# PATTERN-GROWTH METHODS FOR FREQUENT PATTERN MINING

by

Jian Pei

B.Eng., Shanghai Jiaotong University, 1991

M.Eng., Shanghai Jiaotong University, 1993

Ph.D. Candidate, Peking University, 1999

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

SIMON FRASER UNIVERSITY
June 13, 2002

## APPROVAL

| | |
|---|---|
| **Name:** | Jian Pei |
| **Degree:** | Doctor of Philosophy |
| **Title of thesis:** | Pattern-growth Methods for Frequent Pattern Mining |

**Examining Committee:** Dr. Ramesh Krishnamurti
Chair

Dr. Jiawei Han, Senior Supervisor

Dr. Arthur L. Liestman, Supervisor

Dr. Martin Ester, SFU Examiner

Dr. Raymond T. Ng, External Examiner,
Professor of Computer Science,
University of British Columbia

**Date Approved:**

# Abstract

Mining frequent patterns from large databases plays an essential role in many data mining tasks and has broad applications. Most of the previously proposed methods adopt apriori-like candidate-generation-and-test approaches. However, those methods may encounter serious challenges when mining datasets with prolific patterns and/or long patterns.

In this work, we develop a class of novel and efficient pattern-growth methods for mining various frequent patterns from large databases. Pattern-growth methods adopt a divide-and-conquer approach to decompose both the mining tasks and the databases. Then, they use a pattern fragment growth method to avoid the costly candidate-generation-and-test processing completely. Moreover, effective data structures are proposed to compress crucial information about frequent patterns and avoid expensive, repeated database scans. A comprehensive performance study shows that pattern-growth methods, *FP-growth* and *H-mine*, are efficient and scalable. They are faster than some recently reported new frequent pattern mining methods.

Interestingly, pattern growth methods are not only efficient, but also effective. With pattern growth methods, many interesting patterns can also be mined efficiently, such as patterns with some tough non-anti-monotonic constraints and sequential patterns. These techniques have strong implications to many other data mining tasks.

*To my family*

*"I do not feel obliged to believe that the same God who has endowed us with sense, reason, and intellect has intended us to forgo their use."*

*— Galileo Galilei (1564-1642)*

*"I have gathered a posie of other men's flowers, and nothing but the thread that binds them is mine own."*

*— Michel de Montaigne (1533-1592)*

# Acknowledgments

I wish to express my deep gratitude to my supervisor and mentor Dr. Jiawei Han. I thank him for his continuous encouragement, confidence and support, and for sharing with me his knowledge and experience. As an advisor, he taught me practices and skills that I will use in my academic career.

I am very thankful to my supervisor, Dr. Arthur L. Liestman, for his insightful comments and advice. He is always very helpful for discussion about research and career development. I really appreciate his help to improve the quality of my thesis.

Part of this work is done in collaboration with Dr. Laks V.S. Lakshmanan, Dr. Guozhu Dong and Dr. Ke Wang. I thank them for the knowledge and skills they imparted through the collaboration. Working with them is always so enjoyable. My gratitude and appreciation also goes to Dr. Martin Ester and Dr. Raymond Ng for serving as examiners of my thesis. I also want to thank Ms. Joyce Lam and Mr. M. Riadul Mannan for proofreading my thesis.

My deepest thanks to Professor Shiwei Tang and Professor Dongqing Yang in Peking University and Professor Liangxian Xu in Shanghai Jiaotong University for educating and training me for my career.

I would also like to thank the many people in our department, support staff and faculty, for always being helpful over the years. I thank my friends at Simon Fraser University for their help. A particular acknowledgement goes to Carole Edwards, Kan Hu, Kersti Jaager, Eddie Kim, Joyce Lam, Wenmin Li, Runying Mao, Behzad Mortazavi-Asl, Helen Pinto, Wei Wang, Jack Yeung, Yiwen Yin, Osmar R. Zaïane, Senqiang Zhou, and Hua Zhu.

Last but not least, I am very grateful to my parents and my wife for their continuous moral support and encouragement. I hope I will make them proud of my achievements, as I am proud of them. Their love accompanies me wherever I go.

# Contents

# List of Tables

# List of Figures

# Chapter 1

# Introduction

"The universe is full of magical things patiently waiting for our wits to grow sharper."[1] Data mining is to find valid, novel, potentially useful, and ultimately understandable patterns in data [FPSSe96]. In general, there are many kinds of patterns (knowledge) that can be discovered from data. For example, association rules can be mined for market basket analysis, classification rules can be found for accurate classifiers, clusters and outliers can be identified for customer relation management.

Frequent pattern mining plays an essential role in many data mining tasks, such as mining association rules [AS94, KMR$^+$94], correlations [BMS97], causality [SBMU98], sequential patterns [AS95], episodes [MTV97], multi-dimensional patterns [LSW97, KHC97], max-patterns [Bay98], partial periodicity [HDY99], and emerging patterns [DL99]. Frequent pattern mining techniques can also be extended to solve many other problems, such as iceberg-cube computation [BR99] and classification [LHM98]. Thus, effective and efficient frequent pattern mining is an important and interesting research problem.

## 1.1 Motivation

Most of the previous studies on frequent pattern mining, such as [AS94, KMR$^+$94, SON95, PCY95, LSW97, STA98, SVA97, NLHP98, GLW00], adopt an *Apriori*-like approach, which is based on an *anti-monotone* Apriori *heuristic* [AS94]: *if any length $k$ pattern is not frequent in the database, its length $(k+1)$ super-pattern can never be frequent.* The essential idea is

[1]By Eden Phillpotts (1862-1960), English writer, poet, playwright.

to iteratively generate the set of candidate patterns of length $(k+1)$ from the set of frequent patterns of length $k$ (for $k \geq 1$), and check their corresponding occurrence frequencies in the database.

The *Apriori* heuristic achieves good performance gain by (possibly significantly) reducing the size of candidate sets. However, in situations with prolific frequent patterns, long patterns, or quite low minimum support thresholds, an *Apriori*-like algorithm may still suffer from the following two nontrivial costs:

- It is costly to handle a huge number of candidate sets. For example, if there are $10^4$ frequent 1-itemsets, the *Apriori* algorithm will need to generate more than $10^7$ length-2 candidates and test their occurrence frequencies. Moreover, to discover a frequent pattern of size 100, such as $\{a_1, \ldots, a_{100}\}$, it must generate $\begin{pmatrix} 100 \\ 1 \end{pmatrix}$ length-1 candidates, $\begin{pmatrix} 100 \\ 2 \end{pmatrix}$ length-2 candidates, and so on. Ultimately, it generates

$$\begin{pmatrix} 100 \\ 1 \end{pmatrix} + \begin{pmatrix} 100 \\ 2 \end{pmatrix} + \cdots + \begin{pmatrix} 100 \\ 100 \end{pmatrix} = 2^{100} - 1 > 10^{30}$$

  candidates in total. This is the inherent cost of candidate generation, no matter what implementation technique is applied.

- It is tedious to repeatedly scan the database and check a large set of candidates by pattern matching, which is especially true for mining long patterns.

As frequent pattern mining is an essential data mining task, developing efficient frequent mining techniques has been an important research direction in data mining.

There are some interesting questions that need to be answered.

- *Apriori* is one basic principle in frequent pattern mining. As analyzed, it has its advantages and disadvantages. To improve the efficiency of frequent pattern mining substantially, is there any way to obtain this advantage while avoiding the costly candidate-generation-and-test and repeated database scan operations?

- Frequent pattern mining often suffers not only from the lack of efficiency but also from the lack of effectiveness, i.e., there could be a huge number of frequent patterns generated from a database. Can we develop any method to derive some succinct expression of frequent patterns and also push the users' interest focus into the mining process?

- Frequent pattern mining has many potential applications. Can we extend the effective and efficient frequent pattern mining methods to solve some other interesting data mining problems?

This thesis tries to make good progress in answering the above questions.

## 1.2 Contributions

In this thesis, we study the problem of efficient and effective frequent pattern mining, as well as some of its extensions and applications. In particular, we make the following contributions.

- We systematically develop a *pattern-growth* method for frequent pattern mining. A novel algorithm, *FP-growth*, is proposed for efficiently mining frequent patterns from large dense datasets. Furthermore, to achieve efficient frequent pattern mining in various situations, we design *H-mine*, which is highly scalable and space preserving for very large databases.

- As an inherent problem, frequent pattern mining may return too many patterns. Constraint-based data mining is an important approach to solve the problem of effective data mining. We study the problem of constraint-based frequent pattern mining using pattern-growth methods. Our study shows that pattern-growth methods can push constraints deeper into the mining process, even including such constraints using aggregate $AVG()$ and $SUM()$, which other methods cannot handle.

- We extend the pattern-growth method to allow the mining of sequential patterns. Our study shows that pattern-growth methods are more efficient in mining large sequence databases. Interesting techniques are developed to solve the sequential pattern mining problem effectively.

## 1.3 Organization of the Thesis

The remainder of the thesis is structured as follows:

- In Chapter 2, we present the frequent pattern mining problem and an overview of related work systematically.

- In Chapter 3, a novel frequent pattern method, *FP-growth*, is developed. The correctness and efficiency of *FP-growth* are verified by theoretical analysis and experimental tests.

- Even though *FP-growth* is efficient in mining large dense databases, it may have the problem of building many recursive projected databases and *FP-tree*s and thus may be costly in space. In Chapter 4, we propose *H-mine*, which retains the advantages of *FP-growth* but avoids redundant sub-database/tree building. Our performance study shows that *H-mine* achieves good scalability in mining large databases and is also efficient in space.

- The problem of constraint-based frequent pattern mining is studied in Chapter 5 using pattern-growth methods. Not only do we examine the constraints proposed in previous studies, we also attempt to attack some tough ones. A new kind of constraint, called *convertible constraint*, is identified. Pattern-growth methods are developed for pushing various constraints deep into the mining process.

- We extend the pattern-growth methods to solve the sequential pattern mining problem in Chapter 6. The study indicates that, with some modification and customization, pattern-growth methods can be applied to mine patterns from various kinds of databases.

- We summarize the characteristics of pattern-growth methods in Chapter 7. Some interesting extensions and applications of pattern-growth methods are also discussed.

- The thesis concludes in Chapter 8. Interesting applications of pattern-growth methods are discussed and some future directions are presented.

# Chapter 2

# Problem Definition and Related Work

In this chapter, we first define the problem of frequent pattern mining, then we revisit the *Apriori* heuristic and algorithm. Several improvements of the *Apriori* algorithm are also discussed.

## 2.1 Frequent Pattern Mining Problem

The frequent pattern mining problem was first introduced by R. Agrawal, et al. in [AIS93] as mining association rules between sets of items.

Let $I = \{i_1, \ldots, i_m\}$ be a set of *items*. An *itemset* $X \subseteq I$ is a subset of items. Hereafter, we write itemsets as $X = i_{j_1} \cdots i_{j_n}$, i.e. omitting set brackets. Particularly, an itemset with $l$ items is called an *$l$-itemset*.

A *transaction* $T = (tid, X)$ is a tuple where *tid* is a *transaction-id* and $X$ is an itemset. A transaction $T = (tid, X)$ is said to *contain* itemset $Y$ if $Y \subseteq X$.

A *transaction database* $TDB$ is a set of transactions. The *support* of an itemset $X$ in transaction database $TDB$, denoted as $sup_{TDB}(X)$ or $sup(X)$, is the number of transactions in $TDB$ containing $X$, i.e.,

$$sup(X) = |\{(tid, Y)|((tid, Y) \in TDB) \wedge (X \subseteq Y)\}|$$

**Problem statement.** Given a user-specified *support threshold min_sup*, $X$ is called a *frequent itemset* or *frequent pattern* if $sup(X) \geq min\_sup$. The *problem of mining frequent itemsets* is to find the complete set of frequent itemsets in a transaction database $TDB$ with respect to a given support threshold $min\_sup$.

Association rules can be derived from frequent patterns. An *association rule* is an implication of the form $X \Longrightarrow Y$, where $X$ and $Y$ are itemsets and $X \cap Y = \emptyset$. The rule $X \Longrightarrow Y$ has *support* $s$ in a transaction database $TDB$ if $sup_{TDB}(X \cup Y) = s$. The rule $X \Longrightarrow Y$ holds in the transaction database $TDB$ with *confidence* $c$ where $c = \frac{sup(X \cup Y)}{sup(X)}$.

Given a transaction database $TDB$, a *support threshold* $min\_sup$ and a *confidence threshold* $min\_conf$, the *problem of association rule mining* is to find the complete set of association rules that have support and confidence no less than the user-specified thresholds, respectively.

Association rule mining can be divided into two steps. First, frequent patterns with respect to support threshold $min\_sup$ are mined. Second, association rules are generated with respect to confidence threshold $min\_conf$. As shown in many studies (e.g., [AS94]), the first step, mining frequent patterns, is significantly more costly in terms of time than the rule generation step.

As we shall see later, frequent pattern mining is not only used in association rule mining. Instead, frequent pattern mining is the basis for many data mining tasks, such as sequential pattern mining and associative classification. It also has broad applications, such as basket data analysis, cross-marketing, catalog design, sale campaign analysis, web log (click stream) analysis, etc.

## 2.2 *Apriori* Heuristic and Algorithm

To achieve efficient mining frequent patterns, an anti-monotonic property of frequent itemsets, called the *Apriori* heuristic, was identified in [AS94].

**Theorem 2.1 (*Apriori*)** *Any superset of an infrequent itemset cannot be frequent. In other words, every subset of a frequent itemset must be frequent.*

**Proof.** To prove the theorem, we only need to show $sup(X) \leq sup(Y)$ if $X \supseteq Y$.

Given a transaction database $TDB$. Let $X$ and $Y$ be two itemsets such that $X \supseteq Y$. For each transaction $T$ containing itemset $X$, $T$ also contains $Y$, which is a subset of $X$. Thus, we have $sup(X) \leq sup(Y)$. ∎

The *Apriori* heuristic can prune candidates dramatically. Based on this property, a fast frequent itemset mining algorithm, called *Apriori*, was developed. It is illustrated in the following example.

**Example 2.1 (*Apriori*)** Let the transaction database, $TDB$, be Table 2.1 and the minimum support threshold be 3.

| tid | Itemset | (Ordered) Frequent Items |
|---|---|---|
| 100 | $f, a, c, d, g, i, m, p$ | $a, c, f, m, p$ |
| 200 | $a, b, c, f, l, m, o$ | $a, b, c, f, m$ |
| 300 | $b, f, h, j, o$ | $b, f$ |
| 400 | $b, c, k, s, p$ | $b, c, p$ |
| 500 | $a, f, c, e, l, p, m, n$ | $a, c, f, m, p$ |

Table 2.1: A transaction database $TDB$.

*Apriori* finds the complete set of frequent itemsets as follows.

1. Scan $TDB$ once to find frequent items, i.e. items appearing in at least 3 transactions. They are $a, b, c, f, m, p$. Each of these six items forms a *length*-1 *frequent itemset*. Let $L_1$ be the complete set of length-1 frequent itemsets.

2. The set of *length*-2 *candidates*, denoted as $C_2$, is generated from $L_1$. Here, we use the *Apriori heuristic* to prune the candidates. Only those candidates that consist of frequent subsets can be potentially frequent. An itemset $xy \in C_2$ if and only if $x, y \in L_1$. Thus, $C_2 = \{ab, ac, \ldots, ap, bc, \ldots, mp\}$. There are $\begin{pmatrix} 6 \\ 2 \end{pmatrix} = 15$ itemsets in $C_2$. They form the candidate set $C_2$.

3. Scan $TDB$ once more to count the support of each itemset in $C_2$. The itemsets in $C_2$ passing the support threshold form the *length*-2 *frequent itemsets*, $L_2$. In this example, $L_2$ contains itemsets $ac$, $af$, $am$, $cf$, $cm$, and $fm$.

4. Then, we form the set of length-3 candidates. Only those length-3 itemsets for which every length-2 sub-itemset is in $L_2$ are qualified as candidates. For example, $acf$ is a length-3 candidate since $ac$, $af$ and $cf$ are all in $L_2$.

   One scan of $TDB$ identifies the subset of length-3 candidates passing the support threshold and form the set $L_3$ of length-3 frequent itemsets. A similar process goes on until no candidate can be derived or no candidate is frequent.

One can verify that the above process eventually finds the complete set of frequent itemsets in the database $TDB$. ■

The *Apriori* algorithm is presented as follows.

**Algorithm 1 (*Apriori*)**

**Input:** transaction database $TDB$ and support threshold $min\_sup$

**Output:** the complete set of frequent patterns in $TDB$ with respect to support threshold $min\_sup$

**Method:**

```
1. scan transaction database $TDB$ once to find $L_1$, the set of frequent 1-itemsets;
2. for ($k = 2$; $L_{k-1} \neq \emptyset$; $k++$) do
    (a) generate $C_k$, the set of length-$k$ candidates. A $k$-itemset $X$ is in $C_k$ if and
        only if every length-$(k-1)$ subset of $X$ is in $L_{k-1}$;
    (b) if $C_k = \emptyset$ then go to Step 3;
    (c) scan transaction database $TDB$ once to count the support for every itemset
        in $C_k$;
    (d) $L_k = \{X | (X \in C_k) \wedge (sup(X) \geq min\_sup)\}$;
3. return $\bigcup_{i=1}^{k} L_i$
```
■

Now, let us analyze the efficiency of the *Apriori* algorithm using this example.

- *The* Apriori *heuristic helps reducing the number of candidates significantly.* Since there are in total 14 items appearing in the database, there could be $\begin{pmatrix} 14 \\ 2 \end{pmatrix} = 91$ possible length-2 itemsets. As shown in the example, with the Apriori *heuristic*, we only need to check the support counts for 15 length-2 candidates. *Apriori* cuts 83.52% at the length-2 itemset level. As the length of candidates becomes longer, the number of possible combinations becomes larger, thus the cutting effect of the *Apriori* heuristic is sharper.

- *Even though* Apriori *can cut a lot of candidates, it could still be costly to handle a huge number of candidate itemsets in large transaction databases.* For example, if there are 1 million items and only 1% (i.e. $10^4$ items) are frequent length-1 itemsets, *Apriori* has to generate more than $10^7$ length-2 candidates, test each of their support and save them for length-3 candidates generation.

- *It is tedious to repeatedly scan the database and check a large set of candidates by pattern matching, which is particularly true if a long pattern exists. Apriori* is a level-by-level candidate-generation-and-test algorithm. To find a frequent itemset $X = x_1 \cdots x_{100}$, *Apriori* has to scan the database 100 times.

- Apriori *encounters difficulty in mining long patterns.* For example, to find a frequent itemset $X = x_1 \cdots x_{100}$, it has to generate-and-test $2^{100} - 1$ candidates.

## 2.3 Improvements over *Apriori*

In the past several years, many improvements over the *Apriori* algorithm have been proposed. In this section, we review some important proposals.

As shown in Section 2.2, the major bottlenecks in *Apriori* algorithm are in three aspects.

- The *Apriori* algorithm needs to scan the database multiple times. When mining a huge database, multiple database scans are costly. One feasible strategy to improve the efficiency of *Apriori* algorithm is to reduce the number of database scans.

- The *Apriori* algorithm has to generate a huge number of candidates. Storing and counting these candidates are tedious. To attack this problem, some studies focus on reducing the number of candidates.

- One dominant operation in the *Apriori* algorithm is support counting. To speed up the *Apriori*-like algorithms, some facilities are proposed.

A typical example for the effort of reducing the number of database scans can be found in [BMUT97]. In that study, Brin, et al. propose DIC, a dynamic itemset counting algorithm.

Intuitively, DIC works like a train running over the data with stops at intervals $M$ transactions apart. That is, the algorithm reads $M$ transactions at a time and update the appropriate support counts. When the train reaches the end of the transaction database, it has made one scan over the data and it starts over at the beginning for the next scan. The "*passengers*" on the train are candidate itemsets. If an itemset is on the train, its support is updated each time a transaction containing the itemset is scanned.

At the start of the first scan, the passengers on the train are the set of length-1 candidates. At each stop, DIC checks the passengers on the train according to the following rules.

1. When the support count of a candidate itemset $X$ passes the support threshold, we check whether $X$ can be joined with some other frequent itemsets with the same length to generate new candidates. If so, we add the new candidate on the train. We generate a length-$k$ itemset $X = a_1 \cdots a_k$ only when all $\begin{pmatrix} k \\ k-1 \end{pmatrix}$ length-$(k-1)$ subsets of $X$ have accumulated support greater than or equal to the support threshold. For example, when the counter corresponding to itemset *abc* passes the support threshold, we check whether *abc* can be joined with some other length-3 frequent itemsets. If *bcd*, *acd* and *abd* are already determined as frequent itemsets, then we can generate a length-4 candidate *abcd*.

2. When a candidate itemset $X$ has travelled for one complete scan, it is removed from the train. If, at that time, the support for $X$ is greater than or equal to the support threshold, output $X$ and its support as a frequent pattern.

For example, let us consider mining a transaction database $TDB$ with $40,000$ transactions and support threshold 100. Let the interval between stops be $10,000$. If itemset $a$ and $b$ get support counts greater than 100 in the first $10,000$ transactions, DIC will start counting 2-itemset *ab* after the first $10,000$ transactions. Similarly, if *ab*, *ac* and *bc* are contained in at least 100 transactions among the second $10,000$ transactions, DIC will start counting 3-itemset *abc* after $20,000$ transactions. Once DIC gets to the end of the transaction database $TDB$, it will stop counting the 1-itemsets and go back to the start of the database and count the 2 and 3-itemsets. After the first $10,000$ transactions, DIC will finish counting *ab*, and after $20,000$ transactions, it will finish counting *abc*. By overlapping the counting of different lengths of itemsets, DIC can save some database scans.

On the other hand, DIC also explores efficient support counting. It optimizes the trie structure used in the *Apriori* algorithm for counting candidates. Frequent items in each candidate are sorted in support ascending order according to their popularity in the first $M$

transactions. Such an order can reduce the number of inner loops in counting. Reordering items incurs some overhead, but for some data, it may be beneficial overall.

Experimental results reported in [BMUT97] indicates that DIC is faster than *Apriori* when the support threshold is low.

Another interesting study on reducing the number of database scans is from Savasere, et al. [SON95]. The large transaction database is divided into multiple partitions such that each partition can be held in main memory. The whole database is scanned only twice. In the first scan, partitions are read into main memory one by one. Local frequent patterns are mined with respect to relative support threshold using the *Apriori* method. The second scan consolidates global frequent patterns. Each global frequent pattern must be frequent in at least one partition. Therefore, only those local frequent patterns should be counted and tested in the second scan.

The major challenges for this partitioning-based method are in two aspects. On one hand, partitioning the database is non-trivial when the database is biased. On the other hand, a low global support threshold may lead to a much lower local threshold and thus produce a huge number of local frequent patterns. As an extreme example, if the global (absolute) support threshold is 10 and the global database is divided into 10 partitions, then some partitions may have local threshold 1, which means every itemset in those partitions is a local frequent pattern. Counting a huge number of patterns in the second scan could be very costly.

In [Toi96], Toivonen proposes a frequent pattern mining method by sampling. Instead of mining the database directly, the sampling method first mine a sample of the database using the *Apriori* algorithm. The sample should be small enough to fit into main memory. Then, the whole database is scanned once to verify frequent itemsets found in the sample. Only those frequent patterns having no frequent super-pattern are checked. In some rare cases where the sampling method may not produce all frequent patterns, the missing patterns can be found in one more pass of the database. The performance study in [Toi96] shows that the sampling algorithm is faster than both *Apriori* and the partitioning method in [SON95]. In many cases, after mining the sample database, the sampling method needs only one scan to find all frequent patterns.

Park, et al. [PCY95] illustrate that frequent pattern mining can be sped up by reducing the number of candidates. DHP (for *d*irect *h*ashing and *p*runing), a hashing-based algorithm, is introduced. DHP is an *Apriori*-like algorithm, with improvement on candidate generation and counting. In the $k$-th scan, DHP counts not only length-$k$ candidates, but also buckets of length-($k+1$) potential candidates. For example, let $a$, $b$, $c$, $d$, and $e$ be items in a transaction database. In the first scan, $DHP$ counts the supports of the five length-1 candidates. At the same time, potential length-2 candidates, $ab$, $ac$, $\ldots$, $de$, are generated and grouped into buckets. Suppose $ab$, $ad$, and $ae$ are in the same bucket. Every transaction containing $ab$, $ad$, or $ae$ results in an increment of 1 to the support count of the bucket. After the first scan, if the support count of the bucket is below the support threshold, neither $ab$ nor $ad$ nor $ae$ should be a length-2 candidate even if $a$, $b$, $d$ and $e$ are frequent.

DHP is especially effective for the generation of a candidate set for large 2-itemsets. Explicitly, the number of candidate 2-itemsets generated by the proposed algorithm is, in orders of magnitude, smaller than that generated by previous methods.

## 2.4 ***TreeProjection*: Going Beyond *Apriori*-like Methods**

Recently, Agarwal, et al. propose *TreeProjection*, a frequent pattern mining algorithm not in the *Apriori* framework. *TreeProjection* mines frequent patterns by constructing a lexicographical tree and projecting a large database into a set of reduced, item-based sub-databases based on the frequent patterns mined so far. The general idea is shown in the following example.

**Example 2.2** For the same transaction database presented in Example 2.1, we construct the lexicographical tree according to the method described in [AAP00]. The resulting tree is shown in Figure 2.1, and the construction process is presented as follows.

By scanning the transaction database once, all frequent 1-itemsets are identified. As recommended in [AAP00], the frequency ascending order is chosen as the ordering of the items. So, the order is $p$-$m$-$b$-$a$-$c$-$f$. The top level of the lexicographical tree is constructed, i.e. the root and the nodes labelled by length-1 patterns. At this stage, the root node labelled "null" and the nodes which store frequent 1-itemsets are generated. All transactions in the database are projected to the root node, i.e., all infrequent items are removed.

Each node in the lexicographical tree contains two pieces of information: (i) the pattern that the node represents, and (ii) the set of items which by adding to the pattern in the current node may generate longer patterns. The latter piece of information is recorded as *active extensions* and *active items*.

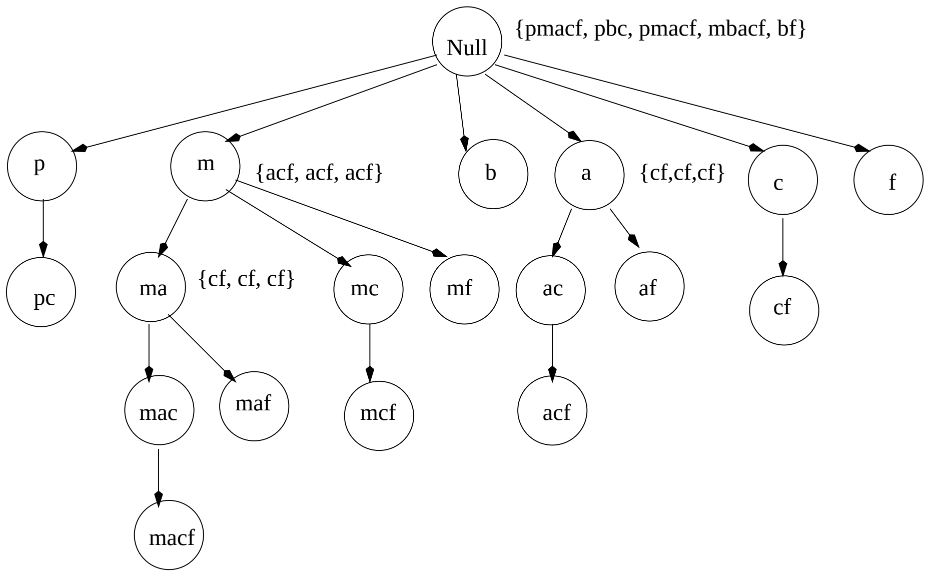

Figure 2.1: A lexicographical tree.

A matrix at the root node is created as shown below. The matrix computes the frequencies of length-2 patterns; thus, all pairs of frequent items are included in the matrix. The items are arranged in ascending order. The matrix is built by adding counts from transactions in the projected database, i.e., computing frequent 2-itemsets based on transactions stored in the root node.

$$\begin{array}{cccccc}
 & p & m & b & a & c & f \\
p & & & & & & \\
m & 2 & & & & & \\
b & 1 & 1 & & & & \\
a & 2 & 3 & 1 & & & \\
c & 3 & 3 & 2 & 3 & & \\
f & 2 & 3 & 2 & 3 & 3 &
\end{array}$$

At the same time as the matrix is built, transactions in the root are projected to level-1 nodes as follows. Let $t = a_1 a_2 \cdots a_n$ be a transaction with all items listed in ascending order. Transaction $t$ is projected to node $a_i$ $(1 \leq i < n-1)$ as $t'_{a_i} = a_{i+1} a_{i+2} \cdots a_n$.

From the matrix, frequent 2-itemsets are found to be: $\{pc, ma, mc, mf, ac, af, cf\}$. The nodes in the lexicographical tree for these frequent 2-itemsets are generated. At this stage, the active nodes for 1-itemsets are $m$ and $a$, because only these nodes contain enough descendants to potentially generate longer frequent itemsets. All other 1-itemset nodes are pruned.

In the same way, the lexicographical tree is grown level by level. From the matrix at node $m$, nodes labelled $mac$, $maf$, and $mcf$ are added, and only $ma$ is active for frequent 2-itemsets. It is easy to see that the lexicographical tree in total contains 19 nodes. ■

The number of nodes in a lexicographical tree is exactly that of the frequent itemsets. *TreeProjection* proposes an efficient way to enumerate frequent patterns. The efficiency of *TreeProjection* can be explained by two main factors:

- On one hand, the transaction projection limits support counting to a relatively small space, and only related portions of transactions are considered.
- On the other hand, the lexicographical tree facilitates the management and counting of candidates and provides the flexibility of picking an efficient strategy during the tree generation phase as well as transaction projection phase.

[AAP00] reports that their algorithm is up to one order of magnitude faster than other recent techniques in literature.

# Chapter 3

# *FP-growth*: A Pattern Growth Method

We presented the conventional frequent pattern mining method, *Apriori*, in Section 2.2. As analyzed, the major costs in *Apriori*-like methods are the generation of a huge number of candidates and the repeated scanning of large transaction databases to test those candidates. In short, *the candidate-generation-and-test operation is the bottleneck for* Apriori*-like methods.*

*Can we avoid candidate-generation-and-test in frequent pattern mining?* To attack this problem, we develop *FP-growth*, a pattern growth method for frequent pattern mining in this chapter. First, we develop an effective data structure, *FP-tree*, in Section 3.1. Then, in Section 3.2, we propose an efficient algorithm for mining frequent patterns from an *FP-tree*, and verify its correctness. In Section 3.3, we discuss how to scale the method to mine large databases which cannot be held in main memory. Experimental results and performance studies are reported in Section 3.4.

## 3.1 Frequent-Pattern Tree: Design and Construction

Information from transaction databases is essential for mining frequent patterns. Therefore, if we can extract the concise information for frequent pattern mining and store it into a compact structure, then it may facilitate frequent pattern mining. Motivated by this thinking, in this section, we develop a compact data structure, called *FP-tree*, to store complete but no redundant information for frequent pattern mining.

### 3.1.1 Frequent-Pattern Tree

To design a compact data structure for efficient frequent-pattern mining, let's first examine an example.

**Example 3.1** Let the transaction database, $TDB$, be the first two columns of Table 3.1 (same as the transaction database used in Example 2.1), and the minimum support threshold be 3 (i.e., $min\_sup = 3$).

| TID | Items Bought | (Ordered) Frequent Items |
|---|---|---|
| 100 | $f, a, c, d, g, i, m, p$ | $f, c, a, m, p$ |
| 200 | $a, b, c, f, l, m, o$ | $f, c, a, b, m$ |
| 300 | $b, f, h, j, o$ | $f, b$ |
| 400 | $b, c, k, s, p$ | $c, b, p$ |
| 500 | $a, f, c, e, l, p, m, n$ | $f, c, a, m, p$ |

Table 3.1: A transaction database.

A compact data structure can be designed based on the following observations:

1. Since only the frequent items will play a role in the frequent-pattern mining, it is necessary to perform one scan of transaction database $TDB$ to identify the set of frequent items (with *frequency count* obtained as a by-product).

2. If the *set* of frequent items of each transaction can be stored in some compact structure, it may be possible to avoid repeatedly scanning the original transaction database.

3. If multiple transactions share a set of frequent items, it may be possible to merge the shared sets with the number of occurrences registered as *count*. It is easy to check whether two sets are identical if the frequent items in all of the transactions are listed according to a fixed order.

4. If two transactions share a common prefix, according to some sorted order of frequent items, the shared parts can be merged using one prefix structure as long as the *count* is registered properly. If the frequent items are sorted in their *frequency descending order*, there are better chances that more prefix strings can be shared.

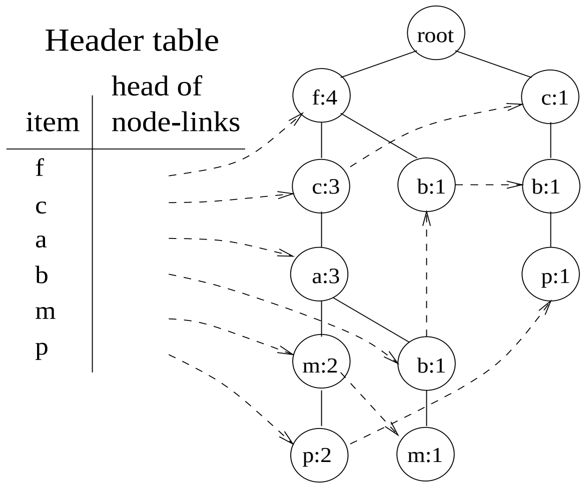

Figure 3.1: The *FP-tree* in Example 3.1.

With the above observations, one may construct a frequent-pattern tree as follows.

1. A scan of $TDB$ derives a *list* of frequent items, $\langle(f{:}4),(c{:}4),(a{:}3),(b{:}3),(m{:}3),(p{:}3)\rangle$ (the number after ":" indicates the support), in which items are ordered in frequency-descending order. (In the case that two or more items have exactly same support count, they are sorted alphabetically.) This ordering is important since each path of a tree will follow this order. For convenience of later discussions, the frequent items in each transaction are listed in this ordering in the rightmost column of Table 3.1.

2. Then, the root of a tree is created and labelled with "*null*". The *FP-tree* is constructed as follows by scanning the transaction database $TDB$ the second time.

   (a) The scan of the first transaction leads to the construction of the first branch of the tree: $\langle(f{:}1)$, $(c{:}1)$, $(a{:}1)$, $(m{:}1)$, $(p{:}1)\rangle$. Notice that the frequent items in the transaction are listed according to the order in the *list* of frequent items.

   (b) For the second transaction, since its (ordered) frequent item list $\langle f, c, a, b, m\rangle$ shares a common prefix $\langle f, c, a\rangle$ with the existing path $\langle f, c, a, m, p\rangle$, the count of each node along the prefix is incremented by 1, and one new node $(b{:}1)$ is created and linked as a child of $(a{:}2)$ and another new node $(m{:}1)$ is created and linked as the child of $(b{:}1)$.

(c) For the third transaction, since its frequent item list $\langle f, b \rangle$ shares only the node $\langle f \rangle$ with the $f$-prefix subtree, $f$'s count is incremented by 1, and a new node ($b$:1) is created and linked as a child of ($f$:3).

(d) The scan of the fourth transaction leads to the construction of the second branch of the tree, $\langle$($c$:1), ($b$:1), ($p$:1)$\rangle$.

(e) For the last transaction, since its frequent item list $\langle f, c, a, m, p \rangle$ is identical to the first one, the path is shared with the count of each node along the path incremented by 1.

To facilitate tree traversal, an item header table is built in which each item points to its first occurrence in the tree via a *node-link*. Nodes with the same item-name are linked in sequence via such *node-links*. After scanning all the transactions, the tree, together with the associated node-links, are shown in Figure 3.1. ■

Based on this example, a *frequent-pattern tree* can be designed as follows.

**Definition 3.1 (*FP-tree*)** A *frequent-pattern tree* (or FP-tree in short) is a tree structure defined below.

1. It consists of one root labeled as "*null*", a set of *item-prefix subtrees* as the children of the root, and a *frequent-item-header table*.

2. Each node in the *item-prefix subtree* consists of three fields: *item-name*, *count*, and *node-link*, where *item-name* registers which item this node represents, *count* registers the number of transactions represented by the portion of the path reaching this node, and *node-link* links to the next node in the *FP-tree* carrying the same item-name, or null if there is none.

3. Each entry in the *frequent-item-header table* consists of two fields, (1) *item-name* and (2) *head of node-link* (a pointer pointing to the first node in the *FP-tree* carrying the *item-name*). ■

Based on this definition, we have the following *FP-tree* construction algorithm.

**Algorithm 2 (*FP-tree* construction)**

**Input:** A transaction database $TDB$ and a minimum support threshold *min_sup*.

**Output:** *FP-tree*, the frequent-pattern tree of $TDB$.

**Method:** The *FP-tree* is constructed as follows.

1. Scan the transaction database $TDB$ once. Collect $F$, the set of frequent items, and the support of each frequent item. Sort $F$ in support-descending order as $FList$, the *list* of frequent items.

2. Create the root of an *FP-tree*, $T$, and label it as "null". For each transaction $t$ in $TDB$ do the following.

   Select the frequent items in transaction $t$ and sort them according to the order of $FList$. Let the sorted frequent-item list in $t$ be $[p|P]$, where $p$ is the first element and $P$ is the remaining list. Call $insert\_tree([p|P], T)$.

   The function $insert\_tree([p|P], T)$ is performed as follows. If $T$ has a child $N$ such that *N.item-name* = *p.item-name*, then increment $N$'s count by 1; else create a new node $N$, with count initialized to 1, parent link linked to $T$, and node-link linked to the nodes with the same *item-name* via the node-link structure. If $P$ is nonempty, call $insert\_tree(P, N)$ recursively.

**Analysis.** The *FP-tree* construction takes exactly two scans of the transaction database:

1. The first scan collects the set of frequent items; and

2. The second constructs the *FP-tree*.

The cost of inserting a transaction $t$ into the *FP-tree* is $\mathcal{O}(|freq(t)|)$, where $freq(t)$ is the set of frequent items in $t$. In next section, we will show that the *FP-tree* contains complete information for frequent-pattern mining. ■

### 3.1.2 Completeness and Compactness of *FP-tree*

Several important properties of *FP-tree* can be observed from the *FP-tree* construction process.

Given a transaction database $TDB$ and a support threshold *min_sup*. Let $F$ be the frequent items in $TDB$. For each transaction $t$, $freq(t)$ is the set of frequent items in $t$, i.e., $freq(t) = t \cap F$, and is called the *frequent item projection* of transaction $t$. According to *Apriori* principle, the set of frequent item projections of transactions in the database is sufficient for mining the complete set of frequent patterns, since the infrequent items play no role in frequent patterns.

**Lemma 3.1** *Given a transaction database $TDB$ and a support threshold min_sup, the support of every frequent itemset can be derived from $TDB$'s* FP-tree*.*

**Proof.** Based on the *FP-tree* construction process, for each transaction in the $TDB$, its frequent item projection is mapped to a path from the root in the *FP-tree*.

Given a frequent itemset $X = x_1 \cdots x_n$ in which items are sorted in the support descending order. Following the side-link of item $x_n$, we can visit all the nodes with label $x_n$ in the tree.

For each path $p$ from the root to a node $v$ with label $x_n$, the support count $sup_v$ in node $v$ is the number of transactions represented by $p$. If $x_1, \ldots, x_n$ all appear in $p$, then the $sup_v$ transactions represented by $p$ contain $X$. Thus, we accumulate such support counts. The sum is the support of $X$. ■

Based on this lemma, after an *FP-tree* for $TDB$ is constructed, it contains the complete information for mining frequent patterns from the transaction database. Thereafter, only the *FP-tree* is needed in the remaining of the mining process, regardless of the number and length of the frequent patterns.

**Lemma 3.2** *Given a transaction database $TDB$ and a support threshold min_sup, the number of nodes in an* FP-tree *is no more than $\sum_{t \in TDB} |freq(t)| + 1$. Further, the number of nodes in the longest path from the root is $\max_{t \in TDB}\{|freq(t)|\}$.*

**Proof.** Based on the *FP-tree* construction process, for any transaction $t$ in $TDB$, let $freq(t) = x_1 \cdots x_k$. There exists a path $root - x_1 - \cdots - x_k$ in the *FP-tree*. Each node in the tree, except for the root node, corresponds to at least one frequent item occurred in the transaction database. In the worst case, there is no overlap among frequent item projections of transactions, and thus all paths from the root to leaves share only the root node. Thus, the number of nodes in the tree is no more than $\sum_{t \in TDB} |freq(t)| + 1$. In the longest path from the root in the tree, there are $\max_{t \in TDB}\{|freq(t)|\}$ nodes. ■

Lemma 3.2 shows an important benefit of the *FP-tree*: the size of an *FP-tree* is bounded by the size of its corresponding database because each transaction will contribute at most one path to the *FP-tree*, with the length equal to the number of frequent items in that transaction. Since transactions often share frequent items, the size of the tree is usually much smaller than its original database. An *FP-tree* never breaks a transaction into pieces. Thus, unlike the *Apriori*-like method which may generate an exponential number of candidates in the worst case, under no circumstances, may an *FP-tree* with an exponential number of nodes be generated.

The *FP-tree* is a highly compact structure which stores the information for frequent-pattern mining. Since a single path "$a_1 \rightarrow a_2 \rightarrow \cdots \rightarrow a_n$" in the $a_1$-prefix subtree registers all the transactions whose maximal frequent set is in the form of "$a_1 \rightarrow a_2 \rightarrow \cdots \rightarrow a_k$" for any $1 \leq k \leq n$, the size of the *FP-tree* is often substantially smaller than the size of the database and that of the candidate sets generated in the association rule mining.

The items in the frequent item set are ordered in the support-descending order: More frequently occurring items are more likely to be shared and thus they are arranged to be closer to the top of the *FP-tree*. In general, this ordering provides a relatively compact *FP-tree* structure.

It is also feasible to construct an *FP-tree* using some other order, and all properties we have discussed before hold. Here, the support descending order is a heuristic to reduce the size of the tree. However, this does not mean that the tree so constructed *always* achieves the maximal compactness. With the knowledge of particular data characteristics, it is sometimes possible to achieve even better compression than the frequency-descending ordering. Consider the following example. Let the transactions be: $\{adef, bdef, cdef, a, a, a, b, b, b, c, c, c\}$, and the minimum support threshold be 3. The frequent item set associated with support count becomes $\{a{:}4, b{:}4, c{:}4, d{:}3, e{:}3, f{:}3\}$. Following the item frequency ordering $a \rightarrow b \rightarrow c \rightarrow d \rightarrow e \rightarrow f$, the *FP-tree* constructed will contain 12 nodes, as shown in Figure 3.2 (a). However, following another item ordering $f \rightarrow d \rightarrow e \rightarrow a \rightarrow b \rightarrow c$, it will contain only 9 nodes, as shown in Figure 3.2 (b).

The compactness of *FP-tree* is also verified by our experiments. Sometimes a rather small *FP-tree* results from a quite large database. For example, for the database *Connect-4* used in *MaxMiner* [Bay98], which contains 67,557 transactions with 43 items in each transaction, when the support threshold is 50% (which is used in the *MaxMiner* experiments [Bay98]), the total number of occurrences of frequent items is 2,219,609, whereas the total number of nodes in the *FP-tree* is 13,449 which represents a reduction ratio of 165.04, while it still holds hundreds of thousands of frequent patterns! (Notice that for databases with mostly short transactions, the reduction ratio is not that high.) Therefore, it is not surprising some gigabyte transaction database containing many long patterns may even generate an *FP-tree*

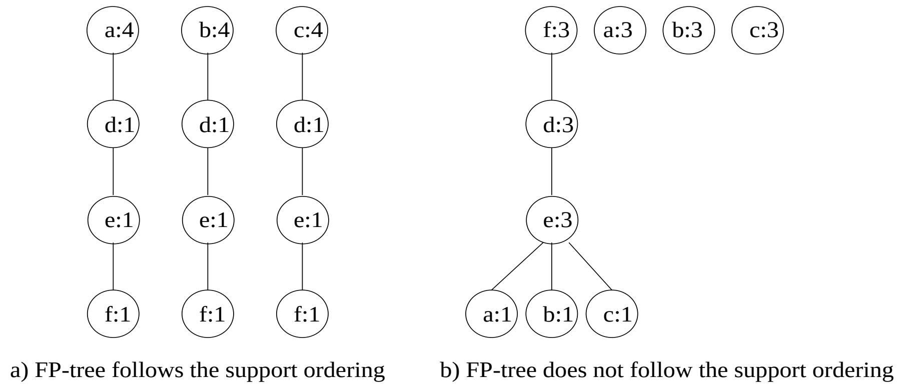
a:4
b:4
c:4
d:1
e:1
f:1
f:3
a:3
b:3
c:3
d:3
e:3
a:1
b:1
c:1
a) FP-tree follows the support ordering
b) FP-tree does not follow the support ordering

Figure 3.2: *FP-tree* constructed based on frequency descending ordering may not always be minimal.

which fits in main memory.

## 3.2 Mining Frequent Patterns Using *FP-tree*

Construction of a compact *FP-tree* ensures that subsequent mining can be performed with a rather compact data structure. However, this does not automatically guarantee that it will be highly efficient since one may still encounter the combinatorial problem of candidate generation if we simply use this *FP-tree* to generate and check all the candidate patterns.

In this section, we will study how to explore the compact information stored in an *FP-tree*, develop the principles of frequent-pattern growth by examination of our running example, explore how to perform further optimization when there exists a single prefix path in an *FP-tree*, and propose a frequent-pattern growth algorithm, *FP-growth*, for mining the *complete set of frequent patterns* using *FP-tree*.

### 3.2.1 Principles of Frequent-pattern Growth for *FP-tree* Mining

In this subsection, we examine some interesting properties of the *FP-tree* structure which will facilitate frequent-pattern mining.

**Property 3.2.1 (Node-link property)** *For any frequent item $a_i$, all the possible patterns containing only frequent items and $a_i$ can be obtained by following $a_i$'s node-links, starting from $a_i$'s head in the* FP-tree *header.* ■

This property is directly from the *FP-tree* construction process, and it facilitates the access of all the frequent-pattern information related to $a_i$ by traversing the *FP-tree* once following $a_i$'s node-links.

To facilitate the understanding of other properties of *FP-tree* related to mining, we first go through an example which performs mining on the constructed *FP-tree* (Figure 3.1) in Example 3.1.

**Example 3.2** Let us examine the mining process based on the constructed *FP-tree* shown in Figure 3.1.

According to the list of frequent items, $f$-$c$-$a$-$b$-$m$-$p$, all frequent patterns in the database can be divided into 6 subsets without overlap:

1. patterns containing item $p$;
2. patterns containing item $m$ but no item $p$;
3. patterns containing item $b$ but no $m$ nor $p$;
4. patterns containing item $a$ but no $b$, $m$ nor $p$;
5. patterns containing item $c$ but no $a$, $b$, $m$ nor $p$; and
6. patterns containing item $f$ but no $c$, $a$, $b$, $m$ nor $p$.

Let us mine these subsets one by one.

1. We first mine patterns having item $p$. An immediate frequent pattern in this subset is ($p$:3).

   To find other patterns having item $p$, we need to access all frequent item projections containing item $p$. Based on Property 3.2.1, all such projections can be collected by starting at $p$'s node-link head and following its node-links.

   Following $p$'s node-links, we can find that $p$ has two paths in the *FP-tree*: $\langle f{:}4, c{:}3, a{:}3, m{:}2, p{:}2\rangle$ and $\langle c{:}1, b{:}1, p{:}1\rangle$. The first path indicates that string "$(f, c, a, m, p)$" appears twice in the database. Notice the path also indicates that string $\langle f, c, a\rangle$ appears three times and $\langle f\rangle$ itself appears even four times. However, they only appear twice *together* with $p$. Thus, to study which string appear together with $p$, only $p$'s prefix path $\langle f{:}2, c{:}2, a{:}2, m{:}2\rangle$ (or simply, $\langle fcam{:}2\rangle$) counts. Similarly, the second path indicates

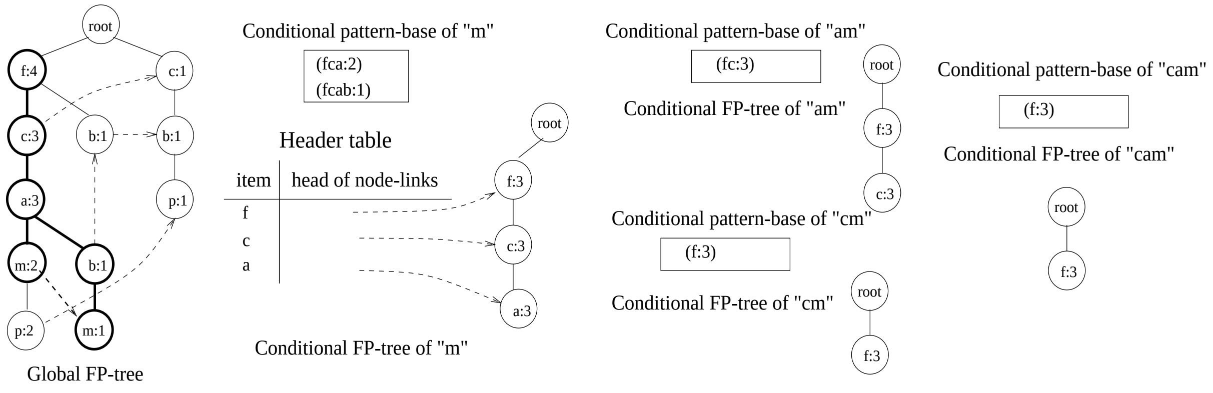
root
f:4
c:1
c:3
b:1
b:1
a:3
p:1
m:2
b:1
p:2
m:1
Global FP-tree
Conditional pattern-base of "m"
(fca:2)
(fcab:1)
Header table
item
head of node-links
f
c
a
root
f:3
c:3
a:3
Conditional FP-tree of "m"
Conditional pattern-base of "am"
(fc:3)
Conditional FP-tree of "am"
root
f:3
c:3
Conditional pattern-base of "cm"
(f:3)
Conditional FP-tree of "cm"
root
f:3
Conditional pattern-base of "cam"
(f:3)
Conditional FP-tree of "cam"
root
f:3

Figure 3.3: Mining FP-tree$|$ $m$, a conditional *FP-tree* for item $m$

string "$(c, b, p)$" appears once in the set of transactions in $DB$, or $p$'s prefix path is $\langle cb{:}1\rangle$. These two prefix paths of $p$, "$\{(fcam{:}2), (cb{:}1)\}$", form $p$'s subpattern-base, which is called $p$'s *conditional pattern-base* (i.e., the subpattern-base under the condition of $p$'s existence). Construction of an *FP-tree* on this conditional pattern-base (which is called $p$'s *conditional* FP-tree) leads to only one branch $(c{:}3)$. Hence, only one frequent pattern $(cp{:}3)$ is derived. (Notice that a pattern is an itemset and is denoted by a string here.) The search for frequent patterns associated with $p$ terminates.

2. Now, let us turn to patterns having item $m$ but no item $p$. Immediately, we identify frequent pattern $(m{:}3)$. By following $m$'s node-links, two paths in *FP-tree*, $\langle f{:}4, c{:}3, a{:}3, m{:}2\rangle$ and $\langle f{:}4,\ c{:}3,\ a{:}3,\ b{:}1,\ m{:}1\rangle$ are found. Notice $p$ appears together with $m$ as well, however, there is no need to include $p$ here in the analysis since any frequent patterns involving $p$ has been analyzed in the previous examination of patterns having item $p$. Similar to the above analysis, $m$'s conditional pattern-base is $\{(fca{:}2), (fcab{:}1)\}$. Constructing an *FP-tree* on it, we derive $m$'s conditional *FP-tree*, $\langle f{:}3, c{:}3, a{:}3\rangle$, a single frequent pattern path, as shown in Figure 3.3. This conditional *FP-tree* is then mined recursively by calling $mine(\langle f{:}3, c{:}3, a{:}3\rangle|m)$.

   Figure 3.3 shows that "$mine(\langle f{:}3, c{:}3, a{:}3\rangle|m)$" involves mining three items $(a)$, $(c)$, $(f)$ in sequence. The first derives a frequent pattern $(am{:}3)$, a conditional pattern-base $\{(fc{:}3)\}$, and then a call "$mine(\langle f{:}3, c{:}3\rangle|am)$"; the second derives a frequent pattern $(cm{:}3)$, a conditional pattern-base $\{(f{:}3)\}$, and then a call "$mine(\langle f{:}3\rangle|cm)$"; and the third derives only a frequent pattern $(fm{:}3)$. Further recursive call of "$mine(\langle f{:}3, c{:}3\rangle|am)$" derives $(cam{:}3)$, $(fam{:}3)$, a conditional pattern-base $\{(f{:}3)\}$, and then a call

"$mine(\langle f{:}3\rangle|cam)$", which derives the longest pattern ($fcam$:3). Similarly, the call of "$mine(\langle f{:}3\rangle|cm)$", derives one pattern ($fcm$:3). Therefore, the whole set of frequent patterns involving $m$ is $\{$($m$:3), ($am$:3), ($cm$:3), ($fm$:3), ($cam$:3), ($fam$:3), ($fcam$:3), ($fcm$:3)$\}$. This indicates *a single path* FP-tree *can be mined by outputting all the combinations of the items in the path.*

3. Similarly, we can mine patterns containing item $b$ but no $m$ nor $p$. Node $b$ derives ($b$:3) and has three paths: $\langle f{:}4, c{:}3, a{:}3, b{:}1\rangle$, $\langle f{:}4, b{:}1\rangle$, and $\langle c{:}1, b{:}1\rangle$. Since $b$'s conditional pattern-base $\{$($fca$:1), ($f$:1), ($c$:1)$\}$ generates no frequent item, the mining for $b$ terminates.

4. For patterns having item $a$ but no $b$, $m$ nor $p$, node $a$ derives one frequent pattern $\{$($a$:3)$\}$ and one subpattern base $\{$($fc$:3)$\}$, a single-path conditional *FP-tree.* Thus, its set of frequent patterns can be generated by taking their combinations. Concatenating them with ($a$:3), we have $\{$($fa$:3), ($ca$:3), ($fca$:3)$\}$.

5. Now, it is the turn to mine patterns having item $c$ but no $a$, $b$, $m$ nor $p$. Node $c$ derives ($c$:4) and one subpattern-base $\{$($f$:3)$\}$, and the set of frequent patterns associated with ($c$:3) is $\{$($fc$:3)$\}$.

6. The last subset, i.e., pattern having item $f$ but no any other items, is $f$ itself and ($f$:4) should be output. No conditional pattern-base need to be constructed.

| Item | Conditional pattern-base | Conditional *FP-tree* |
|---|---|---|
| $p$ | $\{(fcam{:}2), (cb{:}1)\}$ | $\{(c{:}3)\}\|p$ |
| $m$ | $\{(fca{:}2), (fcab{:}1)\}$ | $\{(f{:}3,\ c{:}3,\ a{:}3)\}\|m$ |
| $b$ | $\{(fca{:}1), (f{:}1), (c{:}1)\}$ | $\emptyset$ |
| $a$ | $\{(fc{:}3)\}$ | $\{(f{:}3, c{:}3)\}\|a$ |
| $c$ | $\{(f{:}3)\}$ | $\{(f{:}3)\}\|c$ |
| $f$ | $\emptyset$ | $\emptyset$ |

Table 3.2: Mining frequent patterns by creating conditional (sub)pattern-bases

The conditional pattern-bases and the conditional *FP-tree*s generated are summarized in Table 3.2. ■

The correctness and completeness of the process in Example 3.2 should be justified. This is accomplished by first introducing a few important properties related to the mining process.

**Property 3.2.2 (Prefix path property)** *To calculate the frequent patterns with suffix $a_i$, only the* prefix sub-pathes *of nodes labelled $a_i$ in the* FP-tree *need to be accumulated, and the frequency count of every node in the prefix path should carry the same count as that in the corresponding node $a_i$ in the path.*

**Proof.** Let the nodes along the path $P$ be labelled as $a_1, \dots, a_n$ in such an order that $a_1$ is the root of the prefix subtree, $a_n$ is the leaf of the subtree in $P$, and $a_i$ $(1 \leq i \leq n)$ is the node being referenced. Based on the process of *FP-tree* construction presented in Algorithm 2, for each prefix node $a_k$ $(1 \leq k < i)$, the prefix sub-path of the node $a_i$ in $P$ occurs together with $a_k$ exactly $a_i.count$ times. Thus every such prefix node should carry the same count as node $a_i$. Notice that a postfix node $a_m$ (for $i < m \leq n$) along the same path also co-occurs with node $a_i$. However, the patterns with $a_m$ will be generated at the examination of the postfix node $a_m$, enclosing them here will lead to redundant generation of the patterns that would have been generated for $a_m$. Therefore, we only need to examine the prefix sub-path of $a_i$ in $P$. ■

For example, in Example 3.2, node $m$ is involved in a path $\langle f{:}4, c{:}3, a{:}3, m{:}2, p{:}2\rangle$, to calculate the frequent patterns for node $m$ in this path, only the prefix sub-path of node $m$, which is $\langle f{:}4, c{:}3, a{:}3\rangle$, need to be extracted, and the frequency count of every node in the prefix path should carry the same count as node $m$. That is, the node counts in the prefix path should be adjusted to $\langle f{:}2, c{:}2, a{:}2\rangle$.

Based on this property, the prefix sub-path of node $a_i$ in a path $P$ can be copied and transformed into a count-adjusted prefix sub-path by adjusting the frequency count of every node in the prefix sub-path to the same as the count of node $a_i$. The so transformed prefix path is called the *transformed prefix path* of $a_i$ for path $P$.

Notice that the set of transformed prefix paths of $a_i$ form a small database of patterns which co-occur with $a_i$. Such a database of patterns occurring with $a_i$ is called *$a_i$'s conditional pattern-base*, and is denoted as "*pattern_base* | $a_i$". Then one can compute all the frequent patterns associated with $a_i$ in this $a_i$-conditional pattern-base by creating a small *FP-tree*, called *$a_i$'s conditional* FP-tree and denoted as "*FP-tree*| $a_i$". Subsequent mining can be performed on this small conditional *FP-tree.* The process of constructing conditional pattern-bases and conditional *FP-tree*s has been demonstrated in Example 3.2.

This process is performed recursively, and the frequent patterns can be obtained by a pattern-growth method, based on the following lemmas and corollary.

**Lemma 3.3 (Fragment growth)** *Let $\alpha$ be an itemset in $DB$, $B$ be $\alpha$'s conditional pattern-base, and $\beta$ be an itemset in $B$. Then the support of $\alpha \cup \beta$ in $DB$ is equivalent to the support of $\beta$ in $B$.*

**Proof.** According to the definition of conditional pattern-base, each (sub)transaction in $B$ occurs under the condition of the occurrence of $\alpha$ in the original transaction database $DB$. If an itemset $\beta$ appears in $B$ $\psi$ times, it appears with $\alpha$ in $DB$ $\psi$ times as well. Moreover, since all such items are collected in the conditional pattern-base of $\alpha$, $\alpha \cup \beta$ occurs exactly $\psi$ times in $DB$ as well. Thus we have the lemma. ∎

From this lemma, we can directly derive an important corollary.

**Corollary 3.2.1 (Pattern growth)** *Let $\alpha$ be a frequent itemset in $DB$, $B$ be $\alpha$'s conditional pattern-base, and $\beta$ be an itemset in $B$. Then $\alpha \cup \beta$ is frequent in $DB$ if and only if $\beta$ is frequent in $B$.*

**Proof.** This corollary is the case when $\alpha$ is a frequent itemset in $DB$, and when the support of $\beta$ in $\alpha$'s conditional pattern-base $B$ is no less than $\xi$, the minimum support threshold. We first prove the "if" part. Suppose $\beta$ is frequent in $B$, that is, $\beta$ appears in $B$ at least $\xi$ times. Since $B$ is $\alpha$'s conditional pattern-base, each transaction in $B$ appears under the existence of $\alpha$. That is, $\beta$ appears together with $\alpha$ in $DB$ at least $\xi$ times. Therefore, $\alpha \cup \beta$ is frequent in $DB$.

Then we prove the "only if" part. Suppose $\beta$ is not frequent in $B$, that is, $\beta$ appears in $B$ less than $\xi$ times. Since $B$ is $\alpha$'s conditional pattern-base, all the itemsets containing $\beta$ and co-occurring with $\alpha$ are in $B$. Thus $\beta$ co-occurs with $\alpha$ less than $\xi$ times. Therefore, $\alpha \cup \beta$ is not frequent in $DB$. ∎

Based on Corollary 3.2.1, mining can be performed by first identifying the set of frequent 1-itemsets in $DB$, and then for each such frequent 1-itemset, constructing its conditional pattern-bases, and mining its set of frequent 1-itemsets in the conditional pattern-base, and so on. This indicates that the process of mining frequent patterns can be viewed as first mining frequent 1-itemset and then progressively growing each such itemset by mining its conditional pattern-base, which can in turn be done similarly. By doing so, a frequent $k$-itemset mining problem is successfully transformed into a sequence of $k$ frequent 1-itemset mining problems via a set of conditional pattern-bases. Since mining is done by pattern growth, there is no need to generate any candidate sets in the entire mining process.

Notice also in the construction of a new *FP-tree* from a conditional pattern-base obtained during the mining of an *FP-tree*, the items in the frequent itemset should be ordered in the frequency descending order of node occurrence of each item instead of its support (which represents item occurrence). This is because each node in an *FP-tree* may represent many occurrences of an item but such a node represents a single unit (i.e., the itemset whose elements always occur together) in the construction of an item-associated *FP-tree*.

### 3.2.2 Frequent-pattern Growth With Single Prefix Path of *FP-tree*

The frequent-pattern growth method described above works for all kinds of *FP-tree*s. However, further optimization can be explored on a special kind of *FP-tree*, called *single prefix-path* FP-tree, and such an optimization is especially useful at mining long frequent patterns.

A *single prefix-path FP-tree* is an *FP-tree* that consists of only a single path or a single prefix path stretching from the root to the first branching node of the tree, where a *branching node* is a node containing more than one child.

Let us examine an example.

**Example 3.3** Figure 3.4(a) is a single prefix-path *FP-tree* that consists of one prefix path, $\langle(a{:}10)\rightarrow(b{:}8)\rightarrow(c{:}7)\rangle$, stretching from the root of the tree to the first branching node $(c{:}7)$. Although it can be mined using the frequent-pattern growth method described above, a better method is to split the tree into two fragments: the single prefix-path, $\langle(a{:}10)\rightarrow(b{:}8)\rightarrow(c{:}7)\rangle$, as shown in Figure 3.4(b), and the multiple-path part, with the root replaced by a pseudo-root $R$, as shown in Figure 3.4(c). These two parts can be mined separately and then combined together.

Let us examine the two separate mining processes. All the frequent patterns associated with the first part, the single prefix-path $P = \langle(a{:}10)\rightarrow(b{:}8)\rightarrow(c{:}7)\rangle$, can be mined by enumeration of all the combinations of the sub-pathes of $P$ with the support set to the minimum support of the items contained in the sub-path. This is because each such sub-path is distinct and occurs the same number of times as the *minimum occurrence frequency among the items in the sub-path* which is equal to the support of the last item in the sub-path. Thus, path $P$ generates the following set of frequent patterns, $freq\_pattern\_set(P)$ = $\{(a{:}10), (b{:}8), (c{:}7), (ab{:}8), (ac{:}7), (bc{:}7), (abc{:}7)\}$.

Let $Q$ be the second *FP-tree* (Figure 3.4(c)), the multiple-path part rooted with $R$. $Q$ can be mined as follows.

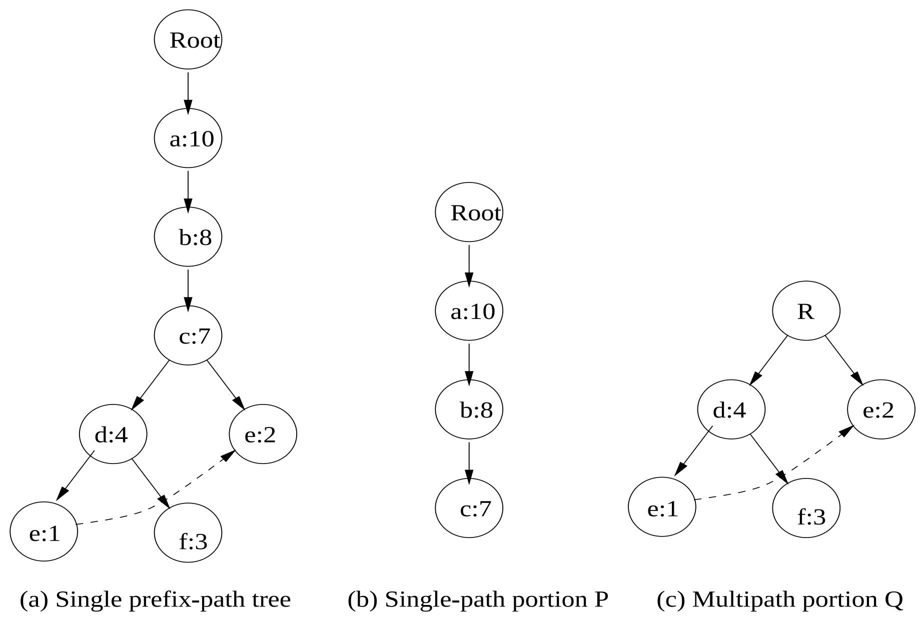

Figure 3.4: Mining an *FP-tree* with a single prefix path.

First, $R$ is treated as a *null* root, and $Q$ forms a multiple-path *FP-tree*, which can be mined using a typical frequent-pattern growth method. The mining result is:

$$freq\_pattern\_set(Q) = \{(d:4), (e:3), (f:3), (df:3)\}$$

Second, for each frequent itemset in $Q$, $R$ can be viewed as a conditional frequent pattern-base, and each itemset in $Q$ with each pattern generated from $R$ may form a distinct frequent pattern. For example, for ($d$:4) in $freq\_pattern\_set(Q)$, $P$ can be viewed as its conditional pattern-base, and a pattern generated from $P$, such as ($a$:10), will generate with it a new frequent itemset, ($ad$:4), since $a$ appears together with $d$ at most four times. Thus, for ($d$:4) the set of frequent patterns generated will be ($d$:4) $\times$ $freq\_pattern\_set(P)$ = {($ad$:4), ($bd$:4), ($cd$:4), ($abd$:4), ($acd$:4), ($bcd$:4), ($abcd$:4)}, where $X \times Y$ means that every pattern in $X$ is combined with every one in $Y$ to form a "cross-product-like" larger itemset with the support being the minimum support between the two patterns. Thus, the complete set of frequent patterns generated by combining the results of $P$ and $Q$ will be $freq\_pattern\_set(Q) \times freq\_pattern\_set(P)$, with the support being the support of the itemset in $Q$ (which is always no more than the support of the itemset from $P$).

In summary, the set of frequent patterns generated from such a single prefix path consists of three distinct sets: (1) $freq\_pattern\_set(P)$, the set of frequent patterns generated from the single prefix-path, $P$; (2) $freq\_pattern\_set(Q)$, the set of frequent patterns generated from the multiple-path part of the *FP-tree*, $Q$; and (3) $freq\_pattern\_set(Q) \times freq\_pattern\_set(P)$, the set of frequent patterns involving both parts. ■

We first show if an *FP-tree* consists of a single path $P$, one can generate the set of frequent patterns according to the following lemma.

**Lemma 3.4 (Pattern generation for an *FP-tree* consisting of single path)** *Suppose that an* FP-tree $T$ *consists of a single path* $P = \langle root \rightarrow a_1{:}s_1 \rightarrow a_2{:}s_2 \rightarrow \cdots \rightarrow a_k{:}s_k \rangle$. *Then, an itemset* $X = a_{i_1} \cdots a_{i_j}$ $(1 \leq i_1 < \cdots < i_j \leq k)$ *is a frequent pattern and* $sup(X)$ *equals to the support count registered in node* $a_{i_j}$ *in the tree* $T$.

**Proof.** According to the construction of the *FP-tree*, $a_1, \ldots, a_k$ are frequent items. Since there is only one path in the tree, every transaction having $a_k$ must also have $a_1, \ldots, a_{k-1}$. That is, $sup(a_1 \cdots a_k) = sup(a_k) = s_k \geq min\_sup$, where $min\_sup$ is the support threshold. By *Apriori* property, we have that an itemset $X$ as stated in the lemma is a frequent pattern.

For an itemset $X$ as stated in the lemma, since the tree has only one path, every transaction containing $X$ must correspond to a sub-path from the root in the tree to some node $a_l$ such that $l \geq i_j$, and thus increases the support count in node $a_{i_j}$ by 1 in the tree construction. That means the support count registered in node $a_{i_j}$ is no less than $sup(X)$. On the other hand, by *Apriori* property, the support of $a_{i_j}$ cannot be larger than that of $X$. Thus, we have $sup(X)$ is exactly the support count registered in $a_{i_j}$. ■

We then show if an *FP-tree* consists of a single prefix-path, the set of frequent patterns can be generated by splitting the tree into two according to the following lemma.

**Lemma 3.5 (Pattern generation for an *FP-tree* consisting of single prefix path)** *Suppose an* FP-tree $T$*, similar to the tree in Figure 3.4(a), consist of (1) a single prefix path* $P$*, similar to the tree* $P$ *in Figure 3.4(b), and (2) the multi-path part,* $Q$*, which can be viewed as an independent* FP-tree *with a pseudo-root* $R$*, similar to the tree* $Q$ *in Figure 3.4(c).*

*The complete set of the frequent patterns of* $T$ *consists of the following three portions:*

1. *The set of frequent patterns generated from* $P$ *by enumeration of all the combinations of the items along path* $P$*, with the support being the minimum support among all the items that the pattern contains.*

2. *The set of frequent patterns generated from $Q$ by taking root $R$ as "null."*

3. *The set of frequent patterns combining $P$ and $Q$ formed by taken the cross-product of the frequent patterns generated from $P$ and $Q$, denoted as $freq\_pattern\_set(P) \times freq\_pattern\_set(Q)$, that is, each frequent itemset is the union of one frequent itemset from $P$ and one from $Q$ and its support is the minimum one between the supports of the two itemsets.*

**Proof.** Based on the *FP-tree* construction rules, each node $a_i$ in the single prefix path of the *FP-tree* appears only once in the tree. The single prefix-path of the *FP-tree* forms a new *FP-tree* $P$, and the multiple-path part forms another *FP-tree* $Q$. They do not share nodes representing the same item. Thus, the two *FP-tree*s can be mined separately.

First, we show that each pattern generated from one of the three portions by following the pattern generation rules is distinct and frequent. According to Lemma 3.4, each pattern generated from $P$, the *FP-tree* formed by the single prefix-path, is distinct and frequent. The set of frequent patterns generated from $Q$ by taking root $R$ as "null" is also distinct and frequent since such patterns exist without combining any items in their conditional databases (which are in the items in $P$. The set of frequent patterns generated by combining $P$ and $Q$, that is, taking the cross-product of the frequent patterns generated from $P$ and $Q$, with the support being the minimum one between the supports of the two itemsets, is also distinct and frequent. This is because each frequent pattern generated by $P$ can be considered as a frequent pattern in the conditional pattern-base of a frequent item in $Q$, and whose support should be the minimum one between the two supports since this is the frequency that both patterns appear together.

Second, we show that no patterns can be generated out of this three portions. Since according to Lemma 3.3, the *FP-tree* $T$ without being split into two *FP-tree*s $P$ and $Q$ generates the complete set of frequent patterns by pattern growth. Since each pattern generated from $T$ will be generated from either the portion $P$ or $Q$ or their combination, the method generates the complete set of frequent patterns. ■

### 3.2.3 The Frequent-pattern Growth Algorithm

Based on the above lemmas and properties, we have the following algorithm for mining frequent patterns using *FP-tree*.

**Algorithm 3 (*FP-growth*: Mining frequent patterns with *FP-tree* by pattern fragment growth)**

**Input:** A database $DB$, represented by *FP-tree* constructed according to Algorithm 2, and a minimum support threshold $\xi$.

**Output:** The complete set of frequent patterns.

**Method: call** *FP-growth*(*FP-tree*, *null*).

```
Procedure FP-growth($Tree, \alpha$)
{
(1)  if $Tree$ contains a single prefix path        // Mining single prefix-path FP-tree
(2)  then {
(3)      let $P$ be the single prefix-path part of $Tree$;
(4)      let $Q$ be the multiple-path part with the top branching node replaced by a null root;
(5)      for each combination (denoted as $\beta$) of the nodes in the path $P$ do
(6)          generate pattern $\beta \cup \alpha$ with $support = minimum\ support\ of\ nodes\ in\ \beta$;
(7)      let $freq\_pattern\_set(P)$ be the set of patterns so generated;    }
(8)  else let $Q$ be $Tree$;
(9)  for each item $a_i$ in $Q$ do {            // Mining multiple-path FP-tree
(10)     generate pattern $\beta = a_i \cup \alpha$ with $support = a_i.support$;
(11)     construct $\beta$'s conditional pattern-base and then $\beta$'s conditional FP-tree $Tree_\beta$;
(12)     if $Tree_\beta \neq \emptyset$
(13)     then call FP-growth($Tree_\beta, \beta$);
(14)     let $freq\_pattern\_set(Q)$ be the set of patterns so generated;    }
(15) return($freq\_pattern\_set(P) \cup freq\_pattern\_set(Q) \cup (freq\_pattern\_set(P)$
             $\times\ freq\_pattern\_set(Q)$))
}
```

**Analysis.** With the properties and lemmas in Sections 2 and 3, we show that the algorithm correctly finds the complete set of frequent itemsets in transaction database $DB$.

As shown in Lemma 3.1, *FP-tree* of $DB$ contains the complete information of $DB$ in relevance to frequent pattern mining under the support threshold $\xi$.

If an *FP-tree* contains a single prefix-path, according to Lemma 3.5, the generation of the complete set of frequent patterns can be partitioned into three portions: the single prefix-path portion $P$, the multiple-path portion $Q$, and their combinations. Hence we have lines

(1)–(4) and line (15) of the procedure. According to Lemma 3.4, the generated patterns for the single prefix path are the enumerations of the sub-paths of the prefix path, with the support being the minimum support of the nodes in the sub-path. Thus we have lines (5)–(7) of the procedure. After that, one can treat the multiple-path portion or the *FP-tree* that does not contain the single prefix-path as portion $Q$ (lines (4) and (8)) and construct conditional pattern-base and mine its conditional *FP-tree* for each frequent itemset $a_i$. The correctness and completeness of the prefix path transformation are shown in Property 3.2.2. Thus the conditional pattern-bases store the complete information for frequent pattern mining for $Q$. According to Lemmas 3.3 and its corollary, the patterns successively grown from the conditional *FP-tree*s are the set of sound and complete frequent patterns. Especially, according to the fragment growth property, the support of the combined fragments takes the support of the frequent itemsets generated in the conditional pattern-base. Therefore, we have lines (9)–(14) of the procedure. Line (15) sums up the complete result according to Lemma 3.5. ■

Let's now examine the efficiency of the algorithm. The *FP-growth* mining process scans the *FP-tree* of $DB$ once and generates a small pattern-base $B_{a_i}$ for each frequent item $a_i$, each consisting of the set of transformed prefix paths of $a_i$. Frequent pattern mining is then recursively performed on the small pattern-base $B_{a_i}$ by constructing a conditional *FP-tree* for $B_{a_i}$. As reasoned in the analysis of Algorithm 2, an *FP-tree* is usually much smaller than the size of $DB$. Similarly, since the conditional *FP-tree*, "*FP-tree*$|\ a_i$", is constructed on the pattern-base $B_{a_i}$, it should be usually much smaller and never bigger than $B_{a_i}$. Moreover, a pattern-base $B_{a_i}$ is usually much smaller than its original *FP-tree*, because it consists of the transformed prefix paths related to only one of the frequent items, $a_i$. Thus, each subsequent mining process works on a set of usually much smaller pattern-bases and conditional *FP-trees*. Moreover, the mining operations consist of mainly prefix count adjustment, counting local frequent items, and pattern fragment concatenation. This is much less costly than generation and test of a very large number of candidate patterns. Thus the algorithm is efficient.

From the algorithm and its reasoning, one can see that the *FP-growth* mining process is a divide-and-conquer process, and the scale of shrinking is usually quite dramatic. If the shrinking factor is around 20$\sim$100 for constructing an *FP-tree* from a database, it is expected to be another hundreds of times reduction for constructing each conditional *FP-tree* from its already quite small conditional frequent pattern-base.

Notice that even in the case that a database may generate an large number of frequent patterns, the size of the *FP-tree* is usually quite small. For example, for a frequent pattern of length 100, "$a_1 \cdots a_{100}$", the *FP-tree* construction results in only one path of length 100 for it, such as "$\langle a_1 \rightarrow \cdots \rightarrow a_{100} \rangle$". The *FP-growth* algorithm will still generate about $10^{30}$ frequent patterns (if time permits!!), such as "$a_1$, $a_2$, ..., $a_1a_2$, ..., $a_1a_2a_3$, ..., $a_1 \ldots a_{100}$". However, the *FP-tree* contains only one frequent pattern path of 100 nodes, and according to Lemma 3.4, there is no need to construct any conditional *FP-tree* in order to find all the patterns. Moreover, with *FP-tree*, we can derive some condensed expression of frequent patterns. In the case that we have only a long pattern $a_1 \cdots a_{100}$, we can only output the long pattern itself and omit all proper sub-patterns.

## 3.3 Scaling *FP-tree*-Based *FP-growth* by Database Projection

*FP-growth* proposed in the last section is essentially a main memory-based frequent pattern mining method. However, when the database is large, or when the minimum support threshold is quite low, it is unrealistic to assume that the *FP-tree* of a database can fit in main memory. A disk-based method should be worked out to ensure that mining is highly scalable. In this section, we develop a method to first partition the database into a set of projected databases, and then for each projected database, construct and mine its corresponding *FP-tree*.

Let us revisit the mining problem in Example 3.1.

**Example 3.4** Suppose the *FP-tree* in Figure 3.1 cannot be held in main memory. Instead of constructing a global *FP-tree*, one can project the transaction database into a set of frequent item-related *projected databases* as follows.

Starting at the tail of the frequent item list, $p$, the set of transactions that contain item $p$ can be collected into *p-projected database*. Infrequent items and item $p$ itself can be removed from them because the infrequent items are not useful in frequent pattern mining, and item $p$ is by default associated with each projected transaction. Thus, the $p$-projected database becomes $\{fcam$, $cb$, $fcam\}$. This is very similar to the the $p$-conditional pattern-base shown in Table 3.2 except $fcam$ and $fcam$ are expressed as ($fcamp$:2) in Table 3.2. After that, the $p$-conditional *FP-tree* can be built on the $p$-projected database based on the *FP-tree* construction algorithm.

Similarly, the set of transactions containing item $m$ can be projected into *$m$-projected database*. Notice that besides infrequent items and item $m$, item $p$ is also excluded from the set of projected items because item $p$ and its association with $m$ have been considered in the $p$-projected database. For the same reason, the *$b$-projected database* is formed by collecting transactions containing item $b$, but infrequent items and items $f$, $m$ and $b$ are excluded. This process continues for deriving *$a$-projected database*, *$c$-projected database*, and so on. The complete set of item-projected databases derived from the transaction database are listed in Table 3.3, together with their corresponding conditional *FP-tree*s. One can easily see that the two processes, *constructing the global* FP-tree *and forming conditional* FP-tree*s*, and *projecting the database into a set of projected databases and constructing conditional* FP-tree*s*, yield identical conditional *FP-tree*s.

| Item | Projected database | Conditional *FP-tree* |
|---|---|---|
| $p$ | $\{fcam, cb, fcam\}$ | $\{(c{:}3)\}\|p$ |
| $m$ | $\{fca, fcab, fca\}$ | $\{(f{:}3,\ c{:}3,\ a{:}3)\}\|m$ |
| $b$ | $\{fca, f, c\}$ | $\emptyset$ |
| $a$ | $\{fc, fc, fc\}$ | $\{(f{:}3, c{:}3)\}\|a$ |
| $c$ | $\{f, f, f\}$ | $\{(f{:}3)\}\|c$ |
| $f$ | $\emptyset$ | $\emptyset$ |

Table 3.3: Projected databases and their *FP-tree*s

As shown in Section 2, a conditional *FP-tree* is usually orders of magnitude smaller than the global *FP-tree*. Thus, construction of a conditional *FP-tree* from each projected database and then mining on it will dramatically reduce the size of *FP-tree*s to be handled. What about that a conditional *FP-tree* of a projected database still cannot fit in main memory? One can further project the projected database, and the process can go on recursively until the conditional *FP-tree* fits in main memory. ■

Let us define the concept of projected database formally.

**Definition 3.2 (Projected database)**

Let $a_i$ be a frequent item in a transaction database, $DB$. The *$a_i$-projected database* for $a_i$, denoted as $DB|_{a_i}$, can be derived as follows. First, all transactions containing $a_i$ are selected. Then, remove any infrequent items from any selected transaction. Delete from each selected transaction $a_i$ and any other items which follow $a_i$ in the list of frequent items.

■

The $a_i$-projected database is derived by projecting the set of items in the transactions containing $a_i$ into the projected database. Alternatively, it can be also achieved by collecting the same set of items from the $a_i$-subtree in the *FP-tree.* Thus, the two methods derive the same sets of conditional *FP-tree*s.

There are two methods for database projection: *parallel projection* and *partition projection.*

*Parallel projection* is implemented as follows: Scan the database to be projected once, where the database could be either a transaction database or an $\alpha$-projected database. For each transaction $T$ in the database, for each frequent item $a_i$ in $T$, project $T$ to the $a_i$-projected database based on the transaction projection rule, specified in the definition of projected database. Since a transaction is projected in parallel to all the projected databases in one scan, it is called *parallel projection.* The set of projected databases shown in Table 3.3 of Example 3.4 demonstrates the result of parallel projection. This process is illustrated in Figure 3.5 (a).

Parallel projection facilitates parallel processing because all the projected databases are available for mining at the end of the scan, and these projected databases can be mined in parallel. However, since each transaction in the database is projected to multiple projected databases, if a database contains many long transactions with multiple frequent items, the total size of the projected databases could be multiple times of the original one. Let each transaction contains on average $l$ frequent items. A transaction is then projected to $l-1$ projected database. The total size of the projected data from this transaction is $1+2+\cdots+(l-1)=\frac{l(l-1)}{2}$. This implies that the total size of the single item-projected databases is about $\frac{l-1}{2}$ times of that of the original database.

To avoid such an overhead, we propose a *partition projection* method. Partition projection is implemented as follows. A transaction $T$ is *projected* to the $a_i$-projected database only if $a_i$ is the last item appearing in $T$. A transaction is projected to only one projected database during the database scan. After the scan, the database is partitioned by projection into a set of projected databases, and hence it is called *partition projection.*

The projected databases are mined in the reverse order of the *list of frequent items.* That is, the projected database of the least frequent item is mined first, and so on. When

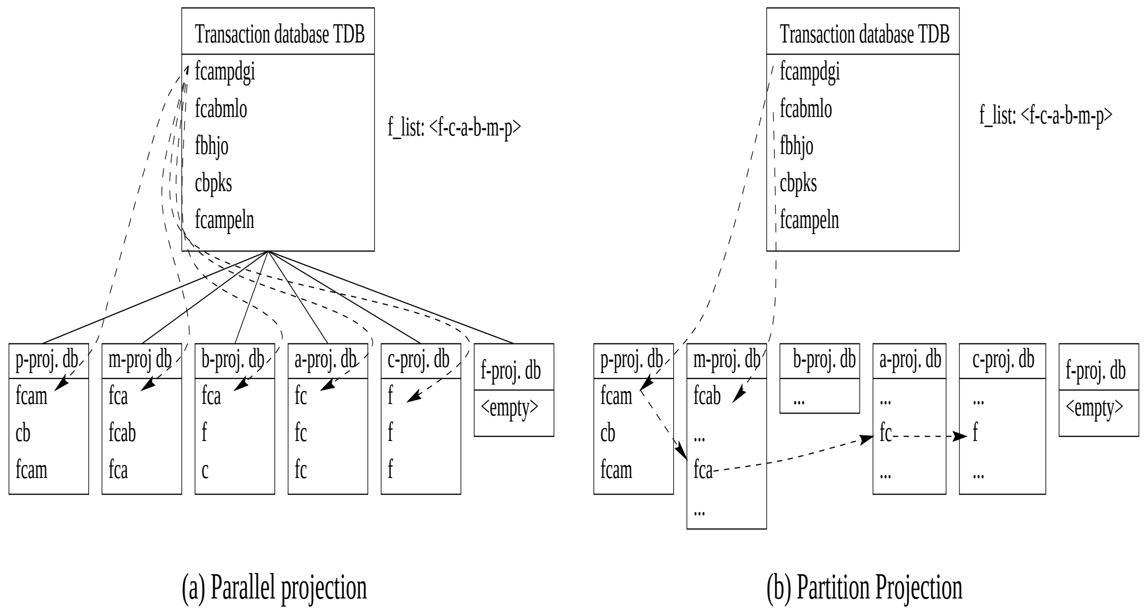

Figure 3.5: Parallel projection vs. partition projection.

a projected database $DB|_{a_k}$ is being mined, transactions in $DB|_{a_k}$ are further projected. Each transaction $t$ in $DB|_{a_k}$ is projected to the $a_j$-projected database $DB|_{a_k}|_{a_j}$ for the last frequent item $a_j$ in $t$. The partition projection process for the database in Example 3.4 is illustrated in Figure 3.5 (b).

The advantage of partition projection is that the total size of the projected databases at each level is smaller than the original database, and it usually takes less memory and I/Os to complete the partition projection. However, the processing order of the projected databases becomes important, and one has to process these projected databases in a sequential manner. Also, during the processing of each projected database, one needs to project the processed transactions to their corresponding projected databases, which may take some I/O as well. Nevertheless, due to its low memory requirement, partition projection is still a promising method in frequent pattern mining.

**Example 3.5** Let us examine how the database in Example 3.4 can be projected by partition projection.

First, by one scan of the transaction database, each transaction is projected to only one projected database. The first transaction, $facdgimp$, is projected to the *p-projected database* since $p$ is the last frequent item in the *list of frequent items*. Thus, $fcam$ (i.e., with infrequent items removed) is inserted into the $p$-projected database. Similarly, transaction $abcflmo$ is projected to the *$m$-projected database* as $fcab$, $bfhjo$ to the *$b$-projected database* as $f$, $bcksp$ to the $p$-projected database as $cb$, and finally, $afcelpmn$ to the $p$-projected database as $fcam$. After this phrase, the entries in every projected databases are shown in Table 3.4.

| item | Projected databases |
|---|---|
| $p$ | $\{fcam, cb, fcam\}$ |
| $m$ | $\{fcab\}$ |
| $b$ | $\{f\}$ |
| $a$ | $\emptyset$ |
| $c$ | $\emptyset$ |
| $f$ | $\emptyset$ |

Table 3.4: Single-item projected databases by partition projection.

With this projection, the original database can be replaced by the set of single-item projected databases, and the total size of them is smaller than that of the original database.

Second, the $p$-projected database is first processed (i.e., construction of $p$-conditional *FP-tree*), where $p$ is the last item in the *list of frequent items*. During the processing of the $p$-projected database, each transaction is projected to the corresponding projected database according to the same partition projection rule. For example, $fcam$ is projected to the $m$-projected database as $fca$, $cb$ is projected to the $b$-projected database as $c$, and so on. The process continues until every single-item projected database is completely processed. ■

## 3.4 Experimental Evaluation and Performance Study

In this section, we present a performance comparison of *FP-growth* with the classical frequent pattern mining algorithm *Apriori*, and a recently proposed database projection-based algorithm, *TreeProjection*.

### 3.4.1 Environments of Experiments

All the experiments are performed on a 266-MHz Pentium PC machine with 128 megabytes main memory, running on Microsoft Windows/NT. All the programs are written in Microsoft/Visual C++6.0. Notice that we do not directly compare our absolute number of runtime with those in some published reports running on the RISC workstations because different machine architectures may differ greatly on the absolute runtime for the same algorithms. Instead, we implement their algorithms to the best of our knowledge based on the published reports on the same machine and compare in the same running environment. Please also note that *run time* used here means the total execution time, that is, the period between input and output, instead of *CPU time* measured in the experiments in some literature. We feel that run time is a more comprehensive measure since it takes the total running time consumed as the measure of cost, whereas CPU time considers only the cost of the CPU resource. Also, all reports on the runtime of *FP-growth* include the time of constructing *FP-tree*s from the original databases.

The experiments are pursued on both synthetic and real data sets. The synthetic data sets which we used for our experiments were generated using the procedure described in [AS94]. We refer readers to it for more details on the generation of data sets.

We report experimental results on two synthetic data sets. The first one is T10.I4.D100K with 1K items. In this data set, the average transaction size and average maximal potentially frequent itemset size are set to 10 and 4, respectively, while the number of transactions in the dataset is set to 100K. It is a sparse dataset. The frequent itemsets are short and not numerous.

The second synthetic data set we used is T25.I20.D100K with 10K items. The average transaction size and average maximal potentially frequent itemset size are set to 25 and 20, respectively. There exist exponentially numerous frequent itemsets in this data set when the support threshold goes down. There are also pretty long frequent itemsets as well as a large number of short frequent itemsets in it. It contains abundant mixtures of short and long frequent itemsets.

To test the capability of *FP-growth* on dense datasets with long patterns, we use the real data set *Connect-4*, compiled from the *Connect-4* game state information. The data set is from the UC-Irvine Machine Learning Database Repository[1]. It contains 67,557 transactions, while each transaction is with 43 items. It is a dense dataset with a lot of long frequent itemsets.

### 3.4.2 Compactness of *FP-tree*

To test the compactness of *FP-tree*s, we compare the sizes of the following structures.

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

- *Alphabetical* FP-tree. It includes the space of all the links. However, in such an *FP-tree*, the alphabetical order of items are used instead of frequency descending order.
- *Ordered* FP-tree. Again, the size covers that of all links. In such an *FP-tree*, the items are sorted according to frequency descending order.
- *Transaction database.* Each item in a transaction is stored as an integer. It is simply the sum of occurrences of items in transactions.
- *Frequent transaction database.* That is the sub-database extracted from the original one by removing all infrequent items.

In real dataset *Connect-4*, *FP-tree* achieves good compactness. As seen from the result shown in Figure 3.6, the size of ordered *FP-tree* is always smaller than the size of the transaction database and the frequent transaction database. In a dense database, the size of the database and that of its frequent database are close. The size of the alphabetical *FP-tree* is smaller than that of the two databases in most cases but is slightly larger (about 1.5 to 2.5 times larger) than the size of the ordered *FP-tree*. It indicates that the frequency-descending ordering of the items benefits data compression in this case.

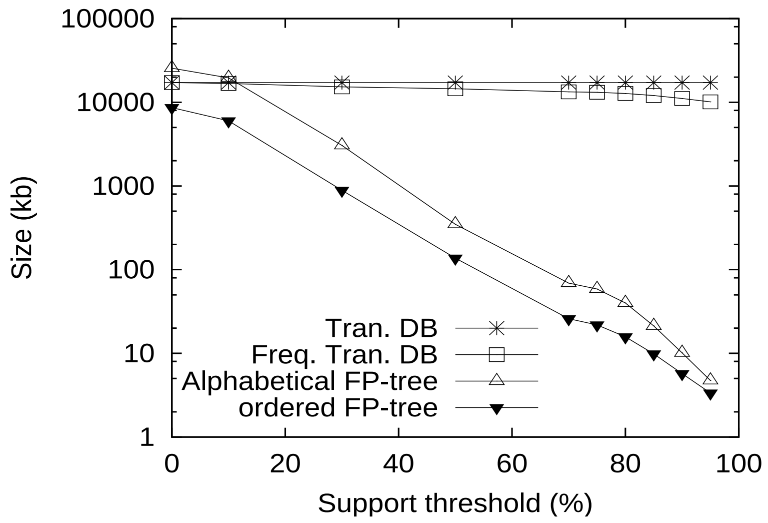

Figure 3.6: Compactness of *FP-tree* over data set *Connect-4*.

In dataset T25.I20.D100k, which contains abundant mixture of long and short frequent patterns, *FP-tree* is compact most of the time. The result is shown in Figure 3.7. Only when the support threshold lower than 2.5% does the size of *FP-tree* larger than that of frequent database. And as long as the support threshold is over 1.5%, the *FP-tree* is smaller than the transaction database. The difference of sizes of ordered *FP-tree* and alphabetical *FP-tree* is quite small in this dataset. It is about 2%.

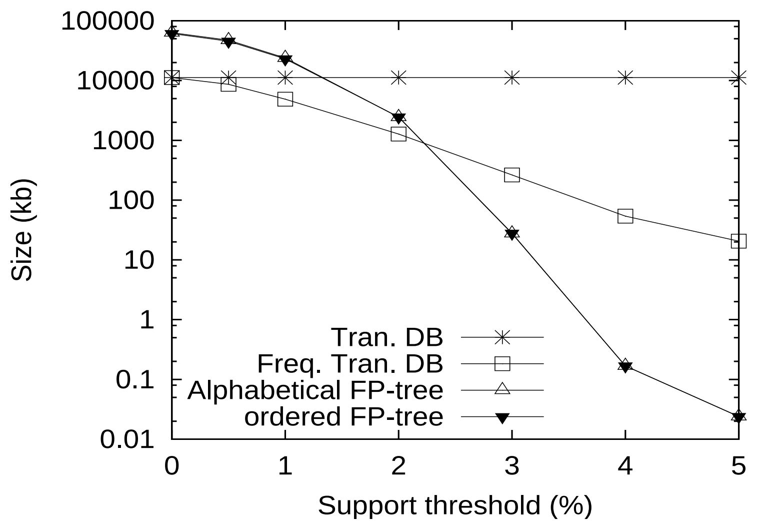

Figure 3.7: Compactness of *FP-tree* over data set T25.I20.D100k.

In sparse dataset T10.I4.D100k, *FP-tree* achieves good compactness when the support threshold is over 3.5%. Again, the difference of ordered *FP-tree* and alphabetical *FP-tree* is trivial. The result is shown in Figure 3.8.

From the above experiments, we have the following conclusions.

- FP-tree *achieves good compactness most of the time.* Especially in dense datasets, it can compress the database many times. Clearly, there is some overhead for pointers and counters. However, the gain of sharing among frequent projections of transactions is substantially more than the overhead and thus makes *FP-tree* space more efficient in many cases, which has been shown in our performance study.

- *When support is very low,* FP-tree *becomes bushy.* In such cases, the degree of sharing in branches of *FP-tree* becomes low. The overhead of links makes the size of *FP-tree* large. Therefore, instead of building *FP-tree*, we should construct projected databases. That is the reason why we build *FP-tree* for transaction database/projected database only when it passes certain density threshold. From the experiments, one can see

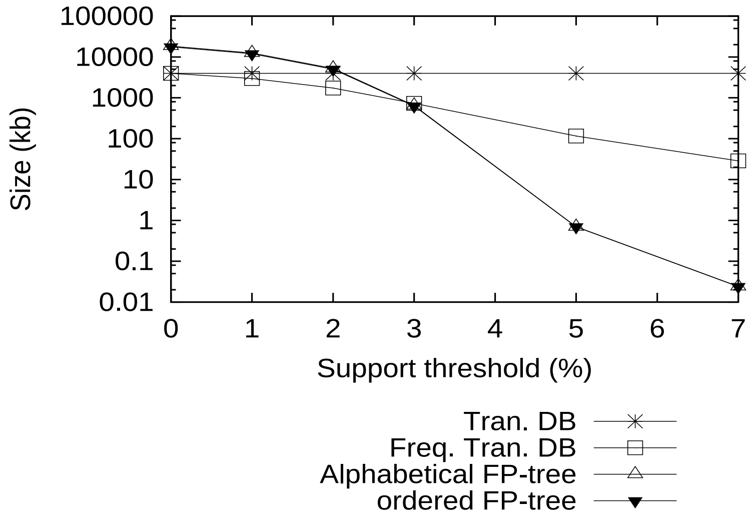

Figure 3.8: Compactness of *FP-tree* over data set T10.I4.D100k.

that such a threshold is pretty low, and easy to touch. Therefore, even for very large and/or sparse database, after one or a few rounds of database projection, *FP-tree* can be used for all the remaining mining tasks.

In the following experiments, we employed an implementation of *FP-growth* that integrates both database projection and *FP-tree* mining. The density threshold is set to 3%, and items are listed in frequency descending order.

### 3.4.3 Scalability Study

The scalability of *Apriori*, *TreeProjection* and *FP-growth* on synthetic data set T10.I4.D100K as the support threshold decreases from 0.15% to 0.01% is shown in Figure 3.9.

*FP-growth* is faster than both *Apriori* and *TreeProjection*. *TreeProjection* is faster and more scalable than *Apriori*. Since the dataset is sparse, as the support threshold is high, the frequent itemsets are short and the number of them is not large, the advantages of *FP-growth* and *TreeProjection* over *Apriori* are not so impressive. However, as the support threshold goes down, the gap is becoming wider. *FP-growth* can finish the computation for support threshold 0.01% within the time for *Apriori* over 0.05%. *TreeProjection* is also scalable, but is slower than *FP-growth*.

The advantages of *FP-growth* over *Apriori* becomes obvious when the dataset contains an

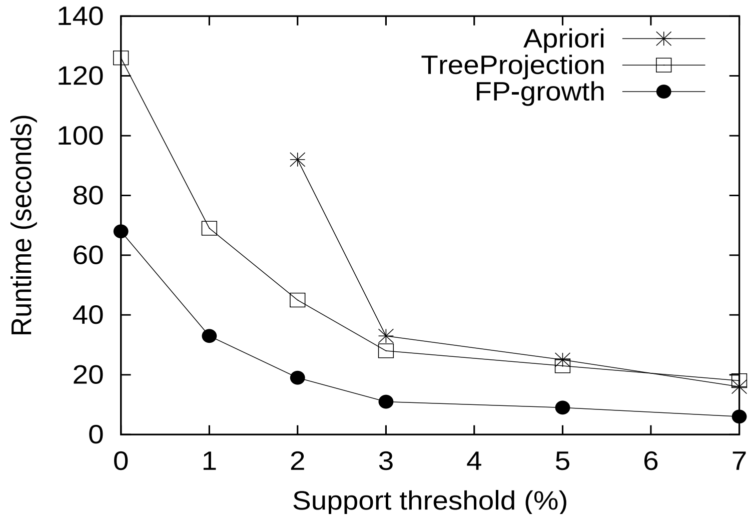

Figure 3.9: Scalability with threshold over sparse data set.

abundant mixtures of short and long frequent patterns. Figure 3.10 shows the experimental results of scalability with threshold over dataset T25.I20.D100k. *FP-growth* can mine with support threshold as low as 0.05%, with which *Apriori* cannot work out within reasonable time. *TreeProjection* is also scalable and faster than *Apriori*, but it is slower than *FP-growth*.

The advantage of *FP-growth* is dramatic in datasets with long patterns, which is challenging to the algorithms that mine the complete set of frequent patterns. The result on mining the real dataset *Connect-4* is shown in Figure 3.11. To the best of our knowledge, this is the first algorithm that handles such dense real dataset in performance study. From the figure, one can see that *FP-growth* is scalable even when there are many long patterns. Without candidate generation, *FP-growth* enumerates long patterns efficiently. In such datasets, neither *Apriori* nor *TreeProjection* are comparable to the performance of *FP-growth*. To deal with long patterns, *Apriori* has to generate tremendous number of candidates, that is a very costly process. The main costs in *TreeProjection* are matrix computation and transaction projection. In a database with a large number of frequent items, the matrices become quite large, and the computation cost jumps up substantially. In contrast, the height of *FP-tree* is limited by the maximal length of the transactions, and many transactions share the prefix paths of an *FP-tree*. This explains why *FP-growth* has distinct advantages when the support threshold is low and when the number of transactions is large.

To test the scalability of *FP-growth* against the number of transactions, a set of synthetic

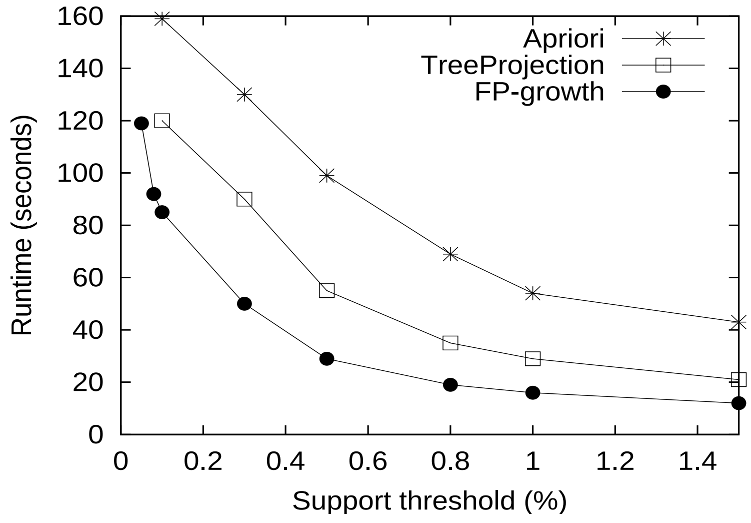

Figure 3.10: Scalability with threshold over dataset with abundant mixtures of short and long frequent patterns.

datasets are generated using the same parameters of T10.I4 and T25.I20, and the number of transactions ranges from 100k to 1M. *FP-growth* is tested over them using the same support threshold in percentage. The result is in Figure 3.12, which shows the linear increase of runtime with the number of transactions. Please note that unlike the way reported in some literature, we do not replicate transactions in real data sets to test the scalability. This is because no matter how many times the transactions are replicated, *FP-growth* builds up an *FP-tree* with the size identical to that of the original one, and the scaling-up of such databases becomes trivial.

### 3.4.4 Comparison Between *FP-growth* and *TreeProjection*

*TreeProjection* is described in Section 2.4. In comparison with the *FP-growth* method, *TreeProjection* suffers from some problems related to efficiency, scalability, and implementation complexity. We analyze them as follows.

First, *TreeProjection* may encounter difficulties at computing matrices when the database is huge, when there are a lot of transactions containing many frequent items, and/or when the support threshold is very low. This is because in such cases there often exist a large number of frequent items. The size of the matrices at high level nodes in the lexicographical tree can be huge, as shown in our introduction section. The study in *TreeProjection* [AAP00]

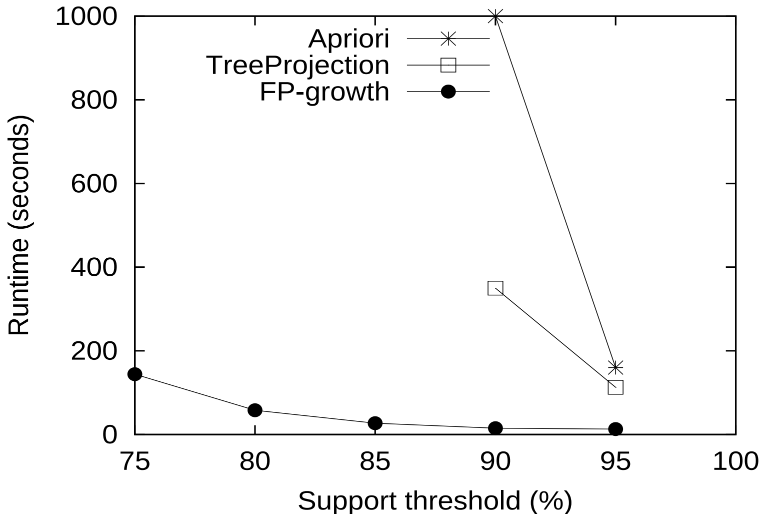

Figure 3.11: Scalability with threshold over *Connect-4*.

has developed some smart memory caching methods to overcome this problem. However, it could be wise not to generate such huge matrices at all instead of finding some smart caching techniques to reduce the cost. Moreover, even if the matrix can be cached efficiently, its computation still involves some nontrivial overhead. To compute a matrix at node $P$ with $n$ projected transactions, the cost is $O(\sum_{i=1}^{n} \frac{|T_i|^2}{2})$, where $\mid T_i \mid$ is the length of the transaction. If the number of transaction is large and the length of each transaction is long, the computation is costly. The *FP-growth* method will never need to build up matrices and compute 2-itemset frequency since it avoids the generation of any candidate $k$-itemsets for any $k$ by applying a pattern growth method. Pattern growth can be viewed as successive computation of frequent 1-itemset (of the database and conditional pattern bases) and assembling them into longer patterns. Since computing frequent 1-itemsets is much less expensive than computing frequent 2-itemsets, the cost is substantially reduced.

Second, since one transaction may contain many frequent itemsets, one transaction in *TreeProjection* may be projected many times to many different nodes in the lexicographical tree. When there are many long transactions containing numerous frequent items, transaction projection becomes an nontrivial cost of *TreeProjection* . The *FP-growth* method constructs *FP-tree* which is a highly compact form of transaction database. Thus both the size and the cost of computation of conditional pattern bases, which corresponds roughly to the compact form of projected transaction databases, are substantially reduced.

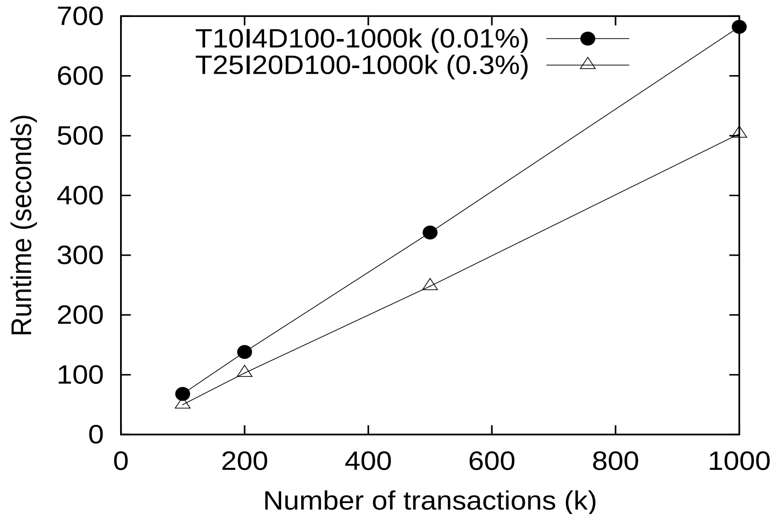

Figure 3.12: Scalability of *FP-growth* with number of transactions.

Third, *TreeProjection* creates one node in its lexicographical tree for each frequent itemset. At the first glance, this seems to be highly compact since *FP-tree* does not ensure that each frequent node will be mapped to only one node in the tree. However, each branch of the *FP-tree* may store many "hidden" frequent patterns due to the potential generation of many combinations using its prefix paths. Notice that the total number of frequent $k$-itemsets can be very large in a large database or when the database has quite long frequent itemsets. For example, for a frequent itemset $(a_1, a_2, \cdots, a_{100})$, the number of frequent itemsets at the 50th-level of the lexicographic tree will be $\begin{pmatrix} 100 \\ 50 \end{pmatrix} = \frac{100!}{50! \times 50!} \approx 1.0 \times 10^{29}$. For the same frequent itemset, *FP-tree* and *FP-growth* will only need one path of 100 nodes.

In summary, *FP-growth* mines frequent itemsets by (1) constructing highly compact *FP-trees* which share numerous "projected" transactions and hide (or carry) numerous frequent patterns, and (2) applying progressive pattern growth of frequent 1-itemsets which avoids the generation of any potential combinations of candidate itemsets implicitly or explicitly, whereas *TreeProjection* must generate candidate 2-itemsets for each projected database. Therefore, *FP-growth* is more efficient and more scalable than *TreeProjection*, especially when the number of frequent itemsets becomes really large. These observations and analyses are well supported by our experiments reported in this section.

## 3.5 Summary

In this chapter, we have proposed a novel data structure, *frequent pattern tree* (*FP-tree*), for storing compressed, crucial information about frequent patterns, and developed a pattern growth method, *FP-growth*, for efficient mining of frequent patterns in large databases.

There are several advantages of *FP-growth* over other approaches.

1. It constructs a highly compact *FP-tree*, which is usually substantially smaller than the original database and thus saves the costly database scans in the subsequent mining processes.

2. It applies a pattern growth method which avoids costly candidate generation and test by successively concatenating frequent 1-itemset found in the (conditional) *FP-tree*s. This ensures that it never generates any combinations of new candidate sets which are not in the database because the itemset in any transaction is always encoded in the corresponding path of the *FP-tree*s. In this context, mining is not *Apriori*-like (*restricted*) *generation-and-test* but *frequent pattern* (*fragment*) *growth only.* The major operations of mining are count accumulation and prefix path count adjustment, which are usually much less costly than candidate generation and pattern matching operations performed in most *Apriori*-like algorithms.

3. It applies a partitioning-based divide-and-conquer method which dramatically reduces the size of the subsequent conditional pattern bases and conditional *FP-tree*s. Several other optimization techniques, including direct pattern generation for single tree-path and employing the least frequent events as suffix, also contribute to the efficiency of the method.

We have implemented the *FP-growth* method, studied its performance in comparison with several influential frequent pattern mining algorithms in large databases. Our performance study shows that the method mines both short and long patterns efficiently in large databases, outperforming the current candidate pattern generation-based algorithms. The *FP-growth* method has also been implemented in the new version of DBMiner system and been tested in large industrial databases, such as in London Drugs databases, with satisfactory performance.

There are a lot of interesting research issues related to *FP-tree*-based mining, including further study and implementation of SQL-based, highly scalable *FP-tree* structure, constraint-based mining of frequent patterns using *FP-tree*s, and the extension of the *FP-tree*-based mining method for mining sequential patterns [AS95], closed patterns [PBTL99], max-patterns [Bay98], partial periodicity [HDY99], and other interesting frequent patterns. Some of them will be addressed in the following chapters.

Chapter 4

# *H-mine*: Scalable Space-preserving Mining

In Chapter 3, we developed *FP-growth*, a pattern-growth method for frequent pattern mining. Although *FP-growth* is more efficient than *Apriori* in many cases, it may still encounter some difficulties in some cases, as illustrated below.

- *Huge space is required to serve the mining.* *FP-growth* avoids candidate generation by compressing the transaction database into an *FP-tree* and pursuing partition-based mining recursively. However, if the database is *huge and sparse*, the *FP-tree* will be large and the space requirement for recursion is a challenge.

- *Real databases contain all the cases.* Real data sets can be sparse and/or dense in different applications. For example, in telecommunication data analysis, calling patterns for home users vs. business users could be very different: some are frequent and dense (e.g., to family members and close friends), but some are huge and sparse. Similar situations arise for market basket analysis, census data analysis, classification and predictive modelling, etc. It is hard to select an appropriate mining method on the fly if no algorithm fits all.

- *Large applications need more scalability.* Many existing methods are efficient when the data set is not very large. Otherwise, their core data structures (such as *FP-tree*) or the intermediate results (e.g., the set of candidates in *Apriori* or the recursively generated conditional databases in *FP-growth*) may not fit in main memory and easily cause thrashing.

This poses a new challenge: "*Can we work out a better method which is (1) efficient in all occasions (dense vs. sparse, huge vs. memory-based data sets), and (2) space requirement is small, even for very large databases?*"

In this chapter, we propose a new data structure, *H-struct*, and a new mining method, *H-mine*, to overcome these difficulties. One major feature of *H-mine* is that it is *space-preserving*, meaning (1) *H-mine* is moderate in memory usage; (2) it can fully utilize all available main memory space if necessary; and (3) it performs well even with very small main memory. We make the following progress.

1. A memory-based, efficient pattern-growth algorithm, *H-mine(Mem)*, is proposed for mining frequent patterns for the data sets that can fit in (main) memory. A simple, memory-based hyper-structure, *H-struct*, is designed for fast mining.

2. We show that, theoretically, *H-mine(Mem)* has polynomial space usage and is thus more space efficient than *FP-growth* and *TreeProjection* when mining sparse data sets, and also more efficient than *Apriori*-based methods which generate a large number of candidates. Experimental results show that *H-mine(Mem)* has exactly predictable space overhead and, in many cases, it is faster than memory-based *Apriori* and *FP-growth* with very limited space usage.

3. Based on *H-mine(Mem)*, we propose *H-mine*, a scalable algorithm for mining large databases by first partitioning the database, mining each partition in memory using *H-mine(Mem)*, and then consolidating global frequent patterns.

4. For dense data sets, *H-mine* is integrated with *FP-growth* dynamically by detecting the swapping condition and constructing FP-trees for efficient mining.

5. Such efforts ensure that *H-mine* is scalable in both large and medium sized databases and in both sparse and dense data sets. Our comprehensive performance study confirms that *H-mine* is highly scalable and is faster than *Apriori* and *FP-growth* in all occasions.

The remainder of the chapter is organized as follows. Section 4.1 is devoted to *H-mine(Mem)*, an efficient algorithm for memory-based frequent pattern mining. In Section

4.2, *H-mine(Mem)* is extended to huge, disk-based databases, together with some further optimization techniques. Our performance study is reported in Section 4.3. We discuss related issues and conclude the chapter in Section 4.4.

## 4.1 ***H-mine(Mem)***: Memory-Based Hyper-Structure Mining

In this section, *H-mine(Mem)* (memory-based hyper-structure mining of frequent patterns) is developed, and in Section 4.2, the method is extended to handle large and/or dense databases.

### 4.1.1 General idea of ***H-mine(Mem)***

Our general idea of *H-mine(Mem)* is illustrated in the following example.

**Example 4.1** Let the first two columns of Table 4.1 be our running transaction database $TDB$. Let the minimum support threshold be $min\_sup = 2$.

| Transaction ID | Items | Frequent-item projection |
|---|---|---|
| 100 | $c, d, e, f, g, i$ | $c, d, e, g$ |
| 200 | $a, c, d, e, m$ | $a, c, d, e$ |
| 300 | $a, b, d, e, g, k$ | $a, d, e, g$ |
| 400 | $a, c, d, h$ | $a, c, d$ |

Table 4.1: The transaction database $TDB$ as our running example.

Following the *Apriori* property (Theorem 2.1), only frequent items play roles in frequent patterns. By scanning $TDB$ once, the complete set of frequent items $\{a : 3, c : 3, d : 4, e : 3, g : 2\}$ can be found and output, where the notation $a : 3$ means item $a$'s support (occurrence frequency) is 3. Let $freq(X)$ (the *frequent-item projection* of $X$) be the set of frequent items in itemset $X$. For the ease of explanation, the frequent-item projections of all the transactions of Table 4.1 are shown in the third column of the table.

Following the alphabetical order of frequent items[1] (called *F-list*): *a-c-d-e-g*, the complete set of frequent patterns can be partitioned into 5 subsets as follows: (1) those containing item $a$; (2) those containing item $c$ but no item $a$; (3) those containing item $d$ but no item $a$

[1]Any ordering should work, and the alphabetical ordering is just for the convenience of explanation.

nor $c$; (4) those containing item $e$ but no item $a$ nor $c$ nor $d$; and (5) those containing only item $g$, as shown in Figure 4.1.

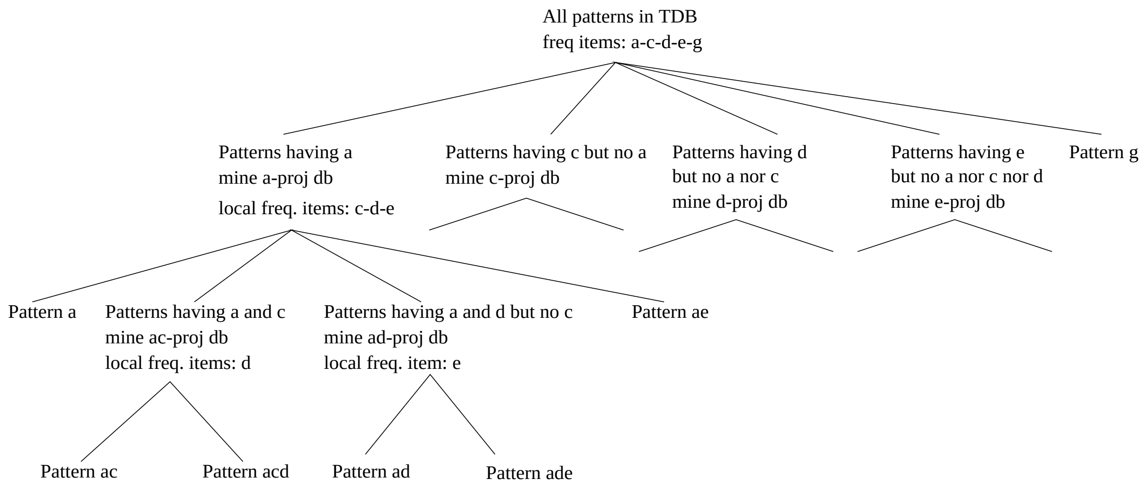

Figure 4.1: Divide-and-conquer tree for frequent patterns.

If the frequent-item projections of transactions in the database can be held in main memory, they can be organized as shown in Figure 4.2. All items in frequent-item projections are sorted according to the *F-list*. For example, the frequent-item projection of transaction 100 is listed as *cdeg*. Every occurrence of a frequent item is stored in an entry with two fields: an *item-id* and a *hyper-link*.

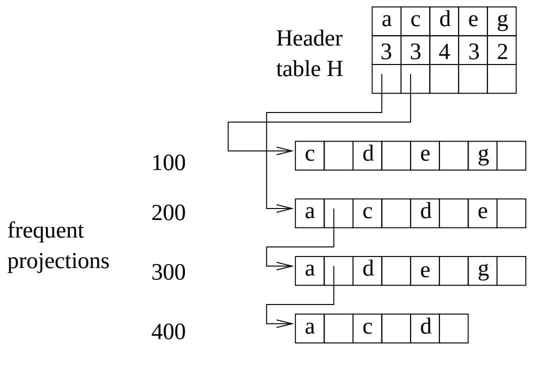

Figure 4.2: *H-struct*, the hyper-structure for storing frequent-item projections.

A *header table H* is created, where each frequent item entry has three fields: an *item-id*, a *support count*, and a *hyper-link*. When the frequent-item projections are loaded into memory, those with the same first item (in the order of *F-list*) are linked together by the hyper-links as a queue, and the entries in header table $H$ act as the heads of the queues. For example, the entry of item $a$ in the header table $H$ is the head of $a$-queue, which links frequent-item projections of transactions 200, 300, and 400. These three projections all have item $a$ as their first frequent item (in the order of *F-list*). Similarly, frequent-item projection of transaction 100 is linked as *c-queue*, headed by item $c$ in $H$. The $d$-, $e$- and $g$-queues are empty since there is no frequent-item projection that begins with any of these items.

Clearly, it takes one scan (the second scan) of the transaction database $TDB$ to build such a memory structure (called *H-struct*). Then the remaining of the mining can be performed on the *H-struct* only, without referencing any information in the original database. After that, the five subsets of frequent patterns can be mined one by one as follows.

First, let us consider how to find the set of frequent patterns in the first subset, i.e., all the frequent patterns containing item $a$. This requires to search all the frequent-item projections containing item $a$, i.e., the *a-projected database*[2], denoted as $TDB|_a$. Interestingly, the frequent-item projections in the $a$-projected database are already linked in the $a$-queue, which can be traversed efficiently.

To mine the $a$-projected database, an *a-header table* $H_a$ is created, as shown in Figure 4.3. In $H_a$, every frequent item except for $a$ itself has an entry with the same three fields as $H$, i.e., *item-id, support count* and *hyper-link*. The support count in $H_a$ records the support of the corresponding item in the $a$-projected database. For example, item $c$ appears twice in $a$-projected database (i.e., frequent-item projections in the $a$-queue), thus the support count in the entry $c$ of $H_a$ is 2.

By traversing the $a$-queue once, the set of locally frequent items, i.e., the items appearing at least twice, in the $a$-projected database is found, which is $\{c : 2, d : 3, e : 2\}$ (Note: $g : 1$ is not locally frequent and thus will not be considered further.) This scan outputs frequent patterns $\{ac : 2, ad : 3, ae : 2\}$ and builds up links for $H_a$ header as shown in Figure 4.3.

Thus the set of frequent patterns containing item $a$ can be further partitioned into four subsets: (1) the pattern $a$ itself; (2) those containing $a$ and $c$; (3) those containing $a$ and $d$ but no $c$; and (4) those containing $a$ and $e$ but no $c$ nor $d$, i.e., pattern $ae$. The divide-and-conquer tree for frequent patterns is shown in Figure 4.1. These four subsets are mined as follows.

[2]The $a$-projected database consists of all the frequent-item projections containing item $a$, but these are all "virtual" projections since no physical projections are performed to create a new database.

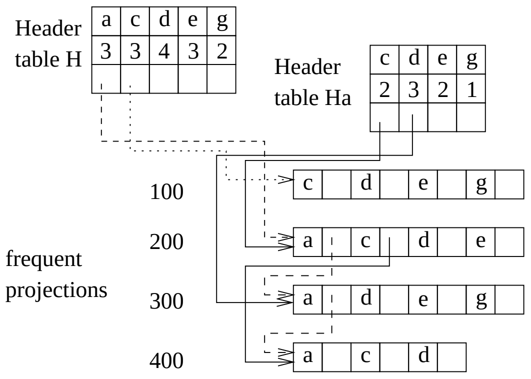

Figure 4.3: Header table $H_a$ and $ac$-queue.

1. The first subset contains only frequent pattern $a$.

2. The $a$-queue is traversed recursively to find frequent patterns containing $a$ and $c$. First, the frequent-item projections whose first local frequent item is $a$ are linked together by the hyper-links as a queue, and the entries in the header table $H_a$ act as the heads of the queues. Here, frequent-item projections of transactions 200 and 400 are added to the $ac$-queue in any order. Transaction 300 are added to the $ad$-queue. At this moment, the $ae$-queue is empty. This situation is shown in Figure 4.3.

   The process continues recursively for the $ac$-projected database by examining the $c$-queue in $H_a$. This process creates the *ac-header table* $H_{ac}$, as shown in Figure 4.4.

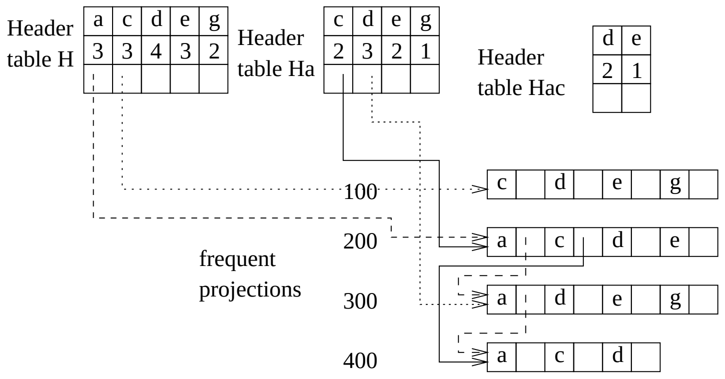

Figure 4.4: Header table $H_{ac}$.

Since only item $d$ : 2 is a locally frequent item in the $ac$-projected database, only $acd$ : 2 is output, and the search along this path is completed.

3. The recursion backtracks to find frequent patterns containing $a$ and $d$ but not $c$. Since the queue started from $d$ in the header table $H_a$, i.e., the $ad$-queue, links all frequent-item projections containing items $a$ and $d$ (but excluding item $c$ in the projection), one can get the complete $ad$-projected database by inserting frequent-item projections having item $d$ in the $ac$-queue into the $ad$-queue. This involves one more traversal of the $ac$-queue. Each frequent-item projection in the $ac$-queue is appended to the queue of the next frequent item in the projection according to *F-list*. Since all the frequent-item projections in the $ac$-queue have item $d$, they are all inserted into the $ad$-queue, as shown in Figure 4.5.

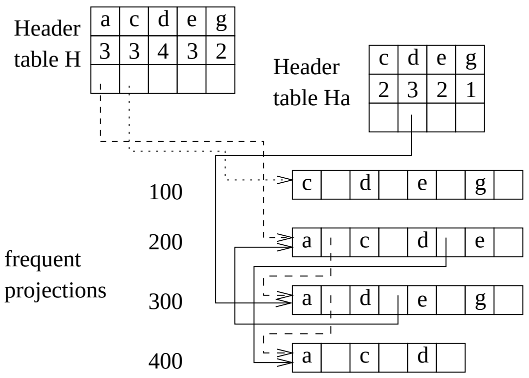

Figure 4.5: Header table $H_a$ and $ad$-queue.

It can be seen that, after the adjustment, the $ad$-queue collects the complete set of frequent-item projections containing items $a$ and $d$. Thus, the set of frequent patterns containing items $a$ and $d$ can be mined recursively. Please note that, even though item $c$ appears in frequent-item projections of $ad$-projected database, we do not consider it as a locally frequent item in any recursive projected database since it has been considered in the mining of the $ac$-queue. This mining generates only one pattern $ade : 2$. Notice also the third level header table $H_{ad}$ can use the table $H_{ac}$ since the search for $H_{ac}$ was done in the previous round. Thus we only need one header table at the third level. Later we can see that only one header table is needed for each level in the whole mining process.

4. Since there is no transaction in the $ae$-projected database, the only frequent pattern in this projected database is $ae$ itself. The search terminates.

After the frequent patterns containing item $a$ are found, the transactions in the $a$-projected database, i.e., $a$-queue, should be further projected to other projected databases. Since the $c$-queue includes all frequent-item projections containing item $c$ except for those projections containing both items $a$ and $c$, which are in the $a$-queue. To mine all the frequent patterns containing item $c$ but no $a$, and other subsets of frequent patterns, we need to insert all the projections in the $a$-queue into the proper queues.

We traverse the $a$-queue once more. Each frequent-item projection in the queue is appended to the queue of the next item in the projection following $a$ in the *F-list*, as shown in Figure 4.6. For example, frequent-item projection $acde$ is inserted into $c$-queue and $adeg$ is inserted into $d$-queue.

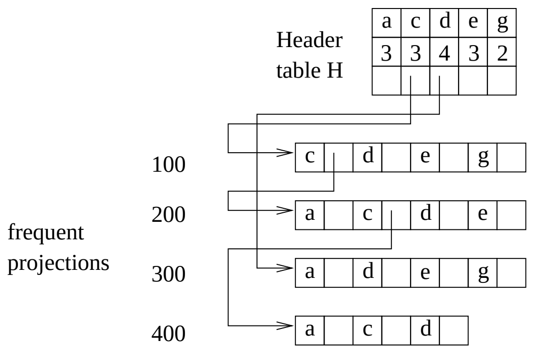

Figure 4.6: Adjusted hyper-links after mining $a$-projected database.

By mining the $c$-projected database recursively (with shared header table at each level), we can find the set of frequent patterns containing item $c$ but no $a$. Notice item $a$ will not be included in the $c$-projected database since all the frequent patterns having $a$ have already been found.

Similarly, the mining goes on. In the next section, we verify that the above mining process finds the complete set of frequent patterns without duplication. The remaining mining process is left as an exercise to interested readers.

Notice also that the depth-first search for mining the first set of frequent patterns at any depth can be done in one database scan by constructing the header tables at all levels simultaneously. ■

### 4.1.2 ***H-mine(Mem)*: The algorithm for memory-based hyper-structure mining**

Now, let us summarize and justify the mining process shown in Example 4.1.

Given a transaction database $TDB$ and a support threshold $min\_sup$, let $L$ be the set of frequent items. *F-list*, a list of frequent items, is a global order over $L$. Let $x$ and $y$ $(x \neq y)$ be two frequent items. We denote $x \prec y$ if and only if $x$ is before $y$ according to the *F-list*. For example, based on the *F-list* in Example 4.1, we have $a \prec c \prec d \prec e \prec g$.

Frequent-item projections of transactions in $TDB$ are organized in an *H-struct* defined below.

1. An *H-struct* contains the set of frequent-item projections of a transaction database. Each item in a frequent-item projection is represented by an entry with two fields: *item-id* and *hyper-link*.

2. An *H-struct* has a *header table*. The header table is an array of frequent items in the order of *F-list*. A support count and a hyper-link are attached to each item in the header table.

3. When the *H-struct* is created, items in the header table are the *heads* of *queues* of frequent-item projections linked by hyper-links.

The hyper-structure shown in Figure 4.2 is an example of *H-struct*. About *H-struct*, we have the following lemma.

**Lemma 4.1 (*H-struct*)** *Given a transaction database $TDB$, a support threshold $min\_sup$, and* F-list*.*

1. H-struct *is unique without considering the order of frequent-item projections in queues of frequent-item projections.*

2. *The space requirement of an* H-struct *is $k\sum_{t \in TDB} |freq(t)|$, where $freq(t)$ is a frequent-item projection of a transaction $t$, and $k$ is a constant.*

3. *Two and only two scans of a transaction database are needed to build an* H-struct*.*

**Proof.** For every transaction, *H-struct* stores its frequent-item projection. Besides frequent-item projections, *H-struct* also stores a header table. The maximal number of entries in the table is at most the number of frequent items, while the size of each entry is fixed. This is all the space needed by the *H-struct*. Therefore, we have the space requirement as shown in the formula of item 2. The remaining part of the lemma follows the definition of *H-struct* immediately. ■

Given a transaction database $TDB$ and *F-list*, the complete set of frequent patterns can be partitioned into a series of subsets without overlap, as stated in the following lemma.

**Lemma 4.2 (Partition of search space)** *Given a transaction database $TDB$ and support threshold min_sup, let* F-list: *"$x_1$-...-$x_n$" be a list of frequent items. The complete set of frequent patterns can be partitioned into $n$ subsets without overlap as follows: the $k$-th subset $(1 \leq k \leq n)$ contains patterns having item $x_k$ but no item $x_i$ $(1 \leq i < k)$.*

**Proof.** Let $P$ be a frequent pattern. We sort items in $P$ according to the *F-list*. Thus, there exists a $k$ $(1 \leq k \leq n)$ such that $x_k$ is the first item in $P$. Pattern $P$ belongs to the $k$-th subset. On the other hand, suppose $P$ also belongs to the $k'$-th subset where $k \neq k'$. Without loss of generality, suppose $k < k'$. $P$ in the $k'$-th subset requires that $P$ does not contain any item before $x_{k'}$, i.e., $P$ does not contain $x_k$. That leads to a contradiction. ■

To mine the subsets of frequent patterns, we introduce the concept of *projected database*. Let $P$ be a frequent pattern. The *P-projected database* is the collection of frequent-item projections containing pattern $P$, denoted as $TDB|_P$.

Clearly, to mine the $k$-th subset of frequent patterns in Lemma 4.2, we only need to look at the $x_k$-projected database $TDB|_{x_k}$ and ignore occurrences of items $x_i$'s $(1 \leq i < k)$. *How can* H-struct *facilitate the construction of projected databases?* We have the following lemma.

**Lemma 4.3 (Projected databases)** *Given a transaction database $TDB$ and support threshold min_sup, let* F-list: *"$x_1$-...-$x_n$" be the list of frequent items. In the* H-struct*,*

1. *The $x_1$-projected database is the $x_1$-queue in the header table.*
2. *The $x_2$-projected database is the $x_2$-queue in the header table and the frequent-item projections starting at item $x_2$ in the $x_1$-queue.*
3. *In general, the $x_k$-projected database $(1 < k \leq n)$ is the $x_k$-queue in the header table and the frequent-item projections starting at $x_k$ in the $x_i$-queues $(1 \leq i < k)$.*

**Proof.** The lemma follows the definitions of *H-struct* and projected databases. ■

Based on Lemma 4.3, we can first find the complete set of frequent patterns containing $x_1$, using the $x_1$-queue available in *H-struct*. Conceptually, we treat the queue of frequent-item projections in $x_1$-projected database as a sub-*H-struct* and apply the techniques recursively. That is, we find the locally frequent items, further partition the subset of frequent patterns and doing recursive mining. The storage of frequent-item projections, i.e., *H-struct*, can be shared. What we need is only a new header table to form queues within the $x_1$-projected database.

Then, we insert those frequent-item projections in $x_1$-queue starting at item $x_2$ to the $x_2$-queue in the header table, and form the complete $x_2$-projected database. Since the projections exclude $x_1$ in the $x_2$-projected database by starting at $x_2$, we can find the complete set of frequent patterns containing item $x_2$ but no item $x_1$.

Similarly, we can find the complete set of frequent patterns. Based on the above reasoning, we have the following algorithm.

**Algorithm 4 (*H-mine(Mem)*)** (Main) memory-based hyper-structure mining of frequent patterns.

**Input:** A transaction database $TDB$ and a support threshold $min\_sup$.

**Output:** The complete set of frequent patterns.

**Method:**

```
1. scan $TDB$ once, find and output $L$, the set of frequent items. Let F-list: "$x_1$-
   $\ldots$-$x_n$" ($n = |L|$) be a list of frequent items.
2. scan $TDB$ again, construct H-struct, with header table $H$, and with each $x_i$-
   queue linked to the corresponding entry in $H$.
3. for $i = 1$ to $n$ do
   (a) call H-mine($x_i, H$, F-list)
   (b) traverse the $x_i$-queue in the header table $H$, for each frequent-item projection
       $X$, link $X$ to the $x_j$-queue in the header table $H$, where $x_j$ is the item in $X$
       following $x_i$ immediately.
```

```
Procedure H-mine($P, H,$ F-list) // $P$ is a frequent pattern
    // Note: The frequent-item projections in the $P$-projected database are linked as a
    $P$-queue in the header table $H$.

    1. traverse $P$-queue once, find and output its locally frequent items and derive
       F-list$_P$: "$x_{j_1}$-...-$x_{j_{n'}}$".
       // Note: Only the items in the F-list and are located to the right of $P$ are
       considered. Items in the F-list$_P$ follow the same order as that in the F-list.
    2. construct header table $H_P$, scan the $P$-projected database, and for each frequent-
       item projection $X$ in the projected database, use the hyper-link of $x_{j_i}$ $(1 \leq i \leq n')$
       in $X$ to link $X$ to the $Px_{j_i}$-queue in the header table $H_P$, where $x_{j_i}$ is the first
       locally frequent item in $X$ according to the F-list$_P$.
    3. for $i = 1$ to $n'$ do
        (a) call H-mine($P \cup \{x_{j_i}\}, H_P,$ F-list$_P$).
        (b) traverse $Px_{j_i}$-queue in the header table $H_P$, for each frequent-item projection
            $X$, link $X$ to the $x_{j_k}$-queue $(i < k \leq n')$ in the header table $H_P$, where $x_{j_k}$
            is the item in $X$ following $x_{j_i}$ immediately according to F-list.
```

**Analysis.** The correctness and completeness of the algorithm can be shown by an induction of enumeration of frequent patterns based on Lemma 4.2. Lemma 4.3 guarantees that by properly adjusting the hyper-links, the algorithm correctly finds the subsets of patterns using the right projected database. ■

### 4.1.3 Space Usage of *H-mine(Mem)*

Now, let us analyze the space usage of Algorithm 4.

As shown in Lemma 4.1, the space usage of constructing an *H-struct* is at most

$$(k \sum_{t \in TDB} |freq(t)|)$$

where $k$ is a constant. To mine the *H-struct*, the only space overhead is a set of local header tables. At first glance, the number of header tables seems to be in the scale of that of frequent patterns. However, a close look at the algorithm finds that only a very limited number of header tables exist simultaneously. For example, to find pattern $P = bcde$, only the header tables for the "prefixes" of $P$, i.e., $H_b$, $H_{bc}$, $H_{bcd}$ and $H_{bcde}$, are needed. All the other header tables either are already used and can be freed, or have not been generated yet. The header tables for patterns with item $a$ have already been used and can be freed since all the patterns having item $a$ have been found before pattern *bcde*. On the other hand, all the other header tables are for patterns to be found later and thus need not be generated at this moment. Therefore, the number of header tables is no more than the maximal length of a single frequent pattern. Thus we have the following lemma.

**Lemma 4.4 (Number of header tables)** *The maximum number of header tables needed in the hyper-structure mining of frequent patterns. i.e.,* H-mine(Mem)*, is at most the maximal length of single frequent pattern that can be found.* ■

Since the maximal length of a single frequent pattern cannot exceed the maximal length of a transaction, and in general, the maximal length of a transaction is much smaller than the number of transactions, we have the following theorem on the space usage of *H-mine(Mem)*.

**Theorem 4.1 (Space usage)** *The space usage of Algorithm 4 is*

$$k \sum_{t \in TDB} |freq(t)|$$

*where $freq(t)$ is a frequent-item projection of a transaction $t$, and $k$ is a constant.*

**Proof.** This is based on (1) as shown in Lemma 4.1, the space requirement of an *H-struct* is $k \sum_{t \in TDB} |freq(t)|$, and (2) the space required for the remaining is a set of $m$ header tables, each being much smaller than *H-struct* and in the worst case being the same scale of *H-struct* , where $m$ is the maximal length of any transaction in $TDB$. ■

Comparing with other frequent pattern mining methods, the efficiency of *H-mine(Mem)* comes from the following aspects.

1. *H-mine(Mem)* avoids candidate generation and test by adopting a frequent-pattern growth methodology. *H-mine(Mem)* absorbs the advantages of pattern-growth.

2. *H-mine(Mem)* confines its search in a dedicated space. Unlike *FP-growth*, it does not need to physically construct memory structures of projected databases. It fully utilizes the information well organized in the *H-struct*, and collects information about projected databases using header tables, which are light-weight structures. That also saves a lot of efforts on managing space.

3. *H-mine(Mem)* does not need to store any frequent patterns in memory. Once a frequent pattern is found, it is output to disk. In contrast, the candidate-generation-and-test method has to save and use the frequent patterns found in the current round to generate candidates for the next round.

The above analysis is verified by our extensive performance study, as presented in Section 4.3.

## 4.2 From *H-mine(Mem)* to *H-mine*: Efficient Mining in Different Occasions

In this section, we first extend our algorithm *H-mine(Mem)* to *H-mine*, which mines frequent-patterns in large data sets that cannot fit in main memory. Then, we explore how to integrate *FP-growth* when the data sets being mined become very dense.

### 4.2.1 *H-mine*: Mining Frequent Patterns in Large Databases

*H-mine(Mem)* is efficient when the frequent-item projections of a transaction database plus a set of header tables can fit in main memory. However, we cannot expect this is always the case. When they cannot fit in memory, a database partitioning technique can be developed as follows.

Let $TDB$ be the transaction database with $n$ transactions and $min\_sup$ be the support threshold. By scanning $TDB$ once, one can find $L$, the set of frequent items.

Then, $TDB$ can be partitioned into $k$ parts, $TDB_1, \ldots, TDB_k$, such that, for each $TDB_i$ ($1 \leq i \leq k$), the frequent-item projections of transactions in $TDB_i$ can be held in main memory, where $TDB_i$ has $n_i$ transactions, and $\sum_{i=1}^{k} n_i = n$. We can apply *H-mine(Mem)* to $TDB_i$ to find frequent patterns in $TDB_i$ with the minimum support threshold $min\_sup_i = \lfloor min\_sup \times \frac{n_i}{n} \rfloor$ (i.e., each partitioned database keeps the same relative minimum support as the global database).

Let $F_i$ ($1 \leq i \leq k$) be the set of (locally) frequent patterns in $TDB_i$. Based on the property of partition-based mining [SON95], $P$ cannot be a (globally) frequent pattern in $TDB$ with respect to the support threshold $min\_sup$ if there exists no $i$ ($1 \leq i \leq k$) such that $P$ is in $F_i$. Therefore, after mining frequent patterns in $TDB_i$'s, we can gather the patterns in $F_i$'s and collect their (global) support in $TDB$ by scanning the transaction database $TDB$ one more time.

Based on the above observation, we can extend *H-mine(Mem)* to *H-mine* as follows.

**Algorithm 5 (*H-mine*)** Hyper-structure mining of frequent-patterns in large databases.

**Input and output:** same as Algorithm 4.

**method:**

1. Scan transaction database $TDB$ once to find $L$, the complete set of frequent items.
2. Partition $TDB$ into $k$ parts, $TDB_1$, ..., $TDB_k$, such that, for each $TDB_i$ ($1 \leq i \leq k$), the frequent-item projections in $TDB_i$ can be held in main memory.
3. For $i = 1$ to $k$, use *H-mine(Mem)* to mine frequent patterns in $TDB_i$ with respect to the minimum support threshold $min\_sup_i = \lfloor min\_sup \times \frac{n_i}{n} \rfloor$, where $n$ and $n_i$ are the numbers of transactions in $TDB$ and $TDB_i$, respectively. Let $F_i$ be the set of frequent patterns in $TDB_i$.
4. Let $F = \bigcup_{i=1}^{k} F_i$. Scan $TDB$ one more time, collect support for patterns in $F$. Output those patterns which pass the minimum support threshold $min\_sup$. ■

One important issue in Algorithm 5 is how to partition the database. As analyzed in Section 4.1.2, the only space cost of *H-mine(Mem)* incurred by the header tables. The maximal number of header tables as well as their space requirement are predictable (usually very small in comparison with the size of frequent-item projections). Therefore, after reserving space for header tables, the remaining main memory can be used to build an *H-struct* that covers as many transactions as possible. In practice, it is good to first estimate the size of available main memory for mining and the size of the overall frequent-item projected database (in the scale of the sum of support counts of frequent items), and then partition the database relatively even to avoid the generation of skewed partitions.

Note that our partition-based mining method shares some similarities with the *partitioned Apriori* method proposed by Savasere et al. [SON95]. In their paper, a transaction database is partitioned. Then, every partition is mined using *Apriori*. After that, all the locally frequent patterns are gathered to form a set of globally frequent candidate patterns. Finally, their global supports are counted by one more scan of the transaction database. However, there are two essential differences between their method and ours.

1. As also indicated in [SON95], it is not easy to get a good partition scheme using the *partitioned Apriori* [SON95] since it is hard to predict the space requirement of *Apriori*. In contrast, it is straightforward for *H-mine* to partition the transaction database, since the space overhead is very small and predictable during mining.

2. *H-mine* first finds globally frequent items. When mining partitions of a database, *H-mine* examines only those items which are globally frequent. In skewed partitions, many globally infrequent items can be locally frequent in some partitions, *H-mine* does not spend any effort to check them but the *partitioned Apriori* [SON95] does.

Furthermore, we can do better in consolidating globally frequent patterns from local ones. When mining a large transaction database, if the database is partitioned relatively even, it is expected that many short globally frequent patterns are frequent in every partition. In this case, a pattern frequent in every partition is a globally frequent pattern, and its global support count is the sum of the counts in all the partitions. *H-mine* does not need to test such patterns in its third scan. Therefore, in the third scan, *H-mine* checks only those locally frequent patterns which are infrequent in some partitions. Furthermore, a pattern is checked against only those partitions where it is infrequent.

We illustrate the idea in the following example.

**Example 4.2** A large transaction database $TDB$ is partitioned into four parts, $P_1$, $P_2$, $P_3$ and $P_4$. Let the support threshold be 100. The four parts are mined respectively using *H-mine(Mem)*. The locally frequent patterns as well as the partition-ids where they are frequent are shown in Table 4.2. The accumulated support count for a pattern is the sum of support counts from partitions where the pattern is locally frequent.

| Local frequent pattern | Partitions | Accumulated support count |
|---|---|---|
| $ab$ | $P_1, P_2, P_3, P_4$ | 280 |
| $ac$ | $P_1, P_2, P_3, P_4$ | 320 |
| $ad$ | $P_1, P_2, P_3, P_4$ | 260 |
| $abc$ | $P_1, P_3, P_4$ | 120 |
| $abcd$ | $P_1, P_4$ | 40 |
| … | … | … |

Table 4.2: Local frequent patterns in partitions.

1. Pattern *ab* is frequent in all the partitions. Therefore, it is globally frequent. Its global support count is its accumulated support count, i.e., 280. So do patterns *ac* and *ad*.

2. Pattern *abc* is frequent in all partitions except in $P_2$. The accumulated support count of *abc* covers the occurrences of the pattern in partitions $P_1$, $P_3$ and $P_4$. Thus, the pattern should be checked only in $P_2$. The global support count of *abc* is its accumulated count plus its support count in $P_2$. Similarly, pattern *abcd* need to be checked in only partitions $P_2$ and $P_3$.

3. In the third scan of *H-mine*, after scanning partition $P_2$, suppose the support count of pattern *abcd* in partition $P_2$ is 20. Since *abcd* is not frequent in partition $P_3$, its support count in $P_3$ must be less than the local support threshold. If the local support threshold is 30, we do not need to check pattern *abcd* in partition $P_3$, since *abcd* has no hope to be globally frequent. ■

As can be seen from the example, we have the following optimization methods on consolidating globally frequent patterns.

1. Accumulate the global support count from local ones for the patterns frequent in every partition.

2. Only check the patterns against those partitions where they are infrequent.

3. Use local support thresholds to derive the upper bounds for the global support counts of locally frequent patterns. Only check those patterns whose upper bound pass the global support threshold.

With the above optimization, the number of patterns to be consolidated can be reduced dramatically. As shown in our experiments, when the data set is relatively evenly distributed, only up to 20% of locally frequent patterns have to be checked in the third scan of *H-mine*.

In general, the following factors contribute to the scalability and efficiency of *H-mine*.

- As analyzed in Section 4.1, *H-mine(Mem)* has small space overhead and is efficient in mining partitions which can be held in main memory. With the current memory technology, it is likely that many medium-sized databases can be mined efficiently by this memory-based frequent-pattern mining mechanism.

- No matter how large the database is, it can be mined by at most three scans of the database: the first scan finds globally frequent items; the second mines partitioned database using *H-mine(Mem)*; and the third verifies globally frequent patterns. Since every partition is mined efficiently using *H-mine(Mem)*, the mining of the whole database is highly scalable.

- One may wonder that, since the *partitioned Apriori* [SON95] takes two scans of $TDB$, whereas *H-mine* takes three scans, how can *H-mine* outperform the one proposed in [SON95]? Notice that the major cost in this process is the mining of each partitioned database. The last scan of $TDB$ for collecting supports and generating globally frequent patterns is fast because the set of locally frequent patterns can be inserted into one compact structure, such as a hashing tree. Since *H-mine* generates less partitions and mines each partition very fast, it has better overall performance than the Apriori-based partition mining algorithm. This is also demonstrated in our performance study.

### 4.2.2 Handling dense data sets: Dynamic integration of H-struct and *FP-tree*-based mining

As indicated in several studies [BAG99, HPY00, PHM00], finding frequent patterns in dense databases is a challenging task since it may generate dense and long patterns which may lead to the generation of very large (and even exponential) number of candidate sets if an *Apriori*-like algorithm is used. The *FP-growth* method proposed in our recent study [HPY00] works well in dense databases with a large number of long patterns due to the effective compression of shared prefix paths in mining.

In comparison with *FP-growth*, *H-mine* does not generate physical projected databases and conditional *FP-tree*s and thus saves space as well as time in many cases. However, *FP-tree*-based mining has its advantages over mining on *H-struct* since *FP-tree* shares common prefix paths among different transactions, which may lead to space and time savings as well. As one may expect, the situation under which one method outperforms the other depends on the characteristics of the data sets: if data sharing is rare such as in sparse databases, the compression factor could be small and *FP-tree* may not outperform mining on *H-struct*. On the other hand, there are many dense data sets in practice. Even though the data sets might not be dense originally, as mining progresses, the projected databases become smaller, and data often becomes denser as the relative support goes up when the number of transactions in a projected database reduces substantially. In such cases, it is beneficial to swap the data structure from *H-struct* to *FP-tree* since *FP-tree*'s compression by common prefix path sharing and then mining on the compressed structures will overweigh the benefits brought by *H-struct*.

The question becomes what should be the appropriate situations that one structure is more preferable over the other and how to determine when such a structure/algorithm swapping should happen. A dynamic pattern density analysis technique is suggested as follows.

In the context of frequent pattern mining, a (projected) database is *dense* if the frequent items in it have *high relative support*. The *relative support* can be computed as follows:

$$\text{relative support} = \frac{\text{absolute support}}{\text{\# of tran (or freq-item projections) in the (projected) database}}.$$

When the relative support is high, such as 10% or over, i.e., the projected database is dense, and the number of (locally) frequent items is not large (so that the resulting *FP-tree* is not bushy), then *FP-tree* should be constructed to explore the sharing of common prefix paths and database compression. On the other hand, when the relative support of frequent items is low, such as far below 1%, it is sparse, and *H-struct* should be constructed for efficient *H-mine*. However, for relative support values in between, it is not clear which method would be more efficient.

With this discussion, one can see that Algorithm 5 (*H-mine*) should be modified as follows.

In Step 3 of Algorithm 5 which mines frequent patterns in each partition $TDB_i$, *H-mine(Mem)* is called. However, instead of simply constructing *H-struct* and mining the *H-struct* iteratively till the end, *H-mine(Mem)* will analyze the basic characteristics of data to determine whether *H-struct* should be constructed or utilized in the subsequent mining or whether *FP-tree*s should be constructed for frequent-pattern growth.

## 4.3 Performance Study and Experimental Results

To evaluate the efficiency and scalability of *H-mine*, we have performed an extensive performance study. In this section, we report our experimental results on the performance of *H-mine* in comparison with *Apriori* and *FP-growth*. It shows that *H-mine* outperforms

*Apriori* and *FP-growth* and is efficient and highly scalable for mining very large databases.[3]

All the experiments are performed on a 466MHz Pentium PC machine with 128 megabytes main memory and 20G hard disk, running Microsoft Windows/NT. *H-mine* and *FP-growth* are implemented by us using Visual C++6.0, while the version of *Apriori* that we used is a well-known version, "*GNU Lesser General Public License*" available at http://fuzzy.cs.uni-magdeburg.de/∼borgelt/. All reports of the runtime of *H-mine* include both the time of constructing *H-struct* and mining frequent-patterns. They also include both CPU time and I/O time.

We have tested various data sets, with consistent results. Limited by space, only the results on some typical data sets are reported here.

### 4.3.1 Mining transaction databases in main memory

In this sub-section, we report results on mining transaction databases which can be held in main memory. *H-mine* is implemented as stated in Section 4.1. For *FP-growth*, the *FP-tree*s can be held in main memory in the tests reported in this sub-section. We modified the source code for *Apriori* so that the transactions are loaded into main memory and the multiple scans of database are pursued in main memory.

Data set Gazelle is a sparse data set. It is a web store visit (click stream) data set from Gazelle.com. It contains $59,602$ transactions, while there are up to 267 item per transaction.

Figure 4.7 shows the run time of *H-mine*, *Apriori* and *FP-growth* on this data set. Clearly, *H-mine* wins the other two algorithms, and the gaps (in term of seconds) become larger as the support threshold goes lower.

*Apriori* works well in such sparse data sets since most of the candidates that *Apriori* generates turn out to be frequent patterns. However, it has to construct a hashing tree for the candidates and match them in the tree and update their counts each time when scanning a transaction that contains the candidates. That is the major cost for *Apriori*.

*FP-growth* has a similar performance as *Apriori* and sometime is even slightly worse. This is because when the database is sparse, *FP-tree* cannot compress data as effectively as what it does on dense data sets. Constructing *FP-tree*s over sparse data sets recursively has its overhead.

[3]A prototype of *H-mine* is also tested by a third party in US (a commercial company) on business data. Their results are consistent with ours. They observed that *H-mine* is more than 10 times faster than *Apriori* and other participating methods in their test when the support threshold is low.

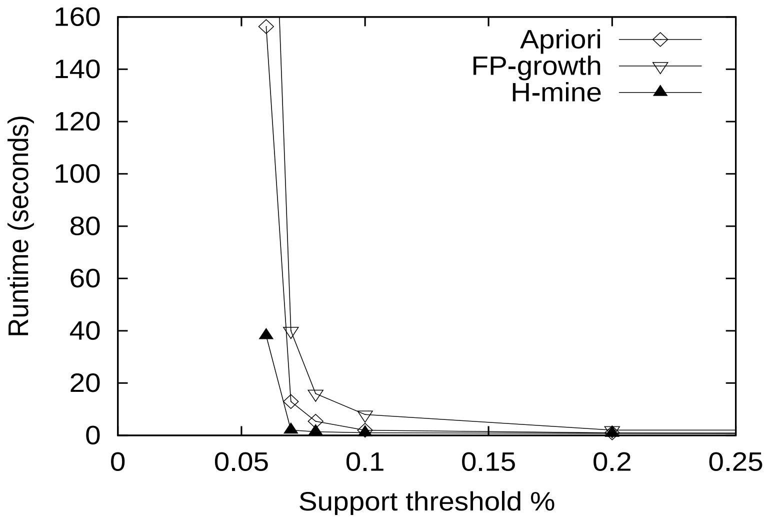

Figure 4.7: Runtime on data set Gazelle.

Figure 4.8 plots the high water mark of space usage of *H-mine*, *Apriori* and *FP-growth* in the mining procedure. To make the comparison clear, the space usage (axis Y) is in logarithmic scale. From the figure, we can see that *H-mine* and *FP-growth* use similar space and are very scalable in term of space usage with respect to support threshold. Even when the support threshold reduces to very low, the memory usage is still stable and moderate.

The memory usage of *Apriori* does not scale well as the support threshold goes down. *Apriori* has to store level-wise frequent patterns and generate next level candidates. When the support threshold is low, the number of frequent patterns as well as that of candidates are non-trivial. In contrast, pattern-growth methods, including *H-mine* and *FP-growth* , do not need to store any frequent patterns or candidates. Once a pattern is found, it is output immediately and never read back.

What are the performance of these algorithms over dense data sets? We use the synthetic data set generator described in [AS94] to generate a data set $T25I15D10k$. The data set generator has been used in many studies on frequent pattern mining. We refer readers to [AS94] for more details on the data set generation.

Data set $T25I15D10k$ contains $10,000$ transactions and each transaction has up to 25 items. There are $1,000$ items in the data set and the average longest potentially frequent

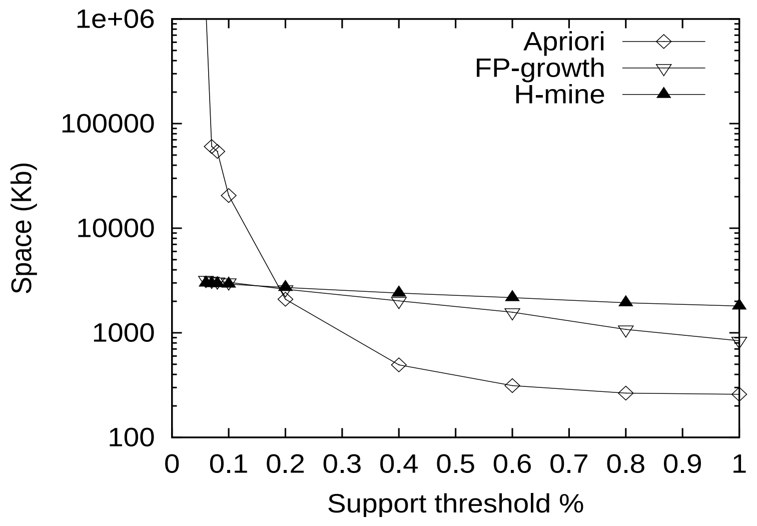

Figure 4.8: Space usage on data set Gazelle.

itemset is with 15 items. It is a relatively dense data set.

Figure 4.9 shows the runtime of the three algorithms on this data set. When the support threshold is high, most patterns are of short lengths, *Apriori* and *FP-growth* have similar performance. When the support threshold becomes low, most items (more than 90%) are frequent. Then, *FP-growth* is much faster than *Apriori*. In all cases, *H-mine* is the fastest one. It is more than 10 times faster than *Apriori* and 4-5 times faster than *FP-growth*.

Figure 4.10 shows the high water mark of space usage of the three algorithms in mining this data set. Again, the space usage is drawn in logarithmic scale. As the number of patterns goes up dramatically as support threshold goes down, *Apriori* requires an exponential amount of space. *H-mine* and *FP-growth* use stable amount of space. In dense data set, an *FP-tree* is smaller than the set of all frequent-item projections of the data set. However, long patterns means more recursions and more recursive *FP-tree*s. That makes *FP-growth* require more space than *H-mine* in this case. On the other hand, since the number of frequent items is large in this data set, an *FP-tree*, though compressing the database, still has many branches in various levels and becomes bushy. That also introduces non-trivial tree browsing cost.

Figure 4.11 and 4.12 explore the runtime per frequent pattern on data sets Gazelle and

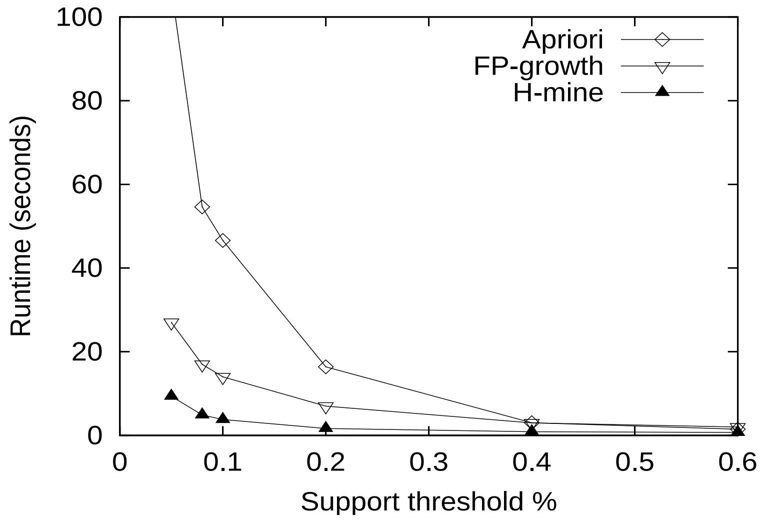

Figure 4.9: Runtime on data set $T25I15D10k$.

$T25I15D10k$, respectively. As the support threshold goes down, the number of frequent patterns goes up. As can be seen from the figures, the runtime per pattern of the three algorithms keeps going down. That explains the scalability of the three algorithms. Among the three algorithms, *H-mine* has the least runtime per pattern and thus has the best performance, especially when support threshold is low. The figures also illustrate that *H-mine* is scalable with respect to the number of frequent patterns.

In very dense data set, such as *Connect-4* (from UC-Irvine[4]) and *pumsb* (from IBM Almaden Research Center[5]), *H-mine* builds *FP-tree*s since the numbers of frequent items are very small. Thus it has the same performance as *FP-growth*. Previous studies, e.g., [Bay98], show that *Apriori* is incapable of mining such data sets.

## 4.3.2 Mining very large databases

To test the efficiency and scalability of the algorithms on mining very large databases, we generate data set $T25I15D1280k$ using the synthetic data generator. It has $1,280,000$

[4]www.ics.uci.edu/~mlearn/MLRepository.html

[5]www.almaden.ibm.com/cs/quest/demos.html

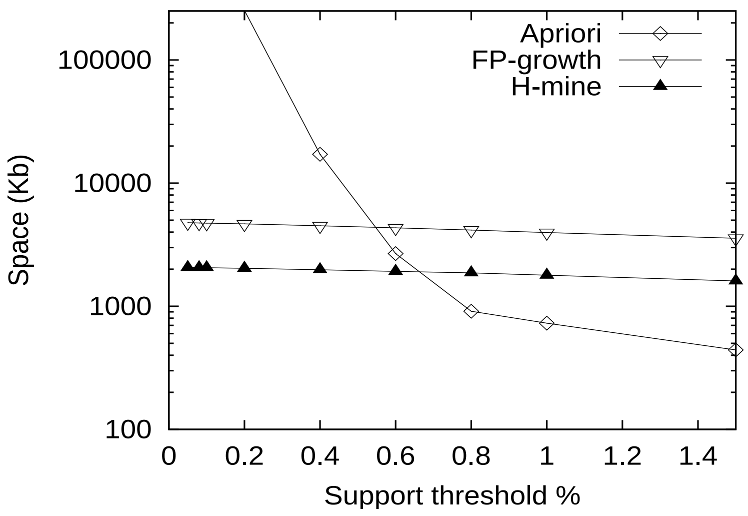

Figure 4.10: Space usage on data set $T25I15D10k$.

transactions with similar statistic features as the data set $T25I15D10k$.

We enforce memory constraints on *H-mine* so that the total memory available is limited to 2, 4, 8 and 16 megabytes, respectively. The memory covers the space for *H-struct* and all the header tables, as well as the related mechanisms. Since the *FP-tree* built for the data set is too big to fit in main memory, we do not report the performance of *FP-growth* on this data set. We do not explicitly compose any memory constraint on *Apriori*.

Figure 4.13 shows the scalability of both *H-mine* (with main memory size constraint 2 megabytes) and *Apriori* with respect to number of transactions in the database. Various support threshold settings are tested. Both algorithms have linear scalability and *H-mine* is a clear winner. From the figure, we can see that *H-mine* is more efficient and scalable at mining very large databases.

To study the effect of memory size constraint on the mining efficiency and scalability of *H-mine* in large databases, we plot Figure 4.14. The figure shows the scalability of *H-mine* with respect to support threshold with various memory constraints, i.e., 2, 4, 8 and 16 megabytes, respectively. As shown in the figure, the runtime is not sensitive to the memory limitation when support threshold is high. When the support threshold goes down, as available space increases, performance gets better.

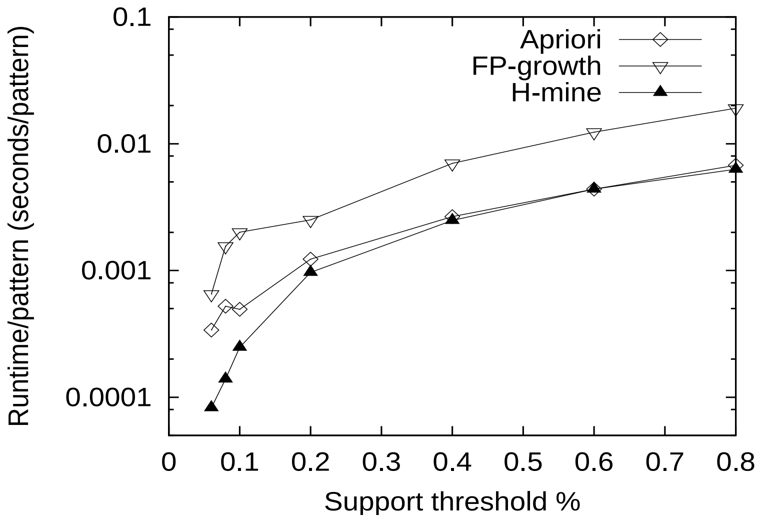

Figure 4.11: Runtime per pattern on data set Gazelle.

Figure 4.15 shows the effect of available memory size on mining large data sets. At high support level, the performance is not sensitive to the available memory size and thus the number of partitions. When the support threshold is low, the memory size plays an important role in performance.

With high support threshold, the number of frequent patterns is small and most frequent patterns are short. The dominant cost is I/O cost and thus is insensitive to size of available memory. When support threshold is low, with larger available memory, *H-mine* has less partitions and thus generates fewer locally frequent patterns, i.e., the locally frequent patterns contain more globally frequent ones and less noise. Therefore, *H-mine* can run faster with more memory. The results show that *H-mine* can fully utilize the available memory to scale up the mining process.

Does *H-mine* have to check all or most of the locally frequent patterns against the whole database in its third scan of the database? Fortunately, the answer is no. Our experimental results show that *H-mine* has a very light workload in its third scan. We consider the ratio of the number of patterns to be checked in the third scan over that of all distinct locally frequent patterns, where a locally frequent pattern is to be checked in the third scan if it is not frequent in every partition. Figure 4.16 shows the ratio numbers. In general, as

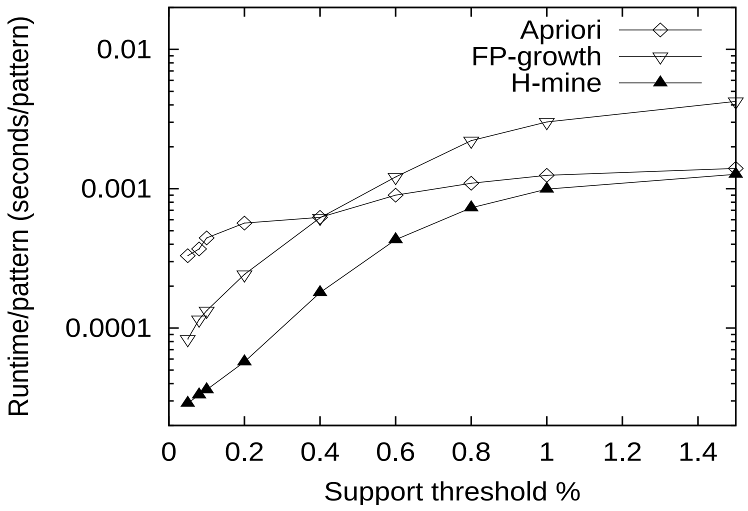

Figure 4.12: Runtime per pattern on data set $T25I15D10k$.

the support threshold goes down, the ratio goes up. That means mining with low support threshold may lead to more patterns frequent in some partitions. On the other hand, less memory (small partition) leads to more partitions and also increase the ratio.

As shown in the figure, only a limited portion of locally frequent patterns, e.g., less than 35% in our test case, needs to be tested in the third scan. This leads to a low cost of the third scan in our partition-based mining.

We also implemented a partition-based *Apriori* method. The partitioned *Apriori* cuts the database into even partitions, exactly as *H-mine* does. Then, it uses main memory based *Apriori* to mine each partition. We do not apply any main memory constraint to it on candidate generation, i.e., it can use as much available memory to store candidates and level-wise frequent patterns as it wants. The partitioned *Apriori* has up to 15% better performance than *Apriori* when support threshold is low, but still lose to *H-mine* with a wide margin. Limited by space, we omit a detailed performance report on partitioned *Apriori* here.

In summary, our experimental results and performance study verify our analysis and support our claim that *H-mine* is an efficient algorithm for mining frequent patterns. It is highly scalable in mining very large databases.

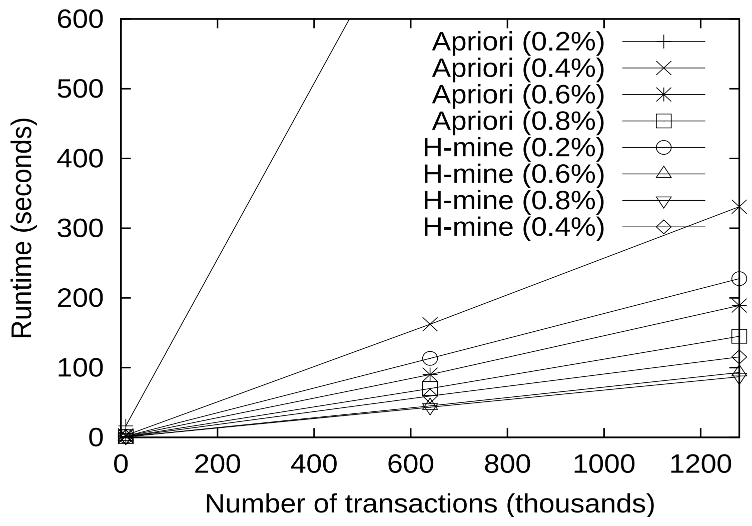

Figure 4.13: Scalability with respect to number of transactions.

## 4.4 Summary

In this chapter, we have proposed a simple and novel hyper-linked data structure, *H-struct*, and a new frequent pattern mining algorithm, *H-mine*, which takes advantage of *H-struct* data structure and dynamically adjusts links in the mining process. As shown in our performance study, *H-mine* has high performance, is scalable in all kinds of data, with very limited and predictable space overhead, and outperforms the previously developed algorithms with various settings.

In this section, we will discuss why *H-mine* has such high performance and what could be the impact of this new method.

First, a major distinction of *H-mine* from the previously proposed methods is that *H-mine* re-adjusts the links when mining different "projected" databases and has very small space overhead, even counting temporary working space; whereas candidate generation-and-test has to generate and test a large number of candidate itemsets, and *FP-growth* has to generate a good number of conditional (projected) databases and *FP-tree*s. The *structure- and space-preserving* philosophy of *H-mine* promotes the sharing of the existing structures in mining, reduces the cost of copying a large amount of data and building new data structures

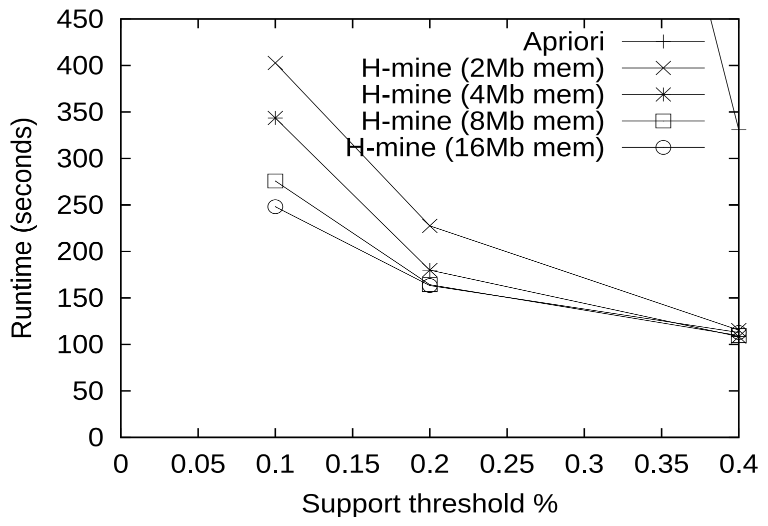

Figure 4.14: Scalability of *H-mine* on large data set $T25I15D1280k$.

on such data, and reduces the cost of updating and checking such data structures as well.

Second, *H-mine* absorbs the nice features of *FP-growth*. It is essentially a frequent-pattern growth approach since it partitions its search space according to both patterns and data based on a divide-and-conquer methodology, without generating and testing candidate patterns. However, unlike *FP-growth*, *H-mine* does not create any physical projected databases nor constructing conditional (local) *FP-tree*s. Instead, it builds and adjusts links dynamically among frequent items during mining to achieve the same effect as construction of physical projected databases. It avoids paying the cost of space and time for the (projected) database re-construction, and thus has better performance than *FP-growth*.

Third, *H-mine* is not confined itself to *H-struct* only. Instead, it watches carefully the changes of data characteristics during mining and dynamically switches its data structure from *H-struct* to *FP-tree* and its mining algorithm from mining on *H-struct* to *FP-growth* when the data set becomes dense and the number of frequent items becomes small. This absorbs the benefits of *FP-growth* which explores data compression and prefix-path shared mining. Since mining on *H-struct* and mining on *FP-tree* are built based on the same frequent-pattern growth methodology, such a dynamic algorithm swapping can be performed naturally and easily.

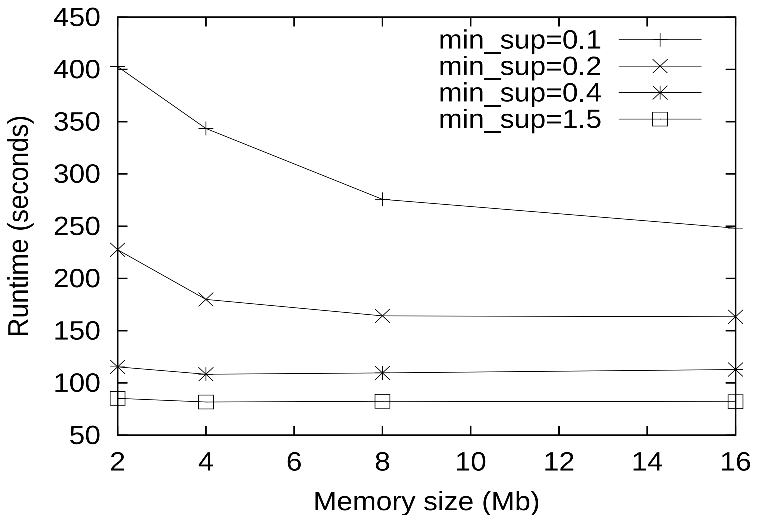

Figure 4.15: Effect of memory size on mining large data set.

Fourth, *H-mine* can be scaled-up to very large databases due to its small and precisely predictable run-time memory overhead and its database partition mining technique. *H-mine* partitions a large database into a set of relatively uniform-sized partitions, and mines each partition using *H-mine(Mem)*. Since mining each partition in main memory is highly efficient, and many mined patterns can be shared among uniformly partitioned databases to reduce the effort of pattern matching in the additional database scan, the overall cost is far less than other proposed methods, which has been demonstrated in our performance study.

Based on the above analysis, one can see that *H-mine* represents a new, highly efficient and scalable mining method. Its *structure- and space-preserving mining* methodology may have strong impact on the development of new, efficient and scalable data mining methods for mining other kinds of patterns, such as closed-itemsets [PHM00], max-patterns [Bay98], sequential patterns [SA96a, PHMA$^+$01], constraint-based mining [NLHP98, PHL01], etc. This should be an interesting direction for further study.

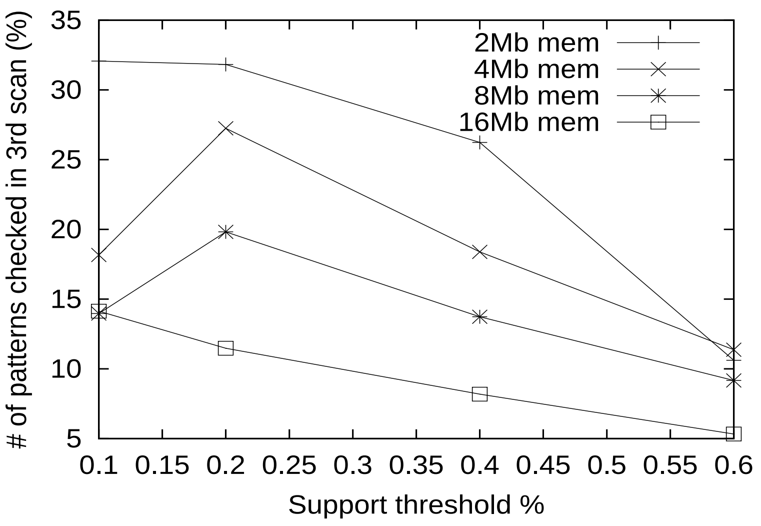

Figure 4.16: The ratio of patterns to be checked by *H-mine* in the third scan.

# Chapter 5

# Constraint-based Pattern-growth Mining

In Chapters 3 and 4, we have developed efficient pattern-growth methods to mine frequent patterns from large databases. Many applications need not only efficient but also effective frequent pattern mining techniques. In many cases, frequent pattern mining may return much more patterns than the users can understand and process. Integrating the users' interests into frequent pattern mining becomes an important issue.

Recent work has highlighted the importance of the constraint-based mining paradigm: the user is allowed to express her focus in mining, by means of a rich class of constraints that capture application semantics. Besides allowing user exploration and control, the paradigm allows many of these constraints to be pushed deep into mining, confining the search for patterns to those of users' interest; therefore, improving performance. *Metarules* or various kinds of *templates* have been proposed as filters to define the forms of rules to be mined [KMR$^+$94, SVA97]. Itemset constraints have been incorporated into association mining [SVA97]. A systematic method for the incorporation of two large classes of constraints, *anti-monotone* and *succinct*, in frequent itemset mining is presented in [NLHP98, LNHP99]. A method for mining association rules in large, dense databases by incorporating user-specified constraints to ensure every mined rule offers a predictive advantage over any of its simplifications, is developed in [BAG99]. Constraints specified using regular expressions are investigated for sequential pattern mining in [GRS99]. Constraint-based mining of correlations, by exploration of *anti-monotonicity* and *succinctness*, as well as *monotonicity*, is studied in

[GLW00].

While previous studies cover a large class of useful constraints, there are still many other useful and natural constraints that have not been considered. For example, consider the constraints $avg(S)\ \theta\ v$, $median(S)\ \theta\ v$, and $sum(S)\ \theta\ v$. The first two are neither anti-monotone, nor monotone, nor succinct. The last one is anti-monotone when $\theta$ is $\leq$ and *all items have non-negative values.* If $S$ contains items of arbitrary values, $sum(S) \leq v$ is rather like the first two constraints. Intuitively, this implies that such constraints are hard to optimize. In this chapter, we investigate a whole class of constraints that subsumes these examples. The main idea is that certain constraints which do not exhibit nice properties in general may do so in the presence of certain item ordering. We make the following contributions.

- We introduce (Section 5.2) the concept of *convertible constraints* and classify them into three classes: *convertible anti-monotone*, *convertible monotone*, and *strongly convertible.* This covers a good number of useful constraints which were previously regarded as tough, including all the examples above.

- We characterize (Section 5.2) the class of convertible constraints using the notion of *prefix monotone* functions, and study the arithmetical closure properties of such functions. As a byproduct, we can show a good class of constraints involving arithmetic are convertible. For example, we show that $max(S)/avg(S) \ \leq\ v$ is convertible anti-monotone and $median(S) - min(S) \ \geq\ v$ is convertible monotone.

- We show that convertible constraints cannot be pushed deep into the basic Apriori framework. However, they can be pushed deep into frequent pattern growth mining. We thus develop (Section 5.3) algorithms for fast mining of frequent itemsets satisfying the various constraints. We also discuss (Section 5.5) how multiple convertible constraints can be incorporated in fast frequent pattern mining.

- We report our results from a detailed set of experiments, which show the effectiveness of the algorithms developed (Section 5.4).

Our study distinguishes itself from the previous works on constraint-based frequent-pattern mining in the following aspects.

- As argued before, the previous works on constraint-based frequent-pattern mining that relied on properties like anti-monotonicity, succinctness, or monotonicity (e.g., [NLHP98, LNHP99, GLW00]) cannot handle the constraints studied in this chapter.
- There have been many studies on constraint-based search algorithms in artificial intelligence, such as [Web95, Rym92]. Our study is distinguished from theirs in two aspects: (i) we find the *complete set of frequent itemsets satisfying the constraints*, while their algorithms find *some* feasible solutions satisfying the constraints; and (ii) our goal is to find methods scalable in large databases, while their algorithms are mostly main memory-based.

Section 5.1 motivates the problem of frequent itemset mining with constraints. Section 5.6 concludes the chapter.

## 5.1 Problem Definition: Frequent Itemset Mining with Constraints

A *constraint* $C$ is a predicate on the powerset of the set of items $I$, i.e., $C : 2^I \rightarrow \{true, false\}$. An itemset $S$ satisfies a constraint $C$ if and only if $C(S)$ is true. The set of itemsets satisfying a constraint $C$ is $\textsc{sat}_C(I) = \{S \mid S \subseteq I \wedge C(S) = \text{ true}\}$. We call an itemset in $\textsc{sat}_C(I)$ *valid*.

**Problem definition**. Given a transaction database $\mathcal{T}$, a support threshold $\xi$, and a set of constraints $\mathcal{C}$, the problem of *mining frequent itemsets with constraints* is to find the complete set of frequent itemsets satisfying $\mathcal{C}$, i.e., find $\mathcal{F}_\mathcal{C} = \{S \mid S \in \textsc{sat}_\mathcal{C}(I) \wedge sup(S) \geq \xi\}$.

Many kinds of constraints can be associated with frequent itemset mining. Two categories of constraints, *succinctness* and *anti-monotonicity*, were proposed in [NLHP98, LNHP99]; whereas the third category, *monotonicity*, was studied in [BMS97, GLW00, PH00] in the contexts of mining correlated sets and frequent itemsets. We briefly recall these notions below.

**Definition 5.1 (Anti-monotone, Monotone, and Succinct Constraints)** A constraint $C_a$ is *anti-monotone* if and only if whenever an itemset $S$ violates $C_a$, so does any superset of $S$. A constraint $C_m$ is *monotone* if and only if whenever an itemset $S$ satisfies $C_m$, so does any superset of $S$. Succinctness is defined in steps, as follows.

- An itemset $I_s \subseteq I$ is a succinct set, if it can be expressed as $\sigma_p(I)$ for some selection predicate $p$, where $\sigma$ is the selection operator.
- $SP \subseteq 2^I$ is a succinct powerset, if there is a fixed number of succinct sets $I_1, I_2, \ldots, I_k \subseteq I$, such that $SP$ can be expressed in terms of the strict powersets of $I_1, \ldots, I_k$ using union and minus.
- Finally, a constraint $C_s$ is *succinct* provided $\textsc{sat}_{C_s}(I)$ is a succinct powerset. ■

We can show the following result.

**Theorem 5.1** *Every succinct constraint involving only aggregate functions can be expressed using conjunction and/or disjunction of monotone and anti-monotone constraints.*

**Proof.** The proof of the theorem is by induction on the structure of $SAT_C(I)$ of the succinct constraint, according to the definition of succinctness. There are three essential cases as follows.

- If $SAT_C(I) = 2^{I_c}$, where $I_c$ is a set, then $C$ is an anti-monotone constraint since any pattern satisfying the constraint must be a subset of $I_c$.
- If $SAT_C(I) = 2^{I_{c1}} \cup 2^{I_{c2}}$, then $C$ can be expressed in terms of $C = C_1 \vee C_2$, where $C_1$ and $C_2$ are corresponding anti-monotone constraints.
- If $SAT_C(I) = 2^{I_1} - 2^{I_2}$, then the constraint can be expressed as the conjunction of two constraints, $C = C_a \wedge C_m$, where $C_a$ is the anti-monotone constraint corresponding to $2^{I_1}$, and $C_m$ is a monotone constraint $S \cap (I_1 - I_2) \neq \emptyset$.

  Especially, if $SAT_C(I) = 2^I - 2^{I_1} - \cdots - 2^{I_m}$, then $C$ is a monotone constraint $S \cap (I - I_1 - \cdots - I_m) \neq \emptyset$. ■

These three categories of constraints cover a large class of popularly encountered constraints. A representative subset of commonly used, SQL-based constraints is listed in Table 5.1[1]. However, there are still many useful constraints, such as $avg(S)\ \theta\ v$ and $sum(S)\ \theta\ v$ where $\theta \in \{\leq, \geq\}$ (shown in the table) that belong to *none* of the three classes.

**Example 5.1** Let Table 5.2 be our running transaction database $\mathcal{T}$, with a set of items $I = \{a, b, c, d, e, f, g, h\}$. Let the support threshold be $\xi = 2$. Itemset $S = acd$ is frequent

[1]For brevity, we show a small subset of representative constraints, involving aggregates. See [NLHP98, LNHP99] for more details.

| Constraint | Anti-monotone | Monotone | Succinct |
|---|---|---|---|
| $min(S) \leq v$ | no | yes | yes |
| $min(S) \geq v$ | yes | no | yes |
| $max(S) \leq v$ | yes | no | yes |
| $max(S) \geq v$ | no | yes | yes |
| $count(S) \leq v$ | yes | no | weakly |
| $count(S) \geq v$ | no | yes | weakly |
| $sum(S) \leq v$ $(\forall a \in S,\ a \geq 0)$ | yes | no | no |
| $sum(S) \geq v$ $(\forall a \in S,\ a \geq 0)$ | no | yes | no |
| $sum(S)\ \theta\ v,\ \theta \in \{\leq, \geq\}$ $(\forall a \in S,\ a\ \theta\ 0)$ | no | no | no |
| $range(S) \leq v$ | yes | no | no |
| $range(S) \geq v$ | no | yes | no |
| $avg(S)\ \theta\ v,\ \theta \in \{\leq, \geq\}$ | no | no | no |
| $sup(S) \geq \xi$ | yes | no | no |
| $sup(S) \leq \xi$ | no | yes | no |

Table 5.1: Characterization of commonly used, SQL-based constraints.

| Transaction ID | Items in transaction |
|---|---|
| 10 | $a, b, c, d, f$ |
| 20 | $b, c, d, f, g, h$ |
| 30 | $a, c, d, e, f$ |
| 40 | $c, e, f, g$ |

Table 5.2: The transaction database $\mathcal{T}$ in Example 5.1.

since it is in transactions 10 and 30, respectively. The complete set of frequent itemsets are listed in Table 5.3.

| Length $l$ | Frequent $l$-itemsets |
|---|---|
| 1 | $a, b, c, d, e, f, g$ |
| 2 | $ac, ad, af, bc, bd, bf, cd, ce, cf, cg, df, ef, fg$ |
| 3 | $acd, acf, adf, bcd, bcf, bdf, cdf, cef, cfg$ |
| 4 | $acdf, bcdf$ |

Table 5.3: Frequent itemsets with support threshold $\xi = 2$ in transaction database $\mathcal{T}$ in Table 5.2.

Let each item have an attribute *value* (such as *profit*), with the concrete value shown in Table 5.4. In all constraints such as $sum(S)\ \theta\ v$, we implicitly refer to this value.

The constraint $range(S) \leq 15$ requires that for an itemset $S$, the value range of the items in $S$ must be no greater than 15. It is an anti-monotone constraint, in the sense that if an itemset, say $ab$, violates the constraint, any of its supersets will violate it; and thus $ab$ can be removed safely from the candidate set during an Apriori-like frequent itemset mining

| Item | Value |
|---|---|
| a | 40 |
| b | 0 |
| c | −20 |
| d | 10 |
| e | −30 |
| f | 30 |
| g | 20 |
| h | −10 |

Table 5.4: The values (such as profit) of items in Example 5.1.

process [NLHP98]. However, the constraint $C_{avg} \equiv avg(S) \geq 25$ is not anti-monotone (nor monotone, nor succinct, which can be verified by readers). For example, $avg(df) = (10{+}30)/2 < 25$, violates the constraint. However, upon adding one more item $a$, $avg(adf) = (40 + 10 + 30)/3 \geq 25$, $adf$ satisfies $C_{avg}$. ■

This example scratches the surface of a large class of useful constraints involving *avg*, *median*, etc. as well as arithmetic. Exploiting them in mining calls for new techniques, which is the subject of this chapter.

## 5.2 Convertible Constraints and Their Classification

Before introducing the concept of convertible constraint, we motivate it with an example.

**Example 5.2** Suppose we wish to mine frequent itemsets over transaction database $\mathcal{T}$ in Table 5.2, with the support threshold $\xi = 2$ and with constraint $C \equiv avg(S) \geq 25$,

The complete set of frequent itemsets satisfying $C$ can be obtained by first mining the frequent itemsets without using the constraint (i.e., Table 5.3) and then filtering out those not satisfying the constraint. Since the constraint is neither anti-monotone, nor monotone, nor succinct, it cannot be directly incorporated into an Apriori-style algorithm. E.g., itemset $fg$ satisfies the constraint, while its subset $g$ and its superset $dfg$ do not.

If we arrange the items in *value-descending* order, $\langle a, f, g, d, b, h, c, e \rangle$, we can observe an interesting property, as follows. Writing itemsets w.r.t. this order leads to a notion of a prefix. E.g., $afd$ has $af$ and $a$ as its prefixes. Interestingly, the average of an itemset is no more than that of its prefix, according to this order. ■

### 5.2.1 Convertible Constraints

The observation made in Example 5.2 motivates the following definition. We will frequently make use of an order[2] over the set of all the items and assume itemsets are written according to this order.

**Definition 5.2 (Prefix itemset)** Given an order $\mathcal{R}$ over the set of items $I$, an itemset $S' = i_1 i_2 \cdots i_l$ is called a *prefix* of itemset $S = i_1 i_2 \cdots i_m$ w.r.t. $\mathcal{R}$, where items in both itemsets are listed according to order $\mathcal{R}$ and $(l \leq m)$. $S'$ is called a *proper prefix* of $S$ if $(l < m)$. ■

We next formalize convertible constraints as follows.

**Definition 5.3 (Convertible Constraints)** A constraint $C$ is *convertible anti-monotone* provided there is an order $\mathcal{R}$ on items such that whenever an itemset $S$ satisfies $C$, so does any prefix of $S$. It is *convertible monotone* provided there is an order $\mathcal{R}$ on items such that whenever an itemset $S$ violates $C$, so does any prefix of $S$. A constraint is *convertible* whenever it is convertible anti-monotone or monotone. ■

Note that any anti-monotone (resp., monotone) constraint is trivially convertible anti-monotone (resp., convertible monotone): just pick any order on items.

**Example 5.3** We show $avg(S)\ \theta\ v$ where $\theta \in \{\leq, \geq\}$ is a convertible constraint.

Let $\mathcal{R}$ be the value-descending order. Given an itemset $S = a_1 a_2 \cdots a_l$ satisfying the constraint $avg(S) \geq v$, where items in $S$ are listed in the order $\mathcal{R}$. For each prefix $S' = a_1 \cdots a_k$ of $S$ $(1 \leq k \leq l)$, since $a_k \geq a_{k+1} \geq \cdots \geq a_{l-1} \geq a_l$, we have $avg(S') \geq avg(S' \cup \{a_{k+1}\}) \geq \cdots \geq avg(S) \geq v$. This implies $S'$ also satisfies the constraint. So, constraint $avg(S) \geq v$ is convertible anti-monotone. Similarly, it can be shown that constraint $avg(S) \leq v$ is convertible monotone.

Interestingly, if the order $\mathcal{R}^{-1}$ (i.e., the reversed order of $\mathcal{R}$) is used, the constraint $avg(S) \geq v$ can be shown convertible monotone. We leave this as an exercise to the reader.

In summary, constraint $avg(S)\ \theta\ v$ is convertible constraint. Furthermore, there exists an order $\mathcal{R}$ such that the constraint is convertible anti-monotone w.r.t. $\mathcal{R}$ and convertible monotone w.r.t. $\mathcal{R}^{-1}$. ■

[2]Unless otherwise stated, every order used in this chapter is assumed to be total over the set of items.

As another example, let us examine the constraints with function $sum(S)$.

**Example 5.4** As shown in Table 5.1, constraint $sum(S) \leq v$ is anti-monotone if items are all with non-negative values. However, if items are with negative, zero or positive values, the constraint becomes neither anti-monotone, nor monotone, nor succinct.

Interestingly, this constraint exhibits a "piecewise" convertible monotone or anti-monotone behavior. If $v \geq 0$ in the constraint, the constraint is convertible anti-monotone w.r.t. item value *ascending* order. Given an itemset $S = a_1 a_2 \cdots a_l$ such that $sum(S) \leq v$, where items are listed in value ascending order. For a prefix $S' = a_1 a_2 \cdots a_j$ $(1 \leq j \leq l)$, if $a_j \leq 0$, that means $a_1 \leq a_2 \leq \cdots \leq a_{j-1} \leq a_j \leq 0$. So, $sum(S') \leq 0 \leq v$. On the other hand, if $a_j > 0$, we have $0 < a_j \leq a_{j+1} \leq \cdots \leq a_l$. Thus, $sum(S') = sum(S) - sum(a_{j+1} \cdots a_l) < v$. Therefore, $sum(S') \leq v$ in both cases, which means $S'$ satisfies the constraint.

If $v \leq 0$ in the constraint, it becomes convertible monotone w.r.t. item value descending order. We leave it to the reader to verify this.

Similarly, we can also show that, if items are with negative, zero or positive values, constraint $sum(S) \geq v$ is convertible monotone w.r.t. value ascending order when $v \geq 0$, and convertible anti-monotone w.r.t. value descending order when $v \leq 0$. ■

The following lemma can be proved with a straightforward induction.

**Lemma 5.1** *Let $C$ be a constraint over a set of items $I$.*

1. *$C$ is convertible anti-monotone if and only if there exists an order $\mathcal{R}$ over $I$ such that for every itemset $S$ and item $a \in I$ such that $\forall x \in S, x \mathcal{R} a$, $C(S \cup \{a\})$ implies $C(S)$.*

2. *$C$ is convertible monotone if and only if there exists an order $\mathcal{R}$ over $I$ such that for every itemset $S$ and item $a \in I$ such that $\forall x \in S, x \mathcal{R} a$, $C(S)$ implies $C(S \cup \{a\})$.*

**Proof.** We show the first part of the lemma. The second part can be shown similarly.

$\Rightarrow$ *(if part)* Suppose constraint $C$ has the property that for every itemset $S$ and item $a \in I$ such that item $\forall x \in S, x \mathcal{R} a$, $C(S \cup \{a\})$ implies $C(S)$. For an itemset $S = a_1 a_2 \cdots a_m$ and its prefix $S' = a_1 a_2 \cdots a_l$ $(l \leq m)$, let $S_k$ be itemset $a_1 a_2 \cdots a_k$. $C(S) = C(S_{m-1} \cup \{a_m\}) = true$ implies $C(S_{m-1}) = true$. By induction, we can show that $C(S') = C(S_l) = true$. Thus, $C$ is convertible anti-monotone.

$\Leftarrow$ *(only-if part)* Given a convertible anti-monotone constraint $C$, following the definition of convertible anti-monotonicity, the property holds that for every itemset $S$ and item $a \in I$ such that item $\forall x \in S, x \mathcal{R} a$, $C(S \cup \{a\})$ implies $C(S)$. ■

The notion of prefix monotone functions, introduced below, is helpful in determining the class of a constraint. We denote the set of real numbers as $\mathbf{R}$.

**Definition 5.4 (Prefix monotone functions)** Given an order $\mathcal{R}$ over a set of items $I$, a function $f : 2^I \rightarrow \mathbf{R}$ is a *prefix (monotonically) increasing function* w.r.t. $\mathcal{R}$ if and only if for every itemset $S$ and its prefix $S'$ w.r.t. $\mathcal{R}$, $f(S') \leq f(S)$. A function $g : 2^I \rightarrow \mathbf{R}$ is called a *prefix (monotonically) decreasing function* w.r.t. $\mathcal{R}$ if and only if for every itemset $S$ and its prefix $S'$ w.r.t. $\mathcal{R}$, $g(S') \geq g(S)$. ■

We have the following lemma on the determination of prefix monotone functions. The proof is similar to that of Lemma 5.1.

**Lemma 5.2** *Given an order $\mathcal{R}$ over a set of items $I$,*

1. *a function $f : 2^I \rightarrow \mathbf{R}$ is a prefix decreasing function w.r.t. $\mathcal{R}$ if and only if for every itemset $S$ and item $a$ such that $\forall x \in S, x\ \mathcal{R}\ a$, $f(S) \geq f(S \cup \{a\})$.*

2. *A function $g : 2^I \rightarrow \mathbf{R}$ is a prefix increasing function w.r.t. $\mathcal{R}$ if and only if for every itemset $S$ and item $a$ such that $\forall x \in S, x\ \mathcal{R}\ a$, $g(S) \leq g(S \cup \{a\})$.*

**Proof.** We show the first part of the lemma. The second part can be proved similarly.
$\Leftarrow$ Let $f : 2^I \rightarrow \mathbf{R}$ be a prefix decreasing function. Itemset $S$ is a prefix of $S \cup \{a\}$ if $\forall x \in S, x\ \mathcal{R} a$. According to the definition of prefix decreasing function, we have $f(S) \geq f(S \cup \{a\})$.
$\Rightarrow$ Suppose function $f : 2^I \rightarrow \mathbf{R}$ has the property that for every itemset $S$ and item $a$ such that $\forall x \in S, x\ \mathcal{R}\ a$, $f(S) \geq f(S \cup \{a\})$. For an itemset $S = a_1 a_2 \cdots a_m$ and its prefix $S' = a_1 a_2 \cdots a_l$ $(l \leq m)$, we have $f(S') = f(a_1 a_2 \cdots a_l) \leq f(a_1 a_2 \cdots a_l a_{l+1}) \leq \cdots \leq f(a_1 a_2 \cdots a_m) = f(S)$. So, $f$ is a prefix decreasing function. ■

It turns out that prefix monotone functions satisfy interesting closure properties with arithmetic. An understanding of this would shed light on characterizing a whole class of convertible functions involving arithmetic. The following theorem establishes the arithmetical closure properties of prefix monotone functions. We say a function $f : 2^I \rightarrow \mathbf{R}$ is positive, provided $\forall S \subseteq I : f(S) > 0$.

**Theorem 5.2** *Let $f$ and $f'$ be prefix decreasing functions, and $g$ and $g'$ be prefix increasing functions w.r.t. an order $\mathcal{R}$, respectively. Let $c$ be a positive real number.*

1. *Functions $-f(S)$, $\frac{1}{f(S)}$, $c \cdot g(S)$ and $g(S)+g'(S)$ are prefix increasing functions. Functions $-g(S)$, $\frac{1}{g(S)}$, $c \cdot f(S)$ and $f(S)+f'(S)$ are prefix decreasing functions.*

2. *If $f$ and $g$ are positive functions, then $f(S) \times f'(S)$ is prefix decreasing, and $g(S) \times g'(S)$ is prefix increasing.*

3. *A constraint $h(S) \geq v$ (resp., $h(S) \leq v$) is convertible anti-monotone (resp., monotone) if and only if $h$ is prefix decreasing. Similarly, $h(S) \geq v$ (resp., $h(S) \leq v$) is convertible monotone (resp., anti-monotone) if and only if $h$ is prefix increasing.*

**Proof.** The theorem follows related definitions immediately. ■

**Example 5.5** As an illustration, notice that $avg(S)$ is a prefix decreasing function w.r.t. value-descending order, and $avg(S) \geq 20$ is convertible anti-monotone w.r.t. the same order. Also, $max(S)$ is a prefix increasing[3] function w.r.t. this order. From Theorem 5.2, it follows that $1/avg(S)$ is prefix increasing and hence $max(S)/avg(S)$ is prefix increasing.[4] Consequently, we immediately deduce that $max(S)/avg(S) \leq v$ is convertible anti-monotone w.r.t. this order. ■

We know from Theorem 5.1 that a succinct constraint can be expressed in terms of conjunction and/or disjunction of anti-monotone and monotone constraints. By definition, every monotone/anti-monotone is convertibly so. A natural question is, what is the relationship between succinct constraints and convertible constraints? The following theorem settles this question.

**Theorem 5.3** *Every succinct constraint is either anti-monotone, or monotone, or convertible.*

**Proof.** The proof of the theorem is by induction on the structure of $\text{SAT}_C(I)$ of a succinct constraint $C$, according to the definition of succinctness.

Suppose $C$ is a succinct constraint over $I$, the set of items.

- $\mathcal{F}_C(I) = 2^{I_C}$, *where* $I_C \subseteq I$. As shown in Theorem 5.1, constraint $C$ is anti-monotone. Interestingly, $C$ is also convertible monotone w.r.t. order $\mathcal{R}$, where $\forall a \in I_C, b \in I - I_C$, $b \, \mathcal{R} \, c$.

[3]It is also prefix decreasing w.r.t. this order.

[4]Assuming all the items have non-negative values.

- $\mathcal{F}_C(I) = 2^{I_1} \cup 2^{I_2}$, *where* $I_1, I_2 \subseteq I$. Constraint $C$ is anti-monotone (Theorem 5.1). However, $C$ is not convertible monotone in this case.
- $\mathcal{F}_C(I) = 2^{I_1} - 2^{I_2}$, *where* $I_1, I_2 \subseteq I$. Constraint $C$ is convertible anti-monotone w.r.t. order $\mathcal{R}$, where $\forall a \in I_1 - I_2, b \in I - (I_1 - I_2)$, $a \; \mathcal{R} \; b$. Please note that $C$ is also convertible monotone w.r.t. $\mathcal{R}^{-1}$.

  Especially, $\mathcal{F}_C(I) = 2^I - 2^{I_1} - \cdots - 2^{I_m}$, where $I_1, \ldots, I_m \subseteq I$. Constraint $C$ is monotone (Theorem 5.1). ■

As an example, consider a succinct constraint $C$ whose solution space $\text{SAT}_C(I)$ is described as $2^{I_1} - 2^{I_2}$, where $I_1, I_2 \subseteq I$, and $I_i = \sigma_{p_i}(I)$, $p_i$ being a selection predicate, $i = 1, 2$. Consider an order $\mathcal{R}$ such that all the items in $I_1 - I_2$ come before any item in $I - (I_1 - I_2)$, but otherwise the items are ordered arbitrarily. Then, it is easy to see that w.r.t. $\mathcal{R}$, $C$ is convertible anti-monotone and w.r.t. $\mathcal{R}^{-1}$, it is convertible monotone.

### 5.2.2 Strongly convertible constraint

Some convertible constraints have the additional desirable property that w.r.t. an order $\mathcal{R}$ they are convertible anti-monotone, while w.r.t. its inverse $\mathcal{R}^{-1}$ they are convertible monotone. E.g., $avg(S) \leq v$ is convertible monotone w.r.t. value ascending order and convertible anti-monotone w.r.t. value descending order (see also Example 5.3). This property provides great flexibility in data mining query optimization.

**Definition 5.5 (Strongly convertible constraint)** A constraint $C_{sc}$ is called a *strongly convertible constraint*, provided there exists an order $\mathcal{R}$ over the set of items such that $C_{sc}$ is convertible anti-monotone w.r.t. $\mathcal{R}$ and convertible monotone w.r.t. $\mathcal{R}^{-1}$. ■

Notice that $median(S) \; \theta \; v$ ($\theta \in \{\leq, \geq\}$) is also strongly convertible. Clearly, not every convertible constraint is strongly convertible. E.g., $max(S)/avg(S) \leq v$ [5] is convertible anti-monotone w.r.t. value descending order, when all the items have a non-negative value. However, it is not convertible monotone w.r.t. value ascending order.

The following lemma links strongly convertible constraints to prefix monotone functions.

[5] It says the proportion of the max price of any item in the itemset over the average price of the items in the set cannot go over certain limit.

**Lemma 5.3** *Constraint $f(S)\ \theta\ v$ is strongly convertible, if and only if there exists an order $\mathcal{R}$ over the set of items such that $f$ is a prefix decreasing function w.r.t. $\mathcal{R}$ and a prefix increasing function w.r.t. $\mathcal{R}^{-1}$.*

**Proof.** The lemma follows Theorem 5.2 immediately. ■

For example, $avg(S)$ and $median(S)$ are both prefix decreasing w.r.t. value descending order and prefix increasing w.r.t. value ascending order.

There still exist some constraints that cannot be pushed by item ordering. For example, the constraint $avg(S) - median(S) = 0$[6] does not admit any natural ordering on items w.r.t. which it is convertible. We call such constraints *inconvertible*.

### 5.2.3 Summary: a classification on constraints

As a general picture, constraints (only involving aggregate functions) can be classified into the following categories according to their interactions with the frequent itemset mining process: *anti-monotone, monotone, succinct* and *convertible*, which in turn can be subdivided into *convertible anti-monotone* and *convertible monotone*. The intersection of the last two categories is precisely the class of *strongly convertible* constraints (which can be treated either as convertible anti-monotone or monotone by ordering the items properly). Figure 5.1 shows the relationship among the various classes of constraints.

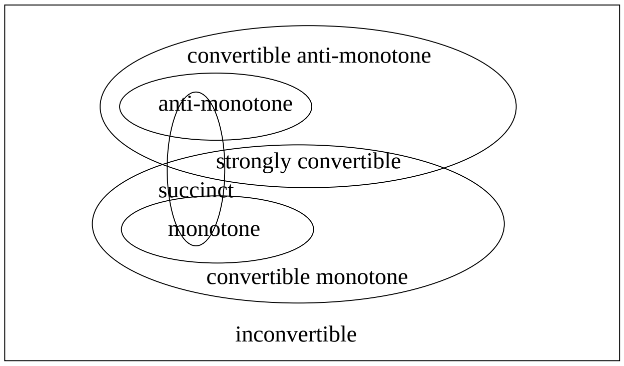

Figure 5.1: A classification of constraints and their relationships

Some commonly used convertible constraints are listed in Table 5.5.

[6]The constraint requires that the median item in the itemset is equal to the average value.

| Constraint | Convertible anti-monotone | Convertible monotone | Strongly convertible |
|---|---|---|---|
| $avg(S)\ \theta\ v\ (\theta \in \{\leq, \geq\})$ | yes | yes | yes |
| $median(S)\ \theta\ v\ (\theta \in \{\leq, \geq\})$ | yes | yes | yes |
| $sum(S) \leq v\ (v \geq 0, \forall a \in S, a\vartheta 0, \theta, \vartheta \in \{\leq, \geq\})$ | yes | no | no |
| $sum(S) \leq v\ (v \leq 0, \forall a \in S, a\vartheta 0, \theta, \vartheta \in \{\leq, \geq\})$ | no | yes | no |
| $sum(S) \geq v\ (v \geq 0, \forall a \in S, a\vartheta 0, \theta, \vartheta \in \{\leq, \geq\})$ | no | yes | no |
| $sum(S) \geq v\ (v \leq 0, \forall a \in S, a\vartheta 0, \theta, \vartheta \in \{\leq, \geq\})$ | yes | no | no |
| $f(S) \geq v$ ($f$ is a prefix decreasing function) | yes | $*$ | $*$ |
| $f(S) \geq v$ ($f$ is a prefix increasing function) | $*$ | yes | $*$ |
| $f(S) \leq v$ ($f$ is a prefix decreasing function) | $*$ | yes | $*$ |
| $f(S) \leq v$ ($f$ is a prefix increasing function) | yes | $*$ | $*$ |

Table 5.5: Characterization of some commonly used, SQL-based convertible constraints. ($*$ means it depends on the specific constraint.)

## 5.3 Mining Algorithms

In this section, we explore how to mine frequent itemsets with convertible constraints efficiently. The general idea is to push the constraint into the mining process as deep as possible, thereby pruning the search space.

In Section 5.3.1, we first argue that the *Apriori* algorithm cannot be extended to mining with convertible constraints efficiently. Then, a new method is proposed by examining an example. Section 5.3.2 presents the algorithm $\mathcal{FIC}^{\mathcal{A}}$ for mining frequent itemsets with convertible anti-monotone constraints. Algorithm $\mathcal{FIC}^{\mathcal{M}}$, which computes the complete set of frequent itemsets with convertible monotone constraint, is given in Section 5.3.3. Section 5.3.4 discusses mining frequent itemsets with strongly convertible constraints.

### 5.3.1 Mining frequent itemsets with convertible constraints: An example

We first show that convertible constraints cannot be pushed deep into the *Apriori*-like mining.

**Remark 5.3.1** *A convertible constraint that is neither monotone, nor anti-monotone, nor succinct, cannot be pushed deep into the* Apriori *mining algorithm.*

**Rationale.** As observed earlier for such a constraint (e.g., $avg(S) \leq v$), subsets (supersets) of a valid itemset could well be invalid and vice versa. Thus, within the level-wise framework, no direct pruning based on such a constraint can be made. In particular, whenever an invalid subset is eliminated without support counting, its supersets that are not suffixes cannot be pruned using frequency.

For example, itemset *df* in our running example violates the constraint $avg(S) \geq 25$. However, an *Apriori*-like algorithm cannot prune such itemsets. Otherwise, its superset *adf*, which satisfies the constraint, cannot be generated. ■

Before giving our algorithms for mining with convertible constraints, we give an overview in the following example.

**Example 5.6** Let us mine frequent itemsets with constraint $C \equiv avg(S) \geq 25$ over transaction database $\mathcal{T}$ in Table 5.2, with the support threshold $\xi = 2$. Items in every itemset are listed in value descending order $\mathcal{R}$: $\langle a(40), f(30), g(20), d(10), b(0), h(-10), c(-20), e(-30)\rangle$. It is shown that constraint $C$ is *convertible anti-monotone* w.r.t. $\mathcal{R}$. The mining process is shown in Figure 5.2.

By scanning $\mathcal{T}$ once, we find the support counts for every item. Since $h$ appears in only one transaction, it is an infrequent items and is thus dropped without further consideration. The set of frequent 1-itemsets are $a$, $f$, $g$, $d$, $b$, $c$ and $e$, listed in order $\mathcal{R}$. Among them, only $a$ and $f$ satisfy the constraint[7]. Since $C$ is a convertible anti-monotone constraint, itemsets having $g$, $d$, $b$, $c$ or $e$ as prefix cannot satisfy the constraint. Therefore, the set of frequent itemsets satisfying the constraint can be partitioned into two subsets:

1. The ones having itemset $a$ as a prefix w.r.t. $\mathcal{R}$, i.e., those containing item $a$; and
2. The ones having itemset $f$ as a prefix w.r.t. $\mathcal{R}$, i.e., those containing item $f$ but no $a$.

The two subsets form two projected databases [HPY00] which are mined respectively.

1. *Find frequent itemsets satisfying the constraint and having $a$ as a prefix.* First, $a$ is a frequent itemset satisfying the constraint. Then, the frequent itemsets having $a$ as a proper prefix can be found in the subset of transactions containing $a$, which is called *$a$-projected database*. Since $a$ appears in every transaction in the $a$-projected database, it is omitted. The $a$-projected database contains two transactions: *bcdf* and *cdef*. Since items $b$ and $e$ are infrequent within this projected database, neither *ab* nor *ae* can be frequent. So, they are pruned. The frequent items in the $a$-projected

[7]The fact that itemset $g$ does not satisfy the constraint implies none of any 1-itemsets after $g$ in order $\mathcal{R}$ can satisfy the constraint *avg*.

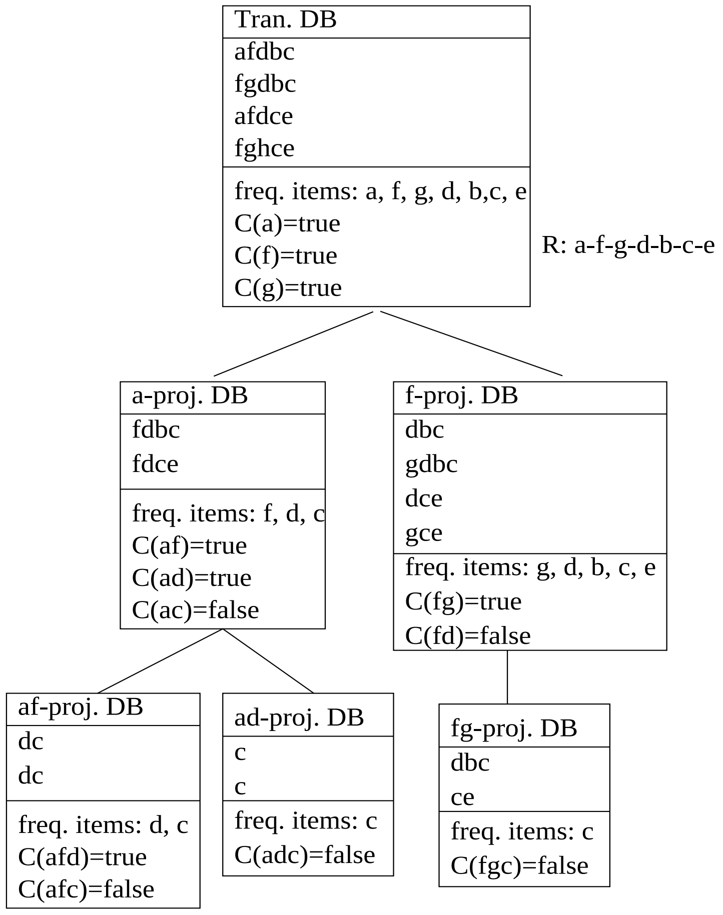

Figure 5.2: Mining frequent itemsets satisfying constraint $avg(S) \geq 25$.

database is $f, d, c$, listed in the order $\mathcal{R}$. Since $ac$ does not satisfy the constraint, there is no need to create an $ac$-projected database.

To check what can be mined in the $a$-projected database with $af$ and $ad$, as prefix, respectively, we need to construct the two projected databases and mine them. This process is similar to the mining of $a$-projected databases. The *$af$-projected database* contains two frequent items $d$ and $c$, and only $afd$ satisfies the constraint. Moreover, since $afdc$ does not satisfy the constraint, the process in this branch is complete. Since $afc$ violates the constraint, there is no need to construct $afc$-projected database. The *$ad$-projected database* contains one frequent item $c$, but $adc$ does not satisfy the constraint. Therefore, the set of frequent itemsets satisfying the constraint and having $a$ as prefix contains $a$, $af$, $afd$, and $ad$.

2. *Find frequent itemsets satisfying the constraint and having $f$ as a prefix.* Similarly, the *$f$-projected database* is the subset of transactions containing $f$, with both $a$ and $f$ removed. It has four transactions: $bcd$, $bcdg$, $cde$ and $ceg$. The frequent items in the projected database are $g, d, b, c, e$, listed in the order of $\mathcal{R}$. Since only itemsets $fg$ and $fd$ satisfy the constraint, we only need to explore if there is any frequent itemset with $fg$ or $fd$ as a proper prefix that satisfies the constraint. The *projected $fg$-database* contains no frequent itemset with $fg$ as a proper prefix that satisfies the constraint. Since $b$ is the item immediately after $d$ in order $\mathcal{R}$, and $fdb$ violates the constraint, any itemset with $fd$ as a proper prefix cannot satisfy the constraint. Thus, $f$ and $fg$ are the only two frequent itemsets having $f$ as a prefix and satisfying the constraint.

In summary, the complete set of frequent itemsets satisfying the constraint contains 6 itemsets: $a$, $f$, $af$, $ad$, $afd$, $fg$. Our new method generates and tests only a small set of itemsets. ■

### 5.3.2 $\mathcal{FIC}^{\mathcal{A}}$: Mining frequent itemsets with convertible anti-monotone constraint

Now, let us justify the correctness and completeness of the mining process in Example 5.6.

First, we show that the complete set of frequent itemsets satisfying a given convertible anti-monotone constraint can be partitioned into several non-overlapping subsets. It leads to the soundness of our algorithmic framework.

**Lemma 5.4** *Consider a transaction database $\mathcal{T}$, a support threshold $\xi$ and a convertible anti-monotone constraint $C$ w.r.t. an order $\mathcal{R}$ over a set of items $I$. Let $a_1, a_2, \ldots, a_m$ be the items satisfying $C$. The complete set of frequent itemsets satisfying $C$ can be partitioned into $m$ disjoint subsets: the $j^{th}$ subset $(1 \leq j \leq m)$ contains frequent itemsets satisfying $C$ and having $a_j$ as a prefix.*

**Proof.** The lemma follows on showing: (i) every frequent itemset $S$ satisfying $C$ must be in the $j^{th}$ subset, for some $j$, $1 \leq j \leq m$, and (ii) no two subsets overlap. ■

We mine the subsets of frequent itemsets satisfying the constraint by constructing the corresponding *projected database*.

**Definition 5.6 (Projected database)** Given a transaction database $\mathcal{T}$, an itemset $\alpha$ and an order $\mathcal{R}$.

1. Itemset $\beta$ is called the *max-prefix projection* of transaction $\langle tid, I_t\rangle \in \mathcal{T}$ w.r.t. $\mathcal{R}$, if and only if (1) $\alpha \subseteq I_t$ and $\beta \subseteq I_t$; (2) $\alpha$ is a prefix of $\beta$ w.r.t. $\mathcal{R}$; and (3) there exists no proper superset $\gamma$ of $\beta$ such that $\gamma \subseteq I_t$ and $\gamma$ also has $\alpha$ as a prefix w.r.t. $\mathcal{R}$.

2. The *$\alpha$-projected database* is the collection of max-prefix projections of transactions containing $\alpha$, w.r.t. $\mathcal{R}$. ■

**Remark 5.3.2** *Given a transaction database $\mathcal{T}$, a support threshold $\xi$ and a convertible anti-monotone constraint $C$. Let $\alpha$ be a frequent itemset satisfying $C$. The complete set of frequent itemsets satisfying $C$ and having $\alpha$ as a prefix can be mined from the $\alpha$-projected database.*

**Rationale.** To mine frequent itemsets having $\alpha$ as a prefix, only the transactions containing $\alpha$ is needed. Furthermore, according to the definition of convertible anti-monotonicity, the information about itemsets having $\alpha$ as a prefix is sufficient to serve the mining with the constraint. That information is completely retained in the max-prefix projections. So we have the lemma. ■

The mining process can be further improved by the following lemma.

**Definition 5.7 (Ascending and descending orders)** An order $\mathcal{R}$ over a set of items $I$ is called an *ascending order* for function $h : 2^I \rightarrow \mathbf{R}$ if and only if (1) for items $a$ and $b$, $h(a) < h(b)$ implies $a \mathcal{R} b$, and (2) for itemsets $\alpha \cup \{a\}$ and $\alpha \cup \{b\}$ such that both of them have $\alpha$ as a prefix and $a \mathcal{R} b$, $f(\alpha \cup \{a\}) \leq f(\alpha \cup \{b\})$. $\mathcal{R}^{-1}$ is called a *descending order* for function $h$. ■

For example, it can be verified that the value ascending order is an *ascending* order for function $avg(S)$ and a *descending order* for function $max(S)/avg(S)$.

**Lemma 5.5** *Given a convertible anti-monotone constraint $C \equiv f(S)\ \theta\ v$ ($\theta \in \{\leq, \geq\}$) w.r.t. ascending/descending order $\mathcal{R}$ over a set of items $I$, where $f$ is a prefix function. Let $\alpha$ be a frequent itemset satisfying $C$ and $a_1, a_2, \ldots, a_m$ be the set of frequent items in the $\alpha$-projected database, listed in the order of $\mathcal{R}$.*

1. *If itemset $\alpha \cup \{a_i\}$ $(1 \leq i < m)$ violates $C$, for $j$ such that $i < j \leq m$, itemset $\alpha \cup \{a_j\}$ also violates $C$.*

*2. If itemset $\alpha \cup \{a_j\}$ $(1 \leq j < m)$ satisfies $C$, but $\alpha \cup \{a_j, a_{j+1}\}$ violates $C$, no frequent itemset having $\alpha \cup \{a_j\}$ as a proper prefix satisfies $C$.*

**Proof.** The constraint $C$ must be in one of the two forms: (1) $f$ is a prefix ascending function w.r.t. descending order $\mathcal{R}$ and $C \equiv f(S) \geq v$ or (2) $f$ is a prefix descending function w.r.t. ascending order $\mathcal{R}$ and $C \equiv f(S) \leq v$. Here, we show the lemma holds for the first case. The second case can be shown similarly.

Suppose $f(\alpha \cup \{a_i\}) < v$S. Since $\mathcal{R}$ is a descending order, $a_i \; \mathcal{R} \; a_j$ implies $f(\alpha \cup \{a_j\}) \leq f(\alpha \cup \{a_i\})$. That means itemset $\alpha \cup \{a_j\}$ also violates $C$.

Alternatively, suppose $f(\alpha \cup \{a_j\}) \geq v$ but $f(\alpha \cup \{a_j, a_{j+1}\}) < v$. For any item $a_{j+k}$ after $a_j$ in the order of $\mathcal{R}$, $f(\alpha \cup \{a_j, a_{j+k}\}) \leq f(\alpha \cup \{a_j, a_{j+1}\})$. So, itemset $\alpha \cup \{a_j, a_{j+k}\}$ must also violate the constraint. ■

Based on the above reasoning, we have the algorithm $\mathcal{FIC}^{\mathcal{A}}$ as follows for mining Frequent Itemsets with Convertible Anti-monotone constraints.

**Algorithm 6** ($\mathcal{FIC}^{\mathcal{A}}$)

**Input:** a transaction database $\mathcal{T}$, a support threshold $\xi$ and a convertible anti-monotone constraint $C$ w.r.t. an order $\mathcal{R}$ over a set of items $I$

**Output:** the complete set of frequent itemsets satisfying the constraint $C$

**Method:** Call $fic_A(\emptyset, \mathcal{T})$;

**Function** $fic_A(\alpha, \mathcal{T}|_\alpha)$

**Parameters:** $\alpha$ is the itemset as prefix and $\mathcal{T}|_\alpha$ is the $\alpha$-projected database.

**Method:**

1. Scan $\mathcal{T}|_\alpha$ once, find frequent items in $\mathcal{T}|_\alpha$. Let $I_\alpha$ be the set of frequent items within $\mathcal{T}|_\alpha$ such that $\forall a \in I_\alpha, C(\alpha \cup \{a\}) = true$.

2. If $I_\alpha = \emptyset$ return, else $\forall a \in I_\alpha$, output $\alpha \cup \{a\}$ as a frequent itemset satisfying the constraint.

3. If $C$ is in form of $f(S)\ \theta\ v$ where $f$ is a prefix function and $\theta \in \{\leq, \geq\}$, using Lemma 5.5 to optimize the mining by removing items $b$ from $I_\alpha$ such that there exists no frequent itemset satisfying $C$ and having $\alpha \cup \{b\}$ as a proper prefix.

4. Scan $\mathcal{T}|_\alpha$ once more, $\forall a \in I|_\alpha$, generate $\alpha \cup \{a\}$-projected database $\mathcal{T}|_{\alpha \cup \{a\}}$.

5. For each item $a$ in $I|_\alpha$, call $fic_A(\alpha \cup \{a\}, \mathcal{T}|_{\alpha \cup \{a\}})$.

**Rationale.** The correctness and completeness of the algorithm has been reasoned step-by-step in this section. The efficiency of the algorithm is at that it pushes the constraint deep into the mining process, so that we do not need to generate the complete set of frequent itemsets in most cases. Only related frequent itemsets are identified and tested. As shown in Example 5.6 and in the experimental results, the search space is decreased dramatically when the constraint is sharp. ■

Based on the above reasoning, we have the following theorem.

**Theorem 5.4** *Given a transaction database, a support threshold and a convertible constraint, $\mathcal{FIC}^{\mathcal{A}}$ (Algorithm 6) computes the complete set of frequent itemsets satisfying the constraint without duplication.* ■

### 5.3.3 $\mathcal{FIC}^{\mathcal{M}}$: Mining frequent itemsets with monotone constraints

In the last two subsections, an efficient algorithm for mining frequent itemsets with convertible anti-monotone constraints is developed. Under similar spirit, an algorithm for mining frequent itemsets with convertible monotone constraints can also be developed. Instead of giving details of formal reasoning, we illustrate the ideas using an example and then present the algorithm.

**Example 5.7** Let us mine frequent itemsets in transaction database $\mathcal{T}$ in Table 5.2 with constraint $C \equiv avg(S) \leq 20$. Suppose the support threshold $\xi = 2$. In this example, we use the value descending order $\mathcal{R}$ exactly as is used in Example 5.6. Constraint $C$ is convertible monotone w.r.t. order $\mathcal{R}$.

After one scan of transaction database $\mathcal{T}$, the set of frequent 1-itemsets is found. Among the 7 frequent 1-itemsets, $g$, $d$, $b$, $c$ and $e$ satisfy the constraint $C$. According to the definition of convertible monotone constraints, frequent itemset having one of these 5 itemsets as a prefix must also satisfy the constraint. That is, the $g$-, $d$-, $b$-, $c$- and $e$-projected databases can be mined without testing constraint $C$, because adding smaller items will only decrease the value of $avg$. But $a$- and $f$-projected databases should be mined with constraint $C$ testing. However, as soon as its frequent $k$-itemsets for any $k$ satisfy the constraint, constraint checking will not be needed for further mining of their projected databases. ■

We present the algorithm $\mathcal{FIC}^{\mathcal{M}}$ for mining frequent itemsets with convertible monotone constraint as follows.

**Algorithm 7** ($\mathcal{FIC}^{\mathcal{M}}$)

**Input:** A transaction database $\mathcal{T}$, a support threshold $\xi$ and a convertible monotone constraint $C$ w.r.t. an order $\mathcal{R}$ over a set of items $I$.

**Output:** The complete set of frequent itemsets satisfying the constraint $C$.

**Method:** Call $fic_M(\emptyset, \mathcal{T}, 1)$;

**Function** $fic_M(\alpha, \mathcal{T}|_\alpha, check\_flag)$

**Parameters:** $\alpha$ is the itemset as prefix, $\mathcal{T}|_\alpha$ is the $\alpha$-projected database, and $check\_flag$ is the flag for constraint checking.

**Method:**

1. Scan $\mathcal{T}|_\alpha$ once, find frequent items in $\mathcal{T}|_\alpha$. If $check\_flag$ is 1, let $I_\alpha^+$ be the set of frequent items within $\mathcal{T}|_\alpha$ such that $\forall a \in I_\alpha^+, C(\alpha \cup \{a\}) = true$, and $I_\alpha^-$ be the set of frequent items within $\mathcal{T}|_\alpha$ such that $\forall b \in I_\alpha^-, C(\alpha \cup \{b\}) = false$. If $check\_flag$ is 0, let $I_\alpha^+$ be the set of frequent items within $\mathcal{T}|_\alpha$ and $I_\alpha^-$ be $\emptyset$.

2. $\forall a \in I_\alpha^+$, output $\alpha \cup \{a\}$ as a frequent itemset satisfying the constraint.

3. Scan $\mathcal{T}|_\alpha$ once more, $\forall a \in I|_\alpha^+ \cup I|_\alpha^-$, generate $\alpha \cup \{a\}$-projected database $\mathcal{T}|_{\alpha \cup \{a\}}$.

4. For each item $a$ in $I|_\alpha^+$, call $fic_M(\alpha \cup \{a\}, \mathcal{T}|_{\alpha \cup \{a\}}, 0)$; For each item $a$ in $I|_\alpha^-$, call $fic_M(\alpha \cup \{a\}, \mathcal{T}|_{\alpha \cup \{a\}}, 1)$;

**Rationale.** The correctness and completeness of the algorithm can be shown based on the similar reasoning in Section 5.3.2. Here, we analyze the difference between $\mathcal{FIC}^{\mathcal{M}}$ with an *Apriori*-like algorithm using constraint-checking as post-processing.

Both $\mathcal{FIC}^{\mathcal{M}}$ and *Apriori*-like algorithms have to generate the complete set of frequent itemsets, no matter whether the frequent itemsets satisfy the convertible monotone constraint. The frequent itemsets not satisfying the constraint cannot be pruned. This is the inherent difficulty of convertible monotone constraint.

The advantage of $\mathcal{FIC}^{\mathcal{M}}$ against *Apriori*-like algorithms lies in the fact that $\mathcal{FIC}^{\mathcal{M}}$ only tests some of frequent itemsets against the constraint. Once a frequent itemset satisfies the constraint, it guarantees all of frequent itemsets having it as a prefix also satisfy the constraint. Therefore, all that testing can be saved. An *Apriori*-like algorithm has to check every frequent itemset against the constraint. In the situation such that constraint testing is costly, such as spatial constraints, the saving over constraint testing could be non-trivial. Exploration of spatial constraints is beyond the scope of this chapter. ■

### 5.3.4 Mining frequent itemsets with strongly convertible constraints

The main value of strong convertibility is that the constraint can be treated either as convertible anti-monotone or monotone by choosing an appropriate order. The main point to note in practice is when the constraint has a high selectivity (fewer itemsets satisfy it), converting it into an anti-monotone constraint will yield maximum benefits by search space pruning. When the constraint selectivity is low (and checking it is reasonably expensive), then converting it into a monotone constraint will save considerable effort in constraint checking. The constraint $avg(S) \leq v$ is a classic example.

## 5.4 Experimental Results

To evaluate the effectiveness and efficiency of the algorithms, we performed an extensive experimental evaluation.

In this section, we report the results on a synthetic transaction database with 100K transactions and 10K items[8]. The dataset is generated by the standard procedure described in [AS94]. In this dataset, the average transaction size and average maximal potentially

[8]The dataset is downloadable at http://www.cs.sfu.ca/~peijian/personal/publications/T25I20D100k.dat.gz.

frequent itemset size are set to 25 and 20, respectively. The dataset contains a lot of frequent itemsets with various lengths. This dataset is chosen since it is typical in data mining performance study.

The algorithms are implemented in C. All the experiments are performed on a 233MHz Pentium PC with 128MB main memory, running Microsoft Windows/NT.

To evaluate the effect of a constraint on mining frequent itemsets, we make use of constraint selectivity, where the *selectivity* $\delta$ of a constraint $C$ on mining frequent itemsets over transaction database $\mathcal{T}$ with support threshold $\xi$ is defined as

$$\delta = \frac{\#\text{ of frequent itemsets NOT satisfying } C}{\#\text{ of frequent itemsets}}$$

Therefore, a constraint with 0% selectivity means every frequent itemset satisfies the constraint, while a constraint with 100% selectivity means that the constraint cannot be satisfied by any frequent itemset. The selectivity measure defined here is consistent with those used in [NLHP98, LNHP99].

To facilitate the mining using projected databases, we employ a data structure called *FP-tree* in the implementations of $\mathcal{FIC}^{\mathcal{A}}$ and $\mathcal{FIC}^{\mathcal{M}}$. *FP-tree* is first proposed in [HPY00], and also be adopted by [PH00, PHM00]. It is a prefix tree structure to record complete and compact information for frequent itemset mining. A transaction database/projected database can be compressed into an *FP-tree*, while all the consequent projected databases can be derived from it efficiently. We refer readers to [HPY00] for details about *FP-tree* and methods for *FP-tree*-based frequent itemset mining.

Since *FP-growth* [HPY00] is the *FP-tree*-based algorithm mining frequent itemsets and much faster than *Apriori*, we include it in our experiment. It is thus more interesting to compare the performance among $\mathcal{FIC}^{\mathcal{A}}$, $\mathcal{FIC}^{\mathcal{M}}$, and *FP-growth* than taking *Apriori* as the only reference method.

### 5.4.1 Evaluation of $\mathcal{FIC}^{\mathcal{A}}$

To test the efficiency of $\mathcal{FIC}^{\mathcal{A}}$ w.r.t. constraint selectivity in mining frequent itemsets with convertible anti-monotone constraints, a test is performed over the dataset with support threshold $\xi = 0.1\%$. The result is shown in Figure 5.3. Various settings are used in the constraint for various selectivity.

As shown in Figure 5.3, $\mathcal{FIC}^{\mathcal{A}}$ achieves an almost linear scalability with the constraint selectivity. As the selectivity goes up, i.e., when fewer itemsets satisfying the constraint,

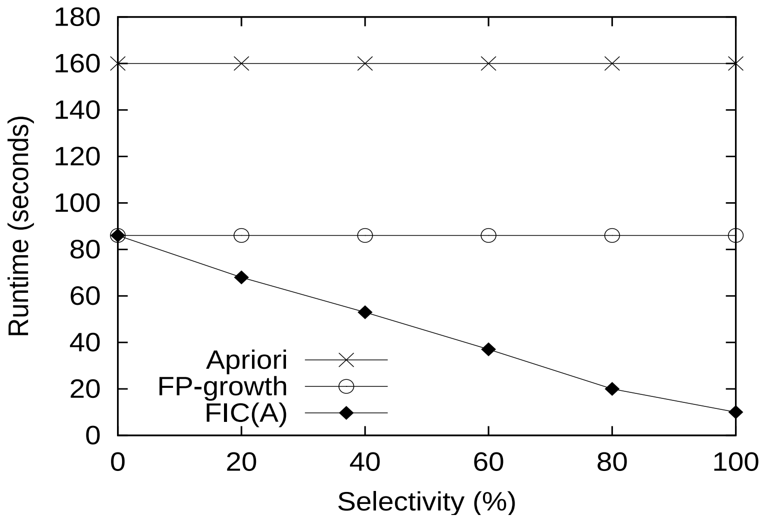

Figure 5.3: Scalability with constraint selectivity.

$\mathcal{FIC}^{\mathcal{A}}$ cuts more search space, since if there is a frequent itemset $s$ which does not satisfy the constraint, it means all the frequent itemsets with $s$ as a prefix can be pruned.

We also compare the runtime of *Apriori* and *FP-growth* in the same figure. Both methods first compute the complete set of frequent itemsets and then use the constraint as a filter. So, their runtime is constant w.r.t. constraint selectivity. However, only when the constraint selectivity is 0%, i.e., when every frequent itemset satisfies the constraint, does $\mathcal{FIC}^{\mathcal{A}}$ need as same runtime as *FP-growth*. In all other situations, $\mathcal{FIC}^{\mathcal{A}}$ always requires less time.

We also tested the scalability of $\mathcal{FIC}^{\mathcal{A}}$ with support threshold and number of transactions, respectively. The corresponding results are shown in Figure 5.4 and Figure 5.5. From the figures, we can see that $\mathcal{FIC}^{\mathcal{A}}$ is scalable in both cases. Furthermore, the higher the constraint selectivity, the more scalable $\mathcal{FIC}^{\mathcal{A}}$ is. This can be explained by the fact that $\mathcal{FIC}^{\mathcal{A}}$ always cuts more search spaces using constraints with higher selectivity.

### 5.4.2 Evaluation of $\mathcal{FIC}^{\mathcal{M}}$

As analyzed before, convertible monotone constraint can be used to save the cost of constraint checking, but it cannot cut the search space of frequent itemsets. In our experiments, since we use relatively simple constraints, such as those involving *avg* and *sum*, the cost

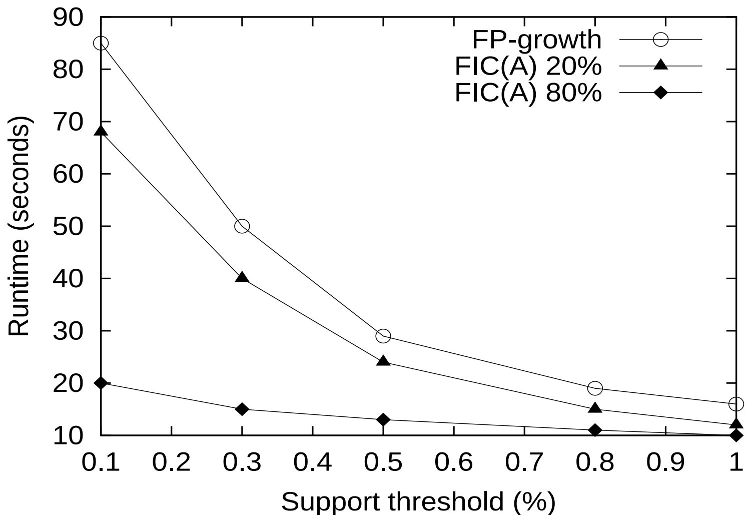

Figure 5.4: Scalability with support threshold.

of constraint checking is CPU-bounded. However, the cost of the whole frequent itemset mining process is I/O-bounded. This makes the effect of pushing convertible monotone constraint into the mining process hard to be observed from runtime reduction. In our experiments, $\mathcal{FIC}^{\mathcal{M}}$ achieves less than 3% runtime benefit in most cases.

However, if we look at the number of constraint tests performed, the advantage of $\mathcal{FIC}^{\mathcal{M}}$ can be evaluated objectively. $\mathcal{FIC}^{\mathcal{M}}$ can save a lot of effort on constraint testing. Therefore, in the experiments about $\mathcal{FIC}^{\mathcal{M}}$, the number of constraint tests is used as the performance measure.

We test the scalability of $\mathcal{FIC}^{\mathcal{M}}$ with constraint selectivity in mining frequent itemsets with convertible monotone constraint. The result is shown in Figure 5.6, which indicates that $\mathcal{FIC}^{\mathcal{M}}$ has a linear scalability. When the constraint selectivity is low, i.e., most frequent itemsets can pass the constraint checking, most of constraint tests can be saved. This is because once a frequent itemset satisfies a convertible monotone constraint, every subsequent frequent itemset derived from corresponding projected database has that frequent itemset as a prefix and thus satisfies the constraint, too.

We also tested the scalability of $\mathcal{FIC}^{\mathcal{M}}$ with support threshold. The result is shown in Figure 5.7. The figure shows that $\mathcal{FIC}^{\mathcal{M}}$ is scalable. Furthermore, the lower the constraint

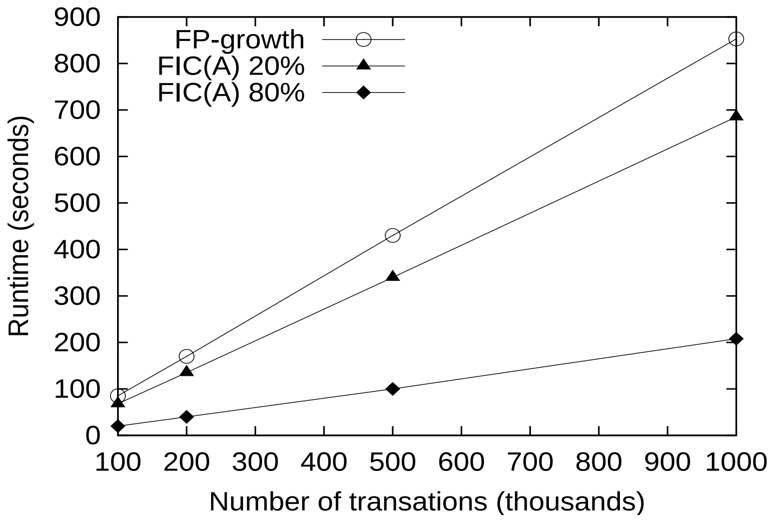

Figure 5.5: Scalability with number of transactions.

selectivity, the better the scalability $\mathcal{FIC}^{\mathcal{M}}$ is.

In summary, our experimental results show that the method proposed in this chapter is scalable for mining frequent itemsets with convertible constraints in large transaction databases. The experimental results strongly support our theoretical analysis.

## 5.5 Mining Frequent Itemsets with Multiple Convertible Constraints

We have studied the push of *single* convertible constraints into frequent itemset mining. "*Can we push multiple constraints deep into the frequent pattern mining process?*"

Multiple constraints in a mining query may belong to the same category (e.g. all are anti-monotone) or to different categories. Moreover, different constraints may be on different properties of items (e.g. some could be on item price, others on sales profits, the number of items, etc.).

As shown in our previous analysis, unlike anti-monotone, monotone and succinct constraints, *convertible* constraints can be mined only by ordering items properly. However, different constraints may require different or even conflicting item ordering. The question

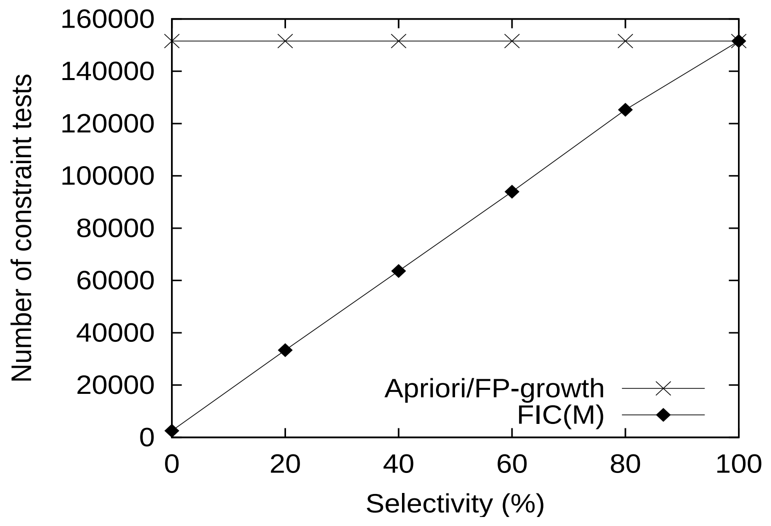

Figure 5.6: Scalability with constraint selectivity.

is how to deal with this nicely. In the following, we refer to a constraint with a high (low) selectivity as a sharp (blunt) constraint.

In the sequel, we consider mining frequent itemsets with a constraint $C_1 \circ C_2$, where both $C_1$ and $C_2$ are convertible constraints, and $\circ \in \{\wedge, \vee\}$.

**Case 1.** *There exists an order $\mathcal{R}$ such that both $C_1$ and $C_2$ are convertible w.r.t. $\mathcal{R}$.* In such a case, there is no conflict between the two convertible constraints. So, we can push both constraints into the mining process using the order $\mathcal{R}$. We suggest some heuristics as shown in Table 5.6.

**Case 2.** *There exists a conflict on the order of items.* Suppose $C_1$ requires $\mathcal{R}_1$ and $C_2$ requires $\mathcal{R}_2$, and $\mathcal{R}_1$ and $\mathcal{R}_2$ is incompatible. In such situations, we should try to satisfy one constraint at first, and then using the order for the other constraint to mine frequent itemsets in the corresponding projected database. The strategies are shown in Table 5.7.

Interested readers may verify the strategies in Tables 5.6 and 5.7 with the similar reasoning as provided in Section 4. We need ways to estimate the selectivity of constraints. In practice, methods such as sampling and business background knowledge often provide useful

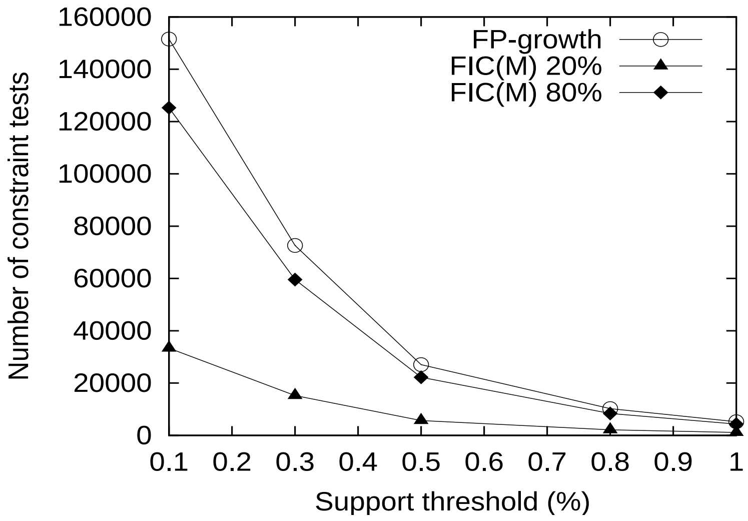

Figure 5.7: Scalability with support threshold.

estimation. Notice that queries may contain an anti-monotone or monotone constraint together with a convertible constraint. Since an anti-monotone or monotone constraint does not impose requirements on item ordering, such a constraint can be treated similarly as Case 1 (Table 5.6). Also, this discussion can be extended to the cases when there are more than two constraints.

## 5.6 Summary

Although there have been interesting studies, such as [NLHP98, LNHP99, GLW00], on mining frequent patterns with constraints, constraints involving holistic functions such as *median*, algebraic functions such as *avg*, or even those involving distributive functions like *sum* over sets with positive and negative item values are difficult to incorporate in an optimization process in frequent itemset mining. The reason is such constraints do not exhibit nice properties like monotonicity, etc. A main contribution of this chapter is showing that by imposing an appropriate order on items, such tough constraints can be converted into ones that possess monotone behavior. To this end, we made a detailed analysis and classification of the so-called convertible constraints. We characterized them using prefix

| Categories of constraints | $C_1 \vee C_2$ | $C_1 \wedge C_2$ |
|---|---|---|
| Both are convertible anti-monotone | Test the blunt constraint first. Only for itemsets violating both $C_1$ and $C_2$, the corresponding projected database can be pruned. | Test the sharp constraint first. For itemsets violating either constraint, their projected database can be pruned. |
| Both are convertible monotone | Test the blunt constraint first. Once an itemset satisfies either constraint, all the follow-up testing can be waived. | Test the sharp constraint first. Only when an itemset satisfies both constraints, can all the following-up testing be waived. |
| One is convertible monotone, while the other is convertible anti-monotone | Test the convertible monotone one first. If it is satisfied, the following-up testing can be waived. | Test the convertible anti-monotone constraint first. If it is violated, the corresponding projected database can be pruned. The convertible anti-monotone constraint-checking has to be done all the time, even when the convertible monotone one is satisfied/waived. |

Table 5.6: Strategies for mining with multiple convertible constraints without conflict on item ordering.

monotone functions and established their arithmetical closure properties. As a byproduct, we shed light on the overall picture of various classes of constraints that can be optimized in frequent set mining. While convertible constraints cannot be literally incorporated into an Apriori-style algorithm, they can be readily incorporated into the *FP-growth* algorithm. Our experiments show the effectiveness of the algorithms developed.

We have been working on a systematic implementation of constraint-based frequent pattern mining in a data mining system. More experiments are needed to understand how best to handle multiple constraints. An open issue is given an arbitrary constraint, how can we quickly check if it is (strongly) convertible. We are also exploring the use of constraints in clustering.

| Categories of constraints | $C_1 \vee C_2$ | $C_1 \wedge C_2$ |
|---|---|---|
| Both are convertible anti-monotone | Test the blunt constraint, say $C_1$, first, using order $\mathcal{R}_1$. When a frequent itemset $\alpha$ violates $C_1$, mine frequent itemsets $\beta$ in $\alpha$-projected database, using $\mathcal{R}_2$, such that $\alpha \cup \beta$ satisfies $C_2$. | Test the sharp constraint, say $C_1$, using order $\mathcal{R}_1$, all the time. Use $C_2$ as a post-filter. |
| Both are convertible monotone | Test the blunt constraint, say $C_1$, first, using order $\mathcal{R}_1$. When a frequent itemset $\alpha$ violates $C_1$, mine frequent itemsets $\beta$ in $\alpha$-projected database, using $\mathcal{R}_2$, such that $\alpha \cup \beta$ satisfies $C_2$. | Test the sharp constraint, say $C_1$, using order $\mathcal{R}_1$ first. When a frequent itemset $\alpha$ satisfies $C_1$, mine frequent itemsets $\beta$ in $\alpha$-projected database, using $\mathcal{R}_2$, such that $\alpha \cup \beta$ satisfies $C_2$. |
| One is convertible monotone, while the other is convertible anti-monotone | Test the convertible monotone one, say $C_1$, first, using $\mathcal{R}_1$. If satisfied, the follow-up testing can be waived. In the $\alpha$-projected database such that $\alpha$ violates $C_1$, mine frequent itemsets $\beta$ using $\mathcal{R}_2$ such that $\alpha \cup \beta$ satisfies $C_2$. | Test the convertible anti-monotone constraint first. If it is violated, corresponding projected database can be pruned. Use $C_2$ as a post-filter. |

Table 5.7: Strategies for mining with multiple convertible constraints with conflict on item ordering.

# Chapter 6

# Pattern-growth Sequential Pattern Mining

In previous chapters, we have developed efficient and effective pattern-growth methods for frequent pattern mining. *Can we extend pattern-growth methods to mine other kinds of patterns?* To examine the power of pattern-growth methods, in this chapter, we solve the sequential pattern mining problem using pattern-growth methods.

Sequential pattern mining, which discovers frequent subsequences as patterns in a sequence database, is an important data mining problem with broad applications, including the analysis of customer purchase patterns or Web access patterns, the analysis of the processes of scientific experiments, natural disasters, disease treatments, DNA analysis, and so on.

The sequential pattern mining problem was first introduced by Agrawal and Srikant in [AS95]: *Given a set of sequences, where each sequence consists of a list of elements and each element consists of a set of items, and given a user-specified min_support threshold, sequential pattern mining is to find all of the frequent subsequences, i.e., the subsequences whose occurrence frequency in the set of sequences is no less than min_support.*

Many studies have contributed to the efficient mining of sequential patterns or other frequent patterns in time-related data [AS95, SA96b, MTV97, WCM$^+$94, Zak98, MCP98, LHF98, BWJ98, ORS98, RMS98, HDY99]. Srikant and Agrawal [SA96b] generalize their definition of sequential patterns in [AS95] to include time constraints, sliding time window, and user-defined taxonomy. Mannila, et al. [MTV97] present a problem of mining frequent episodes in a sequence of events, where episodes are essentially acyclic graphs of events whose edges specify the temporal before-and-after relationship without timing-interval restrictions. Bettini, et al. [BWJ98] consider a generalization of inter-transaction association rules. These are essentially rules whose left-hand and right-hand sides are episodes with time-interval restrictions. Lu, et al. [LHF98] propose inter-transaction association rules which are implication rules whose two sides are totally-ordered episodes with timing-interval restrictions. Garofalakis, et al. [GRS99] propose the use of regular expressions as a flexible constraint specification tool that enables user-controlled focus to be incorporated into the sequential pattern mining process.

Recent works have also extended the scope from mining sequential patterns to mining partial periodic patterns. Özden, et al. [ORS98] introduce cyclic association rules which are essentially partial periodic patterns with *perfect* periodicity in the sense that *each pattern reoccurs in every cycle*, with 100% confidence. Han, et al. [HDY99] developed a frequent pattern mining method for mining partial periodicity patterns which are frequent maximal patterns where each pattern appears in a fixed period with a fixed set of offsets, and with sufficient support.

Almost all of the above proposed methods for mining sequential patterns and other time-related frequent patterns are *Apriori*-like, i.e., based on the *Apriori* heuristic, which states the fact that *any super-pattern of an infrequent pattern cannot be frequent*, and a candidate generation-and-test paradigm proposed in association mining [AS94].

A typical Apriori-like sequential pattern mining method, such as *GSP* [SA96b], adopts a multiple-pass, candidate generation-and-test approach, outlined as follows. The first scan finds all of the frequent items which form the set of single item frequent sequences. Each subsequent pass starts with a *seed set* of sequential patterns, which is *the set of sequential patterns found in the previous pass.* This seed set is used to generate new potential patterns, called *candidate sequences.* Each candidate sequence contains one more item than a seed sequential pattern, where each element in the pattern may contain one item or multiple items. The number of items in a sequence is called the *length* of the sequence. So, all the candidate sequences in a pass will have the same length. The scan of the database in one pass finds the support for each candidate sequence. All the candidates whose support in the database is no less than min_support form the set of the newly found sequential patterns. This set then becomes the seed set for the next pass. The algorithm terminates when no new sequential pattern is found in a pass, or when no candidate sequence can be generated.

The *Apriori*-like sequential pattern mining method, though reduces search space, bears three nontrivial, inherent costs which are independent of detailed implementation techniques.

- *A huge set of candidate sequences could be generated in a large sequence database.* Since the set of candidate sequences includes all the possible permutations of the elements and repetition of items in a sequence, the *Apriori*-based method may generate a very large set of candidate sequences even for a moderate seed set. For example, two frequent sequences of length-1, $\langle a\rangle$ and $\langle b\rangle$, will generate 5 candidate sequences of length-2: $\langle aa\rangle$, $\langle ab\rangle$, $\langle ba\rangle$, $\langle bb\rangle$, and $\langle (ab)\rangle$, where $\langle (ab)\rangle$ represents that two events $a$ and $b$ happen in the same time slot. If there are 1000 frequent sequences of length-1, such as $\langle a_1\rangle$, $\langle a_2\rangle$, ..., $\langle a_{1000}\rangle$, an *Apriori*-like algorithm will generate $1000 \times 1000 + \frac{1000\times 999}{2} = 1,499,500$ candidate sequences, where the first term is derived from the set $\langle a_1a_1\rangle$, $\langle a_1a_2\rangle$, ..., $\langle a_1a_{1000}\rangle$, $\langle a_2a_1\rangle$, $\langle a_2a_2\rangle$, ..., $\langle a_{1000}a_{1000}\rangle$, and the second term is derived from the set $\langle (a_1a_2)\rangle$, $\langle (a_1a_3)\rangle$, ..., $\langle (a_{999}a_{1000})\rangle$.

- *Many database scans in mining.* Since the length of each candidate sequence grows by one at each database scan, to find a sequential pattern $\{(abc)(abc)(abc)(abc)(abc)\}$, the *Apriori*-based method must scan the database at least 15 times. This bears some nontrivial cost.

- *The* Apriori-*based method encounters difficulty when mining long sequential patterns.* This is because a long sequential pattern must grow up from a huge number of short sequential patterns, but the number of such candidate sequences is exponential to the length of the sequential patterns to be mined. For example, suppose there is only a single sequence of length 100, $\langle a_1a_2\ldots a_{100}\rangle$, in the database, and the min_support threshold is 1 (i.e., every occurring pattern is frequent), to (re-)derive this length-100 sequential pattern, the *Apriori*-based method has to generate 100 length-1 candidate sequences, $100 \times 100 + \frac{100\times 99}{2} = 14,950$ length-2 candidate sequences, $\begin{pmatrix} 100 \\ 3 \end{pmatrix} = 161,700$ length-3 candidate sequences[1], .... Obviously, the total number of candidate sequences to be generated is $\Sigma_{i=1}^{100} \begin{pmatrix} 100 \\ i \end{pmatrix} = 2^{100} - 1 \approx 10^{30}$.

[1]Notice that *Apriori* does cut a substantial amount of search space. Otherwise, the number of length-3 candidate sequences would have been $100 \times 100 \times 100 + 100 \times 100 \times 99 + \frac{100\times 99\times 98}{3\times 2} = 2,151,700$.

In many applications, it is not unusual that one may encounter a large number of sequential patterns and long sequences, such as in DNA analysis or stock sequence analysis. Therefore, it is important to re-examine the sequential pattern mining problem to explore more efficient and scalable methods.

Based on our analysis, both the thrust and the bottleneck of an *Apriori*-based sequential pattern mining method come from its step-wise candidate sequence generation and test. Given the success of pattern-growth methods in frequent pattern mining, *can we develop a pattern-growth method for sequential pattern mining which absorbs the spirit of* Apriori *but avoid or substantially reduce the expensive candidate generation and test*?

In this chapter, we systematically develop pattern-growth methods for mining sequential patterns efficiently. The new methods are non-*Apriori* and apply a divide-and-conquer, pattern-growth principle. The general idea is that *sequence databases are recursively projected into a set of smaller projected databases and sequential patterns are grown in each projected databases by exploring only local frequent fragments.* Two pattern growth schemes, *FreeSpan* (for **Fre**qu**e**nt pattern-projected **S**equential **pa**tter**n** mining) and *PrefixSpan* (for **Prefix**-projected **S**equential **pa**tter**n** mining), are proposed. They mine the complete set of sequential patterns but greatly reduce the efforts of candidate subsequence generation. To further improve mining efficiency, three kinds of database projections: *level-by-level projection*, *bi-level projection*, and *pseudo-projection*, are explored. A comprehensive performance study shows that *FreeSpan* and *PrefixSpan* outperform *Apriori*-based *GSP* algorithm and an integrated *PrefixSpan* is the fastest one in mining large sequence databases.

The remainder of the chapter is organized as follows. In Section 6.1, we define the sequential pattern mining problem and illustrate the ideas of *Apriori*-based sequential pattern mining method *GSP*. In Section 6.2, we introduce our projection-based sequential pattern mining with a general introduction to the two basic *FreeSpan* and *PrefixSpan* algorithms. In Section 6.3, methods for scaling up pattern growth using bi-level projection and pseudo-projection are proposed. Our experimental results and performance study are reported in Section 6.4. We summarize the factors contributing to the success of the method and discuss the related issues in Section 6.5. Our study is concluded in Section 6.6.

## 6.1 Problem Definition and Related Works

In this section, we first define the problem of sequential pattern mining and then illustrate the essential mining method *GSP* [SA96b] using an example.

### 6.1.1 Problem Definition

Let $I = \{i_1, i_2, \ldots, i_n\}$ be a set of all **items**. An **itemset** is a subset of items. A **sequence** is an ordered list of itemsets. A sequence $s$ is denoted by $\langle s_1 s_2 \cdots s_l \rangle$, where $s_j$ is an itemset, i.e., $s_j \subseteq I$ for $1 \leq j \leq l$. $s_j$ is also called an **element** of the sequence, and denoted as $(x_1 x_2 \cdots x_m)$, where $x_k$ is an item, i.e., $x_k \in I$ for $1 \leq k \leq m$. For brevity, the brackets are omitted if an element has only one item. That is, element $(x)$ is written as $x$. An item can occur at most once in an element of a sequence, but can occur multiple times in different elements of a sequence. The number of instances of items in a sequence is called the **length** of the sequence. A sequence with length $l$ is called an **$l$-sequence**. A sequence $\alpha = \langle a_1 a_2 \cdots a_n \rangle$ is called a **subsequence** of another sequence $\beta = \langle b_1 b_2 \cdots b_m \rangle$ and $\beta$ a **super-sequence** of $\alpha$, denoted as $\alpha \sqsubseteq \beta$, if there exist integers $1 \leq j_1 < j_2 < \cdots < j_n \leq m$ such that $a_1 \subseteq b_{j_1}$, $a_2 \subseteq b_{j_2}$, $\ldots$, $a_n \subseteq b_{j_n}$.

A **sequence database** $S$ is a set of tuples $\langle sid, s \rangle$, where $sid$ is a **sequence_id** and $s$ is a sequence. A tuple $\langle sid, s \rangle$ is said to *contain* a sequence $\alpha$, if $\alpha$ is a subsequence of $s$, i.e., $\alpha \sqsubseteq s$. The support of a sequence $\alpha$ in a sequence database $S$ is the number of tuples in the database containing $\alpha$, i.e., $support_S(\alpha) = | \{\langle sid, s \rangle | (\langle sid, s \rangle \in S) \wedge (\alpha \sqsubseteq s)\} |$. It can be denoted as $support(\alpha)$ if the sequence database is clear from the context. Given a positive integer *min_support* as the **support threshold**, a sequence $\alpha$ is called a **sequential pattern** in sequence database $S$ if the sequence is contained by at least *min_support* tuples in the database, i.e., $support_S(\alpha) \geq min\_support$. A sequential pattern with length $l$ is called an **$l$-pattern**.

**Example 6.1 (Running example)** Let our running database be *sequence database* $S$ given in the first two columns of Table 6.1 and $min\_support = 2$. The set of *items* in the database is $\{a, b, c, d, e, f, g\}$.

A *sequence* $\langle a(abc)(ac)d(cf) \rangle$ has five *elements*: $(a)$, $(abc)$, $(ac)$, $(c)$ and $(cf)$, where items $a$ and $c$ appear more than once respectively in different elements. It is also a 9-*sequence* since there are 9 instances appearing in that sequence. Item $a$ happens three times in this

| Sequence_id | Sequence | item-pattern |
|---|---|---|
| 10 | $\langle a(abc)(ac)d(cf)\rangle$ | $\{a, b, c, d, f\}$ |
| 20 | $\langle (ad)c(bc)(ae)\rangle$ | $\{a, b, c, d, e\}$ |
| 30 | $\langle (ef)(ab)(df)cb\rangle$ | $\{a, b, c, d, e, f\}$ |
| 40 | $\langle eg(af)cbc\rangle$ | $\{a, b, c, e, f, g\}$ |

Table 6.1: A sequence database

sequence, so it contributes 3 to the *length* of the sequence. However, the whole sequence $\langle a(abc)(ac)d(cf)\rangle$ contributes only one to the *support* of $\langle a\rangle$. Also, sequence $\langle a(bc)df\rangle$ is a *subsequence* of $\langle a(abc)(ac)d(cf)\rangle$. Since both sequences 10 and 30 *contain* subsequence $s = \langle (ab)c\rangle$, $s$ is a *sequential pattern* of length 3 (i.e., 3-*pattern*). ■

**Problem Statement**. Given a sequence database and a *min_support* threshold, the problem of **sequential pattern mining** is to find the complete set of sequential patterns in the database.

### 6.1.2 Algorithm *GSP*

With the *Apriori* heuristic, a typical sequential pattern mining method, *GSP* [SA96b], proceeds as shown in the following example.

**Example 6.2 (*GSP*)** Given the database $S$ and min_support in Example 6.1, *GSP* first scans $S$, collects the support for each item, and finds the set of frequent items (in the form of *item* : *support*) as below,

$$a : 4, b : 4, c : 4, d : 3, e : 3, f : 3, g : 1$$

By filtering infrequent items, $g$, we obtain the first seed set $L_1 = \{\langle a\rangle, \langle b\rangle, \langle c\rangle, \langle d\rangle, \langle e\rangle, \langle f\rangle\}$, each representing a 1-element sequential pattern. Each subsequent pass starts with the seed set found in the previous pass and uses it to generate new potential sequential patterns, called *candidate sequences*.

For $L_1$, a set of 6 length-1 sequential patterns generates a set of $6\times6+\frac{6\times5}{2} = 51$ candidate sequences, $C_2 = \{\langle aa\rangle, \langle ab\rangle, \ldots, \langle af\rangle, \langle ba\rangle, \langle bb\rangle, \ldots, \langle ff\rangle, \langle (ab)\rangle, \langle (ac)\rangle, \ldots, \langle (ef)\rangle\}$.

The multi-scan mining process is shown in Figure 6.1, with the following explanations.

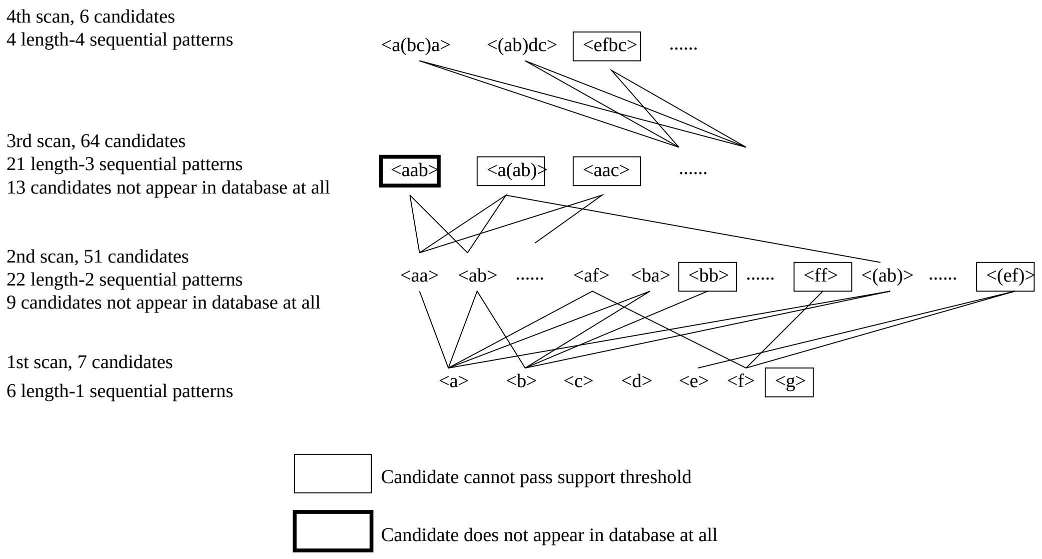
4th scan, 6 candidates
4 length-4 sequential patterns
3rd scan, 64 candidates
21 length-3 sequential patterns
13 candidates not appear in database at all
2nd scan, 51 candidates
22 length-2 sequential patterns
9 candidates not appear in database at all
1st scan, 7 candidates
6 length-1 sequential patterns
<a(bc)a>
<(ab)dc>
<efbc>
<aab>
<a(ab)>
<aac>
<aa>
<ab>
<af>
<ba>
<bb>
<ff>
<(ab)>
<(ef)>
<a>
<b>
<c>
<d>
<e>
<f>
<g>
Candidate cannot pass support threshold
Candidate does not appear in database at all

Figure 6.1: Candidates and sequential patterns in *GSP*

- The set of candidates is generated by a self-join of the sequential patterns found in the previous pass. In the $k$-th pass, a sequence is a candidate only if each of its length-$(k-1)$ subsequences is a sequential pattern found at the $(k-1)$-st pass.

- A new scan of the database collects the support for each candidate sequence and finds the new set of sequential patterns. This set becomes the seed for the next pass. The algorithm terminates when no sequential pattern is found in a pass, or when there is no candidate sequence generated.

- The number of scans is at least the maximum length of sequential patterns. It needs one more scan if the sequential patterns obtained in the last scan still generate new candidates.

- *GSP*, though benefits from the *Apriori* pruning, still generates a large number of candidates. In this example, 6 length-1 sequential patterns generate 51 length-2 candidates, 22 length-2 sequential patterns generate 64 length-3 candidates, etc.

- Candidates generated by *GSP* may not appear in the database at all. For example, 13 out of 64 length-3 candidates do not appear in the database. ■

## 6.2 Mining Sequential Patterns by Projections

In this section, we introduce new approaches for mining sequential patterns efficiently in large databases. The general idea is as follows. Instead of repeatedly scanning the entire database and generating and testing large sets of candidate sequences, we adopt a divide-and-conquer strategy: recursively *project* a sequence database into a set of smaller databases and then mine each projected database to find frequent patterns.

We propose two methods, *FreeSpan* [HPMA$^+$00] and *PrefixSpan* [PHMA$^+$01], for projection-based sequential pattern mining. Both methods create projected databases but they differ at the criteria of database projection: *FreeSpan* creates projected databases based on the current set of frequent patterns, whereas *PrefixSpan* does so based on frequent prefixes only. Our study shows that although both *FreeSpan* and *PrefixSpan* are faster than *GSP*, *PrefixSpan* is substantially faster than *FreeSpan*.

### 6.2.1 *FreeSpan*: Frequent Pattern-Projected Sequential Pattern Mining

Given a sequence $s$, the set of items appearing in it is called the **item-pattern** of the sequence, denoted as $\iota(s)$. For example, $\iota(\langle a(abc)(ac)d(cf)\rangle) = \{a, b, c, d, f\}$. Given a sequence database $S = \{\langle sid_i, s_i\rangle\}$, we obtain the corresponding **item-pattern database**, denoted as $\iota(S) = \{\langle sid_i, \iota(s_i)\rangle\}$, by substituting sequences by their item-patterns. For example, the item-patterns of sequences in the sequence database $S$ in Table 6.1 are listed in the third column in the same table. For an itemset $X$ (i.e. a set of items), the **support** of $X$, denoted as $support_{\iota(S)}(X) = |\{\langle sid_i, \iota(s_i)\rangle | X \subseteq \iota(s_i)\}|$, is the number of tuples in $\iota(S)$ containing $X$. Given a support threshold *min_support*, an itemset $X$ is called a **frequent item-pattern** if and only if $support_{\iota(S)}(X) \geq min\_support$.

The following lemma illustrate an interesting relationship between a sequential pattern in a sequence database and its item-pattern in the corresponding item-pattern database.

**Lemma 6.1 (Item-pattern)** *For a sequential pattern in a sequence database, its item-pattern must be frequent in the corresponding item-pattern database.*

**Proof.** Given a sequence database $S$ and a support threshold *min_support*. Let $s$ be a sequential pattern. Obviously, for every $\langle sid_i, s_i\rangle \in S$ such that $s \sqsubseteq s_i$, $\iota(s) \subseteq \iota(s_i)$. Therefore, we have $support_{\iota(S)}(\iota(s)) \geq support_S(s)$, which is followed by the theorem. ■

Please note that the reverse statement of Lemma 6.1 is not true. For example, $\{a, b, c, d\}$

is a frequent item-pattern in the item-pattern database of $S$ in Table 6.1 with respect to support threshold $min\_support = 2$. However, $\langle abcd \rangle$ is not frequent in the sequence database $S$.

Lemma 6.1 provides a heuristic to prune the search space in mining sequential patterns: If an item-pattern is infrequent, we do not need to examine its corresponding sequential patterns. *FreeSpan* adopts the heuristic to mine sequential patterns by partitioning search space and projecting sequence sub-databases recursively. We show how *FreeSpan* uses the heuristic in mining sequential patterns by an example as follows.

**Example 6.3 (*FreeSpan*)** Given the database $S$ and $min\_support$ in Example 6.1, *FreeSpan* first scans $S$, collects the support for each item, and finds the set of frequent items. This step is similar to *GSP*. Frequent items are listed in support descending order (in the form of $item : support$) as below,

$$F\text{-}list = a:4,\ b:4,\ c:4,\ d:3,\ e:3,\ f:3$$

According to *F-list*, the complete set of sequential patterns in $S$ can be divided into 6 disjoint subsets: (1) the ones containing only item $a$, (2) the ones containing item $b$ but no item after $b$ in *F-list*, (3) the ones containing item $c$ but no item after $c$ in *F-list*, and so on, and finally, (6) the ones containing item $f$.

The subsets of sequential patterns can be mined by constructing *projected databases.* Infrequent items, such as $g$ in this example, are removed from construction of projected databases. The mining process is detailed as follows.

- *Finding sequential patterns containing only item a.* By scanning sequence database once, the only two sequential patterns containing only item $a$, $\langle a \rangle$ and $\langle aa \rangle$, are found.

- *Finding sequential patterns containing item b but no item after b in* F-list. That can be achieved by constructing the $\{b\}$*-projected database.* For a sequence $\alpha$ in $S$ containing item $b$, a subsequence $\alpha'$ is derived by removing from $\alpha$ all items after $b$ in *F-list.* $\alpha'$ is inserted into the $\{b\}$-projected database. Thus, the $\{b\}$-projected database contains four sequences: $\langle a(ab)a \rangle$, $\langle aba \rangle$, $\langle (ab)b \rangle$, and $\langle ab \rangle$. By scanning the projected database once more, all sequential patterns containing item $b$ but no item after $b$ in *F-list* are found. They are $\{\langle b \rangle, \langle ab \rangle, \langle ba \rangle, \langle (ab) \rangle\}$.

- *Finding other subsets of sequential patterns.* Other subsets of sequential patterns can be found similarly, by constructing corresponding projected databases and mining them recursively.

Please note that the $\{b\}$-, $\{c\}$-, $\ldots$, $\{f\}$-projected databases are constructed simultaneously in one scan of the original sequence database. All sequential patterns containing only item $a$ are also found in that pass. This database projection process is performed recursively on the projected-databases. ■

Based on the above example, we can make sequences projections based on item-patterns as follows.

**Definition 6.1 (Projection based on item-pattern)** Let *F-list* be a list of items and $X$ a set of items. Item $\hat{X} \in X$ is called **the leading item** in $X$ with respect to *F-list* , if and only if there exists no item $y \in X$ such that $\hat{X} \neq y$ and $y$ is before $\hat{X}$ according to *F-list*.

Let $s$ be a sequence and $X$ a set of items. The **item-pattern projection** of $s$ against $X$, denoted as $s|_X$, is formed by removing all items $y$ from $s$ such that $y \notin X$ and $y$ is after $\hat{X}$ according to *F-list*.

Given a sequence database $S = \{\langle sid_i, s_i\rangle\}$ and a set of items $X$. The **item-pattern projected database** of $S$ against $X$, denoted as $S|_X$, is defined as $\{\langle sid_i, s_i|_X\rangle | \langle sid_i, s_i\rangle \in S\}$. ■

The *FreeSpan* algorithm is presented as follows.

**Algorithm 8 (*FreeSpan*)** Frequent pattern-projected sequential pattern mining.

**Input:** Sequence database $S$ and support threshold *min_support*

**Output:** The complete set of sequential patterns

**Method:** Call *FreeSpan*$(S, \emptyset)$.

**Procedure** *FreeSpan*$(proj\_db, freq\_item\_pat)$

```
1. Scan projected database $proj\_db$ once, find F-list, the list of frequent items except
   for those in frequent item-pattern $freq\_item\_pat$;
```

2. Scan *proj_db* again,
   - Find sequential patterns with item-pattern $x \cup freq\_item\_pat$, where $x \in$ *F-list*;
   - For each item $x \in$ *F-list*, form projected database $proj\_db|_{x \cup freq\_item\_pat}$ for item-pattern $x \cup freq\_item\_pat$;
3. For each item $x \in$ *F-list*, call *FreeSpan*($proj\_db|_{x \cup freq\_item\_pat}, x \cup freq\_item\_pat$).

**Analysis.** The correctness of Algorithm *FreeSpan* lays in the following two aspects.

- **Problem partitioning based on frequent item-patterns.** Let *F-list* $= x_1, \ldots, x_n$ be a list of all frequent items in sequence database $S$. Then, the complete set of sequential patterns in $S$ can be divided into $n$ disjoint subsets: the first is the set of sequential patterns containing only item $x_1$, the second is those containing item $x_2$ but no item in $\{x_3, \ldots, x_n\}$, and so on. In general, the $i^{th}$ subset $(1 \leq i \leq n)$ is the set of sequential patterns containing only item $x_i$ but no item in $\{x_{i+1}, \ldots, x_n\}$.

  Partitioning based on frequent item-patterns can be applied to the partitions recursively. Lemma 6.1 provides a theoretical support to that only those partitions of frequent item-patterns need to be considered. As long as we find sequential patterns in all partitions of frequent item-patterns, the global sequential pattern mining problem is solved completely without redundancy.

- **Forming and mining proper projected databases.** To find the complete set of sequential patterns in a partition for frequent item-pattern $X$, *FreeSpan* forms an $X$-projected database. The $X$-projected database contains only the segments of sequences potentially supporting sequential patterns in this partition. Any irrelevant information is discarded. The projected database contains the minimal information for finding those sequential patterns in the given partition. Recursively mining projected databases generates the complete set of sequential patterns in the given partition without duplication. ■

From Example 6.3 and the analysis of Algorithm *FreeSpan*, we have the following observations about the efficiency of *FreeSpan*. Our experimental results also verify our observations.

- ***FreeSpan* searches a smaller projected database than *GSP* in each subsequent database projection.** This is because that *FreeSpan* projects a large sequence database recursively into a set of small projected sequence databases based on the currently mined frequent item-patterns, the subsequent mining is confined to each projected database relevant to a smaller set of candidates.

- **The major cost of *FreeSpan* is to deal with projected databases.** If a pattern appears in each sequence of a database, its projected database does not shrink (except for the removal of some infrequent items). For example, the $\{f\}$-projected database in this example is the same as the original sequence database, except for the removal of infrequent item $g$. Moreover, since a length-$k$ subsequence may grow at any position, the search for length-$(k+1)$ candidate sequence will need to check every possible combination, which is costly.

### 6.2.2 *PrefixSpan*: Prefix-Projected Sequential Patterns Mining

Since items within an element of a sequence can be listed in any order, without loss of generality, we assume they are listed in alphabetical order. For example, the sequence in $S$ with Sequence_id 10 in our running example is listed as $\langle a(abc)(ac)d(cf)\rangle$ in stead of $\langle a(bac)(ca)d(fc)\rangle$. With such a convention, the expression of a sequence is unique.

**Definition 6.2 (Prefix, projection and suffix)** Suppose all the items in an element are listed alphabetically. Given a sequence $\alpha = \langle e_1 e_2 \cdots e_n\rangle$, a sequence $\beta = \langle e'_1, e'_2 \cdots e'_m\rangle$ $(m \leq n)$ is called a **prefix** of $\alpha$ if and only if (1) $e'_i = e_i$ for $(i \leq m-1)$; (2) $e'_m \subseteq e_m$; and (3) all the items in $(e_m - e'_m)$ are alphabetically after those in $e'_m$.

Given sequences $\alpha$ and $\beta$ such that $\beta$ is a subsequence of $\alpha$, i.e., $\beta \sqsubseteq \alpha$. A subsequence $\alpha'$ of sequence $\alpha$ (i.e., $\alpha' \sqsubseteq \alpha$) is called **a projection** of $\alpha$ w.r.t. prefix $\beta$ if and only if (1) $\alpha'$ is with prefix $\beta$ and (2) there exists no proper super-sequence $\alpha''$ of $\alpha'$ such that $\alpha''$ is a subsequence of $\alpha$ and also with prefix $\beta$.

Let $\alpha' = \langle e_1 e_2 \cdots e_n\rangle$ be the projection of $\alpha$ w.r.t. prefix $\beta = \langle e_1, e_2 \cdots e_{m-1} e'_m\rangle$ $(m \leq n)$. Sequence $\gamma = \langle e''_m e_{m+1} \cdots e_n\rangle$ is called the **suffix** of $\alpha$ w.r.t. prefix $\beta$, denoted as $\gamma = \alpha/\beta$, where $e''_m = (e_m - e'_m)$.[2] We also denote $\alpha = \beta \cdot \gamma$.

[2] If $e''_m$ is not empty, the suffix is also denoted as $\langle(\_\text{ items in } e''_m) e_{m+1} \cdots e_n\rangle$.

Especially, if $\beta$ is not a subsequence of $\alpha$, Both projection and suffix of $\alpha$ w.r.t. $\beta$ are empty, respectively. ■

For example, $\langle a \rangle$, $\langle aa \rangle$, $\langle a(ab) \rangle$ and $\langle a(abc) \rangle$ are *prefixes* of sequence $\langle a(abc)(ac)d(cf) \rangle$, but neither $\langle ab \rangle$ nor $\langle a(bc) \rangle$ is considered as a prefix. $\langle (abc)(ac)d(cf) \rangle$ is *suffix* w.r.t. prefix $\langle a \rangle$, $\langle (\_bc)(ac)d(cf) \rangle$ is *suffix* w.r.t. prefix $\langle aa \rangle$, and $\langle (\_c)(ac)d(cf) \rangle$ is *suffix* w.r.t. prefix $\langle ab \rangle$.

**Example 6.4** (*PrefixSpan*) For the same sequence database $S$ in Table 6.1 with $min\_sup = 2$, sequential patterns in $S$ can be mined by a prefix-projection method in the following steps.

1. *Find length-1 sequential patterns.* Scan $S$ once to find all the frequent items in sequences. Each of these frequent items is a length-1 sequential pattern. They are $\langle a \rangle : 4$, $\langle b \rangle : 4$, $\langle c \rangle : 4$, $\langle d \rangle : 3$, $\langle e \rangle : 3$, and $\langle f \rangle : 3$, where the notation "$\langle pattern \rangle : count$" represents the pattern and its associated support count.

2. *Divide search space.* The complete set of sequential patterns can be partitioned into the following six subsets according to the six prefixes: (1) the ones with prefix $\langle a \rangle$, (2) the ones with prefix $\langle b \rangle$, ..., and (6) the ones with prefix $\langle f \rangle$.

| prefix | projected (suffix) database | sequential patterns |
|---|---|---|
| $\langle a \rangle$ | $\langle (abc)(ac)d(cf) \rangle$, $\langle (\_d)c(bc)(ae) \rangle$, $\langle (\_b)(df)cb \rangle$, $\langle (\_f)cbc \rangle$ | $\langle a \rangle$, $\langle aa \rangle$, $\langle ab \rangle$, $\langle a(bc) \rangle$, $\langle a(bc)a \rangle$, $\langle aba \rangle$, $\langle abc \rangle$, $\langle (ab) \rangle$, $\langle (ab)c \rangle$, $\langle (ab)d \rangle$, $\langle (ab)f \rangle$, $\langle (ab)dc \rangle$, $\langle ac \rangle$, $\langle aca \rangle$, $\langle acb \rangle$, $\langle acc \rangle$, $\langle ad \rangle$, $\langle adc \rangle$, $\langle af \rangle$ |
| $\langle b \rangle$ | $\langle (\_c)(ac)d(cf) \rangle$, $\langle (\_c)(ae) \rangle$, $\langle (df)cb \rangle$, $\langle c \rangle$ | $\langle b \rangle$, $\langle ba \rangle$, $\langle bc \rangle$, $\langle (bc) \rangle$, $\langle (bc)a \rangle$, $\langle bd \rangle$, $\langle bdc \rangle$, $\langle bf \rangle$ |
| $\langle c \rangle$ | $\langle (ac)d(cf) \rangle$, $\langle (bc)(ae) \rangle$, $\langle b \rangle$, $\langle bc \rangle$ | $\langle c \rangle$, $\langle ca \rangle$, $\langle cb \rangle$, $\langle cc \rangle$ |
| $\langle d \rangle$ | $\langle (cf) \rangle$, $\langle c(bc)(ae) \rangle$, $\langle (\_f)cb \rangle$ | $\langle d \rangle$, $\langle db \rangle$, $\langle dc \rangle$, $\langle dcb \rangle$ |
| $\langle e \rangle$ | $\langle (\_f)(ab)(df)cb \rangle$, $\langle (af)cbc \rangle$ | $\langle e \rangle$, $\langle ea \rangle$, $\langle eab \rangle$, $\langle eac \rangle$, $\langle eacb \rangle$, $\langle eb \rangle$, $\langle ebc \rangle$, $\langle ec \rangle$, $\langle ecb \rangle$, $\langle ef \rangle$, $\langle efb \rangle$, $\langle efc \rangle$, $\langle efcb \rangle$. |
| $\langle f \rangle$ | $\langle (ab)(df)cb \rangle$, $\langle cbc \rangle$ | $\langle f \rangle$, $\langle fb \rangle$, $\langle fbc \rangle$, $\langle fc \rangle$, $\langle fcb \rangle$ |

Table 6.2: Projected databases and sequential patterns

3. *Find subsets of sequential patterns.* The subsets of sequential patterns can be mined by constructing the corresponding set of *projected databases* and mining each recursively. The projected databases as well as sequential patterns found in them are listed in Table 6.2, while the mining process is explained as follows.

(a) *Find sequential patterns with prefix* $\langle a \rangle$. Only the sequences containing $\langle a \rangle$ should be collected. Moreover, in a sequence containing $\langle a \rangle$, only the subsequence prefixed with the first occurrence of $\langle a \rangle$ should be considered. For example, in sequence $\langle (ef)(ab)(df)cb \rangle$, only the subsequence $\langle (\_b)(df)cb \rangle$ should be considered for mining sequential patterns prefixed with $\langle a \rangle$. Notice that $(\_b)$ means that the last element in the prefix, which is $a$, together with $b$, form one element.

The sequences in $S$ containing $\langle a \rangle$ are projected w.r.t. $\langle a \rangle$ to form the $\langle a \rangle$*-projected database*, which consists of four suffix sequences: $\langle (abc)(ac)d(cf) \rangle$, $\langle (\_d)c(bc)(ae) \rangle$, $\langle (\_b)(df)cb \rangle$ and $\langle (\_f)cbc \rangle$. By scanning $\langle a \rangle$-projected database once, all the length-2 sequential patterns prefixed with $\langle a \rangle$ can be found. They are: $\langle aa \rangle : 2$, $\langle ab \rangle : 4$, $\langle (ab) \rangle : 2$, $\langle ac \rangle : 4$, $\langle ad \rangle : 2$, and $\langle af \rangle : 2$.

Recursively, all sequential patterns with prefix $\langle a \rangle$ can be partitioned into 6 subsets: (1) that prefixed with $\langle aa \rangle$, (2) that with $\langle ab \rangle$, ..., and finally, (6) that with $\langle af \rangle$. These subsets can be mined by constructing respective projected databases and mining each recursively as follows.

i. The $\langle aa \rangle$-projected database consists of only one non-empty (suffix) subsequences prefixed with $\langle aa \rangle$: $\langle (\_bc)(ac)d(cf) \rangle$. Since there is no hope to generate any frequent subsequence from a single sequence, the processing of the $\langle aa \rangle$-projected database terminates.

ii. The $\langle ab \rangle$-projected database consists of three suffix sequences: $\langle (\_c)(ac)d(cf) \rangle$, $\langle (\_c)a \rangle$, and $\langle c \rangle$. Recursively mining the $\langle ab \rangle$-projected database returns four sequential patterns: $\langle (\_c) \rangle$, $\langle (\_c)a \rangle$, $\langle a \rangle$, and $\langle c \rangle$ (i.e., $\langle a(bc) \rangle$, $\langle a(bc)a \rangle$, $\langle aba \rangle$, and $\langle abc \rangle$.) They form the complete set of sequential patterns prefixed with $\langle ab \rangle$.

iii. The $\langle (ab) \rangle$-projected database contains only two sequences: $\langle (\_c)(ac)d(cf) \rangle$ and $\langle (df)cb \rangle$, which leads to the finding of the following sequential patterns prefixed with $\langle (ab) \rangle$: $\langle c \rangle$, $\langle d \rangle$, $\langle f \rangle$, and $\langle dc \rangle$.

iv. The $\langle ac \rangle$-, $\langle ad \rangle$- and $\langle af \rangle$- projected databases can be constructed and recursively mined similarly. The sequential patterns found are shown in Table 6.2.

(b) *Find sequential patterns with prefix* $\langle b \rangle$, $\langle c \rangle$, $\langle d \rangle$, $\langle e \rangle$ *and* $\langle f \rangle$, *respectively.* This can be done by constructing the $\langle b \rangle$-, $\langle c \rangle$- $\langle d \rangle$-, $\langle e \rangle$- and $\langle f \rangle$-projected databases and mining them respectively. The projected databases as well as the sequential patterns found are shown in Table 6.2.

4. *The set of sequential patterns is the collection of patterns found in the above recursive mining process.* One can verify that it returns exactly the same set of sequential patterns as what *GSP* and *FreeSpan* do. ■

Now, let us justify the correctness and completeness of the mining process in Example 6.4. Based on the concept of prefix, we have the following lemma on the completeness of partitioning the sequential pattern mining problem.

**Lemma 6.2 (Problem partitioning)**

1. *Let $\{\langle x_1\rangle, \langle x_2\rangle, \ldots, \langle x_n\rangle\}$ be the complete set of length-1 sequential patterns in sequence database $S$. The complete set of sequential patterns in $S$ can be divided into $n$ disjoint subsets. The $i^{th}$ subset $(1 \leq i \leq n)$ is the set of sequential patterns with prefix $\langle x_i\rangle$.*

2. *Let $\alpha$ be a length-$l$ sequential pattern and $\{\beta_1, \beta_2, \ldots, \beta_m\}$ be the set of all length-$(l+1)$ sequential patterns with prefix $\alpha$. The complete set of sequential patterns with prefix $\alpha$, except for $\alpha$ itself, can be divided into $m$ disjoint subsets. The $j^{th}$ subset $(1 \leq j \leq m)$ is the set of sequential patterns prefixed with $\beta_j$.*

**Proof.** We show the correctness of the second half of the lemma. The first half is a special case where $\alpha = \langle\rangle$.

For a sequential pattern $\gamma$ with prefix $\alpha$, where $\alpha$ is of length $l$, the length-$(l+1)$ prefix of $\gamma$ must be a sequential pattern, according to the Apriori heuristic. Furthermore, the length-$(l+1)$ prefix of $\gamma$ is also with prefix $\alpha$, according to the definition of prefix. Therefore, there exists some $j$ $(1 \leq j \leq m)$ such that $\beta_j$ is the length-$(l+1)$ prefix of $\gamma$. Thus, $\gamma$ is in the $j^{th}$ subset. On the other hand, since the length-$k$ prefix of a sequence $\gamma$ is unique, $\gamma$ belongs to only one determined subset. That is, the subsets are non-overlapping. So, we have the lemma. ■

Based on Lemma 6.2, *PrefixSpan* partitions the problem recursively. That is, each subset of sequential patterns can be further divided when necessary. This forms a divide-and-conquer framework. To mine the subsets of sequential patterns, *PrefixSpan* constructs the corresponding projected databases.

**Definition 6.3 (Projected database)** Let $\alpha$ be a sequential pattern in sequence database $S$. the **$\alpha$-projected database**, denoted as $S|_\alpha$, is the collection of suffixes of sequences in $S$ w.r.t. prefix $\alpha$. ■

To collect counts in projected databases, we have the following definition.

**Definition 6.4 (Support count in projected database)** Let $\alpha$ be a sequential pattern in sequence database $S$, and $\beta$ be a sequence with prefix $\alpha$. The **support count** of $\beta$ in $\alpha$-projected database $S|_\alpha$, denoted as $support_{S|_\alpha}(\beta)$, is the number of sequences $\gamma$ in $S|_\alpha$ such that $\beta \sqsubseteq \alpha \cdot \gamma$. ■

Please note that, in general, the following holds in our running example.

$$support_{S|_\alpha}(\beta) \leq support_{S|_\alpha}(\beta/\alpha)$$

For example, $support_S(\langle(ad)\rangle) = 1$ holds in our running example. However, $\langle(ad)\rangle/\langle a\rangle = \langle d\rangle$ and $support_{S|_{\langle a\rangle}}(\langle d\rangle) = 3$.

We have the following lemma on projected databases.

**Lemma 6.3 (Projected database)** *Let $\alpha$ and $\beta$ be two sequential patterns in sequence database $S$ such that $\alpha$ is a prefix of $\beta$.*

1. *$S|_\beta = (S|_\alpha)|_\beta$;*
2. *for any sequence $\gamma$ with prefix $\alpha$, $support_S(\gamma) = support_{S|_\alpha}(\gamma)$; and*
3. *The size of $\alpha$-projected database cannot exceed that of $S$.*

**Proof.** The first part of the lemma follows the fact that, for a sequence $\gamma$, the suffix of $\gamma$ w.r.t. $\beta$, $\gamma/\beta$, equals to the sequence resulted from first do projection of $\gamma$ w.r.t. $\alpha$, i.e., $\gamma/\alpha$, and then do projection $\gamma/\alpha$ w.r.t. $\beta$. That is $\gamma/\beta = (\gamma/\alpha)/\beta$.

The second part of the lemma states that to collect support count of a sequence $\gamma$, only the sequences in the database sharing the same prefix should be considered. Furthermore, only those suffixes with the prefix being a super-sequence of $\gamma$ should be counted. According to the related definitions, the claim is correct.

The third part of the lemma is on the size of a projected database. Obviously, the $\alpha$-projected database can have the same number of sequences as $S$ only if $\alpha$ appears in every sequence in $S$. Otherwise, only those sequences in $S$ which are super-sequences of $\alpha$ also appear in the $\alpha$-projected database. So, $\alpha$ must have at most the same number of sequences as $S$ does. For every sequence $\gamma$ in $S$ such that $\gamma$ is a super-sequence of $\alpha$, $\gamma$ appears in the $\alpha$-projected database in whole only if $\alpha$ is a prefix of $\gamma$. Otherwise, only a subsequence of $\gamma$ appears in the $\alpha$-projected database. Therefore, the size of $\alpha$-projected database cannot exceed that of $S$. ■

Based on the above reasoning, we have the algorithm of *PrefixSpan* as follows.

**Algorithm 9 (*PrefixSpan*)** Prefix-projected sequential pattern mining.

**Input:** A sequence database $S$, and the minimum support threshold *min_sup*

**Output:** The complete set of sequential patterns

**Method:**

Call *PrefixSpan*$(\langle\rangle, 0, S)$.

**Subroutine** *PrefixSpan*$(\alpha, l, S)$

**Parameters:**

- $\alpha$: a sequential pattern;
- $l$: the length of $\alpha$;
- $S|_\alpha$: the $\alpha$-projected database if $\alpha \neq \langle\rangle$; otherwise, the sequence database $S$.

**Method:**

1. Scan $S|_\alpha$ once, find the set of frequent items $b$ such that
   (a) $b$ can be assembled to the last element of $\alpha$ to form a sequential pattern; or
   (b) $\langle b\rangle$ can be appended to $\alpha$ to form a sequential pattern.
2. For each frequent item $b$, append it to $\alpha$ to form a sequential pattern $\alpha'$, and output $\alpha'$;
3. For each $\alpha'$, construct $\alpha'$-projected database $S|_{\alpha'}$, and call *PrefixSpan* $(\alpha', l+1, S|_{\alpha'})$.

**Analysis.** The correctness and completeness of the algorithm can be justified based on Lemma 6.2 and Lemma 6.3, as shown in Theorem 6.1 later. Here, we analyze the efficiency of the algorithm as follows.

- *No candidate sequence needs to be generated by* PrefixSpan. Unlike *Apriori*-like algorithms, *PrefixSpan* only grows longer sequential patterns from the shorter frequent ones. It neither generates nor tests any candidate sequence non-existent in a projected database. Comparing with *GSP*, which generates and tests a substantial number of candidate sequences, *PrefixSpan* searches a much smaller space.

- *Projected databases keep shrinking.* As indicated in Lemma 6.3, a projected database is smaller than the original one because only the suffix subsequences of a frequent prefix are projected into a projected database. In practice, the shrinking factors can be significant because (1) usually, only a small set of sequential patterns grow quite long in a sequence database, and thus the number of sequences in a projected database will become quite small when prefix grows; and (2) projection only takes the suffix portion with respect to a prefix. Notice that *FreeSpan* also employs the idea of projected databases. However, the projection there often takes the whole string (not just the suffix) and thus the shrinking factor is much less than that of *PrefixSpan*.

- *The major cost of* PrefixSpan *is the construction of projected databases.* In the worst case, *PrefixSpan* constructs a projected database for every sequential pattern. If there are a good number of sequential patterns, the cost is non-trivial. In the next section, we will develop strategies to reduce the number of projected databases dramatically.

■

**Theorem 6.1 (*PrefixSpan*)** *A sequence $\alpha$ is a sequential pattern if and only if* PrefixSpan *says so.*

**Proof.** *(Direction if)* A length-$l$ sequence $\alpha$ ($l \geq 1$) is identified as a sequential pattern by *PrefixSpan* if and only if $\alpha$ is a sequential pattern in the projected database of its length-$(l-1)$ prefix $\alpha^-$. If $l = 1$, the length-0 prefix of $\alpha$ is $\alpha^- = \langle\rangle$ and the projected database is $S$ itself. So, $\alpha$ is a sequential pattern in $S$. If $l > 1$, according to Lemma 6.3, $S|_{\alpha^-}$ is exactly the $\alpha^-$-projected database, and $support_S(\alpha) = support_{S|_{\alpha^-}}(\alpha)$. Therefore, $\alpha$ is a sequential pattern in $S|_{\alpha^-}$ means that it is also a sequential pattern in $S$. By this, we show that a sequence $\alpha$ is a sequential pattern if *PrefixSpan* says so.

*(Direction only-if)* Lemma 6.2 guarantees that *PrefixSpan* identifies the complete set of sequential patterns in $S$. So, we have the theorem. ■

## 6.3 Scaling Up Pattern Growth by Bi-Level Projection and Pseudo-Projection

As analyzed before, the major cost of *PrefixSpan* is to construct projected databases. If the number and/or the size of projected databases can be reduced, the performance of sequential pattern mining can be improved substantially. In this section, we develop two techniques to reduce the cost of constructing projected databases: (1) a bi-level projection scheme is proposed to reduce the number and the size of projected databases, and (2) a pseudo-projection is proposed to explore virtual projection when the database to be projected fits in main memory.

### 6.3.1 Bi-Level Projection

Before introducing the method formally, let us examine the following example.

**Example 6.5** Let us re-examine mining sequential patterns in sequence database $S$ in Table 6.1. The first step is the same: Scan $S$ to find the length-1 sequential patterns: $\langle a\rangle$, $\langle b\rangle$, $\langle c\rangle$, $\langle d\rangle$, $\langle e\rangle$ and $\langle f\rangle$.

At the second step, instead of constructing projected databases for each length-1 sequential pattern, we construct a $6 \times 6$ lower triangular matrix $M$, as shown in Table 6.3.

| | | | | | | |
|---|---|---|---|---|---|---|
| $a$ | 2 | | | | | |
| $b$ | (4, 2, 2) | 1 | | | | |
| $c$ | (4, 2, 1) | (3, 3, 2) | 3 | | | |
| $d$ | (2, 1, 1) | (2, 2, 0) | (1, 3, 0) | 0 | | |
| $e$ | (1, 2, 1) | (1, 2, 0) | (1, 2, 0) | (1, 1, 0) | 0 | |
| $f$ | (2, 1, 1) | (2, 2, 0) | (1, 2, 1) | (1, 1, 1) | (2, 0, 1) | 1 |
| | $a$ | $b$ | $c$ | $d$ | $e$ | $f$ |

Table 6.3: The *S-matrix* $M$.

The matrix $M$ registers the supports of all the length-2 sequences which are assembled using length-1 sequential patterns. A cell at the diagonal line has one counter. For example,

$M[c,c] = 3$ indicates sequence $\langle cc \rangle$ appears in $S$ in three sequences. Other cells have three counters respectively. For example, $M[a,c] = (4,2,1)$ means $support_S(\langle ac \rangle) = 4$, $support_S(\langle ca \rangle) = 2$ and $support_S(\langle (ac) \rangle) = 1$. Since the information in cell $M[c,a]$ is symmetric to that in $M[a,c]$, a triangle matrix is sufficient. This matrix is called an *S-matrix*.

By scanning sequence database $S$ again, the *S-matrix* can be filled up, as shown in Table 6.3. All the length-2 sequential patterns can be identified from the matrix immediately.

For each length-2 sequential pattern $\alpha$, construct $\alpha$-projected database. For example, $\langle ab \rangle$ is identified as a length-2 sequential pattern by *S-matrix*. The $\langle ab \rangle$-projected database contains three sequences: $\langle (\_c)(ac)(cf) \rangle$, $\langle (\_c)a \rangle$, and $\langle c \rangle$. By scanning it once, three frequent items are found: $\langle a \rangle$, $\langle c \rangle$ and $\langle (\_c) \rangle$. Then, a $3 \times 3$ *S-matrix* for $\langle ab \rangle$-projected database is constructed, as shown in Table 6.4.

| | | | |
|---|---|---|---|
| $a$ | 0 | | |
| $c$ | (1, 0, 1) | 1 | |
| $(\_c)$ | ($\emptyset$, 2, $\emptyset$) | ($\emptyset$, 1, $\emptyset$) | $\emptyset$ |
| | $a$ | $c$ | $(\_c)$ |

Table 6.4: The *S-matrix* in $\langle ab \rangle$-projected database.

Since there is only one cell with support 2, only one length-2 pattern $\langle (\_c)a \rangle$ can be generated and no further projection is needed. Notice that $\emptyset$ means that there is no possibility to generate such pattern. So, we do not need to look at the database.

To mine the complete set of sequential patterns, other projected databases for length-2 sequential patterns should be constructed. It is to verify that such a *bi-level projection* method produces exactly the same set of sequential patterns as the *PrefixSpan* algorithm introduced in Section 6.2.2. However, in Section 6.2.2, to find the complete set of 53 sequential patterns, 53 projected databases are constructed. In this example, only the projected databases for length-2 sequential patterns are needed. In total, only 22 projected databases are constructed by bi-level projection. ■

Now, let us justify the mining process by bi-level projection.

**Definition 6.5 (*S-matrix*, or sequence-match matrix)** Let $\alpha$ be a length-$l$ sequential pattern, and $\alpha'_1, \alpha'_2, \ldots, \alpha'_m$ be all of length-$(l+1)$ sequential patterns with prefix $\alpha$ within the $\alpha$-projected database. The *S-matrix* of the $\alpha$-projected database, denoted as $M[\alpha'_i, \alpha'_j]$ $(1 \leq i \leq j \leq m)$, is defined as follows.

1. $M[\alpha'_i, \alpha'_i]$ contains one counter. If the last element of $\alpha'_i$ has only one item $x$, i.e. $\alpha'_i = \langle \alpha x \rangle$, the counter registers the support of sequence $\langle \alpha'_i x \rangle$ (i.e., $\langle \alpha x x \rangle$) in the $\alpha$-projected database. Otherwise, the counter is set to $\emptyset$;

2. $M[\alpha'_i, \alpha'_j]$ $(1 \leq i < j \leq m)$ is in the form of $(A, B, C)$, where $A$, $B$ and $C$ are three counters.

   - If the last element in $\alpha'_j$ has only one item $x$, i.e. $\alpha'_j = \langle \alpha x \rangle$, counter $A$ registers the support of sequence $\langle \alpha'_i x \rangle$ in the $\alpha$-projected database. Otherwise, counter $A$ is set to $\emptyset$;
   - If the last element in $\alpha'_i$ has only one item $y$, i.e. $\alpha'_i = \langle \alpha y \rangle$, counter $B$ registers the support of sequence $\langle \alpha'_j y \rangle$ in the $\alpha$-projected database. Otherwise, counter $B$ is set to $\emptyset$;
   - If the last elements in $\alpha'_i$ and $\alpha'_j$ have same number of items, counter $C$ registers the support of sequence $\alpha''$ in the $\alpha$-projected database, where sequence $\alpha''$ is $\alpha'_i$ but inserting into the last element of $\alpha'_i$ the item in the last element of $\alpha'_j$ but not in that of $\alpha'_i$. Otherwise, counter $C$ is set to $\emptyset$. ■

**Lemma 6.4** *Given a length-$l$ sequential pattern $\alpha$.*

1. *The* S-matrix *can be filled up after two scans of the $\alpha$-projected database; and*
2. *A length-$(l+2)$ sequence $\beta$ with prefix $\alpha$ is a sequential pattern if and only if the* S-matrix *in the $\alpha$-projected database says so.*

**Proof.** The first half of the lemma is intuitive. Now, we show the second half of the lemma.

Suppose $\beta$ is a length-$(l+2)$ sequential pattern with prefix $\alpha$. $\beta$ must be formed in four ways: (1) adding two items $x$ and $y$ into the last element of $\alpha$, such that $x$ and $y$ are both alphabetically after the items in the last element of $\alpha$; (2) adding one item $x$ into the last element of $\alpha$, such that $x$ is alphabetically after the items in the last element of $\alpha$, and adding one element containing only one item $y$ as the last element of $\beta$; (3) adding two elements $xy$ to $\alpha$; or (4) adding an element $(x, y)$ as the last element of $\beta$. In cases (1), (2) and (4), as well as while $x \neq y$ in case (3), $\beta$ has two length-$(l+1)$ subsequences $\beta_1$ and

$\beta_2$ such that both of them are sequential patterns with prefix $\alpha$. According to Lemma 4.3, $\beta_1$ and $\beta_2$ are identified as length-$(l+1)$ sequential patterns in the $\alpha$-projected database. Thus, there exists either cell $M[\beta_1, \beta_2]$ or $M[\beta_2, \beta_1]$ in *S-matrix*. Without loss of generality, suppose we have $M[\beta_1, \beta_2]$. Since $\beta$ is a sequential pattern and $\beta$ can be assembled using $\beta_1$ and $\beta_2$, the corresponding counter in $M[\beta_1, \beta_2]$ must pass the minimum support threshold. Therefore, the *S-matrix* will identify $\beta$ as a sequential pattern. When $x = y$ in case (3), $\beta$ has only one length-$(l+1)$ subsequence $\beta'$ such that $\beta'$ is a sequential pattern with prefix $\alpha$. According to the definition of *S-matrix*, we have $M[\beta', \beta']$ in the matrix and it registers the support of $\beta$ in the $\alpha$-projected database. Lemma 4.3 says that the counter passes the minimum support threshold, since $\beta$ is a sequential pattern.

On the other hand, according to the definition of *S-matrix*, a counter registers support count of a unique length-$(l+2)$ sequence with prefix $\alpha$. Lemma 4.3 tells that the counter passes the minimum support threshold if and only if it is a sequential pattern. So, we have the lemma. ∎

Lemma 6.4 provides theoretical guarantee for the correctness of bi-level projection. Do we need to include every item in a suffix into the projected databases?

For example, let us consider the $\langle ac \rangle$-projected database in Example 6.5. The *S-matrix* in Table 6.3 tells that $\langle ad \rangle$ is a sequential pattern but $\langle cd \rangle$ is not. According to the *Apriori* heuristic [AS94], $\langle acd \rangle$ and any super-sequence of it can never be a sequential pattern. The matrix also tells that $\langle (cd) \rangle$ is not frequent. So, we can exclude item $d$ from the $\langle ac \rangle$-projected database. Here, we use the *3-way* Apriori *checking* to prune items from constructing projected databases. We state the principle using the following optimization.

**Optimization 1 (Pruning items from projected database) 3-way *Apriori* checking** should be used to prune items from constructing projected databases, based on the following two rules.

- In the $\alpha$-projected database, where $\alpha$ is a length-$l$ sequential pattern, let $\alpha'$ be a length-$(l-1)$ subsequence of $\alpha$. If $\alpha'(x)$ is not frequent, then item $x$ can be excluded from suffixes except for those $x$'s in the first element of the suffixes and those $x$'s in elements which are supersets of the last element of $\alpha$.

- In the $\alpha$-projected database, let $\alpha'$ be formed by substitute an item in the last element of $\alpha$ by $x$. If $\alpha'$ is not frequent, then item $x$ can be excluded from first elements of suffixes. ■

This optimization applies 3-way Apriori checking to reduce projected databases further. Only fragments of sequences necessary to grow longer patterns are projected.

### 6.3.2 Pseudo-Projection

The major cost of *PrefixSpan* is projection, i.e., forming projected databases recursively. Here, we propose a *pseudo-projection* technique which reduces the cost of projection substantially when a projected database can be held in main memory.

By examining a set of projected databases, one can observe that suffixes of a sequence often appear repeatedly in recursive projected databases. In Example 6.4, sequence $\langle a(abc)(ac)d(cf)\rangle$ has suffixes $\langle (abc)(ac)d(cf)\rangle$ and $\langle (\_c)(ac)d(cf)\rangle$ as projections in the $\langle a\rangle$- and $\langle ab\rangle$-projected databases, respectively. They are redundant subsequences. If the sequence database/projected database can be held in main memory, such redundancy can be avoided by pseudo-projection.

The method is as follows. When the database can be held in main memory, instead of constructing a *physical* projection by collecting all the suffixes, one can use pointers referring to the sequences in the database as a *pseudo-projection*. Every projection consists of two pieces of information: a *pointer* to the sequence in database and an *offset* that indicates the beginning position of the suffix within the sequence.

For example, suppose the sequence database $S$ in Table 6.1 can be held in main memory. When constructing the $\langle a\rangle$-projected database, the projection of sequence $s_1 = \langle a(abc)(ac)d(cf)\rangle$ consists of two pieces: a *pointer* to $s_1$ and *offset* set to 2. The offset indicates that the projection starts from position 2 in the sequence, i.e., $(abc)(ac)d$. Similarly, the projection of $s_1$ in the $\langle ab\rangle$-projected database contains a pointer to $s_1$ and offset set to 4 indicating the suffix starts from item $c$ in $s_1$, i.e., $(ac)d(cf)$.

Pseudo-projection avoids physically copying suffixes. Thus, it is efficient in terms of both running time and space. However, it is not efficient if the pseudo-projection is used for disk-based accessing since random access disk space is very costly. Based on this observation, *PrefixSpan* always pursues pseudo-projection once the projected databases can be held in main memory. Our experimental results show that an integrated solution combining disk-based bi-level projection with pseudo-projection when data can fit into main memory, is always a clear winner in performance.

## 6.4 Experimental Results and Performance Study

In this section, we report our experimental results on the performance of *PrefixSpan* in comparison with *GSP* and *FreeSpan*. It shows that *PrefixSpan* outperforms other previously proposed methods and is efficient and scalable for mining sequential patterns in large databases.

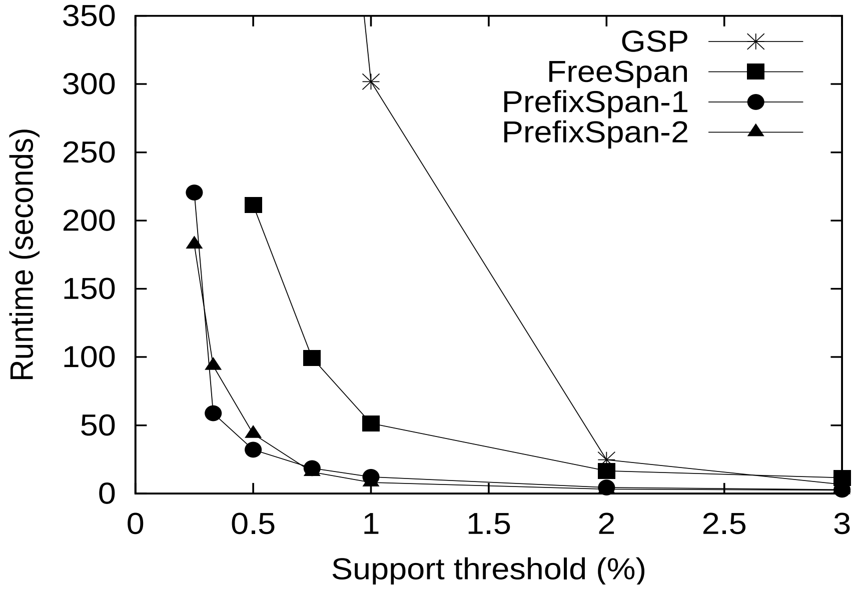

Figure 6.2: Runtime comparison among *PrefixSpan*, *FreeSpan* and *GSP* on data set $C10T8S8I8$.

All the experiments are performed on a 233MHz Pentium PC machine with 128 megabytes main memory, running Microsoft Windows/NT. All the methods are implemented using Microsoft Visual C++ 6.0.

We compare performance of four methods as follows.

- *GSP*. The *GSP* algorithm was implemented as described in [SA96b].
- *FreeSpan*. As reported in [HPMA$^+$00], *FreeSpan* with alternative level projection is more efficient than *FreeSpan* with level-by-level projection. In this chapter, *FreeSpan* with alternative level projection is used.

- *PrefixSpan-1.* *PrefixSpan-1* is *PrefixSpan* with level-by-level projection, as described in Section 6.2.2.
- *PrefixSpan-2.* *PrefixSpan-2* is *PrefixSpan* with bi-level projection, as described in Section 6.3.1.

The synthetic datasets used in our experiments were generated using the standard procedure described in [AS95]. The same data generator has been used in most studies on sequential pattern mining, such as [SA96b, HPMA$^+$00]. Agrawal and Srikant [AS95] give more details on the generation of data sets.

We tested the four methods on various datasets. The results are consistent. In this thesis, we report only the results on a representative dataset $C10T8S8I8$. In this data set, the number of items is set to $1,000$, and there are $10,000$ sequences in the data set. The average number of items within elements is set to 8 (denoted as $T8$). The average number of elements in a sequence is set to 8 (denoted as $S8$). There are a good number of long sequential patterns in it at low support thresholds.

The experimental results on scalability with different support thresholds are shown in Figure 6.2. When the support threshold is high, only a limited number of short sequential patterns appear. The four methods are close in terms of runtime. However, as the support threshold decreases, the gaps become clear. Both *FreeSpan* and *PrefixSpan* are faster than *GSP*. *PrefixSpan* methods are more efficient and more scalable than *FreeSpan.* We focus on the performance of various *PrefixSpan* techniques in the remainder of this section.

As shown in Figure 6.2, the performance curves of *PrefixSpan-1* and *PrefixSpan-2* are close when support threshold is not low. When the support threshold is low, since there are many sequential patterns, *PrefixSpan-1* requires a major effort to generate projected databases. Bi-level projection can leverage the problem efficiently. As can be seen from Figure 6.3, the increase of runtime for *PrefixSpan-2* is moderate even when the support threshold is pretty low.

Figure 6.3 also shows that using pseudo-projections for the projected databases that can be held in main memory further improves the efficiency of *PrefixSpan.* As can be seen from the figure, the performance of level-by-level and bi-level pseudo-projections are close. The runtime of the two methods are very close when the support threshold is very low. The bi-level projection method is more efficient when the savings due to fewer projected databases overcomes the cost of mining and filling the *S-matrix.* That verifies our analysis

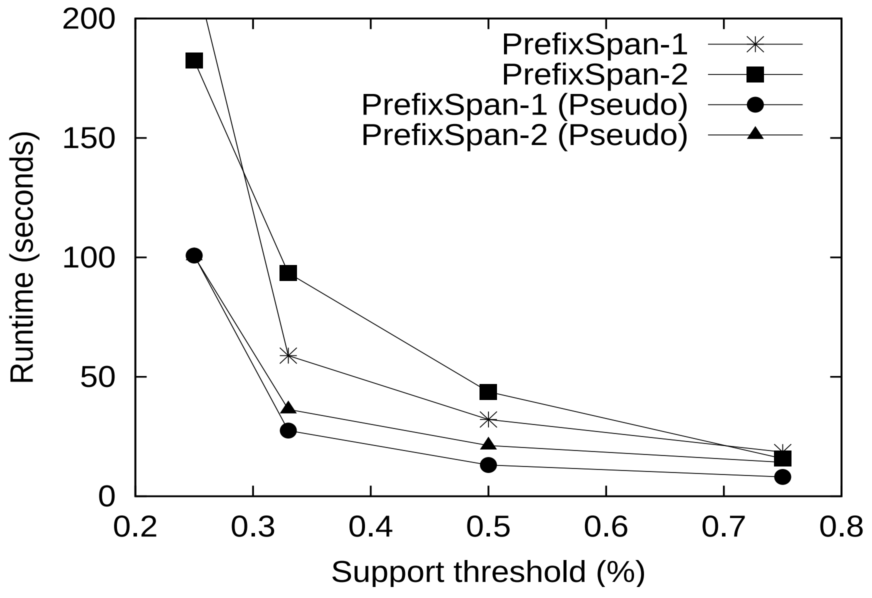

Figure 6.3: Runtime comparison among *PrefixSpan* variations on data set $C10T8S8I8$.

of level-by-level and bi-level projection.

Since pseudo-projection improves performance when the projected database can be held in main memory, it is of interest to consider whether such a method can be extended to disk-based processing. That is, instead of doing physical projection and saving the projected databases in hard disk, should we make the projected database in the form of disk address and offset? To explore such an alternative, we pursue a simulation test as follows.

Let each sequential read, i.e., reading bytes in a data file from the beginning to the end, cost 1 unit of I/O. Let each random read, i.e., reading data according to its offset in the file, cost 1.5 units of I/O. Also, suppose that a write operation costs 1.5 I/O. Figure 6.4 shows the I/O costs of *PrefixSpan-1* and *PrefixSpan-2* as well as of their pseudo-projection variations over data set $C1kT8S8I8$ (where $C1k$ means 1 million sequences in the data set). *PrefixSpan-1* and *PrefixSpan-2* beat their pseudo-projection variations clearly. It can also be observed that bi-level projection outperforms level-by-level projection as the support threshold becomes low. The huge number of random reads in disk-based pseudo-projections is the major cost when the database is too big to fit into main memory.

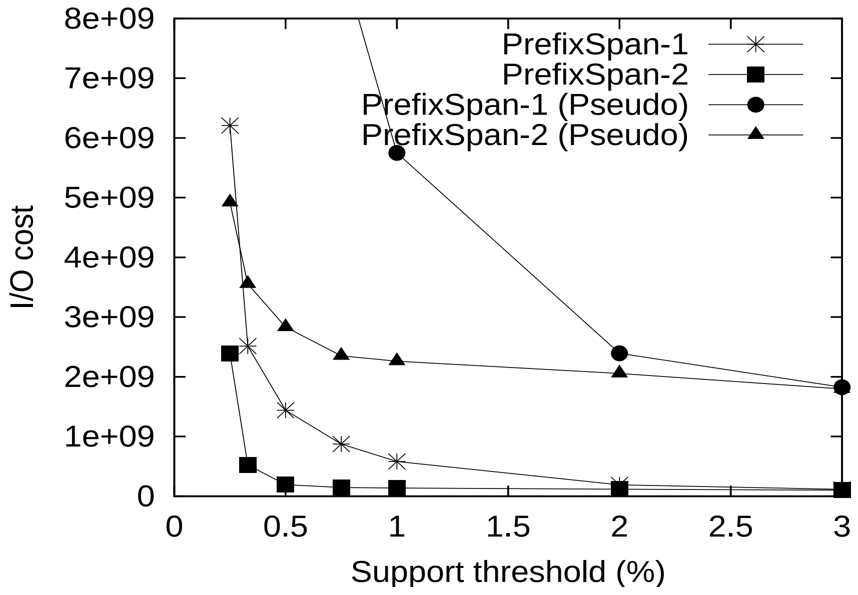

Figure 6.4: I/O cost comparison among *PrefixSpan* variations on data set $C1kT8S8I8$.

Figure 6.5 shows the scalability of *PrefixSpan-1* and *PrefixSpan-2* with respect to the number of sequences. Both methods are linearly scalable. Since the support threshold is set to 0.20%, *PrefixSpan-2* performs better.

In summary, our performance study shows that *PrefixSpan* is more efficient and scalable than *FreeSpan* and *GSP*, whereas *FreeSpan* is faster than *GSP* when the support threshold is low and when there are many long patterns. Since *PrefixSpan-2* uses bi-level projection to dramatically reduce the number of projections, it is more efficient than *PrefixSpan-1* in large databases with low support threshold. Once the projected databases can fit in main memory, pseudo-projection always leads to the most efficient solution. The experimental results are consistent with our theoretical analysis.

## 6.5 Discussion

As supported by our analysis and performance study, both *PrefixSpan* and *FreeSpan* are faster than *GSP*, and *PrefixSpan* is also faster than *FreeSpan*. Here, we summarize the

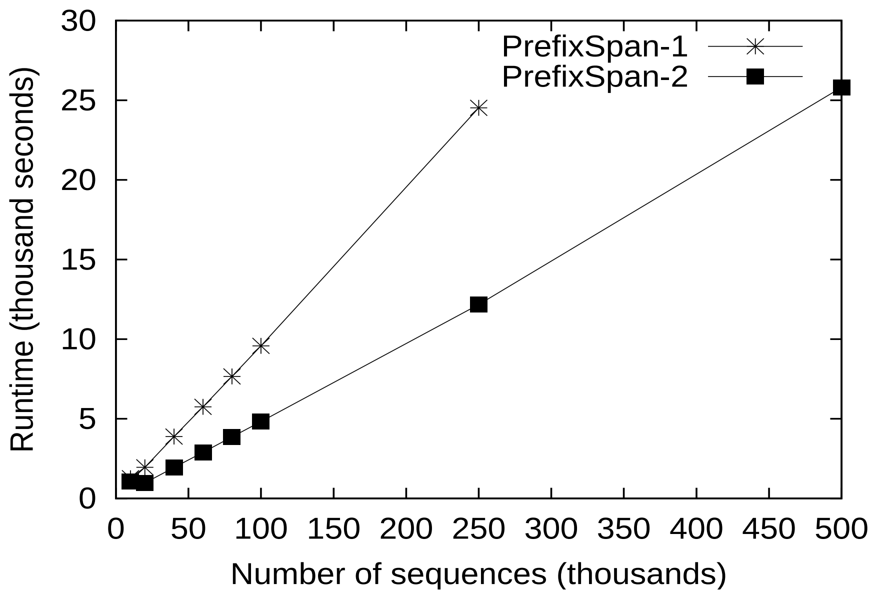

Figure 6.5: Scalability of *PrefixSpan* variations.

factors contributing to the efficiency of *PrefixSpan*, *FreeSpan*, and *GSP* as follows.

- *Both* PrefixSpan *and* FreeSpan *are pattern-growth methods, their searches are more focused and thus more efficient than* GSP. Pattern-growth method try to grow longer patterns from shorter ones. Accordingly, they divide the search space and focus on only the subspace potentially supporting further pattern growth at a time. Thus, their search spaces are more focused. Technically, their search spaces are confined by projected databases. A projected database for a sequential pattern $\alpha$ contains all and only the necessary information for mining the sequential patterns that can be grown from $\alpha$. As mining proceeds to longer sequential patterns, projected databases become smaller. In contrast, *GSP* always searches the original database at each iteration. Many irrelevant sequences have to be scanned and checked, which is not fruitful.

  This argument is supported by our performance study. Figure 6.2 indicates that the average number of candidates per sequential pattern in *GSP* increases exponentially when the support threshold goes down, while the average processing time for each projected database in *PrefixSpan* goes down dramatically.

- *Prefix-projected pattern growth is more elegant than frequent pattern-guided projection.*

Comparing with frequent pattern-guided projection, employed in *FreeSpan*, prefix-projected pattern growth is more progressive. Even in the worst case, *PrefixSpan* still guarantees that projected databases keep shrinking and only takes care suffixes. When mining in dense databases where *FreeSpan* cannot gain much from projections, *PrefixSpan* can still reduce both the length of sequences and the number of sequences in projected databases dramatically.

For example, suppose the database contains only one sequence $\langle a_1 a_2 \cdots a_{100} \rangle$ and the support threshold is set to 1. Let us consider the projected databases without the optimization of predominant prefix. The $\langle a_1 \rangle$-projected database contains sequence $\langle a_2 a_3 \cdots a_{100} \rangle$, while the $\langle a_1 a_2 \rangle$-projected database contains $\langle a_3 a_4 \cdots a_{100} \rangle$. As the sequential pattern becomes longer, the sequences in corresponding projected databases becomes shorter. *PrefixSpan* only needs to find patterns from those suffixes. However, in *FreeSpan*, if the order of frequent items are $a_{100}, a_{99}, \ldots, a_1$, the $\{a_1\}$-, $\{a_1, a_2\}$-, $\ldots$, $\{a_1, a_2, \ldots, a_{99}\}$-projected databases all contain the original sequence $\langle a_1 a_2 \cdots a_{100} \rangle$. In such cases, the sequence in projected databases does not shrink. Furthermore, *FreeSpan* has to take care pattern growth at every possible "pattern growth" point in the pattern template. In a sequential pattern with $n$ elements, there are in total $(2n+1)$ such points, where $n$ of them enable inserting an item into an existing element and a single-item element can be inserted into another $(n+1)$ possible points. This is quite costly.

- *The Apriori principle is integrated in bi-level projection* PrefixSpan*.* The *Apriori* heuristic is the essence in the *Apriori*-like methods. However, the *Apriori*-like methods generate and test many candidates. Can we still fully utilize the *Apriori* heuristic but avoid costly candidate generation-and-test?

  Notice that mining on frequent-pattern projected database itself utilizes the *Apriori* heuristic since only the subsequences related to the frequent patterns in the current databases will be projected and be examined subsequently. Moreover, both our theoretical analysis and experimental results support our claim that bi-level projection is more efficient than level-by-level projection in *PrefixSpan*. Bi-level projection integrates *Apriori* heuristic in its pruning projected databases. Based on the heuristic, bi-level projection achieves a 3-way checking to determine if a sequential pattern can lead to a longer pattern and which items can be used to assemble longer patterns potentially. Only the fruitful projected databases will be constructed. Furthermore, 3-way checking is efficient since only corresponding cells in $S$-*matrix* are checked, while no further assembling is needed.

## 6.6 Summary

We have studied methods for mining sequential patterns in large databases and developed a *pattern-growth approach* for efficient and scalable mining of sequential patterns. Our approach is not a direct extension of a candidate generation-and-test approach, such as GSP [SA96a]. Instead, it is an exploration of a divide-and-conquer, pattern-growth approach, which can be viewed in spirit as an extension of the *FP-growth* algorithm [HPY00] developed for mining (non-sequential) frequent patterns in databases. Our new approach explores a database projection method and grows sequential patterns from locally frequent fragments in the projected databases.

Two pattern growth methods, *FreeSpan* and *PrefixSpan*, are proposed. Both methods mine the complete set of sequential patterns but substantially reduce the efforts of candidate subsequence generation. *PrefixSpan* explores only the prefix-based projection and thus leads to less search spaces and better efficiency than *FreeSpan*. To further improve efficiency, three kinds of database projections: *level-by-level projection*, *bi-level projection*, and *pseudo-projection*, are explored. Our comprehensive performance study shows that *FreeSpan* and *PrefixSpan* outperform *Apriori*-based *GSP* algorithm, and *PrefixSpan* with database projection techniques, especially pseudo-projection when data fits in main memory or bi-level projection otherwise, is the fastest one in mining large sequence databases.

There are many interesting issues which need to be studied further. For example, users may often pose constraints on the sequential patterns to be found. It is an interesting research problem to see how to make full use of these constraints in sequential pattern mining. Also, many sequential pattern mining problems in applications, such as DNA mining, may admit faults, such as allowing insertions, deletions and mutations in DNA sequences. It is another interesting issue to develop efficient fault-tolerant sequential pattern mining algorithms for many applications.

# Chapter 7

# Discussion

We have developed pattern-growth methods for efficient and effective frequent pattern mining. In this chapter, we first summarize the major characteristics of pattern-growth methods, then discuss some interesting extensions and applications of pattern-growth methods.

## 7.1 Characteristics of Pattern-growth Methods

We have developed a new class of pattern-growth methods for effective and efficient data mining. We summarize the major characteristics of pattern-growth methods here.

- *Pattern-growth methods adopt a divide-and-conquer methodology and partition both the data sets and patterns into subsets recursively.* In general, data mining has to search a very huge space. Divide-and-conquer methodology enables the search algorithms to focus on reduced subsets of goals within much smaller sub-spaces. That makes sharper pruning feasible.

- *Pattern-growth methods avoid candidate-generation-and-test.* Instead, pattern-growth methods take the patterns found as seeds and explore extensions of current patterns. The benefits are from two aspects. On one hand, pattern-growth methods search much less than candidate-generation-and-test methods do, since the number of candidates could be huge. On the other hand, pattern-growth methods avoid most of the expensive pattern matching operations. Instead, they search for local frequent items, which is much cheaper.

- *Pattern-growth methods employ effective data structures to fully utilize the available space.* For example, both *FP-tree* and *H-block* try to fully use the available memory and reduce irrelevant information. With highly condensed and well indexed data structures, the search space is presented in a well organized way so that the pattern-growth search can be much more efficient.

## 7.2 Extensions and Applications of Pattern-growth Methods

We have shown that pattern-growth methods are effective and efficient in frequent pattern mining. Interestingly and surprisingly, pattern-growth methods are also applicable to mining other kinds of knowledge and solving some other interesting data processing problems. In this section, we discuss some examples.

### 7.2.1 Mining Closed Association Rules

Association mining may often derive an undesirably large set of frequent itemsets and association rules. There is an interesting alternative, proposed recently by Pasquier, et al. [PBTL99]: *instead of mining the complete set of frequent itemsets and their associations, association mining only needs to find frequent closed itemsets and their corresponding rules.* An important implication is that mining frequent closed itemsets has the same power as mining the complete set of frequent itemsets, but it substantially reduces the redundant rules generated and increases both efficiency and effectiveness of mining.

Let's examine a simple example. Suppose that a database contains only two transactions, $\{(a_1,\ a_2,\ \ldots,\ a_{100}),\ (a_1, a_2, \ldots, a_{50})\}$, the minimum support threshold is 1 (i.e., every occurrence is frequent), and the minimum confidence threshold is 50%. The traditional association mining method will generate $2^{100} - 1 \approx 10^{30}$ frequent itemsets, which are $(a_1)$, $\ldots$, $(a_{100})$, $(a_1, a_2)$, $\ldots$, $(a_{99}, a_{100})$, $\ldots$, $(a_1, a_2, \ldots, a_{100})$, and a tremendous number of association rules, whereas the frequent closed itemset mining will generate only two frequent closed itemsets: $\{(a_1, a_2, \ldots, a_{50}), (a_1, a_2, \ldots, a_{100})\}$, and one association rule, $(a_1, a_2, \ldots, a_{50}) \Rightarrow (a_{51}, a_{52}, \ldots, a_{100})$, since all others can be derived from this one easily.

In [PHM00], we studied efficient mining of frequent closed itemsets in large databases using pattern-growth methods. To mine frequent closed itemsets, Pasquier, et al. [PBTL99] proposed an *Apriori*-based mining algorithm, called *A-close*. Zaki and Hsiao [ZH99] proposed another mining algorithm, *ChARM*, which improves mining efficiency by exploring an item-based data structure. According to our analysis, *A-close* and *ChARM* are still costly when mining long patterns or with low minimum support thresholds in large databases. As a continued study on frequent pattern mining without candidate generation [HPY00], we proposed an efficient method for mining closed itemsets [PHM00]. Three techniques were developed for this purpose: (1) the framework of a recently developed efficient frequent pattern mining method, *FP-growth* [HPY00], is extended, (2) strategies are devised to reduce the search space dramatically and identify the frequent closed itemsets quickly, and (3) a partition-based projection mechanism is established to make the mining efficient and scalable for large databases. Our performance study showed that *CLOSET* is efficient, scalable over large databases, and faster than the previously proposed methods.

### 7.2.2 Associative Classification Using Pattern-growth Methods

Associative classification has attracted increasing attention in the data mining community due to its improved classification accuracy and its flexibility at handling unstructured data. However, associative classification still suffers from the huge set of mined rules and overfitting since the classification is based on rules with highest confidence.

In [LHP01], we proposed a new associative classification method, *CMAR*, i.e., **C**lassification based on **M**ultiple **A**ssociation **R**ules, which performs associative classification based on multiple strong association rules. The method extends an efficient frequent pattern mining method, *FP-growth*, constructs a class distribution-associated *FP-tree*, and mines large databases efficiently. Moreover, it applies a *CR-tree* structure to store and retrieve mined association rules efficiently, and prunes rules effectively based on confidence, correlation and database coverage. The classification is performed based on a weighted $\chi^2$ analysis using multiple association rules.

Our experiments were performed on 26 databases in the popularly referenced UC-Irvine machine learning database repository. The experimental results showed that *CMAR* is consistent, highly effective at classification of various kinds of databases and has better average classification accuracy in comparison with CBA and C4.5. Moreover, our performance study showed that the method is highly efficient and scalable in comparison with other reported associative classification methods.

### 7.2.3 Mining Multi-dimensional Sequential Patterns

Sequential pattern mining, which finds the set of frequent subsequences in sequence databases, is an important data-mining task and has broad applications. Usually, sequence patterns are associated with different circumstances, and such circumstances form a multiple dimensional space. For example, customer purchase sequences are associated with region, time, customer group, and others. It is interesting and useful to mine sequential patterns associated with multi-dimensional information.

In [PHP$^+$01], we proposed the theme of multi-dimensional sequential pattern mining, which integrates multi-dimensional analysis and sequential data mining. We also thoroughly explored efficient methods for multi-dimensional sequential pattern mining. We examined feasible combinations of efficient sequential pattern mining and multi-dimensional analysis methods, as well as developed uniform methods for high-performance mining. Extensive experiments showed the advantages as well as limitations of these methods. Some recommendations on selecting proper method with respect to data set properties were drawn.

### 7.2.4 Computing Iceberg Cubes with Complex Measures

Data cube is an essential facility for online analytical processing. It is often too expensive to compute and materialize a complete high-dimensional data cube. Computing an iceberg cube is an effective way to derive nontrivial multi-dimensional aggregations for OLAP, data mining, data compression, and other applications. An *iceberg cube* is a set of all cells in a data cube that satisfy certain constraints, such as a minimum support threshold. Previous studies developed some efficient methods for computing iceberg cubes for simple measures, such as *count* and *sum of nonnegative values*. However, it is still a challenging problem to efficiently compute iceberg cubes with complex measures, such as *average, sum of mixture of nonnegative and negative values*, etc.

In [HPDW01], we studied efficient methods for computing iceberg cubes with some popularly used complex measures and developed a methodology that uses a weaker but anti-monotonic condition for testing and pruning search space. In particular, we investigated efficient methods for computing iceberg cubes with the *average* measure and proposed a *top-k average* pruning method. Moreover, we extended two previously studied methods, *Apriori* and *BUC*, to *Top-k Apriori* and *Top-k BUC*, for the efficient computation of such iceberg cubes. To further improve the performance, two fast algorithms, *H-cubing* and $H^2$-*cubing*, were developed. They employ hyper-structures, *H-tree* and *H-block*, respectively. Our performance study showed that $BUC$, *H-cubing* and $H^2$-*cubing* are promising candidates for scalable computation, and $H^2$-*cubing* has the best performance in many cases.

## 7.3 Summary

In summary, pattern-growth methods adopt a divide-and-conquer methodology and partition both the data sets and the patterns into subsets recursively. They avoid candidate-generation-and-test. In addition, they employ effective data structures to fully utilize the available space.

Our studies show that pattern-growth methods are not only efficient but also effective. They have strong implication to mining other kinds of knowledge and broad applications, such as closed association rule mining, associative classification, multi-dimensional sequential pattern mining, and iceberg cube computation with complex measures.

# Chapter 8

# Conclusions

As our world is now in its information era, a huge amount of data is accumulated everyday. A real universal challenge is to find actionable knowledge from a large amount of data. Data mining is an emerging research direction to meet this challenge. Many kinds of knowledge (patterns) can be mined from various data. In this thesis, we focus on the problem of mining frequent patterns efficiently and effectively, and develop a new class of pattern-growth methods.

In this chapter, we first summarize the thesis, and then discuss some interesting future directions.

## 8.1 Summary of The Thesis

Mining frequent patterns in transaction databases, time-series databases, and many other kinds of databases has been studied extensively in data mining research. Most previous studies adopt an *Apriori*-like candidate set generation-and-test approach. However, candidate set generation is still costly, especially when there exists an abundance of patterns and/or long patterns. In this thesis, we propose a class of pattern-growth methods for the frequent pattern mining and make the following contributions.

- We propose a novel *frequent-pattern tree* (*FP-tree*) structure, which is an extended prefix-tree structure for storing compressed, crucial information about frequent patterns, and develop an efficient *FP-tree*-based mining method, *FP-growth*, for mining *the complete set of frequent patterns* by pattern fragment growth. Efficiency of mining is achieved with three techniques: (1) a large database is compressed into a highly condensed, much smaller data structure, which avoids costly, repeated database scans, (2) our *FP-tree*-based mining adopts a pattern growth method to avoid the costly generation of a large number of candidate sets, and (3) a partitioning-based, divide-and-conquer method is used to decompose the mining task into a set of smaller tasks for mining confined patterns in conditional databases, which dramatically reduces the search space. Our performance study shows that the *FP-growth* method is efficient and scalable for mining both long and short frequent patterns, and is about an order of magnitude faster than the *Apriori* algorithm and also faster than some recently reported new frequent-pattern mining methods.

- One major cost for *FP-growth* is that it has to build conditional *FP-tree*s recursively. To overcome this disadvantage, we propose a simple and novel hyper-linked data structure, *H-struct*, and a new mining algorithm, *H-mine*, which takes advantage of this data structure and dynamically adjusts links in the mining process. A distinct feature of this method is that it has a very limited and precisely predictable space overhead and runs really fast in memory-based setting. Moreover, it scales to very large databases by database partitioning. When the data set becomes dense, (conditional) *FP-tree*s can be constructed dynamically as part of the mining process. Our study shows that *H-mine* has high performance for various kinds of data. It outperforms previously developed algorithms, and is highly scalable in mining large databases. This study also proposes a new data mining methodology, *space-preserving mining*, which may have strong impact on the future development of efficient and scalable data mining methods.

- In many cases, frequent pattern mining may result in too many patterns. Recent work has highlighted the importance of the constraint-based mining paradigm in the context of mining frequent itemsets, associations, correlations, sequential patterns, and many other interesting patterns in large databases. Constraint pushing techniques have been developed for mining frequent patterns and associations with anti-monotonic, monotonic, and succinct constraints. We study constraints which cannot be handled with existing theory and techniques in frequent pattern mining. For example, $avg(S)\ \theta\ v$, $median(S)\ \theta\ v$, $sum(S)\ \theta\ v$ ($S$ may contain items of arbitrary values) are customarily regarded as "tough" constraints as they cannot be pushed inside an algorithm such as *Apriori*. We develop a notion of *convertible constraints* and systematically analyze, classify, and characterize this class of constraints. We also develop techniques which enable them to be readily pushed deep inside the recently developed *FP-growth* algorithm for frequent itemset mining. Results from detailed experiments show the effectiveness of the techniques we developed.

- Sequential pattern mining is an important data mining problem in time-related or sequence databases with broad applications. It is also a difficult problem since one may need to examine a combinatorially explosive number of possible subsequence patterns. Most of the previously developed sequential pattern mining methods follow the *Apriori* methodology since the *Apriori*-based method may substantially reduce the number of combinations to be examined. However, *Apriori* still encounters performance challenges when a sequence database is large and/or when sequential patterns are numerous and/or long.

  We systematically develop a *pattern-growth approach* for efficient mining of sequential patterns in large databases. It is not based on the *GSP* (*generalized sequential pattern*) algorithm [SA96a], a candidate generation-and-test approach extended from the *Apriori* algorithm [AS94]. Instead, this new approach adopts a divide-and-conquer, pattern-growth principle, by extending the *FP-growth* algorithm [HPY00] to mine (order-dependent) sequential patterns. The general idea is that a sequence database is recursively projected into a set of smaller projected databases. Sequential patterns are grown in each projected database by exploring only local frequent fragments. Two pattern growth methods, *FreeSpan* and *PrefixSpan*, are proposed. Both methods mine the complete set of sequential patterns but substantially reduce the effort of candidate subsequence generation. To further improve mining efficiency, three kinds of database projections: *level-by-level projection*, *bi-level projection*, and *pseudo-projection*, are explored. A comprehensive performance study shows that *FreeSpan* and *PrefixSpan* outperform the *Apriori*-based *GSP* algorithm, and an integrated *PrefixSpan* is the fastest algorithm for mining large sequence databases.

## 8.2 Future Research Directions

With the success of pattern-growth methods, it is interesting to re-examine and explore many related problems, extensions and applications. Some of them are listed here.

- *Fault-tolerant frequent pattern mining.* Real-world data tends to be dirty. Discovering knowledge over large real-world data calls for fault-tolerant data mining, which is a fruitful direction for future data mining research. Fault-tolerant extensions of data mining techniques gain useful insights into the data.

  In [PTH01], we introduced the problem of fault-tolerant frequent pattern mining. With fault-tolerant frequent pattern mining, many novel, interesting and practical knowledge can be discovered. For example, one can discover the following fault-tolerant association rules: 85% of students doing well in three out of the four courses: "data structure", "algorithm", "artificial intelligence", and "database", will receive high grades in "data mining".

  *Apriori* was extended to *FT-Apriori* for fault-tolerant frequent pattern mining. Our experimental results showed that *FT-Apriori* is a solid step towards fault-tolerant frequent pattern mining. However, it is still challenging to develop efficient fault-tolerant mining methods. The extensions and implications of related fault-tolerant data mining tasks are very interesting for future research.

- *Frequent pattern-based clustering.* Although there are many clustering algorithms, new challenges exist. On one hand, many real datasets, like web documents, often have very high dimensionality (5000+) and missing dimensional values. On the other hand, many novel applications, like organizing web documents in categories, distance functions are hard to define properly, and clusters can have overlaps. Frequent pattern mining is a very promising candidate technique for such problems.

  Once a set of frequent patterns are found, we can organize objects into some clusters according to the patterns they share. By using this technique, we avoid the problem of defining distance functions and dealing with high dimensionality explicitly.

- *Mining long sequences.* Recently, some emerging applications requires effective and efficient mining of long sequences, such as bio-sequences. The candidate-generation-and-test framework is not feasible to solve such problems, since the number of candidates is prohibitively large. One interesting approach would be to apply the pattern-growth method to bypass trivial patterns during the mining for long target patterns.

### 8.2.1 Final Thoughts

"Discovery consists of seeing what everybody has seen and thinking what nobody has thought."[1] Data mining is towards an effective and efficient tool for discovery. By mining, we can see the patterns hidden behind the data more accurately, more systematically and more efficiently. However, it is the data miner's responsibility to distinguish the gold from the dust.

"Every science begins as philosophy and ends as art."[2] So does data mining.

[1]By Albert von Szent-Györgyi (1893-1986), Hungarian-born American biochemist.

[2]By Will Durant, The Story of Philosophy, 1926.

# Bibliography

[AAP00] R. Agarwal, C. Aggarwal, and V. V. V. Prasad. A tree projection algorithm for generation of frequent itemsets. In *Journal of Parallel and Distributed Computing (Special Issue on High Performance Data Mining)*, 2000.

[AIS93] R. Agrawal, T. Imielinski, and A. Swami. Mining association rules between sets of items in large databases. In *Proc. 1993 ACM-SIGMOD Int. Conf. Management of Data (SIGMOD'93)*, pages 207–216, Washington, DC, May 1993.

[AS94] R. Agrawal and R. Srikant. Fast algorithms for mining association rules. In *Proc. 1994 Int. Conf. Very Large Data Bases (VLDB'94)*, pages 487–499, Santiago, Chile, Sept. 1994.

[AS95] R. Agrawal and R. Srikant. Mining sequential patterns. In *Proc. 1995 Int. Conf. Data Engineering (ICDE'95)*, pages 3–14, Taipei, Taiwan, Mar. 1995.

[BAG99] R. J. Bayardo, R. Agrawal, and D. Gunopulos. Constraint-based rule mining on large, dense data sets. In *Proc. 1999 Int. Conf. Data Engineering (ICDE'99)*, Sydney, Australia, April 1999.

[Bay98] R. J. Bayardo. Efficiently mining long patterns from databases. In *Proc. 1998 ACM-SIGMOD Int. Conf. Management of Data (SIGMOD'98)*, pages 85–93, Seattle, WA, June 1998.

[BMS97] S. Brin, R. Motwani, and C. Silverstein. Beyond market basket: Generalizing association rules to correlations. In *Proc. 1997 ACM-SIGMOD Int. Conf. Management of Data (SIGMOD'97)*, pages 265–276, Tucson, Arizona, May 1997.

[BMUT97] S. Brin, R. Motwani, J. D. Ullman, and S. Tsur. Dynamic itemset counting and implication rules for market basket analysis. In *Proc. 1997 ACM-SIGMOD Int. Conf. Management of Data (SIGMOD'97)*, pages 255–264, Tucson, Arizona, May 1997.

[BR99] K. Beyer and R. Ramakrishnan. Bottom-up computation of sparse and iceberg cubes. In *Proc. 1999 ACM-SIGMOD Int. Conf. Management of Data (SIGMOD'99)*, pages 359–370, Philadelphia, PA, June 1999.

[BWJ98] C. Bettini, X. Sean Wang, and S. Jajodia. Mining temporal relationships with multiple granularities in time sequences. *Data Engineering Bulletin*, 21:32–38, 1998.

[DL99] G. Dong and J. Li. Efficient mining of emerging patterns: Discovering trends and differences. In *Proc. 1999 Int. Conf. Knowledge Discovery and Data Mining (KDD'99)*, pages 43–52, San Diego, CA, Aug. 1999.

[FPSSe96] U. M. Fayyad, G. Piatetsky-Shapiro, P. Smyth, and R. Uthurusamy (eds.). *Advances in Knowledge Discovery and Data Mining.* AAAI/MIT Press, 1996.

[GLW00] G. Grahne, L. Lakshmanan, and X. Wang. Efficient mining of constrained correlated sets. In *Proc. 2000 Int. Conf. Data Engineering (ICDE'00)*, pages 512–521, San Diego, CA, Feb. 2000.

[GRS99] M. Garofalakis, R. Rastogi, and K. Shim. Spirit: Sequential pattern mining with regular expression constraints. In *Proc. 1999 Int. Conf. Very Large Data Bases (VLDB'99)*, pages 223–234, Edinburgh, UK, Sept. 1999.

[HDY99] J. Han, G. Dong, and Y. Yin. Efficient mining of partial periodic patterns in time series database. In *Proc. 1999 Int. Conf. Data Engineering (ICDE'99)*, pages 106–115, Sydney, Australia, April 1999.

[HPDW01] J. Han, J. Pei, G. Dong, and K. Wang. Efficient computation of iceberg cubes with complex measures. In *Proc. 2001 ACM-SIGMOD Int. Conf. Management of Data (SIGMOD'01)*, Santa Barbara, CA, May 2001.

[HPMA$^+$00] J. Han, J. Pei, B. Mortazavi-Asl, Q. Chen, U. Dayal, and M.-C. Hsu. FreeSpan: Frequent pattern-projected sequential pattern mining. In *Proc. 2000 Int. Conf. Knowledge Discovery and Data Mining (KDD'00)*, pages 355–359, Boston, MA, Aug. 2000.

[HPY00] J. Han, J. Pei, and Y. Yin. Mining frequent patterns without candidate generation. In *Proc. 2000 ACM-SIGMOD Int. Conf. Management of Data (SIGMOD'00)*, pages 1–12, Dallas, TX, May 2000.

[KHC97] M. Kamber, J. Han, and J. Y. Chiang. Metarule-guided mining of multi-dimensional association rules using data cubes. In *Proc. 1997 Int. Conf. Knowledge Discovery and Data Mining (KDD'97)*, pages 207–210, Newport Beach, CA, Aug. 1997.

[KMR$^+$94] M. Klemettinen, H. Mannila, P. Ronkainen, H. Toivonen, and A.I. Verkamo. Finding interesting rules from large sets of discovered association rules. In *Proc. 3rd Int. Conf. Information and Knowledge Management*, pages 401–408, Gaithersburg, Maryland, Nov. 1994.

[LHF98] H. Lu, J. Han, and L. Feng. Stock movement and n-dimensional inter-transaction association rules. In *Proc. 1998 SIGMOD Workshop Research Issues on Data Mining and Knowledge Discovery (DMKD'98)*, pages 12:1–12:7, Seattle, WA, June 1998.

[LHM98] B. Liu, W. Hsu, and Y. Ma. Integrating classification and association rule mining. In *Proc. 1998 Int. Conf. Knowledge Discovery and Data Mining (KDD'98)*, pages 80–86, New York, NY, Aug. 1998.

[LHP01] W. Li, J. Han, and J. Pei. Cmar: Accurate and efficient classification based on multiple class-association rules. In *Proc. IEEE 2001 Int. Conf. Data Mining (ICDM'01)*, San Jose, CA., Novermber 2001.

[LNHP99] L. V. S. Lakshmanan, R. Ng, J. Han, and A. Pang. Optimization of constrained frequent set queries with 2-variable constraints. In *Proc. 1999 ACM-SIGMOD Int. Conf. Management of Data (SIGMOD'99)*, pages 157–168, Philadelphia, PA, June 1999.

[LSW97] B. Lent, A. Swami, and J. Widom. Clustering association rules. In *Proc. 1997 Int. Conf. Data Engineering (ICDE'97)*, pages 220–231, Birmingham, England, April 1997.

[MCP98] F. Masseglia, F. Cathala, and P. Poncelet. The psp approach for mining sequential patterns. In *Proc. 1998 European Symp. Principle of Data Mining and Knowledge Discovery (PKDD'98)*, pages 176–184, Nantes, France, Sept. 1998.

[MTV97] H. Mannila, H Toivonen, and A. I. Verkamo. Discovery of frequent episodes in event sequences. *Data Mining and Knowledge Discovery*, 1:259–289, 1997.

[NLHP98] R. Ng, L. V. S. Lakshmanan, J. Han, and A. Pang. Exploratory mining and pruning optimizations of constrained associations rules. In *Proc. 1998 ACM-SIGMOD Int. Conf. Management of Data (SIGMOD'98)*, pages 13–24, Seattle, WA, June 1998.

[ORS98] B. Özden, S. Ramaswamy, and A. Silberschatz. Cyclic association rules. In *Proc. 1998 Int. Conf. Data Engineering (ICDE'98)*, pages 412–421, Orlando, FL, Feb. 1998.

[PBTL99] N. Pasquier, Y. Bastide, R. Taouil, and L. Lakhal. Discovering frequent closed itemsets for association rules. In *Proc. 7th Int. Conf. Database Theory (ICDT'99)*, pages 398–416, Jerusalem, Israel, Jan. 1999.

[PCY95] J.S. Park, M.S. Chen, and P.S. Yu. An effective hash-based algorithm for mining association rules. In *Proc. 1995 ACM-SIGMOD Int. Conf. Management of Data (SIGMOD'95)*, pages 175–186, San Jose, CA, May 1995.

[PH00] J. Pei and J. Han. Can we push more constraints into frequent pattern mining? In *Proc. 2000 Int. Conf. Knowledge Discovery and Data Mining (KDD'00)*, pages 350–354, Boston, MA, Aug. 2000.

[PHL01] J. Pei, J. Han, and L. V. S. Lakshmanan. Mining frequent itemsets with convertible constraints. In *Proc. 2001 Int. Conf. Data Engineering (ICDE'01)*, pages 433–332, Heidelberg, Germany, April 2001.

[PHM00] J. Pei, J. Han, and R. Mao. CLOSET: An efficient algorithm for mining frequent closed itemsets. In *Proc. 2000 ACM-SIGMOD Int. Workshop Data Mining and Knowledge Discovery (DMKD'00)*, pages 11–20, Dallas, TX, May 2000.

[PHMA$^+$01] J. Pei, J. Han, B. Mortazavi-Asl, H. Pinto, Q. Chen, U. Dayal, and M.-C. Hsu. PrefixSpan: Mining sequential patterns efficiently by prefix-projected pattern growth. In *Proc. 2001 Int. Conf. Data Engineering (ICDE'01)*, pages 215–224, Heidelberg, Germany, April 2001.

[PHP$^+$01] H. Pinto, J. Han, J. Pei, K. Wang, Q. Chen, and U. Dayal. Multi-dimensional sequential pattern mining. In *Proc. ACM 2001 Int. Conf. Information and Knowledge Management (CIKM'01)*, Atlanta, Georgia, November 2001.

[PTH01] J. Pei, A. K. H. Tung, and J. Han. Fault-tolerant frequent pattern mining: Problems and challenges. In *Proc. 2001 ACM-SIGMOD Int. Workshop Data Mining and Knowledge Discovery (DMKD'01)*, Santa Barbara, CA, May 2001.

[RMS98] S. Ramaswamy, S. Mahajan, and A. Silberschatz. On the discovery of interesting patterns in association rules. In *Proc. 1998 Int. Conf. Very Large Data Bases (VLDB'98)*, pages 368–379, New York, NY, Aug. 1998.

[Rym92] R. Rymon. Search through systematic set enumeration. In *Proc. 1992 Int. Conf. Principle of Knowledge Representation and Reasoning (KR'92)*, pages 539–550, Cambridge, MA, 1992.

[SA96a] R. Srikant and R. Agrawal. Mining quantitative association rules in large relational tables. In *Proc. 1996 ACM-SIGMOD Int. Conf. Management of Data (SIGMOD'96)*, pages 1–12, Montreal, Canada, June 1996.

[SA96b] R. Srikant and R. Agrawal. Mining sequential patterns: Generalizations and performance improvements. In *Proc. 5th Int. Conf. Extending Database Technology (EDBT'96)*, pages 3–17, Avignon, France, Mar. 1996.

[SBMU98] C. Silverstein, S. Brin, R. Motwani, and J. Ullman. Scalable techniques for mining causal structures. In *Proc. 1998 Int. Conf. Very Large Data Bases (VLDB'98)*, pages 594–605, New York, NY, Aug. 1998.

[SON95] A. Savasere, E. Omiecinski, and S. Navathe. An efficient algorithm for mining association rules in large databases. In *Proc. 1995 Int. Conf. Very Large Data Bases (VLDB'95)*, pages 432–443, Zurich, Switzerland, Sept. 1995.

[STA98] S. Sarawagi, S. Thomas, and R. Agrawal. Integrating association rule mining with relational database systems: Alternatives and implications. In *Proc. 1998 ACM-SIGMOD Int. Conf. Management of Data (SIGMOD'98)*, pages 343–354, Seattle, WA, June 1998.

[SVA97] R. Srikant, Q. Vu, and R. Agrawal. Mining association rules with item constraints. In *Proc. 1997 Int. Conf. Knowledge Discovery and Data Mining (KDD'97)*, pages 67–73, Newport Beach, CA, Aug. 1997.

[Toi96] H. Toivonen. Sampling large databases for association rules. In *Proc. 1996 Int. Conf. Very Large Data Bases (VLDB'96)*, pages 134–145, Bombay, India, Sept. 1996.

[WCM$^+$94] J. Wang, G. Chirn, T. Marr, B. Shapiro, D. Shasha, and K. Zhang. Combinatiorial pattern discovery for scientific data: Some preliminary results. In *Proc. 1994 ACM-SIGMOD Int. Conf. Management of Data (SIGMOD'94)*, pages 115–125, Minneapolis, MN, May, 1994.

[Web95] G. I. Webb. Opus: An efficient admissible algorithm for unordered search. *Journal of Artificial Intelligence Research*, 3:431–465, 1995.

[Zak98] M. J. Zaki. Efficient enumeration of frequent sequences. In *Proc. 7th Int. Conf. Information and Knowledge Management (CIKM'98)*, pages 68–75, Washington DC, Nov. 1998.

[ZH99] M. J. Zaki and C. Hsiao. Charm: An efficient algorithm for closed association rule mining. In *Technical Report 99-10*, Computer Science, Rensselaer Polytechnic Institute, 1999.