MINING ALGORITHMS FOR GENERIC AND BIOLOGICAL DATA

By

JUN LUO

A DISSERTATION PRESENTED TO THE GRADUATE SCHOOL OF THE UNIVERSITY OF FLORIDA IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF DOCTOR OF PHILOSOPHY

UNIVERSITY OF FLORIDA

2002
To my parents and lovely wife
ACKNOWLEDGMENTS

I would like to express my gratitude toward my advisor, Professor Sanguthevar Rajasekaran, for giving me the opportunity to work with him in the Computer and Information Science and Engineering Department. His excellent advice and greatly helpful guidance were critical for me to finish my thesis smoothly and successfully. My full appreciation goes to Dr. Sartaj K. Sahni. I admire Dr. Sahni’s excellent achievements and respected personalities. It is my honor to have Dr. Sahni in my committee. I want to thank Dr. Douglas D. Dankel II who never hesitated to give me the advice and help whenever I was in difficulties. I also want to thank Dr. Meera Sitharam and Dr. Joseph N. Wilson for their reviews of my dissertation. Their insightful suggestions gave me a lot of inspirations. My thanks also go to Dr. Tim Davis, Dr. Loc Vu-Quoc, Mr. John Bowers, and Mrs. Ardiniece Y. Caudle, etc. Also, I highly appreciate Dr. Reed Ellis at the Editorial Office in the Graduate School for sparing time reviewing my dissertation carefully and patiently.

Last but not least, I extend my utmost gratitude to my wife, Yanjin E, my mother, Yaqin Wang, and my father, Zhensheng Luo, for their enduring support.
# TABLE OF CONTENTS

| MINING ALGORITHMS FOR GENERIC AND BIOLOGICAL DATA | 1 |
| ACKNOWLEDGMENTS | IV |
| ABSTRACT | VIII |

## 1 INTRODUCTION

1.1 What is Data Mining? | 2 |
1.2 Primary Tasks of Data Mining | 3 |
1.2.1 Mining Association Rules | 3 |
1.2.2 Classification | 4 |
1.2.3 Clustering | 5 |
1.2.4 Incremental Mining | 6 |
1.2.5 Sequential Patterns | 6 |
1.2.6 Regression | 7 |
1.2.7 Web Mining | 7 |
1.2.7 Text Mining | 8 |
1.2.8 Biology Mining | 9 |
1.3 Organization of the Dissertation | 9 |

## 2 MINING ASSOCIATION RULES

2.1 Problem Definitions | 11 |
2.2 Mining Association Rules | 12 |
2.3 Examples of Association Rule Mining | 13 |
2.4 A Survey of Algorithms for Association Rule Mining | 15 |
2.4.1 Apriori, Apriori-like Algorithms | 16 |
2.4.2 Partition and Sampling | 19 |
2.4.3 FP-Growth | 20 |
2.4.4 Mining Multi-level Association Rules | 21 |
2.4.5 Mining Quantitative Association Rules | 21 |
2.4.6 Parallel Algorithms | 21 |

## 3 INTERSECTING ATTRIBUTE LISTS USING A HASH TABLE

3.1 IT Algorithm | 23 |
3.1.1 Basic Idea | 23 |
3.1.2 INSERT Algorithm | 28 |
3.1.3 Performance Analysis ................................................................. 30
3.2 Dynamic Rename Algorithm (DRA) .................................................. 32
3.3 Optimization Methods ........................................................................ 33
  3.3.1 Reorder Frequent Itemset (RFI) ...................................................... 33
  3.3.2 Similarity Detection (SD) ............................................................... 37
  3.3.3 Early Stop Detection (ESD) ............................................................ 37

4 SUPER FAST ALGORITHMS FOR DISCOVERING FREQUENT ITEM-SETS ...... 38
  4.1 FIT Algorithm .................................................................................. 39
  4.1.1 Basic Idea .................................................................................... 39
  4.1.2 FIT Algorithm .............................................................................. 42
  4.1.3 Performance Analysis ................................................................... 44
  4.1.4 Implementation Issues ................................................................. 54
  4.2 SFIT Algorithm ................................................................................ 57
  4.2.1 SFIT Algorithm ........................................................................... 58
  4.2.2 Performance Analysis ................................................................... 59
  4.2.3 Implementation Issues ................................................................. 64

5 PARALLEL ALGORITHMS FOR MINING ASSOCIATION RULES .......... 65
  5.1 Basic Idea ....................................................................................... 65
  5.2 Database Division ........................................................................... 68
    5.2.1 Horizontal Division .................................................................... 68
    5.2.2 Randomized Horizontal Division ............................................... 69
  5.3 Parallel Algorithms .......................................................................... 70
    5.3.1 Multithread Algorithms in the SMP Architecture ....................... 71
    5.3.2 Parallel Algorithms in Distributed Computer System .................. 75

6 (1+\varepsilon)-PASS ALGORITHM .............................................................. 78
  6.2 Problem Description and Related Work ............................................. 79
  6.3 (1+\varepsilon)-Pass Algorithm ............................................................... 82
    6.3.1 Basic Idea ................................................................................ 82
    6.3.2 Description of (1+\varepsilon)-Pass Algorithm .................................. 87
    6.3.3 Implementation Issues ................................................................. 90

7 EFFICIENT ALGORITHMS FOR SIMILARITY SEARCH ....................... 92
  7.1 Abstract ....................................................................................... 92
  7.2 Introduction ................................................................................... 92
  7.3 Sorting Based Algorithm (SBA) ....................................................... 93
  7.4 GST Based Algorithm (GSTBA) ...................................................... 94
    7.4.1 Color Set Size (CSS) Problem ................................................... 96
    7.4.2 The GSTBA Algorithm .............................................................. 98
  7.5 An Experimental Comparison of SBA and GSTBA ......................... 99
  7.6 Conclusions .................................................................................. 100
MINING ALGORITHMS FOR GENERIC AND BIOLOGICAL DATA

By

Jun Luo

Chair: Sanguthevar Rajasekaran
Major Department: Computer and Information Sciences and Engineering

With computer technologies developing so fast, especially since the appearance of the Internet, people have experienced the era of the data explosion. As a result, the sizes of normal databases nowadays could be hundreds of gigabytes or even terabytes. On the other hand, people’s abilities to analyze the collected data are limited. This contradiction creates the need to generate new technologies and tools to analyze the collected data intelligently and efficiently, which sparks the emergence of knowledge discovery in databases (KDD) and data mining.

KDD and data mining could be applied to all kinds of data. The most common application domain of the KDD and data mining techniques is the business databases. Applying association rules and sequential pattern techniques on the market database, a store manager can infer the connections among the commodities sold by the store and thus can manage to promote the sale of various goods and obtain better marketing performance. Another common application domain is the biological databases. The
techniques of data mining such as classification and pattern matching have helped people decode the human genetic code.

This thesis is concerned with the development of efficient methodologies for data mining. We propose to develop techniques that can be applied to a generic database independent of the application. Generic techniques need not perform uniformly well on all the applications of concern. It is often possible to design techniques specific to a given application that will outperform generic techniques. This is indeed true for data mining as well. Thus, in addition to developing generic data mining techniques, we will focus on discovering efficient analysis tools for biological data too.
CHAPTER 1
INTRODUCTION

With computer technologies developing so fast, especially since the appearance of the Internet, people have experienced the era of the data explosion. It has been estimated that the amount of information in the world doubles every 20 months and the size and number of databases are increasing even faster [83]. The large databases have been established for the business, government, and scientific research purposes. For example, Wal-Mart retailer created one of the largest databases in the world that handles over 20 million transactions a day. Another example of the large database is described in Fasman et al. [46], where a human genome database has collected gigabytes of data on the human genetic code and much more is expected. There are other examples such as the gigabytes database for health-care transactions in U.S., which is analyzed by the companies in order to control the health-care coast and improve its quality [37]. The NASA Earth Observing System (EOS) or orbiting satellites and other space-borne instruments generate on the order of 50 gigabytes of remotely sensed image data per hour [43]. Google company, which develops the most prevalent Internet search engine, maintains billions of indexes to different web sites.

On the other hand, although it has been realized that information is crucial for decision making and people have the abilities to collect large amounts of data, people lack the abilities to analyze, summarize and extract useful information from the large data-sets timely and efficiently. This contradiction sparks the evolution of knowledge discovery in databases and data mining.
1.1 What is Data Mining?

There are many similar definitions for data mining [32]. Briefly speaking, data mining could be defined as the extraction of implicit and useful information from the large databases.

The data mining could also be thought of as a step in the process of knowledge discovery in database. The process of the knowledge discovery in database could be illustrated in Figure 1-1.

At the beginning, we develop the understanding of the application domain. Then we collect the data into the database from the corresponding domain.

Depending on what kind of knowledge to be discovered, a set of data, called the target data, is selected or sampled from the data collected.

Normally, we can not extract the knowledge from the target data directly. This is because the target data usually contains the noisy data, or the data that have missing values.

![Figure 1-1 Procedure of Knowledge Discovery in Database](image)
The target data need to be preprocessed. Preprocessing is a cleansing stage. In the preprocessing, the noisy data and the data that have missing values will be handled. Some data that are useless will be removed. Also in the preprocessing, the data representation may also be transformed to the input format of the mining algorithm.

After the preprocessing, the target data become the training data. The training data could be used as the input to the data mining tasks.

Depending on the goals of the discovery, we select the data mining task and apply it to the training data. The output of the data mining task is called the patterns.

We evaluate the patterns. If the patterns are correct, then we store them into the knowledge database. Otherwise, we select another set of target data from the database, and repeat the above procedure.

A more detailed and comprehensive description of the procedure of the knowledge discovery in database could be found in Fayyad et al. [84].

1.2 Primary Tasks of Data Mining

The data mining techniques need to use the knowledge of database, algorithm, machine learning (AI), statistics and probability theory, etc. The primary tasks of data mining are briefly described as follows.

1.2.1 Mining Association Rules

The development of the bar-code technology has made it possible for a business to establish a database of a large number of transactions. Each transaction consists of items that are purchased by the same customer at one time. The mining association rule is designed to analyze such transaction data. More specifically, the task of mining association rules tries to find out the sets of items that have strong associative relations with each other. The results of the analysis could be used for sales promotion.
The techniques for the mining associations include Apriori algorithm, Partition algorithm, FP-growth algorithm, Sampling, Parallel, etc. For more information about the mining association rules, the reader can refer to Hipp et al. [36] for an overview.

1.2.2 Classification

The task of classification is to learn a function that classifies a data item into one of several predefined classes. In classification, we are given a set of example records, called the training data, in which each record consists of several attributes. One of these attributes, called the classifying attribute, is used to specify the class to which each record belongs. The goal of the classification is to build a learning model for the classifying attribute based upon other attribute values by using the training set. After the model is built, it is evaluated for its correctness. Then the model is used to determine the class of the future unclassified records.

The techniques for the classification include SPRINT, CLOUD, etc. For more information about the classification, people can refer to Berson and Smith [2] and Freitas [3].

The applications of the classification are in the areas such as retail target marketing, customer retention, fraud detection and medical diagnosis. For example, in order to predict fraudulent cases in credit card transactions, we could use credit card transactions and the information on its account-holder as attributes (such as when does a customer buy, what does he buy, how often he pays on time, etc.), and also label the past transactions as fraud or fair transactions, which forms the class attribute. Then we use the set of the past transactions to learn a model for the class of the transactions. After the model is established, we could use that model to detect fraud by observing credit card transactions on an account.
1.2.3 Clustering

The clustering is a descriptive task that seeks to identify homogeneous groups of objects based on the values of their attributes. Given a set of objects where each object may have several attributes, the task of the clustering is to group some objects together, where each group is called one cluster, such that the objects in the same cluster are more similar to each other than to objects in the different clusters according to the predefined clustering criterion, such as square-error.

There are two major categories of the techniques solving the clustering problems: one is called partitioning, and the other is called hierarchical. For the partitioning, the methods of $k$-means, $k$-modes and $k$-medoids are widely used. The method of $k$-means first determines $k$ cluster representatives. Then it assigns each object to the cluster with representative closest to the object such that the sum of the square error between the objects and their representatives is minimized. Hierarchical clustering techniques organize the data into a nested sequence of groups. There are two types of hierarchical clustering: top-down (divisive) and bottom-up (agglomerative). The top-down method is as follows: start with all objects in one cluster and subdivide the cluster into smaller pieces. The bottom-up method is as follows: place each object in its own cluster and gradually merges these atomic clusters into larger and larger clusters until all objects are in one cluster. For more information about the clustering, the reader can refer to Berson and Smith [2] and Ng and Han [73].

The hierarchical techniques are popular in biological, social, and behavioral sciences because of the need to construct taxonomies. Partitioning techniques are used frequently in engineering applications in which single partitions are important. It is especially appropriate for the efficient representation and compression of large databases.
1.2.4 Incremental Mining

Incremental mining combines the technology of data mining and active database. The basic idea of the incremental mining is to divide the whole data into several sub-data sets. The data mining algorithms will be applied to each sub-dataset, and the results for each sub-data will be calculated. All the rules generated in each sub-data set will be stored in the rule database with some parameters, such as the support and confidence for association rule problems. When the new data come in, and the volume of new data reach a certain level, the data mining algorithm will be applied to the new data set again and then check the rule database. If the rule exists in the rule database, it will update the parameters of the existing rules in the rule database. The database uses triggers to monitor the parameters of the existing rules in the rule database. Once the conditions are met, the specific trigger to monitor will be fired. Readers can refer to Agrawal and Psaila [70] for further details.

1.2.5 Sequential Patterns

Given a database D of customer transactions, and each transaction consists of the following attributes: customer id, transaction time, and the items purchased in the transaction. No customer has more than one transaction with the same transaction items. An itemset is a non-empty set of items. A sequence is an ordered list of itemsets. A sequence \((a_1,a_2...,a_n)\) is contained in another sequence \((b_1,b_2...,b_n)\) if there exist integers \(i_1 < i_2 < ... < i_n\) such that \(a_1 \subseteq b_{i_1}, a_2 \subseteq b_{i_2},...,a_n \subseteq b_{i_n}\). In a set of sequences, a sequence \(s\) is maximal if \(s\) is not contained in any other sequence. All the transactions of a customer can together be viewed as a sequence, where each transaction corresponds to a set of items, and the list of transactions, ordered by increasing transaction-time, corresponds to
a sequence. Such a sequence is called a customer-sequence. A customer supports a sequence $s$ if $s$ is contained in the customer-sequence for this customer. The support for a sequence is defined as the fraction of total customers who support this sequence. Given a database $D$ of customer transactions, the problem of mining sequential patterns is to find the maximal sequences among all sequences that have a certain user-specified minimum support. Each such maximal sequence represents a sequential pattern. Readers can refer to Pinto, Han, Pei, and Wang [23] and Agrawal and Srikant [69] for more information.

The difference between mining association rules and finding sequential patterns is as follows: the problem of mining association rules is concerned with finding intra-transaction patterns, whereas the problem of finding sequential patterns is concerned with inter-transaction patterns. A pattern in the mining association rules consists of an unordered set of items, whereas a pattern in the finding sequential patterns is an ordered list of sets of items.

1.2.6 Regression

Regression is learning a function that is used to predict a value of a given variable based on the values of other variables, assuming a linear or nonlinear model of dependency among the variables.

There are many applications of regression. For example, the company could predict consumer demand for a new product as a function of advertising expenditure. Weather forecast could predict the wind velocities as the function of temperature, humidity, air pressure, etc.

1.2.7 Web Mining

Web mining can be broadly defined as the discovery and analysis of useful information from the World Wide Web. According to the web site [86], the tasks of web
mining can be divided into two categories: 1) describing the automatic search and retrieval of information and resources available from millions of sites and on-line databases (i.e., discovering which URLs tend to be accessed). This kind of task is called as web content mining or web clustering and association; 2) discovering and analyzing user access patterns from one or more Web servers (i.e., find the order in which URLs tend to be accessed). This kind of task is called web usage mining or sequential analysis.

With the explosive growth of information sources available on the World Wide Web, it has become increasingly necessary for users to utilize automated tools in order to find, extract, filter, and evaluate the desired information and resources. In addition, with the transformation of the Web into the primary tool for electronic commerce, it is imperative for organizations and companies, who have invested millions in Internet and Intranet technologies, to track and analyze user access patterns. These factors give rise to the necessity of creating server-side and client-side intelligent systems that can effectively mine for knowledge both across the Internet and in particular Web localities.

As bad exemplars (outliers) and incomplete data can easily occur in the data set, which is due to a wide variety of reasons inherent to web browsing and logging, thus, Web Mining and Personalization require modeling of an unknown number of overlapping sets in the presence of significant noise and outliers, (i.e., bad exemplars). Moreover, the data sets in Web Mining are extremely large.

1.2.7 Text Mining

Text mining [86] is also known as document information mining, or knowledge discovery in textual databases. It can be envisaged as a leap from data mining or knowledge discovery from (structured) databases. The inputs of the text mining tasks are large collections of unstructured documents in the natural language text. The outputs of
the text mining tasks are the interesting and non-trivial patterns or knowledge extracted from the texts. The information extracted might be the author, title and date of publication of an article, the acronyms defined in a text or the articles mentioned in the bibliography.

As the most natural form of storing and exchanging information is written words, text mining has a very high commercial potential. In fact, a recent study indicated that 80% of a company's information is contained in text documents, such as emails, memos, customer correspondence, and reports. The ability to distil this untapped source of information provides substantial competitive advantages for a company to succeed in the era of a knowledge-based economy.

1.2.8 Biology Mining

Starting from 1990, the U.S. Human Genome Project is a 13-year effort coordinated by the Department of Energy and the National Institutes of Health. The data produced by the thousands of research teams all over the world are collected and organized in databases specialized for particular subjects. Well-known examples are GDB, SWISS-PROT, GenBank, and PDB. The tasks of biology mining [61] are to apply the data mining algorithms such as sequence analysis and association analysis on the collected biological data in order to get the interesting knowledge in the most efficient manner.

1.3 Organization of the Dissertation

The remaining thesis is organized as follows.

In Chapter 2, we will first give a more detailed definition for the task of mining association rules. The description of the well-known Apriori algorithm is followed. In the remaining part of Chapter 2, we give a summary of the algorithms relating to the mining association rules. In Chapter 3, we put forward an algorithm, named intersecting
attribute lists using hash table (IT), which is based on the idea that fundamentally different from Apriori. Also in Chapter 3, we explore the different data representation format by adopting the bitmap format to calculate the sets of frequent item-sets. In the remaining part of Chapter 3, we briefly discuss some optimization methods to improve upon IT. In Chapter 4, based on IT algorithm, we further propose the algorithms, called as Fast Intersecting attribute lists using hash Table (FIT) and Super Fast Intersecting attribute lists using hash table (SFIT), which significantly improve upon IT by reducing the total number of intersection operations performed by IT. As a matter of fact, IT and FIT could be treated as special cases of SFIT. In Chapter 5, we discuss how to calculate the sets of frequent item-sets in parallel. We consider two situations: 1) The computer system is of a so-called shared-memory multiprocessor (SMP) architecture. 2) The computer system is a distributed computer system. To deal with the first situation, we utilize multithread techniques. To deal with the second situation, we put forward the parallel process techniques using message passing methods. In Chapter 6, we deal with the situation when the size of the database is large, and we want to reduce the total number of I/O operations. We study how to scan the data in the database as least as far possible. In Chapter 7, we make a study on finding the pattern similarity in the synthetic biological data. In Chapter 8, we give a discussion on the experimental results using the algorithms introduced in previous chapters.
CHAPTER 2
INTRODUCTION
MINING ASSOCIATION RULES

The maturity of the bar-code technology has made it possible for a business to build databases for storing the transactions that consist of the items purchased by the customer. Starting in the early 1990s, the task of association rule mining, emerging as the result of the necessity to analyze the large amounts of transaction data, has become an important data mining problem that has been studied extensively. Association rule mining is to find out the sets of items that have the tendency to be contained by the same transactions more frequently. The challenge of association rule mining lies in its intrinsically intensive computation characteristics.

2.1 Problem Definitions

Mining association rules can be formally stated as follows.

The problem of mining association rules can be formally stated as follows. Let \( I = \{i_1, i_2, \ldots, i_n\} \) be a set of attributes, called items. An item-set is a set that consists of one or more items, which is a subset of \( I \). \( D \) represents a database that consists of a set of transactions. Each transaction \( t \) in \( D \) contains two parts: a unique transaction identification number (tid) and an item-set. The size of an item-set is defined as the number of items in it. We represent the item-set with size \( d \) as \( d \)-item-set. A transaction \( t \) is said to contain an item \( i \) if the item \( i \) appears in the transaction \( t \). A transaction \( t \) is said to contain an item-set \( X \) if all the items in \( X \) are contained by \( t \). That the support of an item-set \( X \) is \( s \) means there are \( s \) transactions containing \( X \) in the database \( D \). An item-set
$X$ is said to be a frequent item-set if the support of $X$ is greater than or equal to the user-specified minimum support. An association rule can be expressed in the form of $X \Rightarrow Y$, where $X, Y \subseteq I$, and $X \cap Y = \emptyset$. That the rule $X \Rightarrow Y$ is said to have a support of $s$ means $s$ percent of transactions in $D$ contains the item-set of $X \cup Y$. Also, that the rule $X \Rightarrow Y$ is said to have a confidence $c$ means $c$ percent of the transactions that contain the item-set $X$ also contain the item-set $Y$. We use \textit{minsup} to stand for the user-specified minimum support and use \textit{minconf} to stand for the user-specified minimum confidence.

Given a database $D$, a \textit{minsup}, and a \textit{minconf}, the problem of mining association rules is to generate all the association rules whose supports and confidences are greater than or equal to the \textit{minsup} and the \textit{minconf}, respectively.

Mining association rules could be used for business sales promotion. For example, suppose a supermarket manager asked a question: if the supply of the bread discontinued, would the sale of the milk drop? If we found out that 80 percent of customers who bought the milk would also buy the bread, then the answer to the above question should be \textit{yes}. To avoid the drop of the bread sale, the supermarket manager had better maintain the supply of the milk.

### 2.2 Mining Association Rules

Generally speaking, the task of mining association rules consists of two steps: the first step is to discover the entire frequent item-sets from the database $D$. The second step is to calculate the rules, whose confidences are no less than the \textit{minconf}, from the frequent item-sets that have been discovered. Between the two steps, the discovery of the entire frequent item-sets plays a prominent role, since after the frequent item-sets are found, association rules are easy to be calculated by using the conditional probability formula of
\[ P(b \mid a) = \frac{P(a \cup b)}{P(a)} = \frac{P(X)}{P(a)} \], where \( X \), \( a \) and \( b \) are all item-sets and \( a \subseteq X \), \( b = X - a \). The conditional probability formula addresses the issue of how to calculate the probability of event \( b \) under the condition that event \( a \) occurs. So the problem of mining association rules is sometimes referred to as the problem of discovery of frequent item-sets. The discovery of frequent item-sets is often a big challenge to people for its intensive computations.

One of the most important properties about the frequent item-set is pointed out by Agrawal, Mannila, Srikant, Toivonen and Verkamo [65] and Agrawal and Srikant [68]. That is, if an item-set \( X \) is a frequent item-set with size \( k \), here \( k > 1 \). Then any subset of \( X \) is also a frequent item-set. The property is very useful for us to decide when to stop the calculation of frequent item-sets, i.e., when all the frequent item-sets from the databases have been discovered.

### 2.3 Examples of Association Rule Mining

In Figure 2-1, the database in (a) consists of four transactions, each of which has a unique transaction identification number (tid), such as 100, 200, 300 and 400. Assume the user-specified minimum support is 2, and the user-specified minimum confidence is 50%. By examining the database, we could find out all the item-sets whose supports are no less than the minimum support. The set of all the frequent item-sets is displayed in (b). For example, the item-set \{Bagel\} is contained by the first three transactions in (a), so its support is 3. Similarly, the item-set \{Bagel, Milk\} is contained by the 2\textsuperscript{nd} and 3\textsuperscript{rd} transactions, so its support is 2. The support calculations for other frequent item-sets are the same.
minsup = 2
minconf = 50%

<table>
<thead>
<tr>
<th>Tid</th>
<th>Transactions</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>Bagel, Beer, Potato Chips</td>
</tr>
<tr>
<td>200</td>
<td>Bagel, Coke, Milk</td>
</tr>
<tr>
<td>300</td>
<td>Bagel, Coke, Egg, Milk</td>
</tr>
<tr>
<td>400</td>
<td>Coke, Potato Chips</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Item-sets</th>
<th>Support</th>
</tr>
</thead>
<tbody>
<tr>
<td>{Bagel}</td>
<td>3</td>
</tr>
<tr>
<td>{Coke}</td>
<td>3</td>
</tr>
<tr>
<td>{Potato Chips}</td>
<td>2</td>
</tr>
<tr>
<td>{Milk}</td>
<td>2</td>
</tr>
<tr>
<td>{Bagel, Coke}</td>
<td>2</td>
</tr>
<tr>
<td>{Bagel, Milk}</td>
<td>2</td>
</tr>
</tbody>
</table>

(c)

Figure 2-1 An example of association rule mining

After calculating the support values for all the frequent item-sets in (b), it will be straightforward for us to calculate the confidence of the rules using the conditional possibility formula as mentioned above. For example, to calculate the confidence of the rule Coke $\Rightarrow$ Bagel, we use the support of \{Coke\} to divide the support of \{Bagel, Coke\}. The result is $\frac{2}{3} \times 100\% = 67\%$. The procedure of confidence calculation for the rule Coke $\Rightarrow$ Bagel is the same.

Some notations that we will consistently deploy in the rest of the dissertation are displayed in Table 2-1.
### Table 2-1 Notations

<table>
<thead>
<tr>
<th>Symbols</th>
<th>Remarks</th>
</tr>
</thead>
<tbody>
<tr>
<td>$L_k$</td>
<td>The set of frequent $k$-itesets.</td>
</tr>
<tr>
<td>$</td>
<td>L_k</td>
</tr>
<tr>
<td>$C_k$</td>
<td>The set of candidate frequent $k$-item-sets. $C_k$ is a superset of $L_k$.</td>
</tr>
<tr>
<td>$l_n$</td>
<td>An attribute list in $L_k$, here $1 \leq n \leq</td>
</tr>
<tr>
<td>$</td>
<td>l_n</td>
</tr>
</tbody>
</table>

#### 2.4 A Survey of Algorithms for Association Rule Mining

We are now ready to give a brief summary of the algorithms in the field of mining association rules. There exist some criteria to divide the algorithms into different categories. For example, one criterion could be the classical association rule mining, which is to find out all the frequent item-sets and enumerate all the association rules. According to this criterion, Apriori algorithm [70], DHP [41], Sampling [28], FP-Growth [38], and Partition [6] are all classified with this category. On the other hand, mining multiple-level association rules [34], mining quantitative association rules [75], and CHARM [54] belong to the non-classical mining association rules. Another example of the criterion to divide the algorithms to different groups may be sequential or parallel. All the algorithms mentioned above belong to the sequential category. Freitas and Lavington [4], Han, Karypis and Kumar [16], Zaki, Parthasarathy, Ogihara and Li [30], Zaki, Ogihara, Parthasarathy, and Li [55], Agrawal and Shafer [67] proposed algorithms for solving the problem in parallel.
2.4.1 Apriori, Apriori-like Algorithms

Agrawal and Srikant proposed Apriori algorithm. In order to identify the sets of frequent item-sets in the database $D$, the Apriori algorithm needs to scan the database multiple times. Each scan is called a pass. In the first pass, the set $L_1$ of frequent single items is calculated. In any subsequent pass $k$, where $k > 1$, the Apriori algorithm first generates the set $C_k$ of candidate frequent $k$-item-sets from $L_{k-1}$. $C_k$ is a superset of $L_k$. To find out $L_k$, the database is scanned for another time. During the scan, the support of each item-set in $C_k$ is counted. At the end of the scan, the item-sets whose supports are no less than the minimum support are selected and placed into $L_k$. Then the value of $k$ is increased by 1. The above procedure is repeated and continues until there are no new frequent item-sets generated.

The generation of candidate frequent $k$-item-sets $C_k$ from frequent $(k-1)$-item-set $L_{k-1}$ consists of two steps. In the first step, the set $L_{k-1}$ is scanned once. Any two frequent $(k-1)$-item-sets whose first corresponding $k-2$ items are the same are combined to generate a new $k$-item-set, denoted as $p$ here. In the second step, for each $p$’s subset, denoted as $sp$ here, that is $(k-1)$ item-set, we check to see if $sp$ is contained by $L_{k-1}$. If all of the subsets of $p$ are contained by $L_{k-1}$, then $p$ is placed into $C_k$. Otherwise, $p$ is discarded. This second step is also called as the prune step. The theory behind the prune step is that if an item-set is a frequent one, then any subset of it should also be a frequent item-set.

Apriori uses a hash tree to store the sets of (candidate) frequent item-sets and to calculate the support values for the item-sets. Compared with other tree data structures, the hash tree has a better memory space utilization, since the null pointers in the tree have been significantly reduced.
Although Apriori is a fast algorithm in calculating the frequent item-sets, it has its drawbacks. As pointed out in many papers and also proved in our experiments, the procedure of support calculations for the candidate frequent item-sets in $C_k$ is very expensive in some situations. More specifically, during the scan of the database, for each transaction $t$, Apriori checks to see if any subset, which consists of $k$ items, of $t$ is contained in the hash tree. Given a transaction $t$, the number of items contained by $t$ is denoted as $|t|$, then the total number of subsets with size $k$ is determined by the value of $\binom{|t|}{k}$. It is known that the value of $\binom{|t|}{k}$ is upper bounded by $\binom{|t|}{|t|/2}$, here the symbol ‘/’ represents the integer division. When the value of $k$ increases in the range of $(1, |t|/2)$, the number of subsets of $t$ grows rapidly. Assume there is a transaction, $t$, that consists of 20 items. Figure 2-2 shows the number of subsets of $t$ when the size of subsets increases. Note that, the larger the size of the subset, the more cost of the support calculation. This is because when the size of the subsets is larger, more comparisons during the calculation are expected.
Agrawal and Srikant also designed AprioriTID on the basis of Apriori. Instead of relying on scanning the data stored in a database to accumulate the support values for the candidate frequent item-sets, AprioriTID keeps transactions in the form of sets of frequent item-sets. As the size of frequent item-sets increases, the number of sets of frequent item-sets in each transaction will decrease. So the support calculations will be faster. As Apriori is efficient in computing the item-sets with small sizes, and AprioriTID is good at calculating the item-sets with large sizes, Agrawal and Srikant further combined Apriori and AprioriTID into AprioriHybrid. At the beginning, AprioriHybrid utilizes Apriori. Then at the some point of the calculations, AprioriHybrid turns to deploy AprioriTid algorithm. As there are some overhead costs for transforming the data in the database to the data representation format used in AprioriTID, there is a performance risk for AprioriHybrid when after transforming from Apriori to AprioriTID, the calculations for the frequent item-sets from the database happens to be near to the end.

Apriori deploys a level-by-level calculation strategy. All the frequent $k$-item-sets are computed from the databases before the calculation for the $(k+1)$-item-sets begins. There are also some other papers [7][21][60][73][75][76][81] which adopt the Apriori-like approach.

Based on Apriori, Park, Chen and Yu put forward DHP (Direct Hashing and Pruning) algorithm. DHP improves upon Apriori by reducing the candidate frequent item-sets to be generated. During each round of support calculations for the candidate frequent item-sets in $C_k$, DHP also calculates the support values for buckets that contain a set of $(k+1)$-item-sets. When DHP generates a candidate frequent $(k+1)$-item-set, say $X$, from the
calculated frequent $k$-item-sets, in addition to check the subsets of $X$ to be frequent item-sets, it also checks to see if the support of the bucket that contains $X$ is no less than the $\text{minsup}$. If it is true, $X$ is generated. Otherwise, $X$ is discarded.

**2.4.2 Partition and Sampling**

When the size of the database to be analyzed is large, the computer’s main memory may not be large enough to hold all the data from the database at the same time. In such a situation, computations for the frequent item-sets will involve in large amounts of I/O operations. As a result, the effect of I/O overheads on the computation performance is expected to be significant. Some papers are proposed to address such issues.

Savasere, Omiecinski, and Navathe put forward Partition algorithm. When the database can not physically fit into the memory, Partition needs to scan the database exactly twice. At the beginning, Partition divides the database into smaller partitions such that each partition can be calculated within the memory independently. Then the item-sets that are frequent from different partitions are combined to form a set of candidate sets. After the combination, the database is scanned for the second time. During the scan, the support value for each candidate item-set is accumulated. At the end of the scan, the item-sets whose support values are no less the $\text{minsup}$ will be the final frequent item-sets of the original database.

Toivonen suggested utilizing the sampling techniques to calculate the sets of frequent item-sets in large databases. The basic idea is to pick a random sample of data first. Then the local frequent item-sets from the sampling data are calculated. To collect the support values for the item-sets, which are calculated from the sampling data, in the sampled database, the sampled database is scanned once. After the scan, the set of frequent item-sets whose supports are no less than the $\text{minsup}$ are the true frequent item-
sets in the sampled database. The sampling data may not cover all the frequent item-sets from the sampled database, since some frequent item-sets that may fail to be calculated from the sampling data. So the calculated frequent item-sets are need to be checked to see if there exists any \( k \)-item-set, denoted as \( X \), such that the support values for \( X \) is unknown but \( X \)'s all subsets that are \((k-1)\)-item-sets are all frequent. If there does exist such item-sets, then they might be missing frequent item-sets, and the database needs to be scanned for another time to verify the support values for the item-sets in question. This procedure may repeat for several times until we are sure that no frequent item-sets from the sampled database are missing. To reduce the number of times to scan the database, Toivonen also proposed the concept of the negative border. The negative border is to predict the set of frequent item-sets in the sampled database more aggressively. Although the negative border may be helpful to reduce the number of scan times, it may also increase the amount of redundant calculations.

2.4.3 FP-Growth

Han, Pei, and Yin put forward FP-Growth algorithm. FP-Growth avoids the phase of candidate frequent item-set generation that is required by the Apriori-like algorithms. FP-Growth scans the database twice. During the first scan, FP-Growth calculates the support values for each single item in \( I \). As a result, it finds out the set \( L_1 \). Then FP-Growth sorts the items in \( L_1 \) in non-decreasing order according to the support values. In the second step, FP-Growth constructs a frequent pattern tree, which is of an extended prefix-tree structure. Initially, only frequent single items have nodes in the tree, and the nodes in the tree are arranged in such a way that more frequently occurring nodes will have better chances to share nodes than less frequently occurring nodes. The frequent item-set patterns could be found by establishing new frequent pattern tree using the items
appear in the suffix pattern. The advantage of the FP-Growth tree is that it does not require the candidate frequent item-set generation phase. The disadvantage is when the number of items in $I$ is large the algorithm requires huge amounts of main memory.

2.4.4 Mining Multi-level Association Rules

Han and Fu defined the problem of mining multiple-level association rules. Some applications may need to explore the multiple abstract levels to obtain the interesting information. For example, instead of finding the associative relations between milk and bread, we might be more interested in knowing the association relation between the wheat bread and the 2% reduced milk. The later association relationship expressed more specific information on the lower level of abstract level.

2.4.5 Mining Quantitative Association Rules

Srikant and Agrawal defined the problem of mining quantitative association rules in large relational databases. Traditional mining association rules could be viewed as a boolean association rule problem in the sense that all the attributes in the rule have the boolean values. In reality, the attributes might be of quantitative values such as age and income, or categorical values such as the model of the car. The problem of mining quantitative association rules is to find out the rules over the quantitative or the categorical attributes. For example, a quantitative rule can be expressed as $[\text{Age: } 30-39]$ and $[\text{Married: yes}] \Rightarrow [\text{Number of cars: } 2]$.

2.4.6 Parallel Algorithms

To calculate the frequent item-sets in the parallel or distributed computing environments, some special issues, such as the communication costs, synchronizations, etc, need to be addressed. Several parallel algorithms based on Apriori have been published. Agrawal and Shafer [69] proposed three parallel algorithms on shared-nothing
parallel machines. The algorithms are designed for dealing with two different situations: 1) the candidate item-sets are duplicated and stored in every processor; 2) only one copy of the item-sets is kept in the system and the data are broadcast among the processors. Zaki, Ogihara, Parthasarathy, and Li [55] put forward the parallel algorithms on the shared-memory multiprocessors by using the multithread techniques.
CHAPTER 3
INTERSECTING ATTRIBUTE LISTS USING A HASH TABLE

In this chapter, we first put forward a fast algorithm for discovering frequent item-sets from large data sets. The algorithm is called as IT\textsuperscript{1}. IT independently adopts the idea of tid-list method that is used in Eclat proposed by Zaki, Parthasarathy, Ogihara, and Li [56]. Compared with Eclat, IT has a better computation efficiency in terms of reduced total comparison times. After introducing IT, we will give a couple of optimization methods based on IT.

3.1 IT Algorithm

3.1.1 Basic Idea

We can bypass the candidate frequent item-set generation phase in Apriori by using the so-called tid-list method. The basic idea of the tid-list method could be explained as follows. If a transaction $t$ contains an item-set $X$, then $t$ is treated as one of the attributes of $X$. All the attributes of $X$ form an attribute list. Each attribute in the list represents a transaction in database D and is described by the transaction’s tid. As each transaction has a unique tid, all the attributes in the same attribute list are distinct. Thus we transform the original database into the attribute list format.

\footnote{IT refers to \textit{Intersecting attribute lists using hash Table}.}
Figure 3-1(a) illustrates a database that consists of four transactions. Each transaction has a unique tid, such as 100. Figure 3-1(b) describes the corresponding attribute list format.

<table>
<thead>
<tr>
<th>tid</th>
<th>transactions</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1, 2, 4</td>
</tr>
<tr>
<td>2</td>
<td>2, 3</td>
</tr>
<tr>
<td>3</td>
<td>1, 3, 4</td>
</tr>
<tr>
<td>4</td>
<td>1, 4</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>item-set</th>
<th>attribute lists</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1, 3, 4</td>
</tr>
<tr>
<td>2</td>
<td>1, 2</td>
</tr>
<tr>
<td>3</td>
<td>2, 3</td>
</tr>
<tr>
<td>4</td>
<td>1, 3, 4</td>
</tr>
</tbody>
</table>

(a) (b)

Figure 3-1 An illustration of transforming a database to an attribute list format

After the transformation, the calculation of frequent item-sets becomes straightforward. The support of the item-set $X$ is determined by the number of attributes in $X$’s attribute list. The support calculation for the union of item-set $X$ and item-set $Y$ consists of two steps. In the first step, the intersection between $X$’s attribute list and $Y$’s attribute list is computed. The attributes that exist in both attribute lists are put into a resulting attribute list. In the second step, the number of attributes in the resulting attribute list is counted. The result is the support value for the combination of $X$ and $Y$.

Note that, as the database is scanned from the first transaction until the last one, after the database is transformed into the attribute list format, all the attributes in the same attribute list are already in the sorted order. So, it is easy for us to adopt any merge sort algorithms [26] to compute the intersections.

The merge sort algorithm deployed in Eclat to calculate the intersections between attribute lists is called as linear merge sort (LMS), which works as follows. Starting from
the first attributes in both attribute lists, LMS checks whether the next attributes in both attribute lists are the same or not. If they are the same, then one of the attributes is put into the result attribute list, and then LMS moves to the next attributes in both attribute lists. If the next attributes are not the same, LMS moves to the next attribute only in the attribute list that contains the smaller attribute in the comparison. This procedure continues until CLM reaches the end of any attribute lists. LMS could be illustrated as in Figure 3-2.

```
void intersectAttributeListsInEclat(int *l1, int *l2)
{
    i = j = 0;
    while(i < |l1| && j < |l2|)
    {
        if(l1[i] == l1[j])
        {
            put l1[i] into the resulting attribute list;
            i++;
            j++;
        }
        else if(l1[i] < l1[j])
        {
            i++;
        }
        else
        {
            j++;
        }
    }
}
```

Figure 3-2 LMS pseudo-code illustration

Given two attribute lists, \(l_1\) and \(l_2\), and let \(N_1\) represent the worst-case comparison times utilizing LMS to calculate the intersection between \(l_1\) and \(l_2\), then the value of \(N_1\) equals \(2 \times (|l_1| + |l_2| - 1)\) when the attributes in both attribute lists are fully interleaved.
Note that if the two attributes being compared are not the same, another comparison is needed to decide in which attribute list should LMS move to the next attribute.

Eclat is not efficient in the sense that given $L_k$, when Eclat calculates the intersections between the attribute list $l_1$ and other attribute lists, say $l_2, l_3, ..., |L_k|$, the attributes in $l_1$ would be scanned for $|l_k|-1$ times. Even though all the attribute values in $l_1$ have already been learned from the intersection between $l_1$ and $l_2$, such information is of no help to the intersection operations of $l_1 \cap l_2$, $l_1 \cap l_3$, ..., $l_1 \cap l_{|L_k|}$, since in Eclat the knowledge of the attribute values in $l_1$ is lost each time when an intersection is finished. If the knowledge of attribute distributions in $l_1$ could be memorized once it has been learned, it will be helpful to reduce the total comparison times by just examining the attributes in other attribute lists and determining which of them belong to the set of attributes in $l_1$.

We can compute the intersection more efficiently by reducing the total comparisons a lot. We achieve the goal by the aide of a hash table. The length of the hash table depends on the maximum attribute value among all the attribute lists, i.e., the maximum transaction identification number. Initially, the value of each entry in the hash table is set to 0.

Given $L_k$, in order to calculate the intersection between the attribute lists $l_1$ and $l_2$, the attribute list $l_1$ is scanned first. During the scan, for each attribute, we use its value as the index to access the corresponding entry in the hash table and set the value of the entry to 1. Then the attribute list $l_2$ is scanned next. During the scan, we also use the values of the attributes as indexes to access the entries of the hash table. If the value of the entry being accessed equals 1, then that attribute is placed into in the resulting attribute list. If not, the attribute is discarded, and we check the next attribute in $l_2$. Obviously, if the
attribute list that contains larger number of attributes is scanned first, the total comparison times for computing the intersection between $l_1$ and $l_2$ is determined by $\min\{|l_1|, |l_2|\}$. The characteristic that the total comparison times only depends on the number of attributes in the attribute list that has less attributes is an interesting property of the basic method and will be used in the later performance analysis. Suppose the attribute lists in $L_k$ have the relations of $|l_i| \geq |l_2| \geq |l_3| \geq \ldots \geq |l_{k_e}|$, which could be easily achieved by deploying the quick sort algorithm at the beginning of the calculation. Then to calculate the intersections of $l_1 \cap l_2$, $l_1 \cap l_3$, ..., $l_1 \cap l_{k_e}$, the hash table is going to be initialized once using the attributes in $l_i$, and then the attribute lists of $l_2$, $l_3$, ..., $L_k$ are scanned consecutively. During the scan, the intersection results of $l_1 \cap l_2$, $l_1 \cap l_3$, ..., $l_1 \cap l_{k_e}$ are calculated sequentially. After all the intersections have been computed, all the entry values in the hash table are reset to 0. In the above procedure, the total number of comparison times is determined by $|l_2| + |l_3| + \ldots + |l_{k_e}|$. This procedure would be used to calculate the intersections between any attribute list $l_i$ and any other attribute list $l_j$, $1 \leq i < j \leq |L_k|$.

Figure 3-3 illustrates an example. In Figure 3-3, (a) shows an attribute list table that contains 3 lists. Assume the minsup is set to 3. (b) displays the snapshot of the hash table after the initialization. All the entries in the hash table are set to 0. The length of the hash table is set to 12, since the maximal attribute value in (a) is 12. To calculate the support value for the itemset $\{1, 2\}$, the attribute list of $\{1\}$ is scanned first, and the entry values in the hash table are set correspondingly. As $l_1$ consists of 1, 4, 5, 8, and 11, the values of the 1st, 4th, 5th, 8th, and 11th entries in the hash table are set to 1, which is shown in (c). Then the attribute list of $\{2\}$ is scanned. As first attribute in $l_2$ is 1, and the value
of the 1\textsuperscript{st} entry in the hash table is non-zero, so attribute 1 is kept in the resulting attribute list. The second attribute in \(l_2\) is 3. But the value of the 3\textsuperscript{rd} entry in the hash table is zero, so attribute 3 is discarded. The method to deal with the rest attribute values in \(l_2\) is the same. All the results except the itemset \{1, 4\}, which is not a frequent 2-itemset, are shown in (d).

<table>
<thead>
<tr>
<th>Attribute Lists</th>
</tr>
</thead>
<tbody>
<tr>
<td>(l_1)</td>
</tr>
<tr>
<td>(l_2)</td>
</tr>
<tr>
<td>(l_3)</td>
</tr>
</tbody>
</table>

**Hash Table after Initialization**

<table>
<thead>
<tr>
<th>Hash Table</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 2 3 4 5 6 7 8 9 10 11 12</td>
</tr>
<tr>
<td>0 0 0 0 0 0 0 0 0 0 0 0</td>
</tr>
</tbody>
</table>

**Scan \(l_1\): 1, 4, 5, 8, 11**

**Scan \(l_2\): 1, 3, 5, 7**

<table>
<thead>
<tr>
<th>Attribute Lists</th>
</tr>
</thead>
<tbody>
<tr>
<td>(l_1 \cap l_2)</td>
</tr>
<tr>
<td>(l_1 \cap l_3)</td>
</tr>
</tbody>
</table>

Figure 3-3 An illustration of how IT works
3.1.2 IT Algorithm

Figure 3-4 gives a formal description of IT.

**Step 1)** Scan the database D once. During the scan, establish the attribute list for each single item in I and calculate \( l_I \). Mark all the items in \( l_I \) as unvisited. Go to step 2);

**Step 2)** Establish a hash table, denoted as \( hb \), whose length equals the transaction number in the database D. Initialize every entry of \( hb \) to 0. Set \( k \) to 1. Go to step 3);

**Step 3)** If all the item-sets in \( L_k \) have been visited and \( k \) equals 1; then the calculation finishes. If all the item-sets in \( L_k \) have been visited, and \( k \) does not equal 1; then \( k \) is decreased by 1. Go to step 4);

**Step 4)** For the first unvisited itemset \( X \) in \( L_k \), scan \( X \)’s attribute list. For each attribute, denoted as \( vx \), in the attribute list, set \( hb[vx] \) to 1. Go to step 5);

**Step 5)** For any other itemset \( Y \) that comes after \( X \) in \( L_k \), scan \( Y \)’s attribute list. For each attribute, denoted as \( vy \), in the attribute list, if \( hb[vy] \) equals 0, then exam the attribute next to \( vy \) in the attribute list. If \( hb[vy] \) equals 1, put \( vy \) into the result list. If the of attributes in the final result list is no less than the \( minsup \), then put the itemset of \( X \cup Y \) and the result attribute list into \( L_{k+1} \). Mark \( X \cup Y \) as unvisited in \( L_{k+1} \). Go to step 6);

**Step 6)** Scan \( X \)’s attribute list once again. For each attribute, denoted as \( vx \), in the attribute list, set \( hb[vx] \) to 0. Mark \( X \) as visited in \( L_k \). Go to step 7);

**Step 7)** If \( L_{k+1} \) is not empty, then \( k \) is increased by 1. Go to step 3).

There are two explanations with regard to the procedure described in Figure 3-4. First, like Eclat, we also utilize the depth-first-calculation strategy in computing the sets of frequent item-sets. Second, in Step 1), we sort the item-sets in \( L_I \) in non-increasing order according to the attribute list length. By doing in this way, we will make sure that the total comparison times will be the minimum, which can be easily inferred from the
above discussion. The average time complexity for using quick sort method is
\( O(|L_1| \log |L_1|) \). To reduce the overheads resulted from the sorting, we only sort the item-sets in \( L_1 \). Our experimental results show that to sort the sets of attribute lists beyond \( L_1 \) has little help in improving the computation performance. This is because in experiments the longer attribute list in \( L_1 \) will tend to generate longer attribute lists in the subsequent sets of attribute lists.

### 3.1.3 Performance Analysis

In IT, the pseudo-code for calculating the intersection between attribute lists \( l_1 \) and \( l_2 \) is illustrated in Figure 3-5. As the information about the attributes in \( l_1 \) is already stored in the hash table \( hb \) before the procedure \texttt{intersectAttributeListsinBM()} is called, \( l_1 \) is not used in the procedure. Apparently, the code in Figure 3-5 is more succinct than that in Figure 3-2. The conditional judgment in while’s control statement in Figure 3-5 is simpler than that in Figure 3-2. In Figure 3-5, for each execution of the while loop, no matter what the current attribute value in \( l_2 \) is, there is only one comparison statement to be performed. However, for each execution of the while loop in Figure 3-2, in the worst case, two comparison statements need to be performed. Moreover, in the worst case, the while loop in Figure 3-5 only executes \(|l_2|\) times, whereas the while loop in Figure 3-2 might execute \(|l_1|+|l_2|-1\) times.

```c
void intersectAttributeListsinBM(int *hb, int *l2)
{
    j=0;
    while(j<|l2|)
    {
        m=l2[j];
        if(!hb[m])
```

put $l_2[j]$ into the resulting attribute list;

$$j++;$$

```}
```

Figure 3-5 IT pseudo-code

We take the comparison statements within the while loop both in Figure 3-5 and Figure 3-2 as the characteristic operation to analyze the computation performance. Given a set of $n$ attribute lists, let $N_1$ and $N_2$ represent the worst case of total comparison times performed by SLM and IT, respectively. Then the values of $N_1$ and $N_2$ are determined by the following equations.

$$N_1 = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} 2(|l_i| + |l_j| - 1)$$

$$= 2 \sum_{i=1}^{n} ((n - i)|l_i| + |l_{i+1}| + \cdots + |l_n| - (n - i))$$

$$= 2(n - 1)|l_1| + |l_2| + \cdots + |l_n| + (n - 2)|l_{n-1}| + \cdots + |l_n|$$

$$+ \cdots + (n - (n - 1))|l_{n-1}| + |l_n| - n(n - 1)$$

$$= 2(n - 1)(|l_1| + |l_2| + \cdots + |l_n|) - n(n - 1)$$

$$= 2\sum_{i=1}^{n} (|l_{i+1}| + \cdots + |l_n|) = |l_2| + |l_3| + \cdots + |l_n| + |l_1| + \cdots + |l_{n-1}| + |l_n| + |l_n|$$

$$= |l_1| + 2|l_2| + 3|l_3| + \cdots + (n - 2)|l_{n-1}| + (n - 1)|l_n|$$

Since each attribute list has at least one attribute, the value of $N_1$ in Equation 3-1 should always be greater than that of $N_2$ in Equation 3-2, which means the basic method always has less total comparison times than SLM. If all the attribute lists had the same number of attributes, then the value of $N_1/N_2$ would be near to 4. Theoretically, $N_1/N_2$ is upper bounded by the value of $\frac{4x|l_i|}{|l_n|}$, where $l_i$ is the longest attribute list, and $l_n$ is the shortest attribute list.
In the previous analysis, we did not take into account the overheads in IT, such as the cost for initializing the hash table and the cost for sorting $L_i$. Compared with the benefits from the reduced total comparison times, these costs are proved to be less significant in experiments.

### 3.2 Dynamic Rename Algorithm (DRA)

To calculate on Figure 3-3(c), we need to set the length of the hash table to 11, since the maximum attribute value is 11 in Figure 3-3(c), even though there are only four different attributes. Obviously, it wastes a lot of memory resources. The problem can be solved using the following method: When the first attribute list is scanned to set values in the entries of the hash table, instead of setting the entry values to 1, we use the position order of the attribute in the attribute list as the value to set the entry being accessed. In the consecutive scan of the other attribute lists, if the value of the entry that is being accessed is nonzero, then we use the entry value to substitute the attribute that is used to access the entry in the resulting attribute list. We call this method as the Dynamic Rename Algorithm (DRA). Figure 3-6 illustrates an example.

Figure 3-6 uses the same set of attribute lists that are used in Figure 3-3(a). In the first step, the position orders of the attributes in $l_i$ are used to set entry values in the hash table. For example, the position order of attribute 1 is 1, then the value of the 1\textsuperscript{st} entry is set to 1. The position order of attribute 3 is 2, then the value of the 3\textsuperscript{rd} entry in the hash table is set to 2, and so on. In the second step, to calculate the intersection of $l_i$ and $l_2$, the attributes in $l_2$ are scanned once. For the attribute 1, the value of the 1\textsuperscript{st} entry in the hash table is 1, so attribute 1 is kept in the resulting attribute list without change. For the attribute 3, the value of the 3\textsuperscript{rd} entry in the hash table is zero, so the attribute 3 is discarded. For the attribute 5, the value of the 5\textsuperscript{th} entry in the hash table is 3, so attribute
value 5 is replaced by 3 in the resulting attribute list, and so on. The calculation of the intersection between list1 and list3 is similar. Now, there are four different attributes in Figure 3-6(b). When we make calculations on Figure 3-6(b), we only need to set the length of the hash table to five.

```
<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>4</td>
<td>0</td>
<td>0</td>
<td>5</td>
<td>0</td>
</tr>
</tbody>
</table>
```

(a)

(b)

Figure 3-6 Illustration of dynamic rename algorithm

### 3.3 Optimization Methods

The performance of IT could be further improved. We discuss three ways: Reorder Frequent Itemset (RFI), Similarity Detection (SD), and Early Stop Detection (ESD).

#### 3.3.1 Reorder Frequent Itemset (RFI)

We call the intersection between two attribute lists as an intersection operation. Given an itemset $L_k$, the total number of intersection operations is determined by the cardiology of $L_k$ chooses 2, i.e., \[ \binom{|L_k|}{2} \]. Assume the total number of frequent $(k+1)$-itemsets that are generated from $L_k$ is $N$. Here we have known that these $N$ of $(k+1)$-itemsets are
calculated in \(|L_k|\) disjoint subsets, separately. Let \(IOS\) stand for the total number of intersection operations that are needed to calculate on all of the \(|L_k|\) subsets. Then the value of \(IOS\) is determined by

\[
IOS = \left( \frac{L_{k+1}^1}{2} \right) + \left( \frac{L_{k+1}^2}{2} \right) + \cdots + \left( \frac{L_{k+1}^m}{2} \right)
\]  

(3-3)

To reduce \(IOS\), we have Lemma 3-1 below.

**Lemma 3-1:** Assume \(n_1, n_2, \ldots, n_m\) are \(m\) positive integers and \(n_1 + n_2 + \ldots + n_m = M\), where \(M\) is an integer and \(M > 0\), the expression of \(\sum \left( \frac{n_i}{2} \right) + \sum \left( \frac{n_i}{2} \right) + \ldots + \left( \frac{n_m}{2} \right)\) will get the minimum value if and only if for any two integers \(n_p\) and \(n_q\), where \(1 \leq p, q \leq m\) and \(p \neq q\), \(|n_p - n_q| \leq 1\).

Proof: Part 1) if: Assume the claim is not true. Then there must exist \(m\) positive integers, \(n_1, n_2, \ldots, n_m\), through which the expression value of Equation 3-3 will be the minimum. Let \(\nu\) represent the minimal value. Obviously, the \(m\) integers should have the following properties: 1) \(n_1 + n_2 + \ldots + n_m = M\); 2) There exist at least two integers, say \(n_a\) and \(n_b\), among \(n_1, n_2, \ldots, n_m\) such that \(|n_a - n_b| > 1\). Without the loss of generality, let \(n_a > n_b\) and \(n_a - n_b = r\), where \(r < M\).

Condition 1: if \(r\) is an even number, then we could get:

\[
\left( \frac{n_b}{2} \right) + \left( \frac{n_b + r}{2} \right) = \frac{n_b \times (n_b - 1)}{2} + \frac{(n_b + r) \times (n_b + r - 1)}{2}
\]

\[
= n_b^2 + (r - 1)n_b + \frac{r(r - 1)}{2} > n_b^2 + (r - 1)n_b + \frac{r(r - 1)}{4}
\]

\[
= 2 \times \frac{1}{2} \times \left( \frac{n_b + r}{2} \right) \times \left( \frac{n_b + r}{2} - 1 \right)
\]
\[
\frac{1}{2} \left(2 \left(2 \left(2 \left(2 n_b^2 + \frac{r^2}{2} + \frac{1}{2} + 2n_b r - 2 n_b - r \right) \right) \right) \right)
\]

Condition 2: if \( r \) is an odd number, then we could get:

\[
\left(\frac{n_b}{2}\right) + \left(\frac{n_b + r}{2}\right)
= \frac{n_b \times (n_b - 1)}{2} + \frac{(n_b + r) \times (n_b + r - 1)}{2}
\]

\[
= \frac{1}{2} \left(2 \left(2 \left(2 \left(2 n_b^2 + \frac{r^2}{2} + \frac{1}{2} + 2n_b r - 2 n_b - r \right) \right) \right) \right)
\]

\[
= n_b^2 + (r - 1) n_b + \frac{r(r - 1)}{2} > n_b^2 + (r - 1) n_b + \frac{r(r - 1)}{4}
\]

\[
= \frac{1}{2} \left(\left(2 \left(2 \left(2 \left(2 n_b^2 + \frac{r^2}{2} + \frac{1}{2} + 2n_b r - 2 n_b - r \right) \right) \right) \right) \right)
\]

\[
= \frac{1}{2} \left(\left(2 \left(2 \left(2 \left(2 n_b^2 + \frac{r^2}{2} + \frac{1}{2} + 2n_b r - 2 n_b - r \right) \right) \right) \right) \right)
\]

\[
= \frac{1}{2} \left(\left(2 \left(2 \left(2 \left(2 n_b^2 + \frac{r^2}{2} + \frac{1}{2} + 2n_b r - 2 n_b - r \right) \right) \right) \right) \right)
\]

\[
= \frac{(n_b + \frac{r - 1}{2})(n_b + \frac{r - 1}{2} - 1)}{2} + \frac{(n_b + \frac{r - 1}{2})(n_b + \frac{r - 1}{2} - 1)}{2}
\]

\[
= \frac{n_b + \frac{r - 1}{2}}{2} + \left(\frac{n_b + \frac{r - 1}{2}}{2}\right).
\]

From condition 1 and condition 2, we can draw the conclusion that if \( |n_a - n_b| > 1 \), the expression of Equation 3-7 will get the smaller value \( \nu' \), here \( \nu' < \nu \), by reassigning the values to \( n_a \) and \( n_b \): 1) if \( r \) is an even number, then the new values of \( n_a \) and \( n_b \) will be the exactly one-half of the sum of their original values; 2) if \( r \) is an odd number, then we let \( n_a \) be greater than \( n_b \) by 1 and their sum of the new values equals the sum of their original values. So, the assumption is not true.

Part 2) only if: Similar to part 1). Assume the expression get the minimum value when there exist at least two integers \( n_a \) and \( n_b \) among \( n_1, n_2, ..., n_m \) such that \( 1 < |n_a - n_b| < M \).
From part 1, we know that by replacing $n_a$ with $n_a'$ and $n_b$ with $n_b'$, where $n_a' + n_b' = n_a + n_b$ and $|n_a' - n_b'| \leq 1$, we could reduce the expression value of Equation 4. So the assumption is not true, since the expression value in the assumption is not the minimum.

From part 1) and part 2), Lemma 3-1 has been proved.

Lemma 3-1 above implies that if we could distribute $N$ of $(k+1)$-itemsets into $|L_k|$ disjoint subsets as evenly as possible, then the total number of intersection operations among all the sets of $(k+1)$-itemsets will be the minimum.

For example, here are two subsets of $L_2$: $L_2^1 = \{(1,2),(1,4),(1,8),(1,12),(1,16)\}$, $L_2^2 = \{(8,9),(8,12)\}$. Using Equation 3-7, the total number of intersection operations will be $\binom{5}{2} + \binom{2}{2} = 10 + 1 = 11$. But if we move the itemsets $(1,8)$ from $L_2^1$ to $L_2^2$, then the total number of intersection operations will be reduced to $\binom{4}{2} + \binom{3}{2} = 6 + 3 = 9$.

Sometimes, it is even impossible to move the itemsets between different subsets. For example, if the combination of any two itese ts in $L_k$ turns out to be a frequent $(k+1)$-itemsets, then numbers of $(k+1)$-itemsets in different subsets would always be $|L_k|-1, |L_k|-2, \ldots, 2, 1$. Then the total intersection operations on those subsets would be constant which equals $\binom{|L_k|-1}{2} + \binom{|L_k|-2}{2} + \ldots + \binom{2}{2} = \binom{|L_k|}{3}$. On the other hand, if this situation does not happen a lot, we then have a good chance to reduce the total number of intersection operations. We do not seek to distribute set of itemsets into disjoint subsets evenly, since it is too hard in reality. Our method is simple and straightforward: before
we start to intersect the attribute lists, we first sort the itemsets in non-ascending order based on their support values. Our intuition is that the itemset whose support is less than that of other itemsets’ will have less opportunity to generate new frequent itemsets. If we let the itemset with less support make combinations with other itemsets earlier, then we might reduce the potential number of the frequent itemsets which are generated by the itemset with larger support.

3.3.2 Similarity Detection (SD)

The purpose of SD is to try to take advantage of the results of the previous computations. Assume we already knew the support value for item-sets $X$, $Y$, and $Z$. After we calculate the support value for item-sets of $X \cup Y$, $X \cup Z$, we could check to see if it is possible that itemset of $Y \cup Z$ would be a frequent one by calculating the value of the expression of $\text{sum} = \min(\text{support}(X \cup Y), \text{support}(X \cup Z)) + \min((\text{support}(X) - \text{support}(X \cup Y)), ((\text{support}(Z) - \text{support}(X \cup Z)))$. If the value of the sum is less than the $\minsup$, then the combination of $Y$ and $Z$ can not generate a frequent item-set. So it is unnecessary to compute the intersection between $Y$’s attribute list $Z$’s attribute list.

3.3.3 Early Stop Detection (ESD)

Given the $\minsup$ and two attribute lists, $l_1$ and $l_2$, it is usually not necessary to keep making comparisons between the attribute lists until at least one of the ends of the attribute lists is reached. For example, assume we reach a point during the computation, where we use $c$ to represent the total number of attributes that are common in both attribute lists among the attributes that have been compared, and use $r_1$ and $r_2$ to stand for the remaining number of attributes in the attribute list $l_1$ and $l_2$, respectively. If the sum of $c$ and $\min(r_1, r_2)$ is less than the $\minsup$, then we could stop the comparison immediately. This method is useful when the $\minsup$ is near to the average length of the
attribute lists. But when the \textit{minsup} is less than the average length of the attribute lists by a magnitude, the performance improvements of ESD is limited.

\section*{CHAPTER 4
SUPER FAST ALGORITHMS FOR DISCOVERING FREQUENT ITEM-SETS}

IT bypasses the candidate frequent item-set generation phase that is required by Apriori-like algorithms, and by using the hash table, it also has a better computation efficiency than Eclat in terms of the reduced total comparison times. Although there exist some optimization techniques that could be used to improve IT’s performance, IT still has its drawbacks. The goal of association rule mining is to find out sets of items that have strong association relations. To achieve the goal, IT makes intersection operations for every possible pair of attribute lists. For example, given $L_k = (l_1, l_2, l_3, \ldots, l_n)$, IT will make a total of $\binom{n}{2} = \frac{n(n-1)}{2}$ intersections. If the number of attributes in the resulting attribute list, generated from the intersection of two attribute lists, is less than the \textit{minsup}, the intersection operation becomes redundant. If most item-sets in $L_k$ turn out to have weak associative relations with majority of other item-sets, it can be foreseen that the large number of redundant intersection operations would badly hurt the IT’s computation performance. In this sense, IT is not efficient. If we could detect the pair of item-sets having weak associations in advance, we would avoid making intersection operations between such pair of item-sets, and thus improve the computation efficiency.
4.1 FIT\textsuperscript{1} Algorithm

In this section, we propose an algorithm FIT. FIT improves upon IT by trying to reduce the total number of intersection operations performed by IT. For the convenience of discussion, we further introduce some notations in Table 4-1.

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Remarks</th>
</tr>
</thead>
<tbody>
<tr>
<td>$CI_{L_k}^{j}$</td>
<td>The set of candidate intersection attribute lists, which consists of attribute lists that come after $l_j$ in $L_k$, here $1 \leq j &lt;</td>
</tr>
<tr>
<td>$</td>
<td>CI_{L_k}^{j}</td>
</tr>
<tr>
<td>$SG_{L_k}^{i,j}$</td>
<td>A subgroup of $L_k$, which consists of attribute lists from attribute list $i$ to attribute list $j$, here $1 \leq i \leq j &lt;</td>
</tr>
<tr>
<td>$F_{L_k}^{i,j}$</td>
<td>The set of attribute lists which come after and have strong associations with $SG_{L_k}^{i,j}$ in $L_k$. Apparently, $F_{L_k}^{i,j} \subseteq CI_{L_k}^{j}$.</td>
</tr>
<tr>
<td>$</td>
<td>F_{L_k}^{i,j}</td>
</tr>
</tbody>
</table>

4.1.1 Basic Idea

The basic idea of cutting down on the total number of intersection operations is based on the following observation:

**Observation:** Let $l$ represent the attribute list generated from the union of $n$ attribute lists, $l_1, l_2, \ldots, l_n$. If $l$ has weak association relation with the attribute $l_{n+1}$, then any attribute list $l_i$, where $1 \leq i \leq n$, has a week association relation with $l_{n+1}$.

\textsuperscript{1} FIT refers to Fast Intersecting attribute lists using hash Table.
Proof:

Assume $l$ has a weak association relation with the attribute list $l_{n+1}$ and, without loss of generality, assume the attribute list $l_1$ has a strong association relation with $l_{n+1}$. So the number of attributes that are both in $l_1$ and $l_{n+1}$, denoted as $a$, is no less than the $\text{minsup}$. As $l_1 \subseteq l$, the number of attributes that are both in $l$ and $l_{n+1}$, denoted as $b$, is greater than or equal to $a$. So $b$ is no less than the $\text{minsup}$. In other words, the attribute list $l$ has a strong association relation with $l_{n+1}$, which is contradictory to the assumption that $l$ has a weak association relation with the attribute list $l_{n+1}$. So the attribute list $l_1$ can not have a strong association relation with $l_{n+1}$. The correctness of the observation has been proved.

Based on the observation, the idea of the basic method to reduce the total intersection operations can be described as follows: given $L_k$ and any attribute list, say $l_j$, that belongs to $L_k$, before we heuristically make intersection operations between $l_j$ and every attribute list in the set $CI_{l_j}$, we first try to reduce the number of attribute lists in $CI_{l_j}$ by sifting out of $CI_{l_j}$ the attribute lists that have weak association relations with $l_j$ as many as possible.

The reduction is achieved by logically partitioning the attribute lists in $L_k$ into subgroups. Suppose each subgroup has $d$, where $1 \leq d < |L_k|$, attribute lists. For simplicity, we also assume the value of $|L_k|$ is multiple integral of $d$. Then the subgroups could be represented as $SG_{L_k}^{0,d-1}$, $SG_{L_k}^{d,d-1}$, ..., $SG_{L_k}^{|L_k|\mod d}$|L_k|. Starting from the first subgroup until the last one, for each subgroup $SG_{L_k}^{id,(i+1)d-1}$, where $0 \leq i \leq \left\lfloor \frac{|L_k|}{d} \right\rfloor - 1$, the set $F_{L_k}^{i,j}$ is calculated first.

We treat all the attribute lists in $SG_{L_k}^{id,(i+1)d-1}$ as a whole and make intersection operations between $SG_{L_k}^{id,(i+1)d-1}$ and every attribute list in $CI_{L_k}^{(i+1)d-1}$ once at a time. Initially, $F_{L_k}^{i,j}$ is set...
to be a copy of $CI_{L_k}^{(i+1)d-1}$. For each attribute list $l_g$ in $SG_{L_k}^{id,(i+1)d-1}$, where $id < (i+1)d$, we delete from the set $F_{L_k}^{i,j}$ those attribute lists that have weak association relations with $SG_{L_k}^{id,(i+1)d-1}$, which is based on the observation. After $F_{L_k}^{i,j}$ is calculated, for each attribute list $l_g$ in $SG_{L_k}^{id,(i+1)d-1}$, the basic method in Figure 3-4 is utilized to calculate the intersections between $l_g$ and any attribute lists that come after $l_g$ in $SG_{L_k}^{id,(i+1)d-1}$ and every attribute list in the $F_{L_k}^{i,j}$.

Let us look at an example. Suppose there are five attribute lists from $l_1$ to $l_5$, and $l_1$, $l_2$ and $l_3$ all have weak association relations with $l_4$ and $l_5$. Figure 4-1 (a) shows one of possible relations among them: $l_1$ and $l_2$ are complete disjoint with $l_4$ and $l_5$. $l_3$ is complete disjoint with $l_5$, too, but $l_3$ only slightly overlaps with $l_4$. Here we do not care the association relations among $l_1$, $l_2$, and $l_3$. In Figure 4-1, the outmost circle, denoted by $L$, represents the set of all the attribute lists. For this example, IT will make intersection operations for three times to draw the conclusions that $l_1 \cap l_4$, $l_2 \cap l_4$, and $l_3 \cap l_4$ are not frequent. Similarly, it will compute another three intersections to exclude $l_1 \cap l_5$, $l_2 \cap l_5$, and $l_3 \cap l_5$. If we union $l_1$, $l_2$, and $l_3$ together into a subgroup $SG_{L}^{1,3}$ as illustrated in Figure 4-1(b), and if we find out that $SG_{L}^{1,3}$ has weak association relations with $l_4$ and/or $l_5$, it will be unnecessary for us to compute the intersections between any of attribute lists of $l_1$, $l_2$, $l_3$ and $l_4$ or $l_5$. Thus we could improve the computation efficiency.
4.1.2 FIT Algorithm

Given $L_k$, the formal description of FIT algorithm is given in Figure 4-2. We assume the range of the variable $i$ in Figure 4-2 is $(1, \frac{|L_k|}{d})$.

**Step 1)** Scan the database $D$ once. During the scan, establish the attribute list for each single item in $I$ and calculate $L_i$. Sort all the item-sets in $L_i$ in non-decreasing order according to the length of the attribute lists. Mark all the items and their attribute lists in $L_i$ as unvisited. Establish a hash table, denoted as $hb$, whose length equals the total number of transactions in the database $D$. Set the value of each entry in $hb$ to 0. Set size $k$ of frequent item-sets to 1. Determine the value of size $d$ of subgroup. Go to Step 2);

**Step 2)** If all the item-sets in $L_k$ have been visited, and $k$ equals 1, then the calculation finishes. If all the item-sets in $L_k$ have been visited, and $k$ does not equal 1; then $k$ is decreased by 1 and go to Step 2). Otherwise, go to Step 4).

**Step 3)** For the first $d$ unvisited attribute lists in $L_k$, denoted as $SG^{(i-1)d,i-1}_{L_k}$, call $sift\_attribute\_list(SG^{(i-1)d,i-1}_{L_k}, C_{L_k}^{id-1})$ in Figure 4-3. Go to Step 4);
Step 4) If all the attribute lists in $SG_{L_k}^{(i-1)d, id-1}$ have been visited, then go to Step 3). For the first unvisited attribute list $X$ in $SG_{L_k}^{(i-1)d, id-1}$, use IT to calculate the intersections between $X$ and any other attribute lists that come after $X$ in $SG_{L_k}^{(i-1)d, id-1}$ and all the attribute lists in $F_{L_k}^{i, id-1}$. Mark item-set $X$ as visited. Go to Step 5);

Step 5) If $L_k+1$ is not empty, then $k$ is increased by 1 and go to Step 2). Otherwise, go to Step 4);

Figure 4-2 Pseudo-code of FIT

The function sift_attribute_list() called in the Step 3) in Figure 4-2 is shown in Figure 4-3. The purpose of the function sift_attribute_list() is to calculate all the intersections between the current subgroup $SG_{L_k}^{(i-1)d, id-1}$ and each of attribute lists in $CI_{L_k}^{(i+1)d, id-1}$.

Step 1) Scan the attribute lists in $SG_{L_k}^{(i-1)d, id-1}$ sequentially. For each attribute, denoted as $vx$, in the attribute lists, set $hh[vx]$ to 1. Go to step 2);

Step 2) Scan the attribute lists $Y$ in $CI_{L_k}^{id-1}$ sequentially. For each attribute, denoted as $ty$, in $Y$, if $hh[vy]$ equals 1, then put $vy$ into the result attribute list. After scanning $Y$, if the number of attributes in the final result list is no less than the $minsup$, then put $Y$ into $F_{L_k}^{i, id-1}$. Go to Step 3);

Step 3) Reset all the entry values in $hh$ to 0.

Figure 4-3 sift_attribute_list($SG_{L_k}^{(i-1)d, id-1}$, $CI_{L_k}^{id-1}$)

Figure 4-4 illustrates an example. Suppose the $minsup$ is 3. In Figure 4-4, (a) shows a database that consists of 8 transactions. (b) displays the corresponding attribute list format. The length of the hash table is set to 8, which equals the total number of transactions in (a). Suppose the size $d$ of subgroup is also set to 3. The procedure of the
calculation on (b) is as follows. The attribute lists of item-sets \{1\}, \{2\}, and \{3\} are scanned consecutively. The snapshot of the hash table after the scan is shown in (c). Then the attribute lists of the item-sets from \{4\} to \{8\} are scanned sequentially. For each attribute list, the attribute values are used as the indexes to access the hash table entries. If the values of the entries are not equal to 0, then the attributes are placed into the resulting attribute lists. If the number of the attributes in the resulting attribute list is no less than 3, then the index of the attribute list in (b) is put into \textit{que2}. The calculation results and the snapshot of the \textit{que2} after scanning all the attribute lists are shown in (d) and (e), respectively. Then the basic method in Figure 3-1 is deployed to calculate the support values of the item-sets \{1, 2\}, \{1, 3\}, \{1, 5\}, \{2, 3\}, \{2, 5\}, and \{3, 5\}. The final results are shown in (f).

In Figure 4-4, to calculate the frequent 2-item-sets that contain at least one of the items 1, 2 or 3, a total of 5+3+2+1=11 intersection operations are performed. If the method in Figure 3-2 is deployed, a total of 7+6+5=18 intersection operations are computed.

4.1.3 Performance Analysis

Compared with IT, FIT tries to reduce the total number of intersection operations at stake of introducing more intersection operations. If no item-sets are sifted out during the first step of FIT, then the total intersection operations will increase. For example, assume \(L_3\) consists of 9 attribute lists, and we set the size of the subgroup to 3. There will be total of \(\binom{9}{2}=36\) intersections when using IT. If it turns out that all of these 36 intersections result in the frequent item-sets,
Figure 4-4 An illustration of how FIT works.

There will be a total of $\sum_{i=1}^{3} ((9 - 3i) + \binom{9}{2}) = 9 + 36 = 45$ intersections when FIT is deployed.

To answer the question of under what conditions can FIT have less intersection operations than IT algorithm, we have Lemma 4-1.
Lemma 4-1: Given a set $L_k$ of attribute lists, the size of each subgroup is $d$, if for each subgroup $SG_{i}^{(i-1)d,id-1}$, here $1 \leq i \leq l$, the inequality of

\[
|\frac{F_{i}^{(i-1)d,id-1}}{CT_{i}^{(i-1)d}}| \leq \frac{d-1}{d}
\]

is true, then FIT algorithm will have less intersection operations than IT algorithm.

Proof:

The total intersection operations using IT algorithm on $L_k$ is:

\[
N_1 = (|L_k| - 1) + (|L_k| - 2) + \cdots + 1 = \frac{|L_k| (|L_k| - 1)}{2} \tag{4-1}
\]

\[
N_2 = \sum_{i=1}^{l} (|L_k| - id) + \sum_{i=1}^{l} ((d - 1 + F_{i}^{(i-1)d,id-1}) + (d - 2 + F_{i}^{(i-1)d,id-1}) + \cdots + F_{i}^{(i-1)d,id-1})
\]

\[
= \frac{|L_k| - d}{d} |L_k| - d \left( \frac{|L_k| - 1 + 1}{2} \right) \left( \frac{|L_k| - 1}{d} \right) + \sum_{i=1}^{l} F_{i}^{(i-1)d,id-1} d + \frac{(d - 1)d}{2} \tag{4-2}
\]

Similarly, when using FIT algorithm, the total number of intersections is determined by Equation 4-2 below.

\[
= \frac{|L_k|^2 - |L_k| d}{2d} - \frac{|L_k|^2 - |L_k| d}{2d} + \frac{(d - 1)d}{2} |L_k| + \frac{|L_k|}{2} \sum_{i=1}^{l} F_{i}^{(i-1)d,id-1}
\]

\[
\]

If FIT algorithm has less number of intersections, then Inequality 4-3 below has to be satisfied.
(4 - 1) - (4 - 2) > 0

\[ \left( \frac{|L_k|}{2} - \frac{|L_k|^2 - 2 |L_k| d + |L_k| d^2}{2d} - \frac{\sum_{i=1}^{L_k} F_{L_k}^{(i-1)d.id-1}}{d} \right) > 0 \]

\[ \frac{|L_k|^2 - |L_k| d - |L_k|^2 + 2 |L_k| d - |L_k| d^2}{2d} - \frac{\sum_{i=1}^{L_k} F_{L_k}^{(i-1)d.id-1}}{d} > 0 \quad (4-3) \]

\[ \frac{|L_k| - d |L_k| - d |L_k| - d}{2} - \frac{\sum_{i=1}^{L_k} F_{L_k}^{(i-1)d.id-1}}{d} > 0 \]

As each \( F_{L_k}^{(i-1)d.id-1} \) is upper bounded by \( |L_k| - id \), here \( 1 \leq i \leq \frac{|L_k|}{d} \), so

\[ d \sum_{i=1}^{\frac{|L_k|}{d}} F_i \leq d \sum_{i=1}^{\frac{|L_k|}{d}} (|L_k| - id) = d |L_k| \left( \frac{|L_k|}{d} - 1 \right) - d^2 \frac{(\frac{|L_k|}{d} - 1 + 1)(\frac{|L_k|}{d} - 1)}{2} \]

\[ = \frac{|L_k| - d |L_k| - d}{2} = \frac{|L_k| - d |L_k| - d}{2} = \frac{|L_k| - d |L_k| - d}{2d} \]

The value of the expression \( \frac{|L_k| - d |L_k| - d}{2d} \) is greater than the value of the expression \( \frac{|L_k| - d |L_k| - d}{2d} \) by \( \frac{|L_k| - d |L_k| - d}{2d} \). In other words, only if at least \( \frac{|L_k| - d |L_k| - d}{2d} \)

pairs of attribute lists are sifted out during the first step in FIT algorithm, can FIT algorithm have smaller number of intersections than IT algorithm. For each subgroup \( SG_{L_k}^{(i-1)d.id-1} \) here \( 1 \leq i \leq \frac{|L_k|}{d} \), the number of candidate attribute lists that might be sifted out is \( |L_k| - id \). As whenever we sift out one attribute list we will reduce \( d \) intersection operations in the second step in FIT algorithm, if we assume the subgroup \( SG_{L_k}^{(i-1)d.id-1} \) can sift out at least \( c \) percent of \( |L_k| - id \) attribute lists, then the total intersections we could
save for the second step will be \( \sum_{i=1}^{|L_k|} (d(|L_k| - id)c) \). We want this number to be no less than \( \frac{|L_k|(|L_k| - d)d}{2d} \). So we got the following:

\[
\frac{|L_k|}{\sum_{i=1}^{|L_k|} (d(|L_k| - id)c)} \geq \frac{|L_k|(|L_k| - d)}{2d}
\]

\[
\Leftrightarrow cd \frac{(|L_k| - d + d)(|L_k| - 1)}{|L_k|} \geq \frac{|L_k|(|L_k| - d)}{2d}
\]

\[
\Leftrightarrow c \geq \frac{1}{d}
\]

So Lemma 4-1 has been proved.

Let’s return to the example before Lemma 4-1. We consider the following situations. As the total number of attribute lists is 9, and the size of the subgroup is 3, if for the first subgroup, it can remove \( \frac{(9 - 3)}{3} = 2 \) attribute lists from the remaining 6 attribute lists in the set, and for the first subgroup, it can remove \( \frac{(6 - 3)}{3} = 1 \) attribute lists from the remaining 3 attribute lists in the set, then the total number of intersection operations would be

\[
(9 - 3) + \sum_{i=1}^3 (3 - i + (6 - 2)) + (6 - 3) + \sum_{i=1}^3 (3 - i + (3 - 1)) + \sum_{i=1}^3 (3 - i) = 6 + 15 + 3 + 9 + 3 = 36.
\]

If for the first subgroup, it can remove one half of the remaining 6 attribute lists, then the total number of intersection operations would be reduced to

\[
(9 - 3) + \sum_{i=1}^3 (3 - i + 3) + (6 - 3) + \sum_{i=1}^3 (3 - i + (3 - 1)) + \sum_{i=1}^3 (3 - i) = 6 + 12 + 3 + 9 + 3 = 33.
\]
Given a set $L_k$ of attribute lists, we now analyze the upper bound for the speedup of FIT algorithm over IT algorithm. We have Lemma 4-2.

**Lemma 4-2:** Given a set of attribute lists $L_k$, the upper bound for the number of total intersections performed by IT over the number of intersections performed by FIT is $\frac{\sqrt{|L_k|} + 1}{2}$.

**Proof:**

From Equation (4-1) and (4-2), we know that in order to calculate $L_k$, the total number of intersections using IT and FIT are

$$\frac{|L_k|^2 - 2|L_k|d + |L_k|d^2}{2d} + d\sum_{i=1}^{\frac{|L_k|}{d}} F_{L_k}^{(i-1)d, id-1},$$

respectively. Then the speedup is determined by

$$\text{speedup} = \frac{|L_k|(|L_k| - 1)}{2}$$

$$\frac{|L_k|^2 - 2|L_k|d + |L_k|d^2}{2d} + d\sum_{i=1}^{\frac{|L_k|}{d}} F_{L_k}^{(i-1)d, id-1}$$

Equation (4-4) is upper bounded by

$$\text{speedup} \leq \frac{|L_k|(|L_k| - 1)}{2}$$

$$\frac{d(1 - \frac{1}{|L_k|})}{1 - 2d + \frac{d^2}{|L_k|}}$$

From (4-5), we get

$$\text{speedup} \leq \frac{d(1 - \frac{1}{|L_k|})}{1 - 2d + \frac{d^2}{|L_k|}} = \frac{1 - \frac{1}{|L_k|}}{\frac{1}{d} - \frac{2}{|L_k|} + \frac{d}{|L_k|}}$$

(4-6)
Let \( f(d) = \frac{1}{d} - \frac{2}{|L_k|} + \frac{d}{|L_k|} \) \hspace{1cm} (4-7)

As the value of \(|L_k|\) is constant, the speedup in Inequality 4-5 would get the maximum value when the value of \( f(d) \) is minimum. Assume \( d \) in Equation 4-7 is a real variable whose range is \((1, |L_k|)\), then \( f(d) \) is a continuous function based on \( d \). We calculate the differential on 4-7, and we get

\[
f'(d) = -\frac{1}{d^2} + \frac{1}{|L_k|}
\]

(4-8)

We discuss (4-5) ~ (4-8) by considering the following two conditions: 1) \( 1 < d \leq \sqrt{|L_k|} \); 2) \( \sqrt{|L_k|} < d \leq |L_k| \).

When \( 1 < d \leq \sqrt{|L_k|} \), the value of \( f'(d) \) in (4-8) is positive except when \( d = \sqrt{|L_k|} \), which means the function of \( f(d) \) in (4-8) is an increasing function, so the value of speedup in (4-7) increases with the value of \( d \) increases, and the speedup gets its maximum value of

\[
\frac{1}{\sqrt{|L_k|}} - \frac{2}{|L_k|} + \frac{\sqrt{|L_k|}}{|L_k|} = \frac{|L_k| - 1}{2\sqrt{|L_k|} - 2} = \frac{(|L_k| - 1)(\sqrt{|L_k|} + 1)}{2(\sqrt{|L_k|} - 1)(\sqrt{|L_k|} + 1)} = \sqrt{|L_k|} + 1
\]

(4-9)

When \( \sqrt{|L_k|} < d \leq |L_k| \), the value of \( f'(d) \) in (4-9) is negative, which means the function of \( f(d) \) in (4-8) is a decreasing function, so the value of speedup in (4-7) decreases with \( d \) increases.
From the discussion of (1) and (2), we can arrive at the conclusions that the speedup of FIT algorithm over IT algorithm is upper bounded by \( \frac{\sqrt{|L_k|} + 1}{2} \) when the size \( d \) of the subgroup equals \( \sqrt{|L_k|} \). So Lemma 4-2 has been proved.

The upper bound in Lemma 4-2 is calculated under the assumption that for each subgroup \( SG^{(i-1)d, id-1} \) here \( i, d \geq 1, \text{and } id \leq \sqrt{|L_k|} \), the corresponding set of \( F^{(i-1)d, id-1} \) is empty. So, we call the value of \( \frac{\sqrt{|L_k|} + 1}{2} \) as an ideal speedup. In reality, it is difficult for us to get the ideal speedup. However, when there exist few frequent item-sets from the datasets, we will expect the performance of FIT will be more closer to the ideal speedup. In practice, the size \( d \) of subgroup should be a positive integer, as \( \sqrt{|L_k|} \) might not be an integer in the reality, it will be fine to set \( d \) to \( \lceil \sqrt{|L_k|} \rceil \).

There is still another question concerned with FIT algorithm: how to determine the value of size \( d \) for each subgroup? In our implementations, we use Chernoff Bounds to estimate the upper bound for the value of \( d \). Chernoff bounds provide close approximations to the probabilities in the tail ends of a Binomial distribution. That is, if \( X_i \) is a random variable with \( \Pr[X_i=1]=p \), \( \Pr[X_i=0]=1-p \), and the \( X_i \)'s are all independent. Tossing a coin is a Bernoulli trial. If \( X = \sum_{i=1}^{n} X_i \) is a sum of \( n \) independent Bernoulli trials, then \( X \) is known to have a Binomial distribution \( B(n, p) \). Chernoff Bounds have following three facts.
Given attribute lists $l_i$ and $l_j$, we want to find out the number of attributes which are in $l_j$ are also in $l_i$. For each attribute $v$ in $l_j$, we treat the search of $v$ in $l_i$ as a Bernoulli trial. Assume the total different attributes in $L_k$ is $N$. Then the possibility that the search is successful is $\frac{|l_i|}{N}$. The expected number of attributes that are both in $l_i$ and $l_j$ is $\frac{|l_j||l_i|}{N}$.

Here we assume $l_i$ and $l_j$ have a weak association. For simplicity, we use the average length, $|\bar{L}_k|$, of attribute lists in the set $L_k$ in the following discussion. Assume the size of the subgroup is $d$, for any attribute $v$ in $l_j$ that comes after the subgroup in $L_k$, the successful possibility that the search of $v$ in the subgroup is $\frac{d|\bar{L}_k|}{N}$. Then the expected number of attributes that are both in $l_j$ and in the subgroup is $\frac{d|\bar{L}_k|^2}{N}$. Here we also assume that $l_j$ has a weak association with the subgroup. Using Inequality (4-13) and the value of the $\text{minsup}$, we can simply estimate the upper bound for the value of $d$ using Inequality (4-14).

\[
(1+\varepsilon)\frac{d|\bar{L}_k|^2}{N} < \text{minsup} \tag{4-14}
\]

\[
\Leftrightarrow d < \frac{\text{minsup} \times N}{(1+\varepsilon)|\bar{L}_k|^2}
\]
Someone may argue that as the assumption that the attribute $l_j$ has weak associations with $l_i$ and/or the subgroup is not true, we cannot use Chernoff Bounds together with the above inequalities to estimate the value of $d$. But remember that FIT algorithm consists of two steps. The only purpose of the first step is to sift out the pairs of item-sets that have weak associations as many as possible. The second step is just the step that truly calculates the intersections between the attribute lists. The assumption that $l_j$ has a weak association with the subgroup in the first step does not affect the correctness of the results at the end of the second step. On the other hand, if the pair of attribute lists, which we predicted wrong, turns out to be strong associative, they will not be deleted and will still be kept in the set of attribute lists in the first step and will be dealt with in the second step anyway. Due to the same reason, when we want the value of $d$ to be large, we could set $\varepsilon$ in Inequality (4-14) to be small, say 0.2.

Here is an example. Assume there are 1000 attribute lists in $L_2$, $|L_2|=2000$, the maximum attribute value in $L_2$ is 100000, if the $\minsup$ is 1,500, the upper bound estimation for the value of $d$ could be calculated with $\varepsilon = 0.2$ using Inequality 3-4:

$$d < \frac{1500 \times 100000}{(1 + 0.2) \times 2000^2} = \frac{15 \times 10^7}{4.8 \times 10^6} = 31.25.$$ So $d$ could take the value as much as 31. According to the second fact of Chernoff Bounds, the probability that if an attribute list has weak association with any of $d$ attribute lists but has strong association with the union of $d$ attribute lists is no greater than $\ell^{-\frac{31 \times 2000^2}{100000}} \leq 7 \times 10^{-8}$. 

4.1.4 Implementation Issues

In our implementation of FIT, we further reduce the number of intersection operations when the size $d$ of the subgroup is small, say $3 \leq d \leq 8$, by deploying the following method. We set aside two more arrays, denoted as $counter$ and $reference$. The length of the array $counter$ is equal to the value of $d$. The length of the array $reference$ equals $2^d + 1$. The initial value of each entry in both arrays is 0. Before we initialize the hash table using the attribute lists in the subgroup $SG^{i,j}_{L_k}$, we ordered the attribute lists in $SG^{i,j}_{L_k}$ from 0 to $d-1$. For each attribute in the $m$-th attribute list of $SG^{i,j}_{L_k}$ here $0 \leq m \leq d-1$, rather than set the value of the corresponding entry in the hash table to 1, we increase the value of the entry by $2^m$. For each intersection operation between the subgroup $SG^{i,j}_{L_k}$ and any other attribute list, when we scan the attribute list $l_n$ in $CI^{i}_{L_k}$, for each attribute in $l_n$, if the value of the corresponding entry is $y$, and $y$ does not equal -1, then the value of the $y$-th entry in the array $reference$ is increased by 1. At the end of the first step, we calculate the value of each entry in the array $counter$ with the aide of the array $reference$. The value of the $i$-th entry, where $0 \leq i \leq d-1$, in the array $counter$ is calculated by accumulating the value of any entry whose index’s $i$-th bit is 1 in the array $reference$. The value in the $i$-th entry in the array $counter$ indicates the number of attributes in the intersection result generated from the intersection between $l_j$ and $l_i$. In the second step, before calculating the intersection between $l_i$ and any attribute list $l_j$ in $F^{i,j}_{L_k}$ again, we check the array $counter$ first. Only when the value of the $i$-th entry in the array $counter$ is greater than the $minsup$ do we need to calculate the intersection between $l_i$ and $l_j$.

$|L| = 3$, $minsup = 3$, $d = 3$
**Figure 4-5** An illustration of the array *reference* and *counter*

Figure 4-5 shows an example. Assume the attribute list table in (a) consists of four attribute lists. The *minup* is set to 3. The size $d$ of subgroup is equal to 3. The value of each entry in the hash table is 0 initially, which is shown in (b). Then the first three attribute lists in the attribute list table are scanned sequentially. During the scan of $l_1$, for every attribute, we use its value to access the entry in the hash table, and increase the corresponding entry value by $2^0$, since $l_1$ is ordered as 0 in the first group. Similarly,
during the scan the second attribute list $l_2$, when we access the entry in the hash table, we increase the corresponding entry value by $2^1$, since $l_2$ is ordered as 1 in the first group. The calculation on the attribute list $l_3$ is the same, except that we increase the entry values in the hash table by $2^2$. The snapshot of the hash table after scanning the first three attribute lists is shown in (c). Then the attribute list $l_4$ is scanned. For the first attribute 2, as the second entry value in the hash table is 0, so need do nothing. For the second attribute 4, as the fourth entry value in the hash table is 2, so the value of the entry whose index is 2 in the array $reference$ is increased by 1. The calculations on the other attributes in $l_4$ are the same. The final result of the array $reference$ is shown in (d). We use the result in (d) to calculate the entry values in the array $counter$. The method is as follows: we go through the array $reference$. For any entry whose index is $p$, if the value of the entry does not equal 0, we examined the binary representation of $p$. If the $i$-th bit of the binary representation of $p$ is equal to 1, then the value of the $i$-th entry in the array $counter$ is increased by 1. The result of the entry values in the array $counter$ is shown in (e). As the number of attributes in the intersection result between the union of the first three attribute lists and the fourth attribute lists is 3, we need to check the intersections between any of the first three attribute lists and the fourth attribute list. But before we do that, we examine the array $counter$ first. As the first and second entry values are both less than the $minsup$, 3, the intersection between the first or the second attribute list and the fourth attribute list can not generate the frequent item-sets. So it is unnecessary for us to make the intersection operations. On the other hand, the third entry value in the array $counter$ is 3 which is no less than the $minsup$. We need to make the intersection operations between the third attribute list and the fourth attribute list in (a).
After transforming the original database into the attribute list format, we keep the frequent single items in $L_1$ in non-decreasing order according to their support values, i.e., the length of the attribute lists. We believe that the item-sets whose support value is larger is more likely to combine with other frequent item-sets to generate new frequent item-sets. So, it is probable that the number of attribute lists in different $F^{(i-1)d,d-1}_{L_k}$ is quite different with each other with the number of attribute lists in the beginning $F^{(i-1)d,d-1}_{L_k}$ very large, and the numbers of attribute lists in the subsequent $F^{(i-1)d,d-1}_{L_k}$ reduce gradually. In our implementations, we tried to dynamically determine the size $d$ of each subgroup. That is, we set the value of $d$ to a relative small value at the beginning of the calculation and increased it gradually. But the experiment results showed that there were no major performance improvements by doing in this way.

### 4.2 SFIT Algorithm

FIT only considers dividing the set $L_k$ into subgroups for only once. As a matter of fact, if the number of attribute lists in each subgroup is still large, after calculating the set of $F^{i,j}_{L_k}$ and before deploying the basic method to calculate the intersections, we could treat each subgroup as a new set of frequent item-sets and apply the dividing method on the subgroup. This procedure could be used recursively until the size of the subgroup is relative small, say 3–8. If a subgroup $SG$ is divided into a set of smaller subgroups. For each such smaller subgroup $SG'$, $SG$ is said to be the parent subgroup of $SG'$. The set $L_k$ is thought as the subgroup at 0-th recursion level. When the set of frequent item-sets are divided recursively, one thing should be paid attention to. Suppose we treat the subgroup $SG^{id,(i+1)d-1}_{L_k}$ as a new set $L_k'$ of frequent $k$-item-sets. When $L_k'$ is divided into smaller
subgroups whose size is $d'$, for each such subgroup $SG^{'d,(i+1)d^{-1}}_{L_k}$, where $0 \leq i \leq \frac{|L_k|}{d} - 1$, the initial set of $F^{i,j}_{L_k}$ is the union of the set of $CI^{(i+1)d^{-1}}_{L_k}$ and the set of $F^{i,j}_{L_k}$. For any subsequent divisions, the set of $F^{i,j}_{L_k}$ is replaced by the set of $F^{i,j}_{L_k}$ that is calculated in the subgroup that is the parent of the current subgroup.

### 4.2.1 SFIT Algorithm

The formal description of SFIT is illustrated in Figure 4-6 below.

**Step 1)** Scan the database $D$ once and establish the attribute list for each single item in $I$. At the same time, calculate $L_1$. Go to Step 2);

**Step 2)** Sort the items in $L_1$ in non-increasing order according to the support values that equal to the length of the attribute lists. Mark all the items and their attribute lists in $L_1$ as unvisited. Go to Step 3);

**Step 3)** Establish a hash table, denoted as $hb$, whose length equals the total number of transactions in the database $D$. Set the value of each entry in $hb$ to -1. Go to Step 4);

**Step 4)** Assume the recursion depth is $t$, set the size array $d[1..t]$ and determine the value of each entry in $d$. The value in the $n$-th entry in $d$ determines the size of the subgroup at the depth $n$. Let $k = p = 1$ and $G = F = \emptyset$, where $k$ represents the size of the current frequent item-sets, $p$ stands for the depth of the current subgroup, $G$ represents the set of attribute lists in the current subgroup, and $F$ stands for the set of attribute lists that have strong association relation with the union of the attribute lists in $G$. Go to Step 5);

**Step 5)** If all the item-sets in $L_1$ have been visited, then the program stops. Otherwise, group the next $d[p]$ unvisited item-sets in $L_1$ into $G$, and group the item-sets that come after the item-sets in $G$ into $F$. Call the function calculate_subgroup($p$, $G$, $F$); Mark the item-sets in $G$ as visited. Repeat Step 5);

(a) Frequent_item-set_calculation()

**Step 1)** Scan the attribute lists in $G$ sequentially. For each attribute met, use the value of
that attribute as the index to access the entry of \( hb \) and set the value of the corresponding entry to 1. Go to Step 2);

**Step 2)** Scan the attribute list in \( F \) separately. For each attribute list, during the scan, use the attribute met in the scan to access the entry in \( hb \). If the value of the entry being accessed is 1, keep the attribute in the result attribute list. Otherwise, discard the attribute. After the scan, if the number of attributes in the result list is less than the \( \text{minsup} \), remove that attribute list and the corresponding item-set from \( F \). Go to Step 3);

**Step 3)** Set the value in each entry in \( hb \) to \(-1\). Go to Step 4);

**Step 4)** If the value of the size of the current subgroup is the last in the size array \( d \), go to step 6). Otherwise, go to Step 5);

**Step 5)** If all the item-sets in \( G \) have been visited, then return. Otherwise denote the first \( d[p+1] \) unvisited item-sets in \( G \) as \( G' \), denote the set of item-sets that come \( G' \) in \( G \) as \( F' \), call calculate_subgroup(\( p+1, G', F+F' \)); Go to Step 5);

**Step 6)** If all the item-sets in \( G \) have been visited, then return. Otherwise for the first unvisited item-set in \( G \), scan its attribute list \( l_i \) sequentially. For each attribute met, set the value of the corresponding entry in \( hb \) to 1. Then scan the attribute lists that come after \( l_i \) in \( G \) and all the attribute lists in \( F \) sequentially. For each such attribute list \( l_j \), if the number of the attribute lists in the result attribute list is no less than the \( \text{minsup} \), then put the result attribute list and the corresponding new item-set into the set \( L_{k+1} \). After scan all the attribute lists in the rest of \( G \) and all the attribute lists in \( F \), if \( R \) is not empty, calculate the frequent item-sets on \( L_{k+1} \) using the same procedure. Mark the first unvisited item-set in \( G \) as visited. Go to Step 6);

(b) calculate_subgroup(\( p, G, F \))

Figure 4-6 SFIT algorithm

### 4.2.2 Performance Analysis

When we recursively divide the set of attribute lists into sets of smaller subsets, all of the subsets form a tree. Each subset constitutes a node in the tree. Each node in the tree
could be pinpointed by two parameters: depth and offset. The depth of a node is defined as the depth of its parent plus one. The depth of the root node is 0. All the nodes with the same depth are numbered from left to right starting at 0. This number is called as the offset of the node. Given \( L_k \), the depth and offset parameters of a node, we can locate the node that consists of a set of attribute lists in \( L_k \) accurately.

For simplicity, in Table 4-2, we listed some symbol notations that are extended from Table 4-1.

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( d_n )</td>
<td>The size of the subgroup at the depth ( n ).</td>
</tr>
<tr>
<td>( SG_{L_k}^{(n,m)} )</td>
<td>A subgroup whose depth parameter is ( n ), and whose horizontal parameter is ( m ). The subgroup ( SG_{L_k}^{(n,m)} ) consists of the attribute lists from the ((d_n \times m))-th attribute list to the ((d_n \times (m+1)-1))-th attribute list in ( L_k ). If ( n ) equals 0, then ( m ) must also be 0. ( SG_{L_k}^{(0,0)} ) is ( L_k ).</td>
</tr>
<tr>
<td>( SG_{L_k}^{(n,r)} )</td>
<td>The set of all the subgroups whose depth parameters are ( n ). ( SG_{L_k}^{(0,r)} ) is ( L_k ).</td>
</tr>
<tr>
<td>( SG_{L_k}^{-(i,m)} )</td>
<td>The ( i )-th attribute list in the subgroup ( SG_{L_k}^{(n,m)} ).</td>
</tr>
<tr>
<td>( CI_{L_k}^{-(n,m)} )</td>
<td>The set of candidate intersection item-sets that come after the ( j )-th attribute list in the subgroup of ( SG_{L_k}^{(n,m)} ), here ( 1 \leq j \leq</td>
</tr>
<tr>
<td>(</td>
<td>CI_{L_k}^{-(n,m)}</td>
</tr>
<tr>
<td>( F_{L_k}^{(n,m)} )</td>
<td>The set of attribute lists which come after and have strong associations with ( SG_{L_k}^{(n,m)} ) in ( L_k ). Apparently, ( F_{L_k}^{(n,m)} \subseteq CI_{L_k}^{-(n,m)} ).</td>
</tr>
<tr>
<td>(</td>
<td>F_{L_k}^{(n,m)}</td>
</tr>
<tr>
<td>(</td>
<td>F_{L_k}^{(n,r)}</td>
</tr>
</tbody>
</table>

Theoretically, SFIT does not guarantee that it can lessen the total intersection operations by FIT, which can be illustrated from the following situations. Suppose the set \( L_2 \) consists of 1200 attribute lists, \( d_1 \), the size of the subgroup at the depth 1, is set to
30, and \( d_2 \), the size of the subgroup at the depth 2 is set to 3. Also assume the value of \( F_2^{(1, m)} \) is 50 percent of (1200-\(m\times30\)), and the value of \( F_2^{(2, m')} \) is 40 percent of \( (F_2^{(1, m')} + 30 - m'\times3) \), here \( 1\leq m\leq40 \) and \( 1\leq m'\leq10 \). Then the total number of intersection operations using IT, FIT, and SFIT are determined by

\[
\binom{1200}{2} = \frac{1200 \times 1199}{2} = 719400 ,
\]

\[
\sum_{i=1}^{40} (1200 - i \times 30 + \sum_{j=1}^{30} (30 + (1200 - i \times 30) \times 0.5 - j)) = 391800 , \text{ and}
\]

\[
\sum_{i=1}^{40} (1200 - i \times 30 + \sum_{j=1}^{10} (30 + (1200 - i \times 30) \times 0.5 - j \times 3 + \sum_{k=1}^{3} (3 + (30 + (1200 - i \times 30) \times 0.5 - j \times 3) \times 0.4 - k))) = 293880 .
\]

, respectively. So under the above assumption, SFIT has less total number of intersections than FIT. If we further assume the value of \( F_2^{(1, m)} \) is 10 percent of (1200-\(m\times30\)), and the value of \( F_2^{(2, m')} \) is 20 percent of \( (F_2^{(1, m')} + 30 - m'\times3) \), then the total number of intersections utilizing SFIT will be further reduced to the number of

\[
\sum_{i=1}^{40} (1200 - i \times 30 + \sum_{j=1}^{10} (30 + (1200 - i \times 30) \times 0.1 - j \times 3 + \sum_{k=1}^{3} (3 + (30 + (1200 - i \times 30) \times 0.1 - j \times 3) \times 0.2 - k))) = 70800 .
\]

However, on the other hand, if we keep the value of \( F_2^{(1, m)} \) to be 10 percent of (1200-\(m\times30\)), and change the value of \( F_2^{(2, m')} \) to be 70 percent of \( (F_2^{(1, m')} + 30 - m'\times3) \), then the total number of intersection operations utilizing SFIT will be increased to the number of

\[
\sum_{i=1}^{40} (1200 - i \times 30 + \sum_{j=1}^{10} (30 + (1200 - i \times 30) \times 0.5 - j \times 3 + \sum_{k=1}^{3} (3 + (30 + (1200 - i \times 30) \times 0.5 - j \times 3) \times 0.8 - k))) = 440760 ,
\]

which is larger than the value of 391800. So SFIT performs more intersection operations than FIT does.

Lemma 4-3 gives a sufficient condition under which the number of intersections performed by SFIT is guaranteed to be less than that performed by FIT. The sufficient
conditions also tell us when we should stop the recursively divide the subgroup into sets of smaller subsets.

**Lemma 4-3:** Given a set $L_k$ of attribute lists, the depth of the current subgroup is $n-1$, and the size of each subgroup is $d_{n-1}$, if there exists a value of $d_n$ such that when the subgroup $S\chi_{L_k}^{(n-1,m)}$ is further divided into smaller subgroups with size of $d_n$, for each subgroup $S\chi_{L_k}^{(n,i)}$ here $d_n \times m \leq i \leq d_n \times (m+1)-1$, the inequality of 

$$\frac{|F_{L_k}^{(n,m)}|}{|CI_{L_k}^{((d_n \times m+1)-1)-(n,m)}|} \leq \frac{d_n - 1}{d_n}$$

is satisfied, then further divide the subgroup with size $d_n$ into the smaller subgroup with size of $d_{n+1}$ will reduce the total intersection operations.

**Proof:**

If the recursion stops at the subgroups whose depths are $n-1$, then the total number of intersection operations would be determined by (1) below.

\[
\begin{align*}
&d_n \sum_{i_1=1}^{d_0} (d_0 - i_1 d_1) + \sum_{i_2=1}^{d_1} (d_1 + F_{L_k}^{(0,i_2)}) - i_2 d_2 + \sum_{i_3=1}^{d_2} (\cdots + \sum_{i_{n-2}=1}^{d_{n-2}} (d_{n-2} + F_{L_k}^{(n-3,i_{n-2})}) - i_{n-2} d_{n-1}) \\
&+ \sum_{i_{n-1}=1}^{d_{n-1}} (d_{n-2} + F_{L_k}^{(n-3,i_{n-2})} - i_{n-1} d_{n-1}) + (d_{n-1} + F_{L_k}^{(n-2,i_{n-1})} - 1) + (d_{n-1} + F_{L_k}^{(n-2,i_{n-1})} - 2) + \cdots + F_{L_k}^{(n-2,i_{n-1})})
\end{align*}
\]

Similarly, if the recursion stops at the subgroups whose depths are $n$, then the total number of intersection operations would be determined by (2) below.

\[
\begin{align*}
&d_n \sum_{i_1=1}^{d_0} (d_0 - i_1 d_1) + \sum_{i_2=1}^{d_1} (d_1 + F_{L_k}^{(0,i_2)}) - i_2 d_2 + \sum_{i_3=1}^{d_2} (\cdots + \sum_{i_{n-2}=1}^{d_{n-2}} (d_{n-2} + F_{L_k}^{(n-3,i_{n-2})}) - i_{n-2} d_{n-1}) \\
&+ \sum_{i_{n-1}=1}^{d_{n-1}} (d_{n-2} + F_{L_k}^{(n-3,i_{n-2})} - i_{n-1} d_{n-1}) \\
&+ \sum_{i_{n-1}=1}^{d_{n-1}} (d_{n-1} + F_{L_k}^{(n-2,i_{n-1})} - i_{n} d_{n}) + (d_{n} + F_{L_k}^{(n-1,i_{n})} - 1) + (d_{n} + F_{L_k}^{(n-1,i_{n})} - 2) + \cdots + F_{L_k}^{(n-1,i_{n})})
\end{align*}
\]
If we want to reduce the total number of intersection operations by dividing the subgroups at the depth of \( n-1 \), then the result in (1) should be no less than (2). Thus,

\[
(1)-(2):
\]

\[
\sum_{i_1=1}^{d_{n-2}} \sum_{i_2=1}^{d_{n-2}} \cdots \sum_{i_{n-1}=1}^{d_{n-2}} \left( (d_{n-1} + F_{L_k}^{(n-2, i_{n-1})} - 1) + (d_{n-1} + F_{L_k}^{(n-2, i_{n-1})} - 2) + \cdots + F_{L_k}^{(n-2, i_{n-1})} \right)
\]

\[
- \sum_{i_{n-1}=1}^{d_{n-1}} \sum_{i_n=1}^{d_n} (d_{n-1} + F_{L_k}^{(n-2, i_{n-1})} - i_n d_n + (d_n + F_{L_k}^{(n-1, i_n)} - 1) + (d_n + F_{L_k}^{(n-1, i_n)} - 2) + \cdots + F_{L_k}^{(n-1, i_n)}) \cdots ))
\]

\[
\L_n \left[ \frac{d_1}{d_2} \frac{d_2}{d_3} \cdots \frac{d_{n-2}}{d_{n-1}} \left( \frac{(d_{n-1} - 1) \times d_{n-1}}{2} + \frac{d_{n-2}}{d_{n-1}} \sum_{i_{n-1}=1}^{d_{n-1}} F_{L_k}^{(n-2, i_{n-1})} - d_{n-1} \times \frac{d_{n-2}}{d_n} \sum_{i_n=1}^{d_n} F_{L_k}^{(n-1, i_n)} - \frac{(d_n - 1) \times d_n}{2} \times \frac{d_{n-2}}{d_{n-1}} \sum_{i_{n-1}=1}^{d_{n-1}} F_{L_k}^{(n-2, i_{n-1})} \right)
\]

\[
+ d_n \times \frac{1 + \frac{d_n - 1}{2}}{d_n} \times \frac{d_{n-2}}{d_{n-1}} - d_n \times \frac{d_{n-2}}{d_n} \sum_{i_n=1}^{d_n} F_{L_k}^{(n-1, i_n)} - \frac{(d_n - 1) \times d_n}{2} \times \frac{d_{n-2}}{d_{n-1}} \sum_{i_{n-1}=1}^{d_{n-1}} F_{L_k}^{(n-2, i_{n-1})}
\]

\[
\L_n \left[ \frac{d_1}{d_2} \frac{d_2}{d_3} \cdots \frac{d_{n-2}}{d_{n-1}} \left( \frac{(d_{n-1} - 1) \times d_{n-1}}{2} + \frac{d_{n-2}}{d_{n-1}} \sum_{i_{n-1}=1}^{d_{n-1}} F_{L_k}^{(n-2, i_{n-1})} - d_{n-1} \times \frac{d_{n-2}}{d_n} \sum_{i_n=1}^{d_n} F_{L_k}^{(n-1, i_n)} - \frac{(d_n - 1) \times d_n}{2} \times \frac{d_{n-2}}{d_{n-1}} \sum_{i_{n-1}=1}^{d_{n-1}} F_{L_k}^{(n-2, i_{n-1})} \right)
\]

\[
+ \frac{(d_n + d_n - 1) \times d_{n-2}}{2 d_n} - \frac{d_{n-2}}{d_n} \times \frac{d_{n-1}}{d_{n-1}} \sum_{i_n=1}^{d_n} F_{L_k}^{(n-1, i_n)} - \frac{(d_n - 1) \times d_{n-2}}{2}
\]

\[
\L_n \left[ \frac{d_1}{d_2} \frac{d_2}{d_3} \cdots \frac{d_{n-2}}{d_{n-1}} \left( \frac{(d_{n-2} - (d_{n-1} - 1) - d_{n-2} - (d_n - 1))}{2} + \frac{d_{n-2}}{d_{n-1}} \sum_{i_{n-1}=1}^{d_{n-1}} F_{L_k}^{(n-2, i_{n-1})} - \frac{d_{n-2} - d_n}{d_n} \sum_{i_n=1}^{d_n} F_{L_k}^{(n-1, i_n)} - \frac{d_{n-2} - d_n}{d_n} \sum_{i_{n-1}=1}^{d_{n-1}} F_{L_k}^{(n-2, i_{n-1})} \right)
\]

\[
\L_n \left[ \frac{d_1}{d_2} \frac{d_2}{d_3} \cdots \frac{d_{n-2}}{d_{n-1}} \left( \frac{(d_{n-2} - (d_{n-1} - 1) - d_{n-2} - (d_n - 1))}{2} + \frac{d_{n-2}}{d_{n-1}} \sum_{i_{n-1}=1}^{d_{n-1}} F_{L_k}^{(n-2, i_{n-1})} - \frac{d_{n-2} - d_n}{d_n} \sum_{i_n=1}^{d_n} F_{L_k}^{(n-1, i_n)} - \frac{d_{n-2} - d_n}{d_n} \sum_{i_{n-1}=1}^{d_{n-1}} F_{L_k}^{(n-2, i_{n-1})} \right)
\]

\[
\L_n \left[ \frac{d_1}{d_2} \frac{d_2}{d_3} \cdots \frac{d_{n-2}}{d_{n-1}} \left( \frac{(d_{n-2} - (d_{n-1} - 1) - d_{n-2} - (d_n - 1))}{2} + \frac{d_{n-2}}{d_{n-1}} \sum_{i_{n-1}=1}^{d_{n-1}} F_{L_k}^{(n-2, i_{n-1})} - \frac{d_{n-2} - d_n}{d_n} \sum_{i_n=1}^{d_n} F_{L_k}^{(n-1, i_n)} - \frac{d_{n-2} - d_n}{d_n} \sum_{i_{n-1}=1}^{d_{n-1}} F_{L_k}^{(n-2, i_{n-1})} \right)
\]
\[
\frac{1}{d_{n-2}} \left( \frac{d_{n-2} - d_{n-1}}{d_n} - (d_n - 1) + d_{n-1} \left[ \frac{F_{L_{k_i}}^{(n-2, \ast)}}{F_{L_{k_i}}^{(n-1, \ast)}} \right] \right)
\]
\[
\left( \frac{d_{n-2} - d_{n-1}}{d_n} - (d_n - 1) + d_{n-1} \left[ \frac{F_{L_{k_i}}^{(n-2, \ast)}}{F_{L_{k_i}}^{(n-1, \ast)}} \right] \right)
\]
\[
\left( \frac{d_{n-1}}{d_n} - 1 \right) \left( \frac{d_{n-1}}{d_n} - 1 \right) + d_{n-2} \left[ \frac{F_{L_{k_i}}^{(n-2, \ast)}}{F_{L_{k_i}}^{(n-1, \ast)}} \right] \left( \frac{1}{d_n} - \frac{1}{d_{n-1}} \right)
\]

As \( \frac{d_{n-1}}{d_n} - 1 > 0, \ d_n - 1 > 0 \), so as long as \( \frac{F_{L_{k_i}}^{(n-2, \ast)}}{F_{L_{k_i}}^{(n-1, \ast)}} \left( \frac{1}{d_n} - \frac{1}{d_{n-1}} \right) > 0 \), the total number of intersection operations will be reduced by further dividing subgroups into the sets of smaller groups. So Lemma 4-3 has been proved.

4.2.3 Implementation Issues

In practice, it does not help to make many recursive divisions on the subgroups. Our experiments show that to make the calculations on \( L_1 \), two or three recursive divisions will be enough. If two recursive divisions are adopted, the sizes of subgroups at the depth 1 will be set to 15 or 30. The sizes of the subgroups at the depth 2 will be set to 3. If three levels of recursive divisions are used, then the sizes of subgroups from depth 1 to depth 3 will be set to 120, 15, and 3. To deal with any other \( L_k, k > 1 \), one division is enough, and the size of the subgroups are usually set to 3 in our experiments.
In this chapter, we focus on how to calculate the sets of frequent item-sets in parallel. As said in Chapter 2, there have been several parallel algorithms published based on Apriori. The parallel algorithms put forward here are based on IT discussed in Chapter 3.

We designed two types of parallel algorithms: one is applied for the computer system that is of the so-called shared-memory multiprocessor (SMP) architecture. In the SMP architecture, there are multiple processors with a shared main memory. The communications between the different processors are achieved through the access of the common memory cells. The other one is used for the distributed computer system that consists of multiple independent computers, each of which has the private main memory and the private hard disk. The communications between the different processors are achieved using the message passing method through the inter-computer connections. For the convenience of the discussion, we assume all the computers in the distributed computer system have the same configurations.

For any computer system discussed, we use $B$ to stand for the number of blocks that can be transferred from the disk into the main memory through a single I/O operation. The size of each block is represented by the symbol $S_b$. The size is measured in byte.

5.1 Basic Idea

The basic idea of the parallel algorithms proposed here consists of four steps. In the first step, the database $D$ is divided into sub-datasets. Then each sub-dataset is calculated separately. In the second step, the results from all of the sub-datasets are combined and
the support values of the same item-set in the different sub-dataset are accumulated. In
the third step, some sub-datasets might need to be re-calculated to find out the support
values for some item-sets whose support information have been discarded at the end of
the first step. In the fourth step, all the results generated at the end of the second step and
the third step are combined, and the final sets of frequent item-sets of the database D will
be generated.

According to the support values of the item-sets generated at the end of the second
step, the item-sets could be logically divided into four categories. 1) The item-sets,
which are frequent in the original database D, are also local frequent in all the sub-
datasets. 2) The item-sets, which are frequent in the original database D, are local
frequent in part of the set of sub-datasets. 3) The item-sets, which may or may not be
frequent in the original database D, are local frequent in some of the set of sub-datasets.
4) The item-sets, which are not frequent item-sets in the original database D, are local
frequent item-sets in some or none of the set of sub-datasets.

For the item-sets in the first category, as they are local frequent item-sets in all the
sub-datasets, their support values in all the sub-datasets have been calculated in the first
step. So the accumulative support values at the end of the second step are their total
support values in the original database. We do not need to make any further calculations
for this kind of item-sets.

For the item-sets in the second category, although their accumulative support values at
the end of the second step exceed the minsup, we still need to calculate their support
values on those sub-datasets that failed to provide the support information at the second
step in order to calculate the confidence of the association rules.
For the item-sets in the third category, we need to consider the number \( n \) of sub-data-sets that calculate the support values for item-set \( X \) and the sum \( s \) of the \( n \) support values. If the value of the expression \( s + (N-n) \) is no less than the \( \text{minsup} \), the item-set \( X \) might be a frequent one in the original database. We then treat \( X \) in the same way as the item-set in the second category and calculate its support values in those sub-datasets where the support value of \( X \) is not provided. If the value of the expression \( s + (N-n) \) is less than the \( \text{minsup} \), then \( X \) can not be frequent in the original database. We will discard it and will not consider it further.

For the item-sets in the fourth category, we just omit them from the further computations.

We introduce some new notations here: We use \( P \) to represent the number of the processors or the number of computers in the computer system. All the processors are supposed to be ordered from 1 to \( P \). If the SMP architecture is deployed, we use \( M \) to stand for the size of the shared memory. If the distributed computer system is adopted, \( M \) will stand for the size of the autonomous memory in the system. All the memories are measured in byte. The term \textit{global support value of the item-set } \( X \) \textit{represents the support value for the item-set } \( X \) \textit{in the original database } \( D \). The term \textit{local support value of the item-set } \( X \) \textit{stands for the support value for the item-set } \( X \) \textit{in the sub-dataset}. If the user-specified minimum support is \( \text{minsup} \), and the database \( D \) is divided into \( N \) sub-datasets, then the value of \( \text{minsup}/N \) is called as the \textit{local minsup}. The term \textit{global frequent item-set} refers to the item-sets whose global support value is no less than the \( \text{minsup} \) in the database \( D \). The term \textit{local frequent item-set} refers to the item-set whose support is no less than the \textit{local minsup} in the sub-dataset. We call an item-set that is a local frequent
item-set but not a global frequent item-set as a \textit{false local frequent item-set}. The appearance of the false local frequent item-sets is due to the transactions that contain the item-set are not distributed into all the sub-datasets equally.

5.2. Database Division

To divide the database D into sub-datasets, there exist many different methods. Different dividing methods will affect the design and the efficiency of the parallel algorithms. We discuss two dividing methods here.

5.2.1 Horizontal Division

The horizontal division is to divide the database D into sub-datasets along its transactions. If we want to divide the database D into \( N \) sub-datasets, we could group the first \( \frac{|D|}{N} \) transactions into the first sub-dataset, the second \( \frac{|D|}{N} \) transactions into the second sub-dataset, \( \ldots \), the \((|D|-1)\)-th \( \frac{|D|}{N} \) transactions into the \((|D|-1)\)-th sub-dataset, and the rest transactions into the last sub-dataset.

Obviously, simplicity is the advantage of the horizontal division method. The division of the database D could be achieved by scanning D once. We only need to keep one buffer of size \( M \) for the purpose of both input and output. There will be a total of \( \frac{|D|}{B \times S_b} \) I/O operations for read and \( \frac{|D|}{B \times S_b} \) I/O operations for write.

The disadvantage of the horizontal division method is that if the data in the database D is not evenly distributed, there will be two bad consequences: 1) Some sub-datasets may contain many false local frequent item-sets, which will waste the CPU time. 2) Some global frequent item-sets may not be local frequent item-sets in some sub-datasets. In
that case, the sub-dataset will omit the corresponding calculation results, which are not
correct and will be punished by re-calculating the support values again.

5.2.2 Randomized Horizontal Division

Like the horizontal division, the randomized horizontal division method also divides
the database D into sub-datasets based on the units of the transactions. The difference is
that the randomized horizontal division method will randomly decide which transaction
should be placed into which sub-dataset.

There are two different ways to implement the randomized horizontal division
method. One is to generate the sub-datasets one by one. The other one is to generate the
multiple sub-datasets simultaneously.

To generate one sub-dataset at a time, we pre-select the transactions in the sub-dataset
first. We randomly generate a set of the transaction identification numbers. Each
transaction identification number represents a transaction that has not been used in the
previous sub-databases and will be put into the current sub-dataset being generated. Then
the database is scanned once, during the scan, the transactions whose transaction numbers
have been pre-selected will be put into the sub-dataset. In the worst case when each
transaction in the sub-dataset needs a separate I/O operation, there will be a total of \( \frac{|D|^2}{N} \)
I/O operations for read and a total of \( \frac{|D|^2}{N \times B \times S_b} \) I/O operations for write in generating all
the sub-datasets. In the best case when all the transactions in the sub-dataset can be read
in to the memory in blocks, there will be a total of \( \frac{|D|^2}{N \times B \times S_b} \) I/O operations for read and
a total of \( \frac{|D|^2}{N \times B \times S_b} \) I/O operations for write in generating all the sub-datasets.
To generate multiple sub-datasets simultaneously, we need to partition the main memory $M$ into $N+1$ buffers whose sizes are $\frac{M}{N+1}$. Among the $N+1$ buffers, $N$ buffers will be used as the output buffers for $N$ sub-datasets, and another one buffer will be used as the input buffer to read the data from the database. We scan the database $D$ once, during the scan, we place the next transaction into one of $N$ buffers randomly. To generate all the sub-datasets, there will be a total of $\frac{|D|}{B \times S_b}$ I/O operations for read. If the size of each buffer is larger than $B \times S$, then there will be a total of $\frac{|D|}{B \times S_b}$ I/O operations for write.

Comparing the two ways of sub-dataset generation, the second one demands fewer I/O operations. But the second one has its restrictions: if the size of the database and number of sub-datasets are both to be large, the main memory may not be able to support $N+1$ buffers at the same time, or although we could allocate the $N+1$ buffers, the size of the buffer may not be larger than $B \times S_b$. In either situation, the total number of I/O operations will increase.

The randomized horizontal division method overcomes the drawback of the horizontal division method by reorganizing the transactions in the database $D$ into different sub-datasets. But it may require more I/O operations, especially when the database is large and we try to divide it into many of sub-datasets.

5.3 Parallel Algorithms

After dividing the database into sub-datasets, the computation for frequent item-sets in parallel is straightforward. We discuss the parallel algorithm with regard to two
implementation techniques: multithread in the SMP architecture and parallel processes
with the message passing in the distributed computer system. We describe two
implementation techniques separately. In both cases, we assume the database is divided
into sub-datasets using the horizontal division method.

5.3.1 Multithread Algorithms in the SMP Architecture

The first multithread algorithm is given in Figure 5-1. We assume the value of $i$ is in
the range of $(1, P)$.

**Step 1**) Divide the database into $P$ subsets using the horizontal division method. Go to
Step 2);

**Step 2**) Let $k = 1$, and $C_1 = I$. Go to Step 3);

**Step 3**) For each processor $i$, transform the data stored in the $i$-th sub-dataset into the
attribute list format. For each item $j$ in $I$, calculate $j$’s local support value. Add $j$’s
local support value into its global support value of the corresponding item in $C_i$.
Go to Step 4);

**Step 4**) Synchronize all the processors. For each processor $i$, check the global support
value of the item-sets from the order of $(i - 1) \times \frac{|C_i|}{P}$ to the order of $i \times \frac{|C_k|}{P} - 1$ in
$C_k$. Delete from $C_k$ the item-sets whose supports are less than the $\text{minsup}$.
Synchronize among all the processors. The remaining item-sets in $C_k$ constitute
$L_k$. Go to Step 5);

**Step 5**) If $L_k$ is empty, then program stops. Otherwise, generate $C_{k+1}$ from $L_k$. Go to Step
6);
Step 6) For each processor \( i \), calculate the local support value for each item-set in \( C_{k+1} \).

At the same time, add the local support value to the global support value of the item-set in \( C_{k+1} \). \( k \) is increased by 1. Go to Step 4);

**Figure 5-1 The 1st multithread algorithm in the SMP architecture**

The order of the frequent item-set generation using the multithread algorithm in Figure 5-1 is different from that using IT in that the multithread algorithm in Figure 5-1 adopts the breadth-first-calculation strategy, that is, all the processors cooperate to generate the whole set of frequent \( k \)-item-sets before the generation of the set of frequent \( (k+1) \)-item-sets begins. IT uses the depth-first-calculation strategy, that is, the calculations of sets of frequent \( k \)-item-sets and sets of frequent \( (k+1) \)-item-sets are overlapped. The depth-first-calculation strategy has the advantage of better cache hit percentage and less memory requirements.

We put forward the second multithread algorithm that deploys the depth-first-calculation strategy. The second multithread algorithm is shown in Figure 5-2.

**Step 1)** Divide the database into \( P \) subsets as even as possible. Go to Step 1);

**Step 2)** For each processor \( i \), transform the data stored in the \( i \)-th sub-dataset into the attribute list format. For each single item in \( I \), calculate the local support value.

Go to Step 3);

**Step 3)** Accumulate the support values for each single item in \( I \) from all the sub-datasets.

The single items in \( I \) whose supports are no less than the \( \text{minsup} \) constitute \( L_1 \).

Mark all the items in \( L_1 \) as unvisited. Let the size \( k \) of current frequent item-sets be 1. Go to Step 4);
Step 4) If all the item-sets in $L_k$ have been visited and $k$ is equal to 1, program stops. If all the item-sets in $L_k$ have been visited and $k$ is not equal to 1, $k$ is decreased by 1. Go to Step 5);

Step 5) If the next unvisited item-set is the first item-set in $L_k$, sort the item-sets in $L_k$ in non-decreasing order according to the length of the local attribute lists. Go to Step 6);

Step 6) For the next unvisited item-set $X$ in $L_k$, all the processors calculate the support values for the item-sets $X \cup Y$, where $Y$ is any item-set that comes after $X$ in $L_k$, using the local attribute lists. After finishing the calculation, sort the $(k+1)$-item-sets generated according into the dictionary order. Mark $X$ as visited. Go to Step 7);

Step 7) Accumulate the support values for each item-set generated in the Step 5). The result is denoted as $C_{k+1}$. Go to Step 8);

Step 8) For each processor $i$, check the global support values of the item-sets from the order of $(i-1) \times \frac{|C_{k+1}|}{P}$ to the order of $i \times \frac{|C_{k+1}|}{P} - 1$ in $C_{k+1}$. Delete the item-sets whose supports are less than the minsup from $C_{k+1}$. The remaining item-sets in $C_{k+1}$ constitute $L_{k+1}$. Mark If $L_{k+1}$ is empty, then go to Step 4. Otherwise, mark all the item-sets in $L_{k+1}$ as unvisited. $k$ is increased by 1. Go to Step 4);

Figure 5-2 The 2nd multithread algorithm in the SMP architecture

Compared with the first multithread algorithm in Figure 5-1, the second multithread algorithm in Figure 5-2 has following disadvantages: First, it has more complicate control mechanism, since it needs to synchronize among the processors more frequently. Second, the processors in the second multithread algorithm need to keep sorting the item-
sets, which results in some overheads. On the other hand, the second multithread algorithm in Figure 5-2 has following advantages: First, for each processor, its local cache has better cache hit percentage, since the depth-first-calculation strategy here takes the advantage of the data locality. The data locality refers to the property that the data that is being accessed will probably be accessed again in the near future. Second, the second multithread algorithm has better computation efficiency. Given a set $L_k$ of attribute lists, each processor sorts the attribute lists in $L_k$ according to the local data it has. The attribute lists may have the different orders in the different sub-datasets, since an item-set $X$ may have more support in one sub-dataset but have less support in other sub-datasets. Assume item-sets $X$ and $Y$ have the attribute lists $l_x$ and $l_y$ from the original database and $|l_x| < |l_y|$. As the database is divided into $N$ sub-datasets, $l_x$ is also thought of as consisting of $N$ parts with part $i$, here $1 \leq i \leq N$, corresponding to the $i$-th sub-dataset. If the attributes in $l_x$ and $l_y$ are not evenly distributed over the set of transactions in the database, although the condition $|l_x| < |l_y|$ is true, in some sub-datasets the attributes belong to $l_y$ may outnumber that belong to $l_x$. This situation is illustrated in Figure 5-3 below.

![Figure 5-3 Attribute distributions in attribute lists](image)

In Figure 5-3, there are two attribute lists, $l_x$ and $l_y$. $l_x$ consists of 11 attributes, and $l_y$ consists of 10 attributes. Each attribute in attribute lists is denoted as a solid circle.
Assume the database is divided into 3 sub-datasets, denoted by part 1, part 2 and part 3, the attributes in the same attribute list are also thought of as being divided into 3 parts. Both IT and the first multithread algorithm in Figure 5-1 sort the item-sets based on the unit of the attribute list. According to these algorithms, the attribute list $l_x$ should be placed before $l_y$, since $l_x$ has more attributes. After a close look at the two attribute lists in Figure 5-3, we find out that although $l_x$ has more attributes than $l_y$ does, $l_y$ do have more attributes than $l_x$ does both in part 1 and part 3. The multithread algorithm in Figure 5-2 sort the attributes in different parts separately, so to the part 1 and part 3, $l_y$ will be placed before $l_x$; to the part 2, $l_x$ will be placed before $l_y$. Remember that the total comparison times will be reduced if the longer attribute lists are placed before the shorter ones. So the multithread algorithm in Figure 5-3 has better computation efficiency.

5.3.2 Parallel Algorithms in Distributed Computer System

In the SMP architecture, the processors exchange the information with each other through accessing the common memory cells, which is efficient in terms of the reduced communication costs. On the other hand, the size of the problem that the SMP architecture can deal with efficiently is limited by the capacity of the computer’s main memory. In this section, we discuss how to implement the parallel algorithm in the environment of the distributed computer system.

We divide the original database into $N$ sub-datasets, where the value of $N$ may be greater than that of $P$. We assume the size of each sub-dataset is small enough such that the data in each subset can be calculated in the main memory of each computer, otherwise, we will increase the value of $N$ and reduce the size of the sub-datasets. We number the set of sub-datasets from 1 to $N$. 
The parallel algorithm used in the environment of distributed computer system is shown in Figure 5-4 below. For the simplicity, we assume the number \( N \) of sub-datasets is the multiple integral of the number \( P \) of processors.

**Step 1)** Divide \( N \) sub-datasets into \( P \) sets such that each set contains \( \frac{N}{P} \) sub-datasets.

Assign one processor to each set of sub-datasets. For each processor, it calculates the sets of local frequent item-sets on each sub-datasets within the assigned set of sub-datasets using the INSERT algorithm discussed in Chapter 3. Go to Step 2);

**Step 2)** Assume the size of the maximum local frequent item-sets in all the sub-datasets is \( S \). If the value of \( S \) is less than the value of \( P \), then divide the \( P \) processors into \( S \) parts such that each part except the last one contains \( \frac{P}{S} \) processors. The last part will contain \( P - (S - 1) \times \frac{P}{S} \) processors. For the set of processors in part \( i \) here \( 1 \leq i \leq S \), calculate the accumulative support for each item-set whose size is \( i \).

If the value of \( S \) is greater than or equal to the value of \( P \), then divide the sets of frequent item-sets into \( P \) parts such that the first part consists of the item-sets whose sizes are in the range of \( (1, \frac{S}{P}) \), the second part consists of the item-sets whose sizes are in the range of \( (\frac{S}{P} + 1, 2 \times \frac{S}{P}) \), ..., the last part consists of the item-sets whose sizes are in the range of \( ((P - 1) \times \frac{S}{P} + 1, S) \). The processor \( j \) here \( 1 \leq j \leq P \) calculates the accumulative support of each item-set whose size is in
the range of \( \left( (j - 1) \times \frac{S}{P} + 1, j \times \frac{S}{P} \right) \). After accumulating the support value for each item-set, the processor checks to see which type of the item-sets it calculates belong to. The processor puts the item-sets whose type is 1 into the set of \( F \), and put the item-set whose type is 2 or 3 into \( C \). Go to Step 3);

**Step 3)** Assign each processor to the original set of sub-datasets in Step 1). For each sub-dataset in the set of sub-dataset, calculate the support of the item-sets in \( C \) resulted from the Step 2). Go to Step 4);

**Step 4)** The local support values for the item-sets in \( C \) are accumulative from all the sub-datasets using \( P \) processors. The method how to assign the \( P \) processors to make the calculation is the same as used in Step 2). After the accumulation, the item-sets whose supports are no less than the \( \text{minsup} \) are put into the set \( F \). The item-sets whose supports are less than the \( \text{minup} \) are discarded. The program stops.

**Figure 5-4 Parallel algorithms in distributed computer systems**
CHAPTER 6
(1+\varepsilon)-PASS ALGORITHM

When the database becomes large and cannot be held by the computer’s physical main memory, it is expected that we will encounter a lot of I/O operations during the process of mining association rules. There are at least two ways to avoid the computation performance deterioration due to the time awaiting I/O operations to finish: 1) try to overlap the computation operations with the I/O operations as much as possible; 2) try to reduce the total number of I/O operations needed. However, if the computations involved are data dependent on the results of the I/O operations and previous computations, the first method becomes not quite effective. The second method might be efficient. But the reduced number of I/O operations can not only be achieved by increasing the amount of the data for one I/O operation. This is not only because any computer system has its limit to deliver a certain amount of data through one I/O operations but also because sometimes the data available to be delivered are limited at the time of I/O operations happen. Thus, in order to reduce the total I/O operations, we also need to try to reduce the total amount of the data to be delivered by I/O operations.

As to the task of mining association rules, to calculate the set of frequent item-sets, the computer needs to read the data stored in the disk into the computer’s main memory through I/O operations. If the database is small enough such that all the data can reside in the main memory at the same time, then there will be a total of $\frac{|D||T|}{BS_b}$ I/O operations for read, where $|D|$ is the number of transactions in the database $D$, $|T|$ is the average length...
of the transactions, $B$ is the number of blocks that the processor can read from the hard
disk into the memory through one I/O operation, $S_b$ is the block size measured in byte.
Some algorithms, like Apriori, need to scan the database multiple times. As the data in
the database are all in the main memory, the multiple scans of the database does not
result in an increase of I/O operations.

On the other hand, the size of nowadays databases may be hundreds of gigabytes or
even terabytes, with the size of the main memory is hundreds of megabytes in the most
recent PC desktops and several of gigabytes in the common high computing performance
workstation servers, it is impossible for us to keep all the data together in the memory,
instead we have to store the data in the hard disk and read the data into the memory when
needed. In such a situation, when the database is scanned for multiple times, it will be
expected that the total number of I/O operations will be increased and be proportional to
the number of the scan times.

In the rest of the chapter, we consider how to deal with the situation when the size of
the database is large, and the database can not physically fit into the computer’s main
memory. We focus on the analysis of the total I/O operations needed. For simplicity, in
our discussion, if the database needs to be scanned for multiple times, we assume the
computer does not keep the data between the scans of the database in its main memory
hierarchy. In other words, if after a scan, the program needs to access some data it once
read, it has no choice but to read it again from the database in the hard disk with
additional I/O operations.

6.2 Problem Description and Related Work

It is interesting to answer the question: Is it possible for us to calculate the sets of
frequent item-sets by scanning the database once? The answer seems to be positive at the
first glance. If we know the set \( I \) of items in the database, we could first generate all the possible candidate frequent item-sets by making combinations between different items. Then we scan the database once, during the scan the support of each candidate frequent item-set is calculated. At the end of the scan, the item-sets whose supports are no less than the \( \text{minsup} \) are the frequent item-sets. However, the method is not feasible in reality, since the item-sets generated in such way may even larger than the database itself. For example, assume the number of items in \( I \) is 10,000, and the size of the candidate frequent item-sets is 6, then there will be a total of \( \sum_{k=1}^{10000} \binom{10000}{6} \approx 1.39 \times 10^{31} \) candidates.

Instead of finding an algorithm that calculates sets of frequent item-sets by scanning the database exactly once, we seek the solution that scans the database approximately once. In other words, we hope to find a solution that scans only small part of the database twice and scans the rest of the data exactly once. We use \( \varepsilon \) to stand for the fraction of the data in the database that need to be scanned twice, here \( \varepsilon \) belongs to \([0, 1]\). If \( \varepsilon \) is 0, then the database is scanned only once. If \( \varepsilon \) is 1, then the database is scanned twice. Obviously, we hope that the value of \( \varepsilon \) is as small as possible.

Parititon, put forwarded by Savasere, Omieckinski, and Navathe, is the algorithm designed to improve the performance of Apriori. It is also effective to deal with the situation when the size of the database exceeds the capacity of the computer’s physical main memory. Briefly speaking, Partition is made up of four steps. In the first step, Partition divides the database into small sub-datasets. Then it deploys Apriori to calculate the sets of local frequent item-sets on each sub-dataset. In the second step, Partition collects the item-sets that are local frequent item-sets in at least one of the sub-
datasets. We denote the resulting item-set collection as the set $S$. The support value for each item-set in the set $S$ was set to 0. In the third step, for each sub-dataset, Partition computes the support value for each item-set in $S$. In the fourth step, the support values from different sub-datasets for the same item-set in $S$ are accumulated. The item-sets whose supports are no less than the $\text{minsup}$ will be placed into the set of frequent item-set. The item-sets whose supports are less than the $\text{minsup}$ will be discarded.

Obviously, Partition scans the database exactly twice. So the total I/O operations for read will be $2 \times \frac{|D||T|}{BS_b}$.

Toivonen proposed Sampling algorithm. Sampling consists of four steps. In the first step, Sampling randomly samples $s$ transactions from the database $D$. Then it calculates the sets of the local frequent item-sets from the sampling data. We use the symbol $S$ to stand for the result. In the second step, Sampling computes the negative border of $S$. The negative border of $S$ is calculated according to the following rule: if a $k$-item-set $X$ is not a local frequent item-set in the sampled data, but all the $(k-1)$-item-sets that are subsets of $X$ are local frequent $(k-1)$-item-sets in the sampled data, then $X$ is added into $S$. After all such item-sets have been put into $S$, $S$ becomes the negative order of its original sets of item-sets. In the third step, the database is scanned once, during the scan, the support values of all the item-sets in $S$ are calculated using Apriori. In the fourth step, the support value of each item-set in $S$ is checked. All the item-sets whose support values are no less than the $\text{minsup}$ would be placed into the set of frequent item-sets, denoted as $F$. If there is a $k$-item-set $X$ such that $X$ is not in the set $F$ but all of its subsets that were $(k-1)$-item-sets are in $F$, then $X$ would be put into a set denoted as $Q$. As it is possible that the item-sets in $Q$ are frequent ones in the sampled database, we have to calculate the support
value for each of them. The support calculations for the item-sets in $Q$ result in another pass of the database scan. During the scan, the support of each item-set in $Q$ is accumulated. Then the set $Q$ is checked again. The item-sets whose support values are no less than the $\text{minsup}$ would be added into the set $F$. The check on the set $F$ is followed. If there exist a $k$-item-set $X$ such that $X$ is not in the set $F$ but all of its subsets, which are $(k-1)$-item-sets, are in $F$, then the database is scanned again and the procedure described in the fourth step is repeated. This procedure continues until no item-sets like $X$ could be found any more.

When the size of the sample is relative large, the computation results produced from the sampling data would give more accurate prediction for the sampled database. If we assume the capacity of the main memory of the computer is denoted as $M$, then the total number of I/O operations for read in the sampling algorithm will be at least as

$$\frac{M}{B_{S_b}} + n \times \frac{|D||T|}{B_{S_b}}$$

where $\frac{M}{B_{S_b}}$ is the number of I/O operations for reading the sampling data into the memory, $n$ is a variable, and the value of which stands for the number of passes that the database is scanned.

The number of database scan in Sampling can not be pre-determined and depends on the real data. On the other hand, the procedure of sample data randomly could also be expensive.

### 6.3 $(1+\epsilon)$-Pass Algorithm

#### 6.3.1 Basic Idea

Suppose that before the program starts, the database is divided into $N$, where $N>1$, sub-datasets, and each subset can be calculated within the main memory. The methods for the database division are the same as the same as discussed in section 5.2. We divide
the $N$ sub-datasets into $P$ groups, where $P$ is a positive integer, such that each group contains $\frac{N}{P}$ sub-datasets. All the groups are assumed to be numbered from 1 to $P$. The sub-datasets in each group are also ordered. We call an item-set $X$ is a local frequent item-set in a sub-dataset if its support in the sub-dataset is no less than the value of $\frac{\min \sup}{N} - r$, where $r$ is an integer variable whose value is greater than 0. The value of $r$ is pre-defined and could be treated as a constant as long as its value has been determined. The reason we introce the variable $r$ here is that some frequent item-sets in the database may not be local frequent in all of the sub-datasets. We want to avoid missing the support calculations for such item-sets by reducing local minimum support for each sub-dataset.

We also associate two attributes, denoted as $g$ and $s$, with each item-set to be calculated. The attribute $g$ of the item-set $X$, denoted as $X_g$, indicates the order of the group that starts to calculate the support of $X$. The attribute $s$ of the item-set $X$, denoted as $X_s$, stands for the number of sub-datasets that have contributed to the support calculation for $X$.

$(1+\varepsilon)$-pass algorithm consists of two phases. The first phase works as follows. Starting from the first group until the last one, for each sub-dataset in a group, IT is applied to each of them. For every sub-dataset in the first group, we only calculate the set of local frequent item-sets which are put into a set $C$. The attribute $g$ of each local frequent item-set in the set $C$ is set to 1 whenever it is put into $C$ for the first time. If the item-set that is being added into $C$ already exists, the value of the attribute $g$ of the item-set in $C$ is increased by 1, and the support value of the item-set in $C$ is also accumulated.
After making the calculations on each sub-dataset in the first group, we examine all the item-sets in $C$ so far. Assume the support value of an item-set $X$ in the current set $C$ is denoted by $X_{support}$, if Inequality 6-1

$$X_{support} - \frac{\minsup}{N} \times g \geq \left( \frac{N}{P} - g \right) \times r \geq 0$$

(6-1)

is satisfied:, we will continue to keep the item-set $X$ in the set $C$. Inequality 6-1 comes from the observation: if the item-set $X$ is not a frequent one in a sub-dataset, then the maximum possible support value for $X$ in the sub-dataset is no more than $\frac{\minsup}{N} - r$. If the item-set $X$ is not a frequent one in the $(\frac{N}{P} - g)$ sub-datasets, the total support value for $X$ from these sub-datasets is no larger than $(\frac{\minsup}{N} - r) \times (\frac{N}{P} - g)$. If the item-set $X$ is a frequent one in the current group, then the total support value of

$$X_{support} + (\frac{\minsup}{N} - r) \times (\frac{N}{P} - g)$$

should be at least $\frac{\minsup}{N} \times \frac{N}{P}$, which means Inequality 6-1 should be satisfied. Also, if the support of the item-set $X$ is calculated in every sub-dataset, the attribute $s$ of $X$ is set to 0. Otherwise, the attribute $s$ is set to 1 and the support value of $X$ is also set to 0. To make the calculation on each sub-dataset of any subsequent group $i$ here $1 < i \leq P$, IT is deployed, too. For any current sub-dataset, we need to calculate the support values for two kinds of item-sets: 1) the item-set is a local frequent item-set, but it may or may not exist in the set $C$; 2) the item-set is not a local frequent item-set, but it does exist in the set $C$. For the first kind of the item-sets, we calculate its support value in the current sub-datasets. Then we add the item-set and its support value into $C$. If the item-set already exists, then the support values are added. If the value of the attribute $s$ of the item-set in $C$ is equal to $-1$, which means the item-set is put into the set
by some previous sub-dataset in the current group and does not exist in the set \( C \) when the calculations on the group \( i-1 \) finish, then the value of attribute \( g \) is increased by 1. If the item-set does not exist, we then set its attribute \( s \) to \(-1\) and the attribute \( g \) to 1. For the second kind of item-sets above, we just simply sum the support value of the item-set already in \( C \) and the value calculated in the current sub-dataset. At the end of the group \( i \), we examined all the data in the set \( C \). For any item-set, if the value of its attribute \( s \) is not equal to \(-1\), we use Inequality 6-2 to make judgements that whether or not the item-set \( X \) has the possibility to be frequent in the original database:

\[
X_{support} - \frac{\text{minsup}}{P} \times (i - X_s + 1) \geq 0 \tag{6-2}
\]

Inequality 6-2 comes from the observation that if the order of the current group is \( i \), then there will be a total of \( i-X_s+1 \) groups between the group \( i \) and the group whose order equals \( X_s \). If item-set \( X \) is a frequent one in some group, then its support value in the group should be no less than \( \frac{\text{minsup}}{P} \). So if the item-set \( X \) is a frequent one in a total of \( i-X_s+1 \) groups, Inequality 6-2 must be true. If Inequality 6-2 is satisfied, we then keep the item-set \( X \) in the set \( C \) for the further calculation. Otherwise, we remove it from the set \( C \). If the attribute \( s \) of the item-set in the set \( C \) is equal to \(-1\), we make the decision according to the result of Inequality 6-2. If Inequality 6-2 is satisfied, we put the item-set into the set \( C \) with the attribute \( s \) set to \( i+1 \) and the support value set to 0. Otherwise, the item-set is discarded. Note that, if the group \( i \) is the last group of the database, we also transfer from the set \( C \) to the set \( F \), the set of frequent item-sets, those item-sets that meet the following two conditions: 1) support values are no less than the \( \text{minsup} \); 2) values of \( s \) attribute are equal to 1.
In the second phase of the \((1+\varepsilon)\)-pass algorithm, we restart to scan the database from the beginning. With each sub-dataset in the group \(i\), where \(1 \leq i \leq P\), we calculate the support values for each item-set in the set \(C\) on the condition that \(C\) is not empty. Whenever the end of each group is reached, we need to check every item-set in the set \(C\) again. For the item-set \(X\), if Inequality 6-3

\[
X_{\text{support}} + \left(\frac{\minsup}{p} - 1\right) \times (X_s - i - 1) \geq \minsup
\] (6-3)

is satisfied, item-set \(X\) will be kept in the set \(C\) for further calculations. Otherwise, it will be deleted. Inequality 6-3 comes from the following observation: if the current group is \(i\), then the total number of groups from which we have calculated the support values for \(X\) is determined by \(p - X_s + i - 1\). Then the number of remaining sub-datasets to be calculated is \(P - (p - X_s + i - 1) = X_s - i + 1\). As for each remaining group, the support value for the item-set \(X\) is no more than \(\frac{\minsup}{p} - 1\), then the maximum support value for the item-set \(X\) in the database is no more than \(X_{\text{support}} + \left(\frac{\minsup}{p} - 1\right) \times (X_s - i - 1)\), which should be greater than or equal to the \(\minsup\). Also at the end of the group \(i\), we transfer from the set \(C\) to the set \(F\) those item-sets that satisfy the following two conditions: 1) Inequality 6-3 is true; 2) The value of the attribute \(s\) equals \(i + 1\).

It is unnecessary for us to scan all of the sub-datasets in the second phase. Whenever the set \(C\) becomes empty, the program will stop and all the frequent item-sets in the database have already been calculated and stored in the set \(F\). Thus the total I/O operations needed by \((1+\varepsilon)\)-pass algorithm are \((1 + \varepsilon) \times \frac{|D||T|}{BS_h}\).
6.3.2 Description of (1+ε)-Pass Algorithm

According to the above discussion, a formal description of the (1+ε)-pass algorithm is summarized in Figure 6-1.

**Step 1)** Determine the value of $P$. Divide $N$ sub-datasets into $P$ groups such that each group consists of $\frac{N}{P}$ sub-datasets. Let $C = F = \phi$, where $C$ stands for the set of candidate frequent item-sets, $F$ represents the set of frequent item-sets. Determine the value of variable $r$. Go to Step 2);

**Step 2)** For each sub-dataset in the first group, calculate the local frequent item-sets whose supports are no less than $\frac{\text{minsup}}{N} - r$. Put the local frequent item-sets into the set $C$. If the item-sets already exist, then the corresponding support values are accumulated, and the values of the attribute $g$ are increased by 1. If not, the values of attribute $g$ of the item-sets are set to 1. Go to Step 3);

**Step 3)** For any item-set $X$ in the set $C$, if the support value of $X$ satisfies Inequality 6-1, then keep $X$ in the set $C$. If the value of attribute $g$ of $X$ equals to $\frac{N}{P}$, set the value of its attribute $s$ to 1, otherwise, set the value of its attribute $s$ to 2. If the support value of $X$ does not satisfy Inequality 6-1, remove $X$ from the set $C$. Go to Step 4);

**Step 4)** For each sub-dataset $i$ in the subsequent group, calculate the local frequent item-sets whose supports are no less than $\frac{\text{minsup}}{N} - r$. For each calculated local frequent item-set $X$, add the item-set $X$ into the set $C$. If the item-set $X$ already exists, the corresponding support values are accumulated. If the value of the $s$ attribute of the item-set $X$ in the set $C$ does not equal 0, increase the value of
attribute \( g \) by 1. If the item-set \( X \) does not exist in the set \( C \), set the value of attribute \( g \) to 1. Also calculate and accumulate the support values of those item-sets that does exist in the set \( C \) but not be local frequent in \( i \). Go to Step 5);

**Step 5**) At the end of any group \( i \), check the set \( C \). For any item-set \( X \) in the set \( C \), if the value of attribute \( s \) equals to 0 and the support value of the item-set \( X \) satisfies Inequality 6-1, keep the item-set \( X \) in the set \( C \) and set the value of its attribute \( s \) to \( i+1 \). If the value of attribute \( s \) equals to 1 and the support value of the item-set \( X \) satisfies Inequality 6-3, also keep the item-set \( X \) in the set \( C \). Remove all the other item-sets from the set \( C \). Go to Step 6);

**Step 6**) After all the sub-datasets are calculated. Transfer from the set \( C \) to the set \( F \) those item-sets whose values of attribute \( s \) equal to 0. Go to the Second Phase.

First Phase

**Step 1**) Set \( i \) to 1. Go to Step 2);

**Step 2**) If the set \( C \) is empty, program stops. Otherwise, for each sub-dataset in the group \( i \), calculate the support value for each item-set in the set \( C \). At the end of the group \( i \), check all the item-sets in the set \( C \). For any item-set \( X \) in the set \( C \), if the support value for \( X \) does not satisfy Inequality 6-3, delete the item-set \( X \) from the set \( C \). If the value of \( X \)'s attribute equals to \( i \), transfer \( X \) from the set \( C \) to the set \( F \). \( i \) is increased by 1. Repeat Step 2;

Second Phase

Figure 6-1 Description of \((1+\varepsilon)\)-pass algorithm

To show that the \((1+\varepsilon)\)-pass algorithm in Figure 6-1 is correct in the sense that it identifies all the frequent item-sets in the database, we have the following claim.
Claim: The \((1+\varepsilon)\)-pass algorithm in Figure 6-1 finds the set of all frequent item-sets in the database.

Proof:

Assume there is a frequent item-set \(X\) in the database but is not found using the \((1+\varepsilon)\)-pass algorithm in Figure 6-1. To see such item-set does not exist, we establish a directed graph which consists of \(P\) nodes. Each node corresponds to a group and is labeled by the group order. Each node has an attribute, the value of which is equal to the support value for the item-set \(X\) in the group it represents. There are exactly \(P\) edges in the graph, which constitute a directed cycle. There is an edge between the node \(i\) and the node \(j\) with the tail at the node \(i\) and the head at the node \(j\), where \(1 \leq i, j \leq P\), if and only if \(i < j\) and \(i = j - 1\) or \(i = P\) and \(j = 1\). The edge between the node \(i\) and the node \(j\) is denoted as \((i, j)\).

Each edge \((i, j)\) in the graph has a weight, denoted as \(w_i\), which is calculated by the value of the attribute of the node \(i\) minus \(\frac{\text{minsup}}{P}\). As the item-set \(X\) is assumed to be a frequent one in the database, the sum of the weights of all the edges in the graph are no less than 0.

If sum of the weights of all the edges are equal to 0, then Inequality 6-1 is satisfied after calculating on the first group, which ensures that the item-set \(X\) will be put into the set \(C\). Also for any subsequent group, both Inequalities 6-2 and 6-3 are satisfied, which ensures the item-set \(X\) to be kept in the set \(C\) by all the groups and be moved to the set \(F\) at the end of the first phase. If not all the weights of the edges are equal to 0, according to the pigeon hole theory, there must exist at least one edge such that the corresponding weight is greater than 0. We denote all the edges whose weights are greater than 0 as \(w_{j_1}, w_{j_2}, ..., w_{j_k}\), where \(1 \leq j_1 < j_2 < ... < j_k \leq P\). We check these edges from \(j_1\) to \(j_k\) sequentially, and the check stops whenever we met the edge \(j\), here \(j_1 \leq j \leq j_k\), which meets the following
conditions: 1) all the prefix sums of the weights from edge 1 to edge \( j-1 \) are less than 0; 2) all the prefix sums of the weights from edge \( j \) to edge 1 are greater than 0. The edge \( w_j \) must exist, otherwise, the sum of the weights of all the edges is less than 0. The existence of \( w_j \) indicates that the item-set \( X \) will be kept in the set \( C \) when the second phase starts. Next we still need to show that if the \((1+\varepsilon)\)-pass algorithm stops in the second phase after it makes calculations on \( w' \), then either \( w' = w_{j-1} \) or \( w' > w_{j-1} \). The item-set \( X \) can not be removed from the set \( C \) before the program stops at \( w' \), otherwise, the sum of the weights of all the edges will be less than 0. Assume when the \((1+\varepsilon)\)-pass algorithm stops in the second phase after it makes calculations on \( w' \) and \( w' < w_{j-1} \), then when the program stops, the support value for the item-set \( X \) has not been calculated on all the groups. In other word, the set \( C \) can not be empty. So the programs can not stop until it met the edge of \( w_{j-1} \) or pass the edge \( w_{j-1} \). So the item-set \( X \) does not exist.

So the claim has been proved.

\[ \square \]

6.3.3 Implementation Issues

In the second phase of the \((1+\varepsilon)\)-pass algorithm, we do not keep all the data from each sub-dataset in the memory. Instead, we only need to keep the data information that is related to calculate the support value of the item-sets in the set \( C \). To differentiate the useful data and the useless data, we keep a bitmap \( bmp \) of the set \( C \). The structure of the \( bmp \) could be implemented by an array. The length of the array depends on the total number of different items in \( I \), \(|I|\). Initially, all the entries in \( bmp \) are set to 0. At the end of the first phase of the \((1+\varepsilon)\)-pass algorithm, when the support values of the item-sets in the set \( C \) are checked for the last time, we also set the entry values in \( bmp \). The method is simple: if the item-set \( X \) will still be kept in the set \( C \) after the first phase, for each item \( q \)
in $X$, if the value of the $q$-th entry in $bmp$ is equal to 0, then the entry value is set to 1. In the second phase of the $(1+\varepsilon)$-pass algorithm, we only keep the attribute lists for those items whose entry value in $bmp$ is equal to 1. The entry values of $bmp$ will be reset each time the set $C$ is examined in the second phase using the same above method.

The determination of the value of the variable $r$ in the first phase of the $(1+\varepsilon)$-pass algorithm depends on the real data set. In our implementations, we usually set $r$ to 2. We also tried not to fix the value of $r$ in different groups, instead we set the value of $r$ to a relative large value and then gradually reduced it. But the experimental results did not be satisfactory.
7.1 Abstract

The problem of our interest takes as input a database of $m$ sequences from an alphabet $\Sigma$ and an integer $k$. The goal is to report all the pairs of sequences that have a matching subsequence of length at least $k$. We employ two algorithms to solve this problem. The first algorithm is based on sorting and the second is based on generalized suffix trees. We provide experimental data comparing the performances of these algorithms. The generalized suffix tree based algorithm performs better than the sorting based algorithm.

7.2 Introduction

We consider the following sequence analysis problem: Given a database of $m$ sequences and an integer $k$ find all pairs of sequences that share a common subsequence of length at least $k$. Tools for identifying similarities among biological sequences are used by biologists on a regular basis. One example is the BLAST program housed at NIH.

In this paper we provide two different solutions to this problem. The first solution is based on sorting and the second is based on generalized suffix trees (GSTs). Our experimental data indicate that the GST based algorithm is faster. The preprocessing time is more for the GST based algorithm. Since this is done only when the trees are built, this is not a drawback.
In section 7.3 we provide a summary of the sorting based algorithm. Section 7.4 is devoted to a discussion of the GST based algorithm. In section 7.5 we compare the two algorithms empirically. Section 7.6 provides some concluding remarks.

### 7.3 Sorting Based Algorithm (SBA)

The idea of SBA is to identify substrings of length $k$ from each sequence; sort all these substrings in alphabetical order (so that identical subsequences come closer); scan through the sorted list to identify identical subsequences and hence output the relevant pairs.

A detailed description of the algorithm follows. Input is $m$ sequences $s_1, s_2, \ldots, s_m$ from some alphabet $\Sigma$ and an integer $k$.

1) Let $\text{sum}=0$. Employ an array $\text{len}[1:m]$. Initializing each element of $\text{len}[]$ to zero;

2) For every $i$, $1 \leq i \leq m$, read the input string $s_i$ and let $\text{sum} = \text{sum} +$ the number of substrings of length $k$ in $s_i$.

3) Use an array $A[1:\text{sum}]$.

4) Put all subsequences of length $k$ from the input sequences into $A$.

5) Use radix sort algorithm to sort all the substrings in $A$ according to alphabetical order. When sorting, the subsequences are divided into several subsections. After finishing a subsection’s sorting, the total substrings are examined to discard the substrings which are surely not matched by other substrings. Continue with the sorting of remaining subsequences.

6) Scan through $A$ to find all the subsequences which have a match with the neighboring subsequences.
7.4 GST Based Algorithm (GSTBA)

Suffix trees have been employed in the past to solve string matching problems and approximation string matching problems in sequence and in parallel.

A suffix tree is a trie-like data structure representing all suffixes of a string. Its linear time construction algorithm is described by Farach [51].

Each edge in the suffix tree is labeled with a symbol from the alphabet $\Sigma$. The sequence of labels in any path from the root to a leaf corresponds to a suffix of the string. We can also associate a string with every node (not necessarily leaves) in the tree. We can say that the node represents the corresponding string. Note that this string need not be a suffix.

A suffix tree for the sequence agcata is shown in Figure 7-1.

Figure 7-1 A suffix tree for the string agcata.

Suffix trees have the following properties. Let $s$ be the sequence under consideration.

1) Every suffix of $s$ is represented by a leaf in the tree. 2) Let $x$ and $y$ be two nodes in the
tree. If \( x \) is an ancestor of \( y \), then the string that \( x \) represents is a prefix of the string that \( y \) represents. 3) If the nodes \( x_1, x_2, \ldots, x_k \) represent the strings \( X_1, X_2, \ldots, X_k \), then the least common ancestor of \( x_1, x_2, \ldots, x_k \) represent the longest common prefix of \( X_1, X_2, \ldots, X_k \).

We can also conceive of a suffix tree corresponding to multiple sequences and we end up with a GST. In a GST, every suffix of every sequence is represented as a leaf. The properties mentioned for suffix trees hold for GSTs as well. If multiple sequences have the same suffix, then there will be a leaf corresponding to each such sequence. A GST can be constructed in time linear in the total length of all input strings. Figure 7-2 shows a GST for the strings gate, atc, and catag.

![Figure 7-2 GST for gate, atc, catag](image)

We will employ the GST data structure to solve the sequence analysis problem of our interest. One of the problems that arise is the color set size (CSS) problem introduced by Farach [51].
7.4.1 Color Set Size (CSS) Problem

Let $T$ be any tree in which each leaf has been colored with a color from the set \{1, 2, \ldots, q\}. The color set size of any node $v$ in $T$, denoted as $\text{css}(v)$ is the number of different colors that can be found in the subtree rooted at $v$. The color set size problem is to determine the color set size of each internal node in $T$.

For example, in Figure 7-3, the color set size of the nodes $p$, $m$, $n$ are 3, 2, 3, respectively.

Figure 7-3 A tree with colored leaves

An optimal $O(n)$ time algorithm for the CSS problem has been given by Chi and Hui [50]. Here $n$ is the number of nodes in the tree. This algorithm can be summarized as follows. Let LeafList refer to the list of leaves in $T$ ordered according to a post-order traversal of the tree. The LeafList of the tree in Figure 7-3 is $a, b, c, d, e, f, g, h$. Also, for any leaf $x$ of color $c$, let $\text{LastLeaf}(x)$ stand for the last leaf that precedes $x$ in LeafList which has the same color $c$. In Figure 7-3, $\text{LastLeaf}(c) = a; \text{LastLeaf}(f) = d; \text{LastLeaf}(e) = \text{Nil};$ and so on.
For any vertices \( u \) and \( v \) in \( T \), let \( LCA(u,v) \) stand for the least common ancestor of \( u \) and \( v \). For example, \( LCA(a,c) = l; \) \( LCA(d, e) = m; \) \( LCA(d, h) = p; \) and so on in Figure 7-3.

For any node \( u \) in \( T \), \( Subtree(u) \) stands for the Subtree rooted at \( u \) and \( LeafCount(u) \) stands for the number of leaves in \( Subtree(u) \). An internal vertex \( x \) is said to be a \( ColorPairLCA \) (of color \( c \)) if there exist leaves \( u \) and \( v \) of color \( c \) with \( LastLeaf(v) = u \) and \( x = LCA(u, v) \). In the tree of Figure 7-3, the node \( l \) is a \( ColorPairLCA \) of color 1 (due to the pair \( a, c \)). The nodes \( m \) and \( n \) are not \( ColorPairLCAs \) for any color.

For any internal node \( x \) of \( T \) let \( CPL-Count(x) = k \) if among all colors there are \( k \) leaf pairs for which \( x \) is their \( ColorPairLCA \). For example, in Figure 7-3, \( CPL-Count(p) = 5 \) (due to the pairs \( (a, c), (g, c), (d, b), (f, d), \) and \( (h, e) \)). \( CPL-Count(m) = 0 \). Also let \( Duplicate(x) = \sum_{v \in Subtree(x)} CPL-Count(v) \).

It can be shown that for any node \( x \) in \( T \), \( css(x) = LeafCount(x) - Duplicate(x) \). The linear time algorithm for \( CSS \) proceeds to compute \( LeafList, LeafCount(), LastLeaf(), CPL-Count(), \) and \( Duplicate() \), in this order. Finally, it computes \( css() \). Computation of \( LeafList, LeafCount, \) and \( Duplicate \) are straightforward and can be done using a post-order traversal of the tree. \( LastLeaf() \) can be computed for every node as follows. Traverse \( LeafList \) from left to right. Keep an array \( Last() \) such that \( Last(c) \) is the last seen node of color \( c \). Whenever a leaf \( x \) is encountered of color \( c \), set \( LastLeaf(x) = Last(c) \) and update \( Last(c) \).

To compute \( CPL-Count() \), we make use of the constant time algorithm for computing least common ancestors [8]. For every leaf \( x \) compute \( u = LCA(x, LastLeaf(x)) \) and increment \( CPL-Count(u) \) by 1. Once we have all the above values \( css() \) can be computed easily.
7.4.2 The GSTBA Algorithm

The sequence analysis problem of our interest can be solved now as follows. First construct a GST corresponding to all the database sequences. A simple algorithm for constructing a suffix tree for a single string has been proposed by Nelson [60], which is based on the algorithm put forward by Ukkonen [18]. We have generalized this algorithm in a nontrivial way. Because in GST each internal node contains a pointer to the next prefix of current prefix node and each leaf node may represent the same suffix that occurs in multiple strings, the implementation of GST will be different from that of the single string suffix tree. Note that, when dealing with the last character of a string, the construction procedure should no stop at the endpoint until it reaches root node so that each leaf node gets the right color size. The color given to any leaf (i.e., suffix) is nothing but the sequence number that the suffix comes from.

After constructing a GST, we solve the CSS problem for this GST. In order to solve the sequence analysis problem of our interest, we do the following. We perform a breadth-first search in the GST. At each node $x$, if its distance from the root is greater than or equal to $k$, $x$ is an internal node, and $css(x) > 1$, we report all the relevant pairs. This is done using two pointers start_point and end_point that index to the first and last leaves of the subtree rooted at $x$. Clearly this algorithm runs in time $O(n+q)$ where $n$ is the number of nodes in the GST and $q$ is the size of the output. Thus we arrive at the following Theorem.

**Theorem 7-1** The sequence analysis of our interest can be solved in time $O(n+q)$, where $n$ is the size of the GST and $q$ is the size of the output. The preprocessing time is $O(n)$ and the memory needed is also $O(n)$. 
7.5 An Experimental Comparison of SBA and GSTBA

In this section we report experimental data obtained in the comparison of the two algorithms SBA and GSTBA. These results were collected on the Sun workstation Ultra Enterprise 4000. This machine has a memory of 2GB and its operations system is SunOS 5.6. Both GSTBA and SBA were tested on randomly generated DNA sequences.

Figure 7-4, 7-5, 7-6, and 7-7 show the results of our experiments. These graphs compare GSTBA with SBA. Two different versions of SBA were built, one based on radix sort and the other based on quick sort. For a description of radix and quick sort algorithms please see the book written by Ellis, Sartah, and Sanguthevar [39].

Figure 7-4 corresponds to the case of 2000 strings. The maximum string length was 100. The pattern length ranged from 10 to 40 in increments of 10.

Figure 7-5 describes the situation when the total number of strings is 2000, the maximum string length is 500, and the pattern length varies from 10 to 40.

The third (Figure 7-6) and fourth (Figure 7-7) sets of data also had 2000 strings each and the maximum string lengths were 1000 and 2000, respectively. In these cases the pattern length was varied from 10 to 40 in increments of 10.

As is seen from these graphs, GSTBA is always faster than the SBA (both versions). Also note that as the database size increases, the difference between GSTBA and SBA are applied to such large databases, we can expect GSTBA to be two orders of magnitudes faster than the SBA. We should point out here that the preprocessing time for GSTBA was larger than that needed for SBA. But this is acceptable since preprocessing is done only once when the GST is constructed.

It is conceivable that the database changes on a regular basis with more and more sequences being added. In such cases, the GST also will have to be updated. But
fortunately, the update time is only proportional to the amount of new data that is being added.

7.6 Conclusions

In this Chapter, we have considered two different algorithms for the problem of identifying similar pairs of sequences in a given database. One was based on sorting (SBA) and the other was based on generalized suffix trees (GSTBA). Experimental data were obtained the GSTBA is faster than SBA. GST has the added advantages of linear space and linear update time. An interesting open problem is to extend the above algorithms for cases where the matches we are looking for can have gaps.

Figure 7-4 First data set
Figure 7-5 Second data set

Figure 7-6 Third data set
Figure 7-7 Fourth data set
CHAPTER 8
EXPERIMENTAL RESULTS

We implemented the algorithms discussed in the previous chapters. All the programs were written in C++. The experiments were performed on the SUN UltraTM 80 workstation which consists of four 450-MHz UltraSPARC II processors with 4-MB L2 cache. The total main memory is 4GB. The operating system is Solaris 8.

The synthetic datasets were created using the data generator programmed by IBM Group Guest. The synthetic datasets used in the first three experiments were

- \( D_1 = T_{26}I_{4}N_{1k}D_{10k} \)
- \( D_2 = T_{10}I_{4}N_{1k}D_{1000k} \)
- \( D_3 = T_{10}I_{4}N_{10k}D_{1000k} \)
- \( D_4 = T_{10}P_{4}N_{100k}D_{1000k} \)

The dataset \( T_{26}I_{4}N_{1k}D_{10k} \) means an average transaction size of 26, an average size of the maximal potentially frequent item-sets of 4, 1000 distinct items, and 10000 generated transactions. The number of patterns in all the three synthetic datasets was set to 10,000. The frequent item-set distributions are displayed in Appendix A.

8.1. Comparison of Apriori, Eclat, IT, FIT and SFIT

Among the five algorithms, the run time performances of Apriori, FIT and SFIT were highly dependent on the parameters that the programs choose to run. Our experimental results showed that the performance of Apriori was closely related to the degree of the hash tree that was used to store the item-sets and calculate the support values for item-sets. Let \( d \) represent the degree of the hash tree and \( p \) stand for the depth of the tree. For the given dataset, the sets of frequent item-sets are fixed. When the value of \( d \) became larger, the average number of item-sets in each leaf node would be smaller. Then the
time needed to search item-sets in each leaf node was lessened and thus the time to search item-sets in the hash tree was reduced. As a result, the program should run faster. On the other hand, since the number of inner nodes and the number of pointers in the tree are proportional to the value of $d^p$, the memory consumed by the program will also grow exponentially with the growth of the tree. So it might be the cases when the memory needed by the program exceeds the physical capacity of the main memory. In our experiments, we try to set the degree of the hash tree as large as possible, and, meanwhile, we also make sure that all the calculations could be performed in the memory.

As to FIT and SFIT, the performances are highly dependent on the sizes of the subgroups. On the other hand, the optimal values of the subgroup sizes are determined by the characteristics of the data in the datasets, such as whether the data in the dataset are sparse or not, etc. The determination of group sizes in SFIT is more complicated, since it needs to decide the sizes for different subgroups at multiple levels. In our experiments, for simplicity, we assigned the values to the subgroup sizes in both FIT and SFIT in advance. We did not guarantee that the subgroup sizes we used would generate the optimal experimental results.

The first set of experiments was performed on the dataset $D_1$. The size of $D_1$ was about 11.1M. Figure 8.1 shows the run time comparisons of Apriori, Eclat, IT, FIT and SFIT. The horizontal axis shows the values of the $\text{minsup}$. The vertical axis shows the values of the run times measured in seconds. In Figure 8.1, the run times of Apriori were recorded when the degree of the hash tree was set to 1,000 that equaled the total number of items in $I$. The size of the subgroups in FIT was set to 3. As to SFIT, we deployed
two levels of subgroups on the set $L_1$. The size of the first level subgroup was set to 20. The size of the second level subgroup was set to 3. For other set $L_k$, $k>1$, we used only one level of subgroup, and the corresponding size was set to 3. The reason that we only use one level of subgroup in the set $L_k$ was that the number of attribute lists in $L_k$ was usually comparatively small.

![Figure 8.1](image1)

![Figure 8-2](image2)
From Figure 8.1, it can be seen that IT, FIT and SFIT ran comparatively faster than Apriori and Eclat. Meanwhile, when the value of the minsup decreased from 200 to 10, the scalabilities of IT, FIT and SFIT were also better than those of Apriori and Eclat. Moreover, when the value of minsup was decreased to be smaller than 50, i.e. 0.5% of total transactions, the run time of Apriori tended to increase exponentially. When the minsup equaled 10, i.e. 0.1% of total transactions, the run time of Apriori, which was too large to be shown in Figure 8.1, was thousands of seconds.

The speedups of Eclat, IT, FIT, SFIT over Apriori, which were derived from Figure 8.1, are illustrated in Figure 8.2. To make a more clear look at the performance comparisons between Eclat and IT, FIT and SFIT, Figure 8.3 shows the speedups of IT, FIT and SFIT over Eclat.
To see how effectively IT reduced the total comparison times performed by Eclat in order to calculate the sets of frequent item-sets from D1, two sample results from the experiments in Figure 8.1 are shown in Figure 8.4 and Figure 8.5. Figure 8.4 shows the total comparison times of Eclat and IT when the \textit{minsup} was set to 1500. Similarly, Figure 8.5 shows the total comparison times of Eclat and IT when the \textit{minsup} was set to 500. In both Figures, the vertical axes represent the total number of comparison times that are displayed in millions. As shown in the first sample result, IT reduced about 71 percent of total number of comparison times performed by Eclat. In the second sample result, IT decreased the total number of comparison times performed by Eclat by 59 percent.

To verify that FIT and SFIT did reduce the total number of intersection operations computed by IT (note that IT has the same number of intersection operations as Eclat),

Figure 8.6 and Figure 8.7 illustrate two sample results from the experiments in Figure 8.1. In both Figures, the vertical axes show the total number of intersections measured in
thousands. Among the sample results, in the best case when the value of the \textit{minsup} was set to 200, i.e. of 2\% of total transactions, SFIT reduced 75 percent of total intersection operations performed by IT. In the worst case when the value of the \textit{minsup} was set to 20, i.e. 0.2\% of total transactions, SFIT reduced about 59 percent of total intersection operations calculated by IT.

The rest experiments were carried out on the datasets D2, D3, and D4, whose parameters except the number of items in I, |I|, were the same. There were 1,000 items in D2, 10,000 items in D3, and 100,000 items in D4. Generally speaking, when the number of items is increased and other parameters are maintained the same, the average length of the attribute lists should be shortened. We hope that by gradually increased the total number of distinct items in the datasets, we could compare the performances of the algorithms in the datasets with different densities.

![Figure 8.6](image1.png)

![Figure 8.7](image2.png)
The second set of experiments was carried out on D2. The size of D2 was about 41.1M. The degree of the hash tree in Apriori was set to 3,000. The size of the subgroup in FIT was set to 3. There were two levels of subgroups on the set \( L_1 \) when SFIT was utilized. The subgroup sizes were set to 15 and 3, respectively. When SFIT was used to calculate the intersections on other \( L_k \) here \( k>1 \), only one level of subgroup was selected. The size of the subgroup was set to 3.
Figure 8.8 shows the runtime comparisons. Figure 8.9 illustrates the corresponding speedups. Figure 8.10 shows the total number of comparison times performed by Eclat and IT. Figure 8.111 shows the total number of intersection operations calculated by IT, FIT, and SFIT.

In Figure 8.8, SFIT was still the fastest algorithm. IT and FIT were faster than Eclat and Apriori too. But Eclat was not always faster than Apriori. The performance of Eclat was better than Apriori only after the $\text{minsup}$ belows some value, which is 1500 in Figure 8.8. Our explanation to the above phenomenon is as follows. As discussed in previous chapters, the procedure of support calculations for the candidate frequent item-sets stored in the hash tree is very expensive in Apriori. So when the $\text{minsup}$ is small, say 1,000 in Figure 8.8, the hash tree contains a lot of item-sets. As a result, the run time performance of Apriori lags behind Eclat. But when the $\text{minsup}$ became larger, the number of (candidate) frequent item-sets in the hash tree was decreased. So the cost for the support calculations in Apriori was reduced correspondingly. On the other hand, as to Eclat, not all the attribute lists have the same effect on the final run time performance. The attribute lists that are longer would need more time to calculate the intersections between them and
other attribute lists. When the minsup was lessened, the attribute lists that have shorter length and less impacts on the computation performance were removed from $L_1$ first. The longer attribute lists that have larger impacts on the computation performance were still kept in $L_1$. As a result, the performance improvement of Eclat was not as significant as that of Apriori. So at some point when the minsup was below some value, say 1500 in Figuer 8.8, Eclat might run slower than Apriori.

The third set of experiments were carried out on D3. The size of D3 was about 51.4M. The run time experimental results are illustrated in Figure 8.12. The degree of the hash tree in Apriori was set to 3,000. To FIT, The size of subgroups in $L_1$ was set to 15. To SFIT, three level of subgroups in $L_1$ were selected. The sizes of subgroups in different levels were 120, 15 and 3, respectively. For other set $L_k$ here $k>1$, only one level of subgroup was selected. The size of the subgroup was set to 3.

Figure 8.12 shows the similar phenoena as revealed in Figure 8.1. The run time performances of Eclat, IT, FIT and SFIT are still much better than Apriori algorithm,
especially when the value of the minsup is below 1,500, i.e. 1.5% of total transactions in D3. The run time of the Apriori algorithm still tends to grow exponentially after the value of the minsup belows some threshold that is 1,000 in Figure 8.12.

Figure 8.13 shows the speedups of Eclat, IT, FIT, and SFIT over Apriori. The comparisons of the speedups of IT, FIT, and SFIT over Eclat are also shown in Figure 8.14. Both in Figure 8.13 and in Figure 8.14, the vertical axes stand for the speedup values. The horizontal axes represent the values of the minsup.
In the third set of experiments on D3, the performances of Eclat and IT were not always better than that of Apriori. For examples, IT was slower than Apriori when the value of minsup was no less than 1,500. Eclat was slower than Apriori when the value of minsup was no less than 1,000. But IT always ran faster than Eclat. On the other hand, FIT and SFIT always outperformed the other three algorithms, and SFIT was always better than FIT. Also in Figure 8.13, FIT and SFIT exhibited pretty good scalabilities. When the value of the minsup decreased below 1,500, Apriori’s run time was increased fast. So the performance gap between Apriori and FIT and SFIT became larger and larger, which is depicted in Figure 8.13. Figure 8.14 also illustrates that the speedup of IT over Eclat was almost constant with the value of the minsup changes.

The fourth set of experiments were performed on the dataset D4. The size of D4 is about 61.5M. All the parameters set for Apriori, FIT, and SFIT were the same as those used in the third set of experiments. The run time experimental results are illustrated in Figure 8.15. Figure 8.16 shows the speedup comparisons of Eclat, IT, FIT, and SFIT over Apriori. Figure 8.17 illustrates the speedup comparisons of IT, FIT, and SFIT over
Eclat. The total number of intersection operations performed by IT, FIT, and SFIT on D4 are shown in Figure 8.18.
The experimental results from D4 are similar to those from D3. We also made other experiments. The results were similar to the above figures. In summary, SFIT is always consistently the fastest algorithm in all the experiments. SFIT and FIT are always faster than Apriori and Eclat. Given a dataset, both SFIT and FIT demonstrated excellent scalabilities when the value of the \textit{minsup} is reduced. Although it is theoretically possible that SFIT and FIT may calculate more intersection operations than IT, it never happened in our experiments. Among the comparisons between IT, Eclat, and Apriori, IT also always outperformed Eclat. Both theoretical analysis and the practical results showed that IT did calculate the less total number of comparison times than Eclat. IT and
Eclat may be slower than Apriori, when the dataset is dense and the value of the \textit{minsup} is large. It also happened in rare case that FIT ran slower than IT even if FIT did reduce the total number of intersection operations performed by IT.

We also carried out on the experiments in some papers [54]. The experimenta results are as follows.
8.2. Experimental Results of \((1+\varepsilon)\) Pass Algorithm

We tested \((1+\varepsilon)\)-algorithm on the datasets D2. The dataset was divided into 50 partitions using the horizontal division method discussed in section 5.2. For each partition, we assigned 2 to the variable \(r\), which was the local \(\text{minsup}\) reduction. The partition interval was selected as 2. Also, whenever the calculation on any partition was finished, all of the results were written into the disk. If the results of the previous calculations were needed, the program read the data from the disk into the memory through I/O operations. Figure 8.24 illustrates the experimental results on D2. The vertical axis represents the value of \(\varepsilon\), which was the fraction of the data that were scanned twice. The horizontal axis stands for the values of the \(\text{minsup}\). The value of \(\varepsilon\) increases with the value of the \(\text{minsup}\) decreases. In our opinions, this phenomenon can be explained as follows. After first scan of the datasets, we collected sets of candidate frequent itemsets whose support values were not accumulated from the partition 1. Given an item-set \(X\), we used the symbol \(\text{current\_support}\) to stand for the accumulative support values for the \(X\) after the first scan, and denoted the partition that began to calculate the support for \(X\) as \(t\). According to the algorithm discussed in Chapter 7, if the expression of
(t-1)×(\text{minsup}/50-2)+\text{current\_support}\geq\text{minsup} was satisfied, then we should keep the item-set \(X\) and continued to calculate the support values for \(X\) during the second scan of the datasets. It can be seen that the smaller the value of the \text{minsup}, the easier the expression can be satisfied. In other words, we should scan more data twice to screen the false candidate frequent item-sets.

The run time results of \((1+\epsilon)\)-algorithm are shown in Figure 8.25. The experiments were also performed on datasets D2. Compared with IT, the run times of \((1+\epsilon)\)-algorithm
were much slower. Since the redundant operations and I/O operations in \((1+\varepsilon)\)-algorithm are more than those in IT, we believe the slower run time results from \((1+\varepsilon)\)-algorithm should be expected.

### 8.4. Experimental Results of Parallel Programs

#### 8.4.1 Multithread Experiments

We implemented the second multithread algorithm discussed in Chapter 5. The multithread program was running on a SUN workstation server that consists of eight 248-MHz processors with 4-MB L2 cache. The total main memory is 2GB. The total virtual memory is 4.5GB. The operating system is SUNOS 5.8.

The experiments were performed on D2. The run time results are shown in Figure 8.26. The corresponding speedups are shown in Figure 8.27.

![Figure 8.26](image-url)
Note that, in the above experiments, we used IT as the sequential program and we compared the run time performance of the multithread programs with IT. The experimental results show that most of the experimental results had very good speedups. We believe this is because the multithread programs have comparatively low costs for the communications. Moreover, when the value of the minsup is decreased, the number of frequent item-sets are increased. As a result, the programs exhibited better computation performances.

8.4.2 Parallel Processes

We implemented the parallel process program discussed in Chapter 5 by utilizing PVM [5]. PVM is short for Parallel Virtual Machine. It is a software package that permits a heterogeneous collection of UNIX and/or Windows computers hooked together by a network to be used as a single large parallel computer. The experiments were performed on a cluster of Sun workstations, each of which consists of one 248-MHz processor. The memory of each workstation is 320M. The operating system is SUNOS 5.8.
Like the multithread experiments, the parallel process programs were tested on the datasets D4. The run times are shown in Figure 8.28. The corresponding speedups are shown in Figure 8.29.
LIST OF REFERENCES


[33] J. Han, J. Pei, and Y. Yin. Mining Frequent Patterns without Candidate Generation. ACM SIGMOD International Conference on Management of Data, 1-12. ACM Press, Dallas, Texas, USA, 2000.


BIOGRAPHICAL SKETCH

In 1994, I graduated from Beijing Polytechnic University in China with my bachelor’s degree in computer science. Then I obtained my master’s degree in computer science from Tsinghua University, China in 1997. I expect to obtain my Ph.D. degree from the Computer and Information Sciences and Engineering Department at the University of Florida in 2002.