Algorithms for Multi-level Frequent Pattern Mining

Xi Zheng

A Thesis
in
The Department
of
Computer Science and Software Engineering

Presented in Partial Fullfilment of the Requirements
for the Degree of Master of Computer Science at
Concordia University
Montreal, Quebec, Canada

April 2008

Your file Votre référence
ISBN: 978-0-494-40957-2
Our file Notre référence
ISBN: 978-0-494-40957-2

ABSTRACT

Algorithms for Multi-level Frequent Pattern Mining

Xi Zheng

Data mining is a database paradigm that is used for the extraction of useful information from huge amounts of data. Amongst the functionality provided by data mining, frequent pattern mining (FPM) has become one of the most popular research areas. Methods such as Apriori and FP-growth have been shown to work efficiently in order to discover useful association rules. However, these methods are usually restricted to a single concept level. Since typical business databases support concept hierarchies that represent the relationships amongst different concept levels, we have to extend the focus to discover frequent patterns in multi-level environments. Unfortunately, not much attention has been paid to this research area. Simply applying the methods from single level frequent mining (SLFPM) several times in sequence does not necessarily work well in multi-level frequent pattern mining (MLFPM).

In this thesis, we present two novel algorithms that work efficiently to discover multi-level frequent patterns. Adopting either a top-down or bottom-up approach, our algorithms make great use of the existing fp-tree structure, instead of excessively scanning the raw dataset multiple times, as would be done with a naive implementation. In addition, we also introduce an algorithm to mine cross level frequent patterns. Experimental results have shown that our new algorithms maintain their performance advantage across a broad spectrum of test environments.

ACKNOWLEDGEMENTS

Thanks to Jing Zhou for her understand and support.

Thanks to Dr. Eavis and Dr. Grahne for their guidance.

Thanks to all of my friends for their helping hand.

Thanks to Concordia for providing this research opportunity.

To my family

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

In an era in which information technology plays an increasingly important role in developing and changing the appearance of the world, both individuals and companies encounter an enormous, and growing, amount of data every day. Data mining has become a major weapon that organizations can utilize to access this kind of data, process it and make decisions that produce positive effects on their future direction. Amongst the functionality that data mining can provide, frequent pattern mining (FPM) has emerged as a particularly popular research area. In short, FPM finds patterns that occur most frequently in a certain environment. Therefore it helps decision makers to improve their strategies in dealing with complex data environments.

Previous research has primarily focused on single level frequent pattern mining (SLFPM), presenting methods including the classic algorithms Apriori [AS94] and FP-growth [HPY00]. However, less attention has been paid to multi-level frequent pattern mining (MLFPM), which typically includes attribute hierarchies. For example, an item in a typical business data set not only has a single universal meaning, but it also represents alternative meanings at different levels in the whole concept hierarchy. Originally, researchers proposed the idea [HF95] of applying Apriori to the case of MLFPM. However, due to the excessive costs

for candidate generation of Apriori [HPY00], it may in fact be better to adopt the FP-growth mechanism to avoid generating so many candidates. That being said, simply applying FP-growth several times to search multi-level rules will cause other performance related problems.

In this thesis, we examine the advantages and disadvantages of the existing algorithms. We propose FPM-B and FPM-T to avoid multiple scans of the raw data set. In short, we reuse the existing tree structure to reduce the cost in constructing multiple level fp-trees. We also propose FPM-Cross to deal with the problem of cross-level frequent pattern mining (CLFPM).

1.1 Research overview

We consider the "benchmark" algorithm in this environment to consist of the iterative application of FP-growth. Briefly, FP-growth first scans the data set to collect frequent items at the bottom level of the concept hierarchy and builds an associated *header table*. Then it does a second scan to visit the data set in order to build the bottom level fp-tree. The mining process follows these two scans. From the second level to the uppermost level, the benchmark algorithm repeats the same procedure as was executed at the bottom level. After the whole process completes, we can identify the full result as the collection of the individual multi-level rules.

Observe that we will have to perform many scans of the data set; the resulting I/O cost can be huge. As well, this technique does not take advantage of the fact that the associated fp-trees are more and more compressed as we move up through the levels. Therefore, our new approach chooses to scan the individual fp-trees themselves rather than the data set. Although we still have to do the initial two scans at the bottom level, these are the only two

scans that we will do to the data set during the entire algorithm.

The visitation of trees can be executed either in a *bottom-up* or *top-down* fashion. FPM-B and FPM-T are the two new algorithms that work on the previous level fp-tree using the core idea. FPM-B first finds pointers to the leaves of the tree, scans the branches starting at the leaves, and proceeds all the way up until we reach the root node of the tree (Root). Meanwhile, it collects item name and count information, which it processes and then inserts into the *next level* fp-tree. When we finish processing all the branches, we get a whole new fp-tree for the next level. We can then perform mining methods on this new tree.

FPM-T works in the other direction. It starts from the Root of the previous level fp-tree and goes all the way down until it reaches the leaf nodes. When we meet a new node, we create a corresponding "next level" node. When we meet a *duplicated node* at the next level, we filter it out. Since the order of items at different levels is not the same, the new fp-tree we get does not maintain the prefix sharing feature. As a result, we may have to re-order the nodes and combine siblings with the same item name.

The advantage of FPM-B over FPM-T is that it can efficiently access the leaf nodes and does not need to resize the structure of the new fp-tree. When it finishes processing, the new fp-tree is in its final stage. The disadvantage is that it needs to scan the raw data set in order to create header tables for all levels, just as the benchmark does. On the other hand, FPM-T scans the fp-tree to generate header tables. This is faster, especially when the fp-tree has a compact size. When we go to higher levels, this is certainly the case. So this feature of FPM-T becomes an advantage over both FPM-B and the benchmark. That being said, FPM-T needs to reshape the tree; this is its primary disadvantage with respect to FPM-B. Ultimately, the characteristics of the two algorithms determine on which kinds of data they can achieve better performance. The details will be discussed later in this thesis.

In the domain of cross level frequent pattern mining (CLFPM), we also propose a new algorithm called FPM-Cross. Instead of performing another scan of the data set in order to create a cross-level fp-tree, we make use of the existing fp-tree, scan it, generate items from the *base level* to the *top level* and then get the cross-level fp-tree. We make greater use of "thresholds" here than in MLFPM in order not to let the branch length of the fp-tree grow too long. As a result, we are able to avoid the costs of scanning the raw data set, thus minimizing the time to construct a cross level fp-tree.

1.2 Experimental results

Experimental results have shown that FPM-T tends to be more effective than both FPM-B and the benchmark in the case when the average length of the transaction is small. This is primarily due to the fact that in this case, it does not need to spend much time resizing the tree. In other cases, FPM-B is superior to both FPM-T and the benchmark.

In the case with short transaction length, for example, when tested on data sets of 250,000 records with transactions of length 3, FPM-T was 18.4% faster than FPM-B and 43.6% faster than the benchmark in terms of total tree construction time. In contrast, when tested on data sets of 1 million records with transactions of length 10, FPM-B was 51.4% faster than the benchmark.

Scalability of the two new algorithms is also quite good. We have tested the algorithms and shown that they can maintain their advantage on data sets with varying sizes and level counts.

In addition, FPM-Cross also works very well compared with the benchmark. Specifically, the relative improvement grows with increasing level count, with a performance advantage of up to 90% with five level concept hierarchies.

1.3 Thesis organization

The thesis is organized as follows. Chapter 2 provides a brief overview of Data Mining. It also introduces the primary architectures and functionality in the field. The remainder of the chapter reviews the background material necessary to understand the core themes of the thesis. It presents the concepts, classification, and work directly related to frequent pattern mining. Specifically, it reviews the most important algorithms in single level FPM, such as Apriori and FP-growth.

The subsequent chapters present the main contributions of this thesis. Chapter 3 first reviews the issues associated with multi-level FPM. Then it describes the two new algorithms, FPM-B and FPM-T, in detail. In addition, it also introduces FPM-Cross, a mechanism used to generate cross level rules. Chapter 4 presents the experimental results from a variety of different perspectives. Chapter 5 then provides the final conclusions and points to possible future work.

Chapter 2

Background Material

2.1 Introduction

With its explosive development, information technology has greatly changed the nature of our society. Directly or indirectly, we receive advertisements, political news, science, entertainment and material from a variety of other domains every day. It would surely look like we have more than enough data in our lives. Nevertheless, being rich in data does not mean being rich in information. Instead, people have to extract useful information from these massive data sources so that it can be utilized efficiently and effectively. Data mining, "the nontrivial extraction of implicit, previously unknown, and potentially useful information from data" [FPSM92], and "the science of extracting useful information from large data sets or databases" [HMS01], typically deals with this very problem.

In this chapter, we will review the concepts, architectures, and core functionality of data mining. Specifically, we will provide a detailed examination of the problem of frequent pattern mining.

2.2 Decision Support Systems

A Decision Support System or DSS is a computer-based information architecture that assists business managers or individuals to make better choices [Fin94]. In practice, it offers a collection of applications that provide an easy-to-use interface, and empowers users to retrieve, summarize and analyze relevant data in order to make smart decisions. These applications differ in their implementations, functionality, and analytical basis. Below, we will introduce the three primary DSS models, including Data Mining, which is the focus of our current research [Eav03].

- **Information Processing**. The main goal here is to gather basic information. We define queries, either ad-hoc or pre-defined, to extract detailed information from database management systems (DBMS). Relatively simple analysis will be provided here, such as sorting and basic aggregation. Consequently, we will only discover the most obvious features of the underlying data.

- **Online Analytical Processing or OLAP**. OLAP extends the basic capabilities offered by Information Processing systems. By providing functions such as drill-down, roll-up and pivot, as well as additional operations for such things as ranking, moving averages, growth rates, statistical analysis, and "what if" scenarios, OLAP allows a robust multidimensional analysis of the raw database.

- **Data Mining**. The objective of data mining is to provide a more comprehensive form of knowledge discovery. Unlike OLAP, where we essentially seek to prove the correctness of a priori assumptions by users, data mining tends to automate the discovery process to uncover patterns or rules with minimal user input. In this sense, we can say

that OLAP is a user-driven process while data mining can be considered a data-driven process. Typical data mining operations include frequent patterns, associations and correlations, classification, prediction and cluster analysis.

2.3 Data mining

2.3.1 Definition of data mining

While there are many competing definitions available, we can say that, data mining is a technology that processes raw data from different perspectives and extracts useful information such as patterns and relationships in order to make valid predictions [Ede05].

2.3.2 Architecture of data mining

As shown in Figure 2.1 [HK06], we can divide a typical data mining system into the following seven parts:

1. The *database, data warehouse, world wide web (WWW) and other data repositories* refer to the location or source where we can retrieve data.

2. *Data cleaning, integration and selection* are the processes by which we detect if the data is corrupt or inaccurate so that we can perform certain operations to make it amenable to the subsequent processes.

3. The *database or data warehouse server* provides the functionality required to to manage the data obtained from the original data sources.

4. The *knowledge base*, as the name itself indicates, provides knowledge that determines whether the given data or information is useful and thus a candidate for final processing. That knowledge can be formulated as rules, patterns or thresholds.

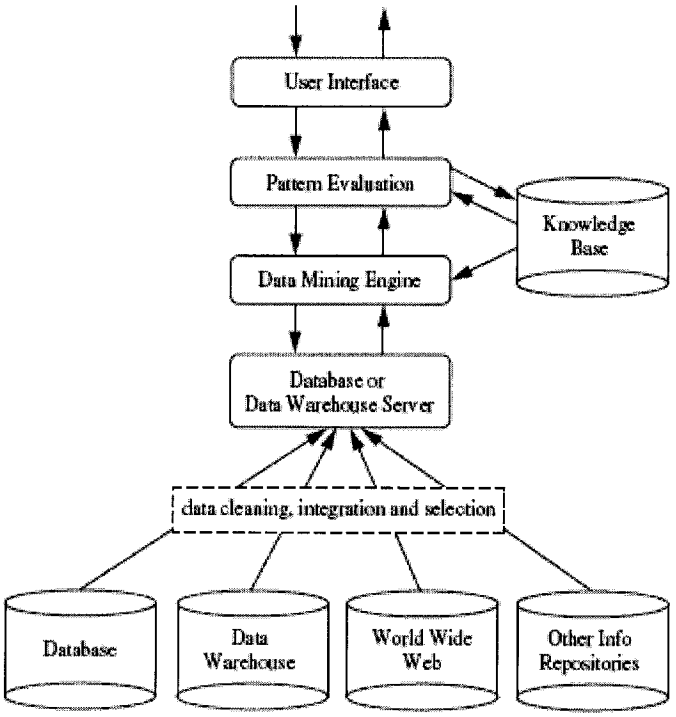

Figure 2.1: Architecture of a data mining system

5. The *data mining engine* is the central component of the architecture and has functionality for data mining operations such as the identification of frequent patterns, associations and correlations, classification and prediction, and cluster analysis.

6. *Pattern evaluation* utilizes user-defined measures that tell the system whether the patterns gathered from the data mining engine are worth considering. Typically, it uses "interestingness" thresholds to filter out these patterns.

7. Finally, the *user interface* is the place when the end users interact with the system.

2.3.3 Data mining Functionality

The various data mining algorithms allow us to explore data from a number of different perspectives. Major data mining forms are listed below.

- **Mining Frequent Patterns, Associations and Correlations**. Mining frequent patterns is an attempt to identify patterns that occur most often in certain circumstances. By adopting this perspective, we discover associations and correlations within a wide variety of underlying data sources. This process also helps decision makers improve their business strategies, such as those related to the psychology of customer shopping and the arrangement of shopping districts.

- **Classification and Prediction**. Classification in data mining is a technology for discovering models that can be used to distinguish data classes. Common classification methodologies are *decision trees* and *neural networks*. Prediction in data mining is a process that predicts missing or unavailable numeric data values in the data source. A popular methodology is regression analysis. The main difference between these two approaches is that classification identifies categorical class labels while prediction deals with continuous-valued functions.

- **Cluster Analysis**. Cluster Analysis in data mining is the process that divides a large amount of data into sub-classes, called clusters. After clustering, a number of similar classes of data — based on their attribute values — are grouped together. The primary difference between cluster analysis and classification is that the former is an automated process in which users do not know the label of each object.

2.4 Mining frequent patterns

Below we identify the core concepts required for a basic understanding of the frequent mining domain.

2.4.1 Concepts

- **Premise:** In order to study frequent pattern mining (FPM), we assume that the database contains (1) an item set I, which includes all of the items $I_i \in I$ in the database where $i = 1, 2, \ldots n$, with n being the number of distinct items and (2) a set of transactions $T_j\{I_p.....I_q\}$, where T_j is a transaction identifier and each transaction is made up of the items from I. This set of transactions is called a data set [HF95].

- **Pattern:** A *pattern* refers to a set of items, subsequences or substructures. In our specific environment, it means a set of items only.

  Definition 1. *A pattern is an item or any combination of a set of items, such as* $I_i \bigwedge ... \bigwedge I_j$ *where* I_i,...,$I_j \in I$.

  Patterns are sometimes called *itemsets* in out research.

  Definition 2. *A k-itemset means an itemset that has k items.*

- **Support:** The word *support* in FPM means the probability that a certain item occurs in the transactions of the data set. For example, given a shopping cart data set that has a total of 100 customer records in which 10 customers bought milk, the support of milk in this data set is 10%.

  Definition 3. *The support of a pattern* $A : S(A \rightarrow T)$ *is the occurrence (number) of transactions containing A versus the number of total transactions T. The support of a pattern A equals the conditional probability of A in T,* $P(A \cup T)$. *We can define* $S(A \rightarrow T) = P(A \cup T)$.

  As every item will have its own specific support in the data set, in research environment we typically use a user-defined minimal support count, called mim_{sup}. This support can also serve as a threshold that we will discuss later.

- **Confidence:** The term *confidence* in FPM means the probability that a certain item occurs when another item also occurs. For example, a confidence of 50% for milk to

jam means that if people buy milk, there is a 50% chance that they buy milk and jam together.

Definition 4. *The confidence of a pattern A to B is the conditional probability that a transaction having A also has B. Similarly, we can denote* $C(A \rightarrow B) = S((A \cup B) \rightarrow T) / S(A \rightarrow T)$

- **Threshold:** In practice, since the total number of transactions will be fixed, we will use *threshold* instead of support. By definition, the threshold TH of a level is the specific number of relevant transactions in the data set. For example, if the threshold of level 1 is 10, and the number of total transactions is 100, any item that has a support of less than 10% will be considered as *infrequent*.

2.4.2 Classification

On the basis of the kind of patterns to be mined, the domain of frequent pattern mining can be divided into several sub-categories. The original focus by Agrawal et al. in [AIS93] was on *boolean association rules.* In fact, this is also true of many of the subsequent algorithms. In each transaction of the data set, items are identified based only upon whether they appear of not; in other words, they are assigned a 1 or 0. Other researchers also worked on *quantitative association rules.* These rules are associated with numeric attributes or characters. For example, if we want to find the shopping habits of people ranging in age from 20 to 29 and with salary ranging from 30k to 50k, we need to concern ourselves with these numeric values. In our research, we work on boolean rules exclusively.

For general data sources from the business or research world, data sets are typically stored in *horizontal* format, which means we employ a *transaction ID* (TID) to identify each transaction that, in turn, includes detailed item information. Other researchers have also introduced a new model called *vertical* format, where each item is represented by a TID-list.

In our research, we will only focus on data sets stored in horizontal format.

Based on the level of abstraction of the data in the data set, we can mine *single level* frequent patterns as well as *multi-level* frequent patterns. Our research, of course, focuses on the second problem. In addition, if we classify the number of dimensions in each item of the data set, we can mine *single dimension* frequent patterns and *multi-dimensional* frequent patterns. Normally, we encounter data sets that have a relatively small number of levels or dimensions. However, if this is not the case, we have to handle *high-dimensional* or *colossal patterns.* A good example of this situation can be found in the data sets from bio-informatics. For instance, in a gene sample, the associated data sets do not have many horizontal lines (like transaction lines in the business world), but each line will have a huge number of columns. Therefore, we have to provide new methods that are different from those we could apply to business data sets.

If we want to mine the complete set of patterns that are discovered, then ultimately we are mining a pattern set called *all frequent itemsets* (FIs). Alternatively, we might only mine *closed frequent itemsets* (CFIs), as well as *maximum frequent itemsets* (MFIs). Other patterns can also be important in certain environments. Examples would be *sequential patterns* and *structured patterns.* Sequential patterns occur when dealing with data sets in which the order of each item has implicit meaning. Common examples would be shopping sequences in retail stores or click streams in a web site. Structured patterns represent even more complex patterns such as graphs, trees and lattices.

Beyond basic patterns, other researchers have explored more interesting or unusual rules. Examples of these alternative rules or patterns include *constraint based mining* and *approximate frequent itemset mining.* Here, users can define varying numbers of complex constraints in order to filter useless patterns and ultimately get the desired ones. In order to reduce

the potentially huge set of generated patterns, some researchers have attempted to mine frequent patterns in an *approximate* set. By definition, this kind of set is more compressed.

Related work

In the domain of quantitative association rules, Srikant et al. first investigated this problem in 1996 [SA96a]. They divided values of attributes into partitions and introduced the idea of *partial completeness*, specifically dealing with the information lost because of partitioning. Miller et al. [MY97] worked on distance-based quality and rule interest measures in order to find quantitative rules. Aumann et al. adopted statistical theory for this same problem [AL99]. Typically, their guiding principle was to use the distribution of numeric values. Moreover, Zhang et al. also worked on statistical theory to search for quantitative patterns. Specifically, they introduced a re-sampling technique as the basis of their approach [ZPT04].

In the context of mining data sets using the vertical format, Zaki presented an algorithm called Equivalence Class Transformation (Eclat) [Zak00]. The author uses an idea similar to Apriori, generating $(k + 1)$-large itemsets from k large itemsets. The author also scans the data set to build a Transaction ID set (TID-set) for each single item. In this case, the algorithm does not need to scan the data set several times to get the count information of each item.

Pan et al. proposed CARPENTER to deal with high-dimensional bio-informatics data sets [PCT+03]. Here, the idea is to transform these kinds of data sets into vertical format so that we can treat the TID-set as a row set that, in turn, can be used to build an fp-tree for the data set. CARPENTER does depth-first, row-wise enumeration. Zhu et al. presented an algorithm called Pattern-Fusion to work with colossal patterns [ZYH+07]. The basic idea here is to fuse the small core patterns of the colossal patterns in one step. With

this approach, they reduce the cost of generating mid-sized patterns, which is the general technique adopted by Apriori and FP-growth.

Generalized Sequential Patterns (GSP), an early algorithm in mining sequential patterns, presented by Srikant et al., also uses an Apriori-like approach [SA96b]. PrefixSpan, proposed by Pei et al., employs a divide and conquer technique, treating each sequential pattern as a prefix and getting all patterns from the divided partitions [PHMA+04].

Grahne et al. studied the problem of mining monotonic constraint based frequent itemsets [GLW00]. They designed algorithms to compute two kinds of semantics to answer correlated constraint queries. Specifically, they identify all minimal answers that are valid and all minimal valid answers. Pei et al. also proposed algorithms to discover convertible constraint based frequent itemsets [PHL01], in this case by exploiting the FP-growth algorithm. Finally, Liu et al. presented a noise-tolerant threshold for mining approximate frequent itemsets [LPS+06]. They also use rules to prune itemsets if their sub-itemsets are not frequent.

2.4.3 Fundamental Algorithms

Since the problem of multi-level mining is closely associated with single level mining, it is important to briefly review the relevant algorithms in the literature. In our research, we are motivated specifically by the classic algorithms in FPM such as Apriori and FP-growth.

Apriori: Candidate Generation

The Apriori algorithm was originally presented by Agrawal et al. in 1994 [AS94]. The basic idea is to first scan the data set and then use thresholds to filter out useless, single items in order to identify frequent items called *large 1-itemsets*. We then use those 1-itemsets to generate new, potential large itemsets called *candidates* for 2 items. We do filtering again

and generate still more candidates for other items. Eventually, we will obtain all frequent patterns.

Before discussing the output of the Apriori algorithm, we first note the following. In business environments such as those of retail companies like Wal-Mart, a typical shopping cart could include transactions like buying milk, jam, juice and bread. At this level of detail, it is not efficient to mine directly on the native attribute values. Instead, we would like to encode every item at a higher level of abstraction. For example, if we assign "A" to milk, "B" to jam, "C" to juice, and "D" to bread, then the above transaction could be encoded as A, B, C, D. For the sake of simplicity, we will continue to use this kind of notation throughout the thesis.

The motivation for candidate generation is illustrated in Figure 2.2 [BMUT97]. We observe that for frequent 1-itemsets, valid results can only come from the four single items A, B, C, and D. Similarly, frequent 2-itemsets can only be acquired from the derivation of the above four single items, which results in AB, AC, AD, BC, BD, CD. The same logic applies to frequent 3-itemsets and 4-itemsets. For 2-itemsets, therefore, we can use 1-itemsets for generation. Similarly, any (k+1)-itemset can be obtained from a k-itemset in a similar fashion. Ultimately, we will generate all relevant results and then filter out some of these as required.

Definition 5. Large *itemsets are those having counts with minimum support. Relatively speaking,* small *itemsets have counts below minimum support. Furthermore, large k-itemsets are those having k items in the pattern.*

The Apriori algorithm begins with a scan of the raw data set to collect all large 1-itemsets. Next, from 2 to the largest k-itemsets (i.e., itemsets that have the largest possible number of items), we generate (k+1)-itemsets based upon the associated k-itemsets. In addition, after generation of every k-itemset, we perform a scan to check if any candidate has a count that

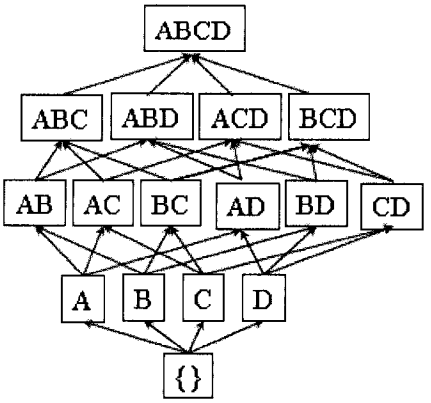

Figure 2.2: A lattice

is above the threshold or minimum support that we have set. Otherwise, the itemsets will be removed. The algorithm terminates if there are no itemsets that can be generated for the next round.

Apriori also utilizes a pruning principle: all nonempty subsets of frequent itemsets must be frequent. This means that if we find any infrequent itemsets, we do not need to generate the supersets of those candidates. This property is important in that we can avoid generating the non-useful candidates, thereby reducing the total size of the result set.

The Apriori algorithm is depicted in Algorithm 1. The procedure *apriori-gen()* generates candidates C_k from large (k-1)-itemsets L_{k-1} for large k-itemsets L_k. In short, it includes two phases, *joining* and *pruning*. The joining phase requires that we join L_{k-1} with itself. In the case where the first (k-2) items of the members of L_{k-1} are the same, then those two members are join-able. The pruning phase requires that we apply the principle discussed earlier. Specifically, we check every member of C_k at the time it is created. If any of them has a subset that is not frequent in L_{k-1}, then it is filtered out. The procedure *subset()* in Algorithm 1 divides the candidate set resulting from the procedure *apriori-gen()* into candidate sets for each transaction. Thus, we can use it to get the count of each member of

C_k if it occurs in that transaction. In practice, we will increment these counts by one.

Algorithm 1 The Apriori algorithm

Input: a transaction data set, minimum support(threshold)
Output: d-itemsets that are frequent.

```
1: for item number k = 2 to item number d do
2:   while L_{k-1} is not empty do
3:     C_k = apriori-gen(L_{k-1}) {a set for candidates}
4:     for each transaction t in the data set do
5:       C_t=subset(C_k,t) {find candidates occurring in this transaction}
6:       for all the candidates in C_t do
7:         update its count information
8:       end for
9:     end for
10:    do filtering based on the minimum support
11:    get result set as L_k of the large k-itemsets
12:  end while
13: end for
14: L=⋃_k L_k
```

While the idea of Apriori is interesting, it does have significant shortcomings. In particular, counting the support of the candidates is costly. This is because (i) the total number of candidates can be huge, and (ii) there may be a lot of items found in each transaction. In order to do all of the counting, we need to perform many scans of the transaction data set, a process that can be quite time consuming.

Some research has attempted to improve the performance of Apriori. Park et al. presented an idea based on a hashing technique to reduce the total number of candidates [PCY95]. In addition, they generated smaller candidate sets at early stages. For example, the number of candidates for 2-itemsets they generated is much smaller than previous methods. In so doing, they were able to decrease the transaction database size early in the process, which allowed them to reduce total costs. Savasere et al. worked on the idea of *partitioning* [SON95]. They pointed out that if an itemset is frequent in the whole data set, then it

should be frequent as well in at least one partition of the data set. Using this logic, they were able to scan the raw data set only twice. Contribution by Toivonen built upon the concept of sampling [Toi96]. The idea here is to first choose a random sample of the data set, performing Apriori to mine frequent patterns. Then, the algorithm does the second scan to find possible missing patterns. This approach is beneficial when efficiency is the primary concern.

FP-growth: Non-candidate Generation

While efforts were made to follow up the success of the classic Apriori algorithm, these improvements were done within the framework of candidate generation. The bottlenecks remain the costly data set scanning and possibly huge number of candidates generated. To avoid such limitations, Han et al. [HPY00] proposed a new data structure and mining method called FP-growth.

FP-growth-whole Algorithm

The basic idea of this method can be found in Algorithm 2. Essentially, it consists of two functional parts. The first one is *fp-tree* construction, which includes two scans of the data set (shown in line 1-2). The second phase is the mining process, called FP-growth (shown in line 3).

Algorithm 2 The algorithm for FP-growth-whole

Input: a transaction data set, minimum support(threshold)
Output: frequent patterns.

1: call Scan1-dataset() {scan the data set once to get the count of frequent items and insert into the header table}
2: call Scan2-dataset {scan the data set a second time to create the fp-tree}
3: call FP-growth() {mine the fp-tree}

TID	Items
1	A, C, B, M, H, G, E, F
2	C, D,B, A, l, O
3	D, A, I, J, O, W
4	D, B, K, S, F, E
5	C, A, B, L, l, F, E, N

Table 2.1: A transaction table

FP-tree

Definition 6. *A Frequent Pattern Tree is a tree structure that has the following properties:*
1. It consists of a root node, labeled Root, prefix-sharing branches serving as children of the root, and a header table.
2. Every node in a prefix-sharing branch has at least three fields: item name, count, and a link to other nodes that have the same item name as the current node, called link-node.
3. The entries in the header table have at least two fields: item name and the head of the link-nodes discussed in 2. Entries may have other field like order of the item.

FP-growth defines a new data structure called the *Frequent Pattern Tree* (FP-tree). Like a normal n-ary tree, every node in this tree will have several links such as those to its parent, siblings and children. In addition, it also has fields such as name, count and the *link-node*. The link-node ensures that nodes having the same item name in the tree are also linked together, as well as to an entry in a structure called the *header table*. These entries have the count and order information of each frequent item in the data set. The order is based on the frequency of each item. In every branch of the fp-tree, nodes follow the order from most frequent to least frequent and make use of the idea of *prefix sharing*. That is, we combine nodes with the same item name together. A sample tree is shown in Figure 2.3. The corresponding encoded data set is shown in Table 2.1.

Scan1-dataset()

Here, we first perform a scan of the encoded data set. For every transaction line, we read it from the disk and get all individual items. These items are identical to those called 1-itemsets in Apriori. The items are then inserted into the header table and their count information is

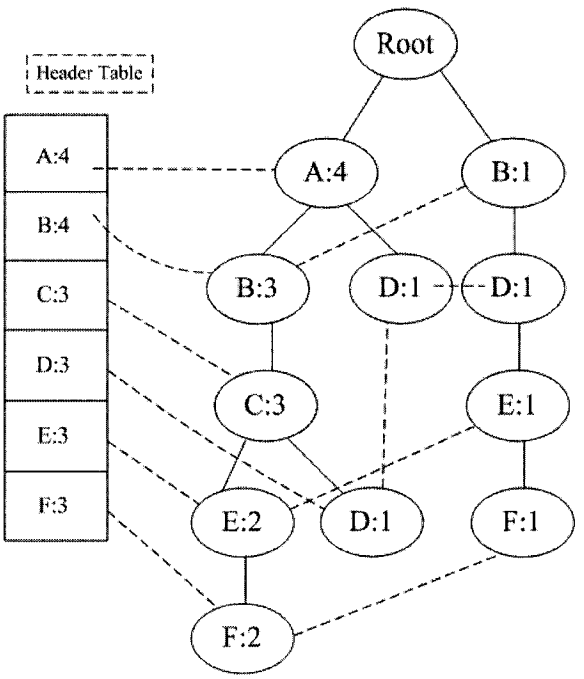

Figure 2.3: A sample fp-tree.

updated. When the iteration finishes, the header table will contain all single items. Based on the minimum support or threshold, we now do filtering to remove infrequent items. This set of frequent items F is the same as the large 1-itemsets in Apriori. We then do an ordering based on the frequency and re-sort the items as appropriate.

Scan2-dataset()

This phase involves the process of fp-tree building. We perform another scan of the data set. As we do this, we create a root node and label it as Root, with null item name and zero item count. With each transaction, we first obtain the set of all frequent items in the transaction, sorting it according to the order in the header table. We refer to this set as I_T. We then call a function named $insert\_tree(I_T, T)$ to start tree-construction. I_T from the first transaction will simply be inserted as a new child branch of Root, its items listed in descending order of their frequency. For all other transactions, let f be the first item in I_T. Now, if the tree T

TID	Items
1	A, B, C, E, F
2	A, B, C, D
3	A, D
4	B, D, E, F
5	A, B, C, E, F

Table 2.2: A filtered and sorted transaction table

has a node N where N.item-name=f.item-name, we simply increase N's item count by 1. If not, we will create a new node, initializing its fields by setting its count to be 1 and linking it to its parent and siblings, as well as its link-node. The same logic applies to the remaining items of I_T; we call $insert\_tree(I_T, T)$ recursively.

An example for the above two functions

Recall the sample data set in Table 2.1. Here, we set minimum support (threshold) to 3. After running Scan1-dataset(), we get a header table that has frequent items such as A, B, C, D, E, and F, with their corresponding counts: 4, 4, 3, 3, 3, 3. This header table can be found in the left part of Figure 2.3. Note that infrequent items are eliminated. The filtered data set is shown in Table 2.2, with items appropriately re-sorted. Please note that the filtered data set does not physically exist; it is for viewing purposes only.

Next we create a root node and label it as "Root". Then, for the first filtered and sorted transaction in Table 2.2, we insert every item in turn. Therefore, we have nodes linked in the first branch as shown in Figure 2.4. For the second branch, when we process the first item "A", we find there is already an "A" in the tree; thus, we only update the count of this node. The same thing happens with B and C as well. After finishing scanning this transaction, we have the two branches of the tree shown in Figure 2.5. As we go through all the transactions, nodes with the same item name in the tree will also be connected. As already illustrated in Figure 2.3, the nodes "B:3" and "B:1", "E:2" and "E:1", "F:2" and

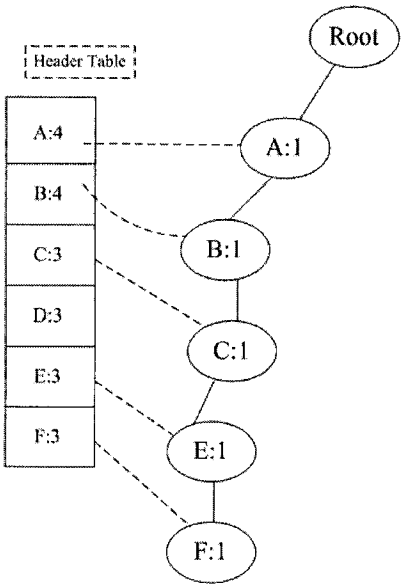

Figure 2.4: First branch

"F:1", as well as three "D:1"s, are connected to each other.

Completeness of the FP-tree

We use the fp-tree to represent the entire encoded data set so as to avoid multiple scans of the data. Of course, we must ensure that the information presented by the tree does not lose any meaning in the process. We note that every branch is in fact created from the data scanned from each and every transaction. In addition, the fp-tree does not break the long patterns of any transaction. Therefore, the full fp-tree preserves the complete information contained in the raw data set.

Compactness of the FP-tree

Another advantage over simple candidate generation is the compactness of the FP-tree. Nodes are inserted in descending order of frequency, allowing us to utilize prefix sharing as much as possible. Also, infrequent items are filtered out, ensuring that the tree is never bigger than the raw data set.

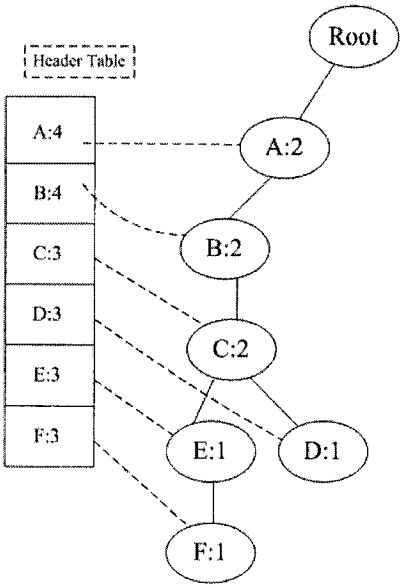

Figure 2.5: Second branch

FP-growth()

After we build up the fp-tree structure, we move on to the mining of frequent patterns. The algorithm is described in Algorithm 3. Before continuing, however, let us first look at an important principle: prefix sharing in pattern generation. To generate patterns of a suffix I_k, only the prefix sub-paths of nodes with item I_k are considered. In practice, only parents and ancestors of this node need to be processed; therefore, the count of the prefix sub-paths in a pattern is equal to the count of the suffix node I_k. Take branch 1 in Figure 2.3 for instance. If we want to generate patterns for item C, we start from the header table and find only the node "C:3". At this point, we only need to process the sub-path "A:4-B:3-C:3". A resulting pattern is "A-B-C:3". The count 3 is obtained from the count of the node "C:3".

In the case of a single path tree SP

This simple case allows opportunities for optimization. When the resulting fp-tree has a single path or when some parts of the fp-tree go straight from the root node to the end

Algorithm 3 The FP-growth Algorithm

Input: an fp-tree generated using the above steps and representing the transaction data set, a minimum support(threshold)
Output: frequent patterns.

```
1: if the tree contains a single path SP then
2:    generate frequent patterns of SP: fp(SP)
3: else
4:    for each item I_k in the corresponding header table of this tree do
5:       starting from least frequent to most frequent {the tree contains multiple path MP}
6:       generate I_k's pattern: p(I_k) with the minimum support
7:       generate p(I_k)'s conditional pattern base
8:       construct p(I_k)'s conditional pattern tree
9:       call fp-growth()
10:      let fp(MP) be the frequent pattern of MP
11:   end for
12:   return fp(SP)⨆ fp(MP) ⨆ (fp(MP)X fp(SP))
13: end if
```

node, we can simplify the mining process. By enumerating node combinations in the prefix sub-path having the appropriate item support, we quickly get the frequent patterns of nodes in such a tree. For example, in Figure 2.3, assume we again we want to generate all frequent patterns of the node "C:3". Since the subtree "A:3-B:3-C:3" is a single path tree, we can generate the rules of "C" directly by enumeration, producing "A-B-C:3", "A-C:3" and so on.

In the case of a multiple path tree MP

In most cases, fp-trees will have multiple paths. Since the nodes in the tree are sorted in descending order, and share prefixes as much as possible, we begin from the most infrequent item in the header table. By iterating from least to most frequent, we are guaranteed not to get duplicated patterns. For all nodes of each item, we traverse them by following the node-link that connects nodes with the same name. Using the example of Figure 2.3, let us take the item "F" for instance. We begin with the entry in the header table, and find two nodes labeled with the name "F", "F:2" and "F:1". For the first node, "F:2", we visit

the branch moving upwards and find four item names, "E,C,B,A", with the count of this node equivalent to 2. For the second branch, we follow the same process and find another three item names "E, D, B", with a count of 1. We refer to this process as *constructing the conditional pattern base*(CPB) of the item "F". The end result is that we get only two additional items along with the original item, from most frequent to least. Specifically, we get "B:3" and "E:3". The other items are filtered out. This filtering is based on the minimum support (threshold) that we set previously.

We now sort the items and generate a header table for the *conditional pattern tree* (CPT) of "F" in the next step. This process of constructing the CPT is similar to the one for building an fp-tree from the data set (Scan2-dataset()), as discussed earlier, except that now it scans the fp-tree in this case. To begin, we initialize the root node of this new tree as Root. By following the pointers of "F", we create just one branch, "B:3-E:3". This is illustrated in Figure 2.6. Observe that this is a single path tree, so we can get frequent patterns by enumeration: "F,E,B:3", "F,B:3", and "F,E:3".

Since the threshold of the data set (3) is relatively high for an illustration, we will "artificially" lower it to 1 just for demonstration purposes. Again, we still focus on the item "F". This time, if we set the threshold to 1, the result of constructing the CPB is that we get five items along with "F", from most frequent to least. These are "B:3", "E:3", "A:2", "C:2" and "D:1". We sort them and create the header table for the CPT of "F". The associated CPT is shown in Figure 2.7. By recursively applying Algorithm 3, we start mining this subtree. First, we check if it is a single path tree. The answer is no. Therefore, we use the guidelines of the multiple path tree. We start from the least frequent item "D" in its header table, and get the CPB of pattern "F,D". Then we construct a CPT for this pattern, as shown in Figure 2.8. Since this CPT is a single path tree, we can generate its patterns by

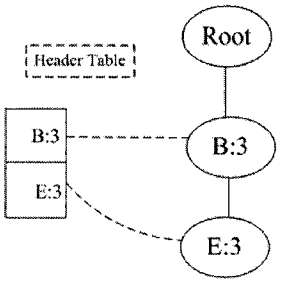

Figure 2.6: Conditional fp-tree for "F", using support = 3

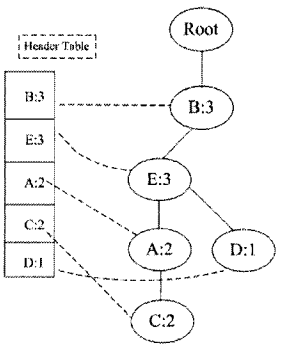

Figure 2.7: Conditional fp-tree for "F", using support = 1

enumeration: "F-D-E-B:1", "F-D-B:1" and "F-D-E:1". Next, we process the node of "C". Following the same steps, we get the CPB (nodes) of "F,C" as "A,E,B" and the CPT as shown in Figure 2.9. Since it is also a single path tree, we list its patterns as "F-C-A-B-E:2", "F-C-A-E:2", "F-C-A-B:2", etc. The nodes remaining in the header table for the CPT of "F", using support of 1, can be processed in a similar way. Generally speaking, in the multiple path situation, we recursively apply the FP-growth algorithm. If we find that any subtree is a single path tree, we output by enumeration; otherwise, we continue doing the recursion.

Closed frequent itemsets and Maximum frequent itemsets

Finally, we note that the concepts of *closed frequent itemsets* (CFI) and *maximum frequent itemsets* (MFI) are also important related research topics in the domain of frequent pattern mining. Nevertheless, they are somewhat outside the scope of the current thesis. The reader

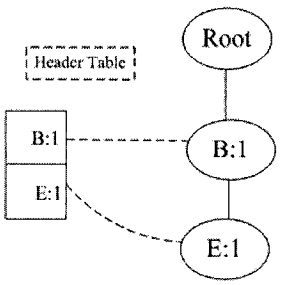

Figure 2.8: Conditional fp-tree for "F,D"

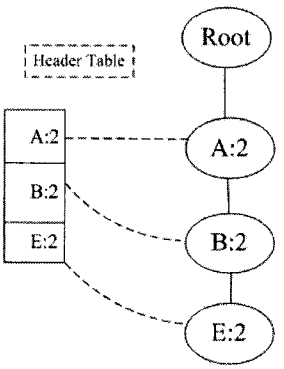

Figure 2.9: Conditional fp-tree for "F,C"

may refer to Appendix A for a more thorough discussion of these two topics.

2.5 Conclusion

In this chapter, we have introduced the concept of Decision Support Systems (DSS), as well as that of data mining, one of its most important components. We examined the architecture and functionality of data mining, of which frequent pattern mining (FPM) is the basis of our research. We then discussed classic algorithms such as Apriori and FP-growth for single level frequent pattern mining (SLFPM). It is important to fully understand these methods as our new approaches for multi-level frequent pattern mining (MLFPM) are motivated by SLFPM techniques.

Chapter 3

Algorithms for multi-level frequent pattern mining

3.1 Introduction

It has not been long since the idea of frequent pattern mining (FPM) was first proposed [AIS93]. However, numerous related methods have been shown to work effectively, with some being particularly efficient ([AS94], [PCY95], [SON95], [CHNW96], [HPY00], [GZ01], [JH07]). As a result, not only are these techniques providing good results in their original target areas, they are also very helpful in general data analysis, task handling, and other research [KHC97]. In short, FPM has become an essential tool in the data mining process.

One of the most important current research problems of FPM is associated with the fact that most of the previous research focused on single level frequent pattern mining (SLFPM). The basic algorithms, Apriori [AS94] and FP-growth [HPY00] do, in fact, work well on a single level. Therefore, researchers tended to apply them to multi-level frequent pattern mining (MLFPM) as well. Typically, they would generate multiple level pattern rules by applying these algorithms several times in sequence. For instance, in Apriori, one may need to generate subsequent candidates from the lowest level to the highest level and then test those candidates against the data set many times. In FP-growth, one may need to scan the data set several times to generate fp-trees for each single level. This is not a problem for

these methods if the source data set is relatively small. However, since for each level we will need to perform a scan of the full data set, the I/O cost increases significantly if the data set is large. In FP-growth, for each level we must scan each transaction line of the data set, generate items for this level one by one, and insert them into the fp-tree for this level. In Apriori, the situation is even worse, since at a single level the cost for testing candidates against the data set is quite large. Therefore, if we apply these algorithms directly to the multi-level case, then as the size of the data set goes up, the cost for scanning the data set eventually becomes unacceptable. In turn, the overall running time for MLFPM increases significantly.

In this chapter, we propose two new algorithms that deal with the problem described above. In general, both focus on the idea of minimizing the time required for scanning the data set. We work on existing fp-trees, discovering the inherent relationships amongst the trees, and then generate new fp-trees based on existing trees. Since our methods extend FP-growth, we call them FPM-B and FPM-T respectively. FPM-B adopts a *bottom-up* approach, scanning the previous level tree from the leaf nodes, then re-ordering and building a new tree for the next level. FPM-T adopts a *top-down* approach, scanning the previous level tree, building an intermediate tree at the same time and, after trimming and resizing, it produces a new tree for the next level. Both of these methods have been shown to work effectively under certain specific conditions.

As an extension to our current work, we apply our algorithms to *cross-level* mining, another important research topic in frequent pattern mining. The new algorithm addresses the cross-level domain, and also provides good performance.

The chapter is organized as follows. Section 3.2 reviews related work in multi-level frequent pattern mining. In Section 3.3, we introduce a *benchmark algorithm*, formulated

as a direct extension to FP-growth. In Section 3.4, we discuss the motivation of our work. Section 3.5 and Section 3.6 describe the two new algorithms in detail. In Section 3.7, the new method for cross-level mining is provided. Section 3.8 provides a review of our research objectives in this chapter, and Section 3.9 offers final conclusions.

3.2 Overview of multi-level mining

3.2.1 Concept

To begin our study of multiple level frequent pattern mining (MLFPM), we use basic definitions similar to those in Chapter 2. The definitions of concepts such as *pattern*, *support*, *confidence* and *threshold* can be found in Section 2.4.1.

Initially, researchers tended to use a *uniform* threshold for MLFPM [HF95] [SA95]. However, in order to model real life data mining more closely, it is more accurate to use different thresholds for different levels. This is because the count of a certain item at a higher level will be greater than the corresponding count at a lower level. In this case, we will need to set the threshold of upper levels to be higher. Consequently, researchers eventually switched to the use of multiple thresholds for different levels [HK06]. In our research, we will use this new technique as well.

Typically, MLFPM utilizes a *concept hierarchy*. This hierarchy represents the relationship amongst different concept levels. For example, in Figure 3.1 we present a concept hierarchy that one might find in a typical retail company.

In MLFPM data sets, each item will have level-specific meanings that are directly related to the hierarchy. In other words, for a d-level hierarchy, each item in the data set can provide d meanings. For example, given the hierarchy of Figure 3.1, the "Wonder" brand wholewheat bread may be viewed from three different perspectives. At the bottom level (Level 1), it

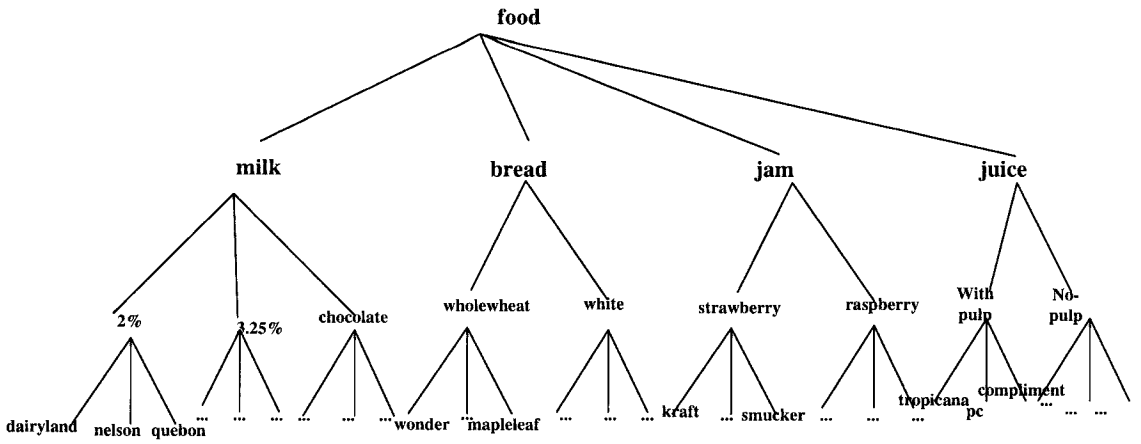

Figure 3.1: Concept hierarchy.

means Wonder wholewheat bread, a specific *product*. At Level 2, it refers to the *kind* of bread. At Level 3, it simply represents bread. In such a way, we can define all three level-specific item interpretations in order to build our 3-level fp-trees. In our specific research, relatively speaking, *higher level* refers to a higher level of abstraction in the hierarchy while *lower level* points to a lower level of abstraction. In the above example, Level 3 is a higher level while Level 1 is a lower level.

3.2.2 Method

Relative to the hierarchy in Figure 3.1, we can investigate problems associated with the *shopping cart* model. For instance, we may encounter the sample shopping cart records shown in Table 3.1.

To simplify the problem, we will need to first encode the detail-level information. For example, we encode hierarchy information by assigning a number to each item of each level. In Figure 3.1, we have 3 levels (excluding "food", since there is only 1 item at the root level). At Level 3, we have milk, bread, jam and juice. We can assign "1" to milk, "2" to bread, "3" to jam and "4" to juice. At the sub-level for milk, Level 2, we assign "1" to 2% milk, "2" to

TID	Items
1	2% diaryland milk, nelson chocolate milk, mapleleaf wholewheat bread
	wonder white bread, kraft strawberry jam
2	2% diaryland milk, wonder wholewheat bread, mapleleaf wholewheat bread
	mapleleaf white bread, pc with-pulp juice, kraft raspberry jam
3	2% diaryland milk, nelson chocolate milk, mapleleaf wholewheat bread
	wonder white bread, kraft strawberry jam
4	2% nelson milk, dairyland chocolate milk, mapleleaf white bread
	smucker strawberry jam,compliment no-pulp juice, smucker raspberry jam
5	2% diaryland milk, 2% nelson milk, chocolate quebon milk
	wonder wholewheat bread, mapleleaf white bread, smucker strawberry jam
6	nelson chocolate milk, wonder wholewheat bread, wonder white bread
	kraft strawberry jam, kraft raspberry jam

Table 3.1: Shopping cart records

TransactionID	Items
1	111 132 212 221 311
2	111 211 212 222 412 321
3	111 132 212 221 311
4	112 131 222 312 423 322
5	111 112 131 211 222 312
6	132 211 221 311 321

Table 3.2: Encoded transaction table

3.25% milk and "3" to chocolate milk. Furthermore, at the sub-level for 2% milk, we assign "1" to Dairyland, "2" to Nelson and "3" to Quebon. As a result, "132" represents Nelson chocolate milk, while "311" implies Kraft strawberry jam. Note that the encoding can be extended to permit more than 10 alternatives per level. The difference between this encoding method and that found in single level FPM is that in the single level case we encode all the items explicitly. However, in the multi-level context we will add "*" to encode higher level items. For example, at Level 2, we will have items like "11*", "13*", and "31*". At Level 3 we will have "1**", "2**", "3**" and "4**". Consequently, we would get the encoded table shown in Table 3.2 for the bottom level corresponding to the transaction lines listed in Table 3.1.

In addition, we define *distinct items per level* and *total transaction elements* as follows.

Definition 7. *The term "distinct items per level" refers to the number of different items per concept level.*

Definition 8. *The term "total transaction elements" refers to items that come from the itemset and occur repeatedly in the levels themselves.*

For example, in the data set shown in Table 3.2, distinct items at Level 1 are "111", "112", "131", "132", "211", "212", "221", "222", "311", "312", "321", "322", "412", and "423". Thus the number of distinct items at Level 1 is 14. On the other hand, since 3 out of the 6 total transactions each have 5 items, and the other 3 transactions each have 6 items, then the number of total transaction elements at Level 1 becomes $3 * 5 + 3 * 6 = 33$.

In single level mining, we only need to concern ourselves with rules for the current level. But in multi-level, rules between different levels may affect each other. It is often suggested that higher level rules are more useful than lower level rules [HF95]. This is the case because, at a higher level, there are fewer distinct items per level compared with the lower levels and therefore information is more compressed. Consequently, from the decision making point of view, it is usually more helpful to understand information at higher levels.

As well, it has been suggested by some researchers that we can mine items only if their ancestors at higher levels are frequent [HF95]. Still other researchers have implied that if the distribution rules of the descendants at a lower level are similar to the distribution rules of the current item at a higher level, then the rules at the lower level are redundant and thus should be removed [SA95].

The very first idea for a practical MLFPM algorithm comes from a variant of the Apriori method proposed by Han et al. [HF95]. The idea is based upon the application of Apriori to the case of multiple levels. Suppose there are 3 levels in a given hierarchy. First we generate large 1-itemsets from the highest level (Level 3) to the lowest level (Level 1). Large 1-itemsets of Level 2 are descendants of large 1-itemsets of Level 3. Similarly, we generate

large 1-itemsets of Level 1 only if their corresponding items at Level 2 are "large". Then at each level, we generate candidates for k-itemsets from large (k-1)-itemsets. After this, we have to test these candidates against the data set so that we can know whether they are "large" or not. These costs can become quite large.

3.3 MLFPM Benchmark based on FP-growth

Because of the performance limitations of Apriori, a more appealing option is to exploit the non-candidate generation algorithm, FP-growth. As shown in Algorithm 4, the idea is to first scan the data set once to generate frequent items for all levels, filtering out items whose counts are below the threshold. After this, we iteratively scan the database d-1 times to generate an fp-tree for each concept level. Next, we do mining on the fp-tree from each level and then discard the tree when mining is finished. In fact, this is a reasonable starting point for our research efforts. Consequently, we will use this approach as a benchmark and will demonstrate later how this initial technique can be improved.

Algorithm 4 MLFPM Benchmark based on FP-growth

Input: an encoded transaction data set
Output: multi-level frequent pattern rules

1: scan the data set once for frequent items at all the levels and build a header table for every level
2: **for** level $k = 1$ to level d **do**
3: scan the database once to build an fp-tree for the current level
4: do mining on the current fp-tree and generate frequent pattern rules for the current level
5: **end for**

3.4 Motivation

The primary motivation for the new algorithms is that the benchmark is forced to do too many scans of the data set. Since an fp-tree stores complete information for the data set, we observe that it may be faster to scan the associated fp-trees instead of scanning the underlying data set. At the same time, when scanning an fp-tree, we can directly create a new fp-tree for the subsequent level.

A second observation is that as the multi-level algorithms move from lower levels to higher levels, the size of the fp-trees becomes significantly smaller as the structure becomes more compressed. If we use the benchmark, we will need to scan the huge data set every time, and the cost of such scans is consistently large. Therefore we can not take advantage of the shrinking size of the higher level fp-trees.

Moreover, there may also be a big time difference in tree-construction. In the bottom level fp-tree, we need to insert items in each transaction of the raw data set in both the benchmark and our new approaches. At higher levels, the benchmark will have to continue the same process, inserting each item one by one. In contrast, as the trees shrink, we can insert far fewer items into the new fp-trees if doing tree-construction from existing trees. Therefore, the new approaches become more attractive. The primary distinction between the new methods will be whether we perform the processing in a bottom-up or top-down fashion.

In presenting our research for MLFPM, we identify a core list of research objectives as follows.

- Reduce the large cost of repeatedly scanning the raw data set.
- Try to use existing fp-trees to generate new fp-tree for multi-level usage.

- The newly generated fp-trees for multi-level mining should be amenable to single level mining methods in order to generate frequent pattern rules.

- Apply these same methods to the problem of cross-level frequent pattern mining.

3.5 FPM-B: A bottom-up approach

As the name suggests, FPM-B starts from every leaf node of a previous fp-tree and traverses each branch upwards until it reaches its root node (Root). Then it adopts a tree-construction method similar to that found in FP-growth in order to create an fp-tree at the next level. The process is described in Algorithm 5. In fact, we can divide the process into several sub-processes.

Algorithm 5 FPM-B

Input: an encoded transaction data set
Output: multi-level frequent pattern rules
1: scan the database once and build a header table for every level
2: do filtering
3: build an fp-tree for the bottom level
4: do mining on the bottom level tree
5: **for** level $k = 2$ to level d **do**
6: call the function defined by Algorithm 7
7: call the function defined by Algorithm 8 {generate next level fp-tree based on previous level fp-tree}
8: do mining on the current fp-tree and generate frequent pattern rules on the current level.
9: **end for**

3.5.1 Step 1: Scan data set for the first time

Hash table construction

During the scanning and discovery process, items are inserted into a storage structure called a *header table*. Every level-based fp-tree will have its own header table to store frequent

items. In our approach, we use *hash tables* to fulfill the function of the header table. As is the case with multi-level frequent pattern mining, encoded items may have many digits. For example, an item "987654" at level 6 will have 6 digits. If we use the traditional method for single level mining in which we encode every possible item from 1 to the maximum number of items, the header table will simply become too large. By applying the idea of hashing, we can address this problem, making the header table relatively small and accessing an item in O(1) time in the average case.

Scanning

Similar to the benchmark, FPM-B will need to scan the data set to generate frequent items. The difference here is that it only identifies *frequent* items at the highest level. For the rest of the levels, FPM-B identifies *all* items at the given concept level. In other words, when building header tables for each level, we will insert *filtered* items at the highest level but, at other levels, we insert all the items into the corresponding header table. Given the encoded transaction table shown in Table 3.2, let us say we set the threshold to "2" at Level 1, "4" at Level 2, and "5" at Level 3. In the first scan, we can get all the frequent items at level 3, which are "1**", "2**", and "3**". The infrequent item is "4**", since its count is 2 and this is obviously below the threshold of Level 3. So the item "4**" will be filtered out at this stage. As well, in this scan, we can get all the items, including frequent and infrequent values, at the bottom level. We will not filter out item "322", even though its count is below the threshold for Level 1. We will discuss the reason for this in the next section.

3.5.2 Step 2: Filtering

Since we will only examine items whose corresponding higher level items are frequent in their header tables, it may appear that we can start checking from the highest level and then

proceed downwards to the lower levels. However, we only perform this process at the highest level because, in FPM-B, we actually scan lower level fp-trees in order to create higher level fp-trees. This means all nodes in the next level fp-tree are coming from the nodes in the previous level fp-tree. Now, even though a node is infrequent at a lower level, its ancestors may be frequent at a higher level. If we eliminate a node in the previous level fp-tree, we will not have this node's corresponding higher level node in the next level fp-tree, which may result in a loss of information. Therefore, items at lower levels can be filtered only based on the processing at the highest level. Note that we refer to the items at subsequent levels as *descendants* of the generated items at a higher level; conversely the latter are called *ancestors* of the former. In the example of Table 3.2, since "4**" is infrequent at Level 3, the highest level, then all its descendants (412, 423, 41*, 42*) will be filtered out. But item "321" will not be filtered, even though item "32*" has a count of 3, which is below the threshold for Level 2. This is because "321" has a corresponding ancestor "3**" that is frequent at Level 3. The filtering process is described in Algorithm 6. Please note that we can also apply this new technique to the benchmark.

Algorithm 6 FPM-B-Filtering

Input: d-1 unfiltered header tables {the header table of Level d is already filtered}
Output: d-1 level filtered header tables

```
1: for level k = 1 to level d − 1 do
2:   while current item is not the last item of the header table of Level k do
3:     if current item is not a descendant of any item at Level d then
4:       remove current item
5:     end if
6:     go to next item
7:   end while
8: end for
```

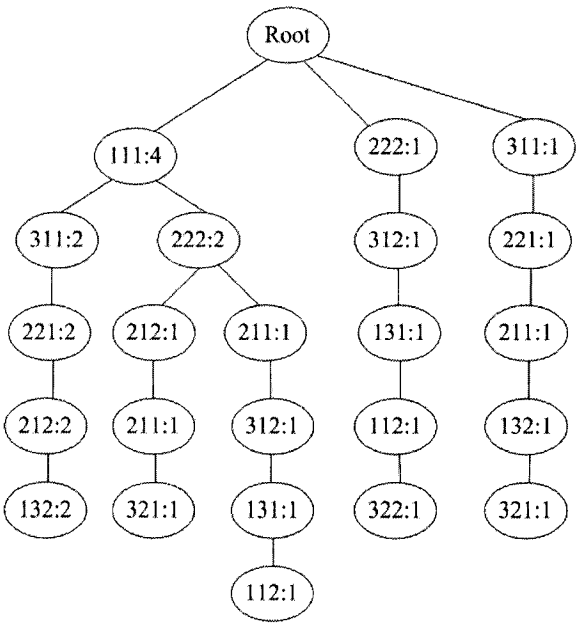

Figure 3.2: Bottom level fp-tree (BLFPT)

3.5.3 Step 3: Scan data set for the second time

After filtering, we now have all the useful items at the bottom level. By following the steps of FP-growth we can start tree-building. First, we create a node called Root that serves as a root node for the tree. Then we grow the tree by performing a second scan of the data set. As discussed in Chapter 2, we scan each transaction, re-ordering every item according to the order in the header table (descending), and then we insert all the items into the tree. After we build the bottom level fp-tree (BLFPT), we can perform the mining process based on the threshold. Please note that although we may have more items in the BLFPT fp-tree than might be the case with the benchmark, the mining process will only be performed on those items that are frequent at their level. In other words, only items that have a count above the threshold of their corresponding level will be considered in the mining phase. For the example of Table 3.2, the resulting bottom level fp-tree is shown in Figure 3.2.

3.5.4 Step 4: Gather leaf node entries

Once we build up the BLFPT, we can use the information represented by the tree. Specifically, since the BLFPT contains complete information from the original data set, it is possible to manipulate this tree directly to build the next level fp-tree. In this phase then, we perform a scan of the previous fp-tree. To do so, we utilize a storage structure such as an array list. As we scan the previous level fp-tree, when we reach a leaf node, we initialize a node in the array list and point it to the leaf node. After we finish the scan, we will have pointers to all the leaf nodes in the fp-tree. This process is defined in Algorithm 7. The running example is illustrated in Figure 3.3.

There is also a special case to note. If a node is not a leaf node physically, but its count exceeds the total count of its children, it will become a *potential leaf node*. This is because we use the leaf node count as the branch's *uniform count* and take all the nodes in the branch for the purpose of generating next level items. However, in this case, since the count exceeds the total count of its children, part of the count information would be ignored, thereby resulting in missing information. So we will test if the nodes are potential leaf nodes at the same time that we gather the leaf node entries. An example is illustrated in Figure 3.4. Note that in this figure the sample fp-tree is different from that found in Figure 3.3. As we can see, the node "111" has a count of 4, but the total count of its children is 2. Therefore, if we do not pay close attention to this node when building a new branch for the next level, then we would begin from the *real* leaf node "132" and get the branch: "11*:2-31*:2-22*:2-21*:2-13*:2" (Note that tree construction is discussed in the next section). Consequently, we lose part of the count for node "111" (i.e., $4 - 2 = 2$). By paying attention to the importance of *potential leaf nodes*, we maintain the completeness of the information we gather from the

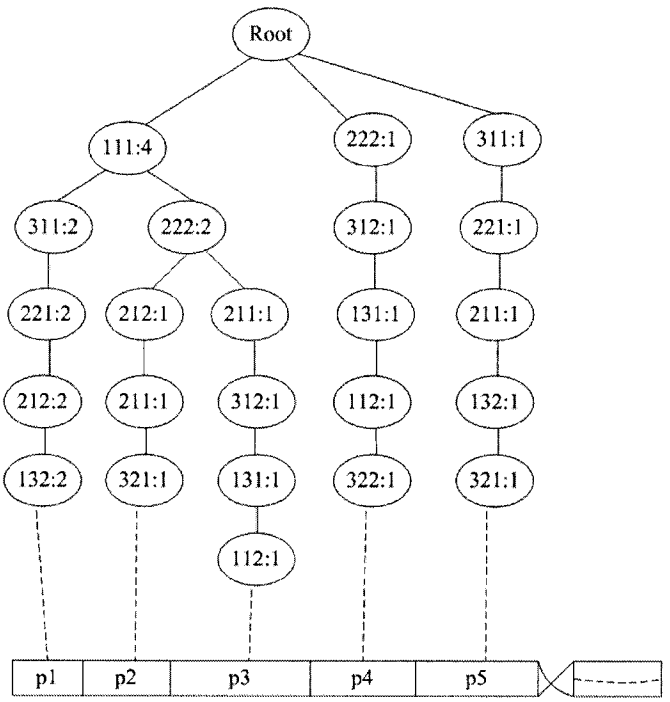

Figure 3.3: Gather leaves

previous level fp-tree.

Algorithm 7 Gather-Leaf-Node

```
1: for every node in the fp-tree do
2:    if it is a leaf node then
3:       make a pointer to the array list {an array list used to store the pointers}
4:    else if it is a potential leaf node then
5:       make a pointer to the array list
6:    end if
7: end for
```

3.5.5 Step 5: Branch scan and tree construction

This phase is described in Algorithm 8. For leaf nodes and potential leaf nodes, we perform a bottom up scan until we reach the Root node. At the same time, we gather the names of all the nodes scanned in the process. We use "*" to replace certain digits of the names in order to form new names that, in turn, are used to create the nodes at the next level.

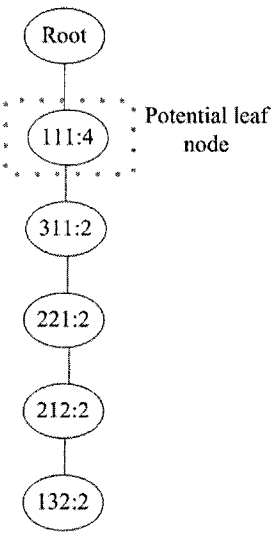

Figure 3.4: Potential leaf nodes

Moreover, some nodes at the previous level would match the names found at the next level. We adopt the idea of boolean rules in [HF95] to do *eliminations*. That is, a repeated item in one transaction line will be treated as though it occurred only once. As in an fp-tree, this means that if two items at the same branch have the same item name, one should be removed. So, for example, for the third branch of the tree in Figure 3.2, we start from the leaf node "112". As we go up, we encounter the names of the nodes in this branch, including "131", "312", "211", "222" and "111". Then we generate all corresponding items at the next level, such as "11*", "13*", "31*", "21*", "22*" and "11*". Since we now have two "11*"s in this branch, we only keep one of them, as shown in Figure 3.5. We then use the order of the header table for Level 2 to sort the names, getting "22*", "21*", "31*", "13*" and "11*". Following this order, we insert all the new nodes into the tree and then form the next level fp-tree. The final result is illustrated in Figure 3.6.

Algorithm 8 Bottom-up build-fp-tree

```
1: for every branch in the fp-tree do
2:   for every node in the branch do
3:     generate the next level item name Name_new based on the name Name_old at the
       current level
4:     if Name_new already exists in the names generated by this branch then
5:       remove this name
6:     else
7:       insert Name_new into an intermediate storage
8:     end if
9:     re-sort all the new names of the new level in descending order
10:    initialize the new nodes based on the new names
11:    insert all the new nodes into the next level fp-tree
12:  end for
13: end for
```

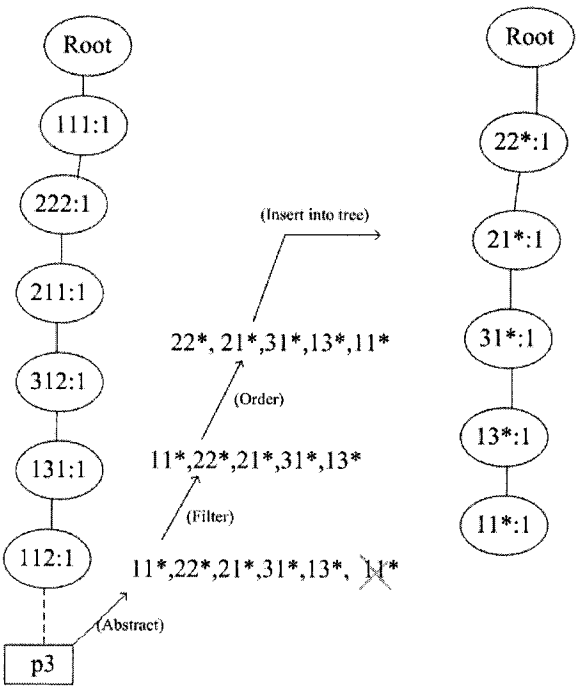

Figure 3.5: Filter duplicate items

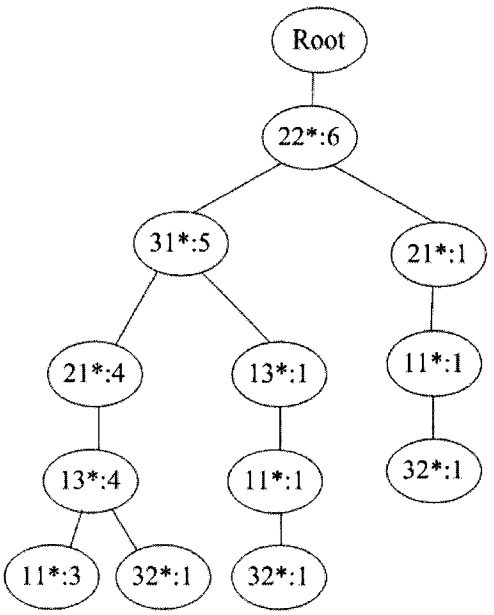

Figure 3.6: Bottom-up construction of new fp-tree

3.5.6 Step 6: Mining

When we finish the tree-construction process, we have a new compressed fp-tree for the current level. At this point, we may initiate the mining process in order to get the association rules that we want. This level-by-level process is very similar to the mining part in single level frequent pattern mining (SLFPM), described in Section 2.4.3. Moreover, relative to the construction of Level 2 fp-tree, trees at higher levels can be generated in a similar way.

3.6 FPM-T: A top-down approach

3.6.1 Step 1: Scan data set for the first time

As is the case with the benchmark and FPM-B, FPM-T needs to scan the database to generate frequent items as well. The primary difference between it and FPM-B is that, at this stage FPM-T only creates header tables for the highest level and the bottom level. In fact, FPM-T traverses fp-trees to generate header tables for the levels between the highest and the bottom level. We will discuss this process in subsequent sections.

3.6.2 Step 2: Filtering

Recall that in Section 3.5.2, we explained how to do filtering in the header tables. In this phase of FPM-T, we are essentially performing the same process. In short, we only perform filtering at lower levels based on the values at the highest level.

3.6.3 Step 3: Scan data set for the second time

Again, this phase is analogous to the one in FPM-B, as described in Section 3.5.3. Specifically, we perform another scan of the data set to generate an fp-tree for the bottom level (BLFPT).

3.6.4 Step 4: Scan previous fp-tree to build next fp-tree

Create a new tree

The basic method is shown in Algorithm 9. We start scanning from the root of the BLFPT and create a new Root for the tree at the next level. As we encounter nodes from the previous level tree, we first create an associated item at the next level (line 1). Next, we again exploit the idea of boolean rules in [HF95]. If two items in the same branch have the same item name, one will be eliminated. Regardless if the item is duplicated or not, we will visit its children and apply Algorithm 9 recursively. In order to make the tree more compressed, a second technique is employed. Specifically, we will combine the nodes between siblings if both have the same item name. As a result, all the children of each node will become siblings to each other. Accordingly, we may need to combine some of the children if they share the same item name. The process of combining sibling nodes is described in Algorithm 10. Please note, in practice we can combine nodes at the same time that we are creating the new tree.

Algorithm 9 Create a new tree

```
1: get New-Itemname from Current-Node {eg: 85 → 8*}
2: if New-Itemname does not exist in the new branch of the next level fp-tree then
3:    initialize this node
4:    append this node to the new branch, modifying all related links
5: end if
6: go to children of the Current-Node
7: call the function defined by Algorithm 9
```

Algorithm 10 Combine

```
1: if current-Node has no child then
2:    return
3: else
4:    check every child of Current-Node
5:    if it has an identical name to an "early" child then
6:       move its children so that they become the children of the "early" child
7:       modify the count of the "early" child
8:       modify all the other related links
9:    end if
10: end if
```

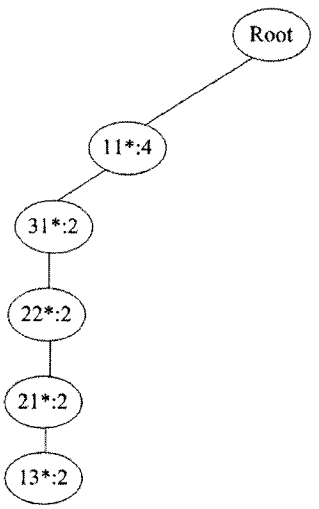

Figure 3.7: New branch created.

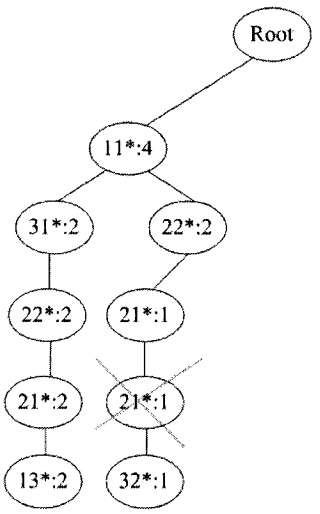

Figure 3.8: Eliminate nodes in a top-down manner.

As illustrated in Figure 3.7, we get the first branch of the new tree by scanning the first branch of the previous tree, creating a corresponding node, renaming it, initializing all necessary connections between this node and other nodes, and appending it to its parent or sibling. With respect to the process of filtering duplicated items in a single branch, we keep the item that is closer to the Root. Accordingly, the item that is closer to the leaves will be deleted. An example can be found in Figure 3.8. In this branch, since both "212" and "211" will generate "21*" at the next level, we will delete the node that is generated by "211" because "212" is "higher" in the tree. Please note, in the real algorithm implementation, we can simplify the process by not generating the duplicated node at all. With respect to the process of combining sibling nodes when possible, an example is shown in Figure 3.10.

Create header table

Recall that in the first scan of the data set, we generate two header tables, one for the bottom level and one for the highest level. Now we will start to build new header tables for the fp-trees whose corresponding levels are between the bottom and the top. We will perform this process with a scan of the fp-tree, which is typically very fast. This process is

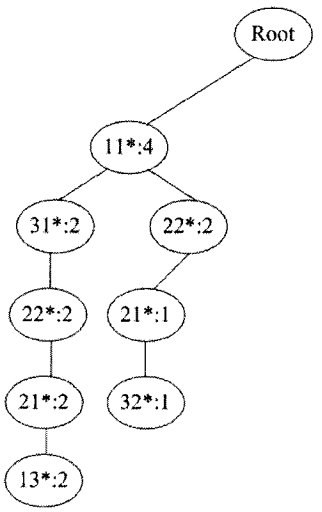

Figure 3.9: After elimination.

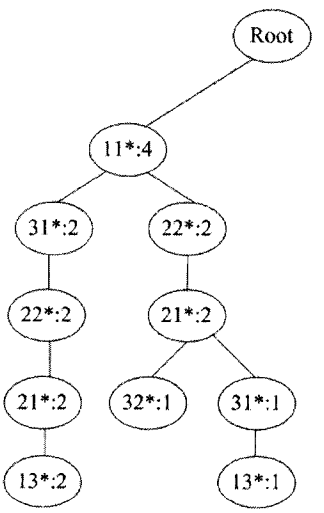

Figure 3.10: Combine nodes between siblings

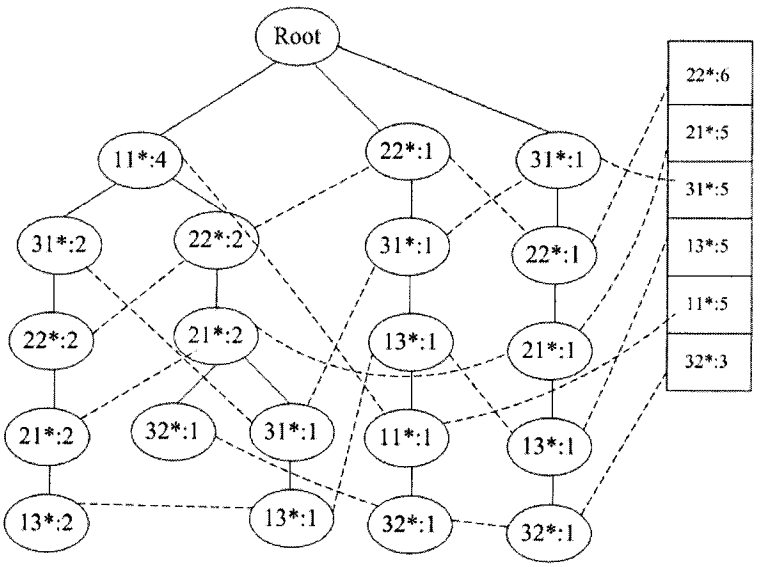

Figure 3.11: Level 2 header table

similar to the creation of header tables by scanning the data set, although now we scan the fp-tree instead. When scanning the tree from the Root, following a top-down manner, every new item is inserted into the header table. If the item already exists in the header table, we will simply update its count. We also use a linked-list to maintain a connection between items that have the same item name but are located in different branches. An example can be seen in Figure 3.11

Re-order the tree

In FP-growth, the most frequent item in the header table is deemed to have the "lowest" order. Typically we assign 0 to it. For example, if we have three items with counts "3, 2, 1", we set the order of the item with count "3" to 0. The more infrequent the item, the larger its corresponding number in the header table. This is called a *descending* pattern, which is also the order we use to build the fp-tree. In other words, for every branch of the fp-tree, items that have smaller order in the header table will appear closer to the Root. Similarly, in the mining process, we start from the largest order in the header table (least frequent),

and use a bottom-up traversal to recursively mine. Details of the mining process in MLFPM are essentially the same as in single level frequent pattern mining. Since we create nodes for the new level from the previous level tree, every new node follows the order relative to the structure of the previous level tree. But at the new level, the order of items in the header table is different from that at the previous level; thus, we have to make the nodes satisfy the descending order pattern discussed above. Consequently, we have to do re-ordering of the nodes in the current level fp-tree. The method for doing so is shown in Algorithm 11. An Example can be seen in Figure 3.12.

Algorithm 11 Re-Ordering

```
1: for each node in the header table in descending order do
2:    with an upward traversal, check the branch where this node is located
3:    if any child's order is smaller than its parent's order then
4:       swap these two nodes
5:       modify the links of each node, such as the parent, siblings, and children
6:       if a new node is created, also insert it into the item's linked list
7:    end if
8: end for
```

Re-combine the tree

At the time we re-order the nodes of the fp-tree, we expand the tree structure as a side effect. To make the tree more compressed, in order to facilitate the mining process, we again exploit a combining function. Typically, this process is similar to the earlier step in which we created the fp-tree from the previous level. If items share a common parent and have the same name, then they will be combined together. A graphical example can be found in Figure 3.13.

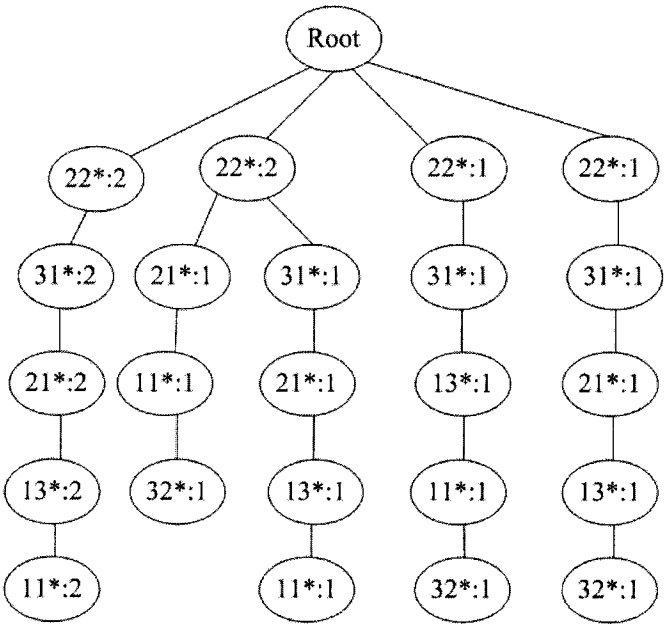

Figure 3.12: After re-ordering

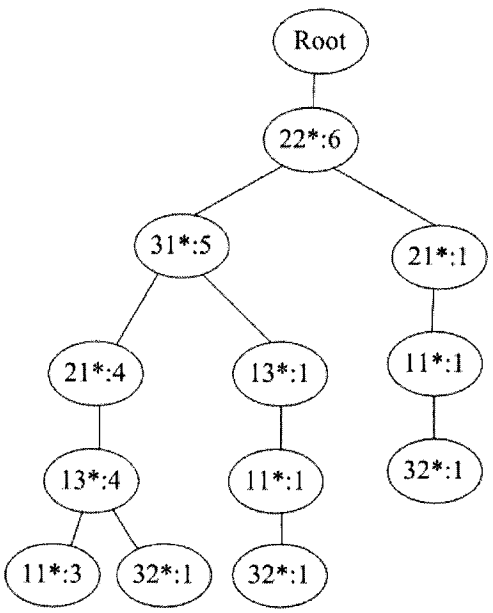

Figure 3.13: After combining

3.6.5 Step 5: Mining

After we finish the re-combining process, we are left with a cleaned, compressed fp-tree for the next level, just as we have with FPM-B. Again, the mining process for each level is very similar to that of single level frequent pattern mining (or FPM-B). Consequently, we will not repeat the details here.

3.7 FPM-Cross: cross level mining

Defining rules at every level in the hierarchy ultimately produces what we call multi-level frequent patterns. Providing FPM-B and FPM-T gives us a fast way to mine those rules at each single level. From time to time, users or decision makers may also want to understand the rules *across* several levels. This is because the frequency of items from different levels may affect each other. For example, knowing that customers buy milk and strawberry jam together may be more useful in making strategic decisions if the company in question has a preference for restocking strawberry jam due to the fact that they can obtain a discount from wholesalers. In this case, buying milk and jam that is at the same level in the concept hierarchy becomes less important than buying milk and *strawberry* jam that is at a different level, or *cross-level* in the concept hierarchy. Therefore, we turn our attention to the concept of cross-level frequent pattern mining (CLFPM).

3.7.1 Filtering considerations

One can imagine that by using an fp-tree to represent relationships among levels, we could insert items of all levels into the fp-trees, and in the process we could discover the co-occurrence between any two items that are from different levels. However, in this way the fp-tree would probably become very big if we do not provide any form of compression.

Therefore, using filtering methods to make the tree smaller becomes very important.

In contrast to MLFPM, when doing cross-level mining we use two kinds of thresholds. The first one is simply a threshold for each level, as was the case in MLFPM. So when we have 9 levels, we have 9 thresholds, one for each level. The second one is a unique threshold for cross-level purposes. Only items of all levels that are above this threshold can be considered as frequent. Consequently, we can use the first threshold as a filter to do level-by-level mining (as in MLFPM), and use both thresholds to do cross-level mining.

Recall that when building an fp-tree from the previous level, we can not use the first threshold to filter items in the tree except at the highest level, because we need to generate the next level items from the previous level items. But in cross-level, the situation changes. In MLFPM, we apply a technique by which we create trees one by one from the lowest level to the highest level. For example, if we eliminate an item at Level 2 that, in fact, we should not delete, it would result in a corresponding item missing in all the levels after Level 2. On the other hand, in FPM-Cross, since we only generate a single "mixed" cross-level tree, we do not apply this principle. As a result, we can use the first threshold to filter out items just as we did in the benchmark in Section 3.3. In this way we will filter all the items whose counts are either below the threshold for their level or the threshold for cross-level mining. This produces a large reduction in tree size. The core method is described in Algorithm 12.

3.7.2 Selecting the supporting method

In terms of supporting algorithms for FPM-Cross, we have the two MLFPM options discussed earlier. We can choose to build upon the top down approach FPM-T. However, the new cross-level tree becomes deeper than before, which means even more long branches will be grown. Recall that in FPM-T, we need to re-order the nodes and then combine them afterwards.

Algorithm 12 Filtering-Cross

Input: header table for each level
Output: a header table for the cross-level

```
1: for level k = 1 to level d do
2:    for every item in the header table of level k do
3:       if the support of this item is above the threshold of its level AND the support of
         this item is above the threshold of cross-level then
4:          insert this item into the header table for the cross level
5:       end if
6:    end for
7: end for
```

Thus it is not ideal to adopt a top-down approach for cross-level mining. Therefore, we switch to the bottom-up method. The integrated process is shown in Algorithm 13.

Algorithm 13 FPM-Cross

Input: base level fp-tree
Output: cross-level frequent pattern rules

```
1: call the function defined by Algorithm 12
2: call the function defined by Algorithm 7
3: for each branch of the base level fp-tree do
4:    for every node of the branch do
5:       generate item names for all levels except for base level
6:       check these items against the cross level header table {including the item at the
         base level}
7:       if it exists in the header table then
8:          insert it into the intermediate storage
9:          re-order the names
10:         initialize cross level nodes
11:         insert these nodes into the cross level fp-tree
12:      end if
13:   end for
14: end for
15: do mining and generate cross-level frequent pattern rules
```

When applying the bottom-up method, we start from the lowest level that the user requests. For example, let us say the user asks for the cross-level relationship between Level 2 and Level 5. For simplicity, we will call Level 2 in this case "the base level fp-tree" and Level 5 "the top level fp-tree". As in MLFPM, we discard the Level 2 fp-tree once we

have finished constructing the Level 3 fp-tree. But now, we keep it until we complete the construction of our cross-level fp-tree. We again begin the process at the leaf entries. For each branch, we scan from the leaf node all the way up until we meet the Root node. In this way, we get all the name information for this single branch. Based on this, we generate corresponding nodes for all the levels. When generating same item names at a higher level, we eliminate duplicate items. When items do not meet the threshold at their concept level, or the threshold of the unique cross-level, their names will be deleted as well. We then take the filtered items, as we do in FPM-B, and insert them into the new cross-level fp-tree. After we scan all the branches of the base level fp-tree, we get a new fp-tree that contains items from the base level to the top level. This fp-tree fulfills the requirements of cross-level frequent pattern mining.

An example of the base level fp-tree is shown in Figure 3.14. Thresholds for each level are 2, 2, and 2. The threshold for cross-level analysis is 4. Let's say we consider the branch containing the leaf node "752:1". Here we get: "752", "236", "456", "235", "45*", "23*", "75*", "4**", "7**", and "2**". We find "752" is infrequent (count = 1), as well as "236" and "75*" are also infrequent in the cross-level context, so only "456", "235", "45*", "23*", "4**", "7**", and "2**" are inserted. In this way, we get the final cross-level fp-tree (CLFP) shown in Figure 3.15. Note that the end result is that the CLFP is larger than the single level tree but not three times larger, since pruning and filtering are happening all the time.

In terms of a viable benchmark in the cross-level domain, any proposed solution must be capable of adding next-level items to each transaction line and then building some form of cross-level tree. For example: "235", "456", "786", "457" → "235", "456", "786", "457", "23*", "45*", "78*", "45*", "2**", "4**", "7**". Of course, we can also do filtering when building this kind of fp-tree.

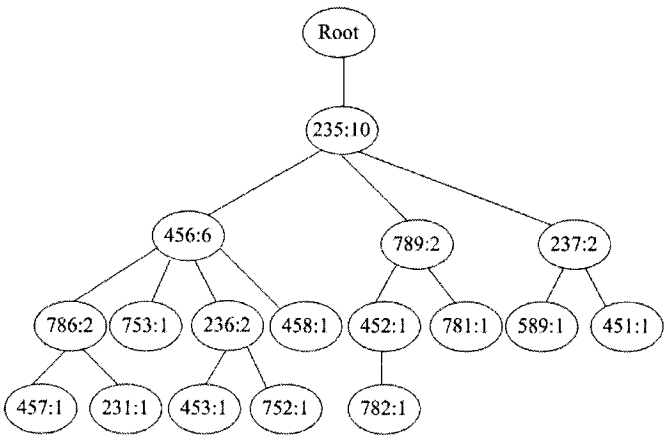

Figure 3.14: Base-level tree

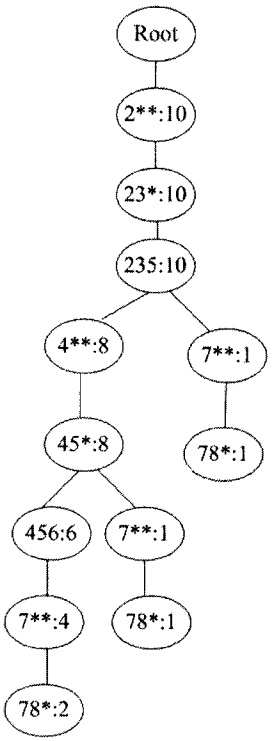

Figure 3.15: Cross-level fp-tree

So the advantage over such a benchmark is that we do not need to do a new scan of the data set in order to build the cross-level fp-tree. Instead, we can do cross-level construction after we build an initial fp-tree, and then use the bottom-up technique to generate the new cross-level fp-tree. This allows us to directly exploit the methods developed for multi-level frequent pattern mining.

One might suggest that we need only generate the cross-level tree and not the fp-trees for each level. But without single level mining techniques, cross-level mining becomes virtually meaningless. In other words, our FPM-Cross can be viewed as a "cooperative" method that works together with algorithms like FPM-T and FPM-B. The base level fp-tree in FPM-Cross always comes directly from the result of a specific fp-tree in the MLFPM algorithms. Besides, one may want only the cross-level output between Level 3 and Level 6. In this case, by using our intuitive approach and applying bottom-up processing on the Level 3 tree, which is a more compressed structure, we get a better result than could be obtained by scanning the data set again. Specifically, since a scan of the data set focuses on items in the raw transaction lines, and each of the associated items has a count of 1, then this form of cross-level construction will be much slower than if we use FPM-Cross directly on the Level-3 fp-tree.

3.8 Review of research objective

In section 3.4, we identified a number of objectives for our current research. We will now check if they are in fact accomplished.

- **Reduce the large cost of repeatedly scanning the raw data set.** Instead of spending our time repeatedly scanning the data set, we switch to scans of the fp-trees, which can be far more effective from a performance perspective.

- **Try to use existing fp-trees to generate new fp-trees for multi-level usage.** Both FPM-B and FPM-T exploit the idea that we can extract information from the previous level fp-tree, generate useful and related information for the next level, and then use this for multi-level mining.

- **The newly generated fp-trees for multi-level mining should be amenable to single level mining methods in order to generate frequent pattern rules.** Since we get the new fp-trees directly from the previous level trees, no information is lost. Therefore, there is no problem in applying standard mining methods to these new trees.

- **Apply the new methods to the problem of cross-level frequent pattern mining.** By applying our bottom-up approach to the cross-level method FPM-Cross, we can generate a new cross-level fp-tree from the base level fp-tree, as described in Section 3.7.

3.9 Conclusions

In this chapter we introduced two new algorithms for multi-level frequent pattern mining. In applying our bottom-up approach, FPM-B, we start from the leaf nodes of the previous level fp-tree, scan each branch, generate next level items, and insert them into the next level fp-tree. To apply our top-down approach, FPM-T, we start from the Root node of the previous level tree, create corresponding nodes for each node visited, combine them when necessary, re-sort the nodes based on the order of the next level header table, combine again afterwards, and finally get a compressed and cleaned next level fp-tree. Moreover, we provide an algorithm called FPM-Cross as a solution to the problem of cross-level frequent pattern

mining.

As described in Section 3.1, although a lot of work has been done in the domain of single level frequent pattern mining, less attention has been paid to the concept of multi-level mining. Since this underlying problem is significant in the frequent pattern mining context, this type of research is in fact quite practical. In our case, we have contributed to the literature by providing a number of new methods to address the core problems in multi-level frequent pattern mining. In the next chapter, we will demonstrate how mining performance can be improved by applying these new approaches.

Chapter 4

Evaluation

4.1 Introduction

With the new techniques in hand, it is time for us to apply these methods in practice to test their performance relative to the more naive approaches. In this chapter, we first introduce the environment in which the tests are performed. We give preconditions to ensure fairness for all the methods being compared. Then we carry out our tests from a series of different perspectives.

4.2 Test environment

A test environment is a neutral operating platform that treats every method to be tested in the same manner; that is, no specific method is favored in order to get better results. With respect to the current thesis, we will describe components of our test environment such as hardware conditions, software conditions, data generation, as well as preconditions that assure justness.

4.2.1 Hardware

- Dell Precision WorkStation 380
- Motherboard Dell 0CJ774

- Intel Pentium Duo Processor @ 1.8 GHz
- DDR Memory 2 GB @ 667 MHz
- Western Digital 250 GB SATA Hard Drive
- Broadcom BCM5751 NetXtreme Gigabit Ethernet Controller

4.2.2 Software

- Linux Fedora Core 5, kernel version 2.6.17-1.2157
- Java SE Development Kit(JDK) version 5.0 update 9
- Java Runtime Environment (JRE) version 5.0 update 9
- Borland JBuilder2006 Enterprise
- UltraEdit 3.2

4.2.3 Data generation

In the original Apriori paper [AS94], Agrawal et al. used several parameters to create synthetic data sets for their testing. These included one to represent the number of transactions, one to denote the average size of transactions, one to show the average size of the maximal potentially large itemsets, later called MFI in [Bay98], and one for the number of all items. Later, in FIMI03 [FIM03], a number of popular data sets were presented by researchers such as "T10I4D100K", "T40I10D100K", "pumsb_star", "chess", "connect" and so on. Most researchers who were doing research in the domain of FPM typically used these data sets.

On the other hand, for multi-level frequent pattern mining (MLFPM), Han et al. in [HF95] presented a data generation method similar to that found in [AS94]. Their parameters include the total number of items, the total number of transactions, and the average size of

Name	Meaning
$\|T\|$	Number of transactions
$\|I\|$	Average length per transaction line
$\|L\|$	Number of levels in the hierarchy
$\|B\|$	Percentage of branch number shrinking
$\|F\|$	Fan-outs from level to level

Table 4.1: Parameter setting

the potential length of large itemsets. They also define the notion of "fan-outs", which represents the relationship between descendants at a lower level to the number of ancestors at a corresponding higher level.

In our case, we employ a number of ideas from these implementations in creating our test data sets and also include some new conditions. The parameters associated with our testing program are shown in Table 4.1.

4.2.4 Preconditions

In each comparative test amongst FPM-B, FPM-T and the benchmark, we set all of the parameters to identical values. This ensures that we always get fair results for all methods. The data sets generated for our testing are considered *dense* when most of the items are spatially similar. We will explain why we use dense data sets later in this chapter.

4.3 Simple data set evaluation

As mentioned above, the advantage of the new algorithms over the benchmark is that an upper level fp-tree is more compressed than a lower level fp-tree. The reason for this can be explained as follows: if two items at a previous level can be "abstracted" into one item at the next level, then the size of the tree shrinks. For example, say we have two branches, one containing only the item "23", and the other only housing the item "24". Then, for the next level, we will have only one branch, "2*", with an increased count. Therefore, we define the

Data set	$\|T\|$	$\|I\|$	$\|L\|$	$\|B\|$	$\|F\|$
D1	250000	3	6	0	5
D2	250000	6	6	0	5
D3	250000	10	6	0	5

Table 4.2: Data sets of 250,000 lines

parameter "$|B|$" as the percentage of the branch count that shrinks. It is anticipated that the greater the percentage, the smaller the size of the tree at the next level. On the other hand, the worst case for a lower level is that none of the branches would be combined; that is, the higher level fp-tree is as bushy as the lower level fp-tree. In our tests, we can use "$|B|$" to control this setting.

4.3.1 A worst case test

We first created a group of data sets, each having 250,000 transactions as shown in Table 4.2. These data sets are considered in a worst case context, where $|B|$ is set to zero. This means that the number of branches in the bottom level fp-tree is equal to that of the second level.

These data sets vary in the average length per transaction line. All data sets have a similar count of distinct items per level, which is around 10,000 to 15,000. This makes the data sets relatively dense. As well, for total transaction elements, we can count according to the transaction number T and the average length per transaction line I. For example, D1 has the following number of total transaction elements: $T * I = 750000$.

Test results for the three data sets are shown in Figure 4.1, Figure 4.2, and Figure 4.3. In D1, since every transaction line has an item count of three, the minimum support is thus set relatively low. The thresholds of all levels are set to 0.01%, 0.02%, 0.1%, 0.4%, 2% and 4%. In D2 and D3, thresholds of all levels are set to 0.02%, 0.04%, 0.2%, 0.4%, 2% and 4%. Time displayed in these three figures is simply the total running time for all levels.

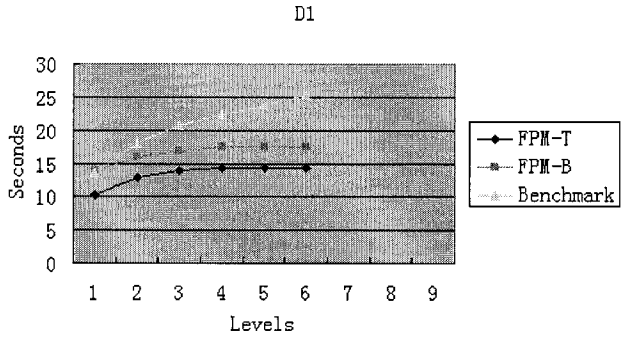

Figure 4.1: Total running time on D1 .

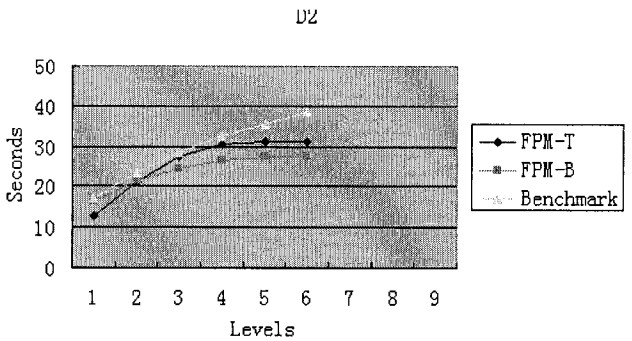

Figure 4.2: Total running time on D2 .

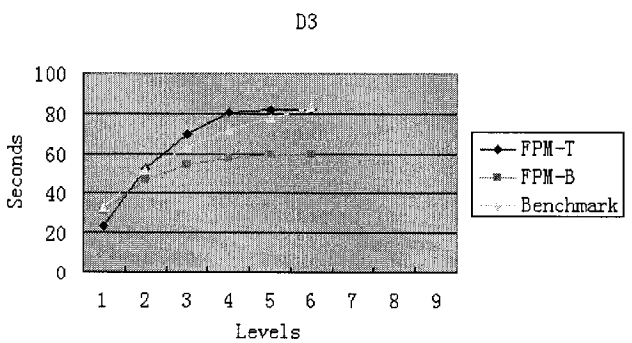

Figure 4.3: Total running time on D3 .

4.3.2 Analysis

Obviously, what we see from these results is that in D1, FPM-T is the best algorithm, while in the other two data sets FPM-B has the best performance. The difference among these three data sets is only the average length per transaction line. In other words, FPM-T is good when the average length of the transaction in the data set is small, while FPM-B works well in other situations. In these cases, note that in the early stage (i.e., lower levels) the benchmark may be faster. However, the two new algorithms take less time at higher levels—where the fp-trees are very compressed—and overtake the benchmark. Note as well that these relatively small data sets underestimate the I/O penalty of the benchmark. The next section will analyze the details of the three methods.

FPM-T Analysis

Besides the benefit of re-using fp-tree structures, one advantage for FPM-T is that it does not need to generate header tables for all the levels at the very beginning. Instead, it only produces header tables for the lowermost level and the topmost level. In the middle levels, it scans the newly generated fp-trees to create header tables. By not introducing excessive scanning of the raw data set, and instead scanning the more compressed tree from the top down, we save a lot in I/O costs. This is illustrated in Figure 4.4. When the size of the data set increases, the cost for scanning and creating header tables directly from the data increases much more quickly relative to the methods that exploit the corresponding fp-tree.

However, the disadvantage for FPM-T is that it needs to perform operations to re-size newly generated trees. By doing FPM-T, we generate higher level items based on lower level items. Meanwhile, we can not guarantee that the new items in the branch keep the frequency order of the header table at the next level. However, as per the idea of prefix sharing in the

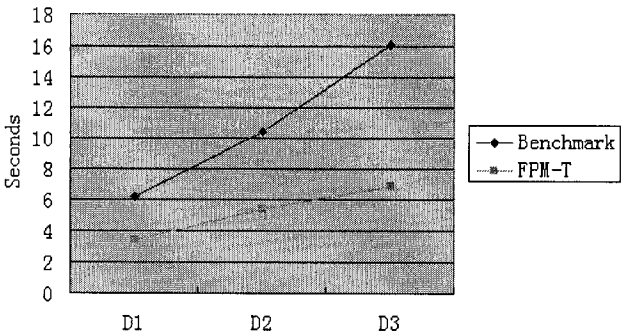

Figure 4.4: Header table processing time.

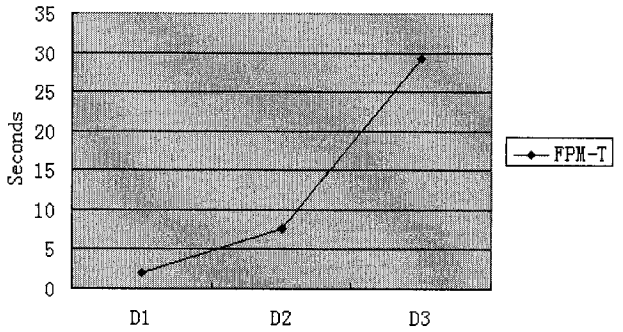

Figure 4.5: Tree re-shaping time of FPM-T on D1, D2 and D3 .

standard FP-growth algorithm, more frequent items must be closer to the Root of the tree than less frequent items. Thus, we will need to swap items in the branches to maintain this order, which is the job of the re-ordering function. In addition, re-ordered trees may not be as compact as we want. Thus we may have to combine sibling nodes in the case of item name duplication.

Please note that for times in Figure 4.5, the X-axis results on D1, D2 and D3 represent the total time for tree processing on each data set respectively. In Figure 4.6, the X-axis results represent the un-aggregated running times for each level, which are of course different from the illustration in Figure 4.5.

Let us have a look at Figure 4.5, which shows the total running time on each data set from Level 2 to Level 6. Since at Level 1, FPM-T scans the data set and builds the bottom

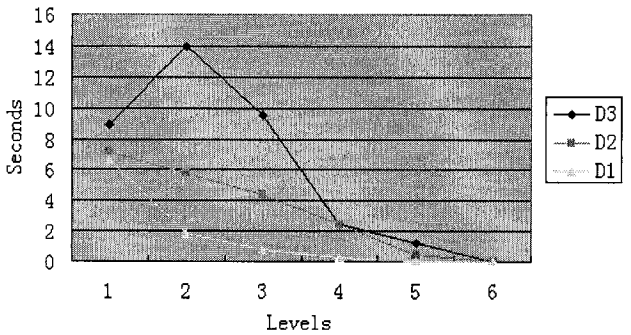

Figure 4.6: Tree re-shaping time of FPM-T on different levels.

level fp-tree, the time in Figure 4.5 displays exactly how much time is taken to re-size the tree structure. We can see that the cost of this in D1 is low. This is because when items per transaction line is relatively small, even though the tree can be bushy, we do not need to do a lot of re-ordering. As $|I|$ increases from three to six to ten, the time for processing the trees rises quickly. This is due to the trees becoming more complex in structure. Thus, we need to swap nodes and combine them more often.

Figure 4.6 provides additional insight. As we can see, the cost after Level 4 is very small, while the running time from Level 1 to Level 3 consumes the greatest percentage of computational resources. Let us look at the latter phase first. At Level 1, what FPM-T does is to scan the raw data set and build header tables for the bottom and top levels. So the difference on D1, D2 and D3 in time is small, only due to different transaction size. At Level 2, the time for FPM-T on D1 drops a lot relative to Level 1, and we can see that it does not spend a lot of time at this stage in reshaping the fp-tree. At the same stage, time for FPM-T on D2 drops far less than on D1 due to the increase in the total number of transaction elements. The time of FPM-T at Level 2 of D3 is even greater than it is at Level 1. Consequently, we know that the cost of re-ordering and combining at this point is larger than is the case with the other data sets.

Perhaps the most important point is that, from Level 4 and above, the cost relative to the whole process is very small. This implies that the technique of re-using fp-trees works very well at higher levels. When a lower level fp-tree is converted into a higher level fp-tree, the number of distinct items per level drops drastically. For example, in D2, the number of distinct items at the bottom level is 12352, then it drops to 4330 at Level 2. At the same time, the number of total transaction elements decreases accordingly. As such, the size of the tree also shrinks considerably. As the level goes deeper, we have a more and more compressed tree. Consequently, if we have more levels in the concept hierarchy, FPM-T is more advantageous. Also, what is good to see is that FPM-T works extremely well when the number of items in each transaction line is very small (e.g., three). In Figure 4.1, for example, the total running time of FPM-T on D1 is 56% less than that of the benchmark. As the length of each transaction line goes up, the advantage drops off. On D2, we have 20% less computational time. On D3, the difference in total time between FPM-T and the benchmark is very small.

FPM-B Analysis

Though both algorithms re-use existing tree structures, FPM-B adopts a different approach from FPM-T. It scans branches to gather necessary node information and inserts them into the new tree. The node insertion function is similar to that of a normal FP-tree construction. Unlike FPM-T, FPM-B does not need to do tree re-shaping since the insertion of nodes maintains the prefix sharing feature and guarantees that more frequent nodes stay further from the leaves than less frequent nodes. The disadvantage of FPM-B compared with FPM-T is that FPM-B needs to build a header table for all levels during the first scan of the data set. Therefore FPM-B does not save time at this stage as FPM-T does.

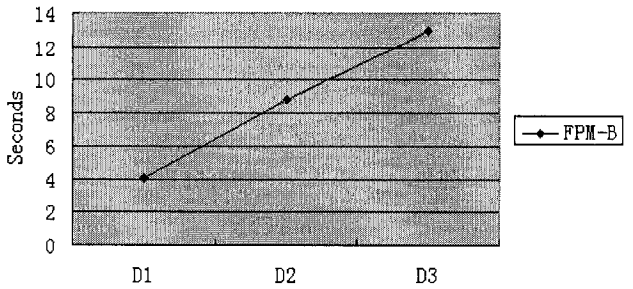

Figure 4.7: Tree rebuilding time of FPM-B on D1, D2 and D3 .

Let us have a look at Figure 4.7. The running time of FPM-B on the three data sets increases almost linearly as this figure shows. FPM-B is potentially affected by many factors, but one of the primary parameters actually has little influence in this particular test. This is the number of tree branches. Using a bottom-up technique, FPM-B collects links to all leaf nodes. In fact, having fewer branches will improve the performance. The main difference among D1, D2 and D3 is simply the length of transactions which, in turn, implies that the branch number is likely similar for each data set. Therefore, the main factor affecting the performance in Figure 4.7 is the length of the transactions. Since in D1, D2 and D3, $|I|$ is 3, 6 and 10 respectively, we can assume that the total time on these data sets will follow a similar pattern.

The results shown in Figure 4.8 basically tell us that FPM-B performs better and better as the level goes up. As we have discussed, when the level count increases, the number of distinct items per level decreases, as does the size of the fp-tree. In addition, the branch number of the tree also decreases significantly. Although in this figure we expect to see a worst case scenario in which the branches of the bottom level are almost as numerous as those of Level 2, the branch count of the higher level trees still drops from level to level, as we will discuss later.

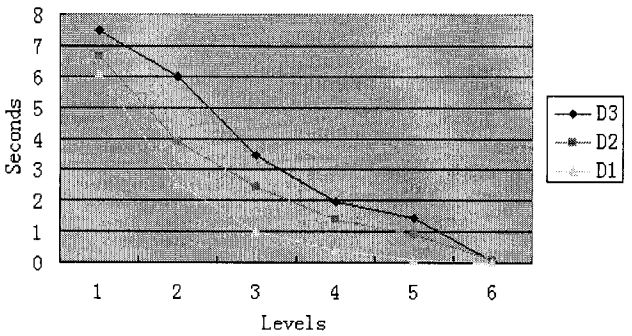

Figure 4.8: Tree rebuilding time of FPM-B on different levels.

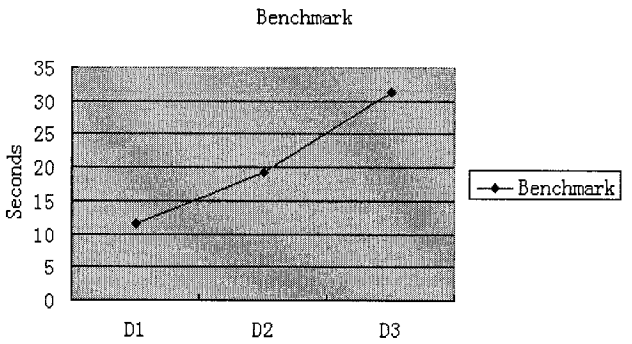

Figure 4.9: Tree rebuilding time of the Benchmark on D1, D2 and D3.

Benchmark Analysis

The case for the benchmark is relatively simple. The benchmark is not faster than FPM-T in header table building, nor faster than FPM-B in tree reconstruction. The running time for the benchmark increases linearly. This is because although it would be advantageous to exploit the item number reduction, the benchmark algorithm has to consistently pay the overhead of the data set I/O. At each level, it has to scan the raw data set once. The raw data set does not shrink; therefore, the processing time from the bottom level to the highest level can only decrease slowly. The results can be seen in Figure 4.9, which displays the total time to build all level fp-trees on the data sets D1, D2 and D3 and in Figure 4.10, which shows in detail how the running time changes when the level goes up on those three data sets.

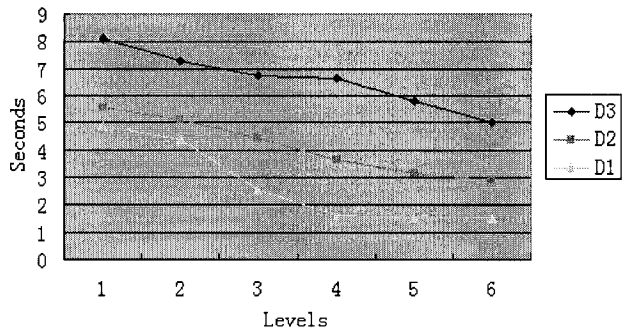

Figure 4.10: Tree rebuilding time of the Benchmark on different levels.

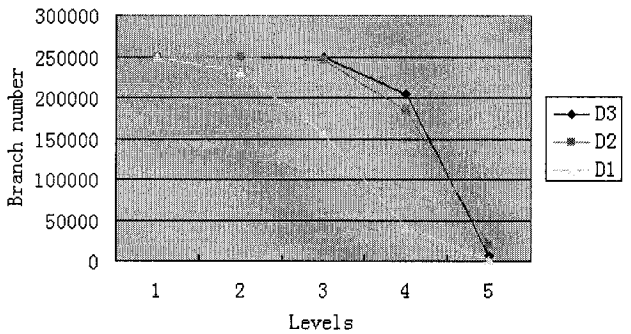

Figure 4.11: Branch number shrinking on D1, D2 and D3.

A final observation

Based on the above tests and analysis, we find that in most cases the two new algorithms surpass the benchmark in terms of running time. When we have a data set in which the transactions are relatively short, we can use FPM-T to generate multi-level fp-trees. Except for this case, FPM-B is a better choice than FPM-T and the benchmark. It performs linearly as the length of the transaction increases, and it also performs well in other cases as we will demonstrate later. In the following sections, we build upon the above observations and analysis. When the data set has short transaction length, we will use FPM-T versus the benchmark. In other cases, we will use FPM-B to compare with the benchmark.

4.4 Branch count reduction

Let us look first at Figure 4.11. It shows how the number of branches changes on the above three data sets as the level count goes up. The coordinate at Level 1 of the curve "D1" represents the branch number of the newly generated Level 1 fp-tree. As we have discussed above, we set the parameter $|B|$ to its worst case. The fact that D1 changes slightly from Level 1 to Level 2 is because it has a short transaction length. In the case of D1, although we set $|B|$ to zero, the branch number of Level 1 still shrinks a little due to item combinations. In contrast, the curves "D2" and "D3" show different trends. In these two tests, the branch count of fp-trees from Level 1 to Level 3 does not make a noticeable change. In the real world, this is not likely to be the case. Usually, many people will buy similar items or create similar transactions. For example, if we find that three transactions out of ten are the same, we can expect only eight branches in the corresponding fp-tree. On the other hand, when lower level items are converted to higher level items, some of them are combined, which causes an additional form of branch reduction. Therefore, in a business transaction data set, branch count will reduce greatly from lower levels to higher levels. The example of the curve "D1", as well as the later levels from Level 4 to Level 5 of the curve "D2" and "D3", illustrate this feature.

At this point, we modify the parameter $|B|$ to see how much effect it will have on the output. Figure 4.12 is a sample test on the data set "D2". We set $|B|$ to 10%, meaning that, after Level 1, one out of ten of the branches are combined with other branches. Conversely, 90% of the branches remain. The results show the time *after* Level 1, since from the beginning to Level 1 we do the normal two scans. We can see that the operational time changes as soon as the branches shrink. Therefore we can conclude that $|B|$ is another major factor

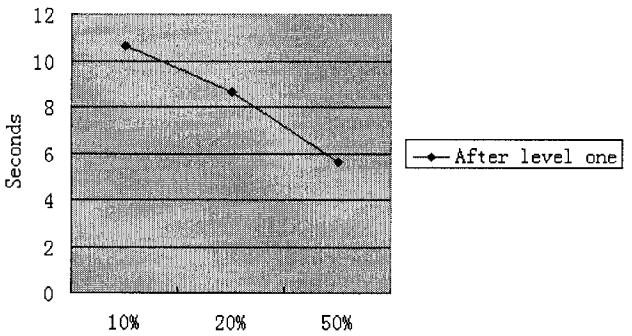

Figure 4.12: Time changes based on branch numbers.

affecting FPM-B. As can be readily seen, this same factor will effect FPM-T in a similar way since they both re-use the existing fp-tree structures. Please note that since this test data set is relatively small, we can expect more prominent changes as the size of the data set goes up. Also please note that right now we can only set the parameter $|B|$ at level one due to a technology limitation. If we could control $|B|$s for all the levels, we would likely see an even more obvious effect upon output time.

4.5 Transaction size

4.5.1 FPM-B

What will the performance of FPM-B be when it operates on bigger data sets? Let us first create the test data sets shown in Table 4.3. Compared with the data sets created earlier, we increase the transactions to 500,000 lines in D4, 1,000,000 lines in D5 and 1,500,000 lines in D6. We set the average length per transaction to be 10. The percentage of branches at the bottom level tree that are combined together to the next level is set to 20%. For each higher level node, it will have 5 corresponding sub-level nodes.

The results of FPM-B versus the benchmark on the above three data sets are shown in Figure 4.13. In this figure we have two curves, with the upper one being the total running time of the benchmark while the lower one represents the time for FPM-B. As we see with

Data set	$\|T\|$	$\|I\|$	$\|L\|$	$\|B\|$	$\|F\|$
D4	500000	10	6	20	5
D5	100000	10	6	20	5
D6	1500000	10	6	20	5

Table 4.3: Bigger data sets for FPM-B

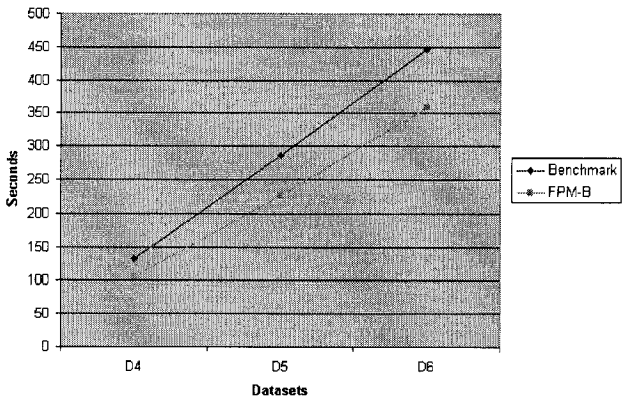

Figure 4.13: FPM-B: Total running time on bigger data sets.

the benchmark, when the size of the data set increases, the running time also increases correspondingly, in a linear manner. With FPM-B, its time also increases linearly as the data set grows. Comparatively, however, the performance of FPM always surpasses the benchmark.

Each of the curves in the figure actually includes two distinct time contributions. The first one is associated with tree construction and the second one with tree mining. Both of these greatly influence the final results. As one can imagine, if at each level the mining process costs significantly more than the tree construction process, we can not expect a big impact based upon reduced tree construction costs. In reality, most of the time this is not the case. This is why we focus on tree building methods. However, we need to be aware that in Figure 4.13, the results do not fully illustrate all the advantages of FPM-B over the benchmark. Specifically, the mining time is the same in both cases and is included in the results. A better illustration can be found in Figure 4.15, which we will discuss later.

In terms of the mining itself, this will be affected by the threshold we set, as well as the size of itemsets to be mined. At lower levels, we are likely to see more items being mined. Therefore, we can expect the mining time contribution in lower levels to be more than that at higher levels.

A detailed version of Figure 4.13 is shown in Figure 4.14, which illustrates how the total time changes from lower levels to higher levels. In this figure, we have three pairs of curves. In each pair, the upper one shows the total running time of the benchmark algorithm while the lower one displays the time for FPM-B. At the first three levels, the time difference between FPM-B and the benchmark is not very apparent. This is because at the first level both of the algorithms employ a similar process — scanning the data set, building header tables for all the levels, and building the first level fp-tree. At the second level, although we use less time to construct the tree by scanning the fp-tree bottom up, the number of items is still quite large. In addition, the benchmark may have fewer items than FPM-B. So for the total time of the second level, the difference between the two algorithms is still not significant. At the third level, FPM-B begins to use less time than the benchmark to construct the relevant fp-tree. However, in this test the mining process consumes a lot of time so the time difference is not as prominent. Starting from level four, as we use less and less time in FPM-B to construct a new fp-tree, the advantage begins to become more pronounced.

As we have discussed, Figure 4.15 is a better representation of how FPM-B can improve the multi-level fp-tree creation process. This figure shows the pure total tree construction time comparison of FPM-B and the benchmark. The result shows that both methods have a linear style running time, which is similar to Figure 4.13. However, putting aside the mining process, FPM-B is significantly better at building fp-trees than the benchmark, with an

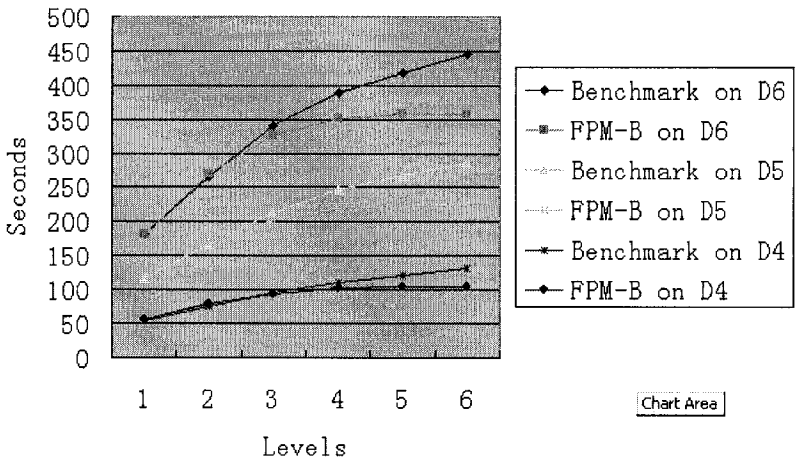

Figure 4.14: FPM-B: Detailed total running time on bigger data sets.

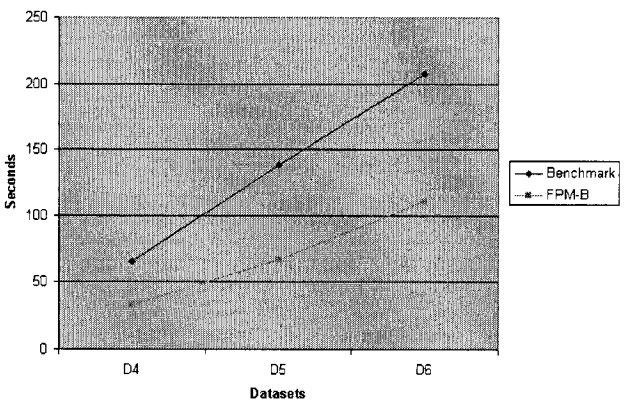

Figure 4.15: FPM-B: Total tree construction time

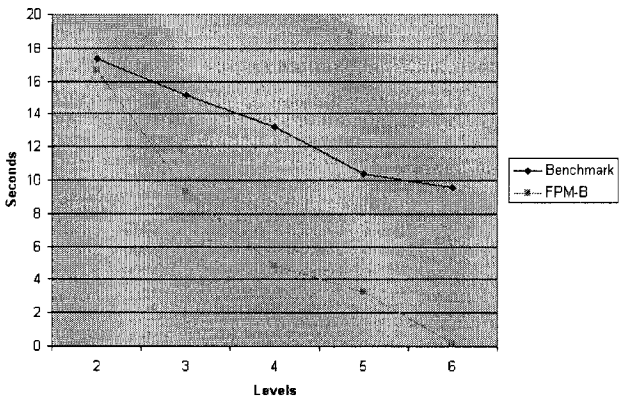

Figure 4.16: FPM-B: Tree construction time of each level on D4

estimated 50% less time in total. While on the other hand, for the whole process, including mining and tree generation, FPM-B takes about 20% less.

Figure 4.16, Figure 4.17 and Figure 4.18 demonstrate detailed tree construction time comparison of the two algorithms as the number of transaction lines increases from 500,000 to 1,500,000. Besides the constant I/O cost, the time reduction for the benchmark is essentially due to the decrease in the number of items in the transactions. When there are fewer items, fewer node insertions into the fp-trees are needed. The same situation applies to FPM-B. What is different is that, with FPM-B, we use a previous level fp-tree, which has good structure and is more compact. Therefore, we need much less time for insertions into the trees. The time drops significantly from Level 2 to Level 3 as the item count drops off. After Level 3, the time curve continues to demonstrate a steady reduction.

4.5.2 FPM-T

In order to test on bigger data sets, we created the data sets shown in Table 4.4. Similar to the data sets in Table 4.3, we only change the number of transactions. Please note, since FPM-B performs better than FPM-T in longer transaction data sets, we will only test data sets of short transactions for FPM-T against the benchmark. Typically, $|I|$ is set to three in

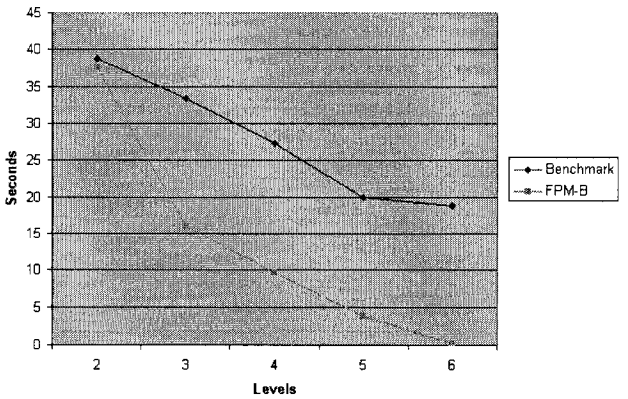

Figure 4.17: FPM-B: Tree construction time of each level on D5

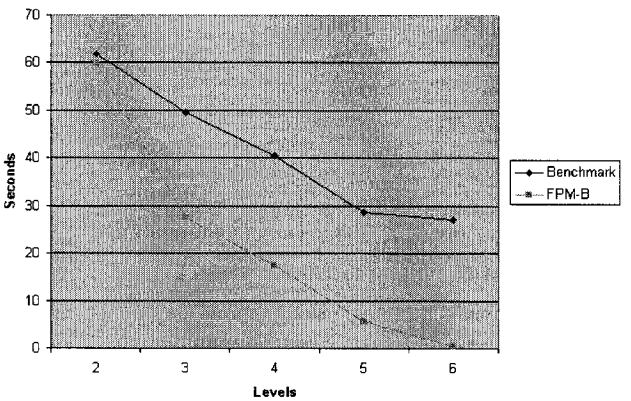

Figure 4.18: FPM-B: Tree construction time of each level on D6

Data set	$\|T\|$	$\|I\|$	$\|L\|$	$\|B\|$	$\|F\|$
D7	500000	3	6	20	5
D8	100000	3	6	20	5
D9	1500000	3	6	20	5

Table 4.4: Bigger data sets for FPM-T

this case.

We perform a similar group of tests for FPM-T on the above data sets. The results for total running time are shown in Figure 4.19. As we can see, on these data sets, FPM-T matches the linear time pattern of the benchmark. But the slope of the curve is smaller than that of the benchmark. Moreover, Figure 4.20 produces a more detailed analysis. In this figure, FPM-T maintains an advantage on all levels over the benchmark, and the advantage grows as we add levels. The time difference is caused by two factors. The first one is the idea of re-using existing fp-trees, the main feature of FPM-T. As discussed, when the transaction is relatively short, we do not need to pay a lot in resizing the structure of the trees. As a result, we get very good running times during fp-tree construction. In Figure 4.22, Figure 4.23 and Figure 4.24, we see the comparative results when FPM-T and the benchmark are operating on the data sets D7, D8 and D9 respectively. These results demonstrate that no matter how great the size of such data sets becomes, the cost to rebuild the next level fp-trees from the previous level is still relatively small. The second factor is related to the fact that we also saved time in header table creation, as discussed earlier. These results can be found in Figure 4.21.

4.6 Increasing the number of levels

After demonstrating that the two algorithms FPM-B and FPM-T work well with both small and big data sets, let us examine if they can also maintain good performance when the

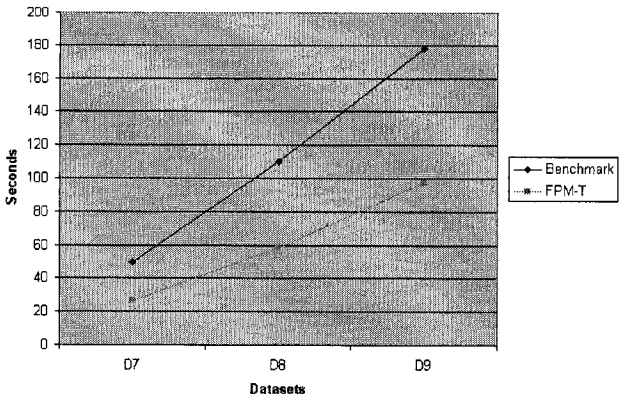

Figure 4.19: FPM-T: Total running time on bigger data sets.

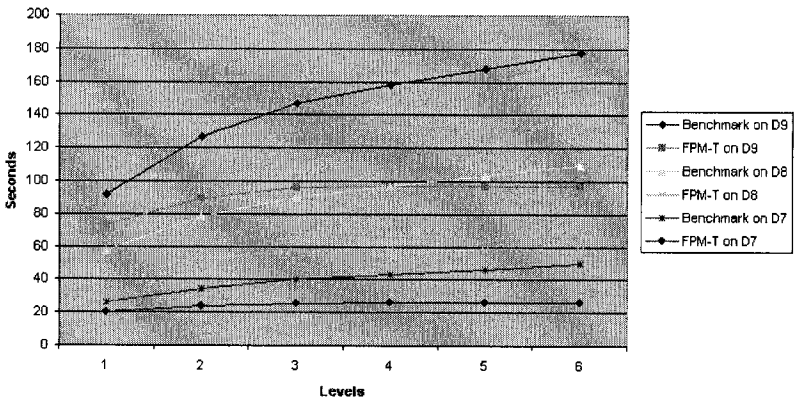

Figure 4.20: FPM-T: Detailed total running time on bigger data sets.

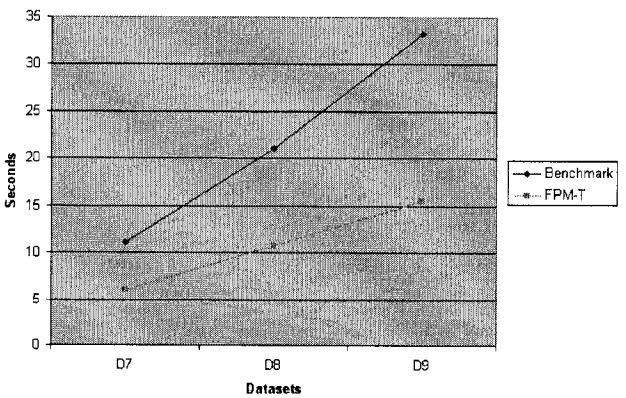

Figure 4.21: FPM-T: Cost in building header tables.

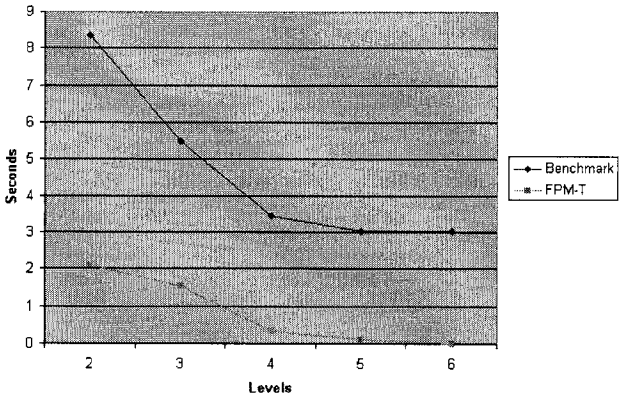

Figure 4.22: FPM-T: Tree construction time of each level on D7

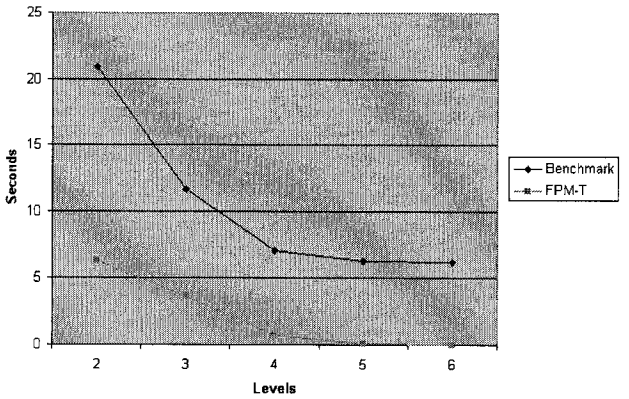

Figure 4.23: FPM-T: Tree construction time of each level on D8

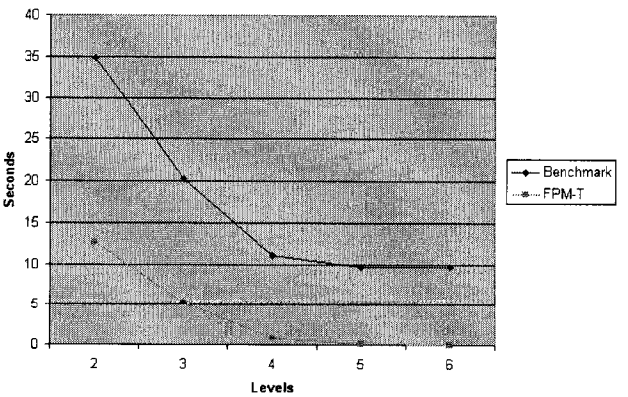

Figure 4.24: FPM-T: Tree construction time of each level on D9

Data set	$\|T\|$	$\|I\|$	$\|L\|$	$\|B\|$	$\|F\|$
D10	1000000	10	9	20	3
D11	1000000	3	9	20	3

Table 4.5: Data sets with more levels

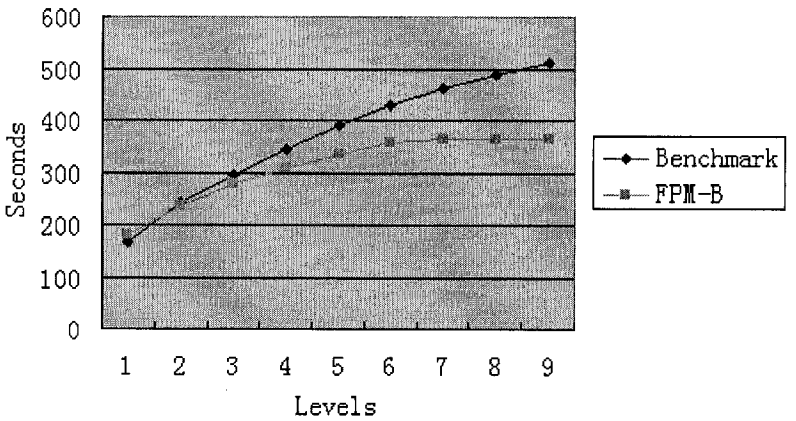

Figure 4.25: FPM-B: Total running time on data sets of more levels

number of levels in the concept hierarchy increases.

Similar to the data generation process discussed earlier, we first create a group of data sets for testing FPM-B and FPM-T. These are shown in Table 4.5. Both of the data sets contain 9 levels and 1,000,000 lines. The difference between them is the $|I|$ parameter. D10 has long transactions and is used to test FPM-B, while D11 has short transactions and is used to test FPM-T.

4.6.1 FPM-B

Figure 4.25 displays the total running time comparison between FPM-B and the benchmark for the case of nine levels in the concept hierarchy of the data set "D10". As we can see, the operational time of both algorithms increases more gently as the level goes up. However, after a certain point — in the case of this figure, Level 3 — the time for FPM-B grows much more slowly than that of the benchmark. In other words, for more levels, using FPM-B will produce much better performance than the benchmark.

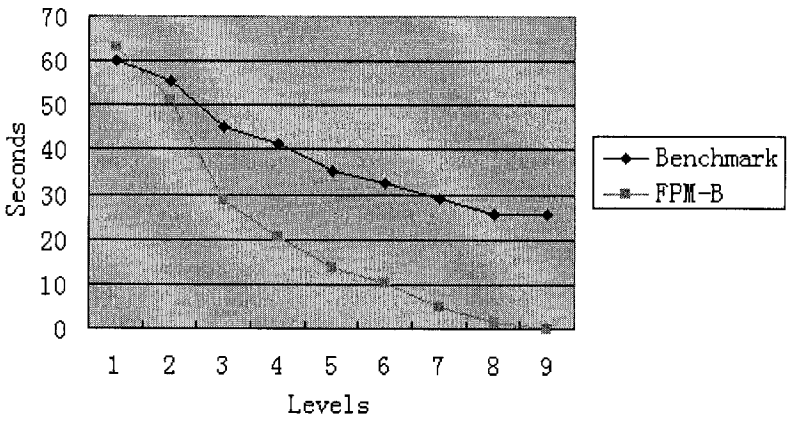

Figure 4.26: FPM-B: Tree construction time on data sets with more levels

Figure 4.26 shows the tree construction time for each level. We find that at the first level, we may need to spend more time in FPM-B. As discussed earlier, this is true since FPM-B may have more items and thus more node insertions into the first level fp-tree. However, as the level goes up, the running time for the FPM-B tree building reduces very quickly. This is because the size of the fp-tree decreases very rapidly in a corresponding manner while, on the other hand, the benchmark can only maintain a slow constant time reduction in the tree construction phase. Therefore, we can conclude that FPM-B is more suitable for data sets with more levels.

4.6.2 FPM-T

Figure 4.27 displays the test results of FPM-T and the benchmark on a nine level data set "D11". The result represents the total operational time up to each level. As we can see, FPM-T performs much better than the benchmark, especially at higher levels. Given transactions with short length, FPM-T minimizes the time to resize the fp-trees but maximizes the advantages inherent in header table creation. The two factors accelerate FPM-T at every stage. Figure 4.28 shows the time for tree-construction for all the levels on D11. Although we may fall behind at the beginning level due to having more nodes than the benchmark, we

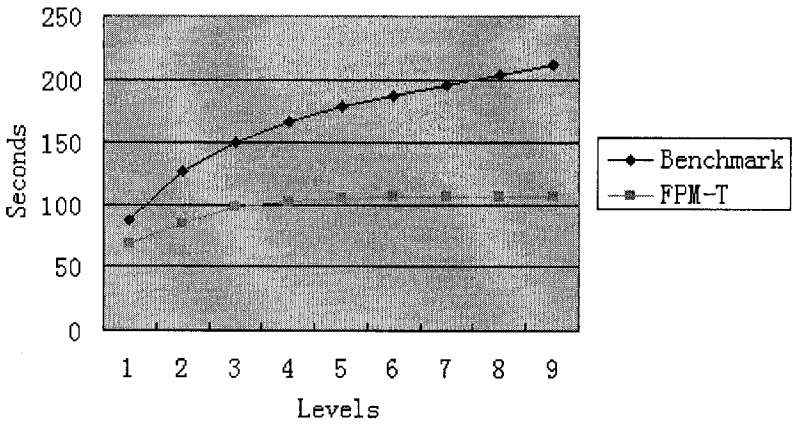

Figure 4.27: FPM-T: Total running time on data sets with more levels

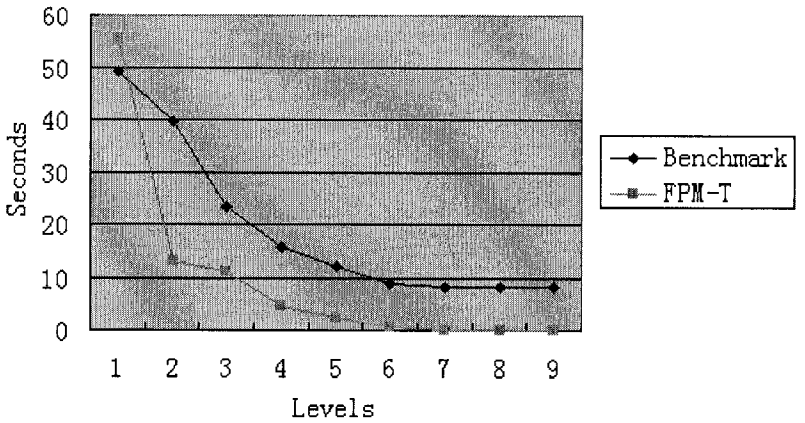

Figure 4.28: FPM-T: Tree construction time on data sets with more levels

quickly overtake it by the second level. What is more, we gain greater advantage as we move up through the levels. We can also notice by comparing FPM-T and FPM-B that FPM-T has extremely good performance in such circumstances.

4.7 Other performance factors

4.7.1 Dense versus sparse data

The benchmark provides a filtering process to filter out all non-frequent items at the header table of each level, while FPM-B and FPM-T can only perform this process at the top level according to the threshold or minimum support. If a majority of the items are filtered, the

new algorithms will not have great advantages over the benchmark. For example, if 90% of the items in the transaction data set are infrequent, it may not be helpful to choose the new algorithms that we propose. In reality, this is rarely the case. Thus, when there are not many items filtered out, FPM-B and FPM-T provide much better performance over the benchmark. In other words, when processing a dense data set, we keep most of the items at most times and we can therefore expect the new algorithms to work well. With sparse sets, where we can easily filter out a lot of items, it may be better to simply use the benchmark.

4.7.2 Change of thresholds

Thresholds can change the performance of the algorithms as well. In the tests listed above, we set the thresholds in a practical way. In the tests of the benchmark, we expect to filter out about 25% to 50% of the distinct items at the starting levels. Because of the features of dense data sets, we do not filter out the majority of the items. Generally speaking, our new algorithms offer greater advantages when the minimum support is set lower, but also maintain impressive performance when the threshold goes up but still remains within practical limits.

4.7.3 Memory consumption

When we use the new algorithms, FPM-B and FPM-T, we have to scan the previous level fp-tree. Typically the fp-tree will have to be loaded into memory. Therefore, when we finish constructing the new fp-tree, we have two fp-trees in memory at a time. With the benchmark, on the other hand, we need to keep the data set and the fp-tree being constructed in memory. In [GZ05], Grahne et al. pointed out that when using FP-growth, an fp-tree is expected to occupy more memory than the original data set. Their memory consumption can sometimes be several times larger than that of the data set. In this case, FPM-B and

FPM-T are expected to use more memory than the benchmark.

4.8 Cross-level mining

After examining algorithms for multi-level frequent pattern mining (MLFPM), let us check in with cross-level frequent pattern mining (CLFPM). We select the data set D4 to test the algorithm FPM-Cross. Figure 4.29 shows the comparative results for FPM-Cross and the benchmark. The coordinate at Level 1 of FPM-Cross represents the time to build the cross-level fp-tree from Level 1 to the topmost level. From the figure we can see that the benchmark algorithm reduces running time as we move the starting base level to a higher level. This is because we will have fewer items to be inserted into the cross-level fp-tree. FPM-Cross demonstrates a similar time reduction pattern relative to the benchmark, but has better overall results than the benchmark. This is primarily due to the fact that instead of scanning the data set, we scan the fp-tree to generate a new cross-level fp-tree, which saves a certain amount of I/O cost. What is different from the MLFPM fp-tree construction is that, in FPM-Cross we have exactly the same items as the benchmark. We no longer have the disadvantage of more items, especially at the beginning levels.

We can also see that we improve as the base level goes up, since at lower levels, it is likely that we will have more items, while at higher levels we only have a few items. Consequently, we can take advantage of the compressed fp-tree structure when we start to build the cross-level fp-tree at higher levels. The difference between FPM-Cross and the benchmark with a base level at Level 1 is about 12.6% . When we extend the base level to Level 5, however, we see an advantage of 93.3 %. Since the key performance factors are simpler in FPM-Cross than in FPM-B and FPM-T, we can expect FPM-Cross will also work better than the benchmark with larger data size or data sets with more levels.

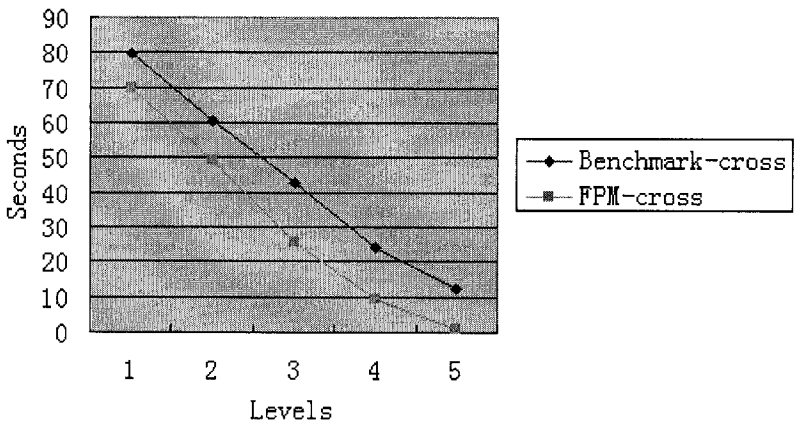

Figure 4.29: FPM-Cross: Tree construction time of cross level fp-tree

4.9 Conclusion

In this chapter, we have carried out a significant number of experiments in order to test our proposed algorithms FPM-B and FPM-T against the benchmark. The results from those tests have shown that FPM-T works best with shorter transactions while FPM-B provides the best solution for longer transactions. Both of the algorithms maintain advantages over the benchmark in specific environments. We have also shown that FPM-Cross can work effectively relative to a simple benchmark in the context of cross level mining.

Chapter 5

Conclusions and future work

5.1 Summary

In the case of single level frequent pattern mining (SLFPM), FP-growth has proven to be a potent method for mining association rules. However, efficient methods for the exploration of multi-level frequent pattern mining (MLFPM) have not yet been well developed. Given a concept hierarchy that contains a number of levels, simply applying FP-growth to each level may result in excessive scanning costs. In fact, this method that repeatedly scans the raw data set serves as the benchmark in our research. Observe that as the levels go from low to high, the sizes of fp-trees shrink as well, with the structure of fp-trees becoming less complex. Our new approaches for mining MLFPM rules exploit this feature and avoid unnecessary costs relative to the benchmark. In addition, we also propose a new method for cross-level frequent pattern mining (CLFPM). All of these algorithms make use of the existing fp-tree structure, and create new fp-trees based on old fp-trees. Experimental results have shown that all of our algorithms are superior to their benchmark in certain environments.

To summarize, we have focused on the following areas:

1. FPM-B: This is an algorithm that adopts a bottom-up traversal approach. It first accesses all leaf nodes of the previous level fp-tree, collecting item information for

the accessed branch, generating items for the next level, filtering out duplicated items and inserting useful items into the next level fp-tree. FPM-B works better than the benchmark in all cases when the data set is dense. It also maintains good scalability in bigger data sets or in data sets with more levels.

2. FPM-T: This is an algorithm that uses a top-down traversal. It begins by scanning the root node (Root) of the previous level fp-tree. When visiting nodes from the previous level, it creates corresponding nodes for the next level. It also filters out duplicated nodes in cases where this is necessary. The algorithm terminates when all the nodes are visited. Since the new fp-tree keeps the structure of the previous level tree, this method preserves complete data set information. Because the order of nodes at difference levels varies, we have to add functions for such operations as re-ordering and combining to ensure that the nodes maintain the right order so that prefix sharing is maximized. FPM-T works better than FPM-B and the benchmark when the transaction length is short, regardless of whether the data set is small or big.

3. FPM-Cross: This algorithm extends the idea of using existing fp-trees to create new fp-trees, and applies specifically to the domain of CLFPM. It traverses the base level fp-tree bottom-up, generating nodes for all levels and inserting them into the cross level fp-tree. By using a group of thresholds, this method can make the new tree fairly shallow. Test results showed that FPM-cross maintains a certain degree of advantage over its benchmark. Moreover, the advantage becomes even greater when we set the base level to higher levels.

5.2 Future work

The research described in this thesis represents fundamental solutions for generating multi-level frequent pattern rules in MLFPM. Moreover, we can also extend the functionality of our current design to work in different environments. Therefore, our expectation is that we may now turn to new research projects or themes. These possibilities can be described as follows:

- Very big data sets. In our current research, the data sets generated are still relatively small. For example, a data set with 1,500,000 transaction lines and length of 10 in our tests has a size of about 100 megabytes. It does not take much time to read and scan the data set; thus, the effect of the I/O cost here is not very remarkable. In commercial environments, data sets can become very big so that memory is unable to hold them in their entirety. We would then need to use partitions to divide data sets. In this case, I/O costs would increase significantly and have a much more negative impact on the final results and we would expect our new approaches to perform much better than the benchmark.

- Fp-tree compression. Right now, we have to keep the fp-trees to be processed in memory. But holding all of these trees takes a lot of system resources. So it would be advantageous if we could represent the fp-trees in more compact ways. This kind of compression is important if we want to increase the efficiency of the algorithms we have proposed.

- Multi-dimensional. While our present focus is multi-level mining, the issue of *multi-dimensional mining* also represents a meaningful objective. In "multi-level", we pay

attention to vertical relationships amongst levels, while in "multi-dimensional", we target horizontal relationships. Working on frequent patterns between dimensions such as "brands", "sorts" and "names" can result in another group of interesting rules. Hence it might be worth applying our new approaches in this domain as well.

- Quantitative rules. We assume boolean rules for mining in the current research program. However, quantitative rules can be another target. As mentioned in Chapter 2, in this case, we need to deal with numeric values. By considering the numeric attribute of every item, and modifying the methods to generate new items for the next level, as well making other required changes, we may be able to apply our new algorithms to the field of mining quantitative rules.

5.3 Final thoughts

The proposed set of novel algorithms, FPM-B, FPM-T and FPM-Cross, provide us with an efficient way to explore frequent patterns at multiple levels and across levels. In contrast, the naive "benchmarks" spend too much time in excessively scanning the raw data set. By taking advantage of existing fp-tree structures, we have explained how the performance can be significantly improved. Experimental results have shown that our research will be a valuable contribution to the literature in this domain.

Bibliography

[AIS93] R. Agrawal, T. Imilienski, and A. Swami. Mining association rules between sets of items in large databases. *In: Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 207–216, 1993.

[AL99] Y. Aumann and Y. Lindell. A statistical theory for quantitative association rules. *In: Proceeding of the 1999 international conference on knowledge discovery and data mining*, pages 261–270, 1999.

[AS94] R. Agrawal and R. Srikant. Fast algorithm for mining association rules in large databases. *In: Proceedings of 1994 International Conference on VLDB*, pages 487–499, 1994.

[Bay98] J. Bayardo. Efficiently mining long patterns from databases. *In: Proceedings of the 1998 ACM-SIGMOD International Conference on Management of Data*, pages 85–93, 1998.

[BCG01] D. Burdick, M. Calimlim, and J. Gehrke. Mafia: A maximal frequent itemset algorithm for transactional databases. *In: Proceedings of International Conference on Data Engineering*, pages 443–452, 2001.

[BMUT97] S. Brin, R. Motwani, J. Ullman, and S. Tsur. Dynamic itemset counting and implication rules for market basket data. *In: Proceedings ACM SIGMOD International Conference on Management of Data*, pages 255–264, 1997.

[CHNW96] D.W. Cheung, J. Han, V. Ng, and C.Y. Wong. Maintenance of discovered association rules in large an incremental updating technique. *In: Proceeding of the 1996 international conference on data engineering (VLDB'96)*, pages 106–114, 1996.

[Eav03] T. Eavis. Parallel relational olap. *PhD thesis*, 2003.

[Ede05] A. Edelstein. *Introduction to Data Mining and Knowledge Discovery.* Two Crows Corporation, 2005.

[FIM03] *Workshop on Frequent Itemset Mining Implementations.* 2003.

[Fin94] P.N. Finlay. *Introducing decision support systems.* Blackwell Publishers, 1994.

[FPSM92] W. Frawley, G. Piatetsky-Shapiro, and C. Matheus. Knowledge discovery in databases: An overview. *AI Magazine*, 1992.

[GLW00] G. Grahne, LVS Lakshmanan, and X. Wang. Efficient mining of constrained correlated sets. *In: Proceeding of the 2000 international conference on data engineering*, pages 512–521, 2000.

[GZ01] K. Gouda and J. Zaki. Efficiently mining maximal frequent itemsets. *In: Proceedings of International Conference on Data Engineering*, pages 163–170, 2001.

[GZ03a] G. Grahne and J. Zhu. Efficiently using prefix-trees in mining frequent itemsets. *In: Proceeding of the ICDM'03 international workshop on frequent itemset mining implementations (FIMI'03)*, pages 123–132, 2003.

[GZ03b] G. Grahne and J. Zhu. High performance mining of maximal frequent itemsets. *In SIAM '03 Workshop on High Performance Data Mining: Pervasive and Data Stream Mining*, 2003.

[GZ05] G. Grahne and J. Zhu. Fast algorithms for frequent itemset mining using fp-trees. *IEEE transactions on kowledge and data engineering*, pages 1347–1362, 2005.

[HF95] J. Han and Y. Fu. Discovery of multiple-level association rules from large databases. *21st VLDB conference Zurich, Swizerland proceedings*, pages 420–431, 1995.

[HK06] J. Han and M. Kamber. *Data mining: concepts and techniques, 2nd edn.* Morgan Kaufmann, 2006.

[HMS01] D. Hand, H. Mannila, and P. Smyth. *Principles of Data Mining.* MIT Press, 2001.

[HPY00] J. Han, J. Pei, and Y. Yin. Mining frequent patterns without candidate generation. *In: Proceeding of the 2000 ACM-SIGMOD international conference on management of data (SIGMOD'00)*, pages 1–12, 2000.

[JH07] D. Jie and M. Han. Bittablefi: An efficient mining frequent itemsets algorithm. *Knowledge-Based Systems*, pages 329–335, 2007.

[KHC97] M. Kamber, J. Han, and J.Y. Chiang. Metarule-guidedmining ofmulti-dimensional association rules using data cubes. *In: Proceeding of the 1997 international conference on knowledge discovery and data mining (KDD'97)*, pages 207–210, 1997.

[LPS+06] J. Liu, S. Paulsen, X. Sun, W. Wang, A. Nobel, and J. Prins. Mining approximate frequent itemsets in the presence of noise. *In: Proceeding of the 2006 SIAM international conference on data mining*, 2006.

[MY97] RJ. Miller and Y. Yang. Association rules over interval data. *In: Proceedings of ACM SIGMOD*, pages 452–461, 1997.

[PBTL99] N. Pasquier, Y. Bastide, R. Taouil, and L. Lakhal. Discovering frequent closed itemsets for association rules. *In: Proceeding of the 7th international conference on database theory*, pages 398–416, 1999.

[PCT+03] F. Pan, G. Cong, A.K.H. Tung, J. Yang, and M. Zaki. Carpenter: Finding closed patterns in long biological datasets. *In: Proceeding of the 2003 ACMSIGKDD international conference on knowledge discovery and data mining (KDD'03)*, pages 637–646, 2003.

[PCY95] J.S. Park, M.S. Chen, and P.S. Yu. An effective hash-based algorithm for mining association rules. *In: Proceeding of the 1995 ACM-SIGMOD international conference on management of data (SIGMOD'95)*, pages 175–186, 1995.

[PHL01] J. Pei, J. Han, and LVS Lakshmanan. Mining frequent itemsets with convertible constraints. *In: Proceeding of the 2001 international conference on data engineering*, pages 433–442, 2001.

[PHM00] J. Pei, J. Han, and R. Mao. Closet: An efficient algorithm for mining frequent closed itemsets. *ACM SIGMOD'00 Workshop on Research Issues in Data Mining and Knowledge Discovery*, pages 21–30, 2000.

[PHMA+04] J. Pei, J. Han, B. Mortazavi-Asl, J. Wang, H. Pinto, Q. Chen, U. Dayal, and M-C Hsu. Mining sequential patterns by pattern-growth: the prefixspan approach. *IEEE TransKnowl Data Eng*, pages 1424–1440, 2004.

[SA95] R. Srikant and R. Agrawal. Mining generalized association rules. *In: Proceeding of the 1995 international conference on very large data bases (VLDB'95)*, pages 407–419, 1995.

[SA96a] R. Srikant and R. Agrawal. Mining quanlitative association rules in large relationaltables. *In: Proceedings of ACM SIGMOD*, pages 1–12, 1996.

[SA96b] R. Srikant and R. Agrawal. Mining sequential patterns: generalizations and performance improvements. *In: Proceeding of the 5th international conference on extending database technology*, pages 3–17, 1996.

[SON95] A. Savasere, E. Omiecinski, and S. Navathe. An efficient algorithm for mining association rules in large databases. *In: Proceeding of the 1995 international conference on very large data bases (VLDB'95)*, pages 432–443, 1995.

[Toi96] H. Toivonen. Sampling large databases for association rules. *In: Proceeding of the 22nd VLDB conference*, pages 134–145, 1996.

[WHP03] J. Wang, J. Han, and J. Pei. Closet+: Searching for the best strategies for mining frequent closed itemsets. *In: Proceedings of ACM SIGKDD'03*, pages 236–245, 2003.

[Zak00] J. Zaki. Scalable algorithms for association mining. *IEEE transactions on knowledge and data engineering*, pages 372–390, 2000.

[ZH02] J. Zaki and C. Hsiao. Charm: An efficient algorithm for closed itemset mining. *In: Proceeding of SIAM international conference on Data Mining*, pages 457–473, 2002.

[ZPT04] H. Zhang, B. Padmanabhan, and A. Tuzhilin. On the discovery of significant statistical quantitative rules. *In: Proceeding of the 2004 international conference on knowledge discovery and data mining*, pages 374–383, 2004.

[ZYH+07] F. Zhu, X. Yan, J. Han, P.S. Yu, and H. Cheng. Mining colossal frequent patterns by core pattern fusion. *In: Proceeding of the 2007 international conference on data engineering (ICDE07)*, pages 706–715, 2007.

Appendix A

Closed frequent itemsets and maximum frequent itemsets

A.1 Closed frequent itemsets

An issue of some importance in FPM is that we sometimes generate too many frequent patterns. This is essentially because if a pattern is frequent, then any of its sub-patterns will be frequent as well. If we set the minimum support too low, the pattern could be very long. Thus the problem becomes even worse. For example, if a simple pattern $I_1, I_2, I_3 ..... I_{100}$ is frequent, the total number of patterns we can get is $100^1 + 100^2 + 100^3 + \ldots + 100^{100} = 2^{100} - 1 \doteq 1.2 \times 10^{30}$. This exponential number of frequent patterns is clearly not acceptable for research or business purposes.

One approach for reducing the number of frequent patterns is to apply the idea of *closed frequent itemsets* (CFI), originally described by Pasquier et al. in 1999 [PBTL99].

Definition 9. *Closed frequent itemset: An itemset I is closed if I is frequent and none of its super-patterns has the same support as I.*

For example, say we have a database with two transactions $\{I_1 ... I_{50}\}, \{I_1 ... I_{100}\}$ with minimum support = 1. Now, instead of generating an exponential number of patterns, we get a simple CFI pattern that is $\{I_1, I_2 ... I_{50}\} : 2, \{I_1, I_2 ... I_{100}\} : 1$.

Since the original proposal, significant improvements have been made by other researchers.

Zaki et al. presented an algorithm called CHARM [ZH02]. In this case, each itemset has a corresponding TID-array. By manipulating the TID-array, we get the desired closed frequent patterns. Pei et al. extended FP-growth and created an algorithm called CLOSET for CFI mining [PHM00]. Later, they extended this method and proposed CLOSET+, which uses a global tree to check all the closed frequent itemsets [WHP03]. Grahne et al. introduced a fast algorithm called FPclose and showed it was faster than other proposed algorithms at that time [GZ03a]. We discuss this algorithm in detail below.

FPclose uses a data structure similar to the FP-tree called CFI-tree. A key difference is that, besides having links to parent, siblings and children, as well as item-name and node-link, each node in a CFI-tree also has a different count and a new level field. The count here does not keep the count of the current node. Instead, it maintains a running count. The field "level" is used for subset checking. The algorithm is shown in Algorithm 14.

We observe that a recently discovered CFI pattern can only be a subset of the already discovered CFI patterns; otherwise, it is a new CFI pattern. Let us use CFI_I for the closed frequent pattern tree of an itemset I. We maintain this tree globally in order to grow branches that contain already discovered CFI patterns of the itemset I, as well as counts of its members. If we find a new pattern that contain I, we only need to compare it with CFI_I to see if it is a subset of the latter. However, if the tested data set is quite big, it may generate very long closed frequent patterns, thus making the global CFI quite big. In order to deal with this situation, we create a new structure called *local CFI*, denoted as LCFI. For each conditional fp-tree T_I of an item I, we include a $LCFI_I$. $LCFI_I$ contains all CFIs in the conditional pattern base of I. Patterns are determined to be frequent or not inside the LCFI. Also, by checking a newly generated local CFI with $LCFI_I$, fewer comparisons need to be performed. Thus we can significantly accelerate the entire process.

Algorithm 14 The algorithm of FPclose: FPclose(T,C)

Input: an fp-tree T, an empty CFI-tree C.
Output: updated C, which will contain all newly discovered CFIs.

```
1: if T is a single path tree then
2:   generate all CFIs from T
3:   for every new generated CFI N do
4:     call subset-checking(N,C)
5:     if false then
6:       insert N into C
7:     end if
8:   end for
9: else
10:  for every item I in the header table of T do
11:    get I's conditional pattern base CPB_I
12:    call subset-checking(CPB_I,C)
13:    if false then
14:      construct I's conditional fp-tree T_I
15:      initial I's conditional CFI-tree C_I
16:      call FPclose(T_I,C_I)
17:    end if
18:  end for
19:  merge C_I with C
20: end if
```

A pruning technique during subset checking can also be applied by using the field "level" as discussed above. Every node in the CFI-tree has a level field. For example, let us say we have a branch from the leaf nodes to the nodes closest to the Root node: "D-C-B-A". In this case, "A"s level value is 1, while the corresponding level values for "B,C,D" are 2, 3, and 4, respectively. If we find a new pattern "D-C-B-A-E", we note that the level of the node "D" is 5, which is larger than that of the node "D" in the pattern "D-C-B-A". Thus, it is not possible for the former pattern to become a subset of the latter one. In this case we can stop the subset checking process immediately.

Let us look at an example. Suppose we have the sample tree shown in Figure A.1. As per Algorithm 14, we begin with an empty CFI tree. Clearly the fp-tree is not a single path tree. So we start from the least frequent item in the header table; that is, "21". As

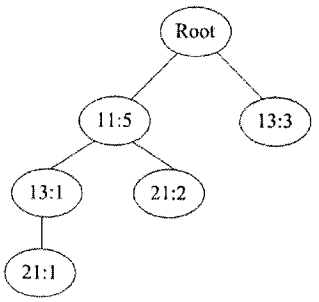

Figure A.1: A sample tree

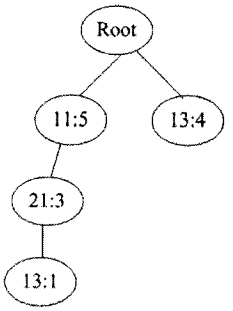

Figure A.2: A sample CFI-tree

we visit nodes, we find the first pattern: "21-13-11:1". Since this is not a subset of any existing CFI, we create three corresponding nodes and insert them as a new branch in the CFI_I. The second pattern is "21-11:3". Again, this is new. In terms of the single item pattern "21:3", we use subset checking and find that it has the superset pattern "21-11:3". Thus, it is not inserted. Then we move up to item "13". We find that the first associated pattern is "13-11:1". However, this is also a subset of the pattern "21-13-11:1". Thus, it is ignored. Next, the pattern with the simple item "13:4" is inserted. The pattern for item "11", which is "11:5", is also inserted. At the end of the process, the updated CFI tree resembles Figure A.2. The final result of CFI processing is therefore "21-11:3; 21-11-13:1; 13:4; 11:5".

A.2 Maximum frequent itemsets

Generally speaking, the number of CFIs is much less than the number of FIs. However, sometimes it is still more than the user expects. For this reason, users may turn to another kind of frequent pattern called *maximum frequent itemsets* (MFI). This idea was first suggested by Bayardo in 1998 [Bay98]. He presented an algorithm called MaxMiner that is an extension of Apriori. By searching only for long patterns, MaxMiner reduces the search space greatly, although it may still require frequent scans of the data set.

Definition 10. *Maximum frequent itemset: An itemset I is a max-pattern if I is frequent and no super-patterns of I exists.*

Using the running example, we can also define the resulting MFI patterns as $\{I_1, I_2, I_3...I_{100}\}$: 1. As we can see, in this case the pattern is even shorter than is the case for CFI.

Other popular algorithms include MAFIA, GENMAX and FPmax. MAFIA, described by Burdick et al. [BCG01], uses a bit-vector to represent an itemset. If the itemset occurs in a transaction, its bit-vector is updated. In this way, operations can be performed directly on bit-vectors in order to improve efficiency. GENMAX, defined by Gouda et al. [GZ01], uses a strategy called *progressive focusing*. The algorithm in this case makes use of local maximum frequent items (LMFI) instead of comparing newly found MFIs with global MFIs.

FPmax, presented by Grahne et al. [GZ03b], uses a similar idea to FPclose. It also uses an extended form of fp-tree, the MFI-tree. Like the fp-tree, each node has links to parent, siblings and children, as well as an item-name, node-link, and a level field used for subset checking. What is different is that it does not have a count field. This is because MFI checking only tends to find maximum frequent patterns. Therefore, count information is useless. The algorithm works in an analogous way to FPclose, but with MFI-specific modifications. It is shown in Algorithm 15.

Algorithm 15 The algorithm of FPmax: FPmax(T,C)

Input: an fp-tree T, an empty MFI-tree M.
Output: updated M, which will contain all newly discovered MFIs.

```
1: if T is a single path tree then
2:   generate all MFIs from T
3:   for every new generated MFI N do
4:     call subset-checking(N,C)
5:     if false then
6:       insert N into M
7:     end if
8:   end for
9: else
10:  for every item I in the header table of T do
11:    get I's conditional pattern base CPB_I
12:    call subset-checking(CPB_I,M)
13:    if false then
14:      construct I's conditional fp-tree T_I
15:      initial I's conditional MFI-tree M_I
16:      call FPmax(T_I,M_I)
17:    end if
18:  end for
19:  merge M_I with M
20: end if
```

By applying the above algorithm to the example in the previous section we may generate the relevant MFI rules. The associated MFI-tree would look like Figure A.3. Since patterns such as "13-11", "21-11", "21", "11" and "13" will all be subsets of "21-13-11", from the MFI point of view, we only get a very simple result. Specifically, we have "21-13-11".

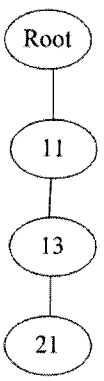

Figure A.3: A sample MFI-tree