EFFICIENT MAXIMAL FREQUENT ITEMSET MINING BY PATTERN-AWARE DYNAMIC SCHEDULING

by

Xinghuo Zeng

B.Sc., Nanjing University, 2005

A THESIS SUBMITTED IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE
in the School
of
Computing Science

SIMON FRASER UNIVERSITY
Summer 2007

APPROVAL

Name: Xinghuo Zeng

Degree: Master of Science

Title of thesis: Efficient Maximal Frequent Itemset Mining by Pattern-Aware Dynamic Scheduling

Examining Committee: Dr. Wo-Shun Luk
Chair

_____

Dr. Jian Pei, Senior Supervisor

_____

Dr. Ke Wang, Supervisor

_____

Dr. Martin Ester, SFU Examiner

Date Approved: July 20th, 2007

Abstract

While frequent pattern mining is fundamental for many data mining tasks, mining maximal frequent itemsets efficiently is important in both theory and applications of frequent itemset mining. The fundamental challenge is how to search a large space of item combinations. Most of the existing methods search an enumeration tree of item combinations in a depth-first manner.

In this thesis, we develop a new technique for more efficient maximal frequent itemset mining. Different from the classical depth-first search, our method uses a novel probing and reordering search method. It uses the patterns found so far to schedule its future search so that many search subspaces can be pruned. Three optimization techniques, namely reduced counting, pattern expansion and head growth, are developed to improve the performance. As indicated by a systematic empirical study, our new approach outperforms the currently fastest maximal frequent itemset mining algorithm FPMax* clearly.

To my beloved family.

"He alone deserves freedom as well as life, who has to win them by conquest everyday."

— Johann Wolfgang von Goethe(1749-1832)

Acknowledgments

I would like to express my deep gratitude to my senior supervisor and mentor Dr. Jian Pei, for his patience and guidance. As a beginner in research, I benefit a lot from discussions with him. I thank him for sharing his experience and skills in research with me, for his warm encouragement and care which help me to overcome the difficulties in my research and life.

I would like to thank my supervisor Dr. Ke Wang, for his insightful comments which help to improve the quality of my thesis. My thanks also go to Dr. Martin Ester, for his valuable comments on my thesis and the generous help he offered me when I worked with him as a TA. I am very thankful to Dr. Wo-Shun Luk for chairing my defence.

I am also grateful to my friends all over the world. I thank Ming Hua, Mag Lau, Bin Zhou, Ji Xu, Yabo Xu, Dan Wang, Feng Wang, Wendy Wang and Xu Cheng for their kind help during my study at SFU. I thank Huizhen Tang in Beijing, Tianyu Cao and Shuxun Cao in Nanjing, Edith Ngai and Raymond Wong in Hong Kong, Ted in Tokyo, Wei Wu in Singapore and Yan Zhang in Netherlands, for their care and understanding.

My deepest gratitude goes to my parents, my sister and my aunt. I thank them for their endless support and love all through my life. Never can I accomplish this without their love.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

The problem of efficient mining of frequent itemsets is fundamental for many data mining tasks, such as mining association rules [6], correlations [9], causality [27], sequential patterns [7], episodes [21], partial periodicity [17], iceberg-cube computation [8], associative classification [20], and subspace clustering [4]. Since it was firstly proposed in [5], this problem has been studied extensively. There exists prolific literature focusing on this problem, in which various algorithms have been proposed.

1.1 The Frequent Itemset and the Maximal Frequent Itemset Mining Problem

Let $I = \{i_1, i_2, ..., i_n\}$ be a set of items, an itemset S is a subset of I. Let $l = |S|$, then l is called the *length* of S, and S is called an l-itemset. An itemset S' *subsumes* an itemset S'' if and only if $S' \supseteq S''$. For the sake of simplicity, we often write an itemset as a string of items. For example, itemset $\{a, c, d\}$ is often written as acd.

A transaction is a tuple (tid, Y) where tid is a unique transaction-id and Y is an itemset. Transaction (tid, Y) *contains* itemset S if and only if $S \subseteq Y$. For a given transaction database D which consists of multiple transactions, the *support* of an itemset S, denoted by $support(S)$, is the number of transactions containing S. That is,

$$support(S) = |\{(tid, Y) \in D | S \subseteq Y\}|$$

For a given *minimum support threshold* min_sup, an itemset S is a *frequent itemset* or as known as a *frequent pattern* if and only if $support(S) \geq min\_sup$. Given a transaction

tid	itemset
10	*bcde*
20	*abcd*
30	*abcdf*
40	*abcde*
50	*def*

Table 1.1: A transaction database.

database and a minimum support threshold, the problem of *frequent itemset mining* [5] is to find the complete set of frequent itemsets.

For example, consider the transaction database D in Table 1.1. Let the support threshold $min\_sup = 2$. Since $support(bc) = 4$, it is a frequent itemset.

Frequent itemsets have the following well-known monotonic *Apriori property*, or *downward closure property* [5].

Theorem 1 (Apriori Property) *If S is frequent in a transaction database D, then every nonempty subset of S is frequent.*

Proof. Let $Trans(X) = \{(tid, Y) \in D | Y \supseteq X\}$, and S' be an nonempty subset of S. For any transaction (tid, Y), if $Y \supseteq S$, $Y \supseteq S'$. Then $Trans(S') \supseteq Trans(S)$, $|Trans(S')| \geq |Trans(S)|$. That is, $support(S') \geq support(S)$. The theorem is proved. □

According to the Apriori property, a long frequent itemset of length l leads to $(2^l - 2)$ shorter frequent itemsets. For example, in Table 1.1, if $min\_sup = 2$, *abcd* is a frequent itemset. All subsets of *abcd* including *a*, *b*, *c*, *d*, *ab*, ..., *bcd* are frequent itemsets as well.

An itemset S is called a *maximal frequent itemset* or an *MFI* for short [25] if S is frequent and every proper superset of S is infrequent. In Table 1.1, when $min\_sup = 2$, the MFIs are *abcd*, *bcde* and *df*. The problem of *mining maximal frequent itemsets* or *mining MFIs* for short is to find the complete set of MFIs.

1.2 Motivation and Contribution

Mining maximal frequent itemsets efficiently is important in both theory and applications of frequent itemset mining. On the theoretical side, the MFIs serve as the border between

the frequent itemsets and the infrequent ones. With the set of MFIs, for any itemset S, whether it is frequent or not can be determined quickly using the Apriori property: if there exist some MFI M such that $S \subseteq M$, then S is frequent, otherwise S is infrequent. In addition, MFIs serve as a summary of all frequent itemsets. By the Apriori property, every non-empty subset of an MFI is FI, thus the number of MFIs is much less than the number of FIs. As an example, on the chess dataset with $min\_sup = 10\%$, the number of FI is $1,394,140,008$, but the number of MFIs is only $2,612,646$.

On the application side, MFIs are used in a few interesting and challenging data mining tasks. For example, using MFIs, we can find emerging patterns [12] which are itemsets frequent in the positive samples and infrequent in the negative samples. If an itemset S is a subset of some MFIs in the positive sample and a proper superset of some MFIs in the negative sample, then S is an emerging pattern. Emerging patterns can be used to construct effective classifiers [18]. As another example, using MFIs with respect to a series of support thresholds, we can summarize and approximate the support information of all frequent itemsets [22].

The fundamental challenge of mining MFIs is how to search a large space of itemsets and identify MFIs. Most of the existing methods search an enumeration tree of the set of itemsets in a depth-first manner, and prune subtrees using the MFIs found before.

While the previous studies focus on catching pruning opportunities sharply, can we systematically create pruning opportunities in the mining? In this thesis, we develop a *Pattern-Aware Dynamic Scheduling* approach to tackle the problem. Different from the classical depth-first search methods, our approach adopts a novel *probing and reordering* method to search for MFIs. It uses the MFIs found so far to schedule its future search so that many search subspaces can be pruned. To further enhance the efficiency, three optimization techniques, namely *reduced counting*, *pattern expansion* and *head growth*, are proposed. As indicated by a systematic empirical study using the benchmark data sets, our new approach outperforms the currently fastest maximal frequent itemset mining algorithm FPMax* [15] in a clear margin.

1.3 Thesis Organization

The organization of this thesis is as follows. In Chapter 2 we give a systematic review on related work. In Chapter 3 we propose our Pattern-Aware Dynamic Scheduling approach.

The three optimization techniques will be introduced in Chapter 4. An extensive empirical comparison between the PADS algorithm and existing state-of-the-art algorithms will be reported in Chapter 5. In Chapter 6 we will discuss how our approach can be applied to other data mining problems, and conclude the study.

Chapter 2

Related Work

In this chapter we will give a systematically review related work on mining maximal frequent itemsets.

The problem of maximal frequent itemset (MFI for short) mining has been extensively studied since it was first proposed in [25]. To tackle this problem, many algorithms have been proposed. Here, we review nine representative algorithms, namely MaxEclat [29], MaxClique [29], Princer-Search [19], MaxMiner [25], DepthProject [1], Mafia [10], GenMax [13], FPMax* [15] and LCMmax [28].

In this chapter we compare those algorithms in four aspects:

1. Whether they are main memory algorithms or out-of-core algorithms[1];
2. Methods used to organize the search procedures;
3. Data structures used to store data in main memory, if they are main memory algorithms or if they are out-of-core algorithms, but require loading a proportion of the database into main memory to speed up computation; and
4. Pruning techniques used to prune unpromising search space.

Table 2.1 gives a summarization of the comparison, where PADS is the new algorithm we will introduce in Chapter 3.

[1]Out-of-core refers to algorithms which process data that is too large to fit into the main memory of a computer at one time. Such algorithms must be optimized to efficiently fetch and access data stored in slow bulk memory such as hard drive or tape drives.

Category	Algorithm	Main Memory	Data Structure	Search Method	Pruning Techniques			
					Lookahead	DynOrdr	ProgFoc	PEP
Lattice Based	MaxClique		vertical TID list	hybrid				
	MaxEclat		vertical TID list	hybrid				
	Princer-Search			hybrid				
Tree Based	MaxMiner			bread-first	*	*	*	
	DepthProject	*	horizontal bivector	depth-first	*	*	*	
	Mafia	*	vertical bivector	depth-first	*	*		*
	GenMax	*	diffsets	depth-first	*	*	*	*
	FPMax*	*	FP-tree	depth-first	*	*	*	
	LCM v2	*	simple array	depth-first and reorder	*	*		
	PADS	*	FP-tree	probe and reorder	*		*	*

Table 2.1: A summarization of previous algorithms on mining MFI itemsets.

2.1 Main Memory Algorithms versus Out-of-core Algorithms

It is assumed that the whole database cannot be held into main memory when various frequent itemsets mining algorithms are first proposed. Since multiple scans of the database are needed, the mining process becomes IO-bounded, and one of the focuses of algorithm design is to reduce the number of database scans. The early algorithms such as MaxEclat, MaxClique, Princer-search, MaxMiner make this assumption.

However, there are some dense datasets with small size but a large number of MFIs, which makes the problem CPU-bounded rather than IO-bounded. A good example is the Chess data set generated from the UCI Chess dataset by Roberto Bayardo, the size of which is only $334KB$. When the minimum threshold is set to 3%, the number of MFIs is $7,682,809$! FPMax*, the winner on mining MFIs at Workshop on Frequent Itemset Mining Implementations in 2003[2] takes more than $2,000$ seconds to mine all the MFIs on a computer with a 3.0 GHz Pentium CPU and 1.0 GB main memory.

Moreover, main memory availability has increased by orders of magnitude in the past decade, and nowadays many small to medium size databases can be held into main memory. In addition, when the length of frequent patterns is long, the number of MFIs increases exponentially, which makes the problem computationally difficult.

Thus, it is reasonable to assume that some databases can be held into main memory and focus on CPU efficiency in algorithm design. Algorithms proposed later such as DepthProject, Mafia, GenMax and FPMax all make this assumption.

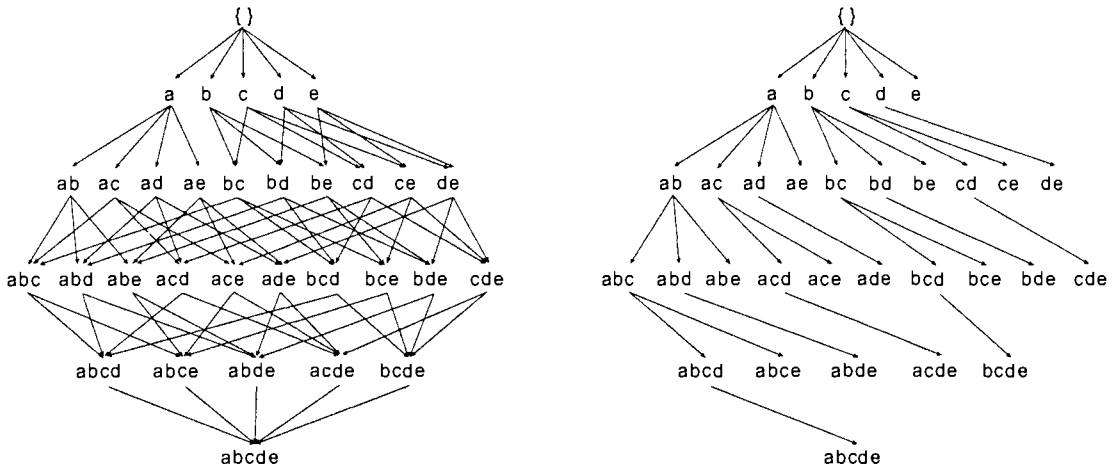
2.2 Search Methods

Given a set of items, all its subsets can be organized by a subset lattice. Figure 2.1(a)[3] gives an example of a lattice. By the downward closure property of the frequent itemsets, that is, all subsets of a frequent itemset are frequent, and all supersets of an infrequent itemset are infrequent, mining maximal frequent itemsets is essentially *finding the border between frequent and infrequent itemsets in the lattice.*

For each k-itemset S in the lattice, we can retain only the edge between S and the $(k-1)$

[2]Held on 19 November 2003, Melbourne, Florida, USA in conjunction with ICDM'03

[3]We draw the lattice with the empty set at the top and the largest set at the bottom, the conventional *bottom-up search*, however, begins from the empty set. It thus actually follows the *top-down* direction in this figure. The similar convention is applied to *the top-down search.*



(a) an item set lattice (b) a set enumeration tree

Figure 2.1: An example of a lattice and a set enumeration tree.

prefix[4] of S and remove all the other edges connecting S and its $(k-1)$-subsets. The subset lattice is then reduced to a set enumeration tree [26], as illustrated by Figure 2.1(b).

Previous maximal frequent itemset mining algorithms organize the search space with the help of either a subset lattice or a set enumeration tree. We thus can divide those algorithms into two categories: lattice-based algorithms and tree-based algorithms.

2.2.1 Searching in a Subset Lattice

With the help of a lattice, we can search the MFIs with different approaches: the bottom-up search, the top-down search, and the hybrid search which is a combination of both top-down and bottom-up searches.

The *bottom-up search* is proposed in the Apriori algorithm [6]. Figure 2.2 gives the complete Apriori algorithm. It first scans the database once to find the set of frequent single items L_1. At the $k(k \geq 2)$-th level, we

1. joins any pair of frequent length $(k-1)$ itemsets sharing a $(k-2)$- prefix to get the

[4]We assume there exists a lexicographical order among items. For simplicity, an itemset is represented as a string sorted in this order.

Input: a transaction database D and support threshold min_sup;
Output: L, frequent itemsets in D;
Method:

```
1:  L_1 =find_frequent_1-itemsets(D);
2:  for(k = 2; L_{k-1} ≠ ∅; k ++){
3:    C_k =apriori_gen(L_{k-1}, min_sup);
4:    for each transaction t ∈ D{
5:      C_t=subset(C_k,t);
6:      for each candidate c ∈ C_t
7:        c.count++;
      }
8:    L_k = {c ∈ C_k|c.count ≥ min_sup}
    }
9:  return L = ⋃_k L_k;
```

procedure apriori_gen(L_{k-1}: frequent $(k-1)$-itemsets; min_sup: minimum support threshold)

```
1:  for each itemset l_1 ∈ L_{k-1}
2:    for each itemset l_2 ∈ L_{k-1}
3:      if((l_1[1] = l_2[1]) ∧ (l_1[2] = l_2[2]) ∧ ... ∧ (l_1[k-2] = l_2[k-2])
        ∧(l_1[k-1] < l_2[k-1])) then{
4:        c = l_1[1]l_1[2]...l_1[k-2]l_1[k-1]l_2[k-1];
5:        if c has an infrequent (k-1)-subset then
6:          delete c;
7:        else add c to C_k;
          }
8:  return C_k;
```

Figure 2.2: The Apriori algorithm

candidate length k frequent itemsets C_k. For example, suppose $a_1...a_{k-2}a_{k-1}$ and $a_1...a_{k-2}a_k$ are two frequent k-itemsets, we can join them to get $a_1...a_{k-2}a_{k-1}a_k$ as one candidate length k frequent itemsets;

2. delete from C_k any itemsets if one of its length $(k-1)$ subset is infrequent; and

3. count the supports of itemsets in C_k by scanning the database once to get the set of frequent length k itemsets L_k.

This process continues until L_k is empty. All frequent itemsets are found and the set of MFIs can be obtained by deleting those which are subsets of some frequent itemsets. A disadvantage of this method is that it generates and counts the supports of *all frequent itemsets*, which is more than necessary for identifying only the *maximal frequent itemsets.*

In contrary to the bottom-up search, *the top-down search* starts from the maximal itemset of the lattice, that is, the itemset that contains all items. If it is frequent, we are done and output it as an MFI. Otherwise, suppose its length is k, we generate each of its $(k-1)$-subsets and check their supports. This process repeats until the itemset is frequent. Let C be the set of frequent itemsets obtained in this process, then C is a superset of the set of all MFIs. We need to eliminate from C those itemsets which are subsets of other frequent itemsets in C to obtain the set of MFIs. The top-down search requires examining all infrequent itemsets.

Due to the aforementioned reasons, for the maximal frequent itemset mining problem, both the top-down and the bottom-up searches have their disadvantages, hence lattice-based algorithms MaxEclat, MaxClique and Princer-Search combine the two methods in some way to achieve higher efficiency.

The *Princer-Search* algorithm combines the bottom-up search and the top-down search: both the bottom-up and the top-down searches are conducted simultaneously, and they help each other to prune the search space. The pruning is based on the following observations: (1) when an itemset is known frequent, then all of its subsets must be frequent and they do not need to be examined anymore, and (2) when an itemset is known infrequent, then all of its supersets must be infrequent, and they do not need to be examined anymore. While the top-down search and the bottom-up search only use the first observation and the second observation, respectively, to prune search space, Princer-Search combines those two observations to prune search space. Long MFIs can be found in early scans of the database by the top-down search, thus the bottom-up search does not need to examine their subsets

anymore. Similarly, short infrequent itemsets can be found in early scans by the bottom-up search, top-down search then does not need to examine their supersets anymore. Thus, compared with the bottom-up or the top-down search, Princer-Search needs fewer database scans.

Both MaxEclat and MaxClique use the *hybrid search* to search for MFIs in the lattice. The search consists of two phases: the hybrid phase which tries to get a long frequent itemset $S1$, and the bottom-up phase which examines the rest nodes of the lattice which are non-subsets of $S1$.

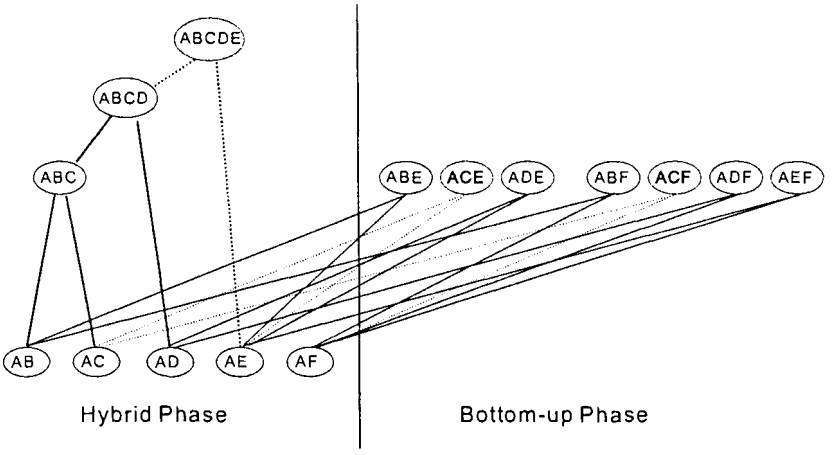


Figure 2.3: Hybrid search in the sublattice with prefix A.

It should be mentioned that either MaxEclat or MaxClique does not search in the entire lattice. Instead, they first partition the entire lattice into small sub-lattices, and conduct hybrid search to find MFIs in each sub-lattice. The union of all MFIs in these sub-lattices is a superset of the MFIs in the entire lattice since some itemsets may be *locally maximal* but not *globally maximal.* For example, suppose *abcde* is a maximal frequent itemset in the sublattice having prefix *a*, and *bcde* is a maximal frequent itemset in the sublattice having prefix *b*, then *bcde*is a locally maximal frequent itemset but not globally maximal frequent itemset. Thus, a post-processing phase is needed to eliminate those itemsets which are not

globally maximal. We will give an example to show the hybrid search later.

The difference between MaxEclat and MaxClique lies in the way they partition the lattice: MaxEclat partitions the lattice into multiple *prefix-based equivalent classes* while MaxClique partitions it into *maximal-clique-based pseudoequivalent classes.* For further details, please refer to [29].

Figure 2.3 gives an example of the hybrid search. After partitioning, the entire lattice is decomposed into several sub-lattices. In the sublattice with prefix a, suppose ab, ac, ad, ae and af are known frequent and they form the starting point of the hybrid search. The algorithm first combines ab with ac to generate abc and checks its support. If it is frequent, the algorithm further extends abc with ad to get $abcd$ and then checks its frequency. The process repeats until the extension turns out to be infrequent. In this example, suppose extension to $abcde$ fails, then $abcd$ is a candidate for maximal frequent itemset. This ends the hybrid phase. In the folowing bottom-up phase, the algorithm examines the frequencies of abe, ace, ade, abf, acf, adf and aef. The frequent ones, abe, ade, abf, adf and aef, are retained and an Apriori-like bottom-up search is conducted beginning from those itemsets to find the rest MFIs in this sublattice.

2.2.2 Searching in a Subset Enumeration Tree

When searching in a subset enumeration tree, we can conduct either the *breadth-first search*, as done in MaxMiner, or the *depth-first search*, as done in DepthProject, Mafia, GenMax, FPMax and LCMmax.

To make the explanation easier, we first introduce some definitions. Let P be an itemset, when we are visiting the P node in the set enumeration tree, P is also called the *head* of the current search. We assume that we can sort the items according to an order R, and $i \prec_R j$ denote item i precedes item j in order R. Then, we can define the $Tail$ and $Untrimmed\ Tail$ of the current search as follows.

Definition 1 *The tail of an empty itemset $\emptyset$, $Tail(\emptyset)$, is the set of frequent items. That is, $Tail(\emptyset) = \{t | support(\{t\}) \geq min\_sup\}$.*

Definition 2 *Suppose P is the parent of Q in the set enumeration tree, R is an order on $Tail(P)$, and $Q = P \cup \{i\}$, the untrimmed tail of an itemset Q, $F(Q)$, is the set of items ordered after item i in $Tail(P)$, that is, $F(Q) = \{t | t \in Tail(P)$ and $\forall i \in Q : i \prec_R t\}$.*

Definition 3 *The tail of an non-empty itemset Q, $Tail(Q)$, is the set of items in $F(Q)$ that are frequent extensions of Q. That is, $Tail(Q) = \{t \in F(Q) | Q \cup \{t\}$ is frequent$\}$.*

Let us look at an example. Suppose $L_1 = \{a, b, c, d, e, f, g\}$ is the set of frequent items, and $P = \emptyset$, then $Tail\{P\} = L_1$. Let $Q = \{b\}$ be a child of P, and the alphabetical order is used to sort the items in $Tail(P)$, then the set of items sorted after b in $Tail(P)$, that is, $\{c, d, e, f, g\}$, is $F(Q)$. For each item $i \in F(Q)$, if $Q \cup \{i\}$ is frequent, then i is inserted into $Tail(Q)$. Thus we have the folllowing lemma.

Lemma 1 *If P is the parent of Q in the set enumeration tree, then $Tail(P) \supset F(Q) \supseteq Tail(Q)$.*

In the *breadth-first search*, for each frequent k-itemset Q, for each item $i \in F(Q)$, $Q \cup \{i\}$ is generated as a child of Q and inserted into C_{k+1}. The supports of itemsets in C_{k+1} are counted by the next scan of the database. It should be noticed that the *children generation* here is equivalent to the *candidate generation* in Apriori. Thus, the breadth-first search in a set enumeration tree is in fact equivalent to the bottom-up search in its corresponding lattice.

The tree structure, however, is more useful than the lattice as it possesses some nice properties that can be employed to prune the search space. The *lookahead pruning* and the *dynamic reordering* techniques are based on the properties of the set enumeration trees. They will be introduced in Section 2.4.

The *depth-first search* in the set enumeration tree without any pruning proceeds in the following way: at each node Q, $F(Q)$ is generated from the *Tail* of its parent. For each item $i \in F(Q)$, count the support of $Q \cup \{i\}$. If it is frequent, insert i into $Tail(Q)$. After $Tail(Q)$ is obtained, for each item $i \in Tail(Q)$, node $Q \cup \{i\}$ is generated and processed recursively. Figure 2.4 gives the depth-first search algorithm in the set enumeration tree. The function *DFS* is first called with $N = \emptyset$, $T = L_1$, that is, the set of frequent 1-itemsets, and $L = \emptyset$. All frequent itemsets will be returned in L when the search terminates.

The depth-first search can be implemented by constructing *projected databases.* An *P-projected database* consists of all and only the transactions subsuming P, and in each transaction, only the items in $Tail(P)$ are retained, and items not in $Tail(P)$ are removed. To search maximal frequent itemsets in the subtree of a in the set enumeration tree, we only need to check the *a-projected database.* Similarly, to search the maximal frequent itemsets in the subtree of ab, we only need to check the *ab-projected database*, which can be constructed

Algorithm: depth-first search in a set enumeration tree.
Input: Database D; minimum support threshold min_sup.
Output: frequent itemsets L in database D.
Methods:

```
1:  L = ∅;
2:  L_1 =find_frequent_1-itemsets(D);
3:  L = L ∪ L_1;
4:  call DFS(∅, L_1, L);
5:  return L;

procedure DFS(N: an itemset, T: tail of N, F: the set of frequent itemsets)
1:  if (T = ∅) then
2:    L = L ∪ N;
    else
3:    for each item i ∈ T
4:      N' = N ∪ {i};
5:      F' = {t ∈ T | i ≺ t};
6:      T' = {t ∈ F' | support(N' ∪ t) ≥ min_sup};
7:      call DFS(N', T', L);
```

Figure 2.4: A depth-first search algorithm in a set enumeration tree

ALGORITHM: LCMmax (P:itemset, H:items to be added)
1: H' :=the set of items e in H s.t. $P \cup \{e\}$ is frequent
2: **If** $H' = \emptyset$ **then**
3: **If** $P \cup \{e\}$ is infrequent for any e **then**
4: **output** P; **return**
5: **End if**
6: **End if**
7: Choose an item $e* \in H'$; $H' := H' - \{e*\}$
8: LCMmax($P \cup \{e\}$, H')
9: P' := frequent itemset of the maximum size found in the recursive call in 7
10: **For each** item $e \in H - P'$ **do**
11: $H' := H' - \{e\}$
12: LCMmax($P \cup \{e\}$, H')
13: **End for**

Figure 2.5: The LCMmax algorithm.

from the *a-projected database*. Since ab is a child of a in the set enumeration tree, the depth-first search takes a divide-and-conquer strategy.

The examination of *all frequent itemsets* in either the breadth-first search or depth-first search mentioned above is not necessary for the maximal frequent itemset mining problem. In Section 2.4 we will introduce various pruning techniques which can be employed to prune unpromising branches in the set enumeration tree.

LCMmax uses an interesting search method which is different from both depth-first search and breadth-first search. When searching the I-subtree, it first picks an item e in the tail, and searches the branch $I \cup \{e\}$ recursively. Let P' be a frequent itemset of maximum length obtained by the recursive search in the $I \cup \{e\}$-subtree, in the following steps, it reorders the items in $Tail(I)$ so that any item not in P' has an index less than any item in P', then a recursive call is generated for each $e \in Tail(I) - P'$. In Section 3.6, we will give a thorough analysis of LCMmax and compare it with our PADS algorithm.

2.3 Data Structures

Main memory algorithms need some data structures to store the database in main memory. Out-of-core algorithms such as MaxEclat and MaxClique also hold part of the database into main memory to accelerate search. In the algorithms we review in this chapter, five data structures, namely horizontal bitvector, vertical transaction ID list, vertical bitvector, FP-tree and simple array, have been used. Simple array is used by LCMmax. It is a two-dimensional array which consists of item lists for each transaction. It is a straightforward representation of the database. Thus, in this subsection, we will introduce the rest four data structures one by one.

2.3.1 Horizontal Bitvector

Horizontal bitvector(or bitstring) is firstly used in the algorithm TreeProject [2] for mining all frequent itemsets. Based on this data structure, DepthProject was proposed for mining maximal frequent itemsets.

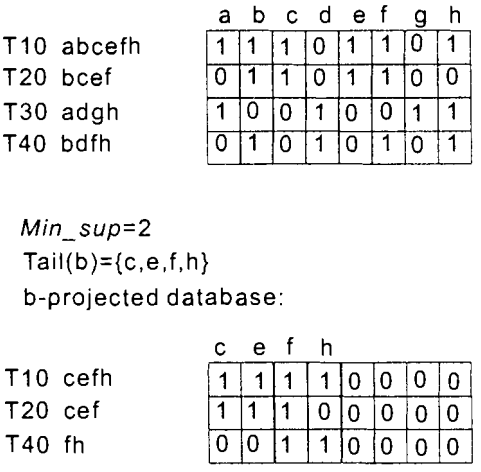


Figure 2.6: An example of horizontal bitvectors.

In this data structure, every transaction is represented by one bitvector, every item corresponds to one bit in each bitvector. The bit is set to 1 if the item appears in the

trasaction, and 0 otherwise. Figure 2.6 gives an example of horizontal bitvectors for a sample database and a projected database.

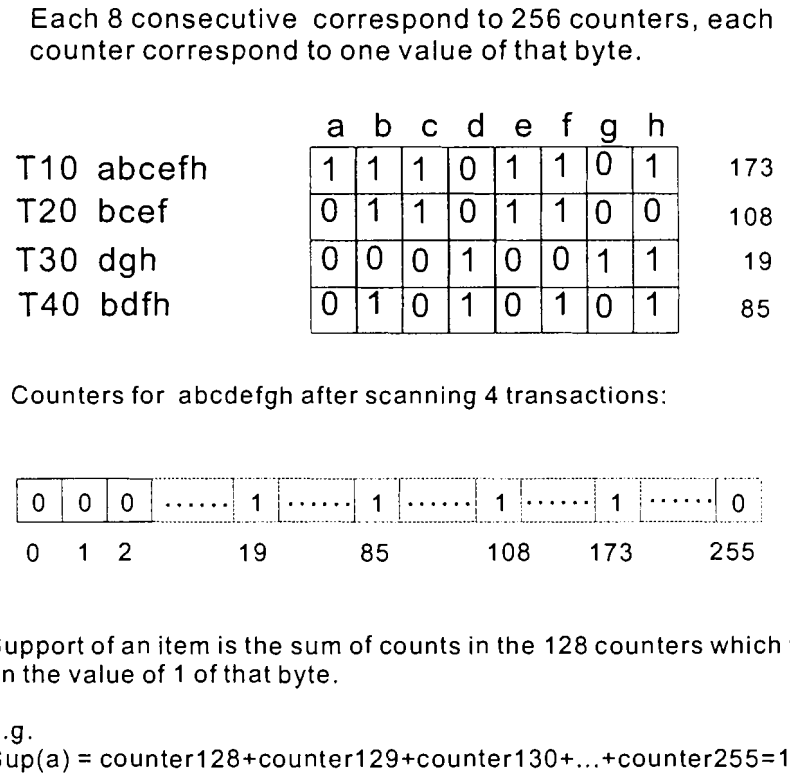


Figure 2.7: Counting technique using horizontal bitvectors.

Specialized counting technique is developed for support counting in horizontal bitvector, as illustrated by Figure 2.7. Assume that each transaction T contains n bits, and can therefore be expressed in the form of $\lceil n/8 \rceil$ bytes. Each byte of the transaction contains the information about the presence or absence of eight items, and the integer value of a byte can take on any value from 0 to $2^8 - 1 = 255$. For each 8 items represented by a byte, 256 counters are maintained. When a transaction is scanned, for each byte in the transaction, add 1 to the counter which represents the value of that transaction byte. This

process repeats for each transaction in the database. Therefore, at the end of scanning, we have $256 * \lceil n/8 \rceil$ counts. For an item i, there are $256/2 = 128$ counters corresponding to its presence. The support of an item i is the sum of those 128 counts in those counters.

The benefit of this counting technique is that it performs only 1 operation for each *byte* in the transaction, which contains 8 items. Thus, this method is a factor of 8 faster than the naive counting technique.

2.3.2 Vertical Transaction ID List

Vertical transaction ID list(tid-list for short) is firstly used for mining maximal frequent itemsets in [33] in the algorithms MaxEclat and MaxClique. An improved version of tid-list called *diffsets* is developed in [13] in the algorithm GenMax.

It should be noted that though MaxEclat, MaxClique and GenMax all use vertical data representation, the search method of MaxEclat and MaxClique is different from that of GenMax. While MaxEclat and MaxClique use a *lattice-based* search method, GenMax uses a *tree-based* search method.

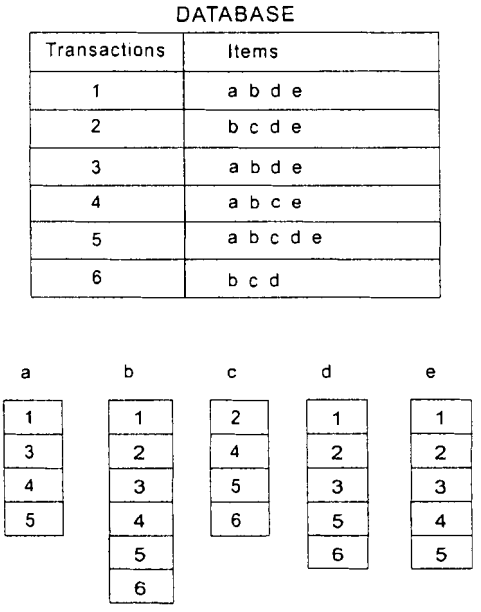


Figure 2.8: An example of vertical transaction ID list

In this data structure, each item has one *transaction ID list.* The list of item i contains the IDs of all transactions having i. Figure 2.8 gives an example. There are two advantages of tid-lists: first, the support of a k-itemset can be computed by simply intersecting the tid-lists of any of its two $(k-1)$-subsets; second, to compute the support of an itemset, using tid-lists can avoid scanning the whole database, only tid-lists relevant to the itemset are needed.

An improved version of tid-lists is the *diffsets*, which is firstly proposed in [30]. Diffsets are used in *tree-based* search method. The diffset of an itemset Q stores the difference between the transaction id list of its parent P in the set enumeration tree and the transaction id list of Q. Let $t(X)$ denotes the tid-list of an itemset X, diffset of a node Q stores $t(P) - t(Q)$.

The diffset for an itemset Q, $d(Q)$, can be computed as follows. If Q is a 1-itemset, $d(Q) = t(D) - t(Q)$, where $t(D)$ be the list of all transaction IDs. If the size of Q is larger than 1, suppose the parent of Q in the search tree is $P = Q - \{j\}$, and the parent of P is $P' = P - \{i\}$, then $Q = P' \cup \{i\} \cup \{j\}$, and

$$\begin{aligned} d(Q) &= d(P' \cup \{i\} \cup \{j\}) \\ &= t(P' \cup \{i\}) - t(P' \cup \{j\}) \\ &= t(P' \cup \{i\}) - t(P' \cup \{j\}) + t(P') - t(P') \\ &= t(P') - t(P' \cup \{j\}) - (t(P') - t(P' \cup \{i\})) \\ &= d(P' \cup \{j\}) - d(P' \cup \{i\}) \end{aligned}$$

The support of $P \cup \{i\}$ can be computed by $support(P \cup \{i\}) = support(P) - |d(P \cup \{i\})|$.

Figure 2.9 gives an example of diffsets. In this example, the diffsets on level 2 can be computed from either the tid-lists or the diffsets on level 1. Diffsets on level $k (k \geq 2)$ are computed using the diffsets on level $(k-1)$.

2.3.3 Vertical Bitvector

Vertical bitvector is firstly introduced in [10] in the algorithm Mafia. Vertical bitvector is similar to tid-list, except that it uses bitmap, instead of a list of transaction IDs, to store the transactions in which an item(set) occurs. There is one bit for each transaction in the database. If item i appears in transaction j, then bit j of the bitvector for item i is set to one; otherwise the bit is set to zero. This definition is the same for itemsets. The support

DATABASE

Transactions	Items
1	a b d e
2	b c d e
3	a b d e
4	a b c e
5	a b c d e
6	b c d

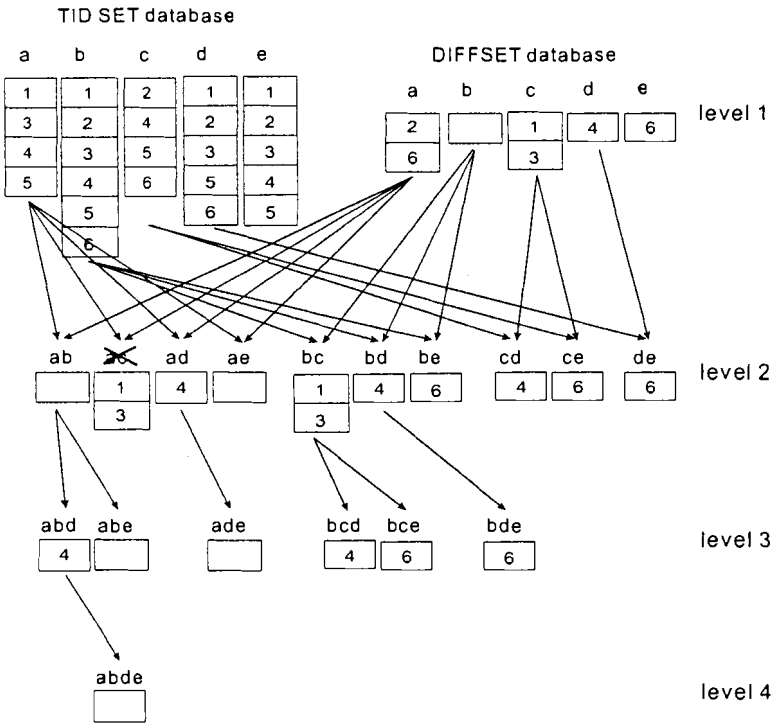


Figure 2.9: An example of diffset

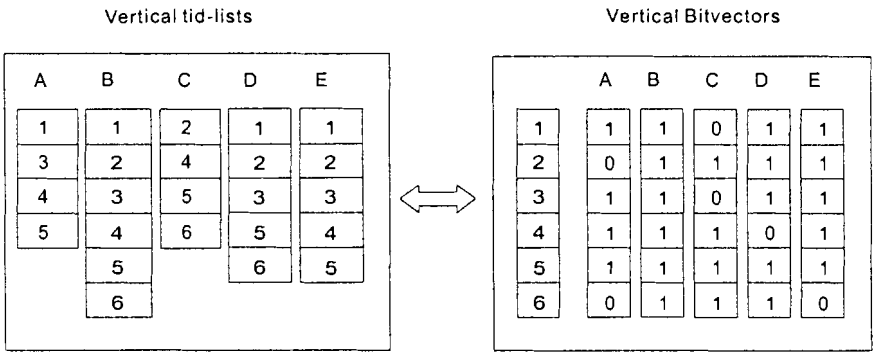


Figure 2.10: Differences between tid-lists and vertical bitvectors

of an item(set) can be obtained by simply counting the number of ones in its bitvector. Figure 2.10 shows the difference between tid-lists and vertical bitvectors.

Bitmap for item set $(X \cup Y)$ can be computed by applying bitwise-AND operation to $bitmap(X)$ and $bitmap(Y)$.

Compared with tid-lists, the benefit of vertical bitvector is its efficiency in space and in the computation. Representing a transaction ID in tid-list require 32 bits, thus when the support of an item(set) is more than $1/32(\approx 3\%)$, which is around 3%, vertical bitvector is less costly than tid-list in space. The time complexity of intersecting two tid-lists t_1 and t_2 is $O(|t_1| + |t_2|)$, while the time complexity of intersecting two vertical bitvectors is $O(|D|/32)$, where $|D|$ is the number of transactions of database D. The saving is obvious when the support is high.

When the support of an itemset is low, the bitvector becomes sparse since most bits in its bitvector are zeros. Bitwise-AND operation over the 0 regions is a waste of computation. Compressed bitvector is proposed to avoid this problem. In the tree based searching method, when we are searching in the P-subtree, all transactions that do not contain P is useless in counting the support of $P \cup \{i\}$. Thus we can remove those transactions from the bitvectors of P and items in $Tail(P)$. This leads to projected bitvectors for P. Figure 2.11 gives an example of a projected bitvectors.

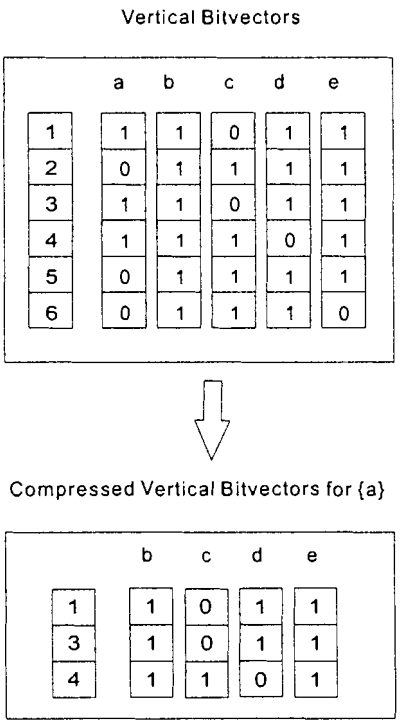


Figure 2.11: An example of compressed bitvectors

2.3.4 FP-tree

FP-tree is an extended prefix-tree structure for compact representation of relevant frequency information in the database. It is firstly proposed in [16] in the context of mining all frequent itemsets.

Each node in the FP-tree has three fields: *item-name*, *count*, and *node-link*. The field *item-name* stores the item this node represents, the field *count* stores the support of the itemset represented by the path from the root to this node, and the field *node-link* stores a pointer to the next node having the same item-name.

Each FP-tree is associated with a header table. Each row in the header table stores the name of a frequent item and a node-link which points to the first node in the FP-tree having the item-name. Items in the header table are sorted in support-descending order.

An FP-tree is constructed by two scans of the database. In the first scan, all frequent

items are found and inserted in support-descending order into the header table. In the second scan, each transaction is scanned from the database, infrequent items in this transaction are removed. The remaining frequent items are sorted according to the order in the header table and inserted into the FP-tree. If this itemset shares prefix with an itemset already in the FP-tree, the count of every node in that prefix is increased by one. For every item not included in the shared prefix, a new node is created and its count is initialized with one.

All nodes in the FP-tree sharing the same item-name are linked via the node-link field, and node-link field in the header table points to the first node in the FP-tree having the item-name.

Similar to other data structures, *projected FP-tree* can be constructed progressively. If P is the parent of Q in the set enumeration tree, the *Q-projected FP-tree* can be constructed by scanning the *P-projected FP-tree* twice. The construction process is similar to the construction of the initial FP-tree.

TID	Items Bought	(Ordered) Frequent Items
100	*a, b, c, d, e, f, g*	*c, g, a, b, d, e, f*
200	*a, c, e, f, h, k*	*c, a, e, f, h*
300	*b, c, d, f, g, m*	*c, g, b, d, f*
400	*a, b, c, d, g, h, n*	*c, g, a, b, d, h*
500	*e, g, i*	*g, e*

Table 2.2: A transaction database ($min\_sup = 2$).

Given $min\_sup = 2$, Figure 2.12(a) gives an example of an FP-tree for the sample database in Table 3.1. Figure 2.12(b) shows an example of projected FP-tree.

In [15], an improved version of FP-tree is proposed to reduce the traverse time. In the improved version, each FP-tree is associated with an two-dimensional array. Each cell in the array stores the count of a 2-itemset in the FP-tree. Figure 2.12(b) and (d) are the array for the FP-tree in Figure 2.12(a) and (c), respectively. The array is updated when constructing the FP-tree. Each cell is initialized as 0. When a transaction with count c is inserted into the FP-tree, for each combination of any two items in that transaction, the corresponding cell is increased by c.

Without using the array technique, for each item i in the header table of an projected FP-tree T_X, we construct a new projected FP-tree $T_{X\cup\{i\}}$ by scanning T_X twice. The first scan generates $Tail(X \cup \{i\})$, sort the items in $Tail(X \cup \{i\})$ and construct the header table

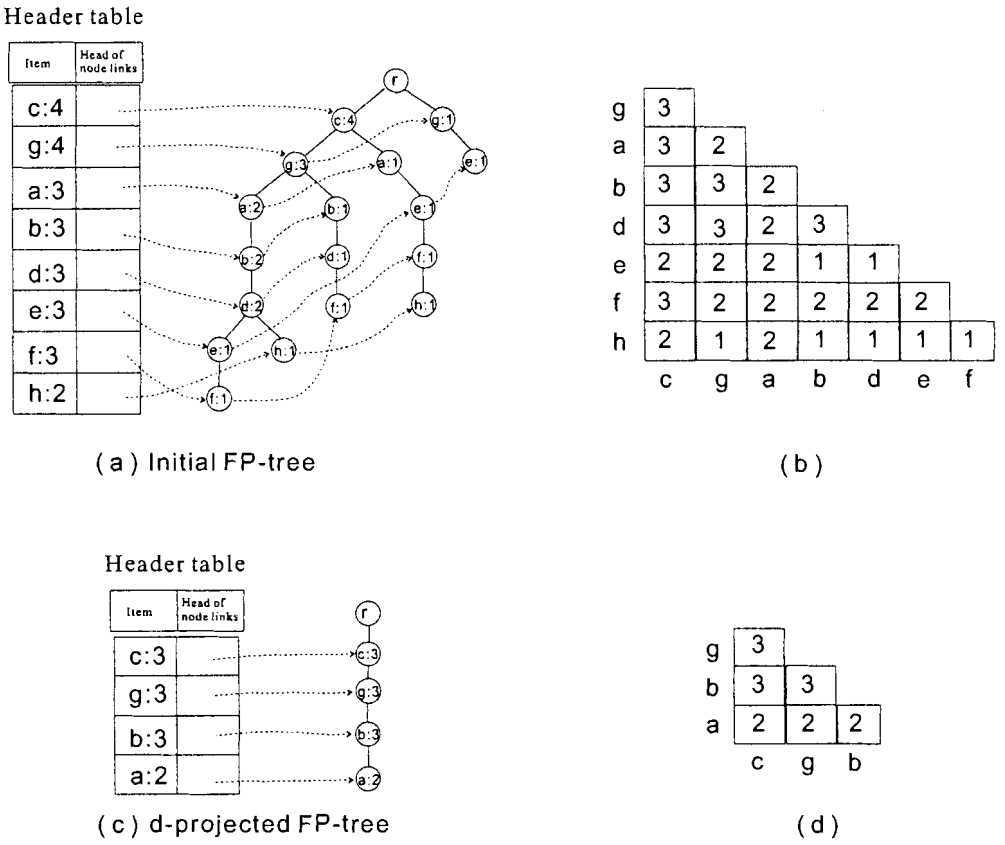


Figure 2.12: FP-tree for the sample database in Table 3.1

of $T_{X\cup\{i\}}$. The second scan construct the FP-tree $T_{X\cup\{i\}}$. With the array technique, the first scan can be omitted by reading the information in the array associated with FP-tree T_X. As shown in [15], this technique can reduce the running time significantly.

2.3.5 Summarization

Each of the four data structures has its advantages and disadvantages. The horizontal bitvector is efficient in counting and memory. However, it has to scan the database once to count the support of one itemset. Using the vertical id-list, only a small proportion of the database is needed read to compute the support of an itemset, however, if the database can

not be held in main memory, to find all frequent length 2 itemset, its has to do intersections of between the tid-lists of any two length 1 itemsets. It thus has to do the intersection for $|L_1|*(|L_1|-1)/2$ times, where L_1 is the set of frequent 1-itemsets. Those intersections require scanning the database for $|L_1 - 1|/2$ times, thus in implementation we need to temporarily transform vertical tid-list into horizontal data format to compute the tid-list for all length 2 frequent itemsets. The vertical bitvector is more efficient than vertical tid list when the support is high, however, when the support is low, this is not the case. This is because intersections between two bitvectors over the 0 region cannot be avoided. The FP-tree is a compact representation of the database in which different transactions can share the common prefix. When the overlaps between transactions in the database is small, FP-tree may not help save the space and the size of FP-tree may exceed the size of the database. Despite this advantage, it should however be noticed that the current fastest algorithm for mining maximal frequent itemsets uses the FP-tree as the main data structure.

2.4 Pruning Techniques

For searching in a set enumeration tree, various pruning techniques have been proposed to avoid searching unpromising branches. In this subsection, we give an introduction to four important pruning techniques that have been used in the previous algorithms.

2.4.1 Lookahead

Lookahead is an important pruning technique in maximal frequent itemset mining. It is firstly proposed in MaxMiner and used by other tree-based maximal frequent itemset mining algorithms. The idea of lookahead is that if the leftmost itemset of a subtree rooted at T is frequent, that is, if $T \cup Tail(T)$[5] is frequent, then we can avoid exploring the rest part of the subtree since all of them are the subsets of the leftmost itemset.

In the breadth-first and the depth-first search algorithms, lookahead is implemented in different ways. Figure 2.13 illustrates the implementation of lookahead in MaxMiner. Let $\{a, b, c, d, e\}$ be the set of frequent items, in the next scan of the database, in addition to counting the support of candidate frequent length 2 itemsets *ab*, *ac*, *ad*, *ae*, *bc*,*bd*, *be*, *cd*, *ce* and *de*, frequencies of *abcde*, *bcde*, and *cde* are also counted as they are in the form of

[5]Some algorithms (e.g., MaxMiner) do the superset checking before trimming $F(T)$ to $Tail(T)$, so they use $T \cup F(T)$ instead of $T \cup Tail(T)$.

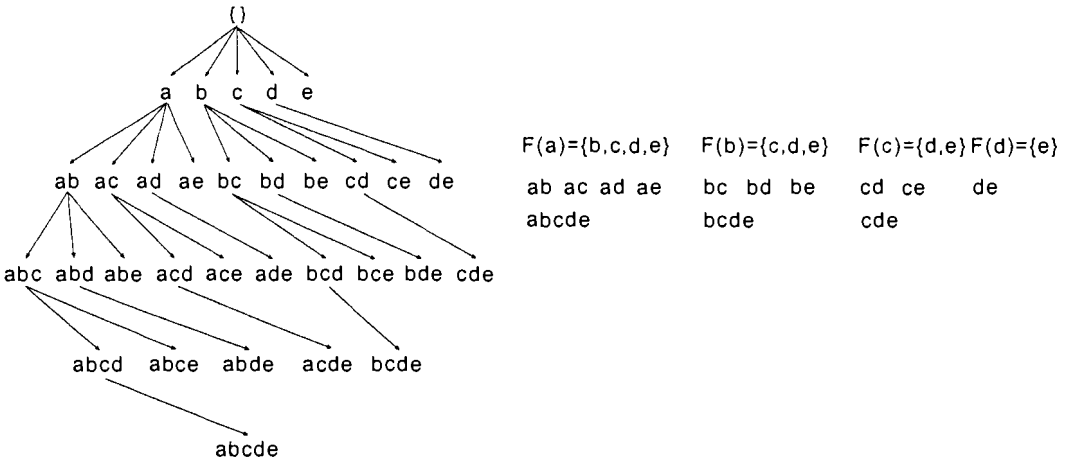


Figure 2.13: Lookahead in breadth-first search using a set enumeration tree

$Q \cup F(Q)$ for some frequent length 1 itemset Q. If $Q \cup F(Q)$ is frequent, then the descendants of Q do not need to be examined in the following scans.

In the depth-first search, we have two approaches to testing whether $Q \cup Tail(Q)$ is frequent or not. One way (named HUTMFI in Mafia) is to check whether $Q \cup Tail(Q)$ is a subset of some MFI already found. Another way (named FHUT in Mafia) is to explore the leftmost path of the subtree rooted at Q, as $Q \cup Tail(Q)$ is the leftmost node in the subtree rooted at Q. If it is frequent, we can skip exploring the rest part of the subtree. Those two ways are usually used together in most algorithms, and HUTMFI is usually applied before FHUT.

2.4.2 Progressive Focusing

There may exist millions of MFIs in a database. The superset check in the lookahead technique can be expensive. To accelerate superset check, the *progressive focusing* technique is proposed in GenMax. The idea is that when exploring the subtree rooted at P, we maintain a list of *local MFIs* MFI_P, which are MFIs subsuming P. For each direct child $Q = P \cup \{i\}$ of P, before unfolding the subtree rooted at Q, we check whether $i \cup Tail(Q)$ is a subset of some MFI in MFI_P, instead of comparing $Q \cup Tail(Q)$ against the whole set of

MFIs. Typically $|MFI_P|$ is far less than $|MFI|$, so this technique can significantly reduce the time spent in superset check.

Similar to the projected database construction, construction of *local MFIs* is progressive, that is, MFI_Q is constructed from MFI_P, where P is the parent of Q in the search tree.

2.4.3 Dynamic Reordering

Dynamic reordering is firstly proposed in MaxMiner and adopted by all the maximal frequent itemset mining algorithms proposed later. The idea of dynamic reordering is that sorting the items in the $Tail(P)$ in ascending order of their support in *P-projected database* may lead to more effective pruning of the search space.

Suppose we are searching the subtree of P, and $Q = P \cup \{i\}$ is a child of P. If we sort items in $Tail(P)$ in support-ascending order, then $Tail(Q)$ will consist of items having higher support than item i in the *P-projected database*. Thus $Q \cup Tail(Q)$ is more likely to be frequent than in the situation where items in $Tail(P)$ are randomly ordered. The *lookahead* pruning technique therefore is more likely to be applied to prune the search space.

2.4.4 Parent Equivalence Pruning (PEP)

PEP is firstly identified in closed frequent itemset mining algorithms CLOSET [24] and CHARM [32], and later used by Mafia and GenMax in maximal frequent itemset mining.

When we search the subtree of P with $Tail(P)$, if there is some item $i \in Tail(P)$ such that $support(P) = support(P \cup \{i\})$, then all transactions having P also have i. It is impossible that some MFIs have P but do not have i. Therefore, we can move i from $Tail(P)$ to P, that is, let $Tail(P) = Tail(P) - \{i\}$ and $P = P \cup \{i\}$. In this way we search the P-subtree with a smaller tail and the search space is reduced.

Chapter 3

Pattern-Aware Dynamic Scheduling

In this chapter we propose a new maximal frequent itemset mining algorithm: *pattern-aware dynamic scheduling* (*PADS* for short).

Our algorithm is tree-based. We assume that the data is main memory resident. Our algorithm can use data structures like vertical tid list, vertical bitvector and FP-tree. In current implementation, we use the FP-tree as the main data structure for the following reasons. Firstly, FP-tree is a compact presentation of the database and has been shown a very efficient data structure. Secondly, to compare the performance of our search method to the currently best algorithm FPMax*, it is better to use the same data structure as FPMax* uses.

Besides the integration of pruning techniques *lookahead* and *progressive focusing*, we use a novel search method called *probing and reordering* which differs significantly from either the previous breadth-first search or the depth-first search methods. Our algorithm also takes advantage of the current MFIs to organize the future search space. Three optimization techniques are proposed to improve the efficiency. As shown by the empirical study, our algorithm outperforms the currently best algorithm FPMax* with a clear margin.

3.1 Ideas

As analyzed in Chapter 2, most previous tree-based algorithms basically follow *the depth-first search or the breadth-first search and pruning* framework. We notice that, though various pruning techniques have been applied, *generating the subsets of MFIs* still constitutes the major cost. In particular, the cost comes from two aspects.

Firstly, *given a set MFIs already found, the current approaches cannot fully utilize them to prune the search space.* For example, in Figure 3.1(a), there are 5 MFIs in the database: *abce*, *abde*, *acd*, *bcd* and *cde*. Suppose items in $Tail(\emptyset)$ are ordered lexicographically. All approaches except LCM v2 search the tree in the following order (the nodes in brackets are pruned by the lookahead pruning technique):

$\emptyset - a - ab - abc - abce - abde - (abe) - ac - acd - (ace - ad - ade - ae) - b - bc - bcd - (bce - bd - bde - be) - c - cd - cde - (ce - d - de - e)$

After finding *abce*, to search the rest MFIs, most current approaches have to search the children of *abd*, *ac*, *b* and *c*, during which nodes *ac*, *b*, *bc* and *c* are visited and their projected databases are generated. Those nodes, however, are subsets of *abce*. Can we avoid visiting them? The answer is Yes. If after the discovery of the MFI *abce*, we reorder the items in $Tail(\emptyset)$ such that any item not in *abce* precedes any item in *abce*, as shown in Figure 3.1(b), then the subtrees of *a*, *b*, *c* and *e* can be pruned immediately, since all nodes in those subtrees are subsets of *abce*. All the other MFIs are now in the *d*-subtree, we thus can focus on only the *d*-subtree and search it recursively.

Here, the MFI *abce* used to reorder the items is called the *key pattern.*

Please note that LCM v2 uses the similar reordering idea, however, the key pattern used to reorder the children of the root node is obtained by fully identifying all MFIs in the subtree of its leftmost child and picking the longest one. In this example, after finding MFIs *abce*, *abde*, *acd* in the *a*-subtree, one of the longest MFI, say *abce*, is used to reorder the rest items in $Tail(\emptyset)$, thus *d* is ordered before *b*, *c* and *e*, then the *ab*, *ac* and *ae* subtrees are pruned and only the *ad*-subtree is searched recursively. The inefficiency of this approach is analyzed in section 4.5.

Secondly, *the discovery of MFIs is costly.* Even with the reordering technique, to find the MFI *abce*, the depth-first search method may still need to construct the projected databases for itemsets *a*, *ab* and *abc*. The projected database construction is costly. Suppose Q is an itemset, to construct the Q-projected database, we firstly need to count the support of

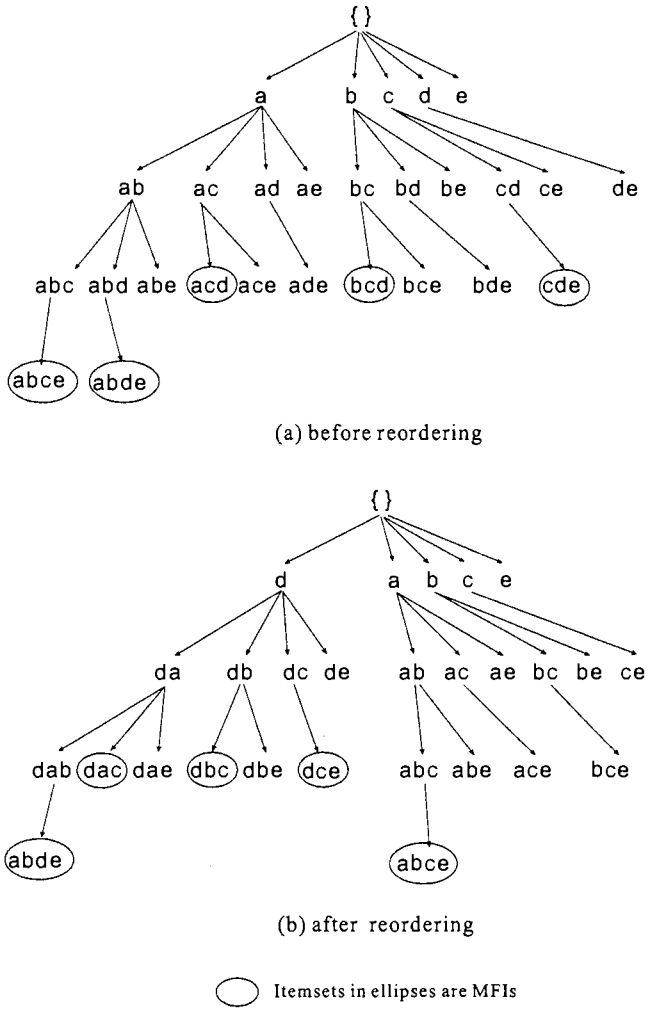


Figure 3.1: Different ordering of the tail may affect the computation efficiency-an example

Input: a transaction database TDB, an itemset S and support threshold min_sup;
Output: the complete set of MFIs;
Function $PADS(TDB, S, min\_sup)$

```
1:  compute Tail(S);
2:  // head-and-tail pruning
    if there exists an MFI Y found before such that Tail(S) ∪ S ⊂ Y then
      return;
3:  if Tail(S) = ∅ then
4:    output S as an MFI;
5:  probe for a frequent itemset M1 with prefix S;
6:  choose a key pattern Y among all MFIs already found;
7:  make a order R on items in Tail(S) according to Y;
8:  construct an FP-tree T_S for S;
9:  for each item i ∈ (Tail(S) − Y) in the order of R
10:   call PADS(T_S, S ∪ {i}, min_sup);
11: return;
```

Figure 3.2: The Framework of the PADS algorithm.

$Q \cup \{i\}$ for each item i in Q's untrimmed tail $F(Q)$. After that $Tail(Q)$ is found and the Q-projected database is generated. How can we avoid this cost as much as possible? We observe that the discovery of an MFI can be implemented efficiently by using a *probing process*, which only require scanning the projected database of Q's parent once. We will introduce this technique in Section 3.2.

The framework of our approach is shown in Figure 3.2. When searching the subtree rooted at an itemset S, after we know that $S \cup Tail(S)$ is not a subset of any MFI found before and $Tail(S) \neq \emptyset$, we firstly apply the probing process to try to find one maximal frequent itemset M_1 in this subtree. Next, among all the MFIs already found, including the M_1 found in the probing process, we pick one MFI Y as the *key pattern*. Then, we reorder items in $Tail(S)$ in such a way that any item not in the key pattern precedes any item in the key pattern, and construct the S-projected FP-tree T_S. All the children of S in the search tree can then be divided into two groups: the *promising children*, whose descendants may have new MFIs, and the *unpromising children*, whose descendants do not have new MFIs. For each $i \in Tail(S) - Y$, $S \cup \{i\}$ is a promising children of S. For each $i \in Tail(S) \cap Y$, $S \cup \{i\}$ is an unpromising children of S. Only the promising children are searched in a

Input: an itemset S, S's untrimmed tail $F(S)$, an order R on $F(S)$,
and support threshold *min_sup*;
Output: a frequent itemset;
Method:

1: $S' = S$;
2: Let i_1, i_2,...,i_n be the itemlist of $F(S)$ after sorted according to R;
3: **for** $(j = 1; j \leq n; j{+}{+})$
4: **if** $support(S' \cup \{i_j\}) \geq min\_sup$
5: $S' = S' \cup \{i_j\}$;
6: **return** S';

Figure 3.3: The Framework of the Probing Process.

recursive way.

3.2 The Probing Process

Let S be an itemset, suppose in the set enumeration tree we are searching the S-subtree and the untrimmed tail $F(S)$ is $\{i_1, i_2, ..., i_k\}$, how can we find one MFI S' in the S-subtree without physically constructing any projected databases? We propose a probing process to tackle this problem. The probing process works as follows. For each item i in $F(S)$, we check whether $S \cup \{i\}$ is frequent. If yes, then let $S = S \cup \{i\}$. Otherwise we keep S unchanged. We repeat this process until all items in $F(S)$ are tried. Figure 3.3 gives the framework of the probing process.

To ensure that all frequent itemsets output are maximal, we have the following Lemma and Theorem.

Lemma 2 (Local Maximality of the Probing Result) *Given a frequent itemset S, its untrimmed tail $F(S)$, and an order R on $F(S)$, let Q be the output of the probing process, then there is not item i in $F(S)$ but not in Q such that $Q \cup \{i\}$ is frequent. That is, Q is local maximal.*

Proof. Let i_1, i_2,..., i_n be the item list we get after sorting $F(S)$ according to R, suppose

i_k is in $F(S)$ but not in Q, and $Q \cup \{i_k\}$ is frequent. Let

$$M = \{i | i \in Q \cap F(S) \text{ and } i \text{ precedes } i_k \text{ in order } R\}$$

Then, before the k-th iteration in Figure 3.3, $S' = S \cup M$. Since $S \cup M \cup \{i_k\} \subseteq Q \cup \{i_k\}$, by the monotonicity property, $S \cup M \cup \{i_k\}$ is frequent. Then, i_k should be included into S' in line 5 in Figure 3.3. Contradiction. □

Theorem 2 (Global Maximality of the Probing Result) *Given a frequent itemset S, its untrimmed tail $F(S)$, and an order R on $F(S)$, let Q be the output of the probing process, if Q is not subsumed by some MFIs found before, then Q itself is an MFI.*

Proof. We firstly prove that all nodes searched after node S-subtree cannot be a superset of S.

Let T be an arbitrary node searched after S-subtree. Suppose $M = a_1 a_2 ... a_i$ is the lowest common ancestor of S and T in the set enumeration tree, $S = a_1 a_2 ... a_i b_1 ... b_m$ and $T = a_1 a_2 ... a_i c_1 ... c_n$. Let R' be an order on $Tail(M)$ and $i \prec_{R'} j$ denote item i precedes item j according to order R'. Since S-subtree is searched before T, $b_1 \prec_{R'} c_1$. Let $F = \{i | i \in Tail(M)$ and $c_1 \prec_{R'} i\}$, then $T \subseteq M \cup \{c_1\} \cup F$, and b_1 is not in $M \cup \{c_1\} \cup F$. Therefore S is not a subset of T .

Equivalently, we can say that in the set enumeration tree, all supersets of Q can either be a node in the S-subtree or a node searched before the S-subtree. By Lemma 2, there is no frequent proper superset of Q in the S-subtree. Thus, if Q is not a subset of some MFIs found before searching the S-subtree, Q itself is an MFI. □

TID	Items Bought	(Ordered) Frequent Items
100	*a, b, c, d, e, f, g, i*	*c, g, a, b, d, e, i, f*
200	*a, c, e, f, h, k*	*c, a, e, f, h*
300	*b, c, d, f, g, m*	*c, g, b, d, f*
400	*a, b, c, d, g, h, i, n*	*c, g, a, b, d, i, h*
500	*e, g, i*	*g, e, i*

Table 3.1: A transaction database.

With different data structures, the probing process can be conducted differently. Since in our algorithm the FP-tree data structure is used, how can we take advantage of the FP-tree to conduct the probing process? Let us look at an example.

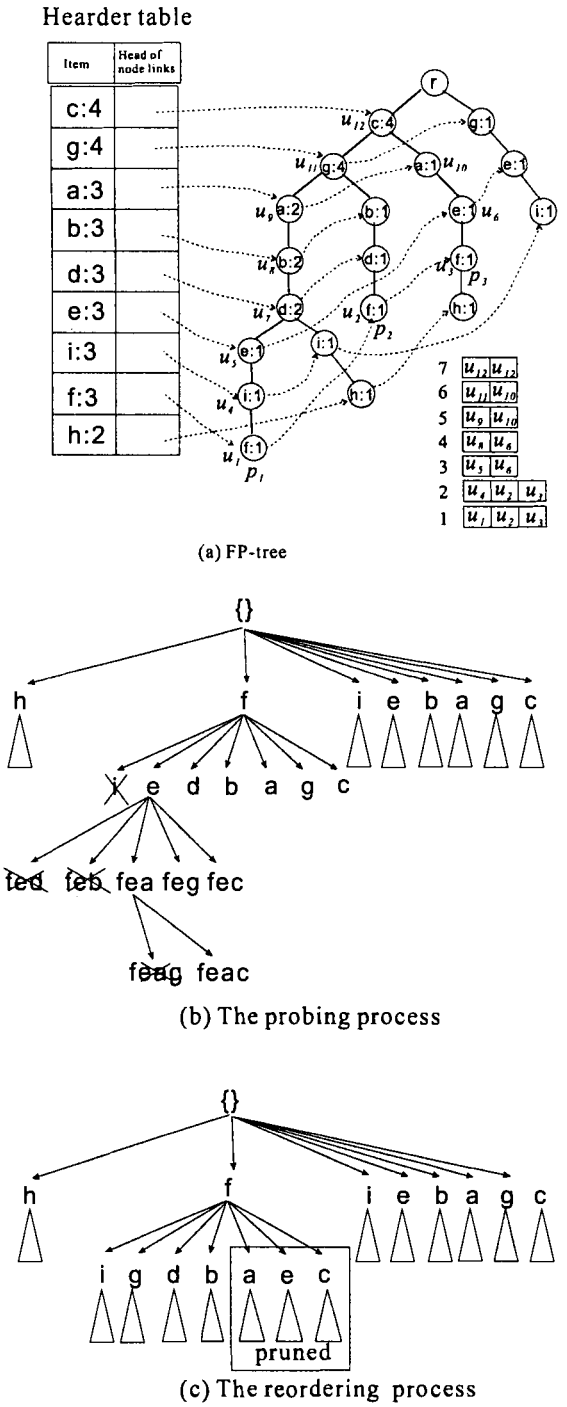


Figure 3.4: Probing and reordering using an FP-tree

Example 1 Given a transaction database in Table 3.1 and $min\_sup = 2$, an FP-tree in Figure 3.4(a) is constructed. Suppose we are probing for an MFI in the f-subtree in the set enumeration tree, as shown in Figure 3.4(b). $F(f) = \{c, g, a, b, d, e, i\}$. We impose an order R on $F(f)$ where R is the reverse of the item order in the header table. That is, we are probing with $F(f)$ in the order i, e, d, b, a, g, c.

In the FP-tree all paths from the root to the f nodes can be obtained by following the node link of f in the header table. The support of each path is recorded in the count field of the f node on it. In Figure 3.4(a), there are three paths having f: $p_1(r{-}c{-}g{-}a{-}b{-}d{-}e{-}i{-}f)$, $p_2(r{-}c{-}g{-}b{-}d{-}f)$ and $p_3(r{-}c{-}i{-}e{-}f)$. The supports of all the three paths are 1. To track these paths, for each path we keep a node on it. We call this node the **representative node** of the path. The representative nodes of all these paths form a list. At the beginning all nodes in the list are the f nodes in the FP-tree, as shown by list 1 in Figure 3.4(a). S' is initialized with $S = \{f\}$.

Let us now look at the parents of the f nodes. On path p_1, p_2 and p_3, they are i, d and e, respectively. Since the nodes on each path are sorted in the order in the header table, thus p_1 cannot have node i, and p_2 cannot have nodes i and e. In other words, a node k can only exist on paths whose representative node has a parent j such that $j \preceq_R k$.

For all $k \in F(f)$, let $L(k)$ be the sum of the supports of paths whose representative nodes have a parents k. That is,

$$L(k) = \sum_{p\text{'s node has a parent } k} support(p)$$

then we have $L(i)=1$, $L(e)=1$, $L(d)=1$, $L(b)=0$, $L(a)=0$, $L(g)=0$, $L(c)=0$.

Since a node k can only exist on paths whose representative node has a parent j such that $j \preceq_R k$, thus

$$support(S' \cup \{k\}) = L(k) \quad if \quad \forall j \prec_R k,\ L(j) = 0 \tag{3.1}$$

and

$$support(S' \cup \{k\}) \leq \sum_{j \preceq_R k} L(j) \tag{3.2}$$

In this example, $support(S' \cup \{i\}) = L(i) = 1$, $support(S' \cup \{e\}) \leq L(i) + L(e) = 2$, $support(S' \cup \{d\}) \leq L(i) + L(e) + L(d) = 3$.

Since $S' \cup \{i\}$ is infrequent, we remove i from $F(f)$ and check next item e. The upper bound of $support(S' \cup \{e\})$ is 2, thus $S' \cup \{e\}$ can be frequent. To calculate $support(S' \cup \{e\})$,

we need to update the node list first. Let $parent(k)$ denote the parent of k node in the FP-tree. For each node in the list, if it has a parent $k \prec_R e$, we replace this node by its ancestor j in the FP-tree such that $j \prec_R e$ and $parent(j) \succeq_R e$. After updating, intuitively, each node in the list is on the frontier between items already tried and items to be tried on the path it represents. Equation 3.1 and 3.2 can thus again be applied to calculate the support and support upper bounds. In this example, node f on p_1 is replaced by node i on p_1, as shown by list 2 in Figure 3.4(a).

Now for each item k in $F(f)$, we recalculate $L(k)$. $L(e)=2$, $L(d)=1$, $L(b)=0$, $L(a)=0$, $L(g)=0$, $L(c)=0$. Thus $support(S' \cup \{e\}) = L(e) = 2 \geq min\_sup$, e is added to S'. In this case, we remove all nodes from the list which do not have parent e, so that only paths having S' are retained. Next we replace the rest nodes by their parent e, as shown by list 3 in Figure 3.4(a). Each node in the list is again brought to the frontier between items already tried and items to be tried on the path it represents.

We repeat this process until all items in $F(f)$ are tried. In this example, $feac$ is the result of the probing process. Figure 3.4(b) shows the probing process in the search tree.

It is easy to see that using FP-tree the probing process can be done by simply traversing the branches having f. It is efficient since no projected databases are physically constructed. □

Our probing process in fact adopts the *pseudo database projection* technique, which is firstly proposed in [23]. In Figure 3.5 we give the pseudo code for the probing process in an FP-tree.

Since the probing process does not check whether S' is subsumed by some MFI already found, according to Theorem 2, before we output it as an MFI, we need to further check whether it is a subset of some MFI found before to guarantee its maximality.

3.3 Choosing the Key Pattern

When searching the S-subtree, after the probing process, we get one frequent itemset M that can be used as a key pattern to reorder $Tail(S)$. Let $\mathcal{M} = \{M | M \text{ is an MFI}, M \supset S \text{ and } M \cap Tail(S) \neq \emptyset\}$, any $M \in \mathcal{M}$ can be chosen as a key pattern for reordering the items in $Tail(S)$. However, using different key patterns may affect the size of the future search space. In this section, we discuss how to choose a good key pattern.

Input: a projected FP-tree T_P, an itemset $S = P \cup \{i_k\}$, S's untrimmed tail $F(S)$,
an order R which is the reverse of the order in the header table,
support threshold min_sup;
Output: an frequent itemset which is maximal in the search tree rooted at S;
Method:

```
1:   Initialize a node list L with all the i_k nodes in the T_P;
2:   for each n ∈ L
3:     let p be the path from the root to node N;
4:     support(p) = the count recorded in the count field in n;
5:   S' = S;
6:   while (F(S) ≠ ∅) {
7:     for each node n ∈ L
8:       Let p be the path it represents;
9:       L(parent(n)) = L(parent(n)) + support(p);
10:    Let j be the item with the least order in F(S) s.t. Σ_{k⪯_R j} L(k) ≥ min_sup;
11:    remove all items that precedes j in R from F(S);
12:    if ∀k ≺_R j, L(k) = 0 then {
13:      S = S ∪ {j};
14:      F(S) = F(S) − {j};
15:      remove all nodes from L that does not have parent j;
16:      replace all nodes in L by their parent;
       }
17:    else
18:      for each node n ∈ L with a parent p ≺_R j
19:        replace n by its ancestor k such that k ≺_R j and parent(k) ⪰_R j;
     }
20:  return S';
```

Figure 3.5: Pseudo Code for the Probing Process Using the FP-tree.

From the previous analysis, the number of potential children of S is equal to $|Tail(S) - M| = |Tail(S)| - |M \cap Tail(S)|$, thus the larger $|M \cap Tail(S)|$, the less the children we need to search in the future.

In the set enumeration tree, any $M \in \mathcal{M}$ exists in two parts: (1) the S-subtree, and (2) the subtrees searched before S. The probing process returns an M in the S-subtree. One question we may ask is, is the probing process necessary? We argue that M returned by the probing process is likely to cover more items in $Tail(S)$ than some MFI searched before the S-subtree. Next let us give some analysis.

Let S' be the result returned by the probing process, then S' exists in the S-subtree. By Lemma 2 we know that $S' \cap Tail(S)$ is a maximal frequent itemset in the S-projected database.

We may sort the items in the tail in different orders and probe for multiple times, and then pick the best result. However, there is no guarantee that this multiple-probing process can return longer MFIs that have sharper pruning power. Moreover, the probing process is computationally costly even though the pseudo projection technique is used. Thus in our algorithm when visiting a node we probe only once.

Now let us consider another key pattern candidate $M \in \mathcal{M}, M \neq S'$ from the subtrees searched before S. Let $N = M - M \cap Tail(S) = M - Tail(S)$, then $M \cap Tail(S)$ is a maximal frequent itemset in N-projected database. Otherwise there exists an item i not in M such that $M \cap Tail(S) \cup \{i\}$ is frequent in N-projected database, then $N \cup (M \cap Tail(S)) \cup \{i\} = M \cup \{i\}$ should be frequent in the input database D, M is not an MFI in D.

Since $M \supset S$, $N = M - Tail(S) \supseteq S - Tail(S) = S$. In addition, $N \neq S$, otherwise $M = S \cup (M \cap Tail(S))$ is in the S-subtree. Thus we have $N \supset S$. Then the N-projected database is a subset of the S-projected database. With the same minimum support min_sup, it is likely that the size of a maximal frequent itemset in S-projected is larger than the size of a maximal frequent itemset in N-projected database. That is, $|S' \cap Tail(S)|$ is likely larger than $|M \cap Tail(S)|$. This is confirmed by the experiment. In our experiment, we find that for most of the time S' covers more items in $Tail(S)$ than any other MFI already found. This is the reason why we actively probe for an frequent itemset whenever we search a subtree.

However, this is a heuristic. There is no guarantee that $|S' \cap Tail(S)|$ is always larger than $|M \cap Tail(S)|$. In some cases, M may have heavier overlap with $Tail(S)$.

To reduce the search space as much as possible, heuristically we can pick one $Y \in \mathcal{M}$

that has the largest overlap with $Tail(S)$ as the key pattern. That is, we pick the key pattern

$$Y = \arg\max_{\text{MFI } Z \supset S}\{|Z \cap Tail(S)|\}$$

Please note that $Tail(S)$ contains at least one item that is not in Y. Otherwise, since $S \cup Tail(S)$ is a subset of Y, S is pruned by the lookahead pruning.

We implement the key pattern selection as a byproduct of the lookahead pruning. For each itemset S, to apply the lookahead pruning, we have to check $X \cup Tail(S)$ against all the MFIs found so far. At the same time, we also collect the information of $|Y \cap Tail(S)|$. Thus, the cost of computing the candidate in this step is very little.

To speed up the lookahead check, we also adopt the *progressive focusing* technique introduced in Section 2.4.2. We use prefix trees to organize MFIs found so far. Each S-projected FP-tree T_S is associated with an MFI tree MFI_T_S, which stores all MFIs having S as their subsets. MFI_T_S is constructed when T_S is constructed. Similar to the FP-trees, the construction of MFI trees takes a divide-and-conquer approach: let P be the parent of Q in the search tree, then MFI_T_Q is constructed by scanning MFI_T_P once.

3.4 The Reordering Process

According to the key pattern Y chosen in the previous step, we can reorder items in $Tail(S)$ so that any item in the key pattern precedes any item not in the key pattern. After reordering, the children of S fall into two categories: the *promising children* and the *unpromising children*. For each $i \in Tail(S) - Y$, $S \cup \{i\}$ is a promising children of S. For each $i \in Tail(S) \cap Y$, $S \cup \{i\}$ is an unpromising children of S. There may exist new MFIs among the descendants of the promising children, but there cannot be any new MFIs among the descendants of the unpromising children. Unpromising children can then be pruned immediately. Only promising children are search in a recursive way.

The above process is called *pattern-aware dynamic scheduling* (PADS for short). Here we prove the correctness of the above scheduling.

Theorem 3 (Correctness of Tail Reorder) *Let S be a frequent itemset, and Y be an MFI such that $S \subset Y$. If a dynamic search order with respect to Y is used to construct the set enumeration subtree of S, then for any item $z \in Tail(S) \cap Y$ and any pattern Z in the subtree of $S \cup \{z\}$, $Z \subset Y$.*

Proof. As discussed before, $Tail(S \cup \{z\}) \subset Tail(S)$. Since a dynamic search order with respect to Y is used, z is behind all items in $Tail(S) - Y$ in the order. That is, $Tail(S \cup \{z\}) \subset Y$. Moreover, since $z \in Tail(S) \cap Y$ and $S \subset Y$, we have $Z \subseteq (S \cup \{z\} \cup Tail(S \cup \{z\})) \subset Y$. The theorem is proved. □

Example 2 Figure 3.4(c) gives an example of the reordering process. Before reordering, the order of items in $Tail(f)$ is e, d, b, a, g, c. Suppose $feac$ is selected to reorder $Tail(f)$. After reordering, the order of items in $Tail(f)$ becomes g, d, b, a, c, e. The children fa, fc and fe are pruned immediately. □

Please note that our Pattern-Aware Dynamic Scheduling is different from the technique of dynamic ordering frequent items developed in the previous studies. Dynamic ordering frequent items is a heuristic method. Due to the correlations among frequent items, there exist counter examples where sorting frequent items in support ascending order does not help pruning. In contrast, the effect of dynamic search scheduling is determined once the key pattern is chosen. To search the subtree of a pattern S, once there exists at least one key pattern $Y \supset S$ found before, a dynamic search order based on Y can be used to prune some children of S by dynamic search scheduling. It is not heuristic.

3.5 Summary

Our Pattern-Aware Dynamic Scheduling approach consists of three phases. When searching the S-subtree, in the first phase, we applies the probing process to search for a frequent itemset M_1. Possibly M_1 covers a good proportion of items in $Tail(S)$ since there is no frequent itemsets in the S-subtree which is a proper superset of M_1. In the second phase, among all the MFIs already found, we pick one that is a superset of S and covers the most items in $Tail(S)$ as the key pattern. In the third phase, we reorder the items in $Tail(S)$ such that any item not in the key pattern precedes any item in the key pattern. After the reordering process, all the children of S are divided into promising children and unpromising children. Only promising children are searched by recursive calls.

Different from classical depth-first search algorithms, in PADS, as long as a key pattern Y is selected to reorder the tail of S, in the S-subtree, any subsets of Y are no longer visited. The key pattern Y is selected efficiently by using a probing process and taking advantage of the lookahead pruning.

Chapter 4

Optimization Techniques

In this section we propose three optimizations, namely *reduced counting*, *pattern expansion* and *head growth*, to improve the efficiency of PADS search.

4.1 Reduced Counting

Counting is the major cost in frequent itemset mining. In PADS, we can reduce the number of items to be counted in each projected database. In this subsection, we introduce the optimization technique.

As introduced in Chapter 2, to reduce the tree-traverse time, in FPMax* [15] an array technique is used. Using this technique, each FP-tree is associated with an array. Let S be an itemset, for every two items i and j in the header table of the FP-tree T_S, the array stores the support of $\{i, j\}$ in the S-projected database. For each item i in the header table of the projected FP-tree T_S, $Tail(S \cup \{i\})$ can be generated by reading the row of i in the array and collecting the item j such that the support in the cell of $\{i, j\}$ is larger than *min_sup*, instead of by scanning T_S once. In our implementation, we also integrate this technique. However, by using the PADS search method, we do not need to construct the complete array. Let Y be the key pattern selected to reorder $Tail(S)$. Since for any item $i \in Y \cap Tail(S)$, $S \cup \{i\}$-subtree do not need to be searched, thus the tail of $S \cup \{i\}$ does not need to be generated. Consequently, the row of i in the array does not need to be constructed. That is, in the array we only need to construct the rows for items in $Tail(S) - Y$.

Let us look at an example.

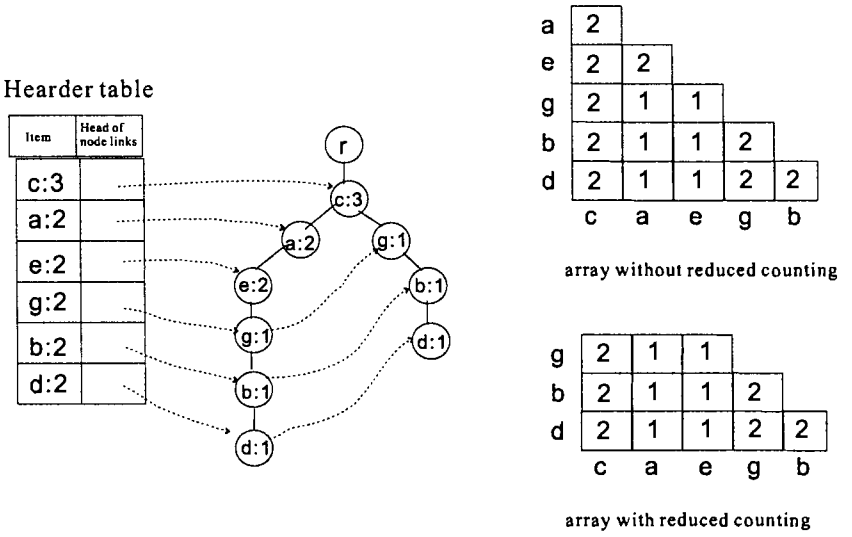


Figure 4.1: Reduced Counting

Example 3 In the database shown in Table 3.1, if **feac** is the selected as the key pattern to reorder the f-subtree, after reordering, the tail of **f** is {**c, a, e, g, b, d**}. When constructing the the projected FP-tree T_f, we do not need to construct the rows for **a** and **e**, as shown in Figure 4.1. □

The space and computation saved by reduced counting depends on the key pattern Y. The more items Y covers in $Tail(S)$, the more space and computation we can save. Let $l = Y \cap Tail(S)$, then the upper $(l-1)$ rows do not need to be constructed, thus we can save the cost of computing and storing $(l-1) * (l-2)/2$ array cells. Since the key pattern maximizes l, it also maximizes the saving greedily.

4.2 Pattern Expansion

When searching the S-subtree, the more items in $Tail(S)$, the larger the S-subtree and the search space. If we can identify some items in $Tail(S)$ that definitely appear in the all the MFIs in the S-subtree, then we can remove these items from $Tail(S)$ and add them to S. This adjustment reduces the size of the S-subtree and thus improves the efficiency. In this

section, we take advantage of the FP-tree data structure and develop a technique called *pattern expansion* to achieve this goal.

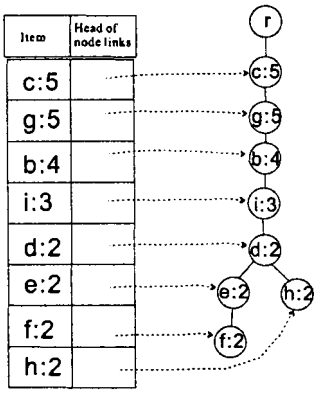


Figure 4.2: Pattern expansion

Our pattern expansion technique is similar to the PEP technique proposed in [10]. With the FP-tree structure, we can do better.

Definition 4 (Single Prefix) *A single prefix in an FP-tree is a path from the root of the FP-tree to a node N such that N is the only node on this path that has more than one child.*

Suppose we are searching the S-subtree, and T_S is the S-projected FP-tree, for any item in the single prefix of the FP-tree, we can simply move it from the $Tail(S)$ to S. Here we prove the correctness of pattern expansion.

Theorem 4 (Correctness of Pattern Expansion) *In the projected FP-tree T_S, let T_1 be the set of items in the single prefix, then all MFIs in the S-subtree is a superset of T_1.*

Proof. Let $T_2 = Tail(S) - T_1$, and $Trans(Y)$ be the set of transactions having Y. It is easy to see that $\forall i \in T_2$, $Trans(S \cup \{i\}) = Trans(S \cup T_1 \cup \{i\})$. Any MFI M in the S-subtree should have at least one item $i \in T_2$, otherwise $M \subseteq X \cup T_1 \subset S \cup T_1 \cup \{i\}$. Since $S \cup \{i\}$ is frequent and $Trans(S \cup \{i\}) = Trans(S \cup T_1 \cup \{i\})$, $S \cup T_1 \cup \{i\}$ is also frequent, thus M is not maximal.

Let M be a frequent itemset in the S-subtree and $N = M \cap T_2$, then $M \supseteq S \cup N$, thus $Trans(M) \subseteq Trans(S \cup N)$. Since for any $i \in T_2$, $Trans(S \cup \{i\}) = Trans(S \cup T_1 \cup \{i\})$,

thus $Trans(M) \subseteq Trans(S \cup N) = Trans(S \cup T_1 \cup N)$. Thus it is impossible that an MFI in the S-subtree does not have T_1 as its subset. □

Let us look at an example.

Example 4 suppose we are searching the S-subtree and the projected FP-tree T_S as shown in Figure 4.2. The single prefix in T_S is the path $c - g - b - i - d$. Then, searching the S-subtree with tail $\{c, g, b, i, d, e, f, h\}$ is equivalent to searching the $S \cup \{c, g, b, i, d\}$-subtree with tail $\{e, f, h\}$. □

Please note that different from PEP, for $i \in T_1$, $support(S \cup \{i\})$ is not necessarily equal to $support(S)$. Thus, compared with PEP, we can move more items from the tail to the head.

4.3 Head Growth

Projected database construction is one of the major costs in the MFI mining process, thus should be avoided as much as possible. Consider the situation where there is only one item $i \in Tail(S)$ that is not contained in the key pattern Y. If we only reorder the items in the tail, then we will need to construct the projected FP-tree T_S, and construct the projected FP-tree $T_{S\cup\{i\}}$ by scanning T_S. Since S has only one promising child $S \cup \{i\}$, T_S is constructed but used only once: for constructing $T_{S\cup\{i\}}$. In this situation, we can move i from the tail to the head. That is, instead of searching the S-subtree, we can directly search the $S \cup \{i\}$-subtree, then we can skip constructing the projected FP-tree T_S.

Please note that this *Head Growth* technique can be applied iteratively. After i is moved into the head, we compute $Tail(S \cup \{i\})$. If again there is only one item i' in $Tail(S \cup \{i\})$ that is not covered by the key pattern, i' is also moved from $Tail(S \cup \{i\})$ to the head, and we search the $S \cup \{i, i'\}$-subtree directly. In this way we may skip constructing multiple projected FP-trees at one time.

4.4 Algorithm

Based on the above analysis, we have the PADS algorithm as shown in Figure 4.3.

4.5 Comparison with LCM v2

As introduced in Chapter 2, the algorithm LCM v2 also reorders the tails and then prunes the unpromising branches. One may wonder what are the differences between PADS and LCM v2. In this section we make a thorough comparison between PADS and LCM v2.

1. PADS and LCM v2 are different in the selection of the key patterns. In LCM v2, when searching the S-subtree, it first chooses an item $i \in Tail(S)$, orders i before other items in $Tail(S)$, and fully searches the $S \cup \{i\}$-subtree. After that it picks the frequent itemset M of the maximum size in the $S \cup \{i\}$-subtree as the key pattern, and reorders items $Tail(S) - \{i\}$ according to M. Similar to PADS, only the children of S extended by items in $Tail(S) - M$ are searched further. The key patterns selected in this way are not necessarily good, since the item i is arbitrarily chosen. In contrast, PADS uses a systematic method to find heuristically good key patterns for tail reordering.

2. PADS selects the key patterns more efficiently than LCM v2. In LCM v2, key patterns are obtained by searching a significant proportion of the S-subtree. During the search, the projected databases are constructed recursively. In contrast, the key patterns in PADS are byproducts of the lookahead pruning. The cost is very little compared to database projections.

3. PADS and LCM v2 use different data structures. PADS uses FP-tree, while LCM v2 uses simple arrays.

4. PADS uses lookahead pruning to decide whether a frequent itemset is maximal. LCM uses a different method for maximality check: let S be a frequent itemset found and $Trans(S)$ be the set of transactions having S, S is maximal if and only if there is no item i not included by S but is frequent in $Trans(S)$. Please note that i may not be included in $Tail(S)$. So, for each transaction T in the S-projected database and each item i not in $Tail(S)$, LCM v2 needs to record the occurrence of i in T. The support of i in $Trans(S)$ is the number of the occurrences of i in all $T \in Trans(S)$. This method does not need to store all MFIs in main memory. The computation time for maximality check depends on the size of database, while in PADS it depends on the number of MFIs already found.

Input: a transaction database TDB and support threshold min_sup;
Output: the set of MFIs;
Method:
1: $F1$ =the set of frequent items;
2: construct FP-tree for TDB;
3: **for** each $i \in F1$
4: **call** $PADS(PDB, \{i\})$;

Function $PADS(PDB, X)$ $//PDB$ is the projected database of X's parent
5: let $Tail(X)$ = the set of frequent items in PDB;
6: `// head-and-tail pruning, progressive focusing search`
`// should be used in the subpattern matching`
if there exists an MFI Y found before such that $Tail(X) \cup X \subset Y$ **then**
return;
7: **if** $Tail(X) = \emptyset$ **then**
8: output X as an MFI;
9: let Y_1 be the candidate key pattern obtained from the probing process;
10: let Y_2 be the candidate key pattern as the byproduct of the subpattern checking;
11: let Y be the better key pattern between Y_1 and Y_2;
12: `//Head Increase`
if $|Tail(X) - Y| = 1$ **then**
$I = Tail(X) - Y$, $X = X \cup \{I\}$, $Tail(X) = Tail(X) - \{I\}$;
call $PADS(PDB, X)$
return;
13: make a dynamic search order R on items in $Tail(X)$ according to Y;
14: construct an FP-tree for PDB_X;
15: `//pattern expansion`
let Z be the set items in the single prefix of PDB_X,
$X = X \cup Z$, $Tail(X) = Tail(X) - Z$;
16: **for** each item $i \in (Tail(X) - Y)$ in the order of R
17: **call** $PADS(PDB_X, X \cup \{i\})$;
18: **return**;

Figure 4.3: The PADS algorithm.

Chapter 5

Empirical Study

We conducted an extensive performance study to evaluation the effectiveness of the dynamic search scheduling and the efficiency of our PADS algorithm. Here we report the experimental results on five real data sets. The five real data sets were prepared by Roberto Bayardo from the UCI datasets [11] and PUMSB. They have been used extensively in the previous studies [1, 10, 14, 15, 25, 28] as the benchmark data sets. Some characteristics of the five data sets are shown in Table 5.1. The Chess and Connect datasets are compiled from game state information, the mushroom dataset contains records describing the characteristics of various mushroom spices, Pumsb is prepared from the PUMS census data[1], and Pumsb* is obtained from Pumsb by removing items with higher than 80% support. The numbers of MFIs in those datasets with some selected minimum supports are shown in Figure 5.1. In this table a support threshold is presented as a percentage with respect to the total number of transactions in the data set, that is, $\frac{min\_sup}{|D|}$ where D is the data set in question.

All the experiments were conducted on a PC computer running the Microsoft Windows XP SP2 Professional Edition operating system, with a 3.0 GHz Pentium 4 CPU, 1.0 GB main memory, and a 160 GB hard disk. The programs were implemented in C/C++ using Microsoft Visual Studio. NET 2003.

We compare our method with two state-of-the-art algorithms FPMax* [14] [15] and LCM v2 [28]. FPMax* is the winner in mining maximal frequent itemsets at the Workshop on Frequent Itemset Mining Implementations 2003 (FIMI'03). According to the extensive

[1] http://www.census.gov/main/www/pums.html

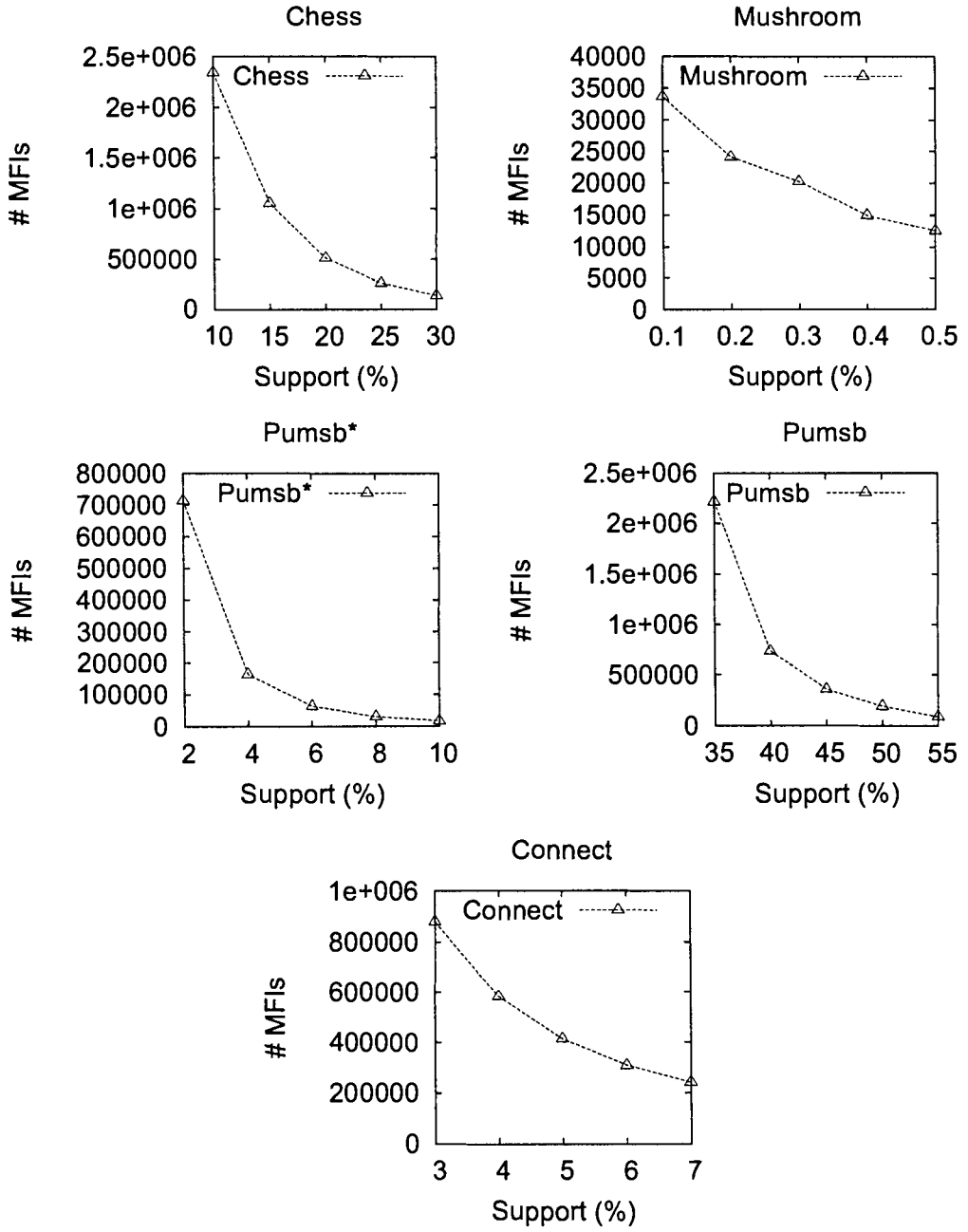


Figure 5.1: The number of MFIs on the five benchmark data sets with some minimum supports.

Data set	# tuples	# items	avg trans len
Chess	3,196	76	37
Mushroom	8,124	120	23
Pumsb*	49,046	2,088	50
Pumsb	49,046	2,113	74
Connect	67,557	150	43

Table 5.1: Characteristics of benchmark data sets.

empirical study reported at FIMI'03, it outperforms MAFIA, GenMax and other MFI mining algorithms. LCM v2 is the winner at FIMI'04. It provides the functionalities for mining all/closed/maximal frequent itemsets. LCM v2 is currently the best algorithm for mining closed frequent itemsets. With respect to mining maximal frequent itemsets, it also demonstrates good performance and is thus included in our empirical study. We obtained the source codes of FPMax* and LCM v2 from the Frequent Itemset Mining Implementations Repository website (http://fimi.cs.helsinki.fi/).

We firstly make comparison on three aspects: the runtime, the memory consumption, and the scalability with respect to the number of transactions in the database. To illustrate the reasons why PADS is more efficient, we also make comparison on the number of database projections and the number of maximality check operations, as database projections and maximality checks are the most costly operations in the mining process.

5.1 The Runtime

Figure 5.2 shows the runtime comparison of the three algorithms on the five data sets. The curve PADS indicates the runtime of PADS with optimizations, while the curve PADS- indicates the runtime without optimizations. It should be mentioned that LCM v2 has execution problems under some circumstances. On the Pumsb dataset with *min_sup* lower than 40%, on the Pumsb* dataset with *min_sup* lower than 6%, and on the Connect dataset with *min_sup* lower than 0.2%, LCM v2 gives segmentation faults and cannot finish properly. Therefore parts of its results are missing.

Figure 5.2 clearly shows that PADS outperforms FPMax* and LCM on the five data sets. Compared with FPMax*, the lower the support threshold, the larger the difference in runtime. With a smaller support threshold, more itemsets and longer itemsets are qualified

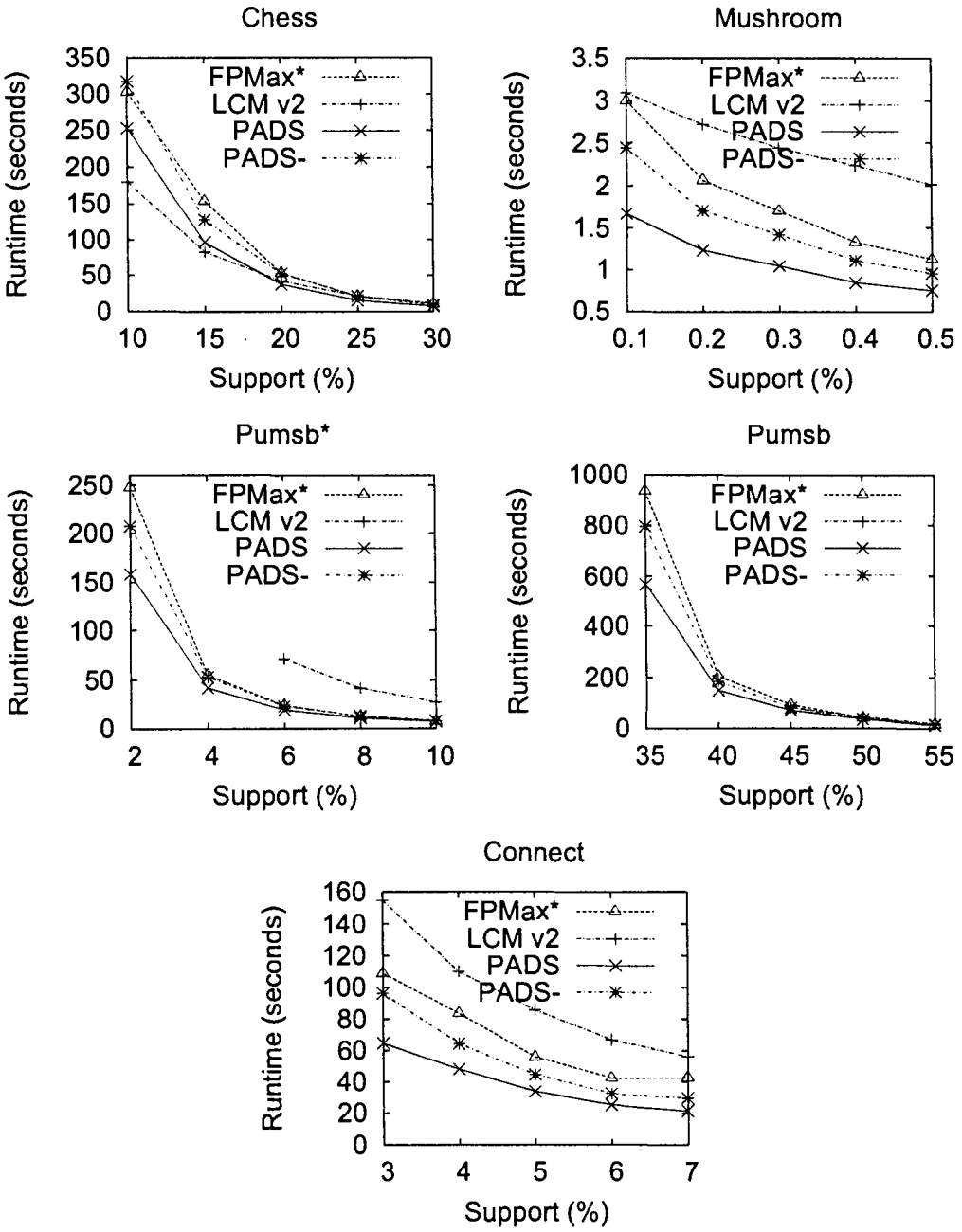


Figure 5.2: The Runtime Comparison of the Three Algorithms.

as frequent itemsets. This trend suggests that PADS is more scalable than FPMax* on mining a large number of long frequent itemsets. When the support threshold is low, the difference in runtime between the two methods can be more than 60%.

PADS outperforms LCM v2 clearly for most of the time, especially on the Mushroom and the Connect datasets. The only circumstance LCM v2 outperforms PADS is on the Chess dataset with $min\_sup \leq 15\%$. The reason is that the number of MFIs is large (more than 1 million) but the database size is relatively small (only 3,196 tuples), so the maximality check method of LCM v2 still works well.

The figures also show the effectiveness of the optimization techniques. On the five datasets, the optimization techniques contribute to the improvement in runtime as much as the reordering technique.

5.2 Memory Consumption

Figure 5.3 shows the virtual memory consumption of the three algorithms on the five datasets. The memory consumption values shown are peak values during the execution. On all these datasets, the memory usages of PADS and FPMax* are very close to each other. This is because PADS and FPMax* use the same data structure to store the projected databases and MFIs. In fact in both PADS and FPMax*, the first FP-tree $T_\emptyset$ and the first MFI tree $MFI\_T_\emptyset$ contribute to the major part of the memory usage. This is because projected FP-trees and projected MFI trees are typically of much smaller sizes.

We also notice that there are some minor differences between the memory consumption of PADS and FPMax*. The reasons are as follows. Firstly, in PADS, to select the best key pattern to reorder the tail of an itemset S, we maintain two arrays: one records $|Tail(S) \cap M|$ and one records the address of M, where M is an MFI already found. The sizes of the two arrays depend on the number of MFIs having S. Thus, compared with FPMax*, PADS needs some additional memory. Secondly, by adopting the probing process and the head growth technique, PADS can avoid constructing the projected databases for some nodes in the search tree. Thus, the maximum memory needed by PADS for holding the projected databases can be less than that needed by FPMax*.

It can also be seen that, different from LCM v2, PADS and FPMax* demonstrate different characteristics in memory consumption. The memory usage of PADS and FPMax* is more sensitive to the number of MFIs, while the memory usage of LCM v2 is more sensitive

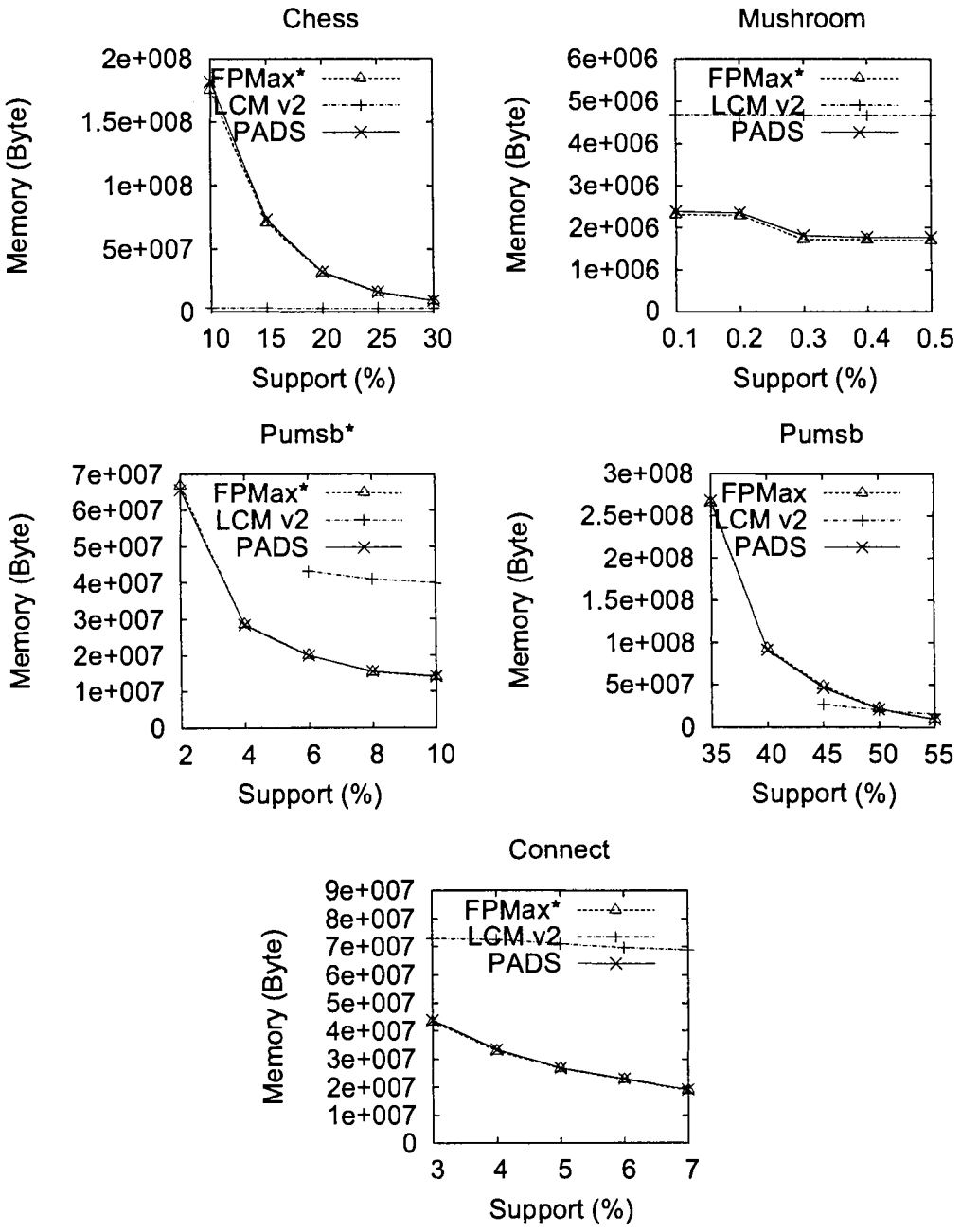


Figure 5.3: The Memory Comparison of the Three Algorithms.

to the size of the database. When the dataset is small but the number of MFIs is large, such as the Chess dataset with low minimum support, LCM v2 consumes less memory. In contrast, when the dataset is large but the number of MFIs is small, such as the Pumsb* dataset with high minimum support, FPMax* and PADS need less memory.

The above difference is due to the fact that LCM uses a different method for maximality check. It does not store the current set of MFIs in main memory. Instead, it uses the following observation: let S be a frequent itemset found and $Trans(S)$ be the set of transactions having S, S is maximal if and only if there is no item i not included by S but is frequent in $Trans(S)$. Please note that i may not be included in $Tail(S)$. Since the S-projected database consists of only items in $Tail(S)$, to conduct maximality check, for each transaction T in the S-projected database and an item i not in $Tail(S)$, LCM v2 needs to record the occurrence of i in T. The support of i in $Trans(S)$ is the number of the occurrences of i in all $T \in Trans(S)$. The memory consumption of LCM v2 is thus more sensitive to the size of the database.

5.3 Scalability

Figure 5.4 shows the scalability of the three algorithms on the three datasets of relatively large size: Pumsb, Pumsb* and Connect. We study the scalability of the three algorithms in two aspects: runtime scalability and memory scalability. We fixed the minimum support *min_sup*. For each dataset, we randomly generate four reduced datasets whose sizes range from 20% to 80% of its original size. Then we record the runtime and memory consumption of the three algorithms on those datasets.

It can be seen that PADS shows the best scalability, while LCM v2 has the worst scalability. The efficiency of PADS shows the advantage of our search method. The cost in LCM v2 is caused by two reasons. Firstly, as mentioned in Section 4.5, in LCM v2, the time needed by the maximality checks depends on the size of the projected databases, instead of the number of MFIs already found. On large databases, the projected databases may still consist of many transactions, making the maximality check slow. Secondly, LCM v2 uses simple arrays, instead of FP-trees, as the main data structure. In FP-trees, when the size of the database increases, more transactions can share common prefixes. In contrast, using simple arrays, only identical transactions can be merged. Therefore the FP-tree is more compact than the simple array, and database projection can be performed more efficiently.

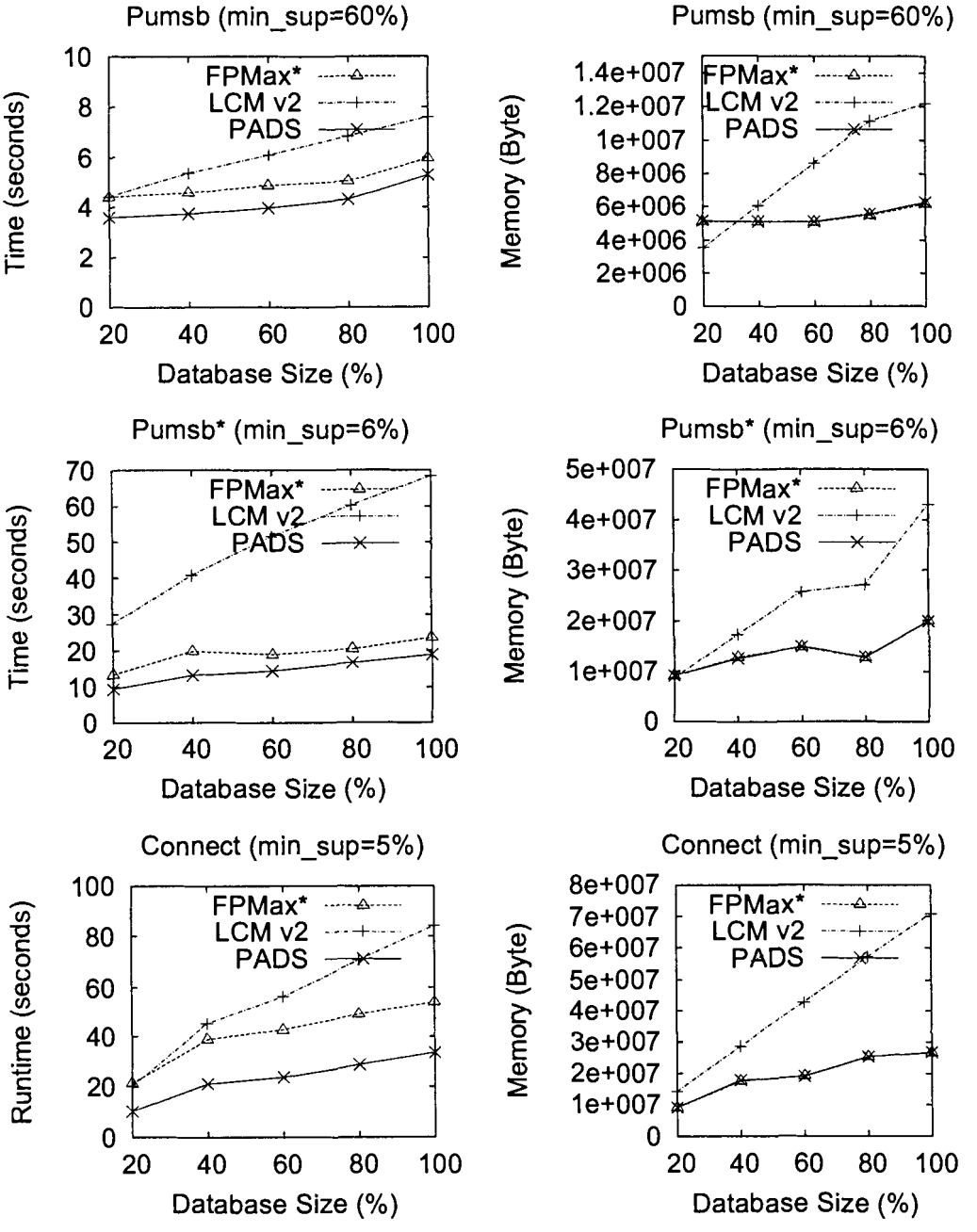


Figure 5.4: Scalability on Three Datasets .

In terms of memory usage, as analyzed in Section 5.2, PADS and FPMax* share the same characteristics. The memory consumption in LCM v2 increases more quickly than both FPMax* and PADS. The first reason is that LCM v2 uses database reduction to conduct maximality checks which have to record extra information for each transaction in the projected database. In addition, LCM v2 uses simple arrays which are not as compact as FP-trees, therefore more memory is needed.

5.4 Number of Database Projections and Maximality Check Operations

What are the major reasons that PADS outperforms FPMax* and LCM v2? The major cost of MFI mining in the depth-first manner comes from two aspects: constructing projected databases and checking whether a frequent itemset or the union of the head and the tail of a node is a subset of some MFIs found before.

5.4.1 Number of Database Projections

Figure 5.5 shows the comparison of the three algorithms in terms of the number of projected databases. For PADS there are two kinds of projections: physical projections and pseudo projections. The number of physical projections is counted for comparison.

It is shown that FPMax* constructs much fewer projected databases than LCM v2, even though LCM v2 adopts the reordering technique which can reduce the number of database projections. The reasons are as follows.

Let S be an itemset, since LCM v2 does not store the MFIs found so far in main memory, it cannot utilize the current set of MFIs to decide whether $S \cup Tail(S)$ is frequent. Instead, it has to explore the leftmost path of the S-subtree, during which up to $|Tail(S)|$ projected databases may be constructed.

In addition, in FPMax*, an optimization technique is used: when the S-projected FP-tree T_S has only a single path, $S \cup Tail(S)$ is known frequent, thus we do not need to construct projected databases for any descendants of S in the search tree. In contrast, LCM v2 needs to construct projected databases recursively until the leftmost child of S is searched.

Due to above two reasons, in FPMax* the number of projected databases can be less

than LCM v2 even though LCM v2 adopts the reordering technique.

It can be seen that on all datasets, PADS constructs much fewer projected databases than FPMax* and LCM v2. The number of projected databases constructed by PADS is about $15 \sim 25\%$ of that by FPMax* and $6 \sim 15\%$ of that by LCM v2. This shows the power of our PADS search method and the effectiveness of the optimization techniques.

5.4.2 Number of Maximality Check Operations

Figure 5.6 shows the comparison of the three algorithms in terms of the number of the maximality check operations. PADS again needs fewer maximality check operations than both FPMax* and LCM v2 in most cases, but the comparison between FPMax* and LCM v2 is not stable. Here we give the following analysis.

Firstly, since maximality check or lookahead check is needed to prune the unpromising subspaces, by reordering the items in the tail, the search spaces of PADS and LCM v2 are more compact. Some subspaces can be pruned immediately without conducting any maximality check. Thus, compared with FPMax*, PADS and LCM v2 can save maximality checks by avoiding searching unpromising subspaces.

Secondly, suppose S is an itemset, and $S \cup Tail(S)$ is a subset of some MFIs already found. Since LCM v2 does not store the current set of MFIs in the main memory, it cannot decide the maximality of $S \cup Tail(S)$ by checking whether $S \cup Tail(S)$ is a subset of some MFI. Its maximality check method can only decide whether S is maximal or not. While PADS and FPMax* can stop searching the S-subtree in this case, LCM v2 has to explore the leftmost child of S. Thus, compared with PADS and FPMax*, LCM v2 needs additional $|Tail(S)|$ maximality checks to prune the S-subtree. This is the reason why with the reordering technique, LCM v2 may still need more maximality checks than FPMax*.

The saving in generating fewer projected databases and fewer maximality checks well explains why PADS is more efficient than FPMax* and LCM v2.

5.5 Summary

The experimental results on the five benchmark datasets show that PADS is more efficient than FPMax* and LCM v2 in most cases. The efficiency of PADS comes from the fewer projected databases generated and fewer maximality check operations needed. With respect

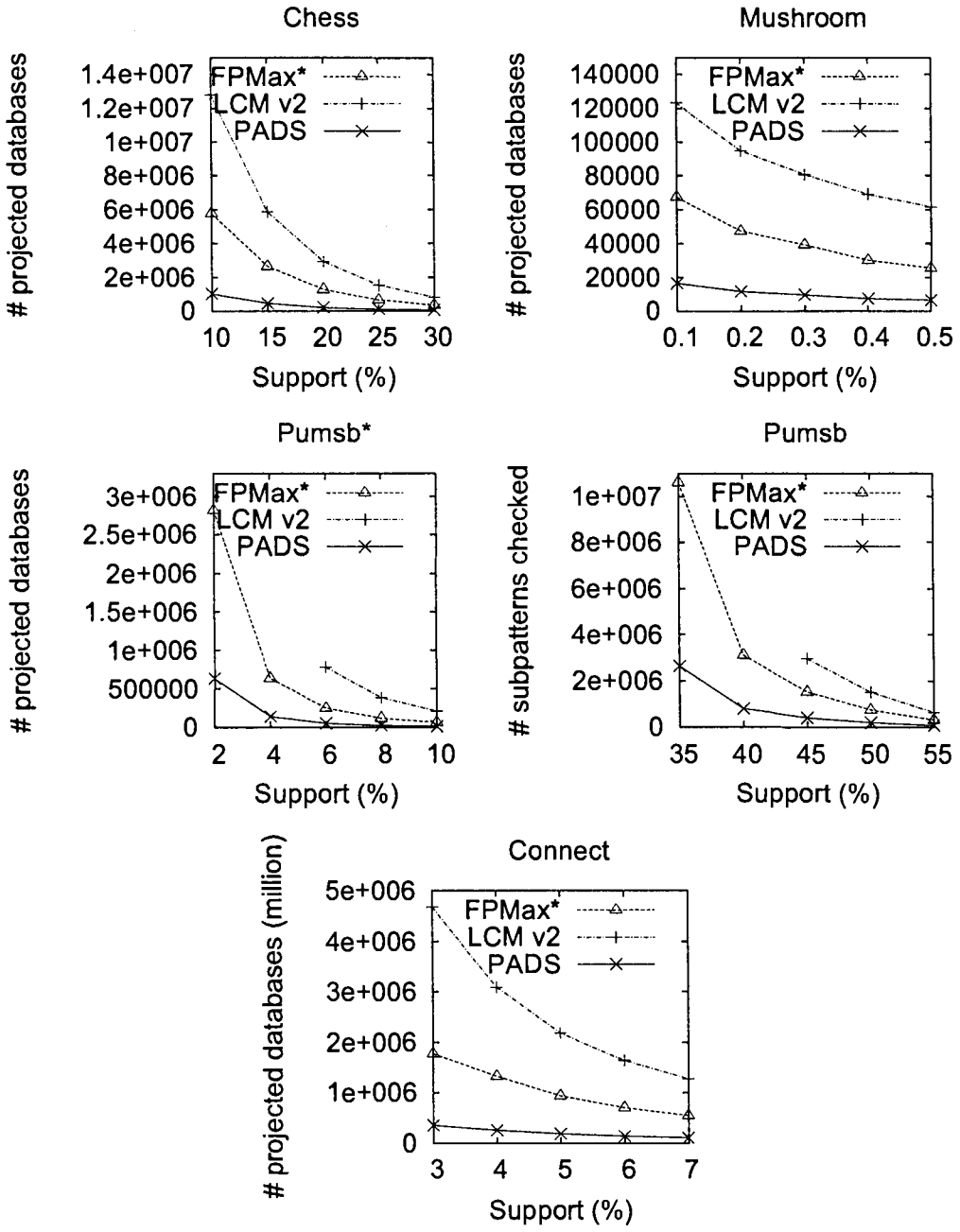


Figure 5.5: Number of Projected Database Generated.

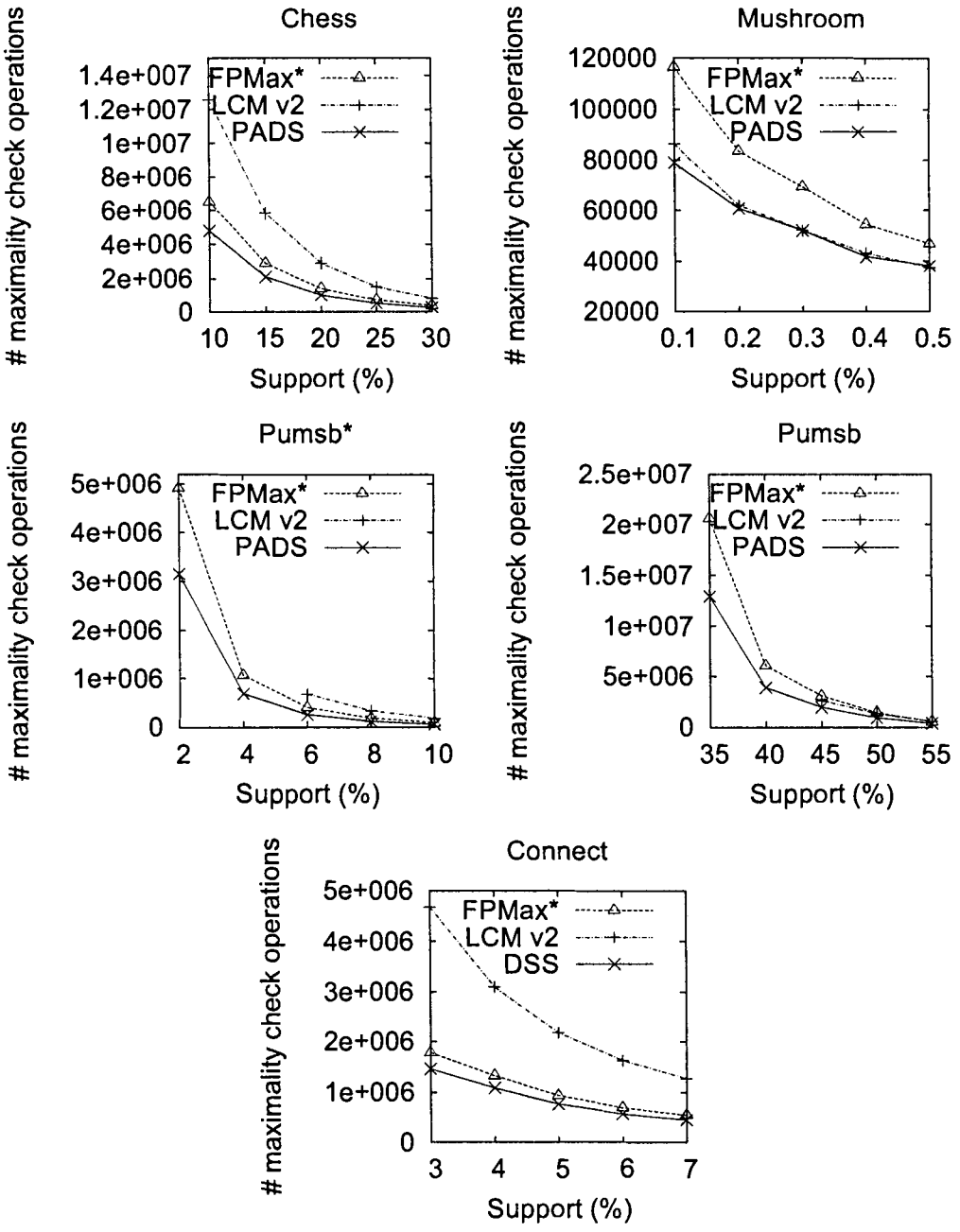


Figure 5.6: Number of Maximality Check Operations

to the memory consumption, PADS shares similar characteristics with FPMax*. The memory needed by PADS and FPMax* is more sensitive to the number of MFIs, instead of the size of the database.

Chapter 6

Conclusion

In this thesis we introduced a new search method for mining maximal frequent itemsets from transaction databases. Our major contribution is that we do not follow the classical enumeration order which is used to enumerate all frequent itemsets. Instead, we proposed a novel *probing and reordering* search method, which takes the current set of mining results to schedule the future search, and makes the search more efficient. We developed three optimization techniques to improve the efficiency. As shown by the extensive empirical study using the benchmark real data sets, our method outperforms FPMax* and LCM v2, two state-of-the-art methods in maximal frequent itemset mining.

Though we introduced our search method in the context of maximal frequent itemset mining, the Pattern-Aware Dynamic Scheduling method, however, may also be applied to other problems. Next we discuss some possible extensions.

The maximal clique search problem bears many similarities to maximal frequent itemset mining. They are both searching for maximal combinations of items or vertices. With some necessary modifications, our method may be applied to the maximal clique search problem.

In addition to frequent itemset mining, mining other frequent patterns is also well studied by the data mining community. For example, frequent sequential patterns, frequent subgraphs and frequent subtrees. One interesting question we may ask is whether our Pattern-Aware Dynamic Scheduling approach can be applied to those problems.

1. Mining closed/maximal sequential patterns. The problem of mining sequential patterns has been well studied. While frequent itemset mining searches for frequent *combinations* of items, sequential pattern mining is interested in frequent *subsequences* of

items. Then, can we search the sequential patterns in a more intelligent way so that we can avoid mining non-closed sequential patterns as much as possible? Can we mine maximal sequential patterns in a similar way by taking advantage of the current set of maximal frequent sequential patterns? Those are the interesting problems to be studied.

2. Mining closed/maximal frequent subgraphs. Graphs are more complex structures than itemsets and sequences. Mining frequent subgraphs is more difficult than mining frequent itemsets. Similar to the set enumeration tree in the frequent itemset mining, in graph mining canonical label systems are used to systematically enumerate all graphs without having any duplicate. In the current approaches to graph mining, once a canonical label system is used, the enumeration order is determined. Our Pattern-Aware Dynamic Scheduling approach, however, requires changing the enumeration order dynamically. Then, can we add dynamics to the canonical label system so that the enumeration order can be changed according to the mining results we already got? This can be a problem to study, too.

Bibliography

[1] Ramesh C. Agarwal, Charu C. Aggarwal, and V. V. V. Prasad. Depth first generation of long patterns. In *KDD '00: Proceedings of the sixth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 108–118, New York, NY, USA, 2000. ACM Press.

[2] Ramesh C. Agarwal, Charu C. Aggarwal, and V. V. V. Prasad. A tree projection algorithm for generation of frequent item sets. *Journal of Parallel Distributed Computing*, 61(3):350–371, 2001.

[3] Charu C. Aggarwal. Towards long pattern generation in dense databases. *SIGKDD Explor. Newsl.*, 3(1):20–26, 2001.

[4] Rakesh Agrawal, Johannes Gehrke, Dimitrios Gunopulos, and Prabhakar Raghavan. Automatic subspace clustering of high dimensional data for data mining applications. In *SIGMOD '98: Proceedings of the 1998 ACM SIGMOD international conference on Management of Data*, pages 94–105, New York, NY, USA, 1998. ACM Press.

[5] Rakesh Agrawal, Tomasz Imielinski, and Arun Swami. Mining association rules between sets of items in large databases. In *SIGMOD '93: Proceedings of the 1993 ACM SIGMOD international conference on Management of data*, pages 207–216, New York, NY, USA, 1993. ACM Press.

[6] Rakesh Agrawal and Ramakrishnan Srikant. Fast algorithms for mining association rules in large databases. In *VLDB '94: Proceedings of the 20th International Conference on Very Large Data Bases*, pages 487–499, San Francisco, CA, USA, 1994. Morgan Kaufmann Publishers Inc.

[7] Rakesh Agrawal and Ramakrishnan Srikant. Mining sequential patterns. In *ICDE*, pages 3–14, 1995.

[8] Kevin Beyer and Raghu Ramakrishnan. Bottom-up computation of sparse and iceberg cube. In *SIGMOD '99: Proceedings of the 1999 ACM SIGMOD international conference on Management of data*, pages 359–370, New York, NY, USA, 1999. ACM Press.

[9] Sergey Brin, Rajeev Motwani, and Craig Silverstein. Beyond market baskets: generalizing association rules to correlations. In *SIGMOD '97: Proceedings of the 1997 ACM SIGMOD international conference on Management of data*, pages 265–276, New York, NY, USA, 1997. ACM Press.

[10] Douglas Burdick, Manuel Calimlim, and Johannes Gehrke. MAFIA: A maximal frequent itemset algorithm for transactional databases. In *Proceedings of the 17th International Conference on Data Engineering*, pages 443–452, Washington, DC, USA, 2001. IEEE Computer Society.

[11] C.L. Blake D.J. Newman, S. Hettich and C.J. Merz. UCI repository of machine learning databases, 1998.

[12] Guozhu Dong and Jinyan Li. Efficient mining of emerging patterns: discovering trends and differences. In *KDD '99: Proceedings of the fifth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 43–52, New York, NY, USA, 1999. ACM Press.

[13] Karam Gouda and Mohammed Javeed Zaki. Efficiently mining maximal frequent itemsets. In *ICDM '01: Proceedings of the 2001 IEEE International Conference on Data Mining*, pages 163–170, Washington, DC, USA, 2001. IEEE Computer Society.

[14] Gosta Grahne and Jianfei Zhu. Efficiently using prefix-trees in mining frequent itemsets. In *FIMI'03 Workshop on Frequent Itemset Mining Implementations*, November 2003.

[15] Gosta Grahne and Jianfei Zhu. Fast algorithms for frequent itemset mining using fp-trees. *IEEE Transactions on Knowledge and Data Engineering*, 17(10):1347–1362, 2005.

[16] Jiawei Han, Jian Pei, and Yiwen Yin. Mining frequent patterns without candidate generation. In *SIGMOD '00: Proceedings of the 2000 ACM SIGMOD international conference on Management of data*, pages 1–12, New York, NY, USA, 2000. ACM Press.

[17] Guozhu Dong Jiawei Han and Yiwen Yin. Efficient mining of partial periodic patterns in time series database. In *ICDE '99: Proceedings of the 15th International Conference on Data Engineering*, page 106, Washington, DC, USA, 1999. IEEE Computer Society.

[18] Jinyan Li, Guozhu Dong, Kotagiri Ramamohanarao, and Limsoon Wong. Deeps: A new instance-based lazy discovery and classification system. *Mach. Learn.*, 54(2):99–124, 2004.

[19] Dao Lin and Zvi M. Kedem. Princer-search: A new algorithm for discovering the maximum frequent itemset. In *Proceedings of the 1998 International Conference on Extending DataBase Technology (EDBT'98)*, pages 105–109, 1998.

[20] Bing Liu, Wynne Hsu, and Yiming Ma. Integrating classification and association rule mining. In *KDD '98: Proceedings of the 9th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 80–86, 1998.

[21] Heikki Mannila, Hannu Toivonen, and A. Inkeri Verkamo. Discovery of frequent episodes in event sequences. *Data Min. Knowl. Discov.*, 1(3):259–289, 1997.

[22] Jian Pei, Guozhu Dong, Wei Zou, and Jiawei Han. Mining condensed frequent-pattern bases. *Knowl. Inf. Syst.*, 6(5):570–594, 2004.

[23] Jian Pei, Jiawei Han, Hongjun Lu, Shojiro Nishio, Shiwei Tang, and Dongqing Yang. H-mine: Hyper-structure mining of frequent patterns in large databases. In *ICDM '01: Proceedings of the 2001 IEEE International Conference on Data Mining*, pages 441–448, Washington, DC, USA, 2001. IEEE Computer Society.

[24] Jian Pei, Jiawei Han, and Runying Mao. CLOSET: An efficient algorithm for mining frequent closed itemsets. In *ACM SIGMOD Workshop on Research Issues in Data Mining and Knowledge Discovery*, pages 21–30, 2000.

[25] Jr. Roberto J. Bayardo. Efficiently mining long patterns from databases. In *SIGMOD '98: Proceedings of the 1998 ACM SIGMOD international conference on Management of data*, pages 85–93, New York, NY, USA, 1998. ACM Press.

[26] R. Rymond. Search through systematic set enumeration. In *Proceedings of the 1992 International Conference on Knowledge Representation and Reasoning (KR'92)*, pages 539–550, 1992.

[27] Craig Silverstein, Sergey Brin, Rajeev Motwani, and Jeffrey D. Ullman. Scalable techniques for mining causal structures. In Ashish Gupta, Oded Shmueli, and Jennifer Widom, editors, *VLDB'98, Proceedings of 24th International Conference on Very Large Data Bases, August 24-27, 1998, New York City, New York, USA*, pages 594–605. Morgan Kaufmann, 1998.

[28] Yuzo Uchida Takeaki Uno, Tatsuya Asai and Hiroki Arimura. LCM ver. 2: Efficient mining algorithms for frequent/closed/maximal itemsets. In *Proceedings of the IEEE ICDM Workshop on Frequent Itemset Mining Implementations(FIMI 04)*, Brighton, UK, 2004.

[29] Mohammed J. Zaki. Scalable algorithms for association rule mining. *IEEE Transactions on Knowledge and Data Engineering*, 12(3):372–390, 2001.

[30] Mohammed J. Zaki and Karam Gouda. Fast vertical mining using diffsets. Technical report, RPI, 11 2001.

[31] Mohammed J. Zaki and Karam Gouda. Fast vertical mining using diffsets. In *KDD '03: Proceedings of the ninth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 326–335, New York, NY, USA, 2003. ACM Press.

[32] Mohammed J. Zaki and C. Hsiao. Charm: an efficient algorithm for closed association rule mining. Technical report, RPI, 1999.

[33] Mohammed J. Zaki, Srinivasan Parthasarathy, Mitsunori Ogihara, and Wei Li. New algorithms for fast discovery of association rules. Technical report, Rochester, NY, USA, 1997.