their prefixes. However, they incur high construction cost especially when the dataset is sparse and large. Array-based structures incur little construction cost but they need much more traversal cost. It is a trade-off to choose between tree-based structures and array-based structures. In general, tree-based structures are suitable for dense databases, and array-based structures are suitable for sparse databases.

Conditional Database Construction Strategy Constructing every conditional database physically can be expensive especially when successive conditional databases do not shrink much. An alternative is to pseudo-construct them, i.e., using pointers pointing to transactions in upper level conditional databases. However, pseudo-construction cannot reduce traversal cost as effectively as physical construction.

Tree Traversal Strategy The traversal cost of a tree is minimal using the

Algorithms	Item Search Order	CDB Format	CDB Construction	Tree Traversal
Tree-Projection [1]	static lexicographic	array	adaptive	-
FP-growth [35]	dynamic frequency	FP-tree	physical	bottom-up
H-mine [73]	static lexicographic	hyper-structure	pseudo	-
OP [54]	adaptive	adaptive	adaptive	bottom-up
PP-mine [95]	static lexicographic	PP-tree	pseudo	top-down
AFOPT [51]	dynamic frequency	adaptive	physical	top-down
CLOSET+ [93]	dynamic frequency	FP-tree	adaptive	adaptive

Table 2.1: Pattern growth algorithms

top-down traversal strategy. The FP-growth algorithm [35] uses the ascending frequency order to explore the search space, while the FP-tree structure is constructed according to descending frequency order. Hence the FP-tree structure has to be traversed using the bottom-up strategy. As a result, an FP-tree has to maintain a parent pointer and a node-link pointer at each node for bottom-up traversal, which increases the construction cost of the tree. We propose an algorithm—AFOPT [51], which uses the ascending frequency order for both search space exploration and prefix-tree construction, so it uses the top-down traversal strategy to traverse the tree and it does not need to maintain additional pointers at each node.

Existing pattern growth algorithms differ mainly in the four dimensions aforementioned as listed in Table 2.1. The FP-growth algorithm [35] uses a compact structure FP-tree to store conditional databases. FP-growth is not efficient on sparse databases where prefix sharing is not common. An array-based structure, called hyper-structure [73], is proposed to deal with sparse databases. Algorithms OP [54] and AFOPT [51] adaptively choose between a tree-based structure and an array-based structure according to the density of the conditional databases. The tree-projection algorithm [1] physically constructs a conditional database only when the size of the conditional database is below a specific ratio of its ancestor's conditional database. Algorithm CLOSET+ [93] is an adaptive algorithm for mining frequent closed itemsets. It uses the FP-tree structure to store conditional databases. It concentrates on studying the conditional database construction strategy and the tree traversal strategy. More specifically, CLOSET+ uses the physical construction strategy and the bottom-up traversal strategy together on dense datasets, and uses the pseudo construction strategy and the top-down traversal strategy on sparse datasets.

Most existing pattern growth algorithms concentrate on optimizing in-memory performance. They either focus on compressing the conditional databases as much as possible such that they can fit into the memory, or try to reduce the construction and traversal cost of the conditional databases. None of them really deals with the situation where the conditional databases are too large to fit into the memory. A recent comprehensive study (FIMI) [28] compares various frequent itemset mining implementations. It concludes that few existing algorithms can scale up to very large databases with millions of transactions. Among all the implementations, the kDCI algorithm [64] shows the best scalability. The kDCI algorithm is an improvement of the DCI algorithm [65]. It incorporates the counting inference technique proposed by Basted et al. [10] into the DCI algorithm. The good scalability of the kDCI algorithm and the DCI algorithm comes from the hybrid support counting strategy they adopted. The two algorithms are level-wise algorithms with two phases. In the first phase, the two algorithms count support by subset matching, accompanied by effective pruning of the dataset. The dataset pruning technique is inspired by the DHP algorithm [68]. It trims the transaction database as mining progresses. In the second phase, support counting is performed by tid bit vector intersection. In this dissertation, we propose a scalable pattern growth approach and compare it with the kDCI algorithm.

2.2 Removing Redundancy from Mining Results

The number of frequent itemsets can be undesirably large on large and/or dense datasets when the minimum support is low. It has been observed that a complete set of frequent itemsets contains too much redundant information. It is necessary to remove the redundancy to make the result manageable to end users. The concepts of frequent closed itemset and maximal frequent itemset have been proposed to reduce result size. Both of them can be substantially smaller than the corresponding complete set of frequent itemsets.

2.2.1 Mining frequent closed itemsets

An itemset is closed if all of its proper supersets are less frequent than it. The basic idea behind frequent closed itemset is to use a single itemset (the longest

one) to represent all the itemsets appearing in the same set of transactions. Some other concepts have been proposed to reduce result size based on the same idea. The disjunction-free sets introduced by Bykowski et al. [15], the generators introduced by Bastide et al. [9] and the key patterns in [10] all refer to the same type of itemsets—the minimal itemsets among all the itemsets appearing in the same set of transactions.

The concept of frequent closed itemset is first proposed by Pasquie et al. [69] and an apriori-based algorithm A-Close is proposed to mine frequent closed itemsets. The CHARM algorithm [102] adopts the vertical mining technique. The above two algorithms first mine the generators, then compute the closures of the generators to get frequent closed itemsets. Algorithms CLOSET [74] and CLOSET+ [93] are based on the FP-growth algorithm. They use a FP-tree like structure to store frequent closed itemsets for subset checking. The CLOSET+ algorithm considers the nature of the database and uses an adaptive mining strategy. Pan et al. [67] consider the situation where the datasets contain a large number of columns but a small number of rows, and propose the CARPENTER algorithm, which performs a row-wise enumeration instead of column-wise enumeration adopted by previous work. Pei et al. [70] propose a more restrictive concept, condensed frequent pattern base, to further reduce result size. An itemset is a condensed frequent pattern base if all of its proper supersets are significantly less frequent than it.

A set of frequent closed itemsets is a concise representation of the corresponding complete set of frequent itemsets without information loss. The problem of removing redundancies while preserving semantic is also studied in other areas. The concepts of *closed itemset* can be regarded as applying the Galois closure operators and formal concept analysis [16] to the itemset lattice. Similar concepts and techniques have been used in data cube computation to reduce cube size, e.g. Dwarf Cube [82] and Quotient Cube [44].

2.2.2 Mining non-redundant association rules

Different definitions have been given to non-redundant association rules. The most appropriate one is the one defined by Bastide et al. [9]. According to their definition, an association rule $r : l_1 \rightarrow l_2$ is non-redundant if there does

not exist another rule $r' : l'_1 \rightarrow l'_2$ such that r and r' have the same support and confidence, $l'_1 \subseteq l_1$ and $l'_2 \supseteq l_2$. Both frequent closed itemsets and generators are needed to mine such non-redundant rules. Another definition of non-redundant association rule is given by M. J. Zaki [99], which requires both the left hand side and the right hand side of a non-redundant rule to be minimal. To mine such rules, generators alone are sufficient. Aggarwal et al. [3] gave a more restrictive definition of non-redundant rule. They removed the support and the confidence constraints from the definition given by Bastide et al. [9]. According to their definition, a rule is non-redundant if its left hand side is minimal and its right hand side is maximal. This definition further reduces the number of rules at the cost of information loss.

2.2.3 Mining maximal frequent itemsets

An itemset is maximal if none of its proper supersets is frequent. Mining maximal frequent itemsets can be viewed as finding a border in the search space such that all the itemsets below the border are infrequent and all the itemsets above the border are frequent. The support information of the itemsets are lost when mining maximal frequent itemsets.

Several algorithms have been proposed to mine maximal frequent itemsets. The MaxMiner algorithm [41] uses an apriori-based approach. It proposes the lookahead technique and the dynamic reordering technique. These two techniques have been proved to be the most effective techniques to prune non-maximal itemsets. The Pincer-Search algorithm [48] utilizes a combination of the bottom-up and top-down search strategy. The top-down strategy is used to find long patterns to prune short patterns. The DepthProjection algorithm [2] adopts the depth-first order to explore the search space. The depth-first order is more suitable for mining maximal itemsets than the breadth-first-search order. The reason being that long patterns are mined first in the depth-first order so that they can be used to prune shorter patterns. The MAFIA algorithm [14] adopts the vertical mining technique. Another algorithm using vertical mining technique is the GenMax algorithm [29]. It employs a progressive focussing technique to reduce the cost for maximality checking.

In this dissertation, we propose a compact structure CFP-tree to store frequent

itemsets [52], which utilizes the redundancy in frequent itemsets to save space. A complete set of frequent itemsets can be efficiently recovered from a CFP-tree without any overhead. Considerable cost is required to derive all frequent itemsets from other concise representations, e.g. frequent closed itemsets and generators. Frequent closed itemsets, generators and frequent maximal itemsets can be efficiently generated from a CFP-tree.

2.3 Incremental Frequent Itemset Mining

Incremental mining of frequent itemsets has been studied in several papers [18, 19, 24, 89, 8]. Algorithm FUP, MLUP, FUP_2 and UWEP are Apriori-based algorithms. They use the original frequent itemsets to reduce the number of candidate itemsets. The FUP algorithm [18] only deals with insertion. The FUP_2 algorithm [19] can handle insertion and deletion as well as transaction update. The UWEP algorithm [8] uses a lookahead technique to remove those itemsets that are no longer frequent. The MLUP algorithm [23] is an algorithm used to update multi-level association rules. The above algorithms mainly save CPU cost but not I/O cost. They still need to scan the original database multiple times. To reduce I/O cost, the concept of *negative border* of frequent itemsets is used to decide when to scan the original database [89]. The update algorithm requires a full scan of the original database only if the negative border expands.

Lee et al. [46] address the problem of when to update frequent itemsets. Their algorithm estimates the difference between the association rules before and after the update using a sampling technique. The update is performed only if the difference is significant.

All existing algorithms on incremental update of frequent itemsets are maintaining a complete set of frequent itemsets. In Chapter 5, we discuss how to update a CFP-tree storing all frequent itemsets or storing only frequent closed itemsets. Updating frequent closed itemsets is more challenging than maintaining a complete set of frequent itemsets. The reason being that when updating frequent closed itemsets, we not only need to maintain the count of the itemsets, but also need to maintain the closure of the itemsets.

2.4 Online Interactive Association Rule Mining

Mining frequent itemsets from scratch requires a long processing time, which is unsuitable for interactive mining. A common approach to supporting online interactive mining is to use previous mining results to improve the efficiency of future mining requests. Nag et al. [62] propose the use of a knowledge cache to reduce response time. They envisage a business environment where a number of users are simultaneously issuing association rule queries on a data warehouse. Their system takes advantages of the large number of concurrent users and the sequence of queries submitted by each user by reusing the results of prior computation to answer subsequent queries. The authors also show that caching and precomputation can benefit each other, and the best performance is obtained by combining these two techniques together. Morzy et al. [60] propose the concept of materialized data mining view to support online interactive mining. Materialized data mining views store selected patterns discovered in a portion of a database and are periodically refreshed. Cong et al. [21] introduce the concept of tree boundary (similar to negative border) to summarize the useful information available from previous mining to facilitate future mining.

Different structures have been used to store and query a large collection of frequent itemsets for future use. Morzy et al. [61] propose a group bitmap index structure for retrieving association rules stored in a relational database. Aggarwal et al. [3] use a graph structure to store frequent itemsets to support online association rule mining. The graph structure contains many pointers and may not be very disk-friendly. Tuzhilin et al. [92] propose a rule query language Rule-QL for querying multiple rulebases and several efficient query evaluation techniques for Rule-QL. They use B+ trees to index the support and the confidence of the rules and use inverted files for subset matching.

In this dissertation, we adopt the precomputation strategy accompanied by an efficient incremental update algorithm to support online interactive mining. We propose a compact and disk-friendly data structure—CFP-tree to manage the precomputed frequent itemsets. Frequent itemsets satisfying user-specified constraints can be efficiently retrieved from a CFP-tree.

2.5 Mining Frequent Itemsets with Constraints

In many real applications, users are often interested in a subset of frequent itemsets or association rules instead of the entire set of frequent itemsets or association rules. It is desirable to allow users to specify their constraints and integrate the constraints into the mining process to save the mining cost. Srikant et al. [86] consider item constraints that are boolean expressions over the presence and absence of items. Ng et al. [63] study a broader set of constraints. They introduce and analyze two properties of the constraints (anti-monotone and succinct) that are critical for pruning, and categorize various constraints into four categories according the two properties. An apriori based algorithm CAP is developed to achieve a maximized degree of pruning. Lakshmanan et al. [42] extend the CAP algorithm to handle dynamic changes to constraints. The same group of authors also address how to integrate 2-variable constraints into the mining process [43]. Pei et al. [71, 72] identify another type of constraints—convertible constraints and integrate the constraints into the pattern growth algorithm FP-growth. Monotone constraints are studied in [30]. The DualMiner algorithm [13] uses both monotone and anti-monotone constraints to prune search space from both the upper border and the lower border.

The focus of the above papers is to minimize the mining cost by utilizing the pruning power of the constraints specified by users. In the scenario considered in this dissertation, all the frequent itemsets have been materialized and stored in a CFP-tree. Our focus is to utilize the pruning power of the constraints to minimize the traversal cost of a CFP-tree. Many techniques discussed in the above papers can be adopted in our scenario.

CHAPTER 3

AFOPT: AN ADAPTIVE ALGORITHM FOR FREQUENT ITEMSET MINING

The FP-growth algorithm [35] is the first pattern growth algorithm on mining frequent itemsets. It is orders of magnitude faster than the candidate generate-and-test algorithms. Several algorithms have been proposed to make improvements based on the FP-growth algorithm, e.g. H-Mine [73] and OP [54]. However, all the algorithms that use the FP-tree structure suffer from several drawbacks: (1) The FP-tree structure is a very complex structure. It maintains several pointers at each node: a child pointer, a sibling pointer, a parent pointer and a node-link pointer, which increases the space overhead and the construction cost of the structure. (2) The FP-tree structure is traversed bottom-up along node-links. Using this traversal strategy, a node is visited multiple times. The times that a node is visited equals the number of descendants of the node. Therefore, the bottom-up traversal strategy is not efficient for a tree structure.

In this chapter, we propose to use a simpler but still compact structure, called AFOPT (Ascending Frequency Ordered Prefix-Trie), to store conditional databases. The structure is traversed using the top-down strategy, so each node is visited exactly once. We develop an efficient pattern growth algorithm AFOPT that utilizes the proposed structure and two other structures to store conditional databases. The distinct features of the AFOPT algorithm include: (1) It adaptively chooses appropriate structures to store conditional databases according to the density of the conditional databases. More specifically, AFOPT uses an array structure for sparse conditional databases, uses the AFOPT structure for dense conditional databases and uses the bucket counting technique for extremely dense conditional databases. This adaptive strategy makes the AFOPT algorithm consume much less space than existing pattern growth algorithms. (2) It adopts the dynamic ascending frequency order to explore the search space. Combined with the AFOPT structure, the ascending frequency order is capable of minimizing both the total number of conditional databases and the size of individual conditional databases. The distinct features of the AFOPT algorithm make it very

suitable for mining maximal frequent itemsets. At the end of this chapter, we extend the AFOPT algorithm to mine maximal frequent itemsets by incorporating several pruning techniques.

3.1 Mining All Frequent Itemsets

Given a transactional database D and a minimum support threshold, the AFOPT algorithm scans the original database twice to mine all frequent itemsets. In the first scan, all the frequent items in D are counted and sorted into ascending frequency order, denoted as $F = \{i_1, i_2, \cdots, i_m\}$. We perform another database scan to construct a *conditional database* for each $i_j \in F$, denoted as D_{i_j}. During the second scan, infrequent items in each transaction t are removed and the remaining items are sorted according to their orders in F. Transaction t is put into D_{i_j} if the first item of t after sorting is i_j. The conditional databases contain the complete information for mining frequent itemsets. The remaining mining is performed on the conditional databases only. There is no need to access the original database.

We use a *header table* to maintain the set of frequent items in a database and their conditional databases. The frequent items are sorted into ascending frequency order in the header table. The conditional databases are processed from left to right according to their orders in the header table. We first perform mining on D_{i_1} to mine all the itemsets containing i_1. Mining individual conditional databases follows the same procedure as mining the original database. After the mining on D_{i_1} is finished, D_{i_1} can be discarded. since D_{i_1} also contains other items, the transactions in D_{i_1} should be inserted into the remaining conditional databases. Given a transaction t in D_{i_1}, suppose the next item after i_1 in t is i_j, then t is inserted into D_{i_j}. This step is called the push-right step. Sorting frequent items into ascending frequency order ensures that a small conditional database is pushed right every time. The pseudo-code of the AFOPT algorithm is shown in Algorithm 1. When the algorithm is first called, $l = \phi$ and D_l is the original database.

There are three steps when mining an itemset l's conditional database: the counting step, the construction step and the push-right step. In the counting step, the support of all the items in D_l are counted. The output of this step is

Algorithm 1 AFOPT Algorithm

Input:
l is a frequent itemset
D_l is the conditional database of l
min_sup is the minimum support threshold;

Description:
1: Scan D_l to count frequent items and sort them into ascending frequency order, F=$\{i_1, i_2, \cdots, i_n\}$;
2: **for all** item $i \in F$ **do**
3: $D_{l \bigcup \{i\}} = \phi$;
4: **for all** transaction $t \in D_l$ **do**
5: $t = t \bigcap F$;
6: Sort items in t according to their orders in F;
7: Let i be the first item of t, insert t into $D_{l \bigcup \{i\}}$.
8: **for all** item $i \in F$, from i_1 to i_n **do**
9: Output $s = l \bigcup \{i\}$;
10: AFOPT(s, D_s, *min_sup*);
11: **for all** transaction $t \in D_s$ **do**
12: $t = t - \{i\}$;
13: Let i' be the first item of t, insert t into $D_{l \bigcup \{i'\}}$.

the set of frequent extensions of itemset l (line 1). In the construction step, D_l is traversed a second time. New conditional databases are constructed for the frequent extensions of l (line 4-7). The push-right step is performed immediately after all the frequent itemsets with l as prefix are discovered. In the push-right step, each transaction in D_l is inserted into l's siblings' conditional databases (line 11-13).

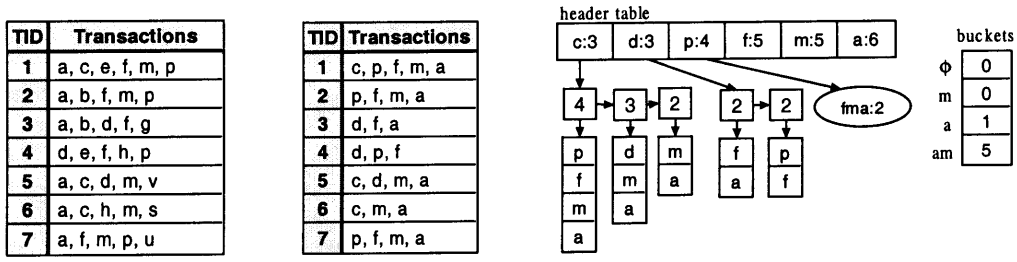
The correctness of the AFOPT algorithm is guaranteed by the fact that whenever a conditional database becomes the next one to be processed, the conditional database contains the complete information needed for mining. When the conditional databases are initially constructed from the original database, most of them are incomplete. The reason being that a transaction can belong to multiple conditional databases depending on the number of frequent items in it. However, we put a transaction into only one conditional database at any time. More specifically, we always put a transaction into the first conditional database it belongs to. The transaction is pushed progressively into the other conditional databases in the push-right step. Therefore, whenever a conditional database becomes the next one to be processed, the conditional database contains all the information needed.

3.1.1 Conditional database representation

Algorithm 1 is independent of the representation of conditional databases. Various structures have been proposed for storing conditional databases. Here we choose appropriate structures according to the density of the conditional databases. Three structures are used: an array structure, the ascending frequency ordered prefix-trie structure (or AFOPT structure for short) and buckets. These three structures are suitable for different situations. The array structure is favorable when conditional databases are sparse. The AFOPT structure is beneficial when conditional databases are dense. The bucket counting technique is very efficient for the conditional databases containing only a few frequent items.

The AFOPT structure is essentially a prefix-trie structure. It allows different transactions to share the storage of their prefixes. Items on a single path are collapsed into one node. Each node in the AFOPT structure stores four pieces of information: one or more items, the count of the transactions matched to this node, a child pointer and a right-sibling pointer. No other information, e.g. a parent pointer or a node-link pointer, is stored at an AFOPT node. In the array structure, we simple store the length of each transaction and all the frequent items in each transaction. The bucket counting technique is appropriate and extremely efficient when the number of distinct frequent items in a conditional database is around 10. The bucket counting technique is proposed by Agrawal et al. [2]. The basic idea is that if the number of frequent items in a conditional database is small enough, we can maintain a counter for every combination of the frequent items instead of constructing a new conditional database for each frequent item. For more details of the bucket counting technique, please refer to [2].

We use four parameters to decide when to use these three structures as follows: (1) the frequent itemsets containing only the top-$bucket\_size$ frequent items are counted using the bucket counting technique; (2) if the minimum support threshold is greater than $tree\_min\_sup$ or the average support of all the frequent items is no less than $tree\_avg\_sup$, then all the remaining conditional databases are represented using the AFOPT structure; otherwise (3) the conditional databases of the next $(tree\_alphabet\_size - bucket\_size)$ most frequent items are represented using the AFOPT structure, and the remaining conditional databases are represented using the array structure.



(a) database (b) (c) conditional databases

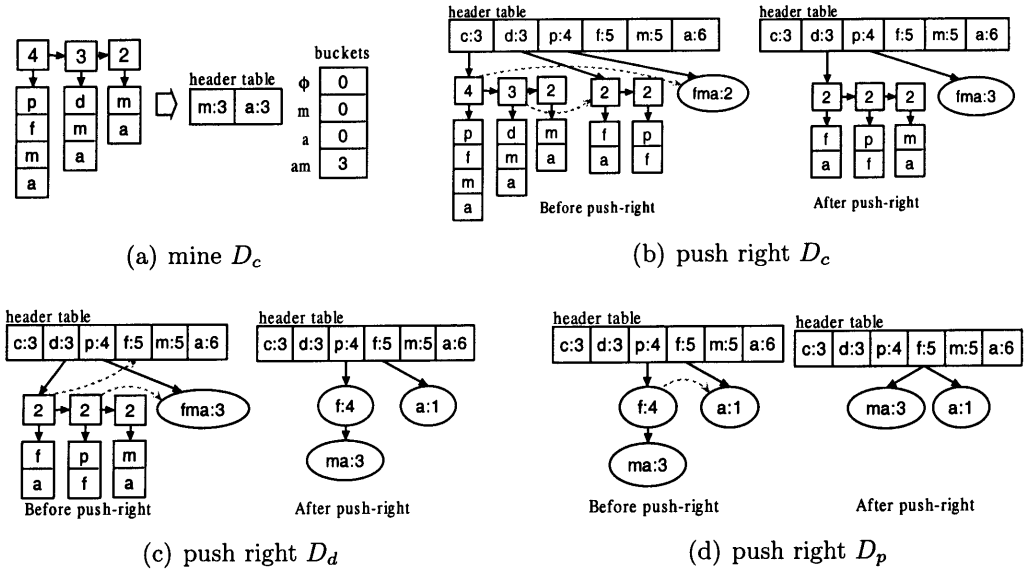
Figure 3.1: Conditional database representation

We use the example database shown in Figure 3.1(a) to illustrate our conditional database representation strategy. We set *min_sup* to 40%. There are 6 frequent items {c:3, d:3, p:4, f:5, m:5,a:6}. Figure 3.1(b) shows the projected database after removing infrequent items and sorting. The values of the parameters for conditional database representation are set as follows: *bucket_size*=2, *tree_alphabet_size*=4, *tree_min_sup*=50% and *tree_avg_sup*=60%. The frequent itemsets containing only m and a are counted using buckets of size 4 (=2^{bucket_size}). The conditional databases of f and p are represented by the AFOPT structure. The conditional databases of item c and d are represented by the array structure. The conditional databases and the header table are shown in Figure 3.1(c). From our experience, the bucket counting technique is appropriate and extremely efficient when the number of frequent items is around 10. A value between 20 and 100 is appropriate for the *tree_alphabet_size* parameter.

3.1.2 A running example

We use the example database shown in Figure 3.1(a) to illustrate the mining process of the AFOPT algorithm. The minimum support threshold is set to 40%. The conditional database representation parameters are set as follows: *bucket_size*=2, *tree_alphabet_size*=4, *tree_min_sup*=50% and *tree_avg_sup*=60% (the same as that used in the previous subsection).

In the first database scan, we count the support of each item. Six items are frequent and they are sorted into ascending frequency order: F={c:3, d:3, p:4, f:5, m:5, a:6}. According to the conditional database representation parameters, the frequent itemsets containing only m and a are counted using the bucket counting technique, the conditional databases of f and p are represented by the



(a) mine D_c

(b) push right D_c

(c) push right D_d

(d) push right D_p

Figure 3.2: The mining process of the AFOPT algorithm

AFOPT structure and the conditional databases of item c and d are represented by the array structure. Then we scan the database a second time. We remove infrequent items from each transaction and sort the remaining items according to their orders in F. The resultant transactions are inserted into the corresponding conditional databases according to their first item after sorting. For example, the first transaction becomes $\{c, p, f, m, a\}$ after sorting, so it is inserted into c's conditional database. The second transaction becomes $\{p, f, m, a\}$, hence it is inserted into p's conditional database. The complete structure constructed from the original database is shown in Figure 3.1(c). When constructing the conditional databases, we also update the counts in the buckets. After all the transactions are processed, we calculate the support of the itemsets containing m and a using the buckets and then dispose of the buckets. The calculation algorithm can be found in [1]. The remaining mining is performed on the conditional databases, and there is no need to access the original database.

The AFOPT algorithm traverses the search space in depth-first order. It first processes c's conditional database D_c to mine all the frequent itemsets containing c. Two items m and a are frequent in D_c. According to the parameter settings, the itemsets containing m and a in D_c are counted using the bucket counting technique as shown in Figure 3.2(a). No new conditional databases are constructed from D_c. The mining on D_c is finished, and D_c should be pushed right. In the push-right step, the first transaction in D_c is inserted into D_p, and

Datasets	AFOPT			H-Mine			FP-growth	
	size	build	pushright	Size	build	pushright	size	build
T10I4D100k (0.01%)	5116 kb	0.55s	0.37s	11838 kb	0.68s	0.19s	20403 kb	1.83s
T40I10D100k (0.5%)	16535 kb	1.85s	1.91s	46089 kb	2.10s	1.42s	104272 kb	6.16s
BMS-POS (0.05%)	17264 kb	2.11s	1.43s	38833 kb	2.58s	1.00s	47376 kb	6.64s
BMS-WebView-1 (0.06%)	711 kb	0.12s	0.01s	1736 kb	0.17s	0.01s	1682 kb	0.27s
chess (45%)	563 kb	0.04s	0.01s	1150 kb	0.05s	0.03s	1339 kb	0.12s
connect-4 (75%)	35 kb	0.73s	0.01s	22064 kb	1.15s	0.55s	92 kb	2.08s
mushroom (5%)	1067 kb	0.08s	0.04s	2120 kb	0.10s	0.03s	988 kb	0.17s
pumsb (70%)	375 kb	0.82s	0.02s	17374 kb	1.15s	0.43s	1456 kb	2.26s
pumsb_star(25%)	4393 kb	0.84	0.16	19034 kb	1.00s	0.44s	5650 kb	1.52s

Table 3.1: Comparison of structure size

the second transaction in D_c is inserted into D_d as shown in Figure 3.2(b). We simply discard the third transaction in D_c because all the itemsets starting with m have been counted using the buckets.

Then we move on to mine d's conditional database. No item is frequent in D_d. The mining on D_d is finished, and D_d is pushed right as shown in Figure 3.2(c). The first transaction in D_d is inserted into D_f, the second transaction is inserted into D_p and the third transaction is simply discarded because it starts with m.

Next the mining is performed on D_p, which contains a single branch. We simply enumerate the itemsets contained in the single branch without constructing any new conditional databases from D_p. The mining on D_p is finished. The single branch in D_p is pushed right to D_f as shown in Figure 3.2(d). There are two items m and a frequent in D_f. Their combinations are counted using the bucket counting technique. The whole mining process is finished.

3.1.3 Structure size comparison

We compared our adaptive conditional database representation strategy with two other typical structures: the FP-tree structure and the hyper-structure. Table 3.1 shows the size, the construction time ("build" column) and the push-right time if applicable, of the initial structure constructed from the original database by AFOPT, H-Mine and FP-growth. The datasets shown in Table 3.1 are publicly available. They are widely used in studying the frequent itemset mining problem. We set *bucket_size* to 8 and *tree_alphabet_size* to 20 in the AFOPT algorithm. The initial structure of AFOPT includes all the three structures. The array

structure in the AFOPT algorithm simply stores all items in a transaction. Each node in the hyper-structure stores three pieces of information: an item, a pointer pointing to the next item in the same transaction and a pointer pointing to the same item in the next transaction. Therefore, the size of the hyper-structure is roughly 3 times larger than the array structure used in AFOPT. A node in the AFOPT structure has only a child pointer and a right-sibling pointer. An FP-tree node has two more pointers for bottom-up traversal: a parent pointer and a node-link pointer. Table 3.1 shows that the AFOPT algorithm consumes the least amount of space and requires the least amount of maintenance time on almost all testing datasets.

3.1.4 The ascending frequency order

In the AFOPT algorithm, we use the ascending frequency order for exploring the search space and for sorting the frequent items in the AFOPT structure. In this subsection, we explain why we choose the ascending frequency order.

By using the ascending frequency order to explore the search space, we can minimize both the total number of conditional databases and the size of individual conditional databases. In the ascending frequency order, the candidate extensions of a frequent item include those items that are more frequent than the frequent item. This has two advantages. First, the ascending frequency order balances the size of the conditional databases, thus ensuring that a small conditional database is pushed right every time. Item i_1 is the most infrequent item, so D_{i_1} contains the fewest transactions. As j increases, the number of transactions in D_{i_j} increases, but the average transaction length decreases because items before i_j are removed. Therefore, a single conditional database cannot be very large, and it is usually much smaller than the original database. Secondly, the ascending frequency order balances the size of the sub search spaces, thus ensuring that the memory consumption for mining the conditional databases cannot be large. Intuitively, an itemset with higher support usually has more frequent extensions than an itemset with lower support. The most infrequent item is the first item. It has the largest set of candidate extensions but few of them are actually frequent. The frequency of successive items increases, while at the same time, the number of their candidate extensions decreases. Therefore, we need to

Datasets	Asc			Lex			Des		
	#cdb	time	max_mem	#cdb	time	max_mem	#cdb	time	max_mem
T10I4D100k (0.01%)	53688	4.52s	5199 kb	47799	4.89s	5471 kb	36725	5.32s	5675 kb
T40I10D100k (0.5%)	311999	30.42s	17206 kb	310295	33.83s	20011 kb	309895	43.37s	21980 kb
BMS-POS (0.05%)	115202	27.83s	17294 kb	53495	127.45s	38005 kb	39413	147.01s	40206 kb
BMS-WebView-1 (0.06%)	33186	0.69s	731 kb	65378	1.12s	901 kb	79571	2.16s	918 kb
chess (45%)	312202	2.68s	574 kb	617401	8.46s	1079 kb	405720	311.19s	2127 kb
connect-4 (75%)	12242	1.31s	38 kb	245663	2.65s	57 kb	266792	14.27s	113 kb
mushroom (5%)	9838	0.34s	1072 kb	258068	3.11s	676 kb	464903	272.30s	2304 kb
pumsb (70%)	272373	3.87s	383 kb	649096	12.22s	570 kb	469983	16.62s	1225 kb
pumsb_star(25%)	27916	3.13s	4397 kb	405381	10.57s	4387 kb	364897	62.77s	10007 kb

Table 3.2: Comparison of three item search orders (bucket size=0)

build a small number of descendant conditional databases in subsequent mining and their size shrinks quickly.

We compared three item search orders: the dynamic ascending frequency order ("Asc" column), the static lexicographic order ("Lex" column) and the dynamic descending frequency order ("Des" column). Table 3.2 shows the total number of conditional databases constructed ("#cdb" column), the total running time and the maximal memory usage when the three orders were adopted into the framework of the AFOPT algorithm. The minimum support threshold on each dataset is shown in the first column. On the first three datasets, the ascending frequency order needs to build a few more conditional databases than the other two orders, but its total running time and its maximal memory usage is less than that of the other two orders. This implies that the conditional databases constructed using the ascending frequency order are smaller. On the remaining datasets, the ascending frequency order requires the building of fewer conditional databases and needs less running time and memory usage, especially on dense datasets connect-4 and mushroom.

3.2 Time Complexity Analysis

In this section, we compare the complexity of the AFOPT algorithm with that of the FP-growth algorithm and the H-Mine algorithm. We mainly focus on comparing the three structures: the FP-tree structure used in the FP-growth algorithm, the hyper-structure used in the H-Mine algorithm and the AFOPT structure proposed in this chapter.

There are three steps when mining the AFOPT structure T_l representing a frequent itemset l's conditional database: (1) In the support counting step, T_l is traversed once to find the frequent extensions of l. (2) In the construction step, if l has more than one frequent extension, T_l is traversed one more time to construct the conditional databases of the frequent extensions. If the number of the frequent extensions is 1 or 0, nothing is done in this step. (3) In the push-right step, the subtrees of T_l are merged with T_l's right siblings, which involves mainly pointer adjustment and support update. The merge is done by traversing a subtree of T_l and the corresponding right sibling of T_l simultaneously. During the traversal, we match the two trees and update the counts of the nodes in the right sibling of T_l. The time complexity of the merging operation is two times of the size of the smaller tree in the worst case, and is much smaller than the size of the smaller tree in the average case. In summary, if the number of frequent items in a conditional database D_l is greater than 1, the AFOPT algorithm needs to traverse D_l no more than four times, otherwise the AFOPT algorithm needs to traverse D_l no more than three times. We call D_l an internal conditional database if the number of frequent items in D_l is more than one, otherwise if the number of frequent items in D_l is 0 or 1, D_l is called a leaf conditional database.

One optimization that can be made is that we can do counting in the push-right step to save some traversal cost. In the push right step, when merging two trees to get the next conditional database to be processed, we can count the support of the items in these two trees. After the push-right step, we do not need to traverse the next conditional database to be processed to find the frequent items. By using this optimization technique, the traversal times can be reduced by one.

Given a conditional database D_l, let $cost_T(D_l)$ be its traversal cost, and $cost_C(D_l)$ be its construction cost. Assume that N_I internal conditional databases are constructed during the whole mining process, denoted as $D_1^I, D_2^I, \cdots, D_{N_I}^I$, and N_L leaf conditional databases are constructed, denoted as $D_1^L, D_2^L, \cdots, D_{N_L}^L$, then the total mining cost of the AFOPT algorithm is $\sum_{j=1}^{N_I}(3 \cdot cost_T(D_j^I) + cost_C(D_j^I)) + \sum_{j=1}^{N_L}(2 \cdot cost_T(D_j^L) + cost_C(D_j^L))$. The FP-growth algorithm does not have the push-right step. It needs to traverse an internal conditional database twice and a leaf conditional database only once. Therefore, the mining cost of the FP-growth algorithm is $\sum_{j=1}^{N_I}(2 \cdot cost_T(D_j^I) + cost_C(D_j^I)) + \sum_{j=1}^{N_L}(cost_T(D_j^L) + cost_C(D_j^L))$.

The H-Mine algorithm needs the same number of traversals as the AFOPT algorithm for both internal and leaf conditional databases, but H-Mine constructs conditional databases via link adjustment, which is less expensive than physical construction.

At a first glance, it seems that the AFOPT algorithm requires the highest cost since it needs more traversal cost than the FP-growth algorithm, and more construction cost than the H-Mine algorithm. However, the running time of the three algorithms also depends on the traversal and construction cost of individual conditional databases and the total number of conditional databases constructed during the mining process. In the rest of this section, we prove that the AFOPT structure combined with the ascending frequency order results in the minimal traversal cost and the minimal number of conditional databases.

Lemma 1 *Given a conditional database D_l, we use three different structures to represent it and use a particular traversal strategy for each structure: (1) the hyper-structure and the top-down traversal strategy, (2) the FP-tree structure and the bottom-up traversal strategy, and (3) the AFOPT structure proposed in this thesis and the top-down traversal strategy. We assume that the cost of visiting a single node in the above three structures is the same, then the AFOPT structure combined with the top-down traversal strategy needs the least traversal cost.*

Proof 1 *Let $cost_{node}$ be the cost of visiting a single node in the three structures, and N be the total number of item occurrences in D_l. The hyper-structure does not allow different transactions to share the storage of their prefixes. Therefore, the number of nodes in the hyper-structure is N and the traversal cost of the hyper-structure is $N \cdot cost_{node}$. The other two structures allow different transactions to share the storage of their prefixes, thus the size of the other two structures is smaller than N. Therefore, the traversal cost of the FP-tree structure and the AFOPT structure is smaller than that of the hyper-structure.*

Now we prove that the traversal cost of the FP-tree structure is always larger than that of the AFOPT structure. Let T_{fp} and T_{afopt} denote the FP-tree and the AFOPT structure representing D_l respectively. Let n_{fp} be the number of leaf nodes in T_{fp}, then the bottom-up traversal strategy needs to visit n_{fp} branches. Recall that items in an FP-tree are sorted into descending frequency order, hence items

in the n_{fp} itemsets obtained by traversal these n_{fp} branches bottom-up are sorted into ascending frequency order. If we use these n_{fp} itemsets to build a prefix-trie, which is exactly the AFOPT structure storing D_l. The total number of node visits of the FP-growth algorithm is equal to the total length of the n_{fp} itemsets. We traverse the AFOPT structure in top-down depth-first order. Each node is visited exactly once. The total number of node visits of the AFOPT algorithm is equal to the size of the AFOPT structure, which is smaller than the total length of the n_{fp} itemsets. Therefore, the AFOPT algorithm needs less traversal cost than the FP-growth algorithm.

The total number of the conditional databases constructed during the mining process depends on the order in which the frequent items are sorted and the search space is divided. The ascending frequency order is capable of minimizing the total number of the conditional databases. Let i_{lowest} be the frequent item with the lowest support and $i_{highest}$ be the most frequent item. Consider two situations: (1) i_{lowest} is the first item; and (2) $i_{highest}$ is the first item. In the AFOPT algorithm, the candidate extensions of a frequent item include all the frequent items after it. Therefore, in the two cases, the two items almost have the same set of candidate extensions. However, the frequency of $i_{highest}$ is usually much higher than that of i_{lowest}, i.e. $i_{highest}$'s conditional database contains much more transactions than i_{lowest}'s conditional database. This implies that it is very likely that the number of frequent extensions of $i_{highest}$ is much larger than that of i_{lowest}. Therefore, if we take $i_{highest}$ as the first item, it is very likely that we have to construct bigger and/or more conditional databases in subsequent mining. On the contrary, if i_{lowest} is the first item, the number of items that are frequent in i_{lowest}'s conditional database cannot be very large. The items after i_{lowest} become more and more frequent, but their candidate extension sets become smaller and smaller, so the transactions in their conditional databases become shorter and shorter. This ensures that the number and the size of the conditional databases constructed in subsequent mining cannot be large.

It also explains why the FP-growth algorithm uses the bottom-up traversal strategy. The items are sorted into descending frequency order in an FP-tree. To save the cost for subsequent mining, i.e. to let the most frequent item have the smallest candidate extension set, and let the most infrequent item have the

largest candidate extension set, the candidate extensions of an item include all the frequent items before it. Hence the FP-tree structure has to be traversed bottom-up along the node-links. The hyper-structure does not change except for the hyper-links, therefore the items must be sorted according to a fixed order in the H-Mine algorithm, e.g. the lexicographic order. In the lexicographic order, the candidate extensions of an item include all the items that are lexicographically greater than the item. It is very likely that this item ordering method need to construct more and/or larger conditional databases than the other two algorithms in subsequent mining.

If an itemset l's conditional database D_l can be represented by a single branch, we can enumerate all the frequent patterns contained in the single branch directly. In this case, we do not need to build new conditional databases from D_l even if l has more than one frequent extension. The AFOPT structure can help further reduce the number of conditional databases because conditional databases are more likely to be represented by a single branch using the AFOPT structure.

Lemma 2 *Given a conditional database D_l, if it can be represented by a single branch using the FP-tree structure, then it can also be represented by a single branch using the AFOPT structure, but not vice versa.*

Proof 2 *From Lemma 1, we know that using the AFOPT structure, the number of node visits is minimal. If a conditional database can be represented by a single branch using the FP-tree structure, then it must be able to be represented by a single branch using the AFOPT structure. Otherwise, it conflicts with Lemma 1.*

On the other hand, if a conditional database can be represented by a single branch using the AFOPT structure, the FP-tree representing it may contain multiple branches. Consider an example, the set of frequent extension of an itemset l is $\{f, a, b, c, d\}$, and they are sorted into ascending frequency order. The AFOPT structure representing $l \bigcup \{f\}$'s conditional database $D_{l \bigcup \{f\}}$ contains a single branch $\{a : 3, b : 3, c : 2, d : 2\}$. If we use the FP-tree structure to represent D_f, we have two branches: $\{d : 2, c : 2, b : 2, a : 2\}$ and $\{b : 1, a : 1\}$.

Table 3.3 shows the number of recursive calls (total number of internal conditional databases) of algorithms AFOPT and FP-growth on several datasets. For

Datasets	min_sup	#FI	AFOPT	FP-Growth	ratio
T10I4D100k	0.01%	411366	45734	105945	2.32
T40I10D100k	0.5%	1286038	309580	616922	1.99
BMS-POS	0.05%	582753	111489	282524	2.53
BMS-WebView-1	0.06%	461522	19404	62586	3.23
chess	45%	2832778	177218	597359	3.37
connect-4	75%	1585552	5940	42381	7.13
mushroom	5%	3755512	4314	23170	5.37
pumsb	70%	2698265	159850	488881	3.06
pumsb_star	25%	2064946	13975	68340	4.89

Table 3.3: Number of recursive calls

the AFOPT algorithm, we do not use the array structure and the bucket counting technique, but use only the AFOPT structure to represent conditional databases. The two algorithms uses the same order to explore the search space. Hence the difference in the total number of recursive calls is caused by the difference between the two structures—the AFOPT structure and the FP-tree structure. The AFOPT algorithm needs much less recursive calls than the FP-growth algorithm. This implies that more conditional databases are represented by single branches in the AFOPT structure than in the FP-tree structure.

The AFOPT structure usually requires less construction cost than the FP-tree structure as shown in Table 3.1. Both the AFOPT structure and the FP-tree structure are essentially prefix-trees. The AFOPT structure may contain more nodes than the FP-tree structure because the ascending frequency order reduces the possibility of prefix sharing. However, the FP-growth algorithm needs to maintain a parent pointer and a node-link pointer at each node, which incurs additional construction cost and consumes more space. In the AFOPT structure, we collapse nodes on a single path into one single node, which can lead to significant space saving and construction cost saving. The ascending frequency order also helps reduce tree construction cost. The reason being that to insert a transaction into a prefix-tree, we need to match the transaction against the prefix-tree. Let p be the node that a transaction t currently matches. Then to find the node that is matched by the next item of t, we have to traverse along the right sibling pointers of p's children. In the FP-tree structure, the most frequent items are at the top levels and they usually have many children. In contrast, the items

are sorted into ascending frequency order in the AFOPT structure. The least frequent items are at the top levels and they usually have a moderate number of children. Therefore, the construction cost of the AFOPT structure is usually less than the FP-tree structure.

Note that in the AFOPT algorithm, additional traversal cost is caused by the push-right step. What if we do not have this step? The answer is that if we do not have this step, a conditional database will consist of multiple subtrees. The number of subtrees constituting an item i's conditional database is exponential to the number of items before i in the worst case. While the number of merging operations needed is equal to the number of items before i in the worst case. To save the traversal cost, we choose to perform the merging operation.

3.3 Mining Maximal Frequent Itemsets

Mining maximal frequent itemsets can be viewed as given a transaction database and a minimum support threshold *min_sup*, find a border through the search space tree such that all the nodes below the border are infrequent and all the nodes above the border are frequent. As shown in Figure 2.1, the dotted line represents the border of the frequent itemsets. Among all the nodes above the border, only leaf nodes can be maximal because any other node has at least one frequent child (superset). The goal of maximal frequent itemset mining is to find the border by counting support for as few as possible itemsets.

Algorithm 1 explores the whole pattern space with respect to a given minimum support threshold. In this section, we describe how to extend the AFOPT algorithm to mine maximal frequent itemsets.

3.3.1 Incorporating pruning techniques

Several pruning techniques have been proposed to remove non-maximal itemsets. The basic idea is that if we know the itemsets in a conditional database cannot be maximal, then we do not mine the conditional database to save mining cost. These pruning techniques can be easily and efficiently incorporated into the AFOPT algorithm because the distinct features of the AFOPT structure.

Lookahead Technique: The lookahead technique is the most effective prun-

ing technique. The basic idea is to check whether $l \bigcup cand\_exts(l)$ is frequent before/when mining l's conditional database D_l. If $l \bigcup cand\_exts(l)$ is frequent, then there is no need to perform mining on D_l because any transaction in D_l is a subset of $cand\_exts(l)$. There is also no need to mine D_l's right siblings for the same reason. The AFOPT structure and the top-down traversal strategy make the lookahead operation very easy. We can simply examine the left-most branch of the AFOPT structure representing D_l. Itemset $l \bigcup cand\_exts(l)$ is frequent if and only if two conditions hold: (1) the length of the left-most branch is equal to $\|cand\_exts(l)\|$, and (2) the support of the leaf node of the left-most branch is no less than the minimum support threshold. For example, in Figure 3.2(c), before mining D_p, we first check whether $\{p\} \bigcup \{f, m, a\}$ is frequent by checking the left-most branch of D_p, which is actually the only branch in D_p. We find that $pfma$ is frequent, which implies that there is no need to mine D_f and the remaining conditional databases. In the AFOPT algorithm, the lookahead pruning is performed before a conditional database is traversed. While in other maximal frequent itemset mining algorithms, the lookahead pruning is performed during the first traversal of the conditional databases. Hence if the lookahead pruning succeeds, the AFOPT algorithm saves one traversal of the conditional database compared with other algorithms.

Dynamic Reordering Technique: The dynamic reordering technique is to sort the frequent items in every conditional database into ascending frequency order. This technique is first proposed in [41] to improve the possibility that the lookahead technique succeeds. The rationale behind is that the candidate extensions of an item include all the items after it, and the item appearing last in the ordering appears in all the other items' candidate extension set. An item with high frequency has much more chance to appear in long patterns than an item with low frequency. Hence the most frequent item should be at the end. We already adopt this technique in the AFOPT algorithm. In fact, the effect of the ascending frequency order in the AFOPT algorithm is not limited to improving the success possibility of the lookahead technique. It also has a great effect on the efficiency of the AFOPT algorithm as explained before.

Removing Frequent Non-closed Itemsets: If a frequent itemset is not closed, then it cannot be maximal. Some non-closed itemsets can be easily identified in the AFOPT algorithm. There are two cases where we can do

non-closed itemset pruning. The first case is after we find all the frequent extensions of an itemset l, if there exists some item $i \in freq\_exts(l)$ such that $support(l \bigcup \{i\}) = support(l)$, then we remove i from $freq\_exts(l)$ and put i into l to exclude those patterns containing l but not containing item i. The benefits are two-fold. On the one hand, the search space is reduced. On the other hand, the construction cost of the AFOPT structure is reduced because item i is excluded from the conditional databases. We use an example to illustrate this. In Figure 3.1(c), items m and a appear in every transaction of D_c. We can perform mining directly on D_{cma} instead of D_c. D_{cma} is empty. Therefore, the mining cost on D_c is saved. The second case is due to the adoption of the AFOPT structure and the top-down traversal strategy. When mining D_l, we check whether the root of D_l has only one child. If it is true, let q denote the child, then the mining can be performed on the subtree rooted at q directly to avoid mining the patterns containing l but not containing the item in node q.

Superset Pruning Technique: Like the lookahead technique, the superset pruning technique is also capable of removing a whole subtree from the search space. It checks whether there exists some maximal frequent itemset such that the maximal frequent itemset is a superset of $l \bigcup cand\_exts(l)$. If there exists such a maximal itemset, then there is no need to mine D_l and l's right siblings' conditional databases. When the number of maximal patterns is large, how to efficiently implement the superset pruning technique is very challenging. We discuss this issue in the next subsection.

3.3.2 Subset checking

We need to do subset checking in two situations: (1) when we perform superset pruning, we need to check whether $l \bigcup cand\_exts(l)$ is a subset of some existing maximal frequent itemsets; and (2) before output a frequent pattern, we need to check whether it is maximal. One property of the search space tree is that given a leaf itemset l, only the itemsets on l's left can be a superset of l. It also holds for the portion above the dotted line. The AFOPT algorithm explores the search space in depth-first order. When it reaches a leaf node, all the supersets of the itemset represented by the leaf node have been discovered. It implies that when checking whether a frequent itemset is maximal, we can check it against the set of

Algorithm 2 AFOPT-Max Algorithm

Input:
l is a frequent itemset;
D_l is l's conditional database;
min_sup is the minimum support threshold;
LMFI_l is the set of maximal frequent itemsets containing l;
Output:
NewMFI_l is the set of new maximal itemsets mined from D_l;
Description:
1: Scan D_l to count frequent items and sort them in ascending frequency order, F=$\{i_1, i_2, \cdots, i_n\}$;
2: $I = \{i | i \in F \text{ and } support(l \bigcup \{i\}) = \|D_l\|\}$;
3: **if** $I \neq \phi$ **then**
4: $F = F - I$; $l = l \bigcup I$;
5: **for all** item $i \in F$ **do**
6: $D_{l \bigcup \{i\}} = \phi$;
7: **for all** transaction $t \in D_l$ **do**
8: $t = t \bigcap F$;
9: Sort items in t according to their orders in F;
10: Let i be the first item of t, insert t into $D_{l \bigcup \{i\}}$.
11: **for all** maximal pattern p in LMFI_l **do**
12: Let i be the first extensions of l appearing in p, put p into $\text{LMFI}_{l \bigcup \{i\}}$;
13: **for all** $i \in F$, from i_1 to i_n **do**
14: $s = l \bigcup \{i\}$; $F = F$ - $\{i\}$;
15: **if** $s \bigcup F$ is a subset of some pattern in LMFI_s **then**
16: **return**
17: **if** $s \bigcup F$ is frequent in D_l **then**
18: Add $s \bigcup F$ to NewMFI_l;
19: **return** ;
20: NewMFI_s = AFOPT-Max(s, D_s, *min_sup*, LMFI_s);
21: $\text{NewMFI}_l = \text{NewMFI}_l \bigcup \text{NewMFI}_s$;
22: **for all** transaction $t \in D_s$ **do**
23: $t = t - \{i\}$;
24: Let i' be the first item of t, insert t into $D_{l \bigcup \{i'\}}$.
25: **for all** pattern $p \in \text{LMFI}_s \bigcup \text{NewMFI}_s$ **do**
26: Let i' be the first frequent extension of l after i appearing in p, put p into $\text{LMFI}_{l \bigcup \{i'\}}$;

maximal itemsets discovered so far. Therefore, we can push maximality checking into the mining process instead of waiting until all the maximal patterns have been found. The set of existing maximal frequent itemsets keeps growing as the mining going on. It is possible that the number of maximal patterns gets very large on dense datasets. To do subset checking against a large set of patterns is very costly. In this subsection, we introduce a pattern projection technique to reduce the cost for subset checking.

During the mining process, we maintain the set of maximal frequent itemsets discovered so far and distribute them to where they are needed for maximality checking. Similar to constructing a conditional database for each itemset, we construct a local maximal pattern set for each itemset. The local maximal pattern set of an itemset l contains all the maximal itemsets containing l, denoted as LMFI_l. When checking whether an itemset l is maximal, we check l against

LMFI_l instead of the whole set of maximal patterns. The construction of local maximal pattern sets is similar to the construction of conditional databases. We also have a construction step and a push-right step. In the construction step, itemset l's frequent extensions' local maximal pattern sets are constructed from l's local maximal pattern set (line 11-12). In the push-right step, l's local maximal pattern set is distributed to l's right siblings's local maximal pattern set (line 25-26). The pattern projection technique can also be costly when the number of patterns is large and the depth of the recursion is deep. To implement the pattern projection technique efficiently, instead of physically constructing the local maximal frequent itemsets, we adopt a pseudo construction strategy, i.e. LMFI_l is a collection of pointers pointing to those patterns containing itemset l.

The pseudo-code of the AFOPT-Max algorithm is shown in Algorithm 2. The non-closed itemsets are removed at line 2-4. The superset pruning is performed at line 15-16. The lookahead technique is performed at line 17-19.

3.4 A Performance Study

In this section, we compare the performance of the AFOPT algorithm and the AFOPT-Max algorithm with other frequent itemset mining algorithms. All the experiments were conducted on a 1Ghz Pentium III with 256MB memory running Microsoft Windows 2000 Professional. All codes were complied using Microsoft Visual C++ 6.0.

Table 3.4 shows the datasets used for the performance study. BMS-WebView-1 is a real world sparse dataset. It contains several months of click-stream data from two e-commerce Web sites[104]. T10I4D100k and T25I20D100k are two synthetic datasets generated by IBM Quest Synthetic Data Generation Code[1]. The detailed dataset generation process can be found in [5]. Dataset T25I20D100k is used in [35]. Pumsb, connect-4 and mushroom are three dense datasets obtained from the UCI machine learning repository[2]. Table 3.4 lists some statistical information about the datasets, including their size, the number of transactions, the number of distinct items, the maximal transaction length and the average transaction length. The statistical information provides some rough description of the

[1]http://www.almaden.ibm.com/cs/quest/syndata.html

[2]http://www.ics.uci.edu/ mlearn/MLRepository.html

Data Sets	Size	#Trans	#Items	AvgTransLen	MaxTransLen
BMS-WebView-1	1.28MB	59601	497	2.51	267
T10I4D100k	5.06MB	98413	23423	10.16	29
T25I20D100k	12.11MB	100,000	5137	28.00	50
pumsb	14.75MB	49046	2113	74.00	74
connect-4	12.14MB	67557	129	43.00	43
mushroom	0.83MB	8124	119	23.00	23

Table 3.4: Datasets

density of the datasets. Generally speaking, BMS-WebView-1 and T10I4D100k are two sparse datasets, Pumsb, connect-4 and mushroom are three dense datasets and T25I20D100k can be regarded as something in between.

Figure 3.3 shows the length distributions of the maximal patterns in the datasets with a specific minimum support threshold. The minimum support threshold is shown in the parentheses. The x-axis is the length of the patterns and the y-axis is the number of maximal patterns of that length. The pattern length distributions in the several datasets are quite different: dataset BMS-WebView-1 and the two synthetic datasets are highly left-skewed, i.e. the majority of the patterns are very short; the pattern distribution in pumsb is close to a normal distribution; datasets mushroom and connect-4 are two highly right-skewed datasets, especially the mushroom dataset.

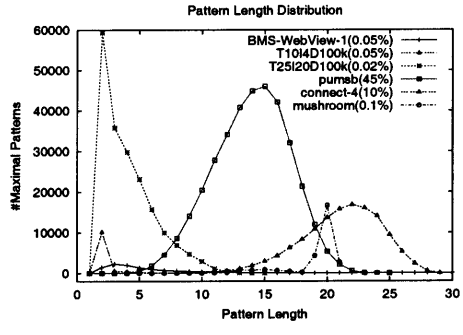
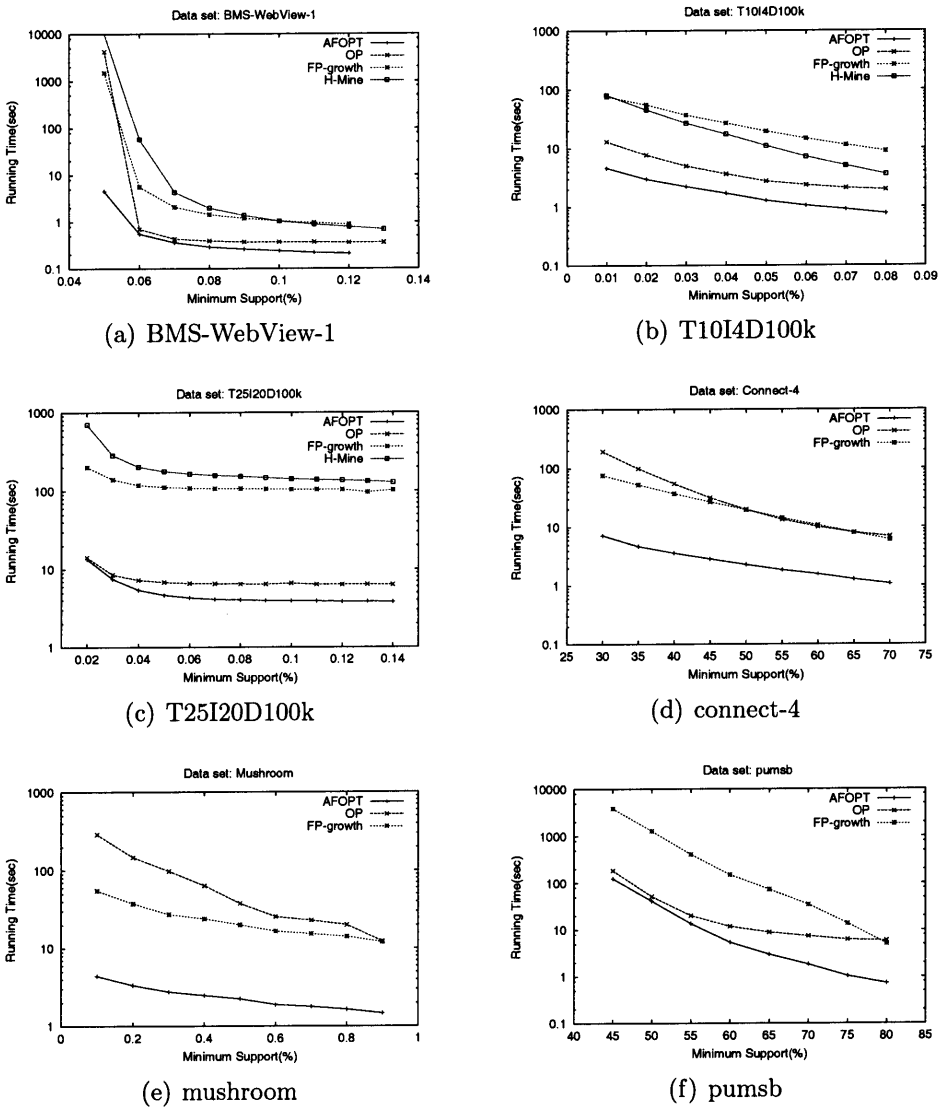


Figure 3.3: Pattern length distribution

3.4.1 Mining all frequent itemsets

We compared the efficiency of the AFOPT algorithm with the FP-growth algorithm, the H-Mine algorithm and the OP algorithm. We obtained the executables



(a) BMS-WebView-1 (b) T10I4D100k

(c) T25I20D100k (d) connect-4

(e) mushroom (f) pumsb

Figure 3.4: Mining all frequent itemsets

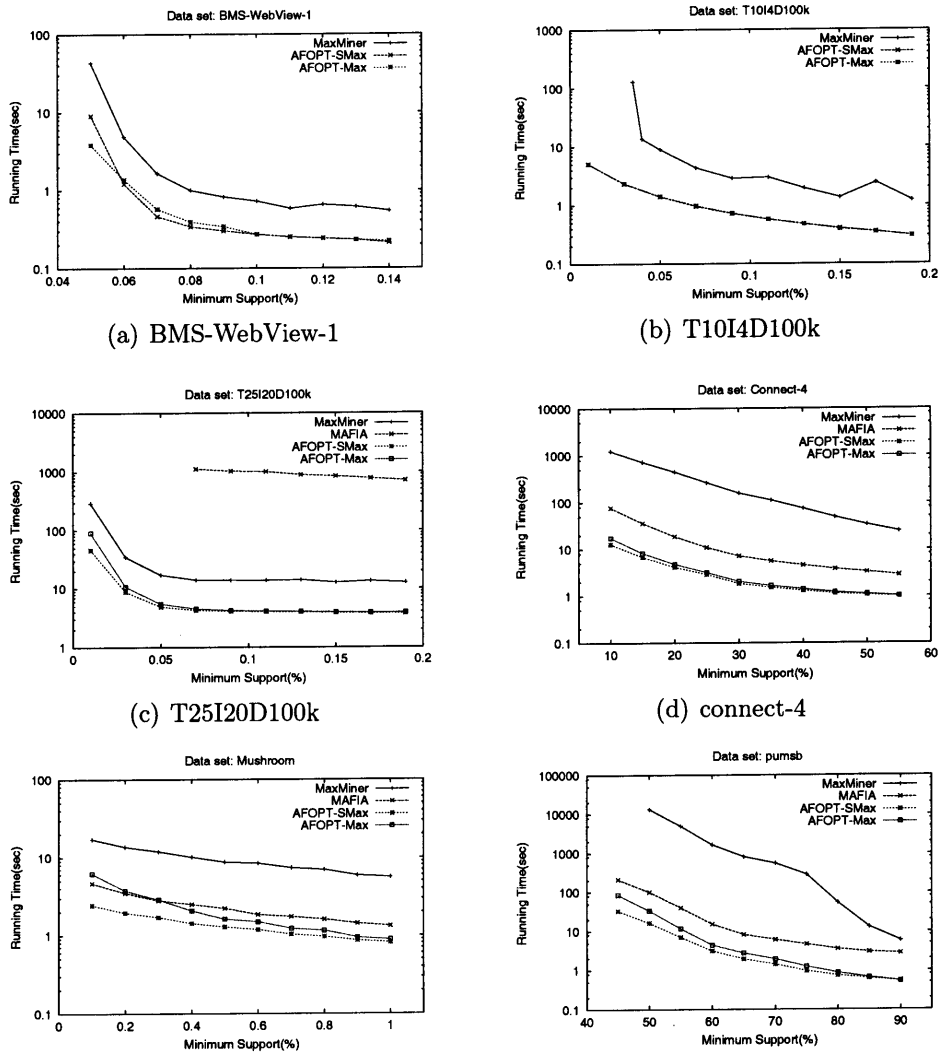
of the FP-growth algorithm and the OP algorithm from their authors. We implemented the H-Mine algorithm by ourselves. Figure 3.4 shows the running time of the four algorithms over the six datasets. The running time shown in the figures includes only input time and CPU time. The pattern output time is excluded because it is the same for all the algorithms.

Figure 3.4 shows that the running time of AFOPT is quite stable with respect to the decreasing of the minimum support threshold, and AFOPT outperforms the other three algorithms on all testing datasets. On the sparse dataset BMS-WebView-1, when the minimum support is greater than 0.1%, FP-growth shows the worst performance. With the decreasing of the minimum support thresh-

old, H-Mine becomes the slowest one. When the minimum support is greater than 0.06%, the running time of the AFOPT algorithm and the OP algorithm is very close (both less than 1 second). Both of them are significantly faster than H-Mine and FP-growth. When the minimum support is decreased to 0.05%, the number of patterns increases sharply from 461521 (0.06%) to 1985712679. The AFOPT algorithm still can finish mining in less than 5 seconds, while the other three algorithms need more than 1000 seconds. On synthetic sparse dataset T10I4D100k, FP-growth performs the worst. When the minimum support threshold is decreased to 0.01%, the running time of H-Mine eventually exceeds that of FP-growth. The AFOPT algorithm is nearly 3 times faster than OP algorithm on this dataset.

On dataset T25I20D100k, AFOPT and OP show similar performance. Both of them are significantly faster than the H-Mine algorithm and the FP-growth algorithm. The inefficiency of the FP-growth algorithm is caused by the sparsity of the dataset. It needs to construct a large FP-tree from the dataset, and the tree construction time even exceeds that of the total running time of OP and AFOPT. The factors that deteriorate the performance of the H-Mine algorithm are the static lexicographical item ordering method and the unfiltered array-based conditional database representation.

The H-Mine algorithm uses the FP-tree structure on dense datasets. It has the same performance as FP-growth on dense datasets. So we did not show the running time of H-Mine on three dense datasets. On dataset pumsb, when the minimum support threshold is higher than 60%, AFOPT is much faster than the other algorithms. When the minimum support threshold gets lower, the running time of the AFOPT algorithm becomes close to that of OP, but still significantly faster than FP-growth. On connect-4 and mushroom datasets, OP algorithm needs the longest running time. At the lowest minimum support threshold (0.1% for mushroom and 30% for connect-4), AFOPT is about 10 times faster than FP-growth, and about 60 and 40 times faster than OP respectively. The performance gain of the AFOPT algorithm is due to the adoption of the AFOPT structure and the ascending frequency order.



(a) BMS-WebView-1 (b) T10I4D100k

(c) T25I20D100k (d) connect-4

(e) mushroom (f) pumsb

Figure 3.5: Mining maximal frequent itemsets

3.4.2 Mining maximal frequent itemsets

We compared the performance of AFOPT-Max with MAFIA and MaxMiner. We obtained the executables of MaxMiner and MAFIA from their authors. MaxMiner produces the exact set of maximal patterns using a post-processing step. The output of MAFIA is a superset of the maximal patterns. We also modified the AFOPT-Max algorithm by removing the costly superset pruning to produce a superset of the maximal patterns, denoted as AFOPT-SMax in the figures.

The running time shown in Figure 3.5 includes both CPU time and I/O time. The output time of MAFIA and AFOPT-SMax is a little longer than that of the other two algorithms because they both generate a superset of the maximal

patterns.

Figure 3.5 shows that the running time of AFOPT-Max and AFOPT-SMax is very close, which indicates that the pattern projection technique is very effective in reducing subset checking cost. On two sparse datasets BMS-WebView-1 and T10I4D100k, MAFIA fails to run, therefore we compare only the other three algorithms. On BMS-WebView-1, when minimum support threshold is higher than 0.1%, AFOPT-Max and AFOPT-SMax use identical running time. With the decreasing of minimum support threshold, AFOPT-Max needs a little more running time due to the subset checking cost. When the minimum support threshold is lower than 0.06%, AFOPT-SMax becomes a little slower because it needs more output time.

On dataset T25I20D100k, MAFIA performs the worst. Even with minimum support threshold of 0.11%, its running time exceeds 1000 seconds. Our two algorithms need less than 100 seconds even with a much lower minimum support threshold of 0.01%. On three dense datasets pumsb, connect-4 and mushroom, MaxMiner is significantly slower than the other three algorithms, e.g. on dataset pumsb, its running time is more than 10000 second with minimum support of 50%, while AFOPT-Max and AFOPT-SMax need less than 100 second even with a lower minimum support of 45%. The reason being that MaxMiner uses a breadth-first strategy, which makes MaxMiner not efficient on dense datasets. On mushroom dataset, when minimum support is lower than 0.3%, AFOPT-Max needs little more running time than MAFIA. The reason being that mushroom is a very small and dense dataset, the cost for subset checking is relatively high compared with the mining cost. The AFOPT-SMax algorithm does not need to do subset checking, and it is faster than MAFIA.

3.5 Summary

In this chapter, we propose an efficient algorithm AFOPT for mining all frequent itemsets. We also extend the algorithm to mine maximal frequent itemsets by incorporating several pruning techniques. Both algorithms show significant performance improvements over previous work.

The AFOPT algorithm uses a new data structure—AFOPT structure to store

conditional databases. The distinct feature of the AFOPT structure is that it uses a different combination of traversal strategy and item ordering method. More specifically, the AFOPT structure is traversed in top-down depth-first order and the items in the AFOPT structure are sorted into ascending frequency order. Our analysis and experiment results show that the combination of these two methods is more efficient than the combination of the bottom-up traversal strategy and the descending frequency order, which is adopted by the FP-tree structure. Both of the above two combinations are much more efficient than the other two combinations—the combination of the top-down traversal strategy and the descending frequency order, and the combination of the bottom-up traversal strategy and the ascending frequency order.

In this chapter, we assume that the conditional databases constructed from a database can fit into the memory and concentrate on in-core mining of frequent itemsets. In the next chapter, we describe how to handle the cases when the conditional databases are too large to fit into the memory.

CHAPTER 4

SSP: EFFICIENT OUT-OF-CORE MINING OF FREQUENT ITEMSETS

The AFOPT algorithm proposed in the previous chapter is dedicated for in-memory mining. In this chapter, we focus on studying the situation where the conditional databases are too large to fit into the main memory, and propose an efficient approach SSP to scalable out-of-core mining of frequent itemsets from very large databases. The SSP approach partitions the database according to the search space of the frequent itemset mining problem, which is different from the data-based partitioning algorithm *Partition* [79].

The *Partition* algorithm is based on the observation that if we divide the database into several disjoint partitions, then a frequent itemset must be frequent in at least one partition. The *Partition* algorithm divides the database into disjoint partitions and an Apriori-like algorithm is adopted to mine each partition in the memory. The itemsets frequent in at least one partition form the set of candidate itemsets on the whole database. They are verified by a second database scan. This approach has several problems: (1) The frequent itemset mining task is not only I/O intensive, but also CPU intensive. Mining the partitions independently means duplicating the computation cost several times. (2) It is difficult to estimate the amount of space required for mining a partition. (3) The algorithm requires two database scans only if the union of the frequent itemsets on each partition can be held in the memory. If this assumption does not hold, which may often be the case when the database is dense and/or the minimum support threshold is low, then the algorithm needs to scan the database multiple times.

The partitions in the *Partition* algorithm do not share data but share frequent itemsets. The SSP approach adopts a reverse strategy. With the SSP approach, a transaction can belong to multiple partitions but a frequent itemset belongs to one and only one partition. As such, frequent itemsets can be mined from each partition without the need for scanning the whole database to verify their support.

Since there is data overlap among partitions, the total size of the partitions of a database can be much larger than the original database. If the partitions cannot be held in the memory, writing and reading the partitions can incur high I/O cost if handled improperly. Therefore, the main issue in the SSP approach is how to utilize overlap among partitions to reduce I/O cost. We propose three algorithms to solve the problem.

4.1 The Framework

The SSP approach is based on the framework of the AFOPT algorithm. The AFOPT algorithm has several features that make it suitable for mining very large databases: (1) Thanks to the adaptive conditional database representation strategy, the AFOPT algorithm generally consumes less space than other pattern growth algorithms. Therefore, the AFOPT algorithm has higher chance to do in-core mining even when other pattern growth algorithms have to do out-of-core mining. (2) The AFOPT algorithm adopts the ascending frequency order to explore the search space. The ascending frequency order balances the size of the conditional databases. Hence a single conditional database is substantially smaller than the original database. The ascending frequency order also balances the size of the sub search spaces, thus we need to build a small number of descendant conditional databases in subsequent mining and their size shrinks quickly. This ensures that the maximal memory usage of the AFOPT algorithm is moderate, which is a useful feature when dealing with very large databases.

One observation of the AFOPT algorithm is that if all the conditional databases constructed from a database can be held in the memory, the space required for storing them is much smaller than their total size because of the large amount of data sharing among them. A transaction may belong to multiple conditional databases depending on the number of frequent items in it, but we keep only one copy of the transaction in the first conditional database it belongs to. On average, the total size of the conditional databases is $L_{avg}/2$ times as large as the space required for storing the conditional databases, where L_{avg} is the average transaction length. Therefore, when the conditional databases are too large to fit into the memory, the main issue is how to utilize the overlap among conditional databases to reduce I/O cost. We propose three out-of-core algorithms SSP-naive,

SSP-static and SSP-dynamic to solve the problem. The first two algorithms use static strategies to partition the database before mining. SSP-dynamic adopts a dynamic strategy. It puts data on a disk only when new data are to be loaded but there is no free space in the memory.

4.2 The SSP-naive Algorithm

When all the conditional databases constructed from a database cannot fit into the memory, a simple method is to keep one conditional database in the memory at one time. We call this simple algorithm as SSP-naive. The FP-growth algorithm [35] also uses this method to handle the situation where an FP-tree is too large to fit into the memory.

Datasets	min_sup	DBSize	MaxCDBSize	ratio	MaxTotalSize	ratio
BMS-POS	0.02%	15166.47 kb	632.37 kb	23.98	1426.14 kb	10.63
BMS-WebView-1	0.05%	817.34 kb	21.79 kb	37.51	44.02 kb	18.57
BMS-WebView-2	0.005%	1702.3 kb	34.21 kb	49.76	76.8 kb	22.17
chess	20%	474.41 kb	179.71 kb	2.64	450.42 kb	1.05
connect-4	30%	11611.36 kb	731.89 kb	15.86	735.79 kb	15.78
mushroom	0.5%	761.63 kb	75.73 kb	10.06	165.59 kb	4.60
pumsb	55%	14368.95 kb	582.43 kb	24.67	593.71 kb	24.20
retail	0.005%	3893.51 kb	21.56 kb	180.59	41.79 kb	93.17
T10I4D100k	0.01%	4336.83 kb	55.65 kb	77.93	103.95 kb	41.72
T40I10D100k	0.2%	15861.36 kb	625.63 kb	25.35	1251.18 kb	12.68

Table 4.1: Conditional database size

We have observed that in the SSP approach, it is almost impossible that a single conditional database is too large to be held in the memory given the large size of the main memory nowadays. The reason being that frequent items are sorted into ascending frequency order in the SSP approach, and the ascending frequency order ensures that a single conditional database is much smaller than the original database, and the size of its descendant conditional databases also shrinks quickly. Table 4.1 compares the size of the original database ("DBSize" column), the size of the largest conditional database ("MaxCDBSize" column) and the maximal space occupied by a conditional database and all of its descendant conditional databases ("MaxMemUsage" column). It shows that both the largest conditional database and the maximal space consumed by a conditional database and all of its descendant conditional databases are tens of times smaller

than the original database except on two very small dense datasets chess and mushroom. If it is the case that a single conditional database cannot be held in the memory, we can recursively apply the out-of-core mining algorithm on that conditional database.

Before constructing conditional databases from a database, we need to determine whether the memory is sufficient for holding all the conditional databases. A transaction can belong to multiple conditional databases depending on the number of frequent items in it. However, in our approach we put a transaction into only one conditional database at one time. Hence the space required for storing all the conditional databases cannot exceed the size of the projection of the original database with respect to F. Therefore, we use the size of the projection of the original database, which is $\sum_{t \in D}(\|t \bigcap F\| + 1)$ (The one additional storage unit is used to store the length of a transaction.), to determine whether the available memory is sufficient for holding all the conditional databases. Recall that we use three different structures to represent conditional databases. The size of the array structure can be accurately computed. The space required for the bucketing counting technique is a constant. It is very small and even can be neglected. It is difficult to estimate the size of the AFOPT structure. We can tune the parameter *tree_alphabet_size* to ensure that using the AFOPT structure is beneficial. Therefore, the size of the space required for storing all the conditional databases can be safely estimated as $\sum_{t \in D}(\|t \bigcap F\| + 1)$.

The pseudo-code of the SSP-naive algorithm is shown in Algorithm 3. When constructing new conditional databases from the original transaction database, we keep only the first conditional database in the memory and put all the others onto a disk (line 7-10). To reduce random I/O cost, we maintain an output buffer for each conditional database on the disk. When the mining on a conditional database in the memory is finished, the transactions in the conditional database should be pushed right to the remaining conditional databases. One optimization can be made here is that if a transaction belongs to the next conditional database to be processed, we keep the transaction in the memory (line 16-17).

We use the same example as in the previous chapter to illustrate the mining process of SSP-naive. The transaction database D is shown in Figure 4.1(a). The minimum support threshold is set to 40%. There are 6 items frequent in D

Algorithm 3 SSP-naive Algorithm

Input:
D is the original transaction database
min_sup is the minimum support threshold;

Description:
1: Scan D to count frequent items and sort them into ascending frequency order, F={a_1, a_2,$\cdots$, a_n};
2: **for all** item $a \in F$ **do**
3: $D_{\{a\}} = \phi$;
4: **for all** transaction $t \in D$ **do**
5: Remove infrequent items from t, and sort remaining items according to their orders in F;
6: Let a be the first item of t;
7: **if** $a = a_1$ **then**
8: Keep t in the memory and insert it to $D_{\{a_1\}}$.
9: **else**
10: Write t on the disk and put it into $D_{\{a\}}$;
11: **for all** item $a_j \in F$, from j=1 to n **do**
12: Output $s = \{a_j\}$;
13: SSP(s, D_s, *min_sup*);
14: **for all** transaction $t \in D_s$ **do**
15: $t = t - \{a_j\}$, let a' be the first item of t;
16: **if** a'=a_{j+1} **then**
17: Keep t in the memory and insert it into $D_{a_{j+1}}$;
18: **else**
19: Write t on the disk and put it into $D_{a'}$;
20: Load the part of $D_{a_{j+1}}$ on the disk into the memory;

and they are sorted into ascending frequency order, denoted as F={a:3, d:3, p:4, f:5, m:5, a:6}. The projected database is shown in Figure 4.1(b), in which every transaction is sorted according to F.

We make the following assumptions in the example: (1) The memory size is 50 storage units. (2) An entry in a header table requires 3 storage units to store an item, a count and a pointer pointing to the corresponding conditional database. (3) Every transaction t stored in a flat format, e.g. in the array structure or on the disk, requires $\|t\|$+1 storage units (one for each item and an additional one for the length of the transaction). (4) A node in the AFOPT structure requires five storage units to store a count, a child pointer, a right-sibling pointer, the number of items in the node and an item if the node contains only one item, or a pointer pointing to an array containing the items if the node contains more than one item. In other words, if an AFOPT node contains only one item, the AFOPT node requires 5 storage units. Otherwise, the AFOPT node requires 5+m storage units, where m ($m > 1$) is the number of items in the AFOPT node.

Before constructing the conditional databases, we estimate the size of the projected database as $\sum_{t \in D}(\|t \cap F\| + 1)$=33. We need to take the 18 storage units occupied by the header table into account. Hence the total space required

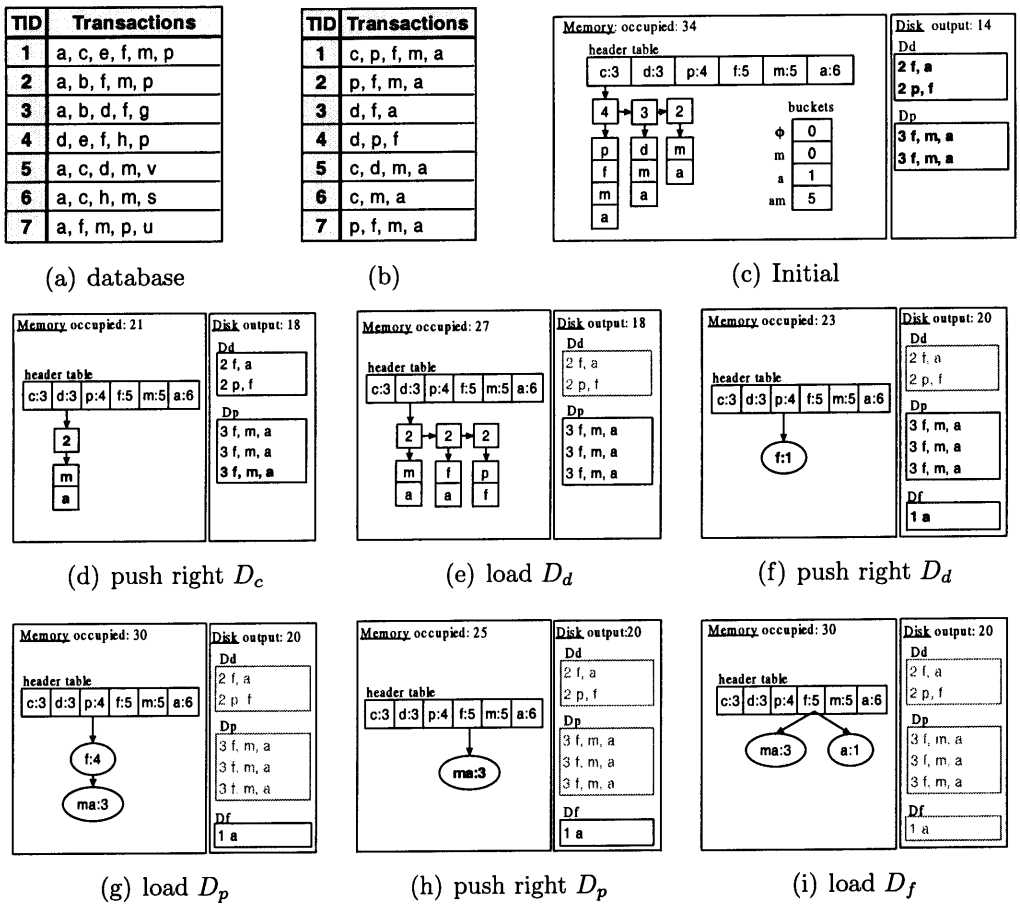


Figure 4.1: The mining process of SSP-naive

is 51, which is greater than the memory size. We have to perform out-of-core mining.

When constructing the conditional databases in a second database scan, we keep D_c in the memory and put all the other conditional databases onto the disk. We also keep the buckets in the memory and update their counts when scanning the database. The status of the main memory and the disk after the second database scan is shown in Figure 4.1(c): 34 storage units in the memory are occupied and 14 units are written on the disk. We first use the buckets to calculate the support of the itemsets containing m and a and release the buckets. Now 30 units in the memory remain occupied. Then we perform mining on D_c. Two items m and a are frequent in D_c. We use the bucket counting technique to get the support of their supersets as shown in Figure 3.2(a). When mining D_c, we need 6 units for the header table and 4 units for the buckets, which does not exceed the size of the available memory. After the mining on D_c is finished, 30

units are occupied in the memory.

Next we push right D_c. The first transaction in D_c belongs to D_p. Conditional database D_p is not the next conditional database to be processed, therefore the first transaction is written onto the disk. The second transaction in D_c should be inserted into D_d, which is the next conditional database to be mined, hence we keep this transaction in the memory. The third transaction is simply discarded because it starts with m, and all the itemsets starting with m have been counted using buckets. The status of the memory and the disk after pushing right D_c is shown in Figure 4.1(d).

Then we move on to mine D_d. A portion of D_d is on the disk. We first load that portion into the memory. The status of the memory and the disk after loading D_d is shown in Figure 4.1(e). No item is frequent in D_d. Then D_d is pushed right as follows: the first transaction in D_d is discarded, the second transaction belongs to D_f and it is written on the disk, and the third transaction is kept in the memory and inserted into D_p. The status of the main memory and the disk after pushing right D_d is shown in Figure 4.1(f).

Next we load D_p into the memory as shown in Figure 4.1(g). Conditional database D_p contains a single branch, we simply enumerate the itemsets contained in that single branch. No new conditional databases are constructed from D_p. Then D_p is pushed right. All the transactions in D_p belong to D_f—the next conditional database to be mined. They are kept in the memory and inserted into D_f as shown in Figure 4.1(h). Finally, we load D_f in the memory as shown in Figure 4.1(i). Two items m and a are frequent in D_f, and the itemsets containing these two items are counted using buckets. The mining is finished.

Lemma 3 *The upper bound of the cost for writing and loading conditional databases in Algorithm 3 is two times of the total size of the conditional databases, which is* $\sum_{t \in D} \|t \bigcap F\| \cdot (\|t \bigcap F\| + 1)$.

Proof 3 *A transaction t belongs to $\|t \bigcap F\|$ conditional databases. Its length is decreased by 1 when it is put into the next conditional database. So the total size of the conditional databases is* $\sum_{t \in D} \|t \bigcap F\| \cdot (\|t \bigcap F\| + 1)/2$. *Every transaction in a conditional database is written to and loaded from the disk at most once. Therefore, the I/O cost is at most two times of the total size of the conditional*

databases, which is $\sum_{t \in D} \|t \cap F\| \cdot (\|t \cap F\| + 1)$.

In Algorithm 3, we utilize data overlap between two adjacent conditional databases to reduce I/O cost. Hence the actual I/O cost for writing and loading conditional databases is usually much smaller than the upper bound. In the above example, the upper bound of the I/O cost for writing and loading conditional databases is 126, and the actual I/O cost of the SSP-naive algorithm is 40.

4.3 The SSP-static Algorithm

Algorithm SSP-naive keeps only one conditional database in the memory at one time. It is a waste of memory given the large amount of memory available nowadays. It is desirable to put as many conditional databases as possible into the memory to utilize overlap among all the conditional databases in the memory to reduce I/O cost. The question is how many conditional databases should be kept in the memory at one time to maximize the usage of the memory and minimize I/O cost. A simple method is to keep several adjacent conditional databases in the memory if their total size does not exceed the size of the available memory. The size of each conditional database can be obtained by one database scan. However this simple method does not consider overlap among conditional databases. If we keep conditional databases $D_{a_{i+1}}$, $\cdots$, $D_{a_{i+m}}$ in the memory, the space required for storing these m conditional databases is much smaller than the total size of these m conditional databases because of overlap among them.

To accurately calculate the borders between partitions, we use a matrix C, called *differential matrix*, to record the differences among conditional databases. The value of entry $C[i,j]$ $(j > i)$ is the total size of the transactions in D_{a_j} but not in D_{a_k}, where k=i, $\cdots$, j-1, i.e. $C[i,j]$=$\sum_{t \in (D_{a_j} - \bigcup_{k=i}^{j-1} D_{a_k})} \|t\|$. The value of entry $C[i,i]$ is the size of the conditional database D_{a_i}, i.e. $C[i,i]$=$\sum_{t \in D_{a_i}} \|t\|$. The rationale behind the differential matrix is that if we have kept D_{a_i}, $D_{a_{i+1}}$, $\cdots$, $D_{a_{j-1}}$ $(j > i)$ in the memory, then to load D_{a_j} into the memory, we only need to load those transactions in D_{a_j} that do not appear in any of the conditional databases already in the memory. Entry $C[i,j]$ records the total size of those transactions.

An example differential matrix is shown in Figure 4.2(b), which is computed

from the database shown in Figure 4.1(a) with minimum support of 40%. There are three transactions starting with the first item c, and their total size is 12. Hence the value of entry $C[1,1]$ is set to 12. Among all the transactions containing d, transaction 5 starts with item c. If we already keep D_c in the memory, then to load entire D_d into the memory, we need to load only the other two transactions—transaction 3 and transaction 4 into the memory. Therefore, the value of entry $C[1,2]$ is set to 6. Similarly, among all the transactions containing item p, transaction 1 starts with c and transaction 4 starts with d. If D_c and D_d are already in the memory, then we need to load only transaction 2 and transaction 7 into the memory to make D_p in the memory. Therefore, entry $C[1,3]$ is set to 8. If a partition starts with D_d, we also need to load transaction 1 into the memory to make entire D_p in the memory. Hence entry $C[2,3]$ is set to 12.

TID	Transactions
1	c, p, f, m, a
2	p, f, m, a
3	d, f, a
4	d, p, f
5	c, d, m, a
6	c, m, a
7	p, f, m, a

(a) projected db

		c	d	p	f	m	a
		1	2	3	4	5	6
c	1	12	6	8	0	0	0
d	2		9	12	0	2	0
p	3			14	2	4	0
f	4				11	4	0
m	5					10	0
a	6						0

(b) differential matrix

		c	d	p	f	m	a
		1	2	3	4	5	6
c	1	5	0	0	0	0	0
d	2		0	4	0	0	0
p	3			4	0	0	0
f	4				3	0	0
m	5					2	0
a	6						0

(c) 1st transaction

		c	d	p	f	m	a
		1	2	3	4	5	6
c	1	5	0	4	0	0	0
d	2		0	8	0	0	0
p	3			8	0	0	0
f	4				6	0	0
m	5					4	0
a	6						0

(d) 2nd transaction

Figure 4.2: Differential matrix

Lemma 4 *Given conditional databases $D_{a_{i+1}}$, $\cdots$, $D_{a_{i+m}}$, the space for storing these m conditional databases is $\sum_{k=i+1}^{i+m} C[i+1,k]$.*

Proof 4 $\sum_{t \in \bigcup_{k=i+1}^{i+m} D_{a_k}} \|t\| = \sum_{t \in D_{a_{i+1}}} \|t\| + \sum_{t \in (D_{a_{i+2}} - D_{a_{i+1}})} \|t\| + \cdots +$

$\sum_{t \in (D_{a_{i+m}} - \bigcup_{k=i+1}^{i+m-1} D_{a_k})} \|t\| = C[i+1,i+1] + C[i+1,i+2] + \cdots + C[i+1,i+m].$

The *differential matrix* can be calculated by one scan of the original database. The pseudo-code is shown in Algorithm 4. Function $Order(a)$ returns the order of item a in F. We use the above example to illustrate the computation process. Transaction 1 becomes $\{c,\ p,\ f,\ m,\ a\}$ after sorting according to F. The first item is c and its order is 1. We increase $C[1,1]$ by 5 (line 7). The second item is p and its order is 3. We increase $C[3,3]$ by 4 (line 7). Transaction 1 is not included in any conditional database between D_c and D_p. Therefore, if a partition starts with a conditional database between D_c and D_p and includes D_p,

transaction 1 should be counted as in D_p. Therefore, we increase all the entries $C[j,3]$ ($j \in (Order(a), Order(p))$=(1,3)) by 4 (line 12-13). The status of the differential matrix after processing transaction 1 is shown in Figure 4.2(c). Figure 4.2(d) shows the status of the differential matrix after processing transaction 2.

Algorithm 4 CalcDiffMatrix Algorithm

Input:
D is the original database
F is the set of frequent items in ascending frequency order
Output:
A differential matrix C
Description:
1: **for all** i from 1 to $\|F\|$ **do**
2: **for all** j from i to $\|F\|$ **do**
3: $C[i,j] = 0$;
4: **for all** transaction $t \in D$ **do**
5: Remove infrequent items from t, and sort remaining items according to their orders in F;
6: **for all** i from 1 to $\|t\| - 1$ **do**
7: $C[Order(t[i]), Order(t[i])]$ += $\|t\|$-i+1
8: **if** i=1 **then**
9: $k = 1$;
10: **else**
11: k = Order($t[i-1]$)+1;
12: **for all** j from k to Order($t[i]$)-1 **do**
13: $C[j, Order(t[i])]$ += $\|t\|$-i+1;

After the differential matrix C is computed, we then use it to partition the conditional databases as follows: Let a_i be the first item of a partition p. The last item a_m of partition p satisfies $m = max\{k \mid \sum_{j=i}^{j=k} C[i,j] + c_d \cdot max_{j \in [i,k]}\{C[j,j]\} \leq M\}$, where M is the size of the available memory and $c_d \cdot max_{j \in [i,k]}\{C[j,j]\}$ is the space reserved for mining the conditional databases in partition p. Given a conditional database D_{a_i}, the space needed for storing all its child conditional databases cannot exceed the size of D_{a_i}. In the SSP approach, frequent items are sorted into ascending frequency order. A single child conditional database of D_{a_i} is substantially smaller than D_{a_i}, so does the space required for mining a single child conditional database of D_{a_i}. We preserve $c_d \cdot \sum_{t \in D_{a_i}} \|t\| = c_d \cdot C[i,i]$ space to store descendant conditional databases of D_{a_i}, where c_d is a tunable parameter. In our experiments, we found that with c_d=1, the space preserved was more than enough in most cases. The pseudo-code for partitioning the conditional databases is given in Algorithm 5. For a partition p, we use $p.start$ and $p.end$ to denote the order of the first and the last conditional database of p in the header table.

We use the example shown in Figure 4.2 to illustrate the static partition process. We set c_d=1. The space required for storing the first two conditional

Algorithm 5 StaticPartition Algorithm

Input:
C is the differential matrix
M is the size of the available memory
Output:
partitions P
Description:
1: $P = \phi$;
2: $start = 0$; $max\_cdb\_size = 0$; $part\_size = 0$;
3: **for all** i from 1 to $\|F\|$ **do**
4: **if** $max\_cdb\_size < C[i,i]$ **then**
5: $max\_cdb\_size = C[i,i]$;
6: **if** $part\_size + C[start,i] + c_d \cdot max\_cdb\_size > M$ **then**
7: Create a new partition p;
8: $p.start = start$; $p.end = i$-1;
9: $P = P \bigcup \{p\}$;
10: $start = i$; $part\_size = C[i,i]$; $max\_cdb\_size = C[i,i]$;
11: **else**
12: $part\_size$ += $C[start,i]$;

databases in the memory is 18 and the maximal size of the two conditional databases is 12. Hence the estimated memory usage for mining the first two conditional databases is 30, which does not exceed the total memory size after taking the 18 units for the header table into account. However, the estimated memory usage for the first three conditional databases is 40 (26 for storing the three conditional databases and 14 reserved for mining them), which exceeds the size of the memory after taking the 18 units for the header table into consideration. Therefore, the first partition contains D_c and D_d. It is not difficult to find that the remaining two conditional databases D_p and D_f should be in the same partition.

The pseudo-code of the SSP-static algorithm is shown in Algorithm 6. When constructing conditional databases from the original database, the SSP-static algorithm keeps all the conditional databases in the first partition in the memory (line 9-10) and writes all the others onto the disk (line 11-12). After the partition in the memory is processed, the next partition is loaded into the memory. In the push-right step, the SSP-static algorithm utilizes overlap between adjacent partitions to reduce I/O cost. More specifically, if a transaction belongs to a conditional database in the current partition or in the next partition, the transaction is kept in the memory instead of writing onto the disk (line 19-20).

We use the example shown in Figure 4.1 to illustrate the mining process of SSP-static. According to Algorithm 5, the conditional databases are divided into two partitions: $p_1 = \{D_c, D_d\}$ and $p_2 = \{D_p, D_f\}$. When we construct

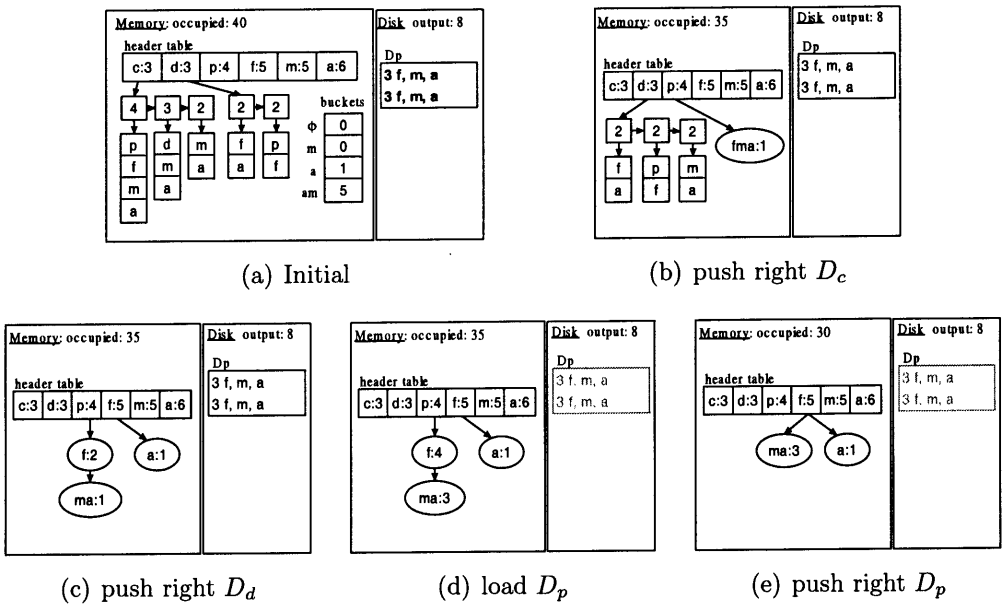
Algorithm 6 SSP-static Algorithm

Input:
D is the original transaction database
min_sup is the minimum support threshold;

Description:
1: Scan D to count frequent items and sort them into ascending frequency order, F=$\{a_1, a_2, \cdots, a_n\}$;
2: **for all** item $a \in F$ **do**
3: $D_{\{a\}} = \phi$;
4: CalcDiffMatrix(D, C);
5: StaticPartition(C, P), where $P = \{p_1, p_2, \cdots, p_m\}$;
6: **for all** transaction $t \in D$ **do**
7: Remove infrequent items from t, and sort remaining items according to their orders in F;
8: Let a be the first item of t;
9: **if** $p_1.start \leq a \leq p_1.end$ **then**
10: Keep t in the memory and insert it into $D_{\{a\}}$.
11: **else**
12: Write t to partition p such that $p.start \leq a \leq p.end$;
13: **for all** i from 1 to m **do**
14: **for all** j from $p_i.start$ to $p_i.end$ **do**
15: Output $s = \{a_j\}$;
16: SSP(s, D_s, *min_sup*);
17: **for all** transaction $t \in D_s$ **do**
18: $t = t - \{a_j\}$, let $a_{j'}$ be the first item of t;
19: **if** $p_i.start < j' \leq p_{i+1}.end$ **then**
20: Keep t in the memory and insert it into $D_{a_{j'}}$;
21: **else**
22: Write t to p such that $p.start \leq j' \leq p.end$;
23: Load p_{i+1} into the memory;

conditional databases from the original database, we keep D_c and D_d in the memory and put the other two conditional databases onto the disk as shown in Figure 4.3(a). When pushing right D_c, we keep transactions starting with d in the memory because they belong to the current partition (Figure 4.3(b)). We also keep transactions starting with p or f in the memory when pushing right D_c and D_d because they belong to the next partition to be mined as shown in Figure 4.3(b) and Figure 4.3(c). The total I/O cost is 16.

In Algorithm 6, the upper bound of the I/O cost for writing and reading conditional databases is the same as that of Algorithm 3. Algorithm 6 utilizes overlap among partitions to reduce I/O cost. Overlap among partitions is more significant than overlap among individual conditional databases. Therefore, Algorithm 6 requires much less I/O cost for reading and writing conditional databases than Algorithm 3 in practice. Algorithm 6 needs an additional database scan to calculate the differential matrix. To reduce the additional I/O cost, if the projection of the original database is much smaller than the original database, we write the projection of the original database on the disk when calculating the differential



(a) Initial (b) push right D_c

(c) push right D_d (d) load D_p (e) push right D_p

Figure 4.3: The mining process of SSP-static

matrix. Then we scan the projected database instead of the original database to construct the conditional databases.

4.4 The SSP-dynamic Algorithm

Algorithm 6 requires extra I/O cost to partition the database. Furthermore, it uses the total length of the transactions in a conditional database as the estimation of the size of the conditional database. When a conditional database is represented by an array structure, the above estimation is accurate. However if a conditional database is very dense and it is represented by a tree structure, the estimation is a rather loose upper bound. In the SSP approach, the conditional databases of the most frequent items except for those counted using buckets are represented using the AFOPT structure. Table 4.2 shows the space required for storing the conditional databases of the 50 most frequent items using the array structure and the AFOPT structure respectively. The last column shows the compression ratio of the AFOPT structure. The datasets in Table 4.2 are relatively small. On very large databases with millions of transactions, the compression ratio can be much higher.

Algorithm 6 cannot fully utilize the available memory when the compression ratio of the AFOPT structure is high, which is often the case. To overcome

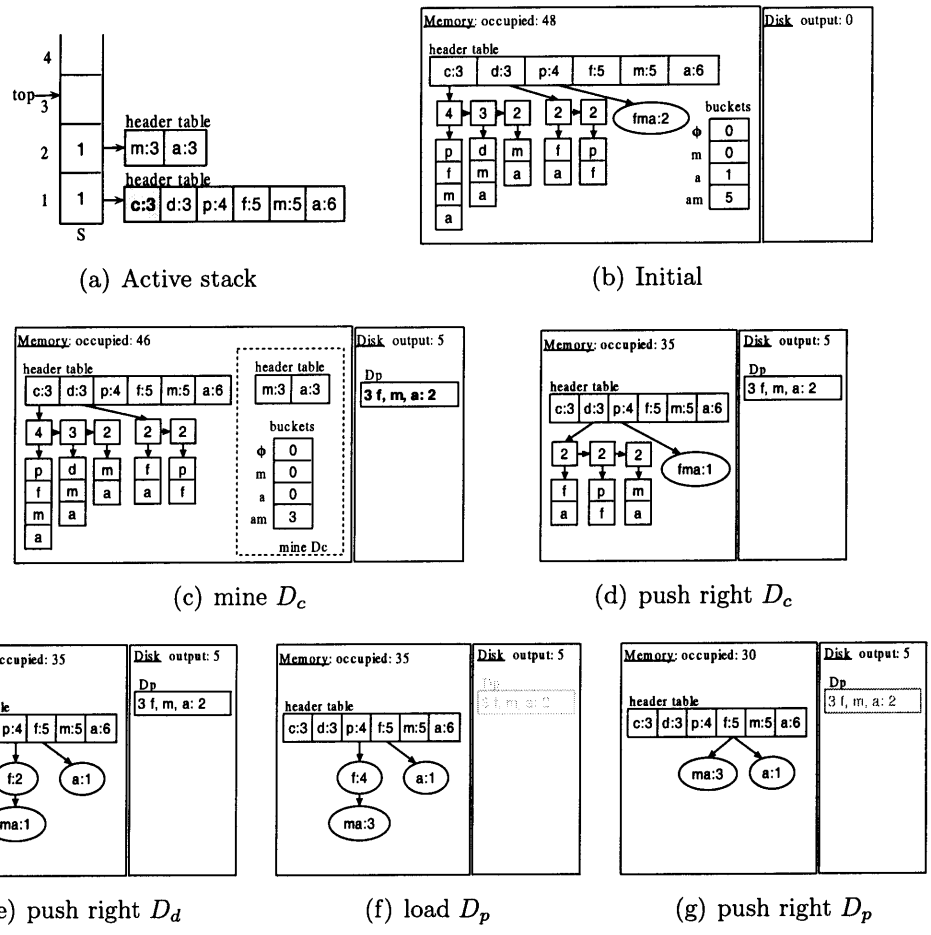
Datasets	Array	Prefix-trie	Ratio
BMS-POS	7.24 MB	6.28 MB	1.15
BMS-WebView-1	0.2 MB	0.15 MB	1.33
chess	0.45 MB	0.36 MB	1.25
connect-4	10.57 MB	2.79 MB	3.79
mushroom	0.64 MB	0.34 MB	1.88
pumsb	7.96 MB	0.99 MB	8.04
pumsb_star	5.44 MB	0.73 MB	7.45
retail	0.79 MB	0.42 MB	1.88

Table 4.2: Prefix-trie compression ratio

the drawbacks of the SSP-static algorithm, we propose another algorithm SSP-dynamic. The SSP-dynamic algorithm uses a dynamic strategy. It writes conditional databases on a disk only when new structures are to be created but there is no free memory.

The SSP approach traverses the search space in depth-first order. The exact access order of the conditional databases is known. We exploit this information to put the conditional databases that are accessed last onto the disk to release memory for new data. We maintain an *active stack* S which traces the conditional databases that have not been processed yet at every level. Entry $S[k]$ stores two pieces of information: a pointer pointing to the header table of level k, denoted as $S[k].header\_table$, and the position of the *active* conditional database of level k in the header table, denoted as $S[k].pos$. A conditional database D_l is said to be *active* if either D_l itself is being processed or one of D_l's descendant conditional databases is being processed. Since we traverse the search space in depth-first order, there is one and only one active conditional database at each level. We use the transaction database shown in Figure 4.1(a) to illustrate the active stack. With minimum support of 40%, Figure 4.4(a) shows the status of the active stack when mining conditional database D_c. The active conditional database at level 1 is D_c and its has two frequent extensions m and a.

In depth-first order, a conditional database at a lower level is always accessed after a conditional database at a higher level if the former is after the ancestor of the latter. Conditional databases at the same level are accessed according to their positions in the header table. For example, the access order of the conditional databases at level 1 in Figure 4.4(a) is as follows: D_c, D_d, D_p, D_f, D_m and D_a. Therefore, when releasing memory for new data, we start from the lowest level.



(a) Active stack (b) Initial

(c) mine D_c (d) push right D_c

(e) push right D_d (f) load D_p (g) push right D_p

Figure 4.4: The mining process of SSP-dynamic

At each level, we start from the last conditional database and move backward. The pseudo-code for releasing memory is given in Algorithm 7.

Algorithm 7 is called in two situations: (1) new conditional databases are constructed from a database, or (2) a conditional database is loaded into the memory. Calling Algorithm 7 for every transaction to be inserted is very inefficient. Therefore, when calling Algorithm 7, we estimate the space required for future mining. If we are constructing new conditional databases from a conditional database D_l, we use $(S_{D_l} - R_{D_l}) \cdot M_{new}/R_{D_l}$ as the estimation, where M_{new} is the size of the memory that has been occupied by the new conditional databases, S_{D_l} is the size of D_l and R_{D_l} is the size of the portion of D_l that has been scanned for constructing the new conditional databases. If we are loading a conditional database from the disk, we use $M_{D_l} \cdot D_{D_l}/(\sum_{t \in D_l} \|t\| - D_{D_l})$ as the estimation, where M_{D_l} is the size of the memory that has been occupied by D_l, and D_{D_l} is the size of the portion of D_l on the disk.

Algorithm 7 ReleaseMemory Algorithm

Input:
S is the active stack
M is the required memory size
Description:
1: $m = 0$;
2: **for all** i from 1 to $S.top$-1 **do**
3: **for all** j from $\|S[i].header\_table\|$ to $S[i].pos$+1 **do**
4: m += $\sum_{t \in S[i].header\_table[j].cond\_db} \|t\|$;
5: write $S[i].header\_table[j].cond\_db$ to disk;
6: **if** $m \geq M$ **then**
7: **return** ;

Figure 4.4 shows the mining process of SSP-dynamic on the database shown in Figure 4.1(a) with minimum support of 40%. Initially all the conditional databases are kept in the memory as shown in Figure 4.4(b) because their actual size does not exceed the size of the main memory. We first process the buckets to count the support of the itemsets containing m and a, and then release the buckets. The memory usage becomes 44. Next we perform mining on D_c. Two items m and a are frequent in D_c and their supersets should be counted using the bucket count technique. The space required for mining D_c is 10 storage units, i.e. 6 units for the header table and 4 units for the buckets. However, there is no enough space in the memory to hold both the new header table and the buckets. We have to release some space by calling Algorithm 7 with M=4. When releasing memory, we start from the last conditional database at level one and move backward. Conditional database D_p is the first non-empty conditional database, and it is written onto the disk as shown in Figure 4.4(c). When writing tree branches to the disk, we use an extra unit to store the count of the branch instead of writing the branch multiple times. After putting D_p on the disk, the memory usage becomes 36. The available memory is sufficient to hold the structures for mining D_c. Then we move on to mine the remaining conditional databases. The memory is sufficient for the rest of the mining as shown in Figure 4.4(d)-4.4(g).

The pseudo-code of the SSP-dynamic algorithm is shown in Algorithm 8. Algorithm 8 uses a lazy-writing strategy, i.e., it writes data on the disk only when it has to. Therefore, Algorithm 8 can guarantee full utilization of the memory. Both Algorithm 6 and Algorithm 8 need to estimate the size of the memory required for future mining. Algorithm 6 has to guarantee that the size of the memory preserved is sufficient for future mining. Therefore, it might waste

Algorithm 8 SSP-dynamic Algorithm

Input:
l is a frequent itemset
D_l is the conditional database of l
min_sup is the minimum support threshold;

Description:
1: Scan D_l to count frequent items and sort them into ascending frequency order, F=$\{a_1, a_2, \cdots, a_n\}$;
2: **for all** item $a \in F$ **do**
3: $D_{l \bigcup \{a\}} = \phi$;
4: **for all** transaction $t \in D_l$ **do**
5: Remove infrequent items from t, and sort remaining items according to their orders in F;
6: Let a be the first item of t;
7: **if** no free memory **then**
8: ReleaseMemory(S, $(S_{D_l} - R_{D_l}) \cdot M_{new}/R_{D_l}$);
9: Insert t into $D_{l \bigcup \{a\}}$;
10: **for all** item $a_j \in F$, from j=1 to n **do**
11: Output $s = l \bigcup \{a_j\}$;
12: SSP(s, D_s, *min_sup*);
13: **for all** transaction $t \in D_s$ **do**
14: $t = t - \{a\}$, let a' be the first item of t;
15: Keep t in the memory and insert it into $D_{l \bigcup \{a'\}}$;
16: **for all** $t \in D_{l \bigcup \{a_{j+1}\}}$ on disk **do**
17: **if** no free memory **then**
18: ReleaseMemory(S, $M_{D_l} \cdot D_{D_l}$ / $(\sum_{t \in D_{l \bigcup \{a_{j+1}\}}} \|t\| - D_{D_l})$);
19: Insert t into $D_{l \bigcup \{a_{j+1}\}}$;

memory in some cases. In Algorithm 8, if the size of the memory preserved is not sufficient, the only effect is that Algorithm 7 is called again to release memory.

4.5 A Performance Study

We compare the SSP approach with the kDCI algorithm [64] because of its good scalability shown in a recent comprehensive comparative study of frequent itemset mining implementations [28]. The kDCI algorithm adopts a classical level-wise approach based on the candidate generation-and-test approach, but it uses a hybrid method to determine the support of the candidate itemsets. In the initial count-based phase, the kDCI algorithm scans the dataset multiple times to count support as the Apriori algorithm [5]. At each iteration, the dataset is pruned based on frequent itemsets, and a temporary dataset in a horizontal format is written to the disk as in the DHP algorithm [68]. As soon as the pruned dataset fits into the main memory, the kDCI algorithm switches to the second phase. In the second intersection-based phase, the kDCI algorithm stores the dataset in the main memory in a vertical format, and the support of the candidate itemsets is

Datasets	Size	#Trans	#Items	MaxTL	AvgTL
T10I4D5mN10kP10k	238MB	4,922,589	7,692	36	10.14
T40I10D2mN10kP10k	374MB	1,999,994	8,912	84	39.95
T20I15D10mN10kD10k	943MB	9,065,540	9,246	67	22.09
webdocs	1,413MB	1,692,082	5,267,656	71,473	177.23

Table 4.3: Datasets

counted by tid bit vector intersections. Thus kDCI does not need to scan datasets as many times as Apriori, but it may still suffer from the explosion in the number of k-candidates.

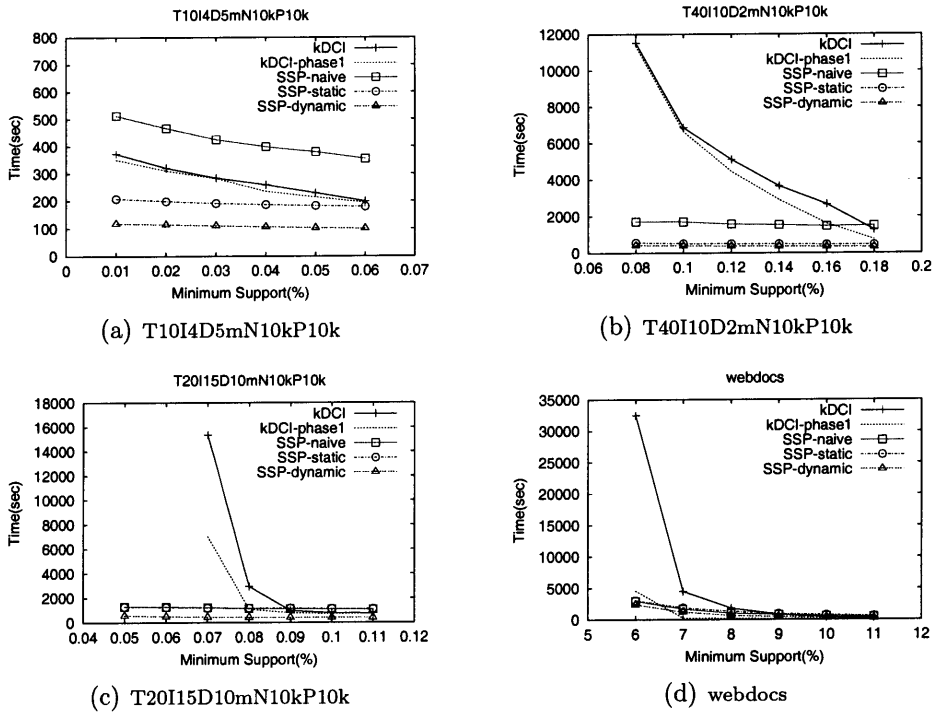
We conducted the experiments on both synthetic datasets and real-world datasets. The synthetic datasets were generated using IBM Quest Synthetic Data Generation Code [1]. The detailed dataset generation process and the description of the parameters used in the generation process can be found in [5]. The experiments were conducted on a PC with 1Ghz Pentium III and 256MB memory running Mandrake Linux.

4.5.1 Varying the minimum support threshold

The first experiment is to show the scalability of the algorithms with respect to the minimum support threshold. We used three synthetic datasets and one real-world datasets. Some statistical information of the datasets is shown in Table 4.3, including the size of the dataset, the number of transactions, the number of distinct items, the average transaction length and the maximal transaction length. Dataset webdocs is provided by the authors of the kDCI algorithm. It was built from a collection of Web html documents. More details about the dataset can be found at http://fimi.cs.helsinki.fi/data/webdocs.pdf.

Figure 4.5 shows the running time of the four algorithms when varying the minimum support threshold. It also shows the running time of the first phase of kDCI (denoted by "kDCI-phase1"). The running time of the kDCI algorithm increases rapidly with the decrease of the minimum support threshold except on the sparse dataset T10I4D5mN10kP10k. We have investigated the reasons and find that on dataset T40I10D2mN10kP10k, most of the time is spent on the first

[1] http://www.almaden.ibm.com/software/quest/Resources/datasets/syndata.htm



(a) T10I4D5mN10kP10k

(b) T40I10D2mN10kP10k

(c) T20I15D10mN10kP10k

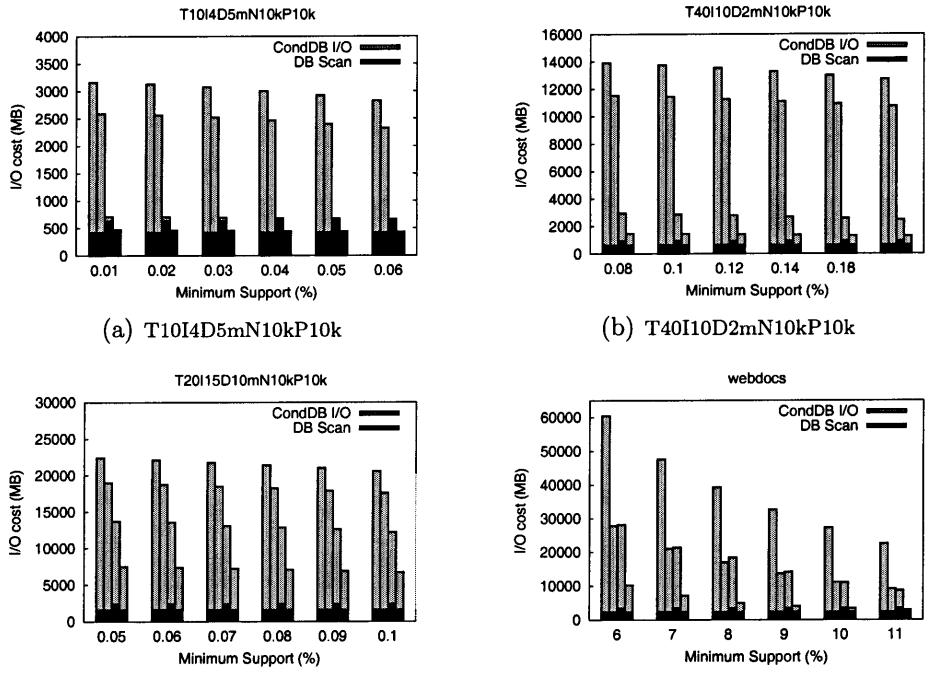
(d) webdocs

Figure 4.5: Varying the minimum support threshold: running time

phase. The kDCI algorithm scans the dataset more than 10 times when minimum support threshold is less than 0.14%. When the minimum support threshold is 0.08%, the number of database scans reaches 14. Reading and writing the pruned datasets many times incurs high I/O cost. The costly subset matching operation also incurs high CPU cost. On dataset T20I15D10mN10kP10k and the real dataset webdocs, the second phase becomes the bottleneck of the algorithm when minimum support threshold is low. The reason being that in the second phase, kDCI uses a k-way intersection method instead of the 2-way intersection method adopted by the traditional vertical mining algorithms. Therefore, the kDCI algorithm needs to perform much more tid bitmap intersections than the traditional vertical mining algorithms. Although kDCI uses a caching strategy to reduce the number of tid bitmap intersections, the number of tid bitmap intersections actually carried out is still much more than that of using the 2-way intersection method.

The three SSP algorithms show stable performance with respect to the minimum support threshold, and SSP-dynamic shows the best performance. The Apriori algorithm [5] is often used as baseline for studying frequent itemset mining algorithms. We tested the Apriori algorithm implemented by Christian Borgelt on

the four datasets. It takes 27465 seconds on T10I4D5mN10kP10k with minimum support of 0.06%, and tens of hours on the other datasets.



(a) T10I4D5mN10kP10k (b) T40I10D2mN10kP10k

(c) T20I15D10mN10kP10k (d) webdocs

Figure 4.6: Varying the minimum support threshold: I/O cost

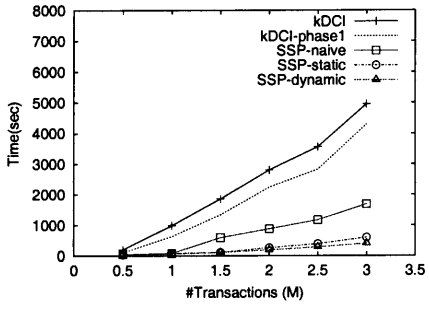
We compared the I/O cost of the three SSP algorithms. Figure 4.6 shows their I/O cost for scanning the original database (denoted as "DB Scan") and for writing and reading the conditional databases (denoted as "CondDB I/O") on four datasets. From left to right, the bars in Figure 4.6 represent the upper bound, the I/O cost of SSP-naive, SSP-static and SSP-dynamic respectively. SSP-naive and SSP-dynamic scan the original database twice. The SSP-static algorithm needs one more database scan. The upper bound of the I/O cost for writing and reading conditional databases is two times of the total size of the conditional databases. The SSP-naive algorithm reduces the conditional database I/O cost by utilizing overlap between two adjacent conditional databases. The average reduction ratio achieved by it is 20.8%, 17.2%, 16.0% and 63.3% respectively on four datasets. SSP-static can further achieve a reduction of 97.2%, 83.1%, 37.2% and 15.8% respectively on the basis of SSP-naive. The conditional database I/O cost of SSP-dynamic is about half of that of SSP-static on three synthetic datasets, and is 16.9% of that of SSP-static on the real dataset webdocs.

4.5.2 Varying the dataset generating parameters

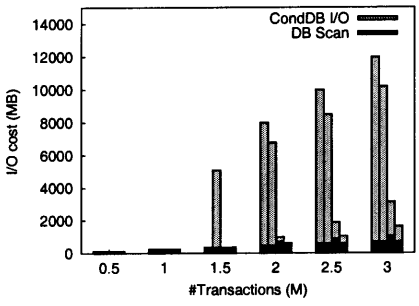
The second experiment is to study the scalability of the algorithms with respect to dataset characteristics, e.g. the size and the density of the datasets. We perturbed the dataset generating parameters along several dimensions: the number of transactions (*-ntrans* parameter), the average size of the transactions (*-tlen* parameter), the average size of the maximal potentially large itemsets (*-patlen* parameter), the number of maximal potentially large itemsets (*-npats* parameter) and the number of distinct items (*-nitems* parameter). The size of the datasets is mainly decided by the first two parameters. The density of the datasets is mainly decided by the last three parameters and the "-tlen" parameter. The detailed description of the parameters can be found in [5]. For all the other parameters, we use their default values. When we vary a parameter, all the other four parameters are set to fixed values. The fixed values for the five parameters are 2500, 30, 10, 10 and 10000 respectively. The minimum support threshold is set to 0.1%. Figure 4.7 shows the running time of the four algorithms and the I/O cost of the three SSP algorithms.

The kDCI algorithm shows good scalability with respect to the number of items, but its running time increases rapidly when varying the other four parameters, especially with respect to the *patlen* parameter and the *npats* parameter. The reason being that the number of frequent itemsets increases sharply with the increase of the *patlen* parameter and the decrease of the *npats* parameter. The kDCI algorithm needs to generate and test a large number of candidate itemsets, which incurs both tremendous CPU cost and high I/O cost. When npats=5k, the kDCI algorithm needs to scan the dataset 14 times in the first phase. When patlen=20, the number of candidate itemsets becomes very large and even causes page swap, which makes kDCI cannot terminate within a reasonable period.

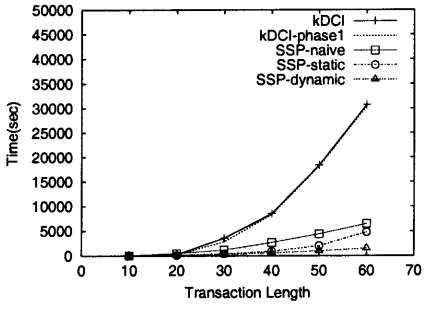
The running time of the three SSP algorithms increases with the increase of the number of transactions and the average transaction length, and is relatively steady with respect to the other three parameters. So does the I/O cost of the three algorithms. The reason being that the I/O cost of the SSP algorithms depends on the total size of the conditional databases constructed from the original database. It is proportional to the number of transaction, quadratic to the average transaction length in the worst case and irrelevant to the other



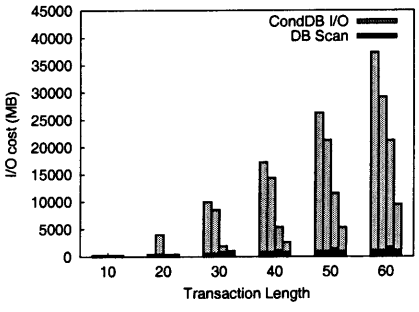
(a) ntrans: time



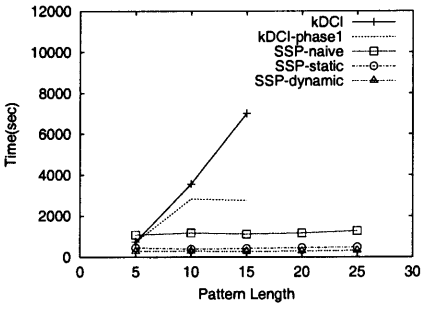
(b) ntrans: I/O cost



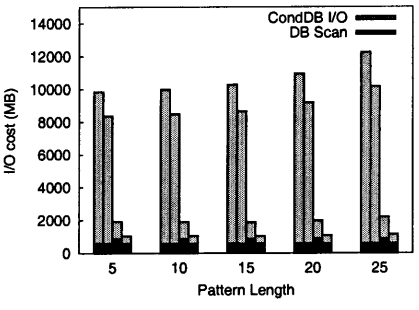
(c) tlen: time



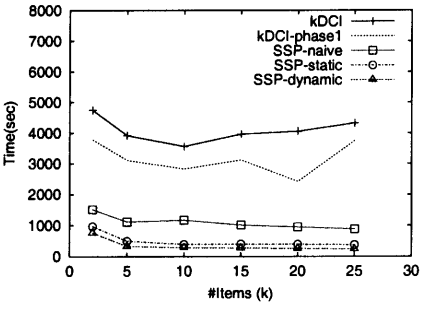
(d) tlen: I/O cost



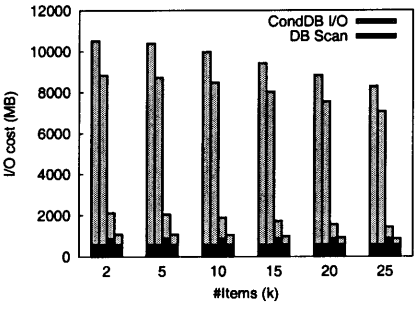
(e) patlen: time



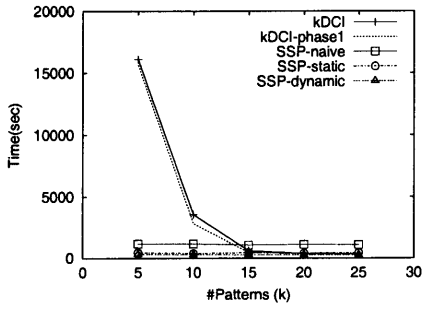
(f) patlen: I/O cost



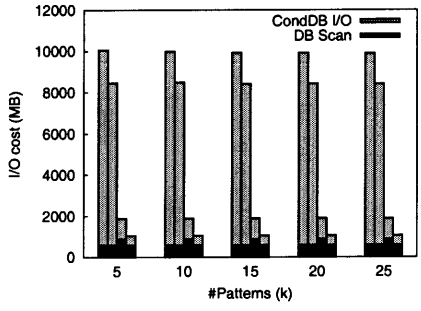
(g) nitems: time



(h) nitems: I/O cost



(i) npats: time



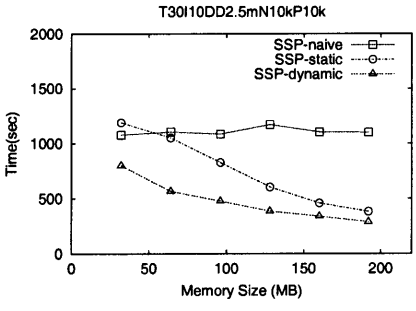
(j) npats: I/O cost

Figure 4.7: Varying dataset generating parameters

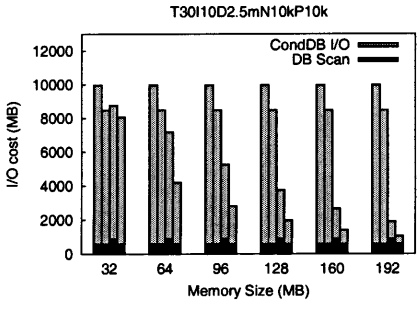
parameters.

4.5.3 Varying the size of the memory

The third experiment is to study the sensitivity of the three SSP algorithms with respect to the size of the memory. The dataset is generated with *ntrans*=2500, *tlen*=30, *patlen*=10, *nitems*=10 and *npat*=10000. The minimum support threshold is set to 0.1%.



(a) memory size: time



(b) memory size: I/O cost

Figure 4.8: Verying memory size

Figure 4.8 shows the running time and the I/O cost of the three algorithms respectively. The running time of SSP-naive is almost constant with respect to the memory size. The reason being that SSP-naive holds only one conditional database in the memory at one time, and 32MB is enough for a single conditional database. Both SSP-static and SSP-dynamic can effectively utilize the available memory to reduce I/O cost. Their running time and I/O cost decreases rapidly with the increase of the memory size. The SSP-dynamic algorithm can take advantage of the AFOPT structure to further reduce I/O cost. When the memory

size is set to 32MB, the running time of SSP-static exceeds that of the SSP-naive algorithm. The reason being that when the memory size is set to 32MB, the conditional database I/O cost of the SSP-static algorithm is close to that of the SSP-naive algorithm, but SSP-static requires one more database scan to compute the differential matrix.

4.6 Summary

In this chapter, we propose an efficient approach SSP to out-of-core mining of frequent itemsets from very large databases.

The SSP approach partitions the database according to the search space of the frequent itemset mining problem, which is different from the data-based partitioning approach. One salient feature of the SSP approach is that there is no need to maintain the local frequent itemsets and scan the original database to verify their support as in the data-based partitioning algorithms. Therefore, the SSP approach is very suitable for the situation where the database is dense and/or the minimum support threshold is low.

In our experiments, we have observed that even the SSP approach is much more efficient than previous algorithms. It may still take thousands of seconds to finish the mining on some very large datasets. It is unacceptable for applications that have to support online interactive mining. In the next chapter, we introduce a disk-based structure to store frequent itemsets on a disk. Online mining requests can be answered efficiently by retrieving itemsets from the structure.

CHAPTER 5

CFP-TREE: STORING AND QUERYING FREQUENT ITEMSETS

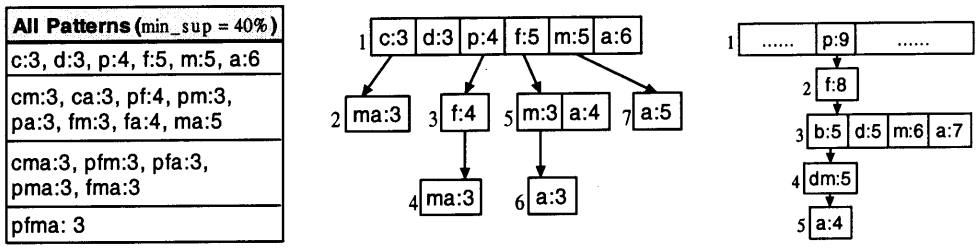
Frequent itemset mining is a time-consuming process due to its intrinsic characteristics. Algorithms that adopt the "mining on demand" strategy cannot answer mining queries in real time, especially on large databases. It is therefore desirable to adopt a "mining once and using many times" strategy to support online interactive mining. In this chapter, we propose a disk-based data structure, called Condensed Frequent Pattern tree, to organize frequent itemsets on a disk, and develop efficient algorithms to retrieve frequent itemsets satisfying user-specified constraints from the precomputed frequent itemsets. Efficient algorithms for constructing and incrementally maintaining a CFP-tree are also proposed.

5.1 The CFP-tree Structure

The CFP-tree structure is a very compact structure for storing frequent itemsets. In this section, we first describe the CFP-tree structure and show how space can be saved by prefix sharing and suffix sharing when storing all frequent itemsets. The CFP-tree structure can also be used to store frequent closed itemsets. To simplify presentation, we use CFP-tree$_{all}$ to denote a CFP-tree storing all frequent itemsets and use CFP-tree$_{closed}$ to denote a CFP-tree storing only frequent closed itemsets.

5.1.1 Storing all frequent itemsets

The CFP-tree structure is a prefix-trie like structure. Figure 5.1(a) shows the complete set of frequent itemsets mined from the database shown in Figure 1.1(a) with minimum support of 40%. The CFP-tree storing these frequent itemsets is shown in Figure 5.1(b). Each node in a CFP-tree is a variable-length array. All entries in a node are sorted into ascending frequency order. An entry can contain multiple items if it is the only child of its parent, e.g. node 2 in Figure 5.1(b)



(a) all itemsets (b) CFP-tree: all (c) Suffix share

Figure 5.1: CFP-tree: storing all frequent itemsets

contains two items m and a. A path starting from an entry in the root node represents one or more frequent itemsets. Let E be an entry and l be an itemset represented by the path from root to E. Entry E stores four pieces of information: (1) m items ($m \geq 1$), (2) the count of l, (3) a pointer pointing to the child node of E, and (4) a hash bitmap which is used to indicate whether a specific item appears in the subtree pointed by E. In the rest of this chapter, for an entry E in a CFP-tree node, we use $E.items$ to denote the set of items in E, $E.support$ to denote E's count, $E.child$ to denote the child pointer of E and $E.bitmap$ to denote E's hash bitmap.

When storing all frequent itemsets, the CFP-tree structure allows different itemsets to share the storage of their prefixes and suffixes. This feature makes the CFP-tree structure a very compact structure for storing a complete set of frequent itemsets. Prefix sharing is easy to understand. For example, itemsets $\{c, m\}$, $\{c, a\}$ and $\{c, m, a\}$ share the same prefix c in Figure 5.1(b). Suffix sharing is not as obvious as prefix sharing. We explain it using the search space tree shown in Figure 2.1.

Definition 6 (Suffix Sharing) *Let l and s be two itemsets and $s \subseteq cand\_exts(l)$. If $\forall q \subseteq cand\_exts(l)$ such that $q \neq \phi$, $q \bigcap l = \phi$ and $q \bigcap s = \phi$, we have $support(l \bigcup q) = support\ (l \bigcup s \bigcup q)$, then we say that l and $l \bigcup s$ share suffixes.*

Lemma 5 *Let l and s be two itemsets and $s \subseteq cand\_exts(l)$. If s is contained in every transaction t such that $l \subseteq t$ and $t \bigcap cand\_exts(l) \neq \phi$, then l and $l \bigcup s$ share suffixes.*

Proof 5 *Let q be any itemset such that $q \subseteq cand\_exts(l)$, $q \neq \phi$, $q \bigcap l = \phi$ and $q \bigcap s = \phi$. Let t be any transaction containing $l \bigcup q$, we have $l \subseteq t$ and*

$t \bigcap cand\_exts(l) \neq \phi$. According to the condition of the lemma, t must contain s. Therefore, support($l \bigcup q$)=support($l \bigcup s \bigcup q$). According to the definition of suffix sharing, l and $l \bigcup s$ share suffixes.

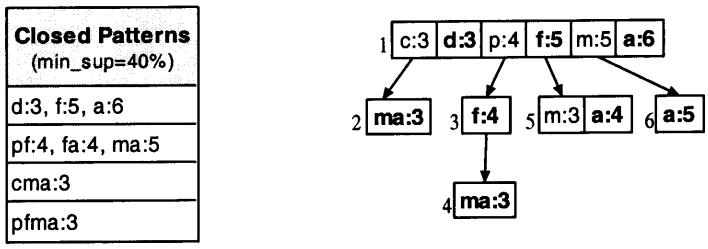
In Figure 5.1(b), itemset $\{f\}$ appears in every transaction containing itemset $\{p\}$. Therefore, itemsets $\{p\}$ and $\{p, f\}$ share suffixes, which is node 4, as shown in Figure 5.1(b).

Suffix sharing is implemented by singleton nodes in a CFP-tree. More specifically, if itemsets l and $l \bigcup s (s \subseteq cand\_exts(l))$ share suffixes, then the child node of l contains only one entry and the items of the entry is s. For example, the child node of $\{p\}$ is a singleton node containing item f, and the child node of $\{c\}$ is a singleton node containing items m and a. If there are more than one singleton node on a path, then the space saved by suffix sharing multiplies. It is formally stated as follows.

Lemma 6 *Let cnode be a CFP-tree node and $snode_1$, $snode_2$, $\cdots$, $snode_k$ be the singleton nodes on the path from root to cnode. Let m_i be the number of items contained in $snode_i$, i=1, 2, $\cdots$, k. Then cnode is shared by $2^{m_1+m_2+\cdots+m_k}$ itemsets.*

We use an example to illustrate the multiplication of suffix sharing. Figure 5.1(c) shows a branch in a CFP-$tree_{all}$. The branch contains two singleton internal nodes. Node 2 contains one item f. Node 4 contains two items. According to the above lemma, node 5 are shared by $2^3 = 8$ itemsets and these itemsets are *pfb*, *pfbd*, *pfbm*, *pfbdm*, *pb*, *pbd*, *pbm* and *pbdm*.

In the CFP-tree structure, all the children of a node are stored in the same node represented by a variable length array. This representation method has several advantages: (1) It saves space. No matter how many children a node has, only one child pointer is needed. (2) It can guide the search when processing queries with item constraints. The items appearing in the child node of an entry E (frequent extensions of E) is a subset of the items appearing in the entries after E (candidate extensions of E). Therefore, when processing queries with item constraints, before searching the child node of an entry E, we can first check whether all the items to be searched are contained in the entries after E.



(a) closed

(b) CFP-tree: closed

Figure 5.2: CFP-tree: storing frequent closed itemsets

5.1.2 Storing frequent closed itemsets

It has been observed that a complete set of frequent itemsets contains too much redundancy, especially when long frequent itemsets exist. Some researchers have proposed to mine only frequent closed itemsets to reduce output size [69, 102, 93]. A frequent itemset is closed if all of its proper supersets are less frequent than it. A set of frequent closed itemsets is the most concise representation of the corresponding set of all frequent itemsets without information loss. It can be significantly smaller than the corresponding set of all frequent itemsets. Instead of storing a complete set of frequent itemsets in a CFP-tree, we can store only frequent closed itemsets in a CFP-tree. Figure 5.2(a) shows the set of frequent closed itemsets mined from the database shown in Figure 1.1(a) with minimum support of 40%. The corresponding CFP-tree is shown in Figure 5.2(b). Some entries in Figure 5.2(b), e.g. entry c, p and m in node 1, do not represent frequent closed itemsets, but they are still kept in the CFP-tree because they are prefixes of some closed itemsets. The entries representing closed itemsets are highlighted in bold in Figure 5.2(b).

When storing frequent closed itemsets, a CFP-tree allows only prefix sharing. The reason being that if l and $l \bigcup s$ ($s \subseteq freq\_exts(l)$) share suffixes, then $\forall q \subseteq cand\_exts(l)$ such that $q \neq \phi$, $q \bigcap l = \phi$ and $q \bigcap s = \phi$, $l \bigcup q$ cannot be closed because it has the same support as one of its supersets $l \bigcup s \bigcup q$. In other words, no two closed itemsets share suffixes.

The number of frequent closed itemsets is smaller than the total number of frequent itemsets. Therefore, a CFP-tree$_{closed}$ is smaller than its corresponding CFP-tree$_{all}$. However, the compression ratio of a CFP-tree$_{all}$ is usually much higher than that of its corresponding CFP-tree$_{closed}$. The reason being that a

Datasets	min_sup	#FCI	#FI	ratio	CFP-closed	CFP-all	ratio
BMS-POS	0.05%	574125	587995	1.02	10.02	10.02	1
BMS-WebView-1	0.05%	131842	7.12×10^{12}	5.40×10^{7}	6.25	31.15	4.98
chess	35%	2677469	15192779	5.67	47.47	49.3	1.04
connect-4	10%	8037778	5.81×10^{10}	7226.72	175.3	175.3	1
mushroom	0.1%	164117	2.31×10^{9}	14084.30	4.84	9.24	1.91
pumsb	60%	1075015	19537365	18.17	19.75	50.38	2.55
pumsb_star	10%	1513782	5.44×10^{11}	359448.14	41.2	88.56	2.15

Table 5.1: Size of CFP-tree w.r.t closed itemsets and all itemsets

CFP-tree$_{all}$ allows both prefix sharing and suffix sharing, while there is only prefix sharing in a CFP-tree$_{closed}$. The space saved by suffix sharing is much more significant than prefix sharing. Table 5.1 shows the number of frequent closed itemsets ("#FCI" column), the total number of frequent itemsets ("#FI" column), the size of the CFP-tree$_{closed}$ ("CFP-closed" column) and the size of the CFP-tree$_{all}$ ("CFP-all" column) on several real datasets with minimum support thresholds shown in the second column. We have observed that on some datasets even the total number of frequent itemsets is hundreds of or even thousands of times larger than the number of frequent closed itemsets, the size of the CFP-tree$_{all}$ is only several times larger than that of the CFP-tree$_{closed}$. One extreme case is dataset BMS-WebView-1. On this dataset, the total number of frequent itemsets is 5×10^7 times larger than the number of frequent closed itemsets, but the size of the CFP-tree$_{all}$ is only about 5 times larger than the size of the CFP-tree$_{closed}$, which proves that the CFP-tree structure is a very compact structure for storing all frequent itemsets.

The compactness of the CFP-tree structure makes it feasible to compute and store all frequent itemsets even when it is not feasible to store all frequent itemsets in a flat format. A complete set of frequent itemsets can be retrieved from the CFP-tree storing it with little overhead. On the contrary, it is not a trivial task to derive all frequent itemsets from a set of frequent closed itemsets. Therefore, if an application requires all frequent itemsets, it is favorable to compute and store all frequent itemsets in a CFP-tree.

5.2 CFP-tree Construction

The framework of the CFP-tree construction algorithm is similar to that of the AFOPT algorithm. The main difference is that we need to identify suffix sharing when constructing a CFP-tree.

5.2.1 Computing and storing all frequent itemsets

The pseudo-code of the algorithm for constructing a CFP-tree storing all frequent itemsets is shown in Algorithm 9. When mining a conditional database D_l, we identify those frequent items appearing in every transaction of D_l, denoted as I (line 2). Then we create a singleton node containing items in I to enable suffix sharing (line 4). The remaining mining is then performed on $D_{l \bigcup I}$ instead of D_l by removing I from F (line 5). The sub CFP-tree constructed from $D_{l \bigcup I}$ is the common suffix of l and $l \bigcup I$.

Algorithm 9 CFPtree-all Construction Algorithm

Input:
l is a frequent itemset;
D_l is the conditional database of l;
E_l is the entry of l in the CFP-tree;
min_sup is the minimum support threshold;

Description:
1: Scan D_l count frequent items and sort them into ascending frequency order, denoted as F;
2: $I = \{i | i \in F$ *and* $support(l \bigcup \{i\}) = \|D_l\|\}$;
3: **if** $I \neq \phi$ **then**
4: Create a node *tnode* with one entry E and set $E.items = I$;
5: $E_l.child = tnode$; $F = F - I$; $l = l \bigcup I$; E_l=E;
6: Create a new CFP-tree node *cnode* with $\|F\|$ entries, and each entry corresponds to one item in F;
7: $E_l.child = cnode$;
8: **for all** transaction $t \in D_l$ **do**
9: $t = t \bigcap F$;
10: Sort items in t according to their orders in F;
11: Let i be the first item of t, insert t into $D_{l \bigcup \{i\}}$.
12: **for all** entry E in *cnode* **do**
13: $s = l \bigcup \{E.items\}$;
14: CFPtree-all(s, D_s, E, *min_sup*);
15: push right D_s;

There are several ways to identify those items appearing in every transaction of D_l. One way is to check whether support($l \bigcup \{i\}$) equals support(l). If support($l \bigcup \{i\}$) equals support(l), then we can safely remove i from F. The AFOPT structure can also facilitate the identification of suffix sharing. If D_l is represented using the AFOPT structure and it has only one child node, then the

items contained in that only child node appear in every transaction of D_l, and they should be removed from F. Consequently, if all the ancestors of an AFOPT node are the only child of their parents, then all the items contained in those ancestor nodes should be removed from F. For example, in Figure 3.2(c), D_p has only one child containing f, and the only child node of D_p also has only one child node containing m and a. Thus we create a singleton CFP-tree node for f and ma respectively as shown in Figure 5.1(b).

5.2.2 Computing and storing frequent closed itemsets

The main issue in constructing a CFP-tree storing only frequent closed itemsets is to prune non-closed itemsets efficiently. Non-closed itemsets are pruned based on the following two lemmas.

Lemma 7 *In Algorithm 9, an itemset l is closed if and only if two conditions hold: (1) $\forall i \in freq\_exts(l)$, $support(l \bigcup \{i\}) < support(l)$; (2) there does not exist an itemset q such that q is discovered before l, $l \subset q$ and $support(q)=support(l)$.*

Proof 6 *If l is closed, then condition 1 and condition 2 must hold. We need to prove that if both condition 1 and condition 2 hold, then l must be closed. According to the order in which the search space is explored in Algorithm 9, only itemsets discovered before l or discovered from D_l can be supersets of l. If condition 1 holds, then all the itemsets discovered from D_l must be less frequent than l according to the Apriori property. If condition 2 also holds, then no supersets of l have the same support as l. Hence l is closed.*

Lemma 8 *In Algorithm 9, for an itemset l if there exists another itemset q such that q is discovered before l, $l \subset q$ and $support(q)=support(l)$, then l is not closed, nor the itemsets mined from D_p can be closed. We say that l is pruned by q.*

Proof 7 *According to the definition of frequent closed itemset, l is not closed. We need to prove that none of the itemsets mined for D_l is closed. Itemset q is a superset of l and it has the same support as l. It implies that every transaction containing l must also contain q. Let s be any itemset mined from D_l. Then every transaction containing s must contain l, hence must contain q. Therefore,*

itemset s has the same support as $s \bigcup q$, and $s \bigcup q$ is a proper superset of s. Hence s is not closed.

Based on Lemma 7, there are two pruning conditions for a non-closed itemset l: (1) Check whether there exists some item i such that i appears in every transaction of D_l, which actually is the same as identifying suffix sharing in Algorithm 5.1(b). If such i exists, then all the itemsets containing l but not i cannot be closed. Hence we directly perform mining on $D_{l \bigcup \{i\}}$ instead of D_l to avoid mining those non-closed itemsets. (2) Examine whether there exists an itemset q discovered before l such that q is a superset of l and q has the same support as l. This checking can be done before the mining on l's conditional database starts. If such q exists, then there is no need to perform mining on D_l based on Lemma 8. Therefore, the identification of a non-closed itemset not only reduces the size of the tree, but also avoids unnecessary mining cost. The pseudo-code of the algorithm for constructing a CFP-tree$_{closed}$ is almost the same as Algorithm 5.1(b). The only difference is that before mining the conditional database of an itemset l, we check whether l is pruned by some itemset discovered before it. We utilize the CFP-tree structure to facilitate the checking.

During the mining process, the CFP-tree maintains the set of frequent closed itemsets discovered so far. Checking whether condition 2 in Lemma 7 holds is essentially to check whether there exists a frequent itemset in the CFP-tree which is a superset of l and has the same support as l. Here we have both item constraints and support constraints. The CFP-tree structure is very efficient for answering such queries. The algorithms for processing such queries on a CFP-tree are described in Section 5.3.

In the rest of this section, we introduce a technique to further reduce the cost for closed itemset checking. We use a two-layer hash map to maintain some simple statistics of the frequent closed itemsets that have been discovered during the mining process. Before searching in the CFP-tree, the two-layer hash map is checked first to examine condition (2) in Lemma 7. The two-layer hash map is shown in Figure 5.3. We maintain a hash map for each item. The hash map of item a_i is denoted by $a_i.hashmap$. The length of the hash map of an item a_i is set to $\min\{support(a_i)\text{-}min\_sup,\ max\_hashmap\_len\}$, where $max\_hashmap\_len$ is a parameter to control the maximal size of hash maps.

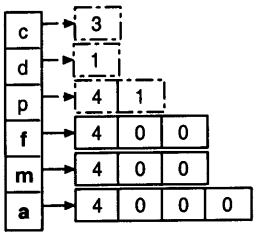


Figure 5.3: Two-layer hash map

Entry $a_i.hashmap[k]$ records the maximal length of the itemsets mapped to it, i.e. those itemsets l containing item a_i and with a support satisfying $(support(l) - min\_sup)\%max\_hashmap\_len + 1 = k$. Given an itemset $l = \{a_1, a_2, \cdots, a_m\}$, l is mapped to m entries $a_j.hashmap[(support(l) - min\_sup)\%max\_hashmap\_len + 1]$ ($j \in [1, m]$). For example, itemset $cma : 3$ set $c.hashmap[1]$, $m.hashmap[1]$ and $a.hashmap[1]$ to 3. Figure 5.3 shows the status of the two-layer hash map before mining D_f.

Lemma 9 *Given an itemset $l = \{a_1, a_2, \cdots, a_m\}$, if l is mapped to an entry E in the two-layer hash map, and E contains a value no greater than m, then there does not exist an itemset q such that q is discovered before l, $l \subset q$ and $support(l) = support(q)$.*

Proof 8 *Suppose there exists an itemset q which prunes l. Then q is mapped to the two-layer hash map before l because q is discovered before l. Itemset q should also be mapped to entry E because q is a superset of l and $support(l) = support(q)$. Then entry E should contain a value no less than $\|q\|$ ($\|q\| > m$) when checking l, which conflicts with the fact that E contains a value no greater than m. Therefore, there does not exist an itemset that prunes l.*

Checking the two-layer hash map is much more efficient than traversing the CFP-tree. If we can be sure that an itemset cannot be pruned by itemsets discovered before it using the two-layer hash map, then there is no need to traverse the CFP-tree.

Besides maintaining the maximal length of the itemsets mapped to an entry, we also maintain the maximal item id, the minimal item id and a signature of all the itemsets mapped to that entry. Given an itemset l, if l is mapped to an entry E such that E contains a smaller (larger) item id than the maximal (minimal) item

id of l, then l cannot be pruned by the itemsets discovered before it. The signature of an itemset is a length-N bit string. Let $l.signature$ denote the signature of an itemset $l = \{a_1, a_2, \cdots, a_m\}$. The signature of l is computed as follows: for each item $a \in l$, the $a\%N$-th bit of $l.signature$ is set to 1, where N is the length of the signature. The signature of an entry E in the two-layer hash map is the bitwise OR of all the signatures of the itemsets mapped to E. Given an itemset l, if l is mapped to an entry E such that $E.signature\ AND\ l.signature \neq l.signature$, then l cannot be pruned by the itemsets discovered before it.

The hash map of an item a_i can be deleted after a_i's conditional database is processed because no itemsets discovered later contain a_i, and the hash map of a_i will not be used in later mining. The two-layer hash map is very effective in avoiding unnecessary tree traversal. Our experiment results indicate that when the minimum support threshold is high, almost all of the frequent closed itemsets can be verified by the two-layer hash map.

5.3 Query Processing

In this section, we describe the properties of the CFP-tree structure and show how to utilize these properties to process queries on a CFP-tree. We consider two basic types of queries: (1) *queries with minimum support constraints*, for example, "find the frequent itemsets with support no less than *min_sup*", where *min_sup* should be no less than the construction minimum support of the CFP-tree; and (2) *queries with item constraints*, e.g. "find the frequent itemsets containing a set of items I'". These two types of queries are very common in practice, and they are also essential for efficiently evaluating more complex queries. At the end of this section, we briefly discuss how to process other types of queries.

5.3.1 The properties of the CFP-tree structure

The CFP-tree structure has two important properties that can be utilized in query processing.

Apriori Property. *The support of an entry E cannot be greater than that of its ancestors.* This property can be used when processing queries with minimum support constraints. If the support of an entry does not satisfy the minimum

support constraint specified in a query, then there is no need to access the subtree pointed by the entry.

Left Containment Property. *In a CFP-tree node, the item of an entry E can appear in the subtrees pointed by the entries before E, but cannot appear in the subtrees pointed by the entries after E.* This property can be utilized when processing queries with item constraints.

The ascending frequency order helps query processing. The left containment property implies that the subtrees pointed by the entries at the beginning of a node are more likely to be visited than the subtrees pointed by the entries at the end of a node. An entry with low minimum support often points to a small subtree. All the entries in a node are sorted into ascending frequency order, so small trees are traversed more frequently. This results in great saving when processing queries with item constraints. For queries with minimum support constraints, the most infrequent item of every itemset is first checked based on the ascending frequency order. Hence those itemsets that contain an item dissatisfying the minimum support constraint can be quickly discarded.

The bitmap maintained at each entry can be used to further avoid unnecessary traversal cost when processing queries with item constraints. The hash bitmap of an entry E is set as follows. For every item i in the subtree pointed by E, the j-th bit of $E.bitmap$ is set to 1, where $j=(i \bmod N)$ and N is the length of the bitmap. Before performing search on the subtree pointed by $E.child$, we first check $E.bitmap$ to see whether all the items being searched are in the subtree pointed by E. The length of the hash bitmap is a trade-off between the size of the CFP-tree and the search time. In our experiments, the length of the bitmap is set to 32.

5.3.2 Queries with minimum support constraints

A query with a minimum support constraint is to output the frequent itemsets with support no less than a user specified minimum support min_sup, where min_sup must be no less than the construction minimum support of the CFP-tree. Here we assume that users are interested in all frequent itemsets and use a CFP-tree storing all frequent itemsets to answer queries. If users are interested in only frequent closed itemsets, then we should use a CFP-tree storing frequent

Algorithm 10 Search_Minsup Algorithm

Input:

cnode is a CFP-tree node;
min_sup is the minimum support threshold;
l_m contains all the items in the nodes with multiple entries on the path from root to *cnode*;
l_s contains all the items in the singleton nodes on the path from root to *cnode*;

Description:

1: **if** *cnode* contains only one entry E AND $E.support \geq min\_sup$ **then**
2: output all itemsets in $\{l_m \bigcup l'_s \bigcup l_E \mid l'_s \subseteq l_s, l_E \subseteq E.items \ AND \ l_E \neq \phi\}$ with support $E.support$;
3: **if** $E.child \neq$ NULL **then**
4: Search_Minsup($E.child$, min_sup, l_m, $l_s \bigcup E.items$);
5: **else if** *cnode* contains more than one entries **then**
6: E' = BinarySearch(*cnode*, *min_sup*);
7: **for all** entry $E \in cnode$, $E = E'$ or E after E' **do**
8: output all itemsets in $\{l_m \bigcup l'_s \bigcup E.items \mid l'_s \subseteq l_s\}$ with support $E.support$;
9: **if** $E.child \neq$ NULL **then**
10: Search_Minsup($E.child$, min_sup, $l_m \bigcup E.items$, l_s);

closed itemsets to answer queries. The query processing algorithms on the two types of CFP-trees are similar.

According to the apriori property of the CFP-tree structure, only subtrees pointed by entries with support no less than the minimum support constraint *min_sup* should be searched. Items are sorted into ascending frequency order at every node, so a binary search can be performed to find the first entry with support no less than *min_sup*. Suppose this entry is E, then only the subtrees pointed by entries after E and by E itself should be accessed. Algorithm 10 shows the pseudo-code for the search algorithm. BinarySearch(*cnode*, *min_sup*) procedure returns the first entry in *cnode* whose support is no less than *min_sup*.

A path in a CFP-tree storing all frequent itemsets can represent multiple itemsets because of suffix sharing. To output all the frequent itemsets satisfying the minimum support constraint, we maintain two set of items l_m and l_s when traversing the CFP-tree. Itemset l_m and l_s together contain the set of items appearing on the path from root to *cnode*. Itemset l_m contains those items appearing in the nodes with multiple entries. Items in l_m must be contained in the output itemsets because they decide the unique path from root to *cnode* in the CFP-tree. Itemset l_s contains those items appearing in the singleton nodes on the path from root to *cnode*. Including or excluding the items in l_s leads to itemsets with the same support. Hence the path from root to an entry E in *cnode* represents $2^{\|l_s\|} \cdot (2^{\|E.items\|} - 1)$ itemsets (line 2, line 8).

Consider an example query "find all the frequent itemsets with support no less

than 50%". In the CFP-tree shown in Figure 5.1(b), a binary search is performed on node 1, and p is found to be the first item with support no less than 50%. All the entries before p can be skipped. The child node of p (node 3) is first visited and itemset pf is found to be frequent. We move on to the node pointed by f (node 4), and itemset fa is found to be frequent. Finally, itemsets ma and a are found to be frequent.

5.3.3 Queries with item constraints

A query with item constraints is to find frequent itemsets containing a set of items I'. Based on the left containment property, if an item i appears in an entry E in a CFP-tree node *cnode*, then in *cnode*, only those subtrees pointed by entries before E can contain i (line 7-11). The subtree pointed by an entry before E may not actually contain i. The hash bitmap maintained at each entry can be utilized to reduce unnecessary search cost. Before the subtree pointed by an entry E is searched, the hash bitmap of E is first checked (line 4 and line 11). If and only if the hash bitmap indicates that all the items being searched appear in that subtree, search on that subtree should continue. Algorithm 11 shows the pseudo-code for evaluating queries with item constraints. Here the two itemsets l_m and l_s are slightly different from that in Algorithm 10. The reason being that every itemset in the output of Algorithm 11 should contain I'. Therefore, even if an item in I' appears in some singleton node, it is always put into l_m (line 5).

To process the query "find all the itemsets containing item p and f" on the CFP-tree shown in Figure 5.1(b), only the subtrees pointed by the first three entries of node 1 should be accessed. We start from entry c, and examine the hash bitmap of entry c. Suppose the bitmap indicates that item f does not appear in the subtree pointed by c, then there is no need to access node 2 and its descendants. The next entry points to an empty subtree. We continue the search at entry p. Its child node is accessed and itemset pf is found to satisfy the item constraints. Finally the child node of pf is visited and itemsets pfm, pfa and $pfma$ are found to satisfy the item constraints.

Algorithm 11 Search_Item Algorithm

Input:
cnode is a CFP-tree node;
I' is the set of items that must be contained in itemsets;
l_s contains all the items in the singleton nodes on the path from root to *cnode* except those items in I';
l_m contains all the other items on the path from root to *cnode*;

Description:
1: **if** *cnode* contains only one entry E **then**
2: **if** $(I' - E.items) = \phi$ **then**
3: output all itemsets in $\{l_m \bigcup (E.items \bigcap I') \bigcup l'_s \bigcup l_E \mid l'_s \subseteq l_s,\ l_E \neq \phi\ AND\ l_E \subseteq (E.items - I')\}$ with support $E.support$;
4: **if** $E.child \neq$ NULL AND all the items in $(I' - \{E.items\})$ are in $E.child$ **then**
5: Search_Item($E.child$, $I' - E.items$, $l_m \bigcup (E.items \bigcap I')$, $l_s \bigcup (E.items - I')$);
6: **else if** *cnode* contains more than one entries **then**
7: **if** $I' \neq \emptyset$ **then**
8: $\bar{E}$ = the first entry of *cnode* such that $\bar{E}.item \in I'$;
9: **else**
10: $\bar{E}$ = the last entry of *cnode*;
11: **for all** entry $E \in$ *cnode*, E before $\bar{E}$ or E=$\bar{E}$ **do**
12: **if** $(I' - E.items) = \phi$ **then**
13: output all itemsets in $\{l_m \bigcup l'_s \bigcup E.items \mid l'_s \subseteq l_s\}$ with support $E.support$;
14: **if** $E.child \neq$ NULL AND all the items in $(I' - E.items)$ are in $E.child$ **then**
15: Search_Minsup($E.child$, min_sup, $l_m \bigcup E.items$, l_s);

5.3.4 Queries with both constraints

Another type of queries has both minimum support constraints and item constraints. The algorithm for evaluating such queries can also combine the pruning power of the apriori property, the left containment property and the hash bitmap. It is not difficult to obtain the algorithm for processing queries with both constraints from Algorithm 10 and Algorithm 11.

When constructing a CFP-tree storing only frequent closed itemsets, we need to check whether an itemset can be pruned by some closed itemset discovered before the itemset (condition (2) in Lemma 7). It is essentially to search in the CFP-tree for an itemset containing l and with the same support as l, which is a query with both item constraint and support constraint. Algorithm 12 shows the pseudo-code for processing such queries on a CFP-tree storing closed itemsets. A CFP-tree storing frequent closed itemsets stores some itemsets which are prefixes of some closed itemsets, but themselves are not closed, e.g. itemset c, p and m in Figure 5.2(b). To output only closed itemsets, we need to check whether an itemset has a greater support than its children before outputting the itemset (line 2 and line 10).

Let us consider the query "find the frequent closed itemsets with support no

Algorithm 12 Search_Both Algorithm

Input:
cnode is a CFP-tree node
min_sup is the minimum support threshold
I is a set of items to be contained in itemsets
l contains all the items on the path from root to *cnode*;

Description:
1: **if** *cnode* has only one entry E **then**
2: **if** $I = \phi$ AND $support(l) \geq E.support$ **then**
3: output l with its support;
4: **if** $E.support \geq min\_sup$ **then**
5: **if** $E.child \neq$NULL AND all the items in $(I - E.items)$ are in $E.child$ **then**
6: Search_Both($E.child$, min_sup, $I - E.items$, $l \bigcup E.items$);
7: **else if** $(I - E.items) = \phi$ AND $E.child$=NULL **then**
8: output $l \bigcup E.items$ with support $E.support$;
9: **else if** *cnode* has more than one entries **then**
10: **if** $I = \phi$ **then**
11: output l with its support;
12: **if** $I \neq \phi$ **then**
13: $\bar{E}$ = the first entry of *cnode* such that $\bar{E}.item \in I$;
14: **else**
15: $\bar{E}$ = the last entry of *cnode*;
16: E' = BinarySearch(*cnode*, *min_sup*);
17: **for all** entry $E \in cnode$, E between E' and $\bar{E}$ **do**
18: **if** $E.child \neq$ NULL AND all the items in $(I - E.items)$ are in $E.child$ **then**
19: Search_Both($E.child$, min_sup, $I - E.items$, $l \bigcup E.items$);
20: **else if** $(I - E.items) = \phi$ AND $E.child$=NULL **then**
21: output $l \bigcup E.items$ with its support $E.support$;

less than 50% and containing items p and f" on the CFP-tree shown in Figure 5.2(b). According to the item constraint, we need to check only the first three entries in node 1. According to the minimum support constraint, only the last four entries should be accessed. Therefore, only the child node of entry p should be accessed. We find only one itemset pf satisfying both constraints.

5.3.5 Other types of queries

The CFP-tree structure is also efficient for processing other types of queries, e.g. subset queries and similarity queries.

A subset query is to find the frequent itemsets that are subsets of a given itemset, e.g. "find the frequent itemsets that are subsets of itemset $\{c, f, m\}$". To process subset queries, we simply match the given itemset against the CFP-tree. When matching a node with multiple entries, only the subtrees pointed by those entries whose item appears in the given itemset should be visited. When matching a singleton node, the child node of the singleton node is always visited. Among the items of the singleton node, those that do not appear in the given itemset are discarded, and those items contained in the given itemset are put into l_s. We use the above example query and the CFP-tree shown in Figure 5.1(b)

to show how to process subset queries. We start from the first entry of node 1. Item c appears in the given itemset, so we output itemset $c : 3$. The child node of entry c (node 2) is visited, and it contains two items m and a. Item a is not contained in the given itemset, so it is discarded. We output itemset $cm : 3$. Then we move on to the next entry in node 1. Items d and p do not appear in the given itemset, hence we do not need to visit their child nodes. Then we move onto entry f in node 1, and find itemset $f : 5$. The child node of entry f (node 5) contains two entries. The item of one entry is m, and m is contained in the given itemset. So we output itemset $fm : 3$. Next we visit entry m in node 1 and find itemset $m : 5$. The search is finished.

A similarity query is to find the frequent itemsets that contain at least m common items with a given itemset I', where $m \leq \|I'\|$. An example similarity query is "find the frequent itemsets that contains at least 2 common items with itemset $\{c,\ p\ ,f\}$". Queries with item constraints can be viewed as a special case of similarity queries with $m = \|I'\|$. Processing similarity queries is similar to processing queries with item constraints. The main difference is the output condition. When processing similarity queries, the output condition at line 2 and line 12 in Algorithm 11 should be changed to "$(\|I' - E.items\| \leq m)$".

5.4 Incremental Maintenance

The database used in mining is often very large and dynamic in nature. To avoid recomputing the whole CFP-tree when the underlying database is updated, we propose efficient algorithms to incrementally maintain a CFP-tree. The CFP-tree structure stores the frequent itemsets computed from a database instead of the database itself. Maintaining a CFP-tree is essentially to maintain the set of frequent itemsets stored in the CFP-tree. Maintaining frequent closed itemsets is more challenging than maintaining a complete set of frequent itemsets because the update of the underlying database may make some closed itemsets become non-closed and some non-closed itemsets become closed in addition to the occurrence of new frequent itemsets and invalidation of existing frequent itemsets. In this section, we first describe how to maintain a CFP-tree storing all frequent itemsets, then we show how to maintain a CFP-tree storing only frequent closed itemsets.

For the ease of presentation, in the rest of this section, we use D, d and U to

denote the old database, the set of transactions to be inserted or deleted and the updated database respectively, use T_x to denoted the CFP-tree constructed from database x, where x=D,d or U, and use L_x to denote the set of itemsets frequent in x (x=D, d or U).

5.4.1 Maintaining the negative border of a CFP+-tree

When the underlying transaction database is updated, some frequent itemsets may become infrequent and some infrequent itemsets may become frequent. It is easy to update the counts of the itemsets that are originally frequent. To check whether some infrequent itemsets become frequent, we may need to perform a full scan of the original database because the support of the infrequent itemsets is not available. The original database is usually very large, therefore one main issue in updating frequent itemsets is to reduce the chance that the original database is re-scanned. Algorithms for incrementally updating a complete set of frequent itemsets [18, 24, 89] use the concept of *negative border* to reduce the possibility of re-scanning the original database. They re-scan the original database only if the update causes the negative border to expand. The *negative border* of the set of frequent itemsets L mined from a database, denoted as $\mathbf{NB}(L)$, is defined as $\mathbf{NB}(L)$=$\{l | l \notin L \wedge \forall l' \subset l, l' \in L\}$. If an itemset $l \in NB(L)$, then l is said to be on the negative border. To facilitate the maintenance of a CFP-tree, we define the concept of the negative border of a CFP-tree.

Definition 7 (Negative Border of an Entry) *Let T be a CFP-tree, E be an entry in T and l_E be the set of items on the path from root to E (including E.items), the negative border of E is defined as $NB(E)$= $\{l_E \bigcup \{i\} | i \in cand\_exts(l_E) \wedge support(l_E \bigcup \{i\}) < min\_sup\}$.*

The empty set is viewed as a special itemset. Its candidate extensions include all the items in the database, and its negative border includes all the infrequent items. The negative border of a CFP-tree is defined as the union of the negative borders of all the entries in the CFP-tree plus the negative border of the empty set.

Definition 8 (Negative Border of a CFP-tree) *The negative border of a CFP-tree T is defined as $NB(T) = (\bigcup_{E \in T} NB(E)) \bigcup NB(\phi)$.*

Figure 5.4 shows the negative border of the CFP-tree in Figure 5.1(b). The itemsets on the negative border are indicated with grey color.

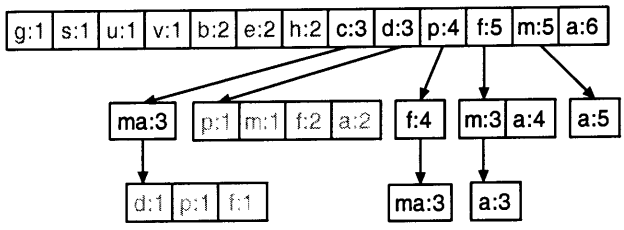


Figure 5.4: CFP+-tree with negative border

5.4.2 Maintaining a complete set of frequent itemsets

It is inefficient to update the CFP+-tree structure every time a new transaction is inserted or an old transaction is deleted because a single transaction can affect many frequent itemsets. It is more suitable to conduct the update in batch so that different transactions can share their updating cost. Both insertion and deletion can cause the occurrence of new frequent itemsets either because the support of some itemsets increases (insertion) or because the size of the database decreases which lowers the minimum count threshold (deletion). Our incremental update algorithm consists of two steps: the first step is to update the support of the original frequent itemsets and the second step is to generate new frequent itemsets.

Updating existing frequent itemsets. In the first step, the original CFP-tree T_D and the set of new transactions to be inserted (or the set of old transactions to be deleted) are scanned simultaneously. The original CFP-tree T_D guides the exploration of the search space over the incremental database d. The update process is similar to Algorithm 9. One difference is that frequent items are now sorted according to their orders in the original CFP-tree instead of being sorted into ascending frequency order. During the update process, if the count of an itemset does not change in U, then there is no need to access the descendants of that itemset in T_D. If the count of an itemset l is changed, l can be in several situations:

1. l is frequent in D and is still frequent in U, i.e. $l \in L_D \bigcap L_U$;

2. l is frequent in D but becomes infrequent in U, i.e. $l \in L_D - L_U$;

3. l is on the negative border of T_D and becomes frequent in U, i.e. $l \in NB(L_D) \bigcap L_U$;

4. l is on the negative border of T_D and is still infrequent in U, i.e. $l \in NB(L_D) - L_U$.

For the first case, we simply update the count of l. For the second case and the fourth case, we update the count of l if they are on the negative border of U, otherwise they are removed from the CFP-tree. We maintain the set of itemsets in the third case and generate new frequent itemsets in the next step. For each itemset in $l \in NB(L_D) \bigcap L_U$, we maintain its count, its candidate extensions and its position in the CFP-tree.

Generating new frequent itemsets. If some infrequent itemsets in D become frequent in U, i.e. the negative border of a CFP-tree expands, we scan the original database to generate new frequent itemsets. The itemsets in $NB(L_D) \bigcap L_U$ are sorted according to their depth-first order in T_U. A conditional database is constructed for each of them. When constructing the conditional databases, only transactions containing l are put into l's conditional database. Each transaction in l's conditional database contains only l's candidate extensions.

When constructing conditional databases for the itemsets in $NB(L_D) \bigcap L_U$, we need to match every transaction with every itemset in $NB(L_D) \bigcap L_U$. It can be a very costly operation especially when the size of the database and the number of itemsets in $NB(L_D) \bigcap L_U$ are both very large. We utilize prefix sharing among itemsets to reduce the cost for subset matching. Let l_1 and l_2 be two itemsets in $NB(L_D) \bigcap L_U$ and they share the same length-m prefix. If a transaction t does not contain the i-th item of itemset l_1, where $i <= m$, then there is no need to match transaction t with l_2. We maintain a length-$\|l\|$ array for each itemset l in $NB(L_D) \bigcap L_U$, called the jumping array of l and denoted as $l.jump\_array$. The i-th entry of $l.jump\_array$ contains the next new itemset to be matched with a transaction t if the i-th item of l is not contained in t.

After the conditional databases of the itemsets in $NB(L_D) \bigcap L_U$ are constructed, all the new frequent itemsets, i.e. all the itemsets in $L_U - L_D$, are discovered from those conditional databases using Algorithm 9.

5.4.3 Maintaining frequent closed itemsets

When updating a CFP-tree storing frequent closed itemsets, we need to maintain the closure of the frequent itemsets in U. It is easy to handle when some closed itemsets become non-closed, but it is difficult to handle the opposite case because non-closed itemsets are not stored in a CFP-tree$_{closed}$. To solve this problem, we include all frequent itemsets in a CFP-tree$_{closed}$ and mark those non-closed itemsets. Insertion and deletion have different effects on the closure of the frequent itemsets. We describe how to maintain a CFP-tree$_{closed}$ upon insertion and deletion separately.

Inserting new transactions

Inserting new transactions can make some non-closed itemsets become closed, but can never make closed itemsets become non-closed. It is formally stated as follows.

Lemma 10 *If itemset l is closed in the original database D, then l must be closed in the updated database U=D+d, where d is the set of transactions inserted.*

Proof 9 *Suppose l is not closed in U, then there exists a frequent itemset l' such that $l \subset l'$ and $support_U(l) = support_U(l')$. It implies that every transaction in U containing itemset l must contain l'. D is a subset of U, so every transaction in D containing l must also contain l', i.e. $support_D(l) = support_D(l')$. It conflicts with the fact that l is closed in D. Therefore, if l is closed in D, then it must be closed in U=D+d.*

Based on the above lemma, among the itemsets in $L_U \bigcap L_D$ (those frequent itemsets updated in step 1), we only need to check those non-closed itemsets to see whether they become closed. It is sufficient to perform the checking on $T_U \bigcap T_D$ instead of T_U because an itemset that can prune an itemset $l \in (L_D \bigcap L_U)$ must also belong to $L_D \bigcap L_U$. In other words, a frequent itemset updated in step 1 should be checked within the itemsets updated in step 1.

The closure of all the new frequent itemsets, i.e. those itemsets mined in step 2, should also be checked. It is sufficient to perform the checking within $L_U - L_D$ based on the following lemma.

Lemma 11 *Let D be the original database, U be the updated database and l be an itemset in $L_U - L_D$. It there exists an itemset l' such that $l \subset l'$ and $support_U(l) = support_U(l')$, then l' must be in $L_U - L_D$.*

Proof 10 *Itemset l is infrequent in D and l' is a superset of l. According to the apriori property, itemset l' must be infrequent in D. Itemset l is frequent in U and $support(l) = support(l')$, then l' must be frequent in U too. Hence we have l' is in $L_U - L_D$.*

The above lemma also works for deletion.

Deleting Old Transactions

Deleting old transactions can make closed itemsets become non-closed, but cannot make non-closed itemsets become closed as stated in the following lemma.

Lemma 12 *If an itemset l is not closed in the original database D, then it cannot be closed in the updated database U=$D - d$, where d is the set of transactions deleted.*

Proof 11 *Itemset l is not closed in D. Then there exists an itemset l' such that $l \subset l'$ and $support_D(l')$=$support_D(l)$. It implies that every transaction in D containing l also contains l'. U is a subset of D, therefore every transaction in U containing l must also contain l', i.e. $support_U(l) = support_U(l')$. Therefore, l is not closed in U.*

Based on the above lemma, if an itemset is not closed in D, then there is no need to check whether it is closed in U. However, if an itemset l is closed in D, it is possible that l becomes non-closed in U because the count of l is decreased and l becomes as frequent as one of its supersets.

For the itemsets in $L_U \bigcap L_D$ (those frequent itemsets updated in step 1), we only need to check those itemsets that are closed in D to see whether they become non-closed in U. To reduce the checking cost, we break the itemsets in $L_U \bigcup L_D$ that are closed in D into two sets according to their support. The first set contains those itemsets closed in D and with count in U no less than $\|D\| \cdot min\_sup$. The

second set contains those itemsets closed in D and with count in U between $\|U\| \cdot min\_sup$ and $\|D\| \cdot min\_sup$. For any itemset l in the first set, it is sufficient to check l within $L_D \bigcap L_U$ because if itemset l' prunes l, l' must be frequent in both D and U ($count_D(l') \geq count_U(l') = count_U(l) \geq \|D\| \cdot min\_sup$). For the itemsets in the second set, we first check them in $L_U \bigcap L_D$. If they cannot be pruned by any itemset in $L_D \bigcap L_U$, then we check whether they can be pruned by the itemsets in $L_U - L_D$ (those itemsets mined in step 2). Therefore, when updating a CFP-tree$_{closed}$ in case of deletion, we have a third step if some itemsets in the second set cannot be pruned by the itemsets in $L_U \bigcap L_D$. The task of the third step is to check whether those itemsets in the second set that cannot be pruned by the itemsets in $L_U \bigcap L_D$ can be pruned by the itemsets in $L_U - L_D$.

5.5 A Performance Study

In this section, we study the efficiency of the CFP-tree related algorithms. The experiments were conducted on a 2.26Ghz Pentium IV with 512MB memory running Microsoft Windows XP. All codes were complied using Microsoft Visual C++ 6.0. Table 5.2 shows the datasets used in our performance study. BMS-POS and BMS-WebView-1 are two sparse datasets containing click-stream data [104]. Datasets chess, connect-4 and pumsb are obtained from the UCI machine learning repository[1] and they are very dense. Dataset pumsb_star is a synthetic dataset produced from pumsb by Roberto J. Bayardo.

Data Sets	Size	#Trans	#Items	MaxTL	AvgTL
BMS-POS	11.62MB	51,5597	1,657	165	6.53
BMS-WebView-1	1.28M	59,601	497	267	2.51
chess	0.34M	3,196	75	37	37.00
connect-4	9.11M	67,557	129	43	43.00
pumsb	16.30M	49,046	2,113	74	74.00
pumsb_star	11.03MB	49,046	2,088	63	50.48

Table 5.2: Datasets

[1]http://www.ics.uci.edu/ mlearn/MLRepository.html

5.5.1 Compactness of the CFP-tree structure

We compared the size of the two types of CFP-trees with the size of flat files. We used flat files to store all frequent itemsets (denoted as "Flat-all") and frequent closed itemsets (denoted as "Flat-closed"). For each itemset stored in flat files, we stored its length, support and the set of items in the itemset.

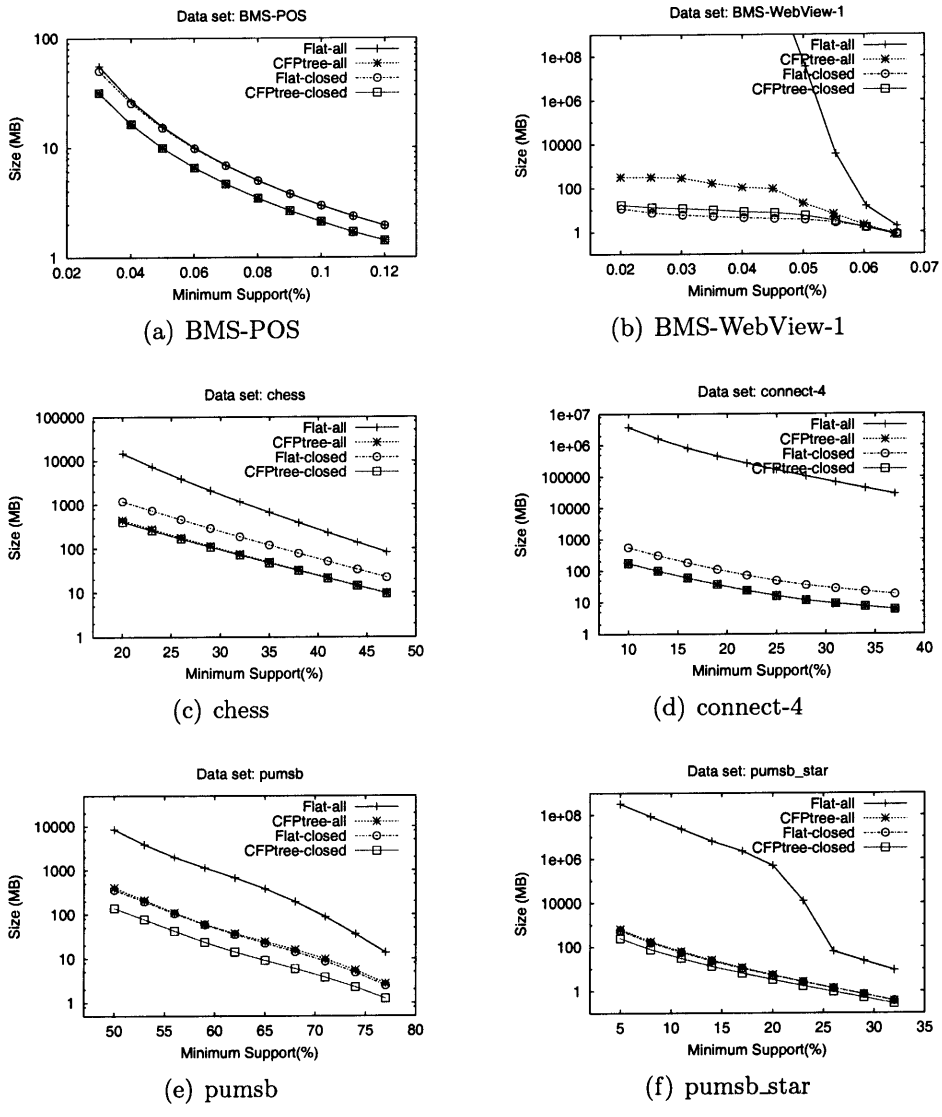


Figure 5.5: Compactness of the CFP-tree structure

The results are shown in Figure 5.5. From Figure 5.5, we can make the following observations: (1) The number of frequent closed itemsets can be substantially smaller than the total number of frequent itemsets, especially on dense dataset and/or minimum support is low. For example, "Flat-closed" is tens of times smaller than "Flat-all" on datasets chess and pumsb, and thousands of times

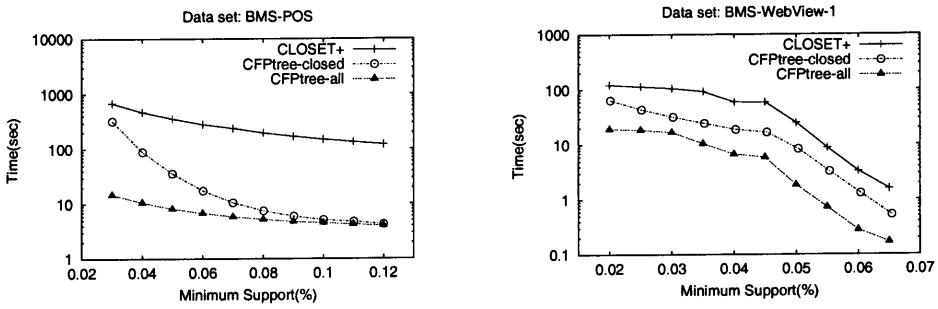
smaller on dataset connect-4. The reduction ratio on datasets pumsb_star and BMS-WebView-1 is even higher. (2) The CFP-tree structure is a very compact structure for storing a complete set of frequent itemsets because of suffix sharing. The compression ratio of the CFP-tree$_{all}$ reaches tens of thousands or even several millions on datasets pumsb_star and BMS-WebView-1. The size of the CFP-tree$_{all}$ is almost the same as that of the CFP-tree$_{closed}$ on datasets chess and connect-4, while "Flat-all" is tens of or thousands of times larger than "Flat-closed" on these two datasets. (3) The size of the CFP-tree structure is much smaller than the size of the corresponding flat file on most datasets except on BMS-WebView-1. On dataset BMS-WebView-1, the size of the CFP-tree$_{closed}$ is slightly larger than "Flat-closed" because of the space overhead for storing pointers and hash bitmaps in CFP-tree entries. However, on this dataset, the compression ratio of the CFP-tree$_{all}$ is the highest among all datasets.

5.5.2 Construction time

We compared the CFP-tree construction algorithms with the CLOSET+ algorithm [93]. Figure 5.6 shows the running time of the algorithms on several datasets. The running time of the CFP-tree construction algorithms in Figure 5.6 includes both CPU time and I/O time.

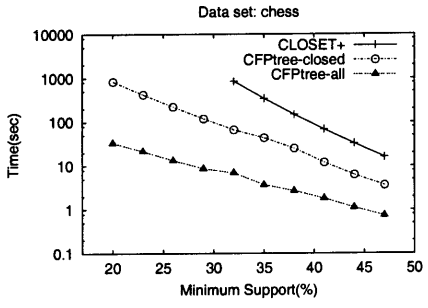
Figure 5.6 shows that a CFP-tree$_{all}$ needs less construction time than its corresponding CFP-tree$_{closed}$ even though a CFP-tree$_{all}$ requires more output time. The reason being that when constructing a CFP-tree storing only frequent closed itemsets, we need additional overhead to prune non-closed itemsets. The pruning cost increases with the increase of the size of the CFP-tree.

The construction algorithm of a CFP-tree$_{closed}$ is faster than CLOSET+ on all tested datasets. The CLOSET+ algorithm uses an FP-tree like structure to store frequent closed itemset for closed itemset checking. The FP-tree structure needs more child pointers than the CFP-tree structure. It also needs to maintain parent pointers and node-link pointers at each node. Thus the FP-tree structure can get very large when the number of frequent closed itemsets is large. For example, on datasets chess and pumsb_star, the size of the FP-tree structure exceeds the size of the main memory when the minimum support threshold is low, which makes the CLOSET+ algorithm fail to terminate within a reasonable period of time.

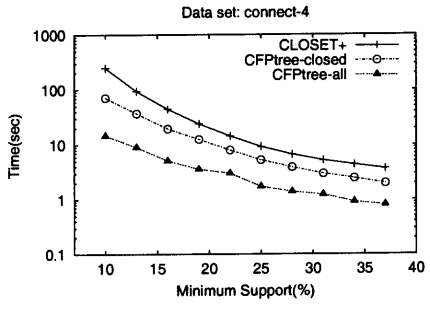


(a) BMS-POS

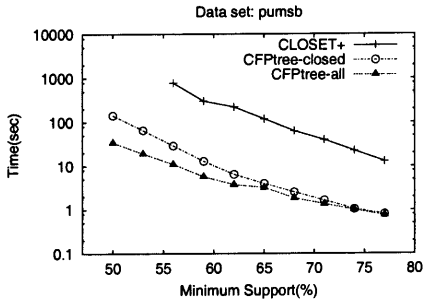
(b) BMS-WebView-1



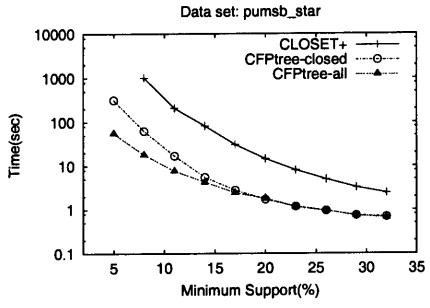
(c) chess



(d) connect-4



(e) pumsb



(f) pumsb_star

Figure 5.6: CFP-tree construction time

The efficiency of the CFP-tree$_{closed}$ construction algorithm is due to two factors: (1) The identification of suffix sharing effectively removes non-closed itemsets. The last three columns in Table 5.3 show the total number of non-closed itemsets ("#non-closed" column), the number of non-closed itemsets that are removed by suffix sharing thus not searched neither in the CFP-tree nor in the two-layer hash map ("#not-checked" column) and the ratio of the above two columns. The last column indicates that more than 99% non-closed itemsets are removed by the identification of suffix sharing. (2) The two-layer hash map is efficient for closed itemset checking and effective in avoiding unnecessary traversal of the CFP-tree. Table 5.4 shows the number of itemsets that are checked in the CFP-tree$_{closed}$ construction algorithm and the number of closed itemsets verified

Datasets	min_sup	#FI	#FCI	#non-closed	#not-checked	ratio
BMS-POS	0.03%	1939308	1761608	177700	177700	1
BMS-WebView-1	0.055%	69417074	99697	69317377	69258994	0.99
chess	20%	289154814	22808625	266346189	266337224	≈1
connect-4	10%	58062343952	8035412	58054308540	58054308540	1
pumsb	50%	165903541	7121265	158782276	158690215	0.99
pumsb_star	5%	4.07×10^{12}	9370737	4.07×10^{12}	4.07×10^{12}	≈1

Table 5.3: Number of non-closed itemsets removed by suffix sharing

Datasets	min_sup	#checked	#2-layer-map	ratio	min_sup	#checked	#2-layer-map	ratio
BMS-POS	0.03%	1761607	801977	0.46	0.1%	122369	117404	0.96
BMS-WebView-1	0.055%	158079	22251	0.14	0.08%	9501	8315	0.88
chess	20%	22817589	817187	0.04	70%	23892	21231	0.89
connect-4	10%	8035411	3731149	0.46	50%	130101	113617	0.87
pumsb	50%	7213325	1556445	0.22	75%	101229	94766	0.94
pumsb_star	5%	10464920	1722205	0.16	25%	46766	42079	0.90

Table 5.4: Pruning power of the two-layer hash map

by the two-layer hash map. When the minimum support threshold is relatively high, more than 80% of the closed itemsets can be validated by the two-layer hash map.

5.5.3 Query processing cost

We compared the CFP-tree structure with the inverted files on processing queries because the inverted files exhibit the best performance in a comprehensive study [37] of four index structures for set-valued data (sequential signature files, signature trees, extendible signature hashing and inverted files). To make the comparison fare, we used frequent closed itemsets to conduct the experiments, i.e. we stored only frequent closed itemsets in a CFP-tree, and compared the CFP-tree with the inverted files constructed on the same set of frequent closed itemsets.

We constructed the inverted files as follows. Frequent closed itemsets were stored in a binary file. We maintained a directory containing all the distinct frequent items. For each frequent item, we created a list consisting of the references to the frequent closed itemsets containing that item. Since the number of frequent items in a database is not very large, we kept the search values of the directory in a hash table. We used two different sorting methods to sort the

frequent closed itemsets in the binary files, and then constructed the inverted files on the sorted itemsets. The first sorting method is to sort the frequent closed itemsets into support descending order, denoted as "INV-SupDes". The second sorting method is to keep the frequent closed itemsets in the order in which they are discovered, denoted as "INV-DFS". The second order is the same as the order obtained by traversing the CFP-tree in depth-first order.

Queries with item constraints are processed on the inverted files using the traditional intersection based algorithm. To process queries with minimum support constraints on the inverted files constructed using the first sorting method, we simply scan the frequent closed itemsets from the beginning of the binary file until we reach an itemset such that the support of the itemset is less than the given minimum support threshold. There is no need to access the directory or the lists. Therefore, the first sorting method is optimal in answering queries with minimum support constraints. However, during our experiments, we found that this sorting method was not good at processing queries with item constraints. Hence, we tried the second sorting method.

For the second sorting strategy, we maintain one more list, called support list, to cope with queries with minimum support constraints. Each entry in the support list contains a reference to a frequent closed itemset and the count of the frequent closed itemset. The entries in the support list are sorted into descending order of count. To process queries with minimum support constraints, we first read from the support list those entries with support no less than the given minimum support constraint, and sort those entries into ascending order according to their references, then retrieve frequent closed itemsets from the binary file using the sorted list.

We test the query processing algorithms on three datasets. One each dataset, we generate a CFP-tree and a binary flat file containing the set of frequent closed itemsets with a low minimum support threshold. Table 5.5 shows the construction minimum support threshold, the number of frequent closed itemsets, the size of the CFP-tree ("CFP-size" column), the size of the binary file storing the frequent closed itemsets ("PatSet-size" column) and the size of the inverted files ("INV-size" column) on three datasets.

We compared both the evaluation time and the I/O cost of the query pro-

Datasets	min_sup	#FCI	CFP-size	PatSet-size	INV-size
BMS-POS	0.02%	4,255,421	78.36 MB	128.36 MB	106.14 MB
connect-4	15%	3,254,779	69.50 MB	215.34 MB	190.69 MB
pumsb	50%	2,729,795	51.13 MB	129.72 MB	109.16 MB

Table 5.5: Size of CFP-tree, pattern set and inverted files

cessing algorithms. To make each run has a "cold start", we cleared buffers and caches before each run. Figure 5.7 shows the I/O cost and the running time of the query processing algorithms when processing queries with minimum support constraints. Figure 5.8 shows the I/O cost and the running time of the algorithms when processing queries with item constraints. The sequential scan method is shown as baseline.

For queries with minimum support constraints, the "INV-SupDes" method does not need to access the inverted files. It accesses only those itemsets satisfying the minimum support threshold, and these itemsets are clustered together. Therefore, the "INV-SupDes" method needs to scan the minimal number of pages with sequential accesses, which is optimal for processing queries with minimum support constraints. The "INV-SupDes" method shows the best performance when the minimum support threshold is high. However, when the minimum support threshold is low, the CFP-tree structure can benefit from its compactness and performs even better than the"INV-SupDes" method on dataset connect-4. The "INV-DFS" method uses the support list to retrieve the itemsets satisfying the minimum support constraint, so for queries with minimum support constraints, the "INV-DFS" method does not need to access the inverted files either. However, frequent closed itemsets are not clustered according to support in the "INV-DFS" method. Hence the "INV-DFS" method has to access much more pages than the "INV-SupDes" method with many random accesses. It even performs worse than the sequential scan method on some datasets when the minimum support threshold is low. The "CFP-tree" method has to perform some random accesses too, so it is slower than the "INV-SupDes" method on dataset pumsb even though its I/O cost is lower than that of the "INV-SupDes" method when the minimum support threshold is low.

For queries with item constraints, we selected two items as constraints on each dataset. The x-axis of Figure 5.8 shows the number of closed itemsets satisfying

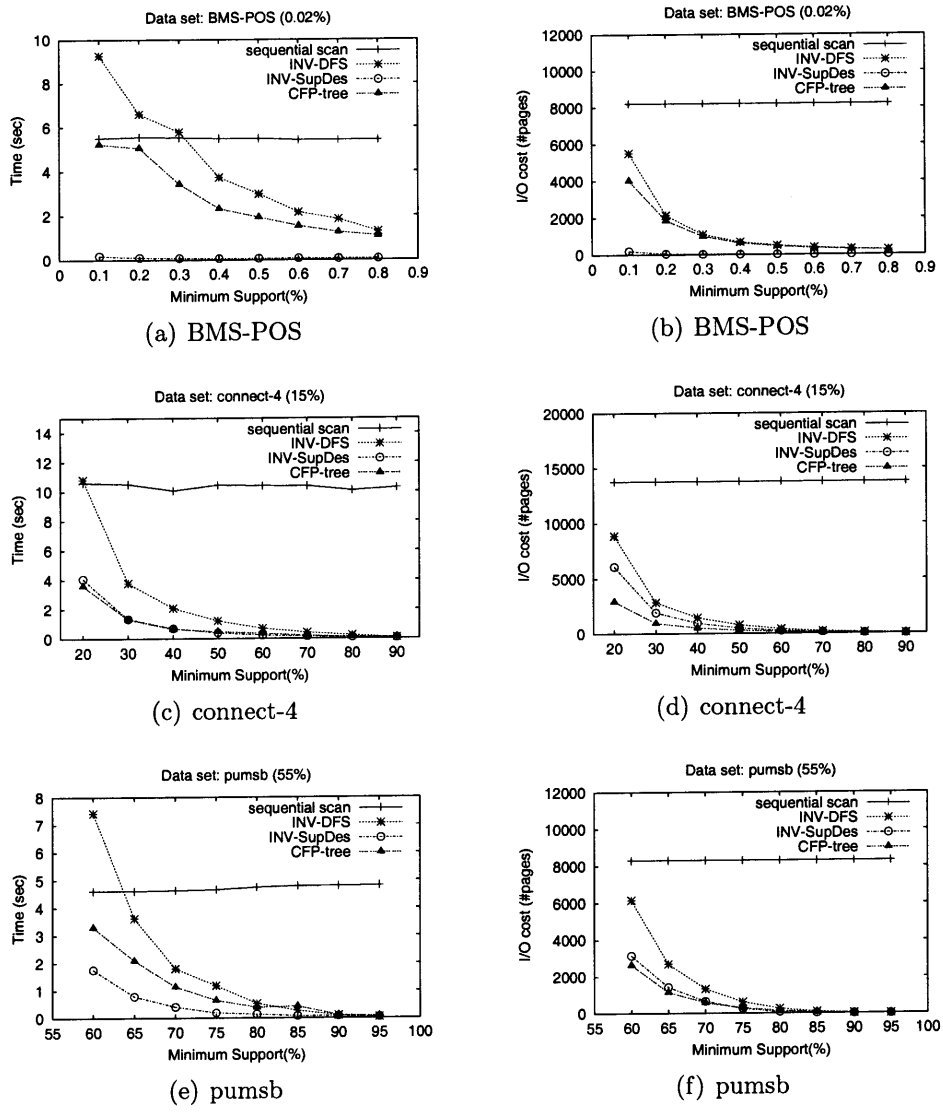


Figure 5.7: Queries with minimum support constraints

the item constraints. For queries with item constraints, the "CFP-tree" method and the "INV-DFS" method perform similarly, while the "INV-SupDes" method shows inferior performance. The reason being that retrieving frequent itemsets is still the major cost in "INV-SupDes" and "INV-DFS". Frequent itemsets containing the same item are scattered in different pages in the "INV-SupDes" method, but they are more clustered in the "INV-DFS" method.

5.5.4 Incremental update cost

We compared the efficiency of the CFP-tree incremental update algorithm with the CFP-tree construction algorithm which constructs a CFP-tree from scratch.

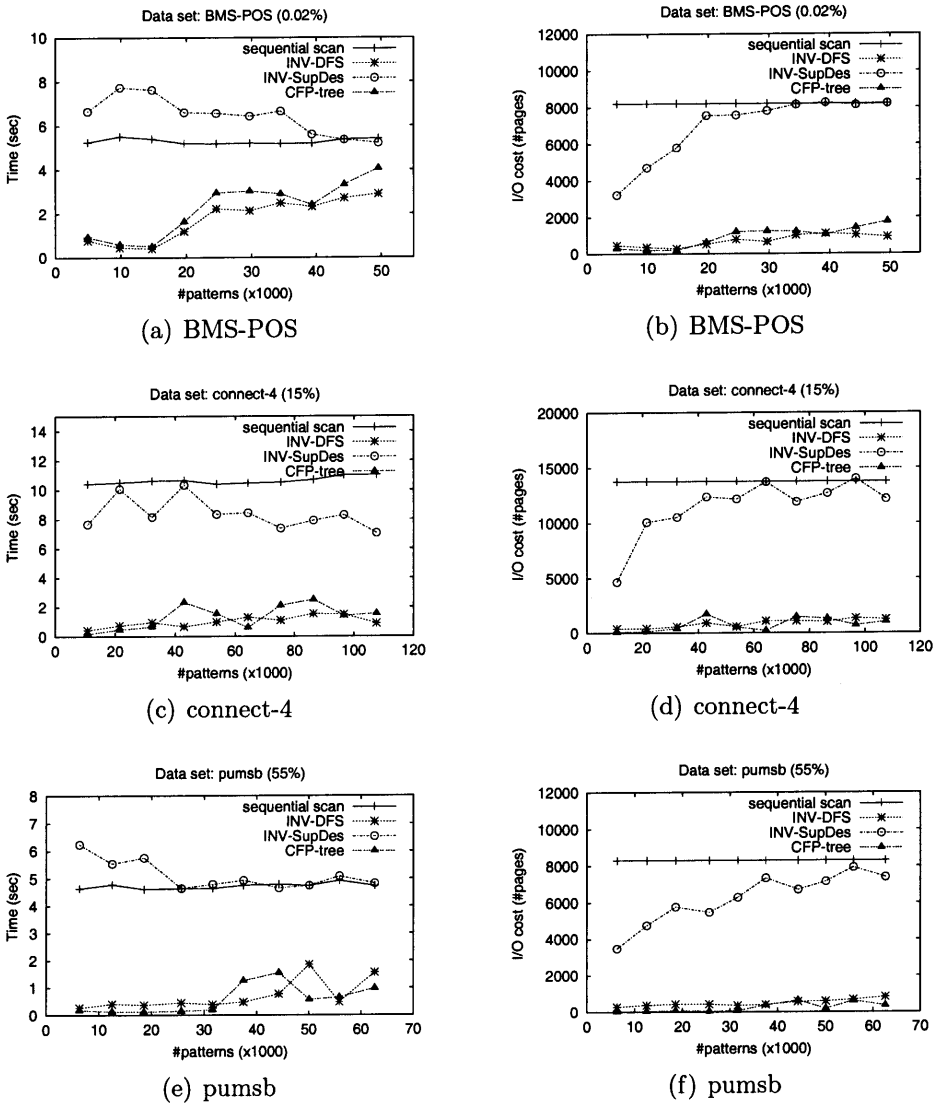
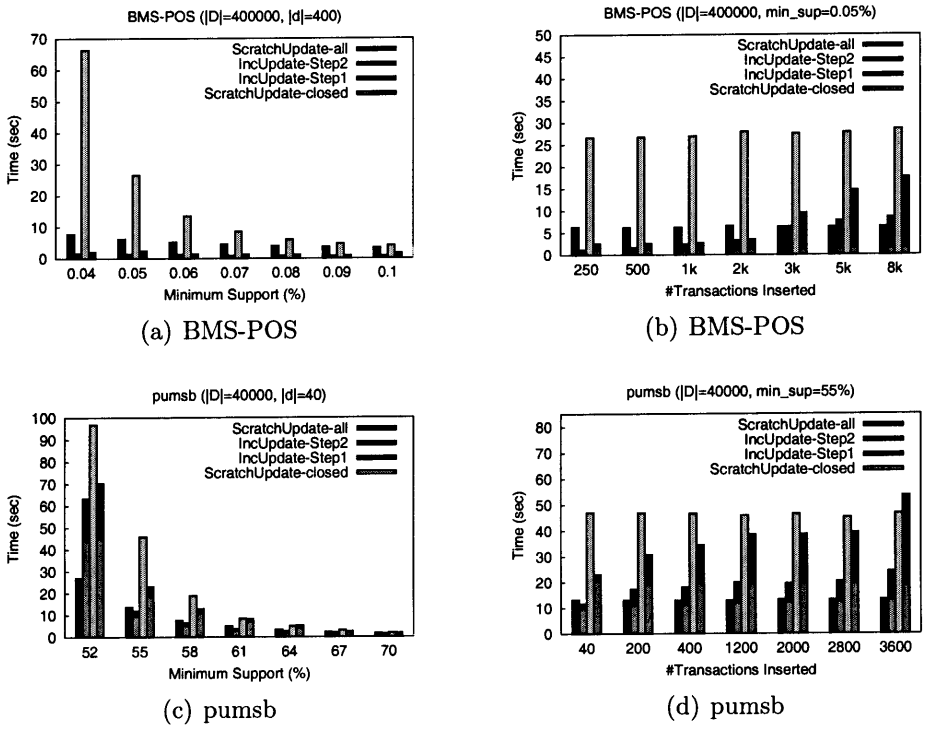


Figure 5.8: Queries with item constraints

The two largest datasets from Table 5.2, BMS-POS and pumsb, were used for performance study. The efficiency of the incremental update algorithm is affected by several factors, e.g. the minimum support threshold, the number of transactions to be inserted or deleted and the characteristics of the transactions to be inserted or deleted. We studied the effects of the first two factors on insertion and deletion respectively. For the insertion case, we randomly selected roughly 80% transactions as the original database, and randomly selected some transactions from the remaining transactions as the incremental database. For the deletion case, we used the whole database as the original database and randomly picked some transactions to delete.



(a) BMS-POS (b) BMS-POS

(c) pumsb (d) pumsb

Figure 5.9: Inserting new transactions

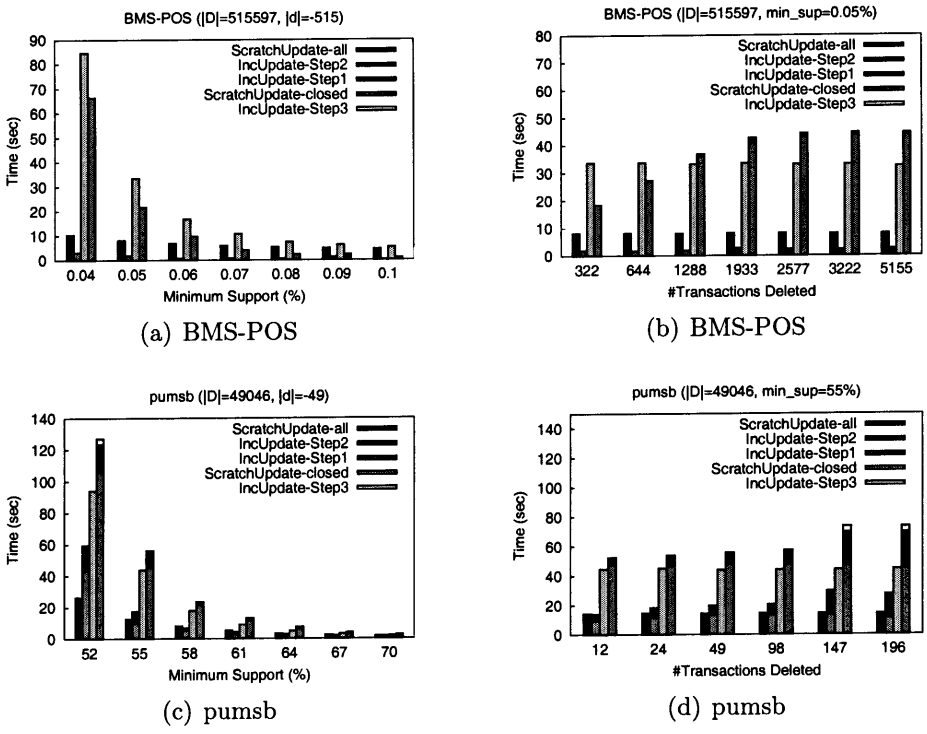
During our experiments, we found that the running time of the incremental update algorithm varied greatly when different sets of transactions were inserted or deleted even though the number of the transactions inserted or deleted and the minimum support thresholds were the same. To obtain reliable results, we generated 10 different sets of transactions randomly for each run and used the average running time as the final result. From left to right, the four columns in the figures represent the running time of constructing a CFP-tree$_{all}$ from scratch, incrementally updating a CFP-tree$_{all}$, constructing a CFP-tree$_{closed}$ from scratch and incrementally updating a CFP-tree$_{closed}$ respectively.

Figure 5.9 shows the running time of the incremental update algorithm with respect to the minimum support threshold and the size of incremental database when inserting new transactions. When varying the minimum support threshold, we set the size of the incremental database to be 0.1% of that of the original database. When varying the number of transactions inserted, the minimum support threshold was set to a fixed value as shown on the top of the figures.

We can make the following observations from Figure 5.9: (1) The incremental update algorithm can be significantly faster than the mining from scratch

method, especially on the sparse dataset BMS-POS. The smaller the incremental database, the higher the speed-up ratio of the incremental update algorithm. (2) When the number of the transactions inserted reaches certain thresholds, the incremental update algorithm is even slower than the update from scratch method. The reason being that compared with the update from scratch method, the incremental update algorithm requires additional cost to scan the original CFP-tree and maintain the negative border. Furthermore, when the number of new frequent itemsets is very large, constructing conditional databases for those new frequent itemsets is very time-consuming because subset matching is very costly. While in the update from scratch method, the conditional databases are constructed using the divide-and-conquer methodology and the subset matching operation is avoided. (3) The incremental update algorithm is not very beneficial on very dense dataset, e.g. pumsb. We observed the same scenario on other dense datasets. Most of the time was spent on updating the existing frequent itemsets (the first step). The reason being that on dense datasets, a few transactions can affect a large portion of the CFP-tree. (4) With the increase of the number of transactions inserted, both the cost for updating existing frequent itemsets (step 1) and the cost for generating new frequent itemsets (step 2) increase, but the cost for step 2 increases more quickly. The reason being that with the increase of the number of transactions inserted, more infrequent itemsets becomes frequent. For example, on dataset BMS-POS, there are averagely 89 new frequent itemsets when the number of transactions inserted is 250. However, when the number of transactions inserted is 8000, the average number of new frequent itemsets is 10191. Constructing conditional databases for such a large number of new frequent itemsets is very costly. (5) In case of insertion, incrementally updating a CFP-tree$_{closed}$ is more beneficial than incrementally updating a CFP-tree$_{all}$. The reason being that when inserting new transaction, the incremental update algorithm does not need to re-check those itemsets which are closed in the original database based on Lemma 10. The number of itemsets that are not closed in the original database and are not pruned by suffix sharing is very small. Therefore, the incremental update algorithm needs to check a very small number of itemsets in case of insertion.

Figure 5.10 shows the running time of the incremental update algorithm when deleting old transactions. We set the number of the transactions deleted to be



(a) BMS-POS (b) BMS-POS

(c) pumsb (d) pumsb

Figure 5.10: Deleting old transactions

0.1% of the size of the original database when varying the minimum support threshold. When updating a CFP-tree storing only frequent closed itemsets in case of deletion, the incremental update algorithm is not as efficient as it is for insertion. Updating existing frequent itemsets is the dominate factor. The reason being that the incremental update algorithm needs to re-check more frequent itemsets for deletion than for insertion in the first step. When deleting old transactions, a frequent itemset should be checked if the count of the frequent itemset is changed and the frequent itemset is closed in the original database. While for insertion, the incremental update algorithm need to re-check a frequent itemset if the count of the frequent itemset is changed and the frequent itemset is not closed in the original database. Table 5.3 shows that suffix sharing prunes most of the non-closed itemsets. Column "#checked" in Table 5.4 shows the total number of itemsets that are checked during the construction process. Column "#FCI" in Table 5.3 is the total number of frequent closed itemsets. Since every closed itemset should be checked, the difference between the above two columns is the number of non-closed itemsets that are checked. We can find that among the itemsets that are checked, most are closed itemsets. When deleting old transactions, the incremental update algorithm needs to re-check the closed itemsets

hence it needs to re-check many itemsets.

5.6 Summary

In this chapter, we propose a compact and efficient data structure, CFP-tree, for storing and querying frequent itemsets to support frequent itemset mining based applications.

The CFP-tree structure allows different itemsets to share the storage of their prefixes when storing frequent closed itemsets, and allows both prefix sharing and suffix sharing when storing all frequent itemsets. The space saved by suffix sharing is much more significant than prefix sharing. Therefore, the CFP-tree structure is an ideal structure for storing a complete set of frequent itemsets.

In the next chapter, we describe how to utilize the CFP-tree structure to generate association rules efficiently.

CHAPTER 6

FREQUENT ITEMSET CLASS: FACILITATING ONLINE ASSOCIATION RULE MINING

Many decision support systems need to support online interactive mining. One obstacle to online association rule mining is that the number of association rules grows explosively with respect to the minimum support threshold and the minimum confidence threshold. Generating too many rules not only requires extensive computation cost, but also defeats the primary purpose of data mining in the first place [3]. The concept of non-redundant association rule has been introduced to reduce the number of association rules by removing the redundancy in association rules [9]. An association rule is non-redundant if its left hand side is minimal and its right hand side is maximal among all the association rules that have the same support and confidence as it. It has been observed that a complete set of frequent itemsets contains lots of redundancy, which is the cause of the redundancy in association rules. As discussed in the previous chapter, mining frequent closed itemsets can effectively remove the redundancy in a complete set of frequent itemsets. However, frequent closed itemsets are not adequate for generating non-redundant association rules because they provide only the maximal right hand sides. We need to derive the set of minimal left-hand-side itemsets from the set of frequent closed itemsets, which is not a trivial task.

In this chapter, we present our approach towards online association rule mining. The central idea of our approach is the concept of *frequent itemset class.* Frequent itemsets in the same class occur in the same set of transactions, and they behave similarly in association rule generation. Two itemsets in the same class almost have the same set of right hand side itemsets. Thus the rule generation cost of the itemsets in the same class can be shared. To expedite the process of association rule generation, we define *class association rules* that represent the association among itemset classes. We propose efficient algorithms to generate class association rules in addition to an algorithm to efficiently compute frequent itemset classes.

6.1 Frequent Itemset Class

In Chapter 1, we defined the concepts of frequent itemset and traditional itemset association rule. The frequent itemsets discovered from the database shown in Figure 1.1(a) with minimum support of 40% are shown in Figure 1.1(b). Some of the frequent itemsets in Figure 1.1(b) can be inferred from other frequent itemsets. For example, knowing that c and cma have the same support 3, we can infer that the support of all the itemsets between c and cma (cm, ca) must also be 3 based on the apriori property. Thus those itemsets between c and cma are redundant, and they can be eliminated to reduce the result size. The redundancy is caused by the fact that the two itemsets c and cma are contained in the same set of transactions. For an itemset l, we use T_l^D to denote the set of transactions in a database D containing l, i.e. $T_l^D = \{t | t \in D \text{ and } l \subseteq t\}$. To identify the redundancy, we define a relation between itemsets as follows.

Definition 9 (Cover Equivalent) *Given two itemsets l_1 and l_2, if $T_{l_1}^D = T_{l_2}^D$, then l_1 and l_2 are called cover equivalent in D, denoted as $l_1 =_{cq} l_2$.*

According to the definition, two cover equivalent itemsets must have the same support, but not vice versa. Cover equivalent relation is reflexive, symmetric and transitive. Based on the cover equivalent relation, the set of frequent itemsets contained in a database D can be partitioned into disjoint classes such that all the frequent itemsets in the same class are cover equivalent, and any two itemsets from different classes are not cover equivalent.

Definition 10 (Frequent Itemset Class) *Given the set of frequent itemsets F with respect to min_sup in a database D, a partition $P = \{C_1, C_2, \cdots, C_m\}$ is a cover equivalent partition of F if (1) $\bigcup_{i=1}^{m} C_i = F$; (2) $\forall C_i$, C_j, $i \neq j$, $C_i \bigcap C_j = \phi$; (3) $\forall l_1$, $l_2 \in C_i$, l_1 and l_2 are cover equivalent; (4) $\forall l_1 \in C_i$, $l_2 \in C_j$, $i \neq j$, l_1 and l_2 are not cover equivalent. Each C_i is called a cover equivalent class or a frequent itemset class.*

The support of a class C in a database D is defined as $support_D(C) = support_D(l)$, $l \in C$. If $support_D(C) \geq min\_sup$, then C is called frequent in D. The maximal and minimal itemsets in a frequent itemset class form the upper bound and the lower bound of the class respectively. They are adequate

for representing the class, i.e. all the other itemsets in the class can be inferred from them without accessing any other information. The maximal itemset in a frequent itemset class is actually a frequent closed itemset. Here we give another definition of frequent closed itemset in the context of frequent itemset class. A frequent closed itemset must be the maximal itemset in its class.

Definition 11 (Closed Itemset) *A closed itemset l is the maximal itemset in a frequent itemset class, i.e. there does not exist l' such that $l' =_{cq} l$ and $l \subset l'$.*

Definition 12 (Base Itemset) *A base itemset l is a minimal itemset in a frequent itemset class, i.e. there does not exist l' such that $l' =_{cq} l$ and $l' \subset l$.*

According to the definition, the closed itemset in a frequent itemset class is unique, and it forms the upper bound of the class. The set of base itemsets of a class form the lower bound of the class.

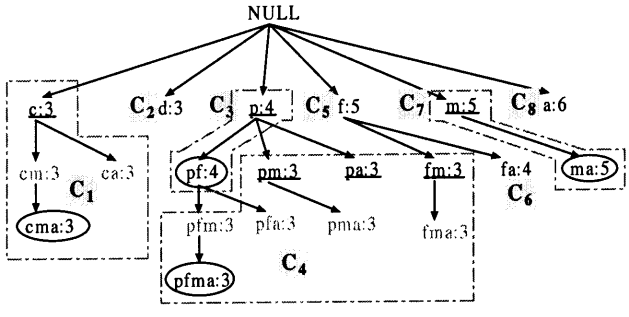


Figure 6.1: Cover equivalent classes

All the frequent itemsets shown in Figure 1.1(b) are partitioned into 8 classes as shown in Figure 6.1. The frequent itemset classes containing more than one itemsets are bordered by dotted lines. The base itemsets in these classes are underlined and the closed itemsets are circled. A class can be represented with its upper bound and lower bound. For example, class C_1 in Figure 6.1 can be represented as $[\{c\}, cma]$. For brevity, a singleton class is represented by the only itemset in it. For example, class C_2 can be written as $[d]$. In the rest of the paper, we use $C.closed$ to denote the closed itemset of class C, and use $C.baseset$ to denote the set of base itemsets of C. Given two classes C_1 and C_2, if $C_1.closed \subset C_2.closed$, then C_1 is called an *ancestor* class of C_2, and C_2 is a *descendant* class of C_1.

The itemsets in the same class behave similarly in association rule generation. Given two cover equivalent itemsets l_1 and l_2, whenever there is a rule $l_1 \rightarrow l$, there is another rule $l_2 \rightarrow l$ with the same support and confidence, where $l \bigcap l_1 = \phi$ and $l \bigcap l_2 = \phi$. The reason being that extending two cover equivalent itemsets by the same set of items leads to two cover equivalent itemsets. It is formally stated as follows.

Lemma 13 *Given* $l_1 =_{cq} l_2$, $\forall$ *itemset* l, $l_1 \bigcup l =_{cq} l_2 \bigcup l$.

Proof 12 *It can be obtained directly from the definition of cover equivalent:* $T_{l_1 \bigcup l} = T_{l_1} \bigcap T_l = T_{l_2} \bigcap T_l = T_{l_2 \bigcup l}$.

According to Lemma 13, the itemsets in the same class have the almost same set of RHS itemsets. It motives us to generate association rules among classes instead of itemsets to reduce the number of rules and save rule generation cost.

Definition 13 (Class Association Rule) *A class association rule is of the form* R: $C_1 \rightarrow C_2$, *where* C_1 *and* C_2 *are two frequent itemset classes and* $C_1.closed \subset C_2.closed$. *The support of a rule* R *in a database* D *is defined as* $support_D(R) = support_D(C_2)$. *The confidence of* R *is defined as* $confidence_D(R)$ $= support_D(C_2)/support_D(C_1)$. *Given a transaction database* D, *a minimum support threshold* min_sup *and a minimum confidence threshold* min_conf, *if* $support_D(R) \geq min\_sup$ *and* $confidence_D(R) \geq min\_conf$, *then* R *is called a valid class association rule in* D. *Class* C_1 *is called the left hand side (LHS) class of* R *and* C_2 *is called the right hand side (RHS) class of* R.

A class association rule can represent a set of itemset association rules if its LHS class and/or RHS class contains more than one itemsets, i.e. $R : C_1 \rightarrow C_2 \Leftrightarrow \{r : l_1 \rightarrow (l_2 - l_1) | l_1 \in C_1, l_2 \in C_2 \text{ and } l_1 \subset l_2\}$. A cover equivalent class C containing more than one itemset is regarded as a special type of class association rule. It represents a set of itemset association rules with confidence 100%, i.e. $C \Leftrightarrow \{r : l_1 \rightarrow (l_2 - l_1) | l_1, l_2 \in C \text{ and } l_1 \subset l_2\}$. With min_sup=40% and min_conf=70%, the number of valid itemset association rules in D is 29, while the number of valid class association rules is 9 as shown in Figure 6.2.

conf.	Class Association Rules
100%	[{c}, cma], [{p}, pf], [{pm,pa,fm}, pfma], [{m},ma]
83%	[a]->[{m}, ma]
80%	[f]->[fa], [f]->[pf]
75%	[{p}, pf]->[{pm,pa,fm}, pfma], [fa]->[{pm,pa,fm}, pfma]

Figure 6.2: Class association rules (min_sup=40%)

Lemma 14 *Given a minimum support threshold and a minimum confidence threshold, all the valid itemset association rules in a database D can be fully recovered from the valid class association rules in D.*

Compared with the itemset association rules, the class association rules are more concise and informative. Generating class association rules not only saves time and space, but also makes it easier for users to discover valuable information.

The non-redundant association rule defined in [9] can be generated from the base itemsets and the closed itemsets of the frequent itemset classes directly.

Definition 14 (Non-redundant Association Rule) *Itemset association rule $r : l_1 \rightarrow l_2$ is a non-redundant association rule if there does not exist another association rule $r' : l_1' \rightarrow l_2'$ such that $support(r) = support(r')$, $confidence(r) = confidence(r')$, $l_1' \subseteq l_1$ and $l_2' \supseteq l_2$.*

Lemma 15 *If $r : l_1 \rightarrow l_2$ is a non-redundant itemset association rule, then l_1 must be a base itemset and $l_1 \bigcup l_2$ must be a closed itemset, and vice versa.*

The above lemma is easy to prove. One advantage of class association rules is that they captures the relationships among the frequent itemsets, which is not true for non-redundant association rules. A class association rule can represent a set of non-redundant itemset association rules if its LHS class contains more than one base itemsets, i.e. $R : C_1 \rightarrow C_2 \Leftrightarrow \{r : l_1 \rightarrow (l_2 - l_1) | l_1$ *is a base itemset of* C_1 *and* l_2 *is the closed itemset of* $C_2\}$. Hence the number of class association rules can be smaller than the number of non-redundant itemset association rules.

6.2 Mining Frequent Itemset Classes

To mine frequent itemset classes, we not only need to mine both frequent closed itemsets and frequent base itemsets, but also need to associate them together according to the cover equivalent relationship. In this section, we propose a two-phase approach to computing frequent itemset classes. In the first phase, we compute frequent itemsets and store them in a CFP-tree. In the second phase, we utilize the CFP-tree to discover cover equivalent relationships among the frequent itemsets.

6.2.1 Computing frequent itemset classes

We utilize an algorithm similar to Algorithm 9 to compute all frequent itemsets and store them in a CFP-tree. During the computation, we identify the frequent closed itemsets using the techniques described in Section 5.2.2, and mark them in the CFP-tree to distinguish them from the non-closed itemsets.

In the second step, we traverse the CFP-tree in depth-first order from right to left. During the traversal, we compute the base itemsets for every closed itemset as follows. For each non-closed itemset l, we use it to update the candidate base itemsets of its cover equivalent closed itemset if l is a potential base itemset. A non-closed itemset is a potential base itemset if it is the shortest itemset represented by its end entry. The shortest itemset represented by an entry E contains only items in the nodes with multiple entries on the path from root to E and at most one item in E. Whether including one item in E depends on whether the support of E equals to that of its parent. For example, the shortest itemset represented by the path ended at node 2 in Figure 5.1(b) is c and the shortest itemset represented by the path ended at node 4 in Figure 5.1(b) is pm and pa.

According to the left containment property of the CFP-tree structure, all the subsets of an itemset are either on the right of the itemset or on the path of the itemset. Therefore, at the time a closed itemset l is visited using the depth-first and right-to-left traversal strategy, all the subsets of l have been visited and those cover equivalent to l have been used to update the base itemsets of l (line 4-5). In other words, when a closed itemset l is visited, the class border of l's class has been completely computed (line 1-2). The pseudo-code of the algorithm is shown

Algorithm 13 ComputeFIC Algorithm

Input:
cnode is a CFP-tree node;
l contains all the items on the path from root to *cnode*;
$L_{shortest}$ is the set of the shortest itemsets on the path from root to *cnode*;
parent_sup is the support of *cnode*'s parent entry;

Description:
1: **if** l is closed **then**
2: output class $[l.baseset, l]$;
3: **else**
4: Let q be the closed itemset cover equivalent to l;
5: Update q's base itemsets using $L_{shortest}$;
6: **if** *cnode*==NULL **then**
7: **return** ;
8: **else if** *cnode* contains only one entry E **then**
9: **if** $E.support = parent\_sup$ **then**
10: ComputeFIC($l \bigcup E.items$, $L_{shortest}$, $E.child$, $E.support$);
11: **else**
12: ComputeFIC($l \bigcup E.items$, $\{l \bigcup \{i\} | l \in L_{shortest} \wedge i \in E.items\}$, $E.child$, $E.support$);
13: **else**
14: **for all** entry $E \in$ *cnode*, from the last entry to the first entry **do**
15: ComputeFIC($l \bigcup E.items$, $\{l \bigcup E.items | l \in L_{shortest}\}$, $E.child$, $E.support$);

in Algorithm 13.

6.2.2 Maintaining frequent itemset classes

The maintenance of frequent itemset classes is basically similar to maintaining frequent closed itemsets as discussed in the previous chapter. The update of the base database may cause some classes to merge or split in addition to the occurrence of new classes and invalidation of existing classes. Merging two classes corresponds to the situation where a closed itemset become non-closed. Splitting a class corresponds to the situation where a non-closed itemset becomes closed. Therefore, inserting new transactions may cause creation of new classes, invalidation of existing classes and splitting of some existing classes, but never causes merging of existing classes. Deleting old transactions may cause creation of new classes, invalidation of existing classes and merging of existing classes, but never causes splitting of existing classes. Here we do not discuss the maintenance of frequent itemset classes further. Interested readers can refer to Chapter 5.

6.3 Generating Class Association Rules

In real applications, usually users are interested in a small number of association rules that satisfy certain constraints. We concentrate on one representative

type of user query: *find all class association rules r such that (1)support(r) ≥ min_sup, (2)confidence(r) ≥ min_conf, (3)the LHS class of r containing a given set of items I_L, and (4)the RHS class of r containing a given set of items I_R.* Itemsets I_L and I_R can be empty and they should contain no common items. The set of itemset association rules satisfying the given constraints can be derived from the generated class association rules. The task of finding all class association rules that satisfy the given constraints is essentially to find all class pairs (C_1, C_2), where $C_1.closed \subset C_2.closed$, $support(C_2) \geq min\_sup$, $support(C_2) \geq min\_conf \cdot support(C_1)$, $I_L \subseteq C_1.closed$ and $I_R \subseteq C_2.closed$. This process involves a self-join of the precomputed frequent itemset classes, which can be a costly operation if the number of classes is very large. In this section we use the CFP-tree structure to index the frequent itemset classes and propose two algorithms to solve the problem.

6.3.1 Storing and querying frequent itemset classes

Algorithm 14 Search_Class Algorithm

Input:
cnode is a CFP-tree node;
min_sup is the minimum support threshold;
I is a set of items to be contained in frequent itemset classes;
l contains all the items on the path from root to *cnode*;

Description:
1: **if** $I = \phi$ AND l is closed **then**
2: Output $[E_l.baseset, l] : E_l.support$;
3: **if** $I \neq \emptyset$ **then**
4: $\bar{E}$ = the first entry of *cnode* such that $\bar{E}.item \in I$;
5: **else**
6: $\bar{E}$ = the last entry of *cnode*;
7: E' = BinarySearch(*cnode*, *min_sup*);
8: **for all** entry $E \in cnode$, E between E' and $\bar{E}$ **do**
9: **if** $E.child \neq$ NULL AND all the items in $(I - E.items)$ are in $E.subtree$ **then**
10: Search_Class($E.child$, min_sup, $I - E.items$, $l \bigcup E.items$);
11: **else if** $(I - E.items) = \phi$ AND $(l \bigcup E.items)$ is closed **then**
12: Output $[E.baseset, l \bigcup E.items] : E.support$;

Frequent itemset classes should be stored in a way such that online association rule mining requests can be efficiently supported. In association rule generation, we have both support and item constraints. We have shown in the previous chapter that we can use the CFP-tree structure to efficiently process queries with minimum support and/or item constraints. Therefore, here we use the CFP-tree structure to index and query frequent itemset classes. The unique closed itemset of each class is chosen as the indexing key. We add one pointer at each CFP-tree

entry to point to the set of base itemsets of a class. We use $E.baseset$ to denote the set of base itemsets pointed by an entry E.

Algorithm 14 shows the pseudo-code for finding all frequent itemset classes with support no less than *min_sup* and containing a given set of items I. A class C is said to contain an itemset I if $I \subseteq C.closed$. Procedure BinarySearch(*cnode*, *min_sup*) returns the first entry in *cnode* whose support is no less than *min_sup*.

6.3.2 Nested loop join

A straightforward algorithm is to use the nested loop join, i.e. for every LHS class C with support $\geq min\_sup$ and containing I_L, search in the CFP-tree for all the RHS classes with support $\geq max\{min\_conf \cdot support(C), min\_sup\}$ and containing $C.closed \bigcup I_R$. Algorithm 15 shows the pseudo-code of the nested loop join algorithm. It calls Algorithm 14 to find the RHS classes for a given LHS class (line 2).

Algorithm 15 CFP_NLJ Algorithm

Input:
T is a CFP-tree
min_sup is the minimum support threshold
min_conf is the minimum confidence threshold
I_L is the set of items that must be in LHS class
I_R is the set of items that must be in RHS class

Description:
1: **for all** class C_L with support$\geq$ *min_sup* and containing I_L **do**
2: P = Search_Class(ϕ, T, $max\{min\_conf \cdot C_L.support, min\_sup\}$, $C_L.closed \bigcup I_R$);
3: **for all** $C_R \in P$ **do**
4: Output (C_L, C_R, $C_R.support$, $\frac{C_R.support}{C_L.support}$);

Algorithm 15 scans a CFP-tree (1+N) times, where N is the number of LHS classes satisfying the given constraints. Each time only a small portion of the tree is scanned because Algorithm 14 can utilize the pruning power of the apriori property, the left containment property and the hash bitmaps to avoid unnecessary scan cost. The available memory can be used to buffer some of the nodes to further reduce I/O cost. One heuristic is to choose the top-k largest nodes to buffer. The rationale behind this heuristic is that a node with more entries contains more items and it is very likely to be pointed by an entry with a high support. In other words such nodes are not easily pruned by the apriori property and the hash bitmap. A node, if it is not a singleton node, is larger than its child node. Hence caching the top-k largest nodes is essentially to buffer the

upper portion of the tree, which is more frequently visited. Another heuristic is to buffer the left portion of the tree because the left portion of the tree are visited more frequently according to the left containment property. In our experiments, we used the second heuristic. When using the second heuristic, it is unnecessary to buffer every class. It is sufficient to buffer only those classes that can appear on the right hand side of a rule because only these classes are visited multiple times.

6.3.3 Block nested loop join

Algorithm 15 can be costly when N is large. To reduce the number of scans, we can divide the set of LHS classes satisfying the given constraints into several blocks such that each block can be kept in the memory. The tree is scanned one time for each block to find the RHS classes for each LHS class in the block. The set of LHS classes are partitioned according to their depth-first order, i.e. those classes that are consecutively visited when searching the tree in depth-first order are grouped together. For each block, none of the classes after the last class in the block is visited. Algorithm 16 shows the pseudo-code.

Algorithm 16 CFP_BNLJ Algorithm

Input:
same as Algorithm 15;
Description:
1: P = the first block of LHS classes in T satisfying given constraints;
2: **while** $P \neq \phi$ **do**
3: BNL_GenRules(ϕ, $T.root$, P);
4: P = the next block of LHS classes in T satisfying given constraints;

At each tree node, if the searching condition is set to the disjunction of the RHS constraints of each LHS class in the block, the algorithm can incur vast computation cost compared with Algorithm 15. The reason being that the RHS constraints of a LHS class C are checked at each node being visited using the above method, while some of the nodes will never been visited if the RHS classes of C are searched individually because some upper node dissatisfies the RHS constraints of C. One optimization is to recursively partition the set of LHS classes at each node. Only when the RHS constraints of a LHS class C are satisfied by an entry E, the RHS classes of C are searched in the subtree pointed by E.

Procedure BNL_GenRules(ϕ, T, P) returns the set of association rules whose LHS classes are in block P. At each node, it recursively distributes LHS classes in P to its children according to the search constraints. Its pseudo-code is shown in Algorithm 17. For an entry E, $E.list$ contains the set of LHS classes to be searched in the subtree pointed by E. For a LHS class C, $C.I$ contains the set of items in $C.closed \bigcup I_R$ that have not been found yet. Initially, $C.I$=$C.closed \bigcup I_R$.

Algorithm 17 BNL_GenRules Algorithm

Input:
cnode is a CFP-tree node;
P is the set of LHS classes;
l contains all the items on the path from root to *cnode*;

Description:
1: C_R = the class of l;
2: **for all** class $C_L \in P$ **do**
3: **if** $C_L.I = \phi$ AND (l is closed) **then**
4: Output (C_L, C_R, $C_R.support$, $\frac{C_R.support}{C_L.support}$);
5: **if** $C_L.I \neq \phi$ **then**
6: $\bar{E}$ = the first entry of *cnode* s.t. $\bar{E}.item \in C_L.I$;
7: **else**
8: $\bar{E}$ = the last entry of *cnode*;
9: E' = BinarySearch(*cnode*, $min\_conf \cdot C_L.support$);
10: **for all** entry E between $\bar{E}$ and E' **do**
11: $E.list = E.list \bigcup \{C_L\}$;
12: **for all** entry $E \in$ *cnode* **do**
13: **for all** class $C_L \in E.list$ **do**
14: $C_L.I = C_L.I - E.item$;
15: **if** $E.child \neq$NULL AND $E.list \neq \phi$ **then**
16: BNL_GenRules($l \bigcup E.item$, $E.child$, $E.list$);
17: **else if** $E.list \neq \phi$ **then**
18: **for all** class $C_L \in E.list$ **do**
19: **if** $C_L.I = \phi$ **then**
20: Output (C_L, C_R, $C_R.support$, $\frac{C_R.support}{C_L.support}$);

The tree scan times of Algorithm 16 is $1 + N_B$, where N_B is the number of blocks. In each scan, the portion of the tree being visited is the union of the trees being visited if each LHS class in the block is searched individually. Algorithm 15 and Algorithm 16 use a buffer for different purposes. Algorithm 15 uses a buffer to reduce the cost of a single scan, while Algorithm 16 uses a buffer to reduce the number of scans.

6.4 A Performance Study

We conducted the experiments on a 2.26Ghz Pentium IV with 512MB memory running Microsoft Windows XP. All codes were complied using Microsoft Visual C++ 6.0. Table 6.1 shows the several datasets used for performance study.

Datasets	Size	#Trans	#Items	MaxTL	AvgTL
BMS-POS	11.62MB	515597	1657	165	6.53
BMS-WebView-1	1.28M	59601	497	267	2.51
chess	0.34M	3196	75	37	37.00
connect-4	9.11M	67557	129	43	43.00
mushroom	0.56M	8124	119	23	23.00
pumsb	16.30M	49046	2113	74	74.00
pumsb_star	11.03MB	49046	2088	63	50.48

Table 6.1: Datasets

6.4.1 The reduction in the number of rules

The first experiment is to study the reduction in the number of rules when organizing itemsets into classes. Table 6.2 lists the number of frequent itemset classes, the number of frequent itemsets, the number of class rules ("#c_rule" column) and the number of itemset rules ("#i_rule" column) on several datasets. We set the minimum confidence threshold to 80% on all datasets. The number of class association rules can be dramatically smaller than the number of itemset association rules. For example, on dataset mushroom, the number of class association rules is more than 2000 times smaller than the number of itemset rules. The reduction ratio mainly depends on the reduction ratio of frequent itemset classes. Here we set minimum support thresholds to relatively high values. When the minimum support thresholds are lowered, the reduction ratio can be much more dramatic.

Datasets	min_sup	#class	#FI	ratio	#c_rule	#i_rule	ratio
BMS-POS	0.1%	122918	123001	1.001	84505	84559	1.001
BMS-WebView-1	0.07%	24329	36008	1.48	82730	692241	8.37
chess	65%	49241	111779	2.27	6132915	18206471	2.97
connect-4	85%	8256	142272	17.23	1349555	43896880	32.53
mushroom	15%	2261	98576	43.60	11740	25849274	2201.81
pumsb	80%	33308	142195	4.27	3697562	28276846	7.65
pumsb_star	35%	6134	117018	19.08	488239	45062496	92.30

Table 6.2: Number of class rules and itemset rules

The number of class association rules can be smaller than the number of non-redundant itemset association rules. Table 6.3 shows the number of classes, the number of classes containing more than one itemsets ("#mi_class" column), the

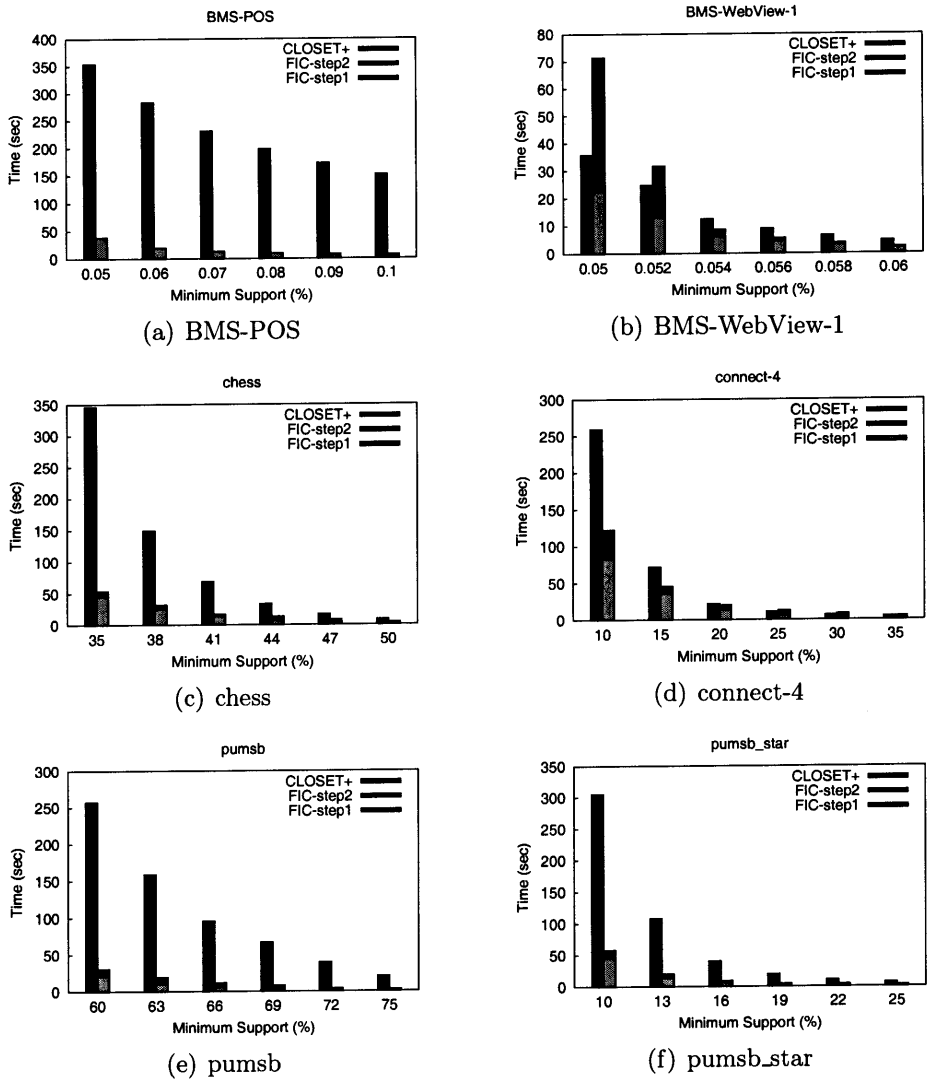
Datasets	min_sup	#class	#mi_class	#mb_class	#c_rule	#nr_irule	ratio
BMS-POS	0.08%	201098	574	0	150925	150925	1
BMS-WebView-1	0.06%	76260	29486	12824	1264968	1543080	1.22
chess	65%	49240	29488	0	6132915	6132915	1
connect-4	80%	15111	14984	0	3964181	3964181	1
mushroom	1%	51672	51672	16062	179215	277149	1.55
pumsb	70%	241258	195341	187850	92554733	190587691	2.06
pumsb_star	30%	16158	13132	6364	1308625	1859063	1.42

Table 6.3: Number of class rules and non-redundant itemset rules

number of classes with more than one base itemsets ("#mb_class" column), the number of class association rules ("#c_rule" column) and the number of non-redundant association rules ("#nr_irule" column). When there is no frequent itemset class which contains more than one base itemsets, the number of class association rules is exactly the same as the number of non-redundant association rules, e.g. on datasets BMS-POS, chess and connect-4. If there exist some classes with more than one base itemsets, then the number of class rules is smaller than the number of non-redundant itemset rules, e.g. on datasets BMS-WebView-1, mushroom, pumsb and pumsb_star.

6.4.2 Computation time of frequent itemset classes

We compared the frequent itemset class computation algorithm with the frequent closed itemset mining algorithm CLOSET+ [93]. Figure 6.3 shows the running time of the two algorithms on several datasets. Compared with the CLOSET+ algorithm, the frequent itemset class computation algorithm requires additional cost for computing frequent base itemsets and associating them with their cover equivalent closed itemsets in the second step. We show the time for mining all the frequent itemsets (the first step) and computing the base itemsets (the second step) separately in the figures. They are denoted as "FIC-step1" and "FIC-step2" respectively. The cost of the second step mainly depends on the number of non-closed itemsets stored in the CFP-tree that are potential base itemsets. On dataset BMS-WebView-1, the frequent itemset class computation algorithm requires a relative long time to compute the base itemsets, which is consistent with Table 5.1.



(a) BMS-POS (b) BMS-WebView-1

(c) chess (d) connect-4

(e) pumsb (f) pumsb_star

Figure 6.3: Frequent itemset classes computation time

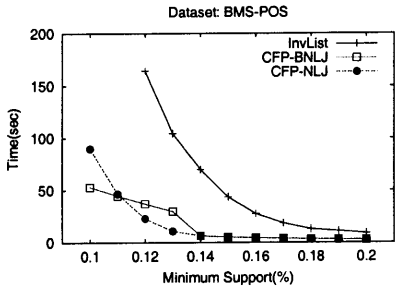
6.4.3 Rule generation time

The inverted file structure are a popular indexing structure for set-valued data. We compared our rule generation algorithms on the CFP-tree structure with a rule generation algorithm using inverted files, denoted as "InvList". The inverted files are constructed on the closed itemsets using the second sorting strategy as described in Section 5.5.3. The rule generation algorithm first uses the inverted files to retrieve LHS classes, then retrieves the RHS classes for each LHS class.

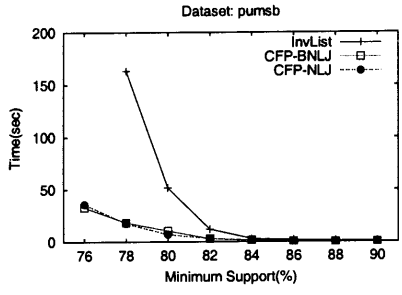
We studied the effects of various parameters on the rule generation algorithms. The studied parameters are the minimum support threshold, the minimum confidence threshold, the buffer size and the size of the precomputed CFP-tree. When

Data Sets	min_sup	min_conf	buf_size	tree_size	#rules
BMS-POS	0.2%	80%	1MB	10.02MB	53787
Pumsb	80%	80%	0.5MB	11.65MB	2259346

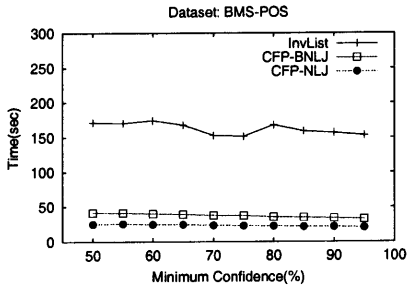
Table 6.4: Fixed values for four parameters



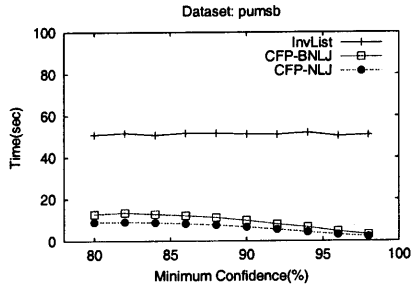
(a) BMS-POS



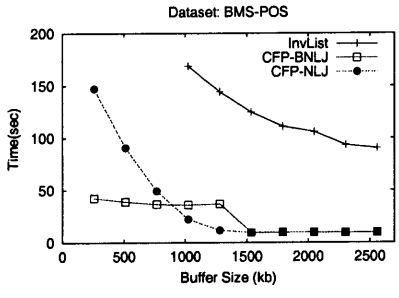
(b) pumsb



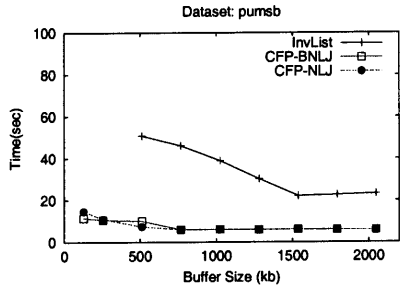
(c) BMS-POS



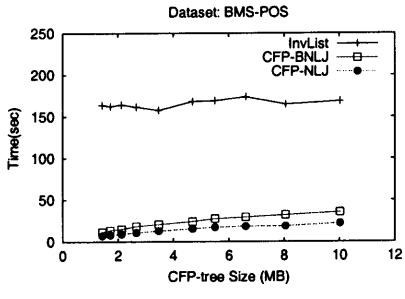
(d) pumsb



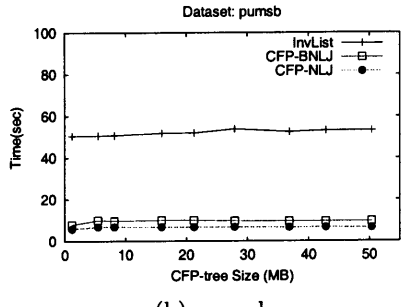
(e) BMS-POS



(f) pumsb



(g) BMS-POS



(h) pumsb

Figure 6.4: Rule generation time

varying a parameter, all the other parameters are set to a fixed value. Table 6.4 shows the fixed value for each parameter on two datasets when varying the other parameters. The last column is the number of rules generated when all four parameters are set to their fixed values. We did not put item constraints on association rules.

Figure 6.4 shows the running time of the rule generation algorithms when varying the four parameters separately. When the minimum support threshold is relatively high ($\geq$0.13% on BMS-POS and $\geq$82% on pumsb), all the possible RHS classes (those classes satisfying predefined constraints) can be held in the buffer. The two rule generation algorithms on the CFP-tree structure perform similarly. They both load the RHS classes in the buffer and searches RHS classes in the memory for each LHS class. Similarly, when the size of the buffer reaches certain threshold ($\geq$1.5MB on BMS-POS and $\geq$768kb on pusmb), the buffer is large enough to hold all possible RHS classes. The increase of the buffer size does not benefit the performance of the two algorithms on the CFP-tree structure. The running time of the rule generation algorithms is relatively steady with respect to the minimum confidence threshold and the size of the CFP-tree.

When all the possible RHS classes cannot be held in the buffer, the running time of the CFP-NLJ algorithm has a sharp increase with the decrease of the minimum support and the decrease of the buffer size. The performance of the CFP-BNLJ algorithm is more stable than that of CFP-NLJ.

6.5 Summary

In this chapter, we propose the concept of frequent itemset class to facilitate online association rule mining.

Organizing frequent itemsets into classes can reduce the number of rules dramatically. The number of class association rules can be even smaller than the number of non-redundant itemset association rules. Meanwhile all frequent itemsets and association rules can be recovered from frequent itemset classes and class association rules at very little cost, while for other concepts that aim to reduce the number of frequent itemsets, considerable cost is needed to recover all frequent itemsets. For example, to derive all frequent itemsets from a set of frequent

closed itemsets, a self-join of the set of frequent closed itemsets is needed. The original database is needed to derive all itemsets from generators. Therefore, frequent itemset classes are superior to frequent closed itemsets and generators in supporting association rule mining based applications.

CHAPTER 7

CONCLUSION

7.1 Summary of The Thesis

Frequent itemset mining is a fundamental and important problem in the data mining area. Many variations and new problems have been proposed based on the basic association rule mining problem. In this dissertation, we focused on the basic association rule mining problem, and proposed several techniques to support efficient and scalable frequent itemset mining.

We investigated the key factors of a frequent itemset mining algorithm, and proposed an adaptive algorithm AFOPT for efficient mining of frequent itemsets. We introduced a simple and compact structure AFOPT to represent conditional databases. Items are sorted into ascending frequency order in the AFOPT structure. The same order is used for search space exploration in the AFOPT algorithm. Therefore, the AFOPT structure can be traversed top-down instead of bottom-up like the FP-tree structure. The top-town traversal strategy is more efficient than the bottom-up traversal strategy for tree structures. For top-down traversal, only a child pointer and a sibling pointer should be maintained at each node, and every node is visited exactly once. Two additional pointers—a parent pointer and a node-link pointer have to be maintained at each node for bottom-up traversal. Furthermore, the visiting times of a node equals the number of the descendants of the node using the bottom-up traversal strategy. Besides the AFOPT structure, the AFOPT algorithm uses two other structures to represent conditional databases. The AFOPT algorithm uses some simple heuristics to choose a proper representation for a specific conditional database. It is desirable to develop more sophisticated methods to achieve better performance.

The AFOPT algorithm is dedicated to in-core mining. We proposed a scalable approach SSP to out-of-core mining. The SSP approach performs exactly like the AFOPT algorithm when all the conditional databases can fit into the main memory. When the conditional databases are too large to fit into the main

memory, the SSP approach uses a search space based partitioning technique to partition the database, which is different from the previously proposed data based partitioning technique. In the search space based partitioning, a transaction can belong to multiple partitions but a frequent itemset can belong to one and only one partition. Consequently, a local frequent itemset is also a global itemset. There is no need to maintain local frequent itemsets and scan the original database to verify them as in the data based partitioning algorithms. We developed three algorithms that utilize overlap among data partitions to reduce I/O cost. Our experiment results show that the SSP approach outperforms previous out-of-core mining algorithm significantly, especially when the number of frequent itemsets is large.

Many frequent itemset mining algorithms have been proposed since the problem was introduced. However, frequent itemset mining is still a time-consuming process despite all the efforts devoted. Hence, it is impossible to use "mining on demand" strategy to support online interactive mining on very large databases. We proposed a compact disk-based structure CFP-tree to manage frequent itemsets and support online mining requests. The CFP-tree structure is an ideal structure for storing a complete set of frequent itemsets because it allows both prefix sharing and suffix sharing. When storing a complete set of frequent itemsets, the compression ratio of a CFP-tree can be as high as several thousands or even several millions. We have developed efficient algorithms to retrieve frequent itemsets and association rules satisfying user-specified constraints from a CFP-tree, as well as efficient algorithms to construct and incrementally update a CFP-tree.

The number of frequent itemsets and association rules can be unacceptably large. We proposed the concept of frequent itemset class and class association rule to reduce result size without information loss. The number of class association rules can even be significantly smaller than the number of non-redundant itemset association rules. A salient feature of frequent itemset classes and class association rules is that all frequent itemsets and association rules can be derived from frequent itemset classes and class association rules with little overhead.

In summary, we have proposed efficient and scalable algorithms to mine frequent itemsets, introduced a compact disk-based structure CFP-tree to store fre-

quent itemsets, developed efficient algorithms to retrieve frequent itemsets and association rules from a CFP-tree and efficient algorithms to incrementally maintain frequent itemsets stored in a CFP-tree. We also proposed the concept of frequent itemset class and class association rule to reduce result size without information loss and recovery overhead. Putting all these techniques together, we can build a system to support many frequent itemset mining based applications.

7.2 Future Research Directions

My future research work will focus on two directions.

One interesting problem is to integrate frequent itemset mining techniques with classification techniques to improve classification performance. Traditional rule based classification algorithms, e.g. decision tree, use a greedy strategy to find classification rules. It may not be possible to find the true classification structure in the data due to the local search strategy. Association rule mining searches exhaustively for all rules satisfying a user-specified minimum support and minimum confidence threshold. The key problem is how to select an appropriate set of association rules to build a high-accurate classifier. The CFP-tree structure proposed in this dissertation is a powerful tool for analyzing frequent itemsets and association rules. With its assistance, hopefully we can develop a highly accurate classifier.

Another interesting direction is to study data mining problems in the bioinformatics area. In the past few years, research in molecular biology and molecular medicine has accumulated enormous amounts of data. However, the understanding of the biological processes underlying these data lags far behind. There is a strong interest in employing data mining methods to these biological data. Mining biological databases imposes some new challenges on knowledge discovery and data mining methods, e.g. extremely high dimensions and relatively few samples, which calls for new data mining techniques.

REFERENCES

[1] R. C. Agarwal, C. C. Aggarwal, and V. Prasad. A tree projection algorithm for finding frequent itemsets. *Journal on Parallel and Distributed Computing*, 61(3):350–371, 2001.

[2] R. C. Agarwal, C. C. Aggarwal, and V. V. V. Prasad. Depth first generation of long patterns. In *Proc. of the 6th ACM SIGKDD Conference*, pages 108–118, 2000.

[3] C. C. Aggarwal and P. S. Yu. Online generation of association rules. In *Proc. of the 14th IEEE ICDE Conference*, pages 402–411, 1998.

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

[5] R. Agrawal and R. Srikant. Fast algorithms for mining association rules in large databases. In *Proc. of the 20th VLDB Conference*, pages 487–499, 1994.

[6] R. Agrawal and R. Srikant. Mining sequential patterns. In *Proc. of the 11th ICDE Conference*, pages 3–14, 1995.

[7] Y. Aumann and Y. Lindell. A statistical theory for quantitative association rules. In *Proc. of the 5th ACM SIGKDD Conference*, pages 261–270, 1999.

[8] N. F. Ayan, A. U. Tansel, and M. E. Arkun. An efficient algorithm to update large itemsets with early pruning. In *Proc. of the 5th SIGKDD Conference*, pages 287–291, 1999.

[9] Y. Bastide, N. Pasquier, R. Taouil, G. Stumme, and L. Lakhal. Mining minimal non-redundant association rules using frequent closed itemsets. In *Proc. of Computational Logic Conference*, pages 972–986, 2000.

[10] Y. Bastide, R. Taouil, N. Pasquier, G. Stumme, and L. Lakhal. Mining frequent patterns with counting inference. *SIGKDD Exploration*, 2(2):66–75, 2000.

[11] T. Brijs, G. Swinnen, K. Vanhoof, and G. Wets. Using association rules for product assortment decisions: a case study. In *Proc. of the 5th ACM SIGKDD Conference*, pages 254–260, 1999.

[12] S. Brin, R. Motwani, J. D. Ullman, and S. Tsur. Dynamic itemset counting and implication rules for market basket data. In *Proc. of the 1997 ACM SIGMOD Conference*, pages 255–264, 1997.

[13] C. Bucila, J. Gehrke, D. Kifer, and W. M. White. Dualminer: a dual-pruning algorithm for itemsets with constraints. In *Proc. of the 8th ACM SIGKDD Conference*, pages 42–51, 2002.

[14] D. Burdick, M. Calimlim, and J. Gehrke. Mafia: A maximal frequent itemset algorithm for transactional databases. In *Proc. of the 17th IEEE ICDE Conference*, pages 443–452, 2001.

[15] A. Bykowski and C. Rigotti. A condensed representation to find frequent patterns. In *Proc. of the 20th PODS Symposium*, 2001.

[16] C. Carpineto and G. Romano. Galois: An order-theoretic approach to conceptual clustering. In *Proc. of the 10th ICML Conference*, 1993.

[17] M.-S. Chen, J. S. Park, and P. S. Yu. Efficient data mining for path traversal patterns. *IEEE Transactions on Knowledge and Data Engineering*, 10(2):209–221, 1998.

[18] D. W. Cheung, J. Han, V. T. Ng, and C. Wong. Maintenance of discovered association rules in large databases: An incremental updating technique. In *Proc. of the 12th IEEE ICDE Conference*, pages 106–114, 1996.

[19] D. W.-L. Cheung, S. D. Lee, and B. Kao. A general incremental technique for maintaining discovered association rules. In *Proc. of the 5th DASFAA Conference*, pages 185–194, 1997.

[20] E. Cohen, M. Datar, S. Fujiwara, A. Gionis, P. Indyk, R. Motwani, J. D. Ullman, and C. Yang. Finding interesting associations without support pruning. In *Proc. of the 16th ICDE Conference*, pages 489–499, 2000.

[21] G. Cong and B. Liu. Speed-up iterative frequent itemset mining with constraint changes. In *Proc. of the 2002 ICDM Conference*, pages 107–114, 2002.

[22] G. Cong, A. K. H. Tung, X. Xu, F. Pan, and J. Yang. Farmer: Finding interesting rule groups in microarray datasets. In *Proc. of the 2004 ACM SIGMOD Conference*, pages 143–154, 2004.

[23] B. W. T. David Wai-Lok Cheung, Vincent Ng. Maintenance of discovered knowledge: A case in multi-level association rules. In *Proc. of the 2nd ACM SIGKDD Conference*, pages 307–310, 1996.

[24] R. Feldman, Y. Aumann, A. Amir, and H. Mannila. Efficient algorithms for discovering frequent sets in incremental databases. In *Workshop on Research Issues on Data Mining and Knowledge Discovery*, 1997.

[25] T. Fukuda, Y. Morimoto, S. Morishita, and T. Tokuyama. Constructing efficient decision trees by using optimized numeric association rules. In *Proc. of the 22th VLDB Conference*, pages 146–155, 1996.

[26] T. Fukuda, Y. Morimoto, S. Morishita, and T. Tokuyama. Data mining using two-dimensional optimized accociation rules: Scheme, algorithms, and visualization. In *Proc. of the 1996 ACM SIGMOD Conference*, pages 13–23, 1996.

[27] B. C. M. Fung, K. Wang, and M. Ester. Hierarchical document clustering using frequent itemsets. In *Proc. of SIAM International Conference on Data Mining*, 2003.

[28] B. Goethals and M. J. Zaki. Advances in frequent itemset mining implementations: Introduction to fimi03. In *FIMI'03 workshop*, 2003.

[29] K. Gouda and M. J. Zaki. Efficiently mining maximal frequent itemsets. In *Proc. of the 2001 IEEE ICDM conference*, pages 163–170, 2001.

[30] G. Grahne, L. V. S. Lakshmanan, and X. Wang. Efficient mining of constrained correlated sets. In *Proc. of the 16th ICDE Conference*, pages 512–521, 2000.

[31] J. Han, G. Dong, and Y. Yin. Efficient mining of partial periodic patterns in time series database. In *Proc. of the 15th ICDE Conference*, pages 106–115, 1999.

[32] J. Han and Y. Fu. Discovery of multiple-level association rules from large databases. In *Proc. of the 21th VLDB Conference*, pages 420–431, 1995.

[33] J. Han, W. Gong, and Y. Yin. Mining segment-wise periodic patterns in time-related databases. In R. Agrawal, P. E. Stolorz, and G. Piatetsky-Shapiro, editors, *Proc. of the 4th ACM SIGKDD Conference*, pages 214–218, 1998.

[34] J. Han and J. Pei. Mining frequent patterns by pattern-growth: Methodology and implications. *SIGKDD Explorations*, 2(2):14–20, 2000.

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

[36] J. Han, J. Wang, Y. Lu, and P. Tzvetkov. Mining top-k frequent closed patterns without minimum support. In *Proc. of the 2002 ICDM Conference*, pages 211–218, 2002.

[37] S. Helmer and G. Moerkotte. A performance study of four index structures for set-valued attributes of low cardinality. *The VLDB Journal*, 12(3):244–261, 2003.

[38] A. Icev, C. Ruiz, and E. F. Ryder. Distance-enhanced association rules for gene expression. In *Proc. of the 3nd BIOKDD Workshop*, pages 34–40, 2003.

[39] A. Inokuchi, T. Washio, and H. Motoda. An apriori-based algorithm for mining frequent substructures from graph data. In *Proc. of the 4th PKDD Conference*, pages 13–23, 2000.

[40] M. L. Jos Borges. Mining association rules in hypertext databases. In *Proc. of the 4th ACM SIGKDD Conference*, pages 149–153, 1998.

[41] R. J. B. Jr. Efficiently mining long patterns from databases. In *Proc. of the 1998 ACM SIGMOD Conference*, pages 85–93, 1998.

[42] L. V. S. Lakshmanan, C. K.-S. Leung, and R. T. Ng. Efficient dynamic mining of constrained frequent sets. *ACM Transactions on Database Systems (TODS)*, 28(4), 2003.

[43] L. V. S. Lakshmanan, R. T. Ng, J. Han, and A. Pang. Optimization of constrained frequent set queries with 2-variable constraints. In *Proc. of the 1999 ACM SIGMOD Conference*, pages 157–168, 1999.

[44] L. V. S. Lakshmanan, J. Pei, and J. Han. Quotient cube: How to summarize the semantics of a data cube. In *Proc. of the 28th VLDB Conference*, pages 778–789, 2002.

[45] B. Lan, B. C. Ooi, and K.-L. Tan. Efficient indexing structures for mining frequent patterns. In *Proc. of the 18th IEEE ICDE Conference*, pages 453–462, 2002.

[46] S. D. Lee and D. W.-L. Cheung. Maintenance of discovered association rules: When to update? In *Workshop on Research Issues on Data Mining and Knowledge Discovery*, 1997.

[47] C. K.-S. Leung, R. T. Ng, and H. Mannila. Ossm: A segmentation approach to optimize frequency counting. In *Proc. of the 18th IEEE ICDE Conference*, pages 583–592, 2002.

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

[49] J.-L. Lin and M. H. Dunham. Mining association rules: Anti-skew algorithms. In *Proc. of the 14th ICDE Conference*, pages 486–493, 1998.

[50] B. Liu, W. Hsu, and Y. Ma. Integrating classification and association rule mining. In *Proc. of the 4th ACM SIGKDD Conference*, pages 80–86, 1998.

[51] G. Liu, H. Lu, W. Lou, Y. Xu, and J. X. Yu. Efficient mining of frequent patterns using ascending frequency ordered prefix-tree. *Data Mining and Knowledge Discovery Journal*, 9(3):249–274, 2004.

[52] G. Liu, H. Lu, W. Lou, and J. X. Yu. On computing, storing and querying frequent patterns. In *Proc. of the 9th ACM SIGKDD Conference*, pages 607–612, 2003.

[53] G. Liu, H. Lu, Y. Xu, and J. X. Yu. Ascending frequency ordered prefix-tree: Efficient mining of frequent patterns. In *Proc. of the 8th DASFAA Conference*, pages 65–72, 2003.

[54] J. Liu, Y. Pan, K. Wang, and J. Han. Mining frequent item sets by opportunistic projection. In *Proc. of the 8th ACM SIGKDD Conference*, pages 229–238, 2002.

[55] H. Mannila and H. Toivonen. Discovering generalized episodes using minimal occurrences. In *Proc. of the 2nd ACM SIGKDD Conference*, pages 146–151, 1996.

[56] H. Mannila, H. Toivonen, and A. I. Verkamo. Efficient algorithms for discovering association rules. In *KDD workshop*, pages 181–192, 1994.

[57] H. Mannila, H. Toivonen, and A. I. Verkamo. Discovery of frequent episodes in event sequences. *Data Mining and Knowledge Discovery Journal*, 1(3):259–289, 1997.

[58] F. D. Marchi, S. Lopes, and J.-M. Petit. Efficient algorithms for mining inclusion dependencies. In *Proc. of the 8th EDBT Conference*, pages 464–476, 2002.

[59] D. Meretakis, D. Fragoudis, H. Lu, and S. Likothanassis. Scalable association-based text classification. In *Proc. of the 9th CIKM Conference*, pages 5–11, 2000.

[60] T. Morzy, M. Wojciechowski, and M. Zakrzewicz. Materialized data mining views. In *PKDD*, pages 65–74, 2000.

[61] T. Morzy and M. Zakrzewicz. Group bitmap index: A structure for association rules retrieval. In *Proc. of the 4th ACM SIGKDD Conference*, pages 284–288, 1998.

[62] B. Nag, P. M. Deshpande, and D. J. DeWitt. Using a knowledge cache for interactive discovery of association rules. In *Proc. of the 5th ACM SIGKDD Conference*, pages 244–253, 1999.

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

[64] S. Orlando, C. Lucchese, P. Palmerini, R. Perego, and F. Silvestri. kdci: a multi-strategy algorithm for mining frequent sets. In *FIMI'03 workshop*, 2003.

[65] S. Orlando, P. Palmerini, R. Perego, and F. Silvestri. Adaptive and resource-aware mining of frequent sets. In *Proc. of the 2002 IEEE ICDM Conference*, pages 338–345, 2002.

[66] B. Özden, S. Ramaswamy, and A. Silberschatz. Cyclic association rules. In *Proc. of the 14th ICDE Conference*, pages 412–421, 1998.

[67] F. Pan, G. Cong, A. K. H. Tung, J. Yang, and M. J. Zaki. Carpenter: finding closed patterns in long biological datasets. In *Proc. of the 9th ACM SIGKDD Conference*, pages 637–642, 2003.

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

[69] N. Pasquier, Y. Bastide, R. Taouil, and L. Lakhal. Discovering frequent closed itemsets for association rules. In *Proc. of the 7th ICDT Conference*, pages 398–416, 1999.

[70] J. Pei, G. Dong, W. Zou, and J. Han. On computing condensed frequent pattern bases. In *Proc. of the 2002 ICDM Conference*, pages 378–385, 2002.

[71] J. Pei and J. Han. Can we push more constraints into frequent pattern mining? In *Proc. of the 6th ACM SIGKDD Conference*, pages 350–354, 2000.

[72] J. Pei, J. Han, and L. V. S. Lakshmanan. Mining frequent item sets with convertible constraints. In *Proc. of the 17th ICDE Conference*, 2001.

[73] 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 the 2001 IEEE ICDM Conference*, pages 441–448, 2001.

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

[75] J. Pei, A. K. H. Tung, and J. Han. Fault-tolerant frequent pattern mining: Problems and challenges. In *Workshop on Research Issues in Data Mining and Knowledge Discovery*, pages 161–168, 2001.

[76] S. Ramaswamy, S. Mahajan, and A. Silberschatz. On the discovery of interesting patterns in association rules. In *Proc. of the 24th VLDB Conference*, pages 368–379, 1998.

[77] R. Raymon. Search through systematic set enumeration. In *Proc. of the Internation Conference on Principles of Knowledge Representation and Reasoning*, 1992.

[78] M. H. Saraee and B. Theodoulidis. Knowledge discovery in temporal databases. In *IEE Colloquium on Knowledge Discovery in Databases*, pages 1–4, 1995.

[79] A. Savasere, E. Omiecinski, and S. B. Navathe. An efficient algorithm for mining association rules in large databases. In *Proc. of the 21th VLDB Conference*, pages 432–444, 1995.

[80] A. Savasere, E. Omiecinski, and S. B. Navathe. Mining for strong negative associations in a large database of customer transactions. In *Proc. of the 14th ICDE Conference*, pages 494–502, 1998.

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

[82] Y. Sismanis, A. Deligiannakis, N. Roussopoulos, and Y. Kotidis. Dwarf: shrinking the petacube. In *Proc. of the 2002 SIGMOD Conference*, pages 464–475, 2002.

[83] R. Srikant and R. Agrawal. Mining generalized association rules. In *Proc. of the 21th VLDB Conference*, pages 407–419, 1995.

[84] R. Srikant and R. Agrawal. Mining quantitative association rules in large relational tables. In *Proc. of the 1996 ACM SIGMOD Conference*, pages 1–12, 1996.

[85] R. Srikant and R. Agrawal. Mining sequential patterns: Generalizations and performance improvements. In *Proc. of the 5th EDBT Conference*, pages 3–17, 1996.

[86] R. Srikant, Q. Vu, and R. Agrawal. Mining association rules with item constraints. In *Proc. of the 3rd ACM SIGKDD Conference*, pages 67–73, 1997.

[87] J. Srivastava, R. Cooley, M. Deshpande, and P.-N. Tan. Web usage mining: Discovery and applications of usage patterns from web data. *SIGKDD Explorations*, 1(2):12–23, 2000.

[88] P.-N. Tan, V. Kumar, and J. Srivastava. Selecting the right interestingness measure for association patterns. In *Proc. of the 8th ACM SIGKDD Conference*, pages 32–41, 2002.

[89] S. Thomas, S. Bodagala, K. Alsabti, and S. Ranka. An efficient algorithm for the incremental updatation of assciation rules in large databases. In *Proc. of the 3rd ACM SIGKDD Conference*, pages 263–266, 1997.

[90] H. Toivonen. Sampling large databases for association rules. In *Proc. of the 22th VLDB Conference*, pages 134–145, 1996.

[91] A. K. H. Tung, H. Lu, J. Han, and L. Feng. Breaking the barrier of transactions: Mining inter-transaction association rules. In *Proc. of the 5th ACM SIGKDD Conference*, pages 297–301, 1999.

[92] A. Tuzhilin and B. Liu. Querying multiple sets of discovered rules. In *Proc. of the 8th ACM SIGKDD Conference*, pages 52–60, 2002.

[93] J. Wang, J. Pei, and J. Han. Closet+: Searching for the best strategies for mining frequent closed itemsets. In *Proc. of the 9th ACM SIGKDD Conference*, pages 236–245, 2003.

[94] G. I. Webb. Efficient search for association rules. In *Proc. of the 6th ACM SIGKDD Conference*, pages 99–107, 2000.

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

[96] X. Yan and J. Han. Closegraph: mining closed frequent graph patterns. In *Proc. of the 9th ACM SIGKDD Conference*, pages 286–295, 2003.

[97] C. Yang, U. Fayyad, and P. S. Bradley. Efficient discovery of error-tolerant frequent itemsets in high dimensions. In *Proc. of the 7th ACM SIGKDD Conference*, pages 194–203, 2001.

[98] Q. Yang, H. H. Zhang, and I. T. Y. Li. Mining web logs for prediction models in www caching and prefetching. In *Proc. of the 7th SIGKDD Conference*, pages 473–478, 2001.

[99] M. J. Zaki. Generating non-redundant association rules. In *Proc. of the 6th ACM SIGKDD Conference*, pages 34–43, 2000.

[100] M. J. Zaki. Efficiently mining frequent trees in a forest. In *Proc. of the 8th ACM SIGKDD Conference*, pages 71–80, 2002.

[101] M. J. Zaki and K. Gouda. Fast vertical mining using diffsets. In *Proc. of the 9th ACM SIGKDD Conference*, pages 326–335, 2003.

[102] M. J. Zaki and C.-J. Hsiao. Charm: An efficient algorithm for closed itemset mining. In *Proc. of SIAM International Conference on Data Mining*, pages 398–416, 2002.

[103] M. J. Zaki, S. Parthasarathy, M. Ogihara, and W. Li. New algorithms for fast discovery of association rules. In *Proc. of the 3rd ACM SIGKDD Conference*, pages 283–286, 1997.

[104] Z. Zheng, R. Kohavi, and L. Mason. Real world performance of association rule algorithms. In *Proc. of the 7th SIGKDD Conference*, pages 401–406, 2001.