SQL Based Frequent Pattern Mining

Dissertation

zur Erlangung des akademischen Grades

Doktoringenieurin (Dr.-Ing.)

vorgelegt der Fakultät für Informatik
der Otto-von-Guericke-Universität Magdeburg

von:	MSc. Xuequn Shang	
geb. am	8. März 1973	in Shaanxi, China

Magdeburg, den 14. Februar 2005

Declaration

I hereby declare that this submission is my own work and to the best of my knowledge it contains no materials previously published or written by another person, nor material which to a substantial extent has been accepted for the award of any other degree or diploma at University of Magdeburg or any other educational institution, except where due acknowledgement is made in the thesis. Any contribution made to the research by others, with whom I have worked at University of Magdeburg or elsewhere, is explicitly acknowledged in the thesis.

I also declare that the intellectual content of this thesis is the product of my own work, except to the extent that assistance from others in the project's design and conception or in style, presentation and linguistic expression is acknowledged.

Zusammenfassung

Data Mining in gross relationalen Datenbanken wird zunehmend eingesetzt und seine Bedeutung ist heute voll anerkannt. Trotzdem fällt die Performanz von SQL-basiertem Data Mining hinter spezialisierten Implementierungen zurück. Dies liegt an den unangemessen hohen Kosten der Wissensextraktion und der fehlenden Unterstützung durch Konstrukte der deklarativen Anfragesprache. Frequent Pattern Mining, d.h. die Suche nach sich wiederholenden Mustern in Daten, ist die Grundlage für eine Reihe von essentiellen Mining-Aufgaben. Dies war die Motivation für die Entwicklung SQL-basierter Ansätze für das Frequent Pattern Mining im Rahmen diese Forschungsvorhabens.

In dieser Arbeit werden Ansätze untersucht, um unter Verwendung von SQL Frequent Patterns in einer Transaktionstabelle zu finden. Von diesen basieren die meisten auf dem Apriori-Algorithmus. Diese Methoden weisen jedoch durch die teuren Operationen zur Kandidatengenerierung und deren -test eine unzureichende Performanz auf, insbesondere bei der Suche nach besonders aussagekräftigen und/oder langen Mustern. Hierfür wurde im hier beschriebenen Dissertationsprojekt eine Klasse von SQL-basierten Methoden zum schrittweisen Finden und Verfeinern von Mustern entwickelt. Die Gemeinsamkeit dieser Methoden besteht im Teile und Herrsche-Ansatz zur Zerlegung von Mining-Aufgaben und in der Anwendung einer Musterverfeinungsmethode zur Vermeidung des kombinatorischen Effekts, der für die Kandidatengenerierung ein typisches Problem darstellt. Apriori-basierte Algorithmen erforderen bei der Verwendung von SQL entweder mehrere Scans über die Datenbank oder aufwändige Verbundoperationen. Demgegenüber vermeiden die hier vorgestellten SQL-basierten Algorithmen mehrere Durchläufe über die Ausgangstabellen als auch die Berechnung komplexer Verbunde zwischen Tabellen.

Eine umfassende Untersuchung der Performanz wurde unter Verwendung eines DBMS (IBM DB2 UDB EEE V8) durchgeführt und die Ergebnisse herkömmlicher Apriori-basierter Ansätze wurden mit denen der in dieser Arbeit vorgestellten Methoden verglichen. Empirische Ergebnisse zeigen, dass die vorgestellten Algorithmen zu einer effizienten Berechnung führen. Darüber hinaus unterstützen die meisten Datenbankmanagementsysteme heutzutage die Parallelisierung, deren Eignung zur Unterstützung des Frequent Pattern Mining im Rahmen dieser Arbeit untersucht wurde.

Abstract

Data mining on large relational databases has gained popularity and its significance is well recognized. However, the performance of SQL based data mining is known to fall behind specialized implementation since the prohibitive nature of the cost associated with extracting knowledge, as well as the lack of suitable declarative query language support. Frequent pattern mining is a foundation of several essential data mining tasks. These facts motivated us to develop original SQL-based approaches for mining frequent patterns.

In this work, we investigate approaches based on SQL for the problem of finding frequent patterns from a transaction table. Most of them adopt *Apriori*-like approaches. However those methods may suffer from the inferior performance since the costly candidate-generation-and-test operation especially when mining datasets with prolific patterns and/or long patterns. We develop a class of efficient SQL based pattern growth methods for mining frequent patterns. The commonality of these approaches is that they use a divide and conquer method to decompose mining tasks and then use a pattern growth method to avoid the combinatory problem inherent to candidate-generation-and-test approach. *Apriori* algorithms with the help of SQL either require several scans over the data or require many and complex joins between the input tables. While our SQL-based algorithms avoid making multiple passes over the large original input table and complex joins between the tables.

A comprehensive performance study evaluates on DBMS (IBM DB2 UDB EEE V8) and compares the performance results between SQL based frequent pattern mining approaches based on *Apriori* and the approaches in this thesis. The empirical results show that our algorithms can get efficient performance. Moreover, recently most major database systems have included capabilities to support parallelization, this thesis examined how efficiently SQL based frequent pattern mining can be parallelized and speeded up using parallel database systems.

Dedication

To my family

Acknowledgements

There are lots of people I would like to thank for a huge variety of reasons.

Firstly, I would like to express my sincere gratitude to my senior supervisor, Prof. Gunter Saake, for his continuous help and support throughout my dissertation and my stay with his group. Prof. Saake always finds time in his busy schedule for attending the group meeting and his creative thinking and insight make our discussions fruitful and interesting. Without his guidance, my endeavors would not have been successful.

I am very thankful to my supervisor, Prof. Kai-Uwe Sattler, for his insightful comments and advice. He always give me continuous encouragement and support, and share with me his knowledge and experience. The discussion with him is very helpful to my research. I really appreciate the effort he put in the development of me and my work and his help to improve the quality of my thesis.

My deepest thanks to Prof. Wolfgang Lehner for serving on my supervisory committee.

I am thankful to Ingolf Geist for his help and suggestions during my initial work on data mining. I would like to say a big 'thank-you' to Dr. Eike Schallehn, Dr. Ingo Schmitt, Hagen Hoepfner for their great help during my study in Magdeburg University. I would also like to thank many other people in our department, support staff and faculty, for helping me in serval ways. In particular I thank Prof. Claus Rautenstrauch, Dr. Soeren Balko, Anke Schneidewind, Qaizar Ali Bamboat, Jubran Rajub, Kerstin Giesswein, Kerstin Lange, and all my colleagues in the database group for their help with everything. Many thanks go to Steffen Thorhauer and Fred Kreutzmann for a well administered research environment. I would like to especially thank Marcel Karnstedt from Technical University Ilmenau, who gave me greatly help in administrating parallel experiment environment.

Last but not least, I am very grateful to my dad Chongxin Shang and mum Yueping Guan, my sister Xuehong Shang and her husband Dr. Xiangru Xu. It is through their continuous moral support and encouragement that I have made it through all the steps to reach this point in life, and I could not have done it without them. My family has always taken care of me and I love them all very much. Special thanks from my heart to my husband Mingjun Mu for his love, understanding and support. He has always been patiently standing beside me during all these years. I hope I will do them proud of my achievements, as I am proud of them. Their love accompanies me forever.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Data Mining

The information revolution is generating mountains of data from sources as diverse as business and science fields. One of the greatest challenges is how to turn these rapidly expending data into accessible, and actionable knowledge.

Data mining is the automated discovery of non-trivial, implicit, previously unknown, and potentially useful information or patterns embedded in databases [FPSM91]. Briefly state, it refers to extracting or mining knowledge from large amounts of data. The motivation for data mining is a suspicion that there might be nuggets of useful information hiding in the masses of unanalyzed or underanalyzed data, and therefore methods for locating interesting information from data would be useful. From the beginning, data mining research has been driven by its applications. While the finance and industries have long recognized the benefits of data mining, data mining techniques can be effectively applied in many areas and can be performed on a variety of data stores, including relational databases, transaction databases and data warehouses.

Many people take data mining as synonym for another popularly used term, Knowledge Discovery in Databases (KDD). Alternatively, others view data mining as simply an essential step in the process of knowledge discovery in databases. The KDD process is depicted in Figure 1.1 [HK00] and consists of an iterative sequence

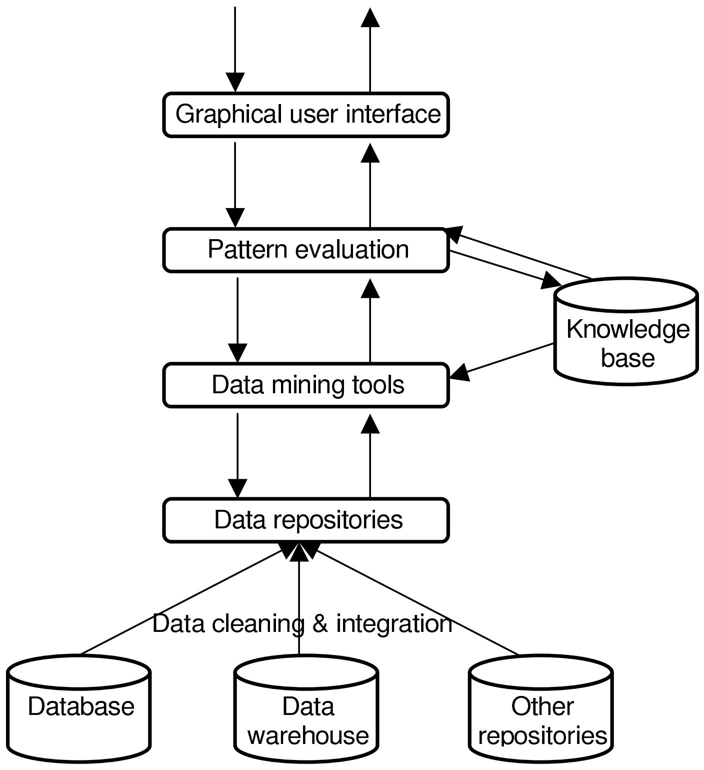


Figure 1.1: Architecture of a typical data mining system

of the following steps:

1. Preprocessing. The process is executed before data mining techniques are applied to the right data. It includes data cleaning, integration, selection and transformation.

2. Data mining process. This is the main process of KDD where intelligent methods are applied in order to extract data patterns.

3. Postprocessing. The process includes pattern evaluation which identify the truly interesting patterns representing knowledge based on some interestingness measures and knowledge presentation where visualization and knowledge representation techniques are used to present the mined knowledge to the user.

1.1.1 Types of data repositories

In principle, data mining should be applicable to any kind of information repository. This includes relational databases, data warehouses, transactional databases, advanced database systems, flat files, and the World Wide Web. Advanced database systems include object-relational and object-oriented databases, and specific application-oriented databases, such as spatial databases, temporal databases, text databases, and multimedia databases. Based on the types of data, the challenges and techniques of mining may differ for each of the repository systems.

1.1.2 Types of mining

Generally speaking, there are two classes of data mining: descriptive and prescriptive. Descriptive mining is to summarize or characterize general properties of data in data repositories, while prescriptive mining is to perform inference on current data, to make predictions based on the historical data.

The initial efforts on data mining research were to cull together techniques from machine learning and statistics to define new mining operations and develop algorithms for them [AGI$^+$92, AIS93, AW97, KI91]. In general, there are many kinds of patterns (knowledge) that can be discovered from data. For example, association rules can be found for market basket or transaction data analysis, classification models can be mined for prediction, clusters can be identified for customer relation management, and outliers can be found for fraud detection. In the remainder of this section, we briefly introduce the various data mining problems with examples.

1.1.2.1 Association rule mining

One of the fundamental methods from the prospering field of data mining is the generation of association rules that describe relationships between items in data sets. The original motivation for searching association rules came from the need to analyze so called supermarket transaction data, that is, to explore customer behavior in terms of purchased products. Association rules describe how often items are purchased together.

Generally speaking, an association rule is an implication

$$X \Rightarrow Y$$

where X and Y are disjunct sets of items. The meaning of such rules is quite intuitive: Let DB be a transaction database, where each transaction $T \in D$ is a set of items. An association rule $X \Rightarrow Y$ then expresses "Whenever a transaction T contains X than this transaction T also contains Y with probability $conf$". The probability $conf$ is called the rule confidence and is supplemented by further quality measures like rule support and interest. The support sup is simply the number of transactions that contain all items in the antecedent and consequent parts of the rule. (The support is sometimes expressed as a percentage of the total number of records in the database.) The confidence $conf$ is the ratio of the number of transactions that contain all items in the consequent as well as the antecedent to the number of transactions that contain all items in the antecedent.

Example 1.1 (*Association rules mining*) *Suppose, we have large number of items, e.g., "bread", "milk." Customers fill their market baskets with some subset of the items, and we get to know what items people buy together, even if we don't know who they are.*

Association rule mining searches for interesting relationship among those items and displays it in a rule form. An association rule "$bread \Rightarrow milk$ ($sup = 2\%, conf = 80\%$)" states that 2% of all the transactions under analysis show that bread and milk are purchased together and 80% of the customers who bought bread also bought milk. Such rules can be useful for decisions concerning product pricing, promotions, sore layout and many things. Association rules are also widely used in various areas such as telecommunication networks, market and risk management, inventory control etc.

How are association rule mined from large databases?

Association rule mining consists of two phases:

- Find all frequent itemsets. By definition, each of these itemsets will occur at least as frequently as a pre-defined minimum support threshold.

- Generate association rules from frequent itemsets. By definition, these rules must satisfy the pre-defined minimum support threshold and minimum confidence threshold.

The second phase is straightforward and less expensive. Therefore the first phase-frequent itemset mining is a crucial step of the two and determines the overall performance of mining association rules.

In addition, frequent itemsets play an essential role in many data mining tasks that try to find interesting patterns from databases, such as association rules [AS94, KMR$^+$94], correlations [BMS97], sequential patterns [AS95], multi-dimensional patterns [KHC97, LSW97], max-patterns [Bay98], partial periodicity [HDY99], emerging patterns [DL99], episodes [MTV97]. Frequent pattern mining techniques also can be extended to solve many other problems, such as iceberg-cube computation [BR99] classifiers [BHM98]. Thus, how to efficiently mine frequent patterns is an important and attracting problem.

1.1.2.2 Sequential Patterns

As we know, data are changing all the time, especially data on the web are highly dynamic. It is obvious that time stamp is an important attribute of each dataset. Sequential pattern mining, which discovers relationships between occurrences of sequential events to find if there exist any specific order of the occurrences, is an important process in data mining with broad applications, including the analyses of customer purchase behavior (Association rule mining does not take time stamp into account, the rule can be $X \Rightarrow Y$. With sequential pattern mining We can get more accurate and useful rules such as: $X \Rightarrow Y$ within a week, or X happens every week.) web access pattern, disease treatments, DNA sequences, and so on.

The sequential pattern mining problem was first introduced in Agrawal and Srikant [AS95] and further generalized in Srikant and Agrawal [SA96]. Given a database of sequence, where each sequence is a list of transactions ordered by the transaction time, the problem of mining sequential pattern is to discover all sequential patterns with a user-specified minimum support. Each transaction contains a set of items. A sequential pattern is an ordered list (sequence) of itemsets. The itemsets that are contained in the sequence are termed the *elements* of the sequence. The support of a sequential pattern is the percentage of data-sequences that contain the sequence.

Example 1.2 (*Sequential pattern mining*) *An example of a sequential pattern is that 80% customers typically buy "computer" and "modem", and then "printer". Then, $\langle$(computer, modem)(printer)$\rangle$ is a sequence with two elements. 80% here represents the percentage of customers who comply this purchasing habit.*

For sequential pattern mining, few constraints are added. First of which is, time constraints that specify a maximum and/or minimum time gaps between adjacent elements. Second, a sliding time window within which items are considered part of the same sequence element. They are specified by three parameters, $max - gap$, $min - gap$ and $window - size$. Third, given a user-defined taxonomy ($is - a$ hierarchy) on items, allow sequential patterns to include items across all levels of the taxonomy.

1.1.2.3 Classification

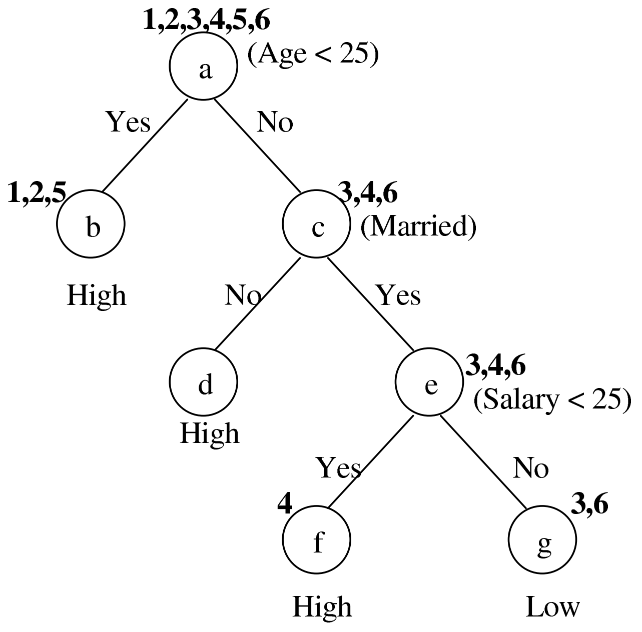
Classification is a well-studied problem [WK91, MAR96, RS00, SAM96]. It is to build (automatically) a model (called classifier) that can classify a class of objects so as to predict the classification or missing attribute value of future objects for which the class label is unknown. It consists of two steps. In the first step, based on the collection of *training set*, a model is generated to describe the characteristics of a set of data classes or concepts. In the second step, the model is used to predict the classes of future objects or data.

Each record of the training set consists of serval attributes which could be *continuous* (coming from an ordered domain) or *categorial* (coming from an unordered domain). A training set is typically used for validating and tuning the model. One of attributes will be the classifying attribute, which indicates the *class* to which each record belongs. Once a model is built from the given examples, it can be used to determine the class of future unclassified records.

Several classification models have been proposed, eg. bayesian classification,

TID	*Age*	*Salary*	*Married*	*Risk*
1	21	30	*No*	*High*
2	18	28	*Yes*	*High*
3	30	50	*Yes*	*Low*
4	35	20	*Yes*	*High*
5	23	60	*No*	*High*
6	47	80	*Yes*	*Low*

(a) Training set



(b) Decision tree

Figure 1.2: An example of a decision tree

neural networks, regression and decision trees. Decision tree classification is probably the most popular model, because it can be constructed relatively fast compared to other methods and it is simple and easy to understand.

A decision tree is a flow-chart-like structure consisting of internal nodes, leaf nodes, and branches. Each internal node represents a decision on a attribute, and each outgoing branch corresponds to a possible outcome of the decision. Each leaf node represents a class. Building a decision tree classifier generally includes two stages, a growing stage and a pruning stage. In tree growing, the decision tree model is built by recursively splitting the training set based on a locally optimal criterion until each partition consists entirely or dominantly of examples from one class. To improve generalization of a decision tree, tree pruning is used to prune the leaves and branches responsible for classification of single or very few data vectors. The Figure 1.2 shows a sample decision tree and the training set from which it is derived, which indicate whether or not a customer's credit is likely to be safe. After this model has been built, we can predict the credit of a new customer based on his attributes such as age, salary, and marital status.

1.1.2.4 Clustering

The fundamental clustering problem is that of grouping together similar data items, based on proximity between pairs of objects. The technique is useful to finding interesting distributions and patterns in the underlying data. It is a process of partition a data points into k cluster such that the data sets within a cluster are similar to each other, but are very dissimilar to data points in other clusters. Dissimilarities are assessed based on the attribute values describing the objects.

Clustering algorithms can be classified into two categories: partitioning and hierarchical. The popular *k-means* and *k-medoids* methods determine k cluster representatives and assign each data points to the cluster with the nearest representative such that the sum of the distances squared between the data points and their representatives is minimized. On the contrary, a hierarchical clustering is a nested sequence of partitions. Two specific hierarchical clustering methods are *agglomerative* and *divisive*. An agglomerative algorithm for hierarchical clustering starts with the disjoint clustering, which places each of the n objects in an individual cluster and then merges two or more these trivial clusters into larger and larger clusters until all objects are in a single cluster. It is a bottom up approach. A divisive algorithm performs the task in the reverse order. A comprehensive survey of current clustering techniques and algorithms is available in [Ber02].

1.2 Motivation

Since their introduction in 1993 by Argawal et al. [AIS93], the frequent pattern mining problems have been studied popularly in data mining research. We can categorize the ongoing work in frequent pattern mining area as follows:

- Developing new efficient mining algorithms. The most of the previous algorithms used in today typically employ sophisticated in-memory data structures, where the data is stored into and retrieved from flat files. In cases where the data are stored in a DBMS, data access is provided through an ODBC or SQL cursor interface. The potential problem with this approach is the high context switching cost between the DBMS and the mining process. Therefore, the integration of data mining with database systems should be naturally considered. We will discuss it in detail later.

- Scaling mining algorithm over large data sets. Broadly speaking, techniques for scaling data mining algorithms can be divided into five basic categories [GG02]: 1) manipulating the data so that it fits into the memory, 2) using specialized data structures to managed out of memory data, 3) distributing the computation so that it exploits several processors, 4) precomputing intermediate quantities of interest, 5) reducing the amount of data mined. One of the genetic way is to using sampling. Whether sampling is appropriate for data mining, and when appropriate, how much to sample and how, is still an important question for data mining. However, use of sampling restrict the scope of data.

1.2.1 Architectural Alternatives

There are a wide range of architectural alternatives for integrating mining process with the DBMS [STA98]. These alternatives are depicted in Figure1.3 and described below.

- Stored-Procedure: This architecture is representative of embedding mining logic as an application on the database server. In this approach, the mining algorithm

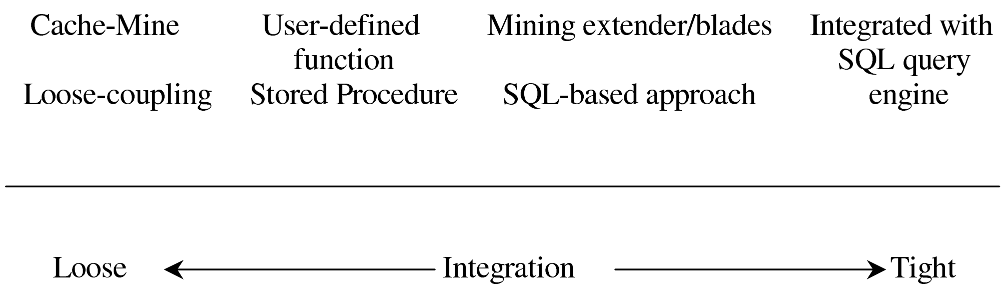


Figure 1.3: Architecture Alternatives

is encapsulated as a stored procedure that is executed in the same address space as the DBMS. The main advantage is great programming flexibility and no extra storage requirement. Also, any previous file system code can be easily transferred to work on data stored in the DBMS.

- Cache-Mine: In this architecture, the mining kernel reads the entire data once from the DBMS and temporarily caches the relevant data in a lookaside buffer on a local disk. The cached data could be transformed to a format that enables efficient future accesses. The cached data is discarded when the execution completes. After the mining algorithm is executed, the results will be first stored in a buffer on local disk, then sent back to the DBMS. This method has all the advantages of the Store-Procedure approach plus it has better performance. The disadvantage is that it requires extra disk space for caching. Note that the permanent data continues to be managed by the DBMS.

- User-defined functions: This approach is another variant of embedding mining as an application on the database server if the user-defined functions are run in the unfenced mode. Here, mining algorithm is expressed as a collection of user-defined functions that are appropriately placed in SQL data scan queries. Most of processing happens in the UDF and the core DBMS engine is used primarily to provide tuples to the UDFs. Query processing capability of the DBMS is little used. The main advantage of this method over Store-Procedure is the execute time because passing tuples to a store procedure is slower than passing it to a UDF. The main disadvantage is the development cost since the entire mining algorithm has to be rewritten.

- SQL-based approach: This is the integration architecture explored in this dissertation. In this approach, mining algorithm is formulated as SQL queries which are executed by the DBMS query processor. A mining-aware optimizer may be used to optimize these complex, long running queries based on the mining semantics.

- Integrated approach: This is the tightest form of integration of data mining with database systems. Mining operations are integrated into the DBMS and become part of the database query engine.

1.2.2 Why Data Mining with SQL

The integration of data mining with database systems is an emergent trend in database research and development area. This is particularly driven by the reasons as follows.

- Explosion of the data amount stored in databases such as Data Warehouses during recent years. Data Warehouses are deploying relational database technology for storing and maintaining data. Furthermore, data in data warehouse will not be exclusively used for data mining, but will be shared also by OLAP and other database utilities. Some of the algorithms have made assumption that ignore this fact. For example, some of the algorithms discuss how the physical design of the database may be tuned for a specific data mining task. However, in many cases the physical design of a data warehouse is unlikely to be guided solely by the requirement of a single data mining algorithm. Therefore, the data mining utilities must assume a relational backend.

- Database systems provide powerful mechanisms for accessing, filtering, and indexing data that the mining algorithm can exploit instead of developing all required functionality from scratch. Rather than devising specialized parallelization, one can potentially exploit the underlying SQL parallelization, especially in an SMP environment. In addition, the DBMS support for check-pointing and space management can be especially valuable for long-running mining algorithms on huge volumes of data. It's surprising that although scalability of mining algorithms has been an active area of work, few significant pieces of work have worked on the issue of data mining algorithms for SQL systems. A detailed study of a SQL-aware scalable implementation of association rules appear in [STA98].

- SQL-aware data mining systems have ability to support ad-hoc mining, ie., allowing to mine arbitrary query results from multiple abstract layers of database systems or Data Warehouses. These techniques need to be re-implemented in part if the data set in main-memory approaches does not fit into the available memory.

1.2.3 Goal

However, from the performance perspective, data mining algorithms that are implemented with the help of SQL are usually considered inferior to algorithms that process data outside the database systems [STA98]. One of the most important reasons is that main memory algorithms employ sophisticated in-memory data structures and try to reduce the scan of data as few times as possible while SQL based algorithms either require several scans over the data or require many and complex joins between the input tables. Almost all previous frequent itemset mining with SQL adopt an *Apriori* approach, which has the bottleneck of the candidate set generation and test. It is impossible to get a good performance out of pure SQL based *Apriori*-like approach.

Recently some commercial data mining solutions to the integration of data mining algorithms and functions directly into the database management system (DBMS) has been presented. Oracle Data mining (ODM) embeds data mining algorithms in the Oracle database. Instead of extracting data from database, the data in ODM never leaves the database. This enables Oracle to provide an infrastructure for data analysts and application developers to integrate data mining seamlessly with database applications. ODM provides DBMS-DATA MINING in PL/SQL packages. However, they don't really exploit SQL queries.

These facts motivated us to develop new SQL based algorithms which avoid making multiple passes over the large original input table and complex joins between tables. On the other hand recently most major database systems have included capabilities to support parallelization, this thesis examined how efficiently SQL based frequent pattern mining can be parallelized and speeded up using parallel database system.

1.3 Contributions

In this thesis, we investigate the problem of efficient and scalable frequent Pattern mining in RDBMSs. In particular, we make the following contributions.

- Based on the performance evaluation of previous methods, we develop a SQL based frequent pattern mining with a novel frequent pattern growth (FP-growth) method, which is efficient and scalable for mining dense databases without candidate generation.

- We further propose *Propad* (PROjection PAttern Discovery) approach, which avoid the cost of materializing frequent pattern tree tables. This approach is efficient and scalable for mining both sparse and dense databases. Furthermore, to achieve efficient frequent pattern mining in various situation, we design a hybrid algorithm, which smartly combines the *Apriori* approach and *Propad* approach together.

- We also examine how efficiently SQL based frequent pattern mining can be parallelized and speeded up using parallel database systems.

1.4 Outline of the Dissertation

The remainder of the thesis is structured as follows:

- In Chapter 2, we explain the frequent pattern mining problem. We present an in depth analysis of the most influential algorithms that were proposed during the last decade.
- In Chapter 3, we present the overview of related work systematically.
- In Chapter 4, a SQL based frequent pattern mining with *FP-tree* method is developed. We also present performance comparison of different approaches using real-life and synthetic datasets.
- In Chapter 5, we further propose *Propad*, which retains the advantages of *FP-tree* but avoids *FP* tables materializing. Even though SQL based *FP-tree* approach is efficient in mining many kinds of databases, it may have the problem of building many recursive *FP* tables and thus may be time consuming. Our performance study shows that *Propad* achieves scalability in mining large databases. Meanwhile, a hybrid method is proposed in mining both dense and sparse data sets.
- In Chapter 6, We also examine how efficiently SQL based frequent pattern mining can be parallelized and speeded up using parallel database systems.
- The thesis concludes and directions in future work in Chapter 7.

Chapter 2

Frequent Pattern Mining

In this chapter, we first explain the problem of frequent pattern mining, then we describe the main techniques used to solve this problem and give a comprehensive survey of the most influential algorithms that were proposed during the last decade.

2.1 Frequent Pattern Mining Problem Description

The formal definition of frequent pattern and association rule mining problems is introduced in [AIS93], it can be stated as follows.

Let $I = \{i_1, i_2, ..., i_m\}$ be a set of *items*. An itemset $X \subseteq I$ is a subset of items. Particularly, an itemset with k items is called an *k-itemset*.

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

Let *transaction database* DB be a set of transactions. The support of an itemset X in transaction database DB, denoted as $sup(X)$, is the number of transactions in DB containing X:

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

The frequency of an itemset X in DB, denoted as $frequency(X)$, is the probability of X occurring in a transaction $T \in DB$:

$$frequency(X) = \frac{sup(X)}{|DB|}$$

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

In practical, we are not only interested in the set of frequent itemsets, but also in the actual supports of these itemsets.

Association rule can be derived from frequent patterns. An $association\ rule$ is an expression $X \Rightarrow Y$, where X and Y are itemsets and $X \bigcap Y = \emptyset$. The support of the rule $X \Rightarrow Y$ in a transaction database DB is given as $sup_{DB}(X \cup Y)$, the $confidence$ as $\frac{sup(X \cup Y)}{sup(X)}$ is the conditional probability that a transaction contains Y, given that it contains X. The rule is called $confidence$ if $\frac{sup(X \cup Y)}{sup(X)}$ exceeds a given $minimal\ confidence\ threshold\ min\_conf$.

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

Association rule mining problem can be decomposed into two subproblems.

- Find all frequent patterns whose support is greater than support threshold min_sup.
- Use the frequent patterns to generate the desired rules having confidence higher than min_conf.

As shown in many studies(eg., [AS94]), finding all frequent patterns is significantly more costly in terms of time than generating rules. In the following section, we will analyze the computational complexity of finding frequent patterns.

2.2 Complexity of Mining Frequent Patterns

The task of discovering all frequent patterns is quite challenging. In the beginning of the mining run each itemset $X \subseteq I$ is potentially frequent. That is, the initial search space consists of the power set of I without the empty set. Therefore, even

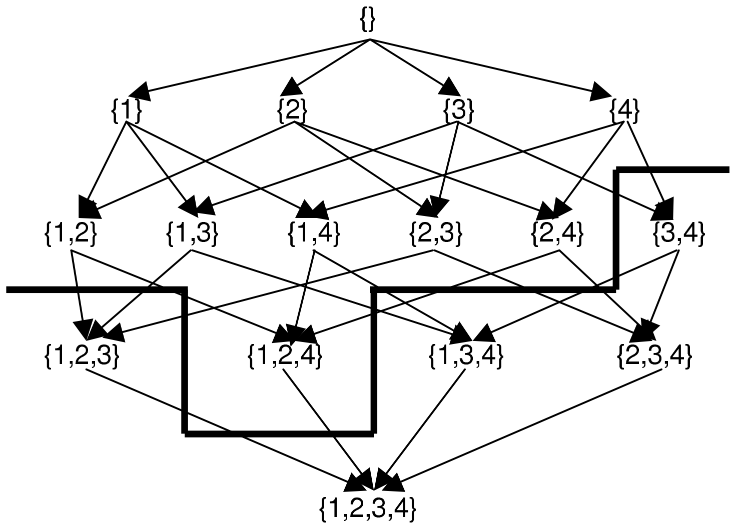


Figure 2.1: The lattice for $I = \{1, 2, 3, 4\}$

for a small $|I|$ the search space easily exceeds all limits. If I is large enough, it's therefore not practicable to determine the support of each of the subset of I in order to decide whether it is frequent or not. Also, support counting is a tough problem when database is massive, containing millions of transactions.

2.2.1 Search Strategy

The search space is exponential with $|I|$, noted by $2^{|I|}$. For a special case $I = \{1, 2, 3, 4\}$, we visualize the search space that forms a lattice in Figure 2.1 [ZPOL97a]. The bold line is an example of actual itemset support and separates the frequent itemsets in the upper part from the infrequent itemsets in the lower part. The existence of such a border is guaranteed by the downward closure property of itemset support. We will describe the property in the following.

To prune the search space, the idea is to traverse the lattice in such a way that all frequent itemsets are found but as few as infrequent itemsets as possible are visited. This can be achieved by employing the downward closure property of itemset: if an itemset is infrequent, all of its supersets must be infrequent. Clearly, the proposed stepwise traversal of the search space should adopt either bottom-up or top-down direction. The main advantage of the former strategy is that it can effectively prune the search space by exploiting downward closure property. However, this advantage fades when most of the maximal frequent itemsets locating near the largest itemset of the search lattice. In this case, there are very few itemsets to be pruned. The later strategy is traditionally adopted for discovering maximal frequent patterns [AAP00, Bay98, TL01]. Although all of the frequent patterns can be derived from the maximal ones, many infrequent itemsets have to be visited before the maximal frequent itemsets are identified if there are large numbers of items and/or the support threshold is very low. This is why most work on frequent pattern mining embraces the bottom-up strategy instead.

Today's common approaches employ both breadth-first search (BFS) and depth-first search (DFS) to traverse the search space. With BFS the support values of all $(k-1)$-itemsets are determined before counting the support values of all (k)-itemsets. This strategy can facilitate the pruning of candidates with monotone property. However, it requires more memory to keep the frequent subsets of the pruned candidates. In contrast, DFS [Bay98, PB99, AAP00, KP03] recursively visits the descendants of an itemset. In [Zak00], this approach is usually combined with the vertical intersection counting strategy because it suffices to keep the tidlists, corresponding to the itemsets on the path from the root down to the currently inspected one, in memory.

2.2.2 Counting Strategy

Computing the supports of a collection of itemsets is a time consuming procedure. The major challenge frequent pattern mining problem faces is how to efficiently count the support of all candidate itemsets visited during the traversal. Up to date, there are two main approaches: horizontal counting and vertical intersection. The horizontal counting determines the support value of a candidate itemset by scanning the transaction one by one, and increasing the counter of the itemset if it is a subset of the transaction. Efficiently looking up candidates in transactions requires specialized data structures, e.g. hashtrees or prefix trees. This approach works well for a rarely occurred candidate, however, is costly for candidates of large size.

The vertical intersection [SON95, DS99, Zak00] is employed when the database is represented as a vertical format, in which the database consists of a set of items, each followed by the identifiers of the transactions containing that item, called tidlist. The transaction set X.tids of an itemset X is defined as the set of all transactions this itemset is contained in:

$$X.\text{tids} = \{T \in DB | X \subseteq T\}$$

For the support follows

$$sup(X) = \frac{|X.tids|}{|DB|}$$

For each itemset Z with $Z = X \cup Y$, the support of itemset Z can easily computed by simply intersecting the tidlists of any two subsets $X, Y \subset Z$. It holds

$$Z.\text{tids} = X.\text{tids} \cap Y.\text{tids}$$

Though the vertical intersection scheme eliminates the I/O cost for database scan, there occurs a large amount of unnecessary intersection when the support count of a candidate itemset is quite less than the number of transactions.

Researchers have been seeking for efficient solutions to the problem of frequent pattern mining since 1993. In the following section we will discuss some most influential algorithms during the last decades.

2.3 Common Algorithms

In this section, we describe and systemize the common algorithms for mining frequent patterns. We characterize each of the algorithms by its strategy to traverse the search space and its strategy to count the support values of the itemsets. Figure 2.2 shows a classification of prevailing approaches.

The foundation of all frequent pattern mining algorithms is from the properties of frequent sets. They are described as follows.

- Support for subsets.

  X, Y are itemsets. If $X \subseteq Y$, then $Sup(X) \geq Sup(Y)$ because all transactions in DB that support Y necessarily support X also.

Counting Strategy	bottom-up		top-down	
	BFS	DFS	BFS	DFS
horizontal	Apriori DHP DIC TreeProjection	DF FP-growth H-mine EFP Propad	Top-down	
	OpportuneProject, Hybrid			
vertical	Partition	Eclat CW		
Hybrid	DCI			
	CBW			

Figure 2.2: Systematization of the algorithms (The algorithms: *EFP*, *Propad*, and *Hybrid* are proposed in this thesis)

- Supersets of infrequent sets are infrequent.

  If itemset $Sup(X) < min\_sup$, then every superset Y of X will not be frequent because $Sup(Y) \leq Sup(X) < min\_sup$ according to the above property.

- Subsets of frequent sets are frequent.

  If itemset Y is frequent in DB, ie., $Sup(Y) \geq min\_sup$, every subset X of Y is frequent in DB because $Sup(X) \geq Sup(Y) \geq min\_sup$ according to the first property.

2.3.1 The *Apriori* Algorithm

The first algorithm to generate all frequent patterns was the AIS algorithm proposed by Agrawal et al. [AIS93], which was given together with the introduction of this mining problem. To improve the performance, an anti-monotonic property of the support of itemsets, called the *Apriori* heuristic, was identified by Agrawal et al. in [AS94, SA97]. The same technique was independently proposed by Mannila et al. [MTV94]. Both works were cumulated afterwards in [AMS+96].

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

TID	*Items*	*Frequent Items*
1	$1, 3, 4, 6, 7, 9, 13, 16$	$1, 3, 6, 13, 16$
2	$1, 2, 3, 6, 12, 13, 15$	$1, 2, 3, 6, 13$
3	$2, 6, 8, 10, 15$	$2, 6$
4	$2, 3, 11, 16, 19$	$2, 3, 16$
5	$1, 3, 5, 6, 12, 13, 16$	$1, 3, 6, 13, 16$

Table 2.1: An example transaction database DB and $\xi = 3$

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

Example 2.1 (*Apriori*) *Let's give an example with five transactions DB and support threshold ξ is set to* 3 *in Table 2.1.*

The process of *Apriori* to find the complete frequent patterns in DB as follows. Figure 2.3 illustrates this process.

1. Scan DB once to generate length-1 frequent itemsets, labeled as F_1. In this example, they are {1, 2, 3, 6, 13, 16}.

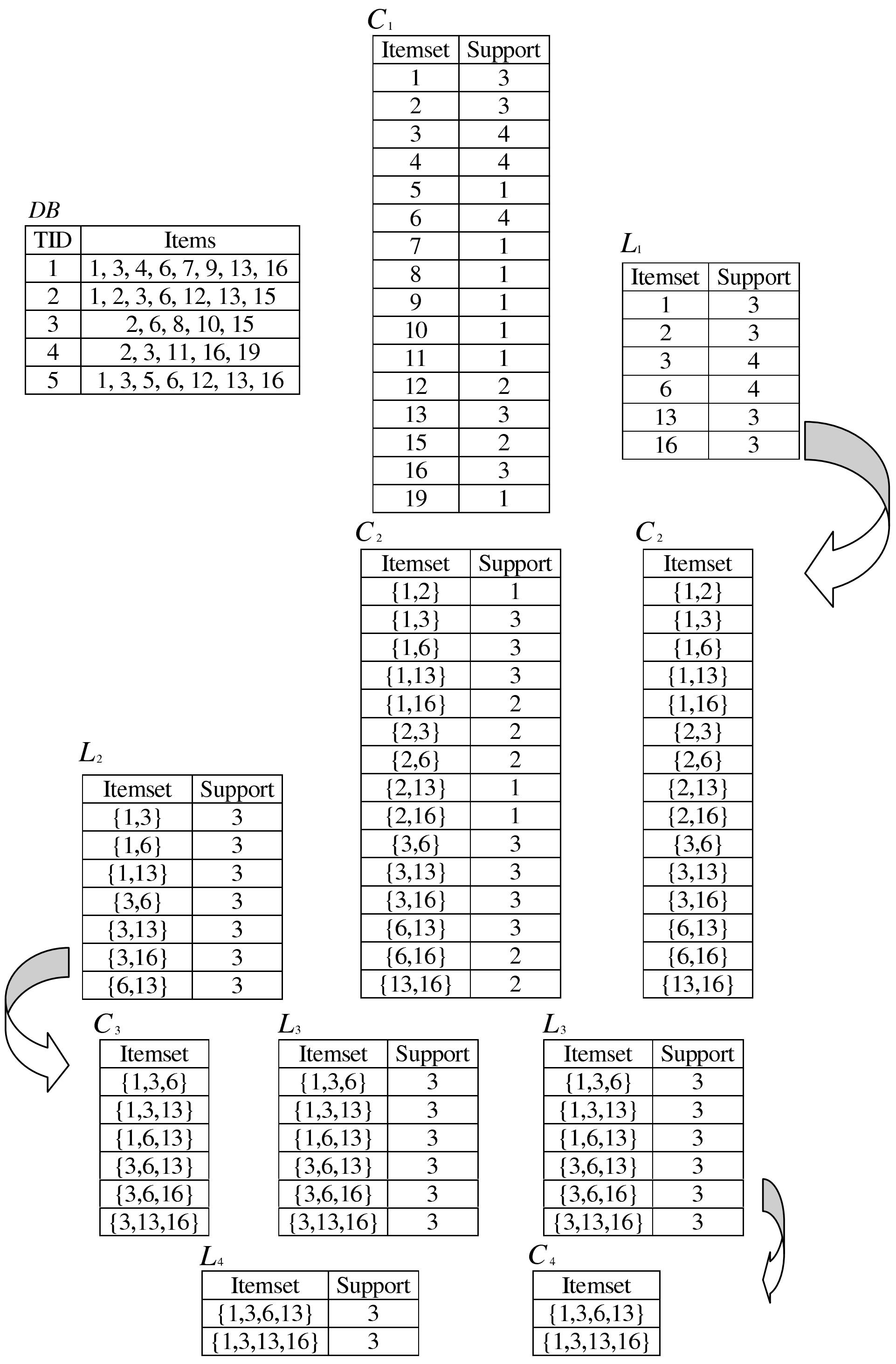
2. Generate the set of length-2 candidates, denoted as C_2 from F_1.

3. Scan DB once more to count the support of each itemset in C_2. All itemsets that turn out to be frequent in C_2 are inserted into F_2. In this example, F_2 contains $\{(1,3),(1,6),(1,13),(3,6),(3,13),(3,16),(6,13)\}$.

4. Then, we form the set of length-3 candidates from F_2 and frequent 3-itemsets F_3 from C_3. The similar process goes on until no candidates can be derived or no candidate is frequent.

The *Apriori* algorithm is presented as follows.

Apriori performs a BFS by iteratively obtaining candidate itemsets of size $(k+1)$ from frequent itemsets of size k, and check their corresponding occurrence frequencies



DB

TID	Items
1	1, 3, 4, 6, 7, 9, 13, 16
2	1, 2, 3, 6, 12, 13, 15
3	2, 6, 8, 10, 15
4	2, 3, 11, 16, 19
5	1, 3, 5, 6, 12, 13, 16

C_1

Itemset	Support
1	3
2	3
3	4
4	4
5	1
6	4
7	1
8	1
9	1
10	1
11	1
12	2
13	3
15	2
16	3
19	1

L_1

Itemset	Support
1	3
2	3
3	4
6	4
13	3
16	3

C_2

Itemset
{1,2}
{1,3}
{1,6}
{1,13}
{1,16}
{2,3}
{2,6}
{2,13}
{2,16}
{3,6}
{3,13}
{3,16}
{6,13}
{6,16}
{13,16}

C_2

Itemset	Support
{1,2}	1
{1,3}	3
{1,6}	3
{1,13}	3
{1,16}	2
{2,3}	2
{2,6}	2
{2,13}	1
{2,16}	1
{3,6}	3
{3,13}	3
{3,16}	3
{6,13}	3
{6,16}	2
{13,16}	2

L_2

Itemset	Support
{1,3}	3
{1,6}	3
{1,13}	3
{3,6}	3
{3,13}	3
{3,16}	3
{6,13}	3

C_3

Itemset
{1,3,6}
{1,3,13}
{1,6,13}
{3,6,13}
{3,6,16}
{3,13,16}

L_3

Itemset	Support
{1,3,6}	3
{1,3,13}	3
{1,6,13}	3
{3,6,13}	3
{3,6,16}	3
{3,13,16}	3

L_3

Itemset	Support
{1,3,6}	3
{1,3,13}	3
{1,6,13}	3
{3,6,13}	3
{3,6,16}	3
{3,13,16}	3

C_4

Itemset
{1,3,6,13}
{1,3,13,16}

L_4

Itemset	Support
{1,3,6,13}	3
{1,3,13,16}	3

Figure 2.3: The Apriori algorithm – example

Algorithm 1 *Apriori*

Input: A transaction database DB and minimum support threshold ξ
Output: The complete set of frequent patterns in DB
Method:

```
1. scan $DB$ once to find the frequent 1-items $F_1$;
2. for (k = 1; $F_k \neq \emptyset$; k++) do begin
3.     generate $C_{k+1}$, the set of length-$k$ candidates, from $F_k$;
4.     for each transaction $t$ in $DB$ do
5.         increment the count of all candidates in $C_{k+1}$ that are contained in $t$
6.     $F_{k+1}$ = candidates in $C_{k+1}$ whose supports are no less than $\xi$
7.     end
8. $\cup_k F_k$
```

in the database. Each iteration requires a scan of the original database. Many variants that improve *Apriori* have been proposed by reducing the number of candidates further, the number of transactions to be scanned, or the number of database scans, the process is still expensive as it is tedious to repeatedly scan the database and check a large set of candidates by pattern matching, which is particularly true if a long pattern exists. In short, the bottleneck for Apriori-like methods is the *candidate-generation-and-test* operation.

2.3.2 Improvements over *Apriori*

In the past several years, many variants that improve *Apriori* have been proposed. In this section, we review some influential algorithms.

AprioriTid is from Agrawal et al. [AS94]. The AprioriTid algorithm reduces the time needed for the support counting procedure by replacing the every transaction in the database by the set of candidates contained in that transaction. This adapted transaction database is denoted as $\overline{C}_k$. The AprioriTid algorithm is much fast in later iterations, but it performs much slower than *Apriori* in early iterations. This is mainly due to the additional overhead that is created when $\overline{C}_k$ does not fit into main memory. **AprioriHyTid** combines the *Apriori* and AprioriTid into a single hybrid. This hybrid algorithm uses *Apriori* for the initial iterations and switches to AprioriTid when it is expected that the set $\overline{C}_k$ fits into main memory. AprioriHyTid performs almost always better than *Apriori*.

Park et al. [PCY95a] proposed an optimization, called **DHP** (*D*irect *H*ashing and *P*runing) to reduce the number of candidate itemsets. In the k-th iteration, DHP counts the supports of length-k candidates. At the same time, potential length-(k+1) candidates are generated and hashed into buckets. Each bucket in the hash table consists of a counter to represent how many itemsets have been hashed to that bucket so far. If the counter of the corresponding bucket is below the support threshold, the potential length-(k+1) candidates should not be length-k candidates.

DHP results in a significant decrease in the number of candidate itemsets, especially in the second iteration. Nevertheless, creating the hash tables and writing the adapted database to disk at every iteration causes a significant overhead.

Partition, proposed by Savasere et al. [SON95], combines the *Apriori* approach with set intersections instead of count occurrences. That is, the database is stored in main memory using the vertical database layout and the support of an itemset is computed by intersecting the tidlists of two of its subsets. In addition, the Partition algorithm partitions the database in several chunks according to such a way that each partition can be held in main memory.

For each partition, the Partition algorithm first mines local frequent patterns with respect to relative support threshold using the *Aprirori* approach. Then, the algorithm merges all local frequent patterns together to consolidate global frequent patterns.

The Partition algorithm is highly dependent on the heterogeneity of the database. That is, partitioning the database is non-trivial when the database is biased. On the other hand, a huge number of local frequent itemsets can be generated when the global support threshold is low.

DIC, a dynamic itemset counting algorithm, is proposed by Brin et al [BMUT97]. It is an extension of *Apriori* that aims at minimizing the number of database scans. The idea is to relax the strict separation between generating and counting candidates. It divides the database into intervals of specific size. Whenever the support count of a candidate itemset passes the support threshold in an interval, that is even this candidate has not yet seen all transactions, DIC starts generating additional candidates based on it and counting the support of them at the next interval. By overlapping the counting of different length of items, DIC can save some database scans. On the other hand, DIC employs a prefix-tree structure to store candidate itemsets. In contrast to the usage of hashtree that means whenever we reach a node we can be sure that the itemset associated with this node is contained in the transaction.

Experimental results reported in [BMUT97] show that DIC is faster than *Apriori* when the support threshold is low. It's performance, however, is heavily dependent on the distribution of the data.

A theoretical analysis of **sampling**(using Chernoff bounds) for association rules was presented in [MTV94, AMS$^+$96]. In [JL96] the authors compare sample selection schemes for data mining. A sampling algorithm, proposed by Toivonen [Toi96], first mine a sample of the database using the *Apriori* algorithm instead of mining the database directly. The sample should be small enough to fit into main memory. Then, the whole database is scanned once to verify frequent itemsets found in the sample. In some rare cases where the sampling method does not produce all frequent patterns, the missing patterns can be found by generating all remaining potentially frequent patterns and verifying their supports during the second pass through the database.

The performance study in [Toi96] shows that the sampling algorithm is faster than both *Apriori* and the partitioning method in [SON95]. To keep the probability of such a failure small, where the sampling method may not produce all frequent patterns, the minimal support threshold can be decreased. However, for a reasonably small probability of failure, the threshold must be drastically decreased, which can cause a combination explosion of the number of candidate patterns. The sampling methods are efficient for two main reasons: 1) they only examine a small sample of the database, frequent patterns can be discovered very efficiently with reasonably high accuracy, 2) it can often fit entirely into main memory since the sample is small in size, thus reducing the I/O overhead of repeated database scanning.

Zaki et al. [ZPLO96] complement the approach proposed in [Toi96], and can help in determining a better support or sample size. They note that there is a trade-off between the performance of the algorithm and the desired accuracy or confidence of the sample.

In [ZPOL97a, Zak00] the algorithm **Eclat** is introduced, that combines DFS with tidlist intersections based approach on vertical data layouts. An efficient implementation of *Eclat* is proposed by Borgelt in [Bor03].

The main difference between *Apriori* and *Eclat* are how they traverse the search space and how they determine the support of an itemset. *Apriori* traverses the search space in a breadth first oeder, that is, it first check itemsets of size 1, then itemsets of size 2 and so on. *Apriori* determines the support of itemsets either by testing for each candidate itemset which transactions it is contained in, or by traversing for a transaction all subsets of the currently processed size and incrementing the corresponding itemset counters. *Eclat*, on the other hand, traverses the search space in depth first order. That is, it extends an itemset prefix until it reaches the boundary between frequent and infrequent itemsets and then backtracks to work on the next prefix. *Eclat* determines the support of itemsets by constructing the list of identifiers of transactions that contain the itemset.

When using DFS it suffices to keep the tidlists on the path from the root down to the class currently investigated in memory. That is, splitting the database as done by Partition is no longer necessary. However, Eclat essentially generates candidate itemsets using only the join step form *Apriori* and doesn't fully exploit the monotonicity property, the number of candidate itemsets that are generated is much larger as compared to a BFS approach.

Recently, Zaki proposed a new approach to efficiently compute the support of an itemset using the vertical database layout [ZG03]. The novel data representation called $diffset$, which only keeps track of differences in the tidlists of a candidate itemset, is presented. Experimental results show that $diffsets$ deliver order of magnitude performance improvement over the best previous methods. Nevertheless, the algorithm still requires the original database to be stored in main memory.

Apriori is more efficient than *Eclat* in the early passes when the itemset cardinality is small, but inefficient in later passes when the length of the frequent itemsets is high and the number of them decreases. But *Eclat* on the other hand, has better performance during these later passes as it uses tidlist intersections, and the tidlists shrink with increase in the size of itemsets. This motivated a study of an adaptive *hybrid* strategy which switches to *Eclat* in higher passes of the algorithm. *Hybrid* strategies were studied in [STA98, HGN00b, RMZ02]. The hybrid method tended to be uniformly less efficient than *Eclat* for the databases but often more efficient than *Apriori* [RMZ02].

DCI (Direct Count & Intersect) algorithm, is presented in [OPP01, OPPS02]. As *Apriori*, at each iteration DCI generates the set of candidates C_k, determines their supports and builds the set of frequent k-itemsets F_k. However, DCI adopts a hybrid approach to determine the support of the candidates. During the first iterations, DCI exploits a novel counting-based technique, accompanied by an efficient pruning of the dataset. Since the counting-based approach becomes less efficient as k increases [SON95]. DCI starts its intersection-based phase as soon as possible. Unfortunately, intersection-based method needs to maintain in memory the vertical representation of the pruned dataset. So, at iteration k, $k \geq 2$, DCI checks whether the pruned data set may fit into the memory. When the dataset becomes small enough, DCI starts to employ a intersection-based approach. The distinct heuristic strategy is dynamically chosen according to the dataset peculiarities. For example, when a data set is dense, identical sections appearing in several bit-vectors are aggregated and clustered, in order to reduce the number of intersections actually performed. Conversely, when a data set is sparse, the runs of zero bits in the bit-vectors to be intersected are promptly identified and skipped. [OPPS02] shows that DCI performs very well on both synthetic and real-world datasets characterized by different density features.

2.3.3 *TreeProjection*: Going Beyond *Apriori*-like Methods

Agarwal, et al. propose **TreeProjection** [AAP01], a frequent pattern mining algorithm not in the *Apriori* framework. *TreeProjection* represents frequent patterns as nodes of a lexicographic tree and counts the support of frequent itemsets by projecting the transactions into the nodes of this tree. An example of the lexicographic tree

TID	$Frequent\ Items$
1	a, c, d, f
2	a, b, c, d, e, f
3	b, d, e
4	c, d, e, f

Table 2.2: An example transaction database DB and $\xi = 2$

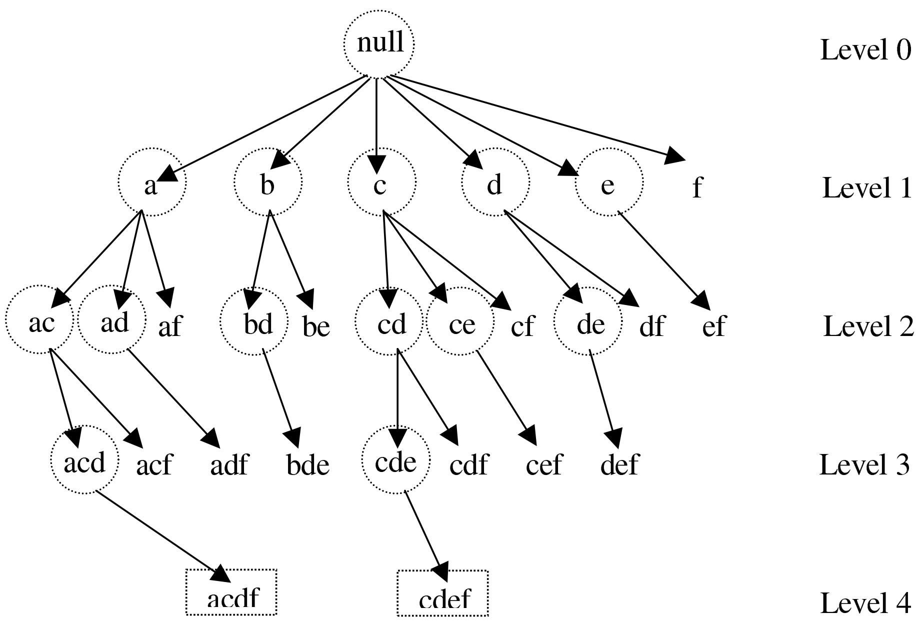


Figure 2.4: The lexicographic tree

is illustrated in Figure 2.4.

Example 2.2 (*TreeProjection*) *Let the transaction database DB, and the minimum support threshold be 2 in Table 2.2. The second column contains frequent items in each transaction.*

In a hierarchical manner, the algorithm looks only at that subset of transactions which can possibly contain that itemset. This significantly improves the performance of counting the number of transactions containing a frequent itemset. The general idea is shown in the following.

By scanning the transaction database once, all frequent 1-itemsets are identified. The top level of the lexicographic tree is constructed, i.e. the root labelled "null" and the nodes labelled by length-1 patterns. In order to count all possible extensions for each node, all transactions in the database are projected to that node. A matrix at the root node is created as shown below. The matrix is built by adding counts from transactions in the projected database, so that it computes the frequencies of length-2 patterns.

$$\begin{array}{c|cccccc}
 & a & b & c & d & e & f \\
a & & & & & & \\
b & 1 & & & & & \\
c & 2 & 1 & & & & \\
d & 2 & 2 & 3 & & & \\
e & 1 & 2 & 2 & 3 & & \\
f & 2 & 1 & 3 & 3 & 2 &
\end{array}$$

From the matrix, frequent 2-itemsets are founded to be: {ac, ad, af, bd, be, cd, ce, cf, de, df, ef}. The nodes in the lexicographic tree for these frequent 2-itemsets are generated. At this stage, the active nodes for 1-itemsets are c, d, and e, because only these nodes contain enough descendants to potentially generate longer frequent itemsets. All other nodes are pruned. The lexicographic tree is grown in the same way until the kth level of the tree equal to $null$. The number of nodes in a lexicographic tree is exactly that of the frequent itemsets.

$TreeProjection$ proposes an efficient way to enumerate frequent patterns. The efficiency of $TreeProjection$ can be explained by two factors:

- The use of projected transaction sets in counting supports is important in the reduction of CPU time for counting frequent itemsets.
- The lexicographic tree facilities the management and counting of candidates and provides the flexibility of choosing an efficient strategy during the tree construction phase as well as transaction projection phase.

[AAP01] reports that their algorithm is up to one order of magnitude faster than other former techniques in literature.

TreeProjection is primarily based on pure BFS. It still suffers from some problems related to efficiency, scalability, and implementation complexity. We describe them as follows.

- *TreeProjection* may generate huge frequent itemset tree in a large database or when the database has quite long frequent itemsets;
- Since one transaction may contain many frequent itemsets, one transaction in *TreeProjection* may be projected many times to many different nodes in the lexicographic tree. When there are many long transactions containing numerous frequent items, transaction projection becomes a nontrivial cost of *TreeProjection*.
- *TreeProjection* may encounter difficulties at computing matrices when the database is huge, when there are a lot of transactions containing may frequent items.

2.3.4 The *FP-growth* Algorithm

Lately, an *FP-tree* based frequent pattern mining method [HPY00], called *FP-growth*, developed by Han et al achieves high efficiency, compared with *Apriori*-like approach. The *FP-growth* method adopts the divide-and-conquer strategy, uses only two full I/O scans of the database, and avoids iterative candidate generation.

In [HPY00], frequent pattern mining consists of two steps:

1. Construct a compact data structure, frequent pattern tree (FP-tree), which can store more information in less space.
2. Develop an *FP-tree* based pattern growth (FP-growth) method to uncover all frequent patterns recursively.

TID	*Items*	*Frequent Items*
1	$1, 3, 4, 6, 7, 9, 13, 16$	$3, 6, 1, 13, 16$
2	$1, 2, 3, 6, 12, 13, 15$	$3, 6, 1, 2, 13$
3	$2, 6, 8, 10, 15$	$6, 2$
4	$2, 3, 11, 16, 19$	$3, 2, 16$
5	$1, 3, 5, 6, 12, 13, 16$	$3, 6, 1, 13, 16$

Table 2.3: A transaction database DB and $\xi = 3$

2.3.4.1 Construction of *FP-tree*

The construction of *FP-tree* requires two scans on transaction database. The first scan accumulates the support of each item and then selects items that satisfy minimum support. In fact, this procedure generates frequent 1-itemsets and then stores them in frequency descending order. The second scan constructs *FP-tree*.

An *FP-tree* is a prefix-tree structure storing frequent patterns for the transaction database, where the support of each tree node is no less than a predefined minimum support threshold ξ. It consists of one root labeled as "*null*", a set of *item-prefix subtrees* as the children of the root, and a *frequent-item-header* table. Each node in the *item-prefix subtree* consists of three fields: *item-name*, *count*, and *node-link*. Where *node-link* links to the next node in the *FP-tree* carrying the same item-name, or null if there is none. For any frequent item i, all the possible frequent patterns that contain i can be obtained by following i's node-links, starting from i's head in the *FP-tree* header. The frequent items in each path are stored in their frequency descending order.

The algorithm of construction *FP-tree* is presented as follows.

Example 2.3 (*FP-tree*) *Let the transaction database DB, be the first two columns of Table 2.3 (same as the transaction database used in Example 2.1), and the minimum support threshold be 3. Figure 2.5 illustrates an FP-tree for the given example in Table 2.3. One may construct a FP-tree as follows.*

1. Scan the transaction databaset DB once to derive a list of frequent items,

Algorithm 2 *FP-tree* construction

Input: A transaction database DB and minimum support threshold ξ
Output: FP
Method: The *FP-tree* is constructed in the following steps.

1. Scan the transaction database DB once. Collect the set of frequent items F and their supports. Sort F in support descending orders as L, the list of frequent itemsets.

2. Create the root of an *FP-tree*, T, and label it as "null". For each transaction t in DB select the frequent items in transaction t and sort them according to the order of F. Let the stored frequent-item list in t be $[p|P]$, where p is the first element and P is the remaining list. Call $insert\_tree([p|P], T)$.

```
$insert\_tree([p|P], T)$
1. if $T$ has a child N such that N.item-name = p.item-name
2.    then increment N's count by 1;
3. else do {create a new node N;
4.    N's count = 1;
5.    N's parent link be linked to T;
6.    N's node-link be linked to the nodes with the same item-name
      via the node-link structure;}
7. if P is nonempty
8.    Call $insert\_tree(P, N)$
```

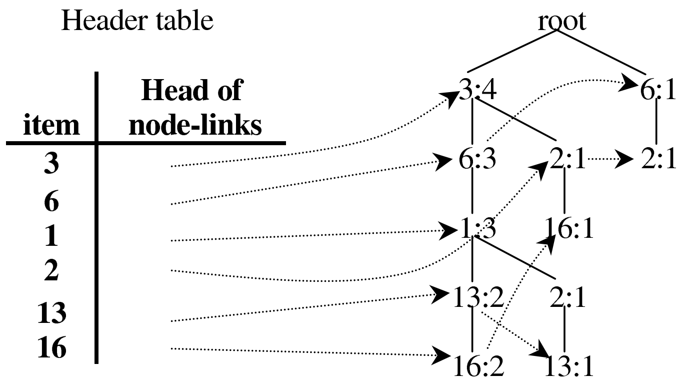


Figure 2.5: An *FP-tree* for Table 2.3

$FList$, $\langle(3:4),(6:4),(1:3),(2:3),(13:3),(16:3)\rangle$, in which items are ordered in frequency descending order.

2. Create the root of an *FP-tree*, T, label it as "null". For each transaction t in DB do the following.
   Select the frequent items in transaction t and sort them according to the order of $FList$, like the last column of Table 2.3. For each item p in t, if T has a child N such that *N.item-name* = *p.item-name*, then increment N's count by 1; else create a new node N, with count initialized to 1, parent link linked to T, and node-link linked to the nodes with the same *item-name* via the node-link structure.

2.3.4.2 Mining Frequent Patterns using *FP-tree*

Based on *FP-tree* structure, an efficient frequent pattern mining algorithm, *FP-growth* method is proposed, which is a divide-and-conquer methodology: decompose mining task into smaller ones, and only need sub-database test.

FP-growth performs as follows:

1. For each node in the *FP-tree* construct its conditional pattern base, which is a "subdatabase" constructed with the prefix subpath set co-occurring with the suffix pattern in the *FP-tree*. *FP-growth* traverses nodes in the *FP-tree* from

Item	*Conditional Pattern Base*	*Conditional FP-tree*	*Frequent Pattern*
16	$\{(13:2, 1:2, 6:2, 3:2), (2:1, 3:1)\}$	$\langle 3:3 \rangle$	3 16 : 3
13	$\{(1:2, 6:2, 3:2), \{(2:1, 1:1, 6:1, 3:1)$	$\langle 3:3, 6:3, 1:3 \rangle$	3 13 : 3, 6 13 : 3 1 13 : 3
2	$\{(1:1, 6:1, 3:1)$, $(6:1)\}$	ϕ	ϕ
1	$\{(6:3, 3:3)\}$	$\langle 3:3, 6:3 \rangle$	3 1 : 3, 6 1 : 3
6	$\{(3:3)\}$	$\langle 3:3 \rangle$	3 6 : 3
3	ϕ	ϕ	ϕ

Table 2.4: Mining of all-patterns based on *FP-tree*

the least frequent item in L. For instance, in the example 2.3, the frequent item 16 is first mined. Following its node-links, 16 has two paths in the *FP-tree*: < 3:4, 6:3, 1:3, 13:2 > and < 3:4, 2:1 >. The first path indicates that $\{3, 6, 1, 13\}$ appears twice together with 16. Similarly, the second path indicates that $\{3, 2, 16\}$ appears once in the set of transactions in DB. These two prefix paths of 16, $\{(3\ 6\ 1\ 13\ 16 : 2), (3\ 2\ 16 : 1)\}$ is called 16's conditional pattern base.

2. Construct conditional *FP-tree* from each conditional pattern base. For example, construction of an *FP-tree* on the 16's conditional pattern base leads to only one branch (3:3). Hence, only one frequent pattern $\{3\ 16 : 3\}$ is derived.

3. Execute the frequent pattern mining recursively upon the conditional *FP-tree*. If the conditional *FP-tree* contains a single path, simply enumerate all the patterns.

The algorithm for mining frequent patterns using *FP-tree* is presented as follows. With the *FP-tree* in Figure 2.5, the mining process and result is listed in Table 2.4.

A performance study has been shown that *FP-growth* is at least an order of magnitude faster than *Apriori*, and such a margin grows even wider when the frequent patterns grow longer.

Algorithm 3 *FP-growth*

Input: A transaction database DB, represented by *FP-tree*, and minimum support threshold ξ
Output: The complete set of frequent patterns in DB
Method: **call** *FP-growth* (*FP-tree*, *null*).

```
FP-growth (Tree, α)
{
1. if $Tree$ contains a single path $P$
2.    for each combination (denoted as $\beta$) of the nodes in the path $P$ do
3.         generate pattern $\beta \cup \alpha$ with support = minimum support of nodes in $\beta$;
4. else for each $i$ in the header of $Tree$ (in verse order) do
      {
5.         generate pattern $\beta = i \cup \alpha$ with support = $i$.support;
6.         construct $\beta$'s conditional pattern base and then $\beta$'s conditional FP-tree $Tree_\beta$;
7.         if $Tree_\beta \neq \phi$
8.            then call FP-growth ($Tree_\beta, \beta$)
      }
}
```

2.3.5 Improvements over *FP-tree*

Although *FP-growth* is more efficient than *Apriori* in many cases, it may still encounter some difficulties in some cases. For instances, huge space is required to serve the mining if the database is huge and sparse; the number of conditional *FP-tree* is the same order of magnitude as number of frequent itemsets. The algorithm is not scalable to sparse and very large databases.

Pei et al. propose a memory-based hyper structure, *H-struct* [PHL+01], to store the sparse databases in main memory, and develops an *H-struct* based pattern-growth algorithm, *H-mine.* In comparison with *FP-growth*, **H-mine** does not generate physic projected databases and conditional *FP-tree* and thus saves space as well as time in many cases. However, In dense data sets, the *FP-tree*-based mining has advantage over mining on *H-struct*. *H-mine* invokes *FP-growth* to mine dense databases and uses a database partitioning technique to deal with very large databases.

A performance study has been shown that *H-mine* has high performance, is scalable in all kinds of data. However, it still suffers the inefficiency caused by recursive creations of conditional *FP-tree*, encounters great difficulties for very large databases because the number of local frequent patterns in all partitioned databases may be huge.

AFOPT, proposed in [LLXY03], is based on *FP*-growth approach. AFOPT adopts the top-down strategy and proposes a compact dta structure-Ascending Frequency Ordered Prefix-Tree (AFOPT) to represent the conditional database. The top-down traversal strategy is capable of minimizing the traversal cost of a conditional database. The ascending frequency order method is capable of minimizing the total number of conditional databases. Intuitively, an itemset with high frequency will very possibly have more frequent extensions than an itemset with lower frequency. That means if the most infrequent item are taken in front, the number of its frequent extension cannot be very large, so smaller and/or less prefix-trees will be build in the following mining. With mining going on, the items become more and more frequent, but their candidate extension sets become smaller and smaller, their prefix-trees cannot be very large. On sparse databases where the compression ratio of the AFOPT

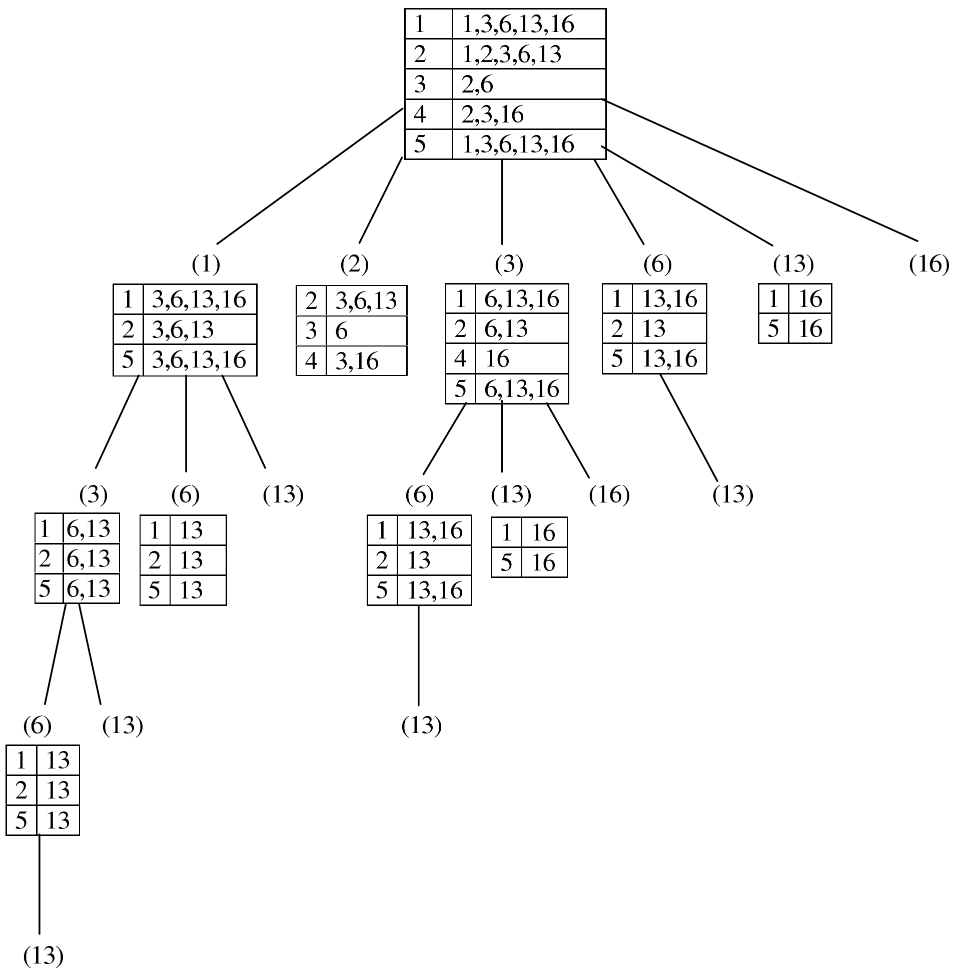


Figure 2.6: Projection of a sample database

structure is low, [LLXY03] chooses to first construct hyper-structure from the database, then constructs AFOPT structures from the hyper-structure. The experiment results show that AFOPT is more efficient than *FP-growth* algorithm.

OpportuneProject is proposed by Liu et al. [LPWH02] for mining complete set of frequent itemsets by projecting databases to grow a frequent itemset tree, abbreviated as *FIST* . Figure 2.6 illustrates the basic idea by of constructing *FIST* by projection.

To achieve maximized efficiency and scalability, the algorithm opportunistically chooses between two different structures, array-based and tree-based, to represent projected transaction subsets, and heuristically decides to build unfiltered pseudo projection or to make a filtered copy according to features of the subsets. The algorithm *OpportuneProject* creates the upper portion of $FIST$ by a recursive BreadthFirst procedure. Suppose the BreadthFirst procedure stops at level k. Then the lower portion of the $FIST$ are generated by a GuidedDepthFirst procedure.

The experimental results show that *OpportuneProject* is not only the most efficient on both sparse and dense databases at all levels of support threshold, but also highly scalable to very large databases.

2.3.6 Comparison of the Algorithms

There are several efficient algorithms that cope with the popular and computationally expensive task of frequent pattern mining, as we described above. Actually, these algorithms are more or less described on their own. The performance of each algorithm is various to both the data sets and the minimum support thresholds. Moreover, different implementations of the same algorithm could still result in significantly different performance results. At least there is no algorithm that is fundamentally beating out the other ones. An interesting performance comparison on the real word datasets presented by Zheng et al. in [ZKM01], in which five well-known association rule mining algorithms are compared on three new real-world data sets. Hipp et al. [HGN00a] implemented several algorithms to mine frequent itemsets, namely *Apriori*, *DIC*, *Partition*, and *Eclat*, in C++. Goethals, in [Goe02, Goe03], implemented the most common algorithms such as *Apriori*, *Eclat*, *Hybrid*, and *FP-growht* in C++.

In this section, we made a comparison of the algorithms based on these implementation.

For the sparse data set, *Eclat* performs much worse than other algorithms. This is because that a huge amount of candidate 2-itemsets is generated. As the support threshold is high, the frequent items are short and the number of item is not large. *FP-growth* is slightly worse than *Apriori*. This result is due to the overhead created by the maintenance of the *FP-tree* structure. As the support threshold goes down, the advantages of *FP-growth* over *Apriori* are becoming more impressive. The reason for the lousy performance of *Apriori* is because of some very large transactions for which the procedure for counting the supports of all candidate itemsets consumes most of the time.

For the dense data set, the hybrid algorithm performed best when the switching point is after the second iteration. Both *Eclat* and *FP-growth* gain good performance, and the performance differences of them are negligible. *Apriori* runs extremely slow as the minimum support thresholds becoming low.

2.4 Summary of Algorithms for Mining Frequent Patterns

Extensive efforts has been put on developing efficient algorithms for mining frequent patterns. In general, two typical approaches are proposed: candidate generate-and-test approach and pattern growth approach, the latter is shown to superior to the former significantly, especially on dense datasets or with low minimum support threshold. In this section, we will give a brief summary of these two kind of algorithms.

The performance advantages and bottlenecks of the generate-and-test approach are as follows.

- Cores:
  - Use frequent $(k-1)$-itemsets to generate candidate k-itemsets;
  - Use database scans and pattern matching to collect counts of the candidate itemsets
- Advantages:
  - Use large itemset property that can prune candidates dramatically;
  - Easily parallelized;
  - Easy to implement.
- Bottlenecks:

  - Many database scans are very costly. One needs (n+1) scans of the database, here n is the length of the longest pattern;
  - Mining long patterns needs generates lot of candidates. For example, if there are 10^4 frequent 1-itemsets, one needs to generate more than 10^7 frequent 2-candidates. To find a frequent pattern of size 100, eg., $\{i_1 i_2 \ldots i_{100}\}$, $2^{100} \approx 10^{30}$ candidates are needed to generate. This is inherent of candidate generation, no matter what implementation technique is applied;
  - Meet challenges when mining with many tough constrains, like $avg() \geq v$.

The performance advantages and bottlenecks of pattern growth approaches are as follows.

- Cores:
  - Adopt a divide-and-conquer approach to decompose both the mining tasks and the databases;
  - Use a compact data structure to compress crucial information about frequent patterns;
  - Use a pattern fragment growth method to discover all frequent patterns.
- Advantages:
  - Avoid the costly candidate-generation-and-test processing at all;
  - Avoid costly repeated database scans;
  - Not only efficient but also effective in constraint-based frequent pattern mining and sequential pattern mining [PH00, PH02].
- Bottlenecks:
  - Employ sophisticated data structures. Compare to *Apriori*-like algorithm, it is difficult to implement these data structures;

  – The recursive mining process to mine these structures is too voracious in memory resources.

All algorithms above employ sophisticated in-memory data structures, that imposes a limitation on the size of data that can be processed. However using RDBMSs provides us the benefits of using their buffer management systems so that the user /applications can free from the size considerations of the data. RDBMSs also give us the advantages of mining over very large datasets as they have the capabilities of managing such large volumes of data. Keeping this in mind, our focus in this thesis is on the use of SQL and some of the Object Relational constructs provided by RDBMSs for frequent pattern mining.

Chapter 3

Integration of Mining with Database

Researchers started to focus on issues to integrate mining with databases [AIS93, Imi96, IM96]. In [IM96], Imielinski et al. suggested that data mining needed new concepts and methods specially for processing data mining queries. They foresaw the need to develop a second generation data mining systems for managing data mining applications similar to DBMSs that manage business applications. Much of the subsequence research in this area has focused on developing tightly coupled systems that make use of DBMS features for data mining.

The research on database integration with mining can be broadly classified into two categories: one which proposes new mining operator and the other which leverages the query processing capabilities of current relational DBMSs. We review some proposals in this section.

3.1 Language Extensions

In the first category, there have been language proposals to extend SQL to support mining operators. A few example are DMQL [HFK$^+$96, HFW$^+$96], MSQL [IVA96, Vir98, IV99], and the Mine rule operator [MPC96]. We will give a brief description in the following.

3.1.1 *MSQL*

Imielinski at al. introduced the *MSQL* [IVA96] language which extends SQL with a special unified operator *MINE* to generate and query a whole set of propositional rules. The *MINE* operator as a query language primitive for database mining that can be embedded in a general programming language in order to provide an Application Programming Interface. *MSQL* comprises four basic statements:

- Create Encoding that encodes continuous valued attributes into discrete values.
- A *GetRules* query computes rules from the data and materializes them into a rule database. Its syntax is as follows:

  [Project Body, Consequent, confidence, support]

  GetRules(c) [$asR1$] [into ⟨*rulebase_name*⟩]

  [where ($RC|PC|MC|SQ$]

  [$sql - group - by - clause$] [$using - encoding - clause$]

- A *SelectRules* query is used for rule post processing, i.e., querying previously extracted rules. Its syntax is as follows:

  SelectRules(*rulebase_name*) [where ⟨*conditions*⟩]

- *Satisfies* and *Violates*, that allows to cross-over data and rules.

By using *SelectRules* on rulebases and *GetRules* on data, *MSQL* is possible to query rules as well as data. By using *Satisfies* and *Violates*, it is quiet simple to test rules against a dataset and to make crossing-over between the original data and query results. *MSQL* has been designed to be an extension of classical SQL, making the language easy to understand.

3.1.2 *DMQL*

DMQL was proposed by Han et al. [HFK+96] as another query language for data mining on relational database, which extends SQL with a collection of operators for classification rules, characteristics rules, association rules, etc. It consists of four primitives in data mining: (1) the set of data in relevance to a data mining process, (2) the kind of knowledge to be discovered, (3) the background knowledge, and (4) the justification of the interestingness of knowledge (i.e., thresholds). *DMQL* adopts an SQL-like syntax to facilitate high level data mining and natural integration with relational query language SQL. It is defined in an extended BNF grammar, where "[]" represents 0 or more occurrence, "{}" represents 0 or more occurrence, and words in sans serif font represent keywords, as shown below.

$< DMQL > :=$
use database $\langle database\_name\rangle$
$\{$use hierarchy $\langle hierarchy\_name\rangle for\langle attribute\rangle\}$
$\langle rule\_spec\rangle$
relate to $\langle attr\_or\_agg\_list\rangle$
from $\langle relation(s)\rangle$
[where $\langle condition\rangle$]
[order by $\langle order\_list\rangle$]
$\{$with $[\langle kinds\_of\rangle]$ threshold $= \langle threshold\_value\rangle[for\langle attribute(s)\rangle]\}$

Example 3.1 *This example shows how to use DMQL to specify the task relevant data in a hospital database to find association rules.*

use database $Hospital$
mine *association rules* as $Heart\_Health$
related to $Salary$, Age, $Smoker$, $Heart\_Disease$
from $Patient\_Financial\ f$, $Patient\_Medical\ m$
where $f.ID = m.ID$ and $m.age \geq 18$
with *support threshold* $= .05$
with $confidence$ *threshold* $= .7$

3.1.3 *MINE RULE*

Meo et al. described *MINE RUlE* [MPC96] as an extension to SQL, extracting several variations to association rule specifications from the database and storing them in the database in a separate relation. The general syntax of $MINE\ RULE$ follows:

$\langle MineRuleOp \rangle$:= MINE RULE $\langle TableName \rangle$ as
select distinct $\langle BodyDescr \rangle, \langle HeadDescr \rangle$ [, *support*] [, *confidence*]
[where] $\langle WhereClause \rangle$
from $\langle FormList \rangle$ [where $\langle WhereClause \rangle$]
group by $\langle AttrList \rangle$ [having $\langle HavingClause \rangle$]
[cluster by $\langle AttrList \rangle$ [[having $\langle HavingClause \rangle$]]
extracting rules with support :$\langle real \rangle$, confidence :$\langle real \rangle$

$\langle BodyDescr \rangle$:= [$\langle CardSpec \rangle$] $\langle AttrList \rangle$ as body
$\langle BodyDescr \rangle$:= [$\langle CardSpec \rangle$] $\langle AttrList \rangle$ as head
$\langle CardSpec \rangle$:= $\langle Number \rangle$ .. ($\langle Number \rangle | n$)
$\langle AttrList \rangle$:= $\langle AttrName \rangle$[, $\langle AttrList \rangle$]

$MINE\ RULE$ allows to dynamically partition the source relation into a first and a second level grouping (the clusters) from which more sophisticated rule constrains can be applied. Furthermore, it looks as the only language having an algebraic semantics [BBMM04], an important factor for in-depth study of optimization issues.

These proposals, as extensions of a database query language, however, do not address the processing techniques for these operators inside a database engine and the interaction of the standard relational operators and the proposed extension.

3.2 Frequent Pattern Mining in SQL

In the second category, researchers have addressed the issue of exploiting the capabilities of conventional relational systems and their object-relational extension to execute mining operations. This entails transforming the mining operations into database queries and in some cases developing newer techniques that are more appropriate in the database context.

The issue of tightly coupling a mining algorithm with a relational database system from a system point of view was addressed in Agrawal and Shim [AS96]. This proposal makes use of user-defined functions (UDFs) in SQL statements to selectively push parts of the computation into the database system. The objective was to avoid one-at-a-time retrieval from the database to the application address space, saving both the copying and process context switching costs. The first SQL-based approach is $SETM$ algorithm [HS95] described in the literature. The subsequent work [PSTK99, YPK00] suggested improvements of $SETM$ by the utilization of SQL query customization and database tuning. The results have shown that $SETM$ does not perform well on large datasets. There are some new approaches proposed to mine frequent patterns [STA98], for example K-Way Joins, Three-Way Joins, Subquery-based, and Two group-bys. These new algorithms are on the base of *Apriori*-like approach. They use the same statement for generating candidate k-itemsets and differ only in the statements used for support counting. The authors report that K-Way Joins approach is the best algorithm overall compared to the other approaches based on SQL-92. [STA98] also use object-relational extensions in SQL like UDFs, BLOBs, Table function etc. to improve performance. As a byproduct of this study, [STA98] identify some primitives for native support in database system for data mining applications.

3.2.1 Candidate Generation in SQL

Recall that the *Apriori* for discovering frequent itemsets proceeds in a level-wise manner. In the kth pass, we need to generate a set of candidate itemsets C_k from frequent itemsets F_{k-1} of the previous pass.

The statement creates a new candidate k-itemsets by exploiting the fact that all of its k subsets of size k-1 have to be frequent. First, in the *join* step, a superset of the candidate itemsets C_k is generated by joining F_{k-1} with itself as shown in Figure 3.1 [STA98].

Next, in the *pure* step, all itemsets $c \in C_k$, where some (k-1)-subset of c is not in F_{k-1}, are deleted. [STA98] performed the prune step in the same SQL statement as

```
insert into $C_k$ select $I_1.item_1, \ldots, I_1.item_{k-1}, I_2.item_{k-1}$
from        $F_{k-1}\ I_1,\ F_{k-1}\ I_2$
where       $I_1.item_1 = I_2.item_1$ and
                        ⋮
            $I_1.item_{k-2} = I_2.item_{k-2}$ and
            $I_1.item_{k-1} < I_2.item_{k-1}$
```

Figure 3.1: Candidate generation phase in SQL-92

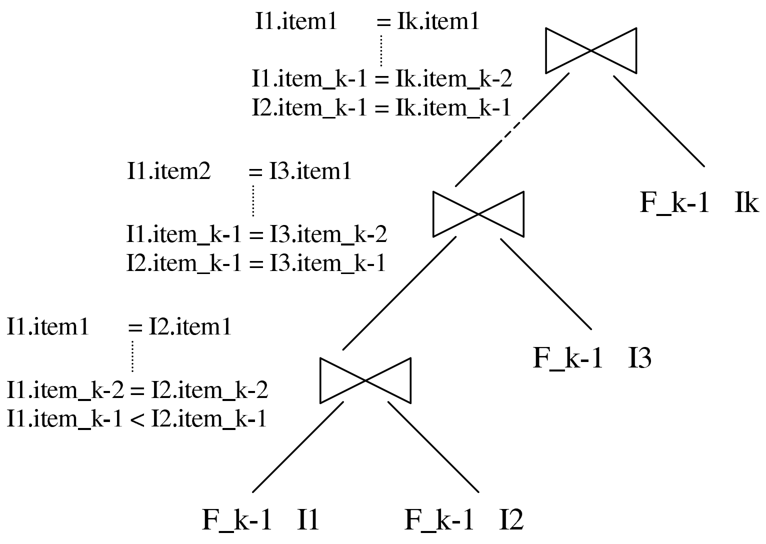

I1.item1 = Ik.item1
I1.item_k-1 = Ik.item_k-2
I2.item_k-1 = Ik.item_k-1
I1.item2 = I3.item1
I1.item_k-1 = I3.item_k-2
I2.item_k-1 = I3.item_k-1
F_k-1 Ik
I1.item1 = I2.item1
I1.item_k-2 = I2.item_k-2
I1.item_k-1 < I2.item_k-1
F_k-1 I3
F_k-1 I1
F_k-1 I2

Figure 3.2: Candidate generation for any k

the join step above by writing it as a $k - way$ join as shown in Figure 3.2.

3.2.2 Counting Support in SQL

This is the most time-consuming part of the frequent itemset mining algorithms. The candidate itemsets C_k and the data table T are used to count the support of the items in C_k. In this section, we review *K-Way join*, *Subquery*, *GatherJoin*, and *Vertical* approaches [STA98] for support counting. The first two represent the better ones among the SQL-92 approaches. The last two are based on SQL object-relational extensions.

```
insert into $F_k$ select $item_1, \ldots, item_k$, count(*)
from        $C_k$, T $t_1, \ldots$ T $t_k$
where       $t_1$.item = $C_k.item_1$ and
                      ⋮
            $t_k$.item = $C_k.item_k$ and
            $t_1$.tid = $t_2$.tid and
                      ⋮
            $t_{k-1}$.tid = $t_k$.tid
group by    $item_1, item_2 \ldots item_k$
having      count(*) $\geq minsupp$
```

Figure 3.3: Support counting by *K-Way join*

K-Way join, uses k instances of the transaction table T and joins it k times with itself and with a single instance of C_k. The statement of *K-Way join* is illustrated in Figure 3.3. This approach is simple and easy to use and understand. But it would involve a lot of multi-ways join.

Subquery makes use of common prefixes between the itemsets in C_k to reduce the amount of work done during support counting. The subquery approach breaks up the support counting into a cascade of k subqueries. The l-th subquery Q_l finds all tids that match the distinct itemsets formed by the first l columns of C_k (call it d_l). The output of Q_l is joined with transaction table T and d_{l+1} to get Q_{l+1}. The queries and the tree diagram for subquery Q_l are given in Figure 3.4. The experimental results presented in [STA98] show that the subquery optimization gave much better performance than the basic *KwayJoin* approach.

The overall observation in [STA98] was that mining implementations in pure SQL-92 are too slow to be practical.

GatherJoin is based on the use of table functions. It generates all possible k-item combinations of items contained in a transaction, joins them with the candidate table C_k and counts the support of itemsets by grouping the join result. It uses two table functions *Gather* and *Comb*$_k$. The table function *Gather* collects all the items of a transaction in memory and outputs a record for each transaction. Each record

```
insert into $F_k$ select $item_1, \ldots, item_k$, count(*)
from        (Subquery $Q_k$) t
group by    $item_1, item_2 \ldots item_l$
having      count(*) $\geq minsupp$
```

```
Subquery $Q_l$ $(1 \leq l \leq$ k)
    select $item_1, \ldots, item_k$, tid
    from  T $t_l$, (Subquery $Q_{l-1}$) as $r_{l-1}$,
            (select distinct $item_1 \ldots item_l$ from $C_k$) as $d_l$
    where $r_{l-1}.item_1 = d_l.item_1$ and ... and
            $r_{l-1}.item_{l-1} = d_l.item_{l-1}$ and
            $r_{l-1}.tid = t_l.tid$ and
            $t_l.item = d_l.item_l$
Subquery $Q_0$: No subquery $Q_0$
```

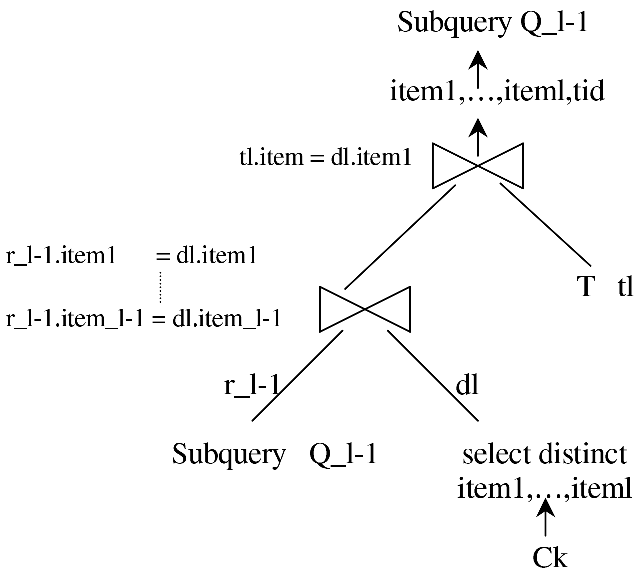


Figure 3.4: Support counting using *subquery*

consists of two attributes, the *tid* and $item - list$ which is a collection of all its items in a VARCHAR or BLOB. The table function $Comb_(k)$ accepts the output of *Gather* and combines k-item formed out of the items of a transaction. Figure 3.5 presents SQL queries for this approach.

$Vertical$ first transforms the transaction table into a vertical format and then count the support of itemsets by merging together this tid-lists. The table function *Gather* is used to create the tid-lists which are represented as BLOBs and stored in a new TidTable with attributes $(item, tid - list)$. The SQL query which does the transformation to vertical format is given in Figure 3.6. In the support counting phase, for each itemset in C_k the tid-lists of all k items is collected and a UDF is used to count the number of *tids* in the intersection of these k lists.

The approaches that made use of the object-relational extensions performed much better than the SQL-92 approaches above [STA98]. As a result, it's possible to mine the frequent patterns efficiently in SQL with a good understanding of the performance of a DBMS engine.

```
insert into F_k select item_1, ..., item_k, count(*)
from        C_k, (select t_2.T_itm_1, ..., t_2.T_itm_k from T,
            table (Gather(T.tid, T.item)) as t_1,
            table (Comb-k(t_1.tid, t_1.item-list)) as t_2)
where       t_2.T_itm_1 = C_k.item_1 and
                          ⋮
            t_2.T_itm_k = C_k.item_k
group by    C_k.item_1, ..., C_k.item_k
having      count(*) ≥ minsupp
```

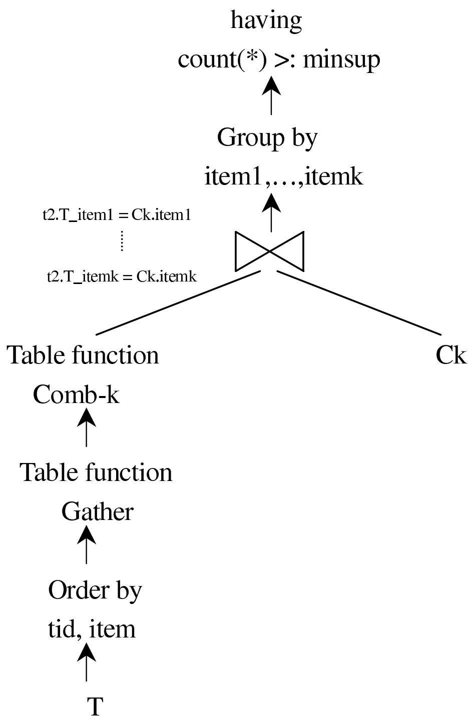


Figure 3.5: Support counting by *GatherJoin*

```
insert into TidTable
select      item_1, tid-list from
            (select * from T order by item, tid) as t1,
                    table(Gather(item, tid, minsupport)) as t2
```

Figure 3.6: Tid-lists creation by *Gather*

Chapter 4

SQL Based Frequent Pattern Mining with *FP-growth* Approach

We presented the conventional frequent pattern mining method, *Apriori*, and the novel frequent pattern mining method, *FP-growth*, in Chapter 2. As analyzed, the bottleneck for *Apriori*-like methods is the candidate-generation-and-test. While, *FP-growth* approach avoids such costs in *Apriori*-like approaches. The performance study [HPY00, HPYM04] shows that *FP-growth* algorithm outperforms the current candidate pattern generation-based algorithms in mining both long and short patterns.

SQL approach, as described in Section 1.2.1, has many advantages such as seamless integration with existing system and high portability. Unfortunately, the performance of SQL-based frequent pattern mining is known to fall behind specialized implementation since the prohibitive nature of the cost associated with extracting knowledge, as well as the lack of suitable declarative query language support. We presented some methods based on a relational database standard in Chapter 3. All of them adopt *Apriori*-like algorithms, which suffer from the inferior performance since the candidate-generation-and-test operation.

Can we develop new SQL-based algorithms which avoid the combinatory problem inherent to candidate-generation-and-test approach. To attack this problem, fist, we propose a SQL-based *FP-tree* mining approach in Section 4.2. Then, in Section 4.3,

TID	$Item$
1	a
1	b
1	c
1	d
2	c
2	d
3	a
3	d
3	e

(a) SC model

TID	$Item_1$	$Item_2$	$Item_3$	$Item_4$
1	a	b	c	d
2	c	d		
3	a	d	e	

(b) MC model

Figure 4.1: Example illustrating the SC and MC data models

we propose an improved FP algorithm, which introduce an extended FP table. In Section 4.4, we implement our SQL based frequent pattern mining approaches on RDBMSs, and to report experimental results and performance studies.

4.1 Input Format

The physical data model used for association rule mining has a significant impact on performance. The *single−column* (SC) data model is a common physical data model used in association rule mining, where transaction data, as the input, is transformed into a table T with two column attributes: transaction identifier (*tid*) and item identifier (*item*). In that case, for a given *tid*, typically there are multiple rows in the transaction table corresponding to different items in the transaction. The number of items per transaction is variable and unknown during table creation time. The alternative, $multi - column$ (MC) data model, was proposed by Rajamani et al. [RCIC99]. The MC model has the scheme (*tid*, $item_1$, $item_2$, ..., $item_k$). In this model, the column $item_i$ contains the ith item of that particular transaction. Figure 4.1 provides an example to illustrate the different representation with the SC and MC.

The MC model, as stated by the authors, is the better physical data model compared to the SC model. However, it's not practical in most cases because:

DataSet	*Numberofnodes*	*SizeofFP-tree*	*Compression*
$T40I10D100K$	3912459	$68650K$	89%

Table 4.1: Memory usage of *FP-growth*

- Often the number of items per transaction can be more than the maximum number of columns that the database supports.
- There will be lot of space wastage if the number of items per transaction is skew. For example, the maximum number of items per transaction is 800, and the corresponding average number of items per transaction is only 9.

Hence, we use SC model for the input data in this thesis.

4.2 *FP-growth* in SQL

One of the advantages of *FP-growth* over other approaches is that it constructs a highly compact *FP-tree*, which is usually substantially smaller than the original database and thus saves the costly database scans in the subsequent mining processes.

Although an *FP-tree* is rather compact, it is unrealistic to construct a main memory-based *FP-tree* when the database is large. The *FP-tree* consists of a trie data structure in which each node stores an item as well as a counter, also a link pointing to the next occurrence of the respective item in the *FP-tree*. Additionally a header table is stored containing each separate item together with its support and a link to the first occurrence of the item in the *FP-tree*. In *FP-growth*, the cover of an item is compressed using the linked list starting from its node-link in the header table, but every node in this linked list needs to store its label, a counter, a pointer to the next node, a pointer to its branches and a pointer to its parent. Therefore, the size of such an complex *FP-tree* should be large. Table 4.1 shows for T40I10D100K the total number of nodes in *FP-growth* and the compression rate of the *FP-tree*.

However using RDBMSs provides us the benefits of using their buffer management systems specifically developed for freeing the user applications from the size considerations of the data. And moreover, there are several potential advantages of building mining algorithms to work on RDBMSs as described in Section 1.2.1. An interesting alternative is to store an *FP-tree* in a relational table and to propose algorithms for frequent patterns mining based on such a table.

We study two approaches in this category - *FP*, *EFP* (Extended Frequent Pattern) [SSG04b, SSG04c, SSG05]. They are different in the process of constructing frequent pattern tree table, named *FP*. *FP* approach checks each frequent item of each transaction table one by one to decide whether it should be inserted into the table *FP* or not to construct *FP*. *EFP* approach introduces an extended frequent pattern table, named *EFP*, which collects the set of frequent items and their prefix items of each transaction, thus table *FP* can generate from *EFP* by combining the items which share a common prefix.

4.2.1 Construction of the *FP* Table

FP-tree is a good compact tree structure. In addition, it has two properties: node-link property (all the possible frequent patterns can be obtained by following each frequent's node-links) and prefix path property (to calculate the frequent patterns for a node i in a path, only the prefix sub-path of i in the path need to be accumulated). For storing the tree in a RDBMS a flat table structure is necessary. According to the properties of *FP-tree*, we represent an *FP-tree* by a table *FP* with three column attributes:

- Item identifier (*item*). Since only the frequent items play a role in the frequent pattern mining, the items in the table *FP* consists of the set of frequent items of each transaction.

- The number of transactions that contain this item in a sub-path (*count*). If multiple transactions share a set of frequent items, the shared sets can be merged with the number of occurrences represented as *count*.

- Item prefix path (*path*). The prefix path of an item is represented by the set of frequent items listed earlier than the item of each transaction.

```
insert into T′ select t.id, t.item
from        T t, ((select item, count(*) from T
            group by item
            having count(*) ≥ minsupp
            order by count(*) desc ) as F (item, count))
where       t.item = F.item
```

Figure 4.2: SQL query using to generate T'

The field *path* is beneficial not only to construct the table FP but also to find all frequent patterns from FP. In the construction of table FP, the field *path* is an important condition to judge whether the frequent item should be inserted into the table FP or not. If an item does not exist in the table FP or there exist the same items as this item in the table FP but their corresponding *path* are different, insert the item into table FP. Otherwise, update the table FP by incrementing the item's count by 1. In the process of mining frequent patterns using FP, we need to recursively construct conditional frequent pattern table, named $ConFP$, for each frequent item. Due to the *path* column of the table FP storing the set of prefix items of each frequent item, it's easy to find the items which co-occurs with this item by deriving all its *path* in the table FP.

The process of the table FP construction is as following:

1. Identify the set of frequent items. This can be done by transferring the transaction table T into table T', in which infrequent items are excluded. The size of the transaction table is a major factor in the cost of joins involving T. It can be reduced by pruning the non-frequent items from the transactions after the first pass. We insert the pruned transactions into table T' which has the same schema as that of T. In the subsequent passes, join with T can be replaces by join with T'. This could result in improved performance especially for datasets which contains lot of non-frequent items. SQL query using to generate T' from T is illustrated in Figure 4.2.

2. Construct the table FP. To describe the process of building FP, let's first

TID	*Item*
1	1
1	3
1	4
1	6
1	7
1	9
1	13
1	16
2	1
2	2
2	3
2	6
2	12
2	13
2	15
...	...

(a) An example table T

Item	*Count*
3	4
6	4
1	3
2	3
13	3
16	3

(b) An example table F

TID	*Item*
1	3
1	6
1	1
1	13
1	16
2	3
2	6
2	1
2	2
2	13
...	...

(c) An example table T'

Figure 4.3: Example table T, F, and T'

examine an example as follows.

Example 4.1 *Let the transaction database, DB (same as the transaction database used in Example 2.3), be stored in the table T, and minimum support threshold be 3. The table T' consists of the frequent items of the table T as shown in Figure 4.3.*

A table FP can be built as the following procedure.

- Sort the items of T' in descending order of their frequency. If the frequent items are sorted in their frequency descending order, there are better chances that more prefix items can be shared.

- For the first transaction in the table T', we insert all the frequent items of it into the table FP and set the *count* of the items be "1". The *path* of each item can be built in the following. We set the *path* of the first item 3 be "null". The *path* of the second item 6 will be *null* : 3. The third item 1's *path* will be *null* : 3 : 6, and so on.

Item	*Count*	*Path*
3	4	*null*
6	3	*null* : 3
1	3	*null* : 3 : 6
13	2	*null* : 3 : 6 : 1
16	2	*null* : 3 : 6 : 1 : 13
2	1	*null* : 3 : 6 : 1
13	1	*null* : 3 : 6 : 1 : 2
6	1	*null*
2	1	*null* : 6
2	1	*null* : 3
16	1	*null* : 3 : 2

Table 4.2: The table FP in Example 4.1

- For the second transaction, we test each of the frequent items to see if it should be insert into the table FP or update the table FP by incrementing the count of the item, which has already appeared in FP and has the same *item* and *path* as the tested item, by 1. The checking conditions consist of two factors: *item* and *path*. If there is an item in the table FP has the same name and path as those of the tested item, then update FP. Otherwise, insert the tested item into FP. Since 3, 6, and 1 share a common prefix with the existing items in FP, the count of each item is incremented by 1. 2 and 13 are inserted into FP with the *path* of *null* : 3 : 6 : 1 and *null* : 3 : 6 : 1 : 2 and the *count* of 1.

- For the last transactions, we do the same process. After scanning all the transactions, the table FP is shown in Table 4.2.

Based on this description, we have the algorithm for constructing the table FP as follows.

4.2.2 Finding Frequent Pattern from FP

After the construction of a table FP, we can use this table to efficiently mine the complete set of frequent patterns. Like FP-*growth* approach, for each frequent item i we construct its conditional pattern base table PB_i, which has three column attributes

Algorithm 4 Table FP construction

```
Input: A transferred transaction table $T'$
Output: A table $FP$
Method:
1. curcnt := 1;
2. curpath := $null$;
3. find distinct $tid$ from the table $T'$
4. for each item $i_k$ of the first $tid$
5.     insert $i_k$ with curcnt and curpath into the table $FP$;
6.     curpath += $i_k$;
7. insertFP ($items$);


insertFP ($items$)
    if $FP$ has an item $f == i_1$ (the first item in the items) and $f.\ path == null$
      for each item $i_k$ in the items
          insert $i_k$ with curcnt and curpath into the table $FP$;
          curpath += $i_k$;
    else
      for each item $i_k$ in the items
          if $FP$ has an item $f == i_k$ and $f.\ path == i_k.\ path$
            curcnt = $i_k.\ count + 1$;
            update the table $FP$;
          else
            insert $i_k$ into the table $FP$;
      curpath += $i_k$;
```

$(tid, item, count)$. And then the conditional FP table, named $ConFP_i$, is built based on PB_i. The table $ConFP$ has the same scheme as that of the table FP. The mining process is successively constructing $ConFP$.

For each frequent item i, to find other patterns having item i, we need to access all transactions containing item i in the table FP. Table PB_i collects items that co-occur with i in the table FP. We can obverse that the *path* attribute in the table FP represents the information of prefix subpath set of each frequent item in a transaction. So the process of constructing PB_i is implemented by a simple *Select* query to get all corresponding counts and paths of i, then split these paths into multiple items. Then we construct the table $ConFP_i$ from each conditional pattern base table PB_i using the same algorithm as the table FP construction, and mine recursively in the table $ConFP_i$.

Let's visit the mining process based on the built FP table shown in Table 4.2.

According to the frequent items found in the process of constructing the table FP, $\{3:4, 6:4, 1:3, 2:3, 13:3, 16:3\}$, all frequent patterns in the database can be divided into 6 subsets without overlap:

1. patterns containing item 3;
2. patterns containing item 6 but no item 3;
3. patterns containing item 1 but no 3 nor 6;
4. patterns containing item 2 but no 3, 6, nor 1;
5. patterns containing item 13 but no 3, 6, 1, nor 2;
6. patterns containing item 16 but no 3, 6, 1, 2, nor 13;

1. We start from the frequent item with the minimum frequency, 16. To construct 16's conditional pattern base table PB_{16}, we select items whose attribute of *item* is 16 from the table FP. We can find that 16 has two different paths: $\langle null : 3 : 6 : 1 : 13\rangle$ with the *count* 2, $\langle null : 3 : 2\rangle$ with the *count* 1. Then we separately split these two paths as shown in Table 4.3. This process can be

TID	*Item*	*Count*
1	3	2
1	6	2
1	1	2
1	13	2
2	3	1
2	2	1

Table 4.3: An example table PB_{16}

```
select $count, path$ from $FP$ where item = $i$;
for each count $cnr$, path $p$
    $id := 1$;
    item[ ] = split($p$);
    for each item $a_i$ in item[ ]
        insert into $PB_i$ values $(id, cnr, i)$;
    $id$ += 1;
```

Figure 4.4: Construction of table PB

achieved as illustrated in Figure 4.4. Construction of a $ConFP$ on this PB_{16} leads to only one frequent item $\{3 : 3\}$. Hence only one frequent pattern (3 16 : 3) is derived. The search for frequent patterns associated with item 16 terminates.

2. Now, we mine the frequent patterns having item 13 but no item 16. We select items whose attribute of *item* is 13 from the table FP. Two paths in the table FP are found: $\langle null : 3 : 6 : 1\rangle$ with the *count* 2, $\langle null : 3 : 6 : 1 : 2\rangle$ with the *count* 1. Then we separately split these two paths to obtain the table PB_{13}. Constructing a conditional FP table on it, we derived a table $ConFP_{13}$ as shown in Table 4.4. This conditional FP table is then mined recursively by calling FindFP($ConFP_{13}$).

3. Similarly, we can mine the frequent pattern having other items till the last subset.

Item	*Count*	*Path*
3	3	$null$
6	3	$null : 3$
1	3	$null : 3 : 6$

Table 4.4: An FP table has a single path

Further optimization can be explored on a special kind of FP, which consists of one single prefix path. In the table FP, we use the *path* attribute to express the prefix path of each frequent item in the database. An FP is a single prefix-path if the *path* of the item is the *path* of the previous item combining with the the previous item. In that case, the set of frequent patterns is generated by enumeration of all the combinations of the distinct items in FP with prefix. Let us examine an example. In Table 4.4, the items $1, 6, 3$ occur in the same transaction, and the table FP consists of one single prefix path. Thus, in stead of building three conditional FP tables: $ConFP_1$, $ConFP_6$, and $ConFP_3$, the following set of frequent patterns is generated, $\{(1 : 3), (6 : 3), (3 : 3), (\{1\}\ \{6\} : 3), (\{1\}\ \{3\} : 3), (\{6\}\ \{3\} : 3), (\{1\}\ \{6\}\ \{3\} : 3)\}$.

Based on the above, we have the following algorithm for finding frequent patterns from table FP as showed.

Algorithm 5 FindFP

Input: A table FP constructed based on Algorithm 4
and a table F collecting all frequent itemsets
Output: A table $Pattern$, which collects the complete set of frequent patterns
Procedure:

```
1. if items in the table $FP$ in a single path
2.    combine all the items with prefix, insert into $Pattern$;
3. else
4.    for each item $i$ in the table $F$
5.        construct table $ConFP_i$;
6.        if $ConFP_i \neq \phi$
7.           call FindFP($ConFP_i$);
```

4.2.3 Optimization

In this subsection, we present two optimizations on the process of mining frequent patterns.

1. Optimize the query for generating frequent 1-itemsets. The SQL query using to generate T', in which infrequent items are excluded and frequent ones are sorted in descending order by frequency, is a subquery as shown in Figure 4.2. It could be expensive since a large number of intermediate results may be led when T is very large and dense. In that case, we materialize the intermediate results as table F to gain the performance improvement. It is shown as follows.

   ```
   insert into F select item, count(*)
   from        T
   group by    item
   having      count(*) ≥ minsupp
   ```

   ```
   insert into T' select T.id, T.item
   from        T, F
   where       T.item = F.item
   ```

2. Avoid materialization cost of some temporary tables. Notice that in the case that a database may generate a large number of frequent 1-itemsets, the total number of pattern based tables and conditional FP tables is usually large. If we build one table PB and one table $ConFP$ for each frequent item, then the cost of creating and dropping these temporary tables is certainly untrivial. In fact, pattern based table PB is only required to the process of constructing $ConFP$ of each frequent item. It can be cleared to use for the next frequent item after the $ConFP$ of the frequent item has been constructed. That is, one PB is created in the whole procedure. As we know, the $ConFP$ tables of all

frequent items are not constructed together. Each *ConFP* table is built, mined before the next *ConFP* table is built. The mining process is done for each frequent item independently with the purpose of finding all frequent k-itemset patterns in which the item at the level k-1 participates. In that case, we can use one *ConFP* table at each level. So that the number of temporary tables is dramatically decreased. For example, the maximum length of the frequent patterns mined is 9. The number of *ConFP* tables constructed during the whole procedure is 8. The total number of the tables constructed (including a *pattern* table that stores all frequent patterns mined and a *name* table that stores all temporary tables) is 15.

4.3 *EFP* Approach

We studied the SQL-based *FP-tree* method in the last subsection and found the construction of table *FP* (table *ConFP*) is the most time-consuming procedure in the whole procedure. The important reason is that in the process to construct *FP* table, one must judge every frequent item of each transaction one by one in the database to decide how to add the frequent item to the *FP*. That is, if the number of frequent items of each transaction is μ_i, the sum of the transactions in the database is θ, the number of frequent items in the table *FP* is λ which increases with *FP* construction proceeding, and then the cost of inserting all the frequent items into the *FP* is $O(w \times \lambda)$, where w is the sum of frequent items in the database:

$$w = \sum_{i=1}^{\theta} \mu_i$$

As we know, the value of w is enormous especially when the database is large, or when the minimum support threshold is quite low. Furthermore, the process of constructing table *FP* is a strict serial computing process, that is, the computing of latter item must be based on the former result. In that case, the test process is inefficient.

From the above discussions, we found the mining performance can be improved if one can avoid computing each frequent item individually. In this subsection, we introduce the extended FP table, called EFP. The table FP can be achieved based on the table EFP.

The EFP has the format as $(item, path)$. The difference between the EFP method and the FP method is the process of the constructing of the two tables. We can obtain EFP by straightly transforming frequent items in the transaction table T' without testing each of them one by one. We first initialize the path of the first frequent item i_1 of each transaction and set it as $\{null\}$. The path of the second frequent item i_2 is $\{null : i_1\}$, and the path of the third frequent item i_3 is $\{null : i_1 : i_2\}$, and so on. Let us revisit the mining problem in Example 4.1.

Starting at the first frequent item of the first transaction, 3, the path of 3 is set to $\{null\}$. after that, for the second item of the first transaction, 6, the path of 6 can be built to $\{null : 3\}$. Similarly, the path of the third item 1 can be built to $\{null : 3 : 6\}$. This process continues for building the rest of frequent items of all transactions. Then we get a table EFP illustrated in Table 4.5.

The table EFP represents all information of frequent itemsets and their prefix path of each transaction. Normally, the table EFP is larger than the table FP especially when items has more chances to share the same prefix path. However, compare to the FP construction, we do not need to test each frequent item to construct the table EFP and can make use of the database powerful query processing capability. We have the following algorithm for building the table EFP as follows.

The construction of the table EFP can be implemented using a recursive query, which iteratively uses result data to determine further results. Recursion is very powerful, because it allows certain kinds of questions to be expressed in a single SQL statement that would otherwise require the use of a host program. The recursion query for constructing the table EFP is illustrated in Figure 4.5.

To obtain the table FP, we combine the items with identical value of *path*. The SQL statement of construct FP from EFP is illustrated as follows.

```
insert into FP
select      item, count(*) as count, path from EFP
group by    item, path
```

Item	*Path*
3	*null*
6	*null* : 3
1	*null* : 3 : 6
13	*null* : 3 : 6 : 1
16	*null* : 3 : 6 : 1 : 13
3	*null*
6	*null* : 3
1	*null* : 3 : 6
2	*null* : 3 : 6 : 1
13	*null* : 3 : 6 : 1 : 2
6	*null*
2	*null* : 6
3	*null*
2	*null* : 3
16	*null* : 3 : 2
3	*null*
6	*null* : 3
1	*null* : 3 : 6
13	*null* : 3 : 6 : 1
16	*null* : 3 : 6 : 1 : 13

Table 4.5: The table EFP in Example 4.1

Algorithm 6 Table EFP construction

Input: A transferred transaction table T'
Output: A table EFP
Method:

```
1. for each transaction in $T'$
2.     call insertEFP ($items$);


insertEFP ($i_k$)
curpath := $null$;
for each item $i_k$ of the transaction
    insert $i_k$ into the table $EFP$;
    curpath += $i_k$;
```

```
for each tid $id$
    select item
    from  $T'$
    where id = $id$
    create table temp (id int, iid int, item varchar(20))
    iid := 1;
    for each item $i$
        insert into temp values(iid, i)
        iid += 1;
        with fp (item, path, nsegs) as
        (( select item, cast ('null' as varchar (200)), 1 from temp
            where iid = 1)
        union all
            (select t.item, cast (f.path ||' :' || f.item as varchar (200)), f.nesgs+1
            from fp f, temp t
            where t.iid = f.nsegs+1 and
            f.nesgs < iid))
    insert into $EFP$ select item, path from $FP$
```

Figure 4.5: Recursive query for constructing the table *EFP*

Let's now examine the efficiency of the approach. Basically, the table EFP is larger than the table FP especially when there are lots of items share the prefix strings. However, from the performance aspect, the EFP approach is more efficient than that of the FP method since the former can insert all the frequent items into the table EFP quickly by avoiding checking each item of each transaction one by one. To construct the table EFP, the cost is only related to w, where w is the sum of frequent items in the database. Compare to the cost of building FP, which is $O(w \times \lambda)$, the EFP approach's performance is enhanced significantly.

4.3.1 Using SQL with object-relational extension

In the following section, we study approaches that use object-relational extension in SQL to improve performance. We consider an approach that use a table function. Table functions are virtual tables associated with a user defined function which generate tuples on the fly. Like normal physical tables they have pre-defined schemas. Table functions can be viewed as user defined functions that return a collection of tuples instead of scalar values.

We use a table function *path* which collects the prefix path of items of each transaction. Here the prefix path of the first item i_1 in a transaction is set to be "null". For other items of the transaction the table function *path* generates the prefix path of them. For example, the prefix of the second item i_2 is $\{null : i_1\}$, the third item i_3's prefix is $\{null : i_1 : i_2\}$, and so on. This function is easy to code.

As a matter of fact, all approaches above have to materialize its temporary table namely T' which contains the frequent items in the table T and PB' which contains the frequent items in the table PB. Those temporary tables are only required in the construction of table FP and table $ConFP$. They are not needed for generating the frequent patterns. So we further use subquery instead of temporary tables. The data table T is scanned in the (tid, item) order and combined with the frequent itemsets table F to remove all infrequent items and sort in support descending order as F, and then passed to the user defined function $Path$. SQL query to generate FP using the user defined function *Path* as follows.

```
insert into $FP$ select $tt_2.item$, $tt_2$.count (*), $tt_2.path$
from        ( select $T.id$, $T.item$ from $T$, $F$
              where $T.item = F.item$
              order by $T.id$, $F.count$ desc) as $tt_1$,
              table (Path ($tt_1.id$, $tt_1.item$)) as $tt_2$
group by    $tt_2.item$, $tt_2.path$
order by    $tt_2.path$
```

4.3.2 Analysis

Almost all previous frequent patten mining algorithms with SQL consist of a sequence of steps proceeding in a bottom-up manner. The result of the kth step is the set of frequent itemsets, denoted as F_k. The first step computes frequent 1-itemsets F_1. The candidate generation phase computes a set of potential frequent k-itemsets C_k from F_{k-1}. The support counting phase filters out those itemsets from C_k that appear more frequently in the given set of transactions than the minimum support and stores them in F_k. The support counting process is the most time-consuming process. Most of these algorithms need to join with T several times to count the support of frequent itemsets. The size of transaction table is a major factor in the costs of joins involving T. FP approach reduces the cost by the following process:

- Pruning the non-frequent items from the transaction table T after the first pass;
- Extracting the concise information and store it into the table FP. The sequent procedures are based on FP instead of T.

In addition, the FP approach explores the table FP by pattern growth method so that it avoids the combination problem of candidate generation. So, the performance of the SQL-based FP approach is better than SQL-based *Apriori*-like algorithm.

4.4 Experimental Evaluation and Performance Study

In this subsection, we present a performance comparison of our approaches with the classical *Apriori* algorithm based on SQL, and a loose-coupling approach based on *FP-tree*. All of our experiments were performed on Version 8 of IBM DB2 EEE installed on Linux operation system with Pentium IV 2.0Ghz.

4.4.1 Data Set

We use synthetic transaction data set generated by the program provided by the Quest research group at IBM Almaden [AS] for experiment. Using this generator, one can vary the number of transactions, number of frequent itemsets per transaction in the data sets, distinct itemsets, and the length of frequent patterns, to generate different kinds of data sets. The nomenclature of these data sets is of the form TxxIyyDzzzK. Where xx denotes the average number of items present per transaction, yy denotes the average support of each item in the data set and zzzK denotes the total number of transactions in K (1000's).

We report experimental results on four data sets, they are respectively T5I5D10K, T25I10D10K, T10I4D100K and T25I20D100K. The first two datasets consist of 10 thousand transactions. Each containing an average of 5 items and 25 items. The average size of the maximal potentially frequent itemsets is 5 and 25 respectively .

The third one has 100 thousand transactions, each containing an average of 10 items and the average size of potentially frequent itemsets is 4. It is a sparse dataset. The frequent itemsets are short and not numerous.

The last one consists of 100 thousand transactions with an average 25 number of items per transaction and the average length of potentially frequent patterns is 20. There exist exponentially numerous frequent itemsets in this data set when the support threshold goes down. There are pretty long frequent itemsets as well as a large number of short frequent itemsets in it. It is a relatively dense dataset and contains mixtures of short and long frequent itemsets. Table 4.6 summarizes the parameters associated with the datasets.

Datasets	*#Records*	*#Transaction in thousands*	*Avg.#items*
$T5I5D10K$	49182	10	5
$T25I10D10K$	247151	10	25
$T10I4D100K$	996622	100	10
$T25I20D100K$	2531427	100	25

Table 4.6: Description of the generated datasets

4.4.2 Comparison between FP and EFP

For these experiment, we built (*tid*, *item*), (*item*, *tid*) index on the data table T and (*item*) index on the frequent itemsets table F and table FP. The goal was to let the optimizer choose the best plan possible. Considering table EFPs, table $ConFP$s and other temporary tables built during the whole procedure are modified frequently, we didn't build indexes on those tables, since each modification to a table must be reflected in all the indexes that are defined on the table, that carries a certain cost. The run time used here means the total execution time, that is the period between input and output.

We compare the FP approach and the EFP approach on data set T5I5D10K. In this data set, with the support threshold of 1%, the number of frequent 1-itemset is 225 and the total number of frequent 1-items in the transaction is 28316. The maximum length of the frequent patterns is 2. It is a relative sparse data set.

The experimental results show that EFP approach can get competitive performance out of FP implementation. An important reason for superior performance of EFP over FP is the avoid testing each frequent item one by one in the construction of table FP.

As seen from the result shown in Figure 4.6, the runtime of constructing table FP on data set T5I5D10K with the support value of 0.2%, in the FP approach, almost 97% of execution time belongs to the construction of table FP. However, in the EFP approach, almost less 49% of execution time belongs to the construction of table FP. In addition, the run time of constructing FP in EFP approach decreases dramatically.

In FP approach, the recursive construction of table $ConFP$ use the same method

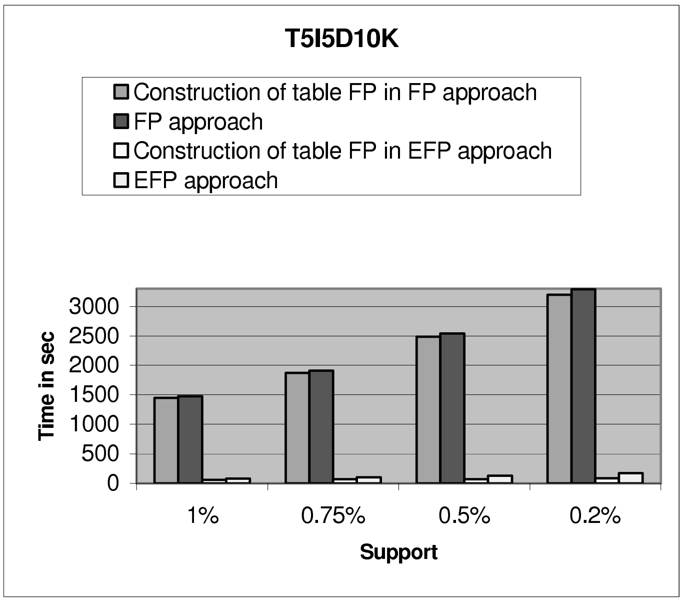


Figure 4.6: Comparison the construction of FP table between FP and EFP over data set T5I5D10K

as the construction of table FP. In that case, avoid the process of checking substantially reduce the expensive cost.

4.4.3 Comparison of Different Approaches

For comparison, we implement a loose-coupling approach based on *FP-tree* and a *k-way joins* approach based on *Apriori* algorithm.

- In the loose-coupling approach, access to the data table in DBMS was provided through a JDBC interface. The construction of the *FP-tree* and mining frequent patterns from the *FP-tree* are completed in memory.

- In *k-way joins* approach, which was found to be the best as for SQL-92 in [AS94], the candidate generation phase computes a set of potential frequent k-itemsets C_k from $F_k - 1$. Here, we optimize the *k-way joins* by pruning the non-frequent items from the transactions after the first pass. We insert the pruned transactions into table T_f. The support counting phase uses k instances of table T_f and joins it k times with itself and with a single instance of C_k to

```
insert into $F_k$ select $item_1, \ldots, item_k$, count(*)
from        $C_k$, $T_f$ $t_1, \ldots$ $T_f$ $t_k$
where       $t_1$.item = $C_k.item_1$ and
                 ⋮
            $t_k$.item = $C_k.item_k$ and
            $t_1$.tid = $t_2$.tid and
                 ⋮
            $t_{k-1}$.tid = $t_k$.tid
group by    $item_1, item_2 \ldots item_k$
having      count(*) $\geq minsupp$
```

Figure 4.7: Support counting by optimized *K-Way join*

filter out those itemsets from C_k that appear more frequently in the given set of transactions than the minimum support and store them in F_k. The statement of *K-Way join* in the experiment is illustrated in Figure 4.7.

Figure 4.8 shows the total time taken by the three approaches: *K-way join* approach, *loose-coupling* approach and SQL-based *EFP* approach, on data set T5I5D10K. From the graph we can make the following observation: *EFP* has the better performance than *K-way join* and *loose-coupling* approach.

In data set T5I5D10K, as the support threshold is high, the frequent items are short and the number of item is not large. The advantages of *EFP* over *Apriori* are not so impressive. *EFP* is even slightly worse than *Apriori*. For example, the maximal length of frequent patterns is 2 and the number of frequent itemsets is 444 with support threshold 0.5%, *Apriori* can finish the computation shorter than the time for *EFP*. This is because when the data is sparse, table *FP* contains tuples not as compressive as what is does on dense data sets. Moreover, constructing over sparse data sets recursively has its overload. However, as the support threshold goes down, the gap is becoming wider. For example, the maximal length of frequent patterns is 12 and the number of frequent patterns is 57742 with the support threshold 0.05%, *EFP* is much faster than *Apriori*. When the support threshold is low, the number of frequent patterns as well as that of candidates are non-trivial. In contrast, *EFP* avoid

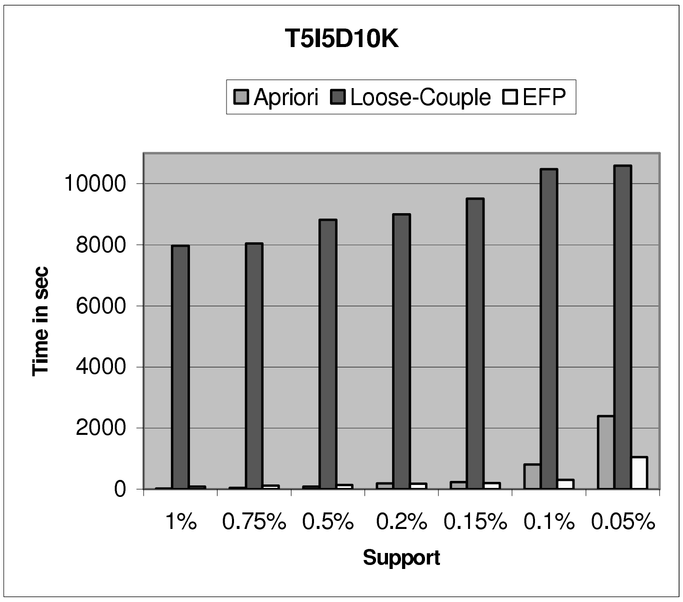


Figure 4.8: Comparison for dataset T5I5D10K

candidates generation and test. That is why *EFP* can get significant performance improvement.

From Figure 4.9 to 4.11, we show the performance comparison between *K-way join* and *EFP* on T25I10D10K, T10I4D100K and T25I20D100K.

T25I10D10K is a relatively dense data set. As we can see, most patterns are of short lengths when the support threshold is high. However, when the support threshold becomes low, most items are frequent. Then, the advantages of *EFP* over *Apriori* are distinct.

Data set T10I4D100K is a very sparse data set. The performance comparison for these two approaches on such data set is similar as that of on T5I5D10K. When the support threshold is 0.1%, the maximal length of frequent patterns is 10.

In data set T25I20D100K, which contains abundant mixture of long and short frequent patterns, the result is shown in Figure 4.14. It can be seen, the advantage of *EFP* is dramatic in such a data set.

We studied the performance of *Path*, a user defined table functions (Path).

The scalability of *Apriori* and *EFP* on synthetic data set T10I4D100K is shown in Figure 4.13. *EFP* can mine with support threshold as low as 0.02%, with which

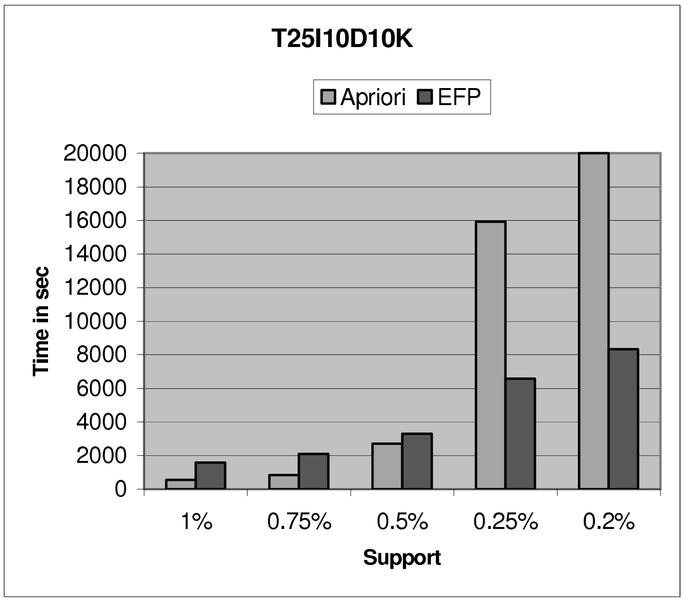


Figure 4.9: Comparison for dataset T25I10D10K. For *K-Way join* with the support threshold that are lesser than 0.2%, the running times were so large that we had to abort the runs in many cases.

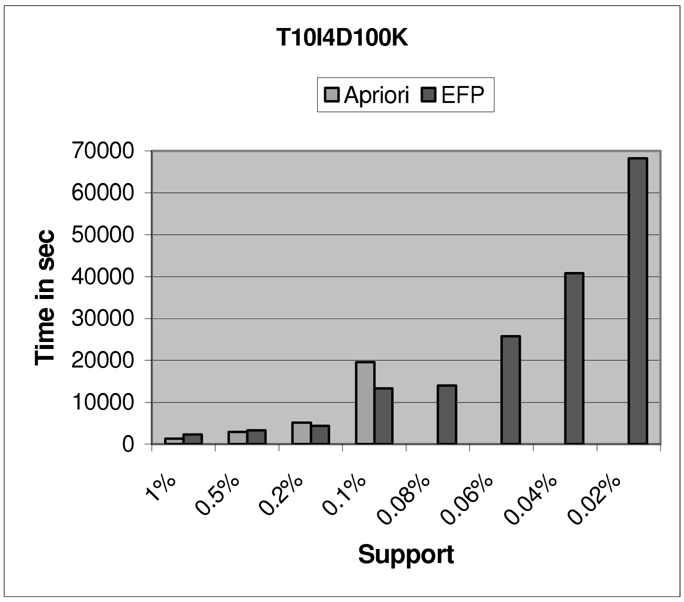


Figure 4.10: Comparison for dataset T10I4D100K. For K-Way join approach with the support value of less than 0.08%, the running times were so large that we had to abort the runs in many cases.

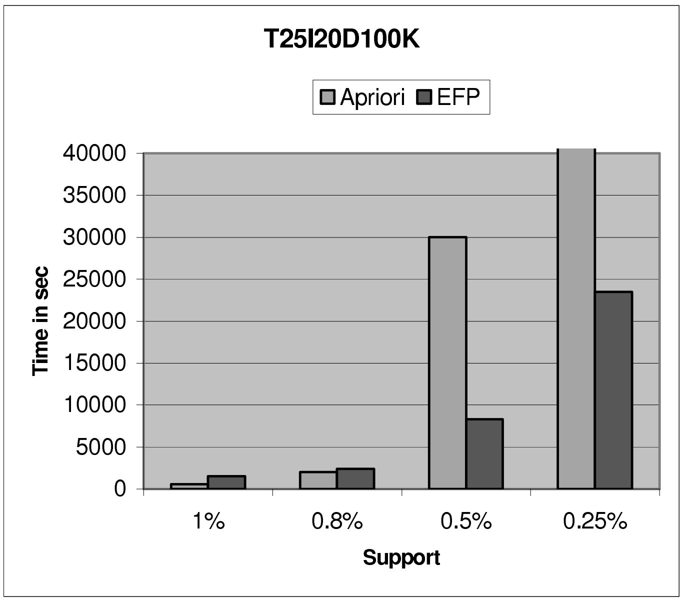


Figure 4.11: Comparison for dataset T25I20D100K. For K-Way join approach with the support value of 0.25%, the running times were so large that we had to abort the runs in many cases.

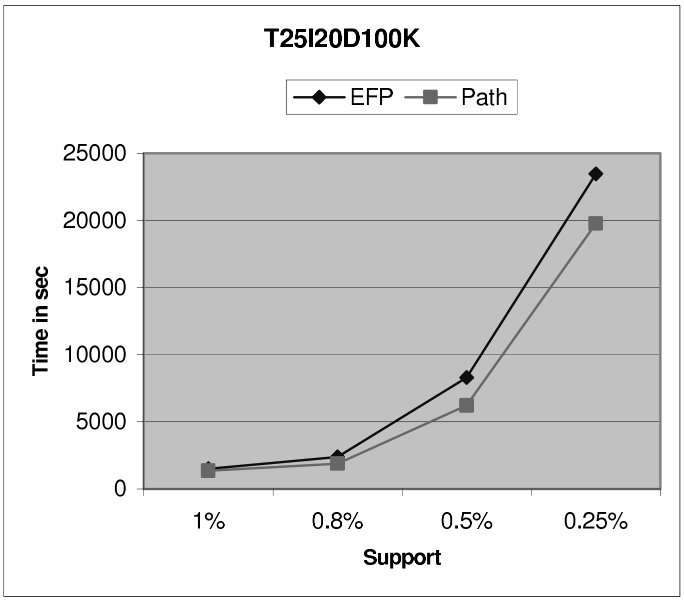


Figure 4.12: Comparison between *EFP* and *Path* over data set T25I20D100K

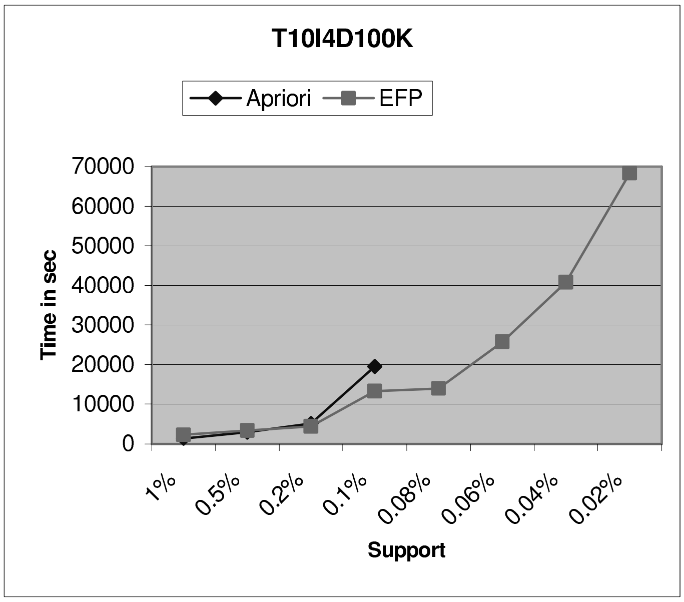


Figure 4.13: Scalability with the threshold over T10I4D100K

Apriori cannot work out within reasonable time.

4.5 Conclusion

In this chapter, we have implemented SQL-based frequent pattern mining using *FP-growth*-like approach.

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

1. It represents *FP-tree* using a relational table *FP* and proposed a method to construct this table. To improve its performance, a table called *EFP* is introduced, which is in fact stores all information of frequent item sets and their prefix path of each transaction. And then, table *FP* can derived from table *EFP*. Compare to the construction of *FP*, the process of the construction of *EFP* avoid testing each frequent item one by one. Table *FP* is usually substantially smaller than the transaction table T and is to be used in the subsequent mining processes. Thus it saves the costly join with the original transaction

table.

2. It applies a pattern growth method which avoids costly candidate generation and test by successively concatenating frequent 1-itemset found in the ($ConFP$) FP. In this context, the major operations of mining are count accumulation and prefix path count adjustment which is usually much less costly than candidate generation and pattern matching operations performed in *Apriori*-like algorithms.

We have studied the performance of FP and EFP method in comparison with SQL-based *Apriori* approach and *loose-coupling* FP-*growth* approach in large databases. The experimental results show that SQL-based frequent pattern mining approach using FP-*growth* can get better performance than *Apriori* on large data sets or long patterns.

There remain lots of further investigations. We plan to implement our SQL based frequent pattern mining approach on parallel RDBMS, and to check how efficiently our approach can be parallelized and speeded up using parallel database system. Additionally, we will investigate an SQL based algorithm which combine *Apriori* and FP-*growth* to scale both small and large data sets.

Chapter 5

Propad Approach

The main issues in frequent itemsets mining are:

- Reducing the database scanning times, since in many cases the transactional database is too large and scanning data is very costly;

- Reducing the search space since every subset of I can be frequent and the number of them is exponential to the size of I;

- Counting the support for itemsets efficiently.

The bottleneck of the *Apriori*-like method is the candidate set generation and test [HPY00]. If one can avoid generating a huge set of candidates, the mining performance can be substantially improved. In addition, the search technique employed in *Apriori*-like approach is a bottom-up generation of frequent itemset combinations. The search cost is high especially for mining long patterns. If one may transform the problems of finding long pattern to looking for shorter ones can be dramatically reduce the search cost.

The *FP-growth* method achieves an efficient performance compared with *Apriori*-like approach. But due to its complex representation and expensive and exhaustive computational process, the algorithm is not scalable to sparse and very large databases.

We proposed a SQL-based *FP-tree* mining approach in Chapter 4. The experiments show that the approach can obtain better performance and scalability than SQL-based *Apriori* approach.

The key factor that influence the performance of the *FP-tree* mining approach are the construction of FP, the total number of conditional FP, $ConFP$, built during the whole mining process, and the cost of mining a single $ConFP$. Can we avoid materializing FP and $ConFP$? To attack this problem, we develop a SQL-based algorithm, called PROjection PAttern Discovery, or *Propad* for short, in this chapter. In Section 5.1, we propose the *Propad* algorithm for mining frequent patterns [SSG04a, SS05]. We discuss combination *Propad* Algorithm with *Apriori* Candidate Generation to mine different data sets in Section 5.2. Experimental results and performance studies are reported in Section 5.3.

5.1 Algorithm for *Propad*

Given a transaction database DB and a support threshold *min_sup*. Following the *Apriori* property (Theorem 2.1), only frequent items play roles in frequent patterns. Frequent item sets can be represented as a tree that is not necessarily materialized.

Example 5.1 *Let us give an example with five transactions and support threshold is set to* 3 *in Table 5.1 (same as the transaction database used in Example 2.3). Each node of the frequent item sets tree is labeled by a frequent item and associated with its support. Each frequent items set is represented by one and only one path starting from the root, and the support of the ending node is the support of the item set. The null root corresponds to the empty item set. Figure 5.1 represents the frequent item set tree for the given example. The path (1:3)-(3:3)-(6:3)-(13:3) represents the frequent pattern* $\{1, 3, 6, 13\}$ *with support of 3.*

We can observe that:

- X and Y are frequent and X is an ancestor of Y, then all patterns between X and Y are frequent.

TID	$Transaction$	$FrequentItems$
1	$1, 3, 4, 6, 7, 9, 13, 16$	$1, 3, 6, 13, 16$
2	$1, 2, 3, 6, 12, 13, 15$	$1, 2, 3, 6, 13$
3	$2, 6, 8, 10, 15$	$2, 6$
4	$2, 3, 11, 16, 19$	$2, 3, 16$
5	$1, 3, 5, 6, 12, 13, 16$	$1, 3, 6, 13, 16$

Table 5.1: A transaction database and $\xi = 3$

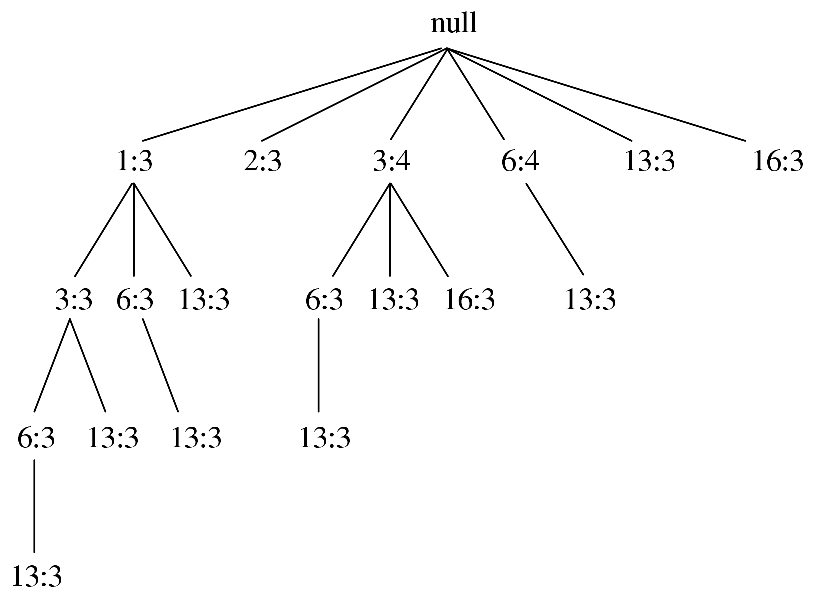


Figure 5.1: A frequent item set tree

- To find the child patterns of X, only frequent items that co-occur with X in the database are needed to be accumulated. It means that only X projected database is needed.

Let I be a set of *items*, F be the frequent items in DB. The set of all frequent itemsets with the same prefix $L \subseteq I$ is denoted as $F(L)$. For each transaction t, the set of frequent items in t is represented as $F(t)$, $F(t) = t \bigcap F$. All frequent itemsets containing item $i \in I$ can be found in the so called i-projected transactions. We present the projected transactions by the table PT with two columns attributes:

1. Transaction identifier (tid). The transactions contain the item i.

2. Item identifier ($item$). The items co-occur with the item i.

The mining process can be regarded as a process of frequent item set tree construction, which is facilitated by successively projecting the transactions into the frequent itemset. In order to avoid repetitiousness and to ensure each frequent item is projected to at most one projected table, we suppose items in alphabetical order.

Before the new algorithm is given, let us define the projected transaction table:

Definition Let $L \subseteq I$ be a frequent itemset in DB. $la(L)$ be the largest item in L. A L-related projected transaction table, is denoted as PT_L, that collects all frequent items in the transactions containing L, restricted to items J with $J > la(L)$ and the support of items J satisfies the minimum support threshold.

Take frequent item 1 in T for example. The transactions containing 1 are $\{1, 2, 5\}$. There are four items (larger than 1) $\{3, 6, 13, 16\}$ contained in the transaction 1. $\{2, 3, 6, 13\}$ (larger than 1) are contained in the transaction 2. Transaction 5 includes $\{3, 6, 13, 16\}$ that are larger than 1. Only $\{3, 6, 13\}$ are frequent in the local transaction table. The projected transaction table PT_1 is shown in Table 5.2.

After some careful analysis, we proposed a SQL-based algorithm, called *PROjection PAttern Discovery*, or *Propad* for short. Like the *FP-growth* method it adopts the divide-and-conquer strategy: successively transforms the original transaction table

TID	*Item*
1	3
1	6
1	13
2	3
2	6
2	13
5	3
5	6
5	13

Table 5.2: An example *PT* table

into a set of frequent item-related projected tables. Then we separately mine each one of the tables as soon as they are built.

Our general idea of *Propad* is illustrated in the following example.

- At the first level, by scanning DB once, the complete set of frequent items $\{1:3,\ 2:3,\ 3:4,\ 6:4,\ 13:3,\ 16:3\}$ can be found and stored in the frequent item table F, illustrated in Figure 5.2 (b). We simply gather the items that satisfy the minimum support and insert them into the transformed transaction table TF that has the same schema as transaction table T. It means that only frequent 1-items are included in the table TF as shown in Figure 5.2 (c);
- Following the order of frequent items: 1-2-3-6-13-16, the complete set of frequent patterns can be partitioned into 6 subsets without overlap as follows:
    1. Those containing item 1;
    2. Those containing item 2 but no item 1;
    3. Those containing item 3 but no 1 nor 2;
    4. Those containing item 6 but no 1, 2, nor 3;
    5. Those containing item 13 but no 1, 2, 3, nor 6;
    6. Those containing item 16 but no 1, 2, 3, 6, nor 13;

Tid	*Item*
1	1
1	3
1	4
1	6
1	7
1	9
1	13
1	16
...	...

(a) T

Item	*Count*
1	3
2	3
3	4
6	4
13	3
16	3

(b) F

Tid	*Item*
1	1
1	3
1	6
1	13
1	16
...	...

(c) TF

Figure 5.2: An example transaction table T, frequent item table F, and transferred transaction table TF

- At the second level, for each frequent item i we construct its respective projected transaction table PT_i. This can be done by two phases. The first step finds all frequent items that co-occur with i and are larger than i from TF. The second step finds the local frequent items. Only those local frequent items are collected into the PT_i. Frequent 1-items are regarded as the prefixes, frequent 2-patterns are gained by simply combining the prefixes and their local frequent itemsets.

  We start from item 1. The 1-related projected transaction table is constructed as follows: we find all frequent patterns with the respect to item 1, which is the base item of the tested projected table. All items that are locally frequent with 1, $\{3 : 3,\ 6 : 3,\ 13 : 3\}$, are inserted into the table PT_1, as shown in Figure 5.3 (a-c). Then, the frequent 2-itemsets associated with item 1 $\{\{1, 3 : 3\}, \{1, 6 : 3\}, \{1, 13 : 3\}\}$ can be found.

  Now, we construct the projected transaction table associated with item 2, noted as PT_2. Since no local frequent items is found, PT_2 is empty. The search for frequent patterns associated with item 2 terminates.

  Similarly, we can build the projected transaction table associated with other items, PT_3, PT_6, PT_{13}. Notice, according to the definition about the i-related projected transaction table, the last subset's projected transaction table is empty. Therefore, it is not necessary to be accumulated.

- At the third level, for each frequent item j in the table PT_i, we successively construct the projected transaction table $PT_{i,j}$ and gain its local frequent items. One projected transaction table is filtered if each transaction in the table only maintains items that contribute to the further construction of descendants.

  For example, to PT_1, three frequent items $\{3 : 3,\ 6 : 3,\ 13 : 3\}$ are found at the level 2. We first construct $PT_{1,3}$ and two local frequent items $\{6 : 3,\ 13 : 3\}$ are achieved as illustrated in Figure 5.3 (d-f). Then, the frequent 3-itemsets associated with item 1 and 3, $\{\{1, 3, 6 : 3\},\ \{1, 3, 13 : 3\}\}$ can be mined. Now, only one local frequent item $\{13 : 3\}$ is found associated with items 1 and 6, $PT_{1,6}$ is filtered as illustrated in Figure 5.3 (g). Then, the frequent 3-itemsets having items 1 and 6, $\{1, 6, 13 : 3\}$ is achieved.

- At the fourth level, we can build the projected transaction table, $PT_{i,j,w}$, associated with item set $\{i,\ j\}$. To $PT_{1,3}$, for example, one frequent item $\{13 : 3\}$ is found shown in Figure 5.3 (h). Then, the frequent 4-itemsets having item 1 and 3 and 6, $\{1, 3, 6, 13 : 3\}$ is achieved. The search for frequent patterns associated with item 1 terminates.

Basically, the projecting process can be facilitated either by breadth first approach or by depth first approach. In breadth first approach, we have two alternatives to represent projected transaction tables. One is: each frequent item has its corresponding projected transaction table and local frequent itemsets table at level k. That is, n projected tables need to be generated if we have n frequent itemsets at level k. It is obviously unpracticable because too many temporary tables have to be held especially for dense database and for low support threshold. The other is: one projected transaction table is used of each level. Let's revisit the mining problem in Table 5.1. Table 5.3 illustrates the projected transactions associated with each frequent item at level 2. Normally, this projected transaction table is too huge to efficiently join in subsequent mining procedures, especially at level 2.

Avoiding creating and dropping cost of many temporary tables, depth first approach is used in our approach. Let $\{i_1, i_2, \ldots, i_n\}$ be the frequent 1-itemset. We can first find the complete set of frequent patterns containing $\{i_1\}$. Conceptually, we

Tid	*Item*
1	3
1	6
1	13
1	16
2	2
2	3
2	6
2	13
5	3
5	6
5	13
5	16

(a) TEMP

Item	*Count*
3	3
6	3
13	3

(b) F

Tid	*Item*
1	3
1	6
1	13
2	3
2	6
2	13
5	3
5	6
5	13

(c) PT_1

Tid	*Item*
1	6
1	13
2	6
2	13
5	6
5	13

(d) TEMP

Item	*Count*
6	3
13	3

(e) F

Tid	*Item*
1	6
1	13
2	6
2	13
5	6
5	13

(f) PT_1_3

Tid	*Item*
1	13
2	13
5	13

(g) TEMP

Tid	*Item*
1	13
2	13
5	13

(h) TEMP

Figure 5.3: Construct frequent items by successively projecting the transaction table T.

TID	*Item*
1	3
1	6
1	13
2	3
2	6
2	13
5	3
5	6
5	13
1	6
1	13
1	16
2	6
2	13
4	16
5	6
5	13
5	16
...	...

Table 5.3: An example *PT* table in breadth first approach

construct $\{i_1\}$-projection table and then apply the techniques recursively. After that, we can find the complete set of frequent patterns containing $\{i_2\}$ but no item $\{i_1\}$. Similarly, we can find the complete set of frequent patterns.

To construct projection table associated with item i, we use temporary projection table, $TEMP$, to collect all frequent items (larger than i) in the transactions containing i. Frequent itemset table, F, is used to store local frequent items of each $TEMP$. The i-related Projected transaction table PT_i is built by joining $TEMP$ and F. In fact, the $TEMP$ and the F are only required in the constructing projection table PT. In that case, we create one $TEMP$ and one F during the whole mining procedure. The table $TEMP$ and F are cleared once a table PT is constructed.

In depth first approach, PT table of each frequent item is not constructed together. Each PT table is built, then mined before the next PT table is built. The mining process is done for each frequent item independently with the purpose of finding all frequent k-itemset patterns in which the item at the level k-1 participates. We use one PT table at the each level. For example, to mine all frequent patterns associated with item 1, four PT tables are built. Each of them corresponds to each level. Let k be the level of mining process, *pass* be the maximum length of current frequent patterns. In the subsequent mining procedure, the PT_k table is cleared if $k \leq pass$. Else, build a new PT_{k+1} table for the level k+1. In that case, the number of PT tables is the same magnitude as the length of maximum frequent pattern. The SQL queries used to create a i-projected transaction table is illustrated in Figure 5.4. PT_k is generated by self joining PT_{k-1}. A temporary table, $TEMP$, stores all the items that are larger than i and co-occur with i in PT_{k-1}. The local frequent items in the table $TEMP$ are inserted into the table F.

Now we summarize the algorithm *PROjection PAttern Discovery*, abbreviated as *Propad*, as follows. The number of iterations in the *for* loop is one less than the number of frequent 1-itemsets.

Further optimization can be explored on a special PT, in which all frequent items co-occur in the same transaction. In that case, the set of frequent patterns is generated by enumeration of all the combinations of the distinct items in PT with prefix instead of constructing the PT for each item. Let us examine an example. Table

Algorithm 7 *Propad*

Input: A transaction table T and a minimum support threshold ξ
Output: A frequent pattern table PAT
Procedure:

```
1. $pass\_num := 0$;
2. $prefix := null$;
3. get the transformed transaction table $TF$ by removing infrequent items from $T$;
4. insert the frequent 1-items into $PAT$;
5. for each distinct frequent item $i$ in $TF$
6.     $prefix := i$;
7.     call findFP$(prefix, 1)$;


findFP$(prefix, k)$
if $PT_k$ has at least one frequent item
  combine $prefix$ with frequent item sets and insert them into $PAT$;
  if $PT_k$ is not be filtered
    if $k + 1 > pass\_num$
      create table $PT_{k+1}$;
      $pass\_num = k + 1$;
    else clear table $PT_{k+1}$;
    construct $PT_{k+1}$ by projection;
    for each frequent item $j$ in $PT_{k+1}$
        $prefix := prefix + j$;
        findFP $(prefix, k + 1)$;
```

```
insert into $TEMP$ $(id, item)$ select $t_1.id, t_1.item$
from        $PT_{k-1}$ $t_1$, $PT_{k-1}$ $t_2$
where       $t_1.id = t_2.tid$ and
            $t_2.item = i$ and
            $t_1.item > i$

insert into $F$ $(item, count)$ select $item$,count(*)
from        $TEMP$
group by    $item$
having      count(*) $\geq minsupp$

insert into $PT\_k$ select $t.id, t.item$
from        $TEMP$ as t, $F$
where       $t.item = F.item$
```

Figure 5.4: PT generation in SQL

5.2 is a *PT* that consists of three different items $\{3, 6, 13\}$, which co-occur in the same transactions $\{1, 2, 5\}$. The prefix of the *PT* is item 1, then the frequent patterns are $\{\{1, 3\}, \{1, 3, 6\}, \{1, 3, 6, 13\}, \{1, 3, 13\}, \{1, 6, 13\}, \{1, 6\}, \{1, 13\}\}$. Such an optimization is especially useful at mining long frequent patterns.

5.1.1 Enhanced Query Using Materialized Query Table

Materialized Query Tables are tables that contain information that is derived and summarized from other tables. They are designed to improve performance of the database by doing some intensive work in advance of the results of that work being needed.

5.1.2 Analysis

The mining process can be facilitated by projecting transaction tables in a top-down fashion. In our method, we are trying to find all frequent patterns with the respect to one frequent item, which is the base item of the tested projected table. All items that are locally frequent with i will participate in building the i projected table.

In Comparison with *Apriori*, our approach dramatically prunes the candidate-2 itemsets. In *Apriori* approach we need to generate 15 candidate-2 itemset which are $\{\{1, 2\}, \{1, 3\}, \{1, 6\}, \{1, 13\}, \{1, 16\}, \{2, 3\}, \{2, 6\}, \{2, 13\}, \{2, 16\}, \{3, 6\}, \{3, 13\}, \{3, 16\}, \{6, 13\}, \{6, 16\}, \{13, 16\}\}$, using our approach we have only 7 patterns to test which are $\{\{1, 3\}, \{1, 6\}, \{1, 13\}, \{3, 6\}, \{3, 13\}, \{3, 16\}, \{6, 13\}\}$.

Comparing with the approach based on *FP-tree*, the *Propad* method will never need to materialize the FP and $ConFP$. Since constructing FP and $ConFP$ are expensive when the database is huge and sparse, the cost is substantially reduced.

Our method of *Propad* has the following merits:

- Avoids repeatedly scan the transaction table, only need one scan to generate transformed transaction table.
- Avoids complex joins between candidate itemsets tables and transaction tables, replacing by simple joins between smaller projected transaction tables and frequent itemsets tables.
- Avoids the cost of materializing frequent itemsets tree tables.

5.2 *Propad* with *Apriori* Candidate Generation

Real databases are skew, which cannot be simply classified as purely dense or purely sparse. We may distinguish between dense data sets and sparse data sets. A dense dataset has many frequent pattern of large size and high support. In those datasets, many transactions are similar to each other. Datasets with mainly short patterns are called sparse. Longer patterns may exist, but with relatively small support.

As we know, it is hard to select an appropriate and efficient mining method in all occasions. In this section, we discuss a hybrid approach to deal with both sparse and dense data sets.

Finding frequent patterns in dense and very large data sets is a challenging task since it may generate dense and long patterns. If an *Apriori*-like algorithm is used,

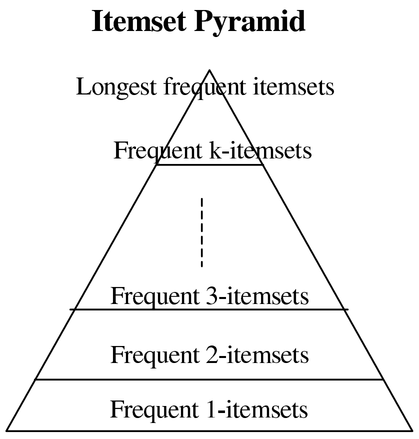


Figure 5.5: The frequent itemsets located at different levels

the generation of very large number of candidates sets is led. The *Propad* approach works well in dense data sets with a large number of long patterns since it avoids the generation of any candidate k-itemsets for any k by applying a recursive pattern growth method. In the *Propad* approach, however, the process of recursively constructing the projected transaction tables would be cost-consuming especially when the data sets is very large. However, for sparse datasets, candidate generation is a very suitable method for finding frequent patterns.

We can observe that the number of transactions that support length k itemsets decreases when k increases, and this decrease is sharp when k is greater than 2 or 3. Figure 5.5 illustrates the frequent itemsets located at different levels. Therefore, We can first employ *Propad* to mine the former k-itemsets. As mining progresses, the projected transaction becomes smaller. Suppose *Propad* stops at level 2, and then we use K-$Way\ join$ to continue mining till the end.

Now we present the hybrid approach from the *Propad* and the *Apriori* as follows.

- Find the frequent k-itemsets with the *Propad* algorithm. This is done by two phases. First, generate frequent 1-itemset. For each item, we recursively build its projected transaction table PT till the level k. The items in the table PT_k together with the prefix insert into F_k, which collects the frequent k-itemsets. The table F_k is built after search of frequent patterns associated with each item terminates at level k.

$Item1$	$Item2$	$Count$
1	3	3
1	6	3
1	13	3
3	6	3
3	13	3
3	16	3
6	13	3

Table 5.4: An example F_2 built by the *Propad* algorithm

Let's illustrate the method by revisiting Example 5.1.

Suppose the turn level is 2. We create a table F_2, which stores all frequent 2-itemset. The table F_2 has the scheme with three columns: $item1$, $item2$, and $count$. By scanning the transaction table, T, the frequent 1-itemset $\{1 : 3, 2 : 3, 3 : 4, 6 : 4, 13 : 3, 16 : 3\}$ can be found. We start from the item 1 and project the transactions into 1. Three local frequent items $\{3 : 3, 6 : 3, 13 : 3\}$ are found. That is, the frequent 2-itemset $\{\{1, 3 : 3\}, \{1, 6 : 3\}, \{1, 13 : 3\}\}$ associated with 1 can be found. Then, we insert 1 as $item1$ and $3, 6, 13$ as $item2$ into the F_2. Similarly, we mine the item 2 till the item 13. After that, the F_2 is constructed as illustrated in Table 5.4.

- Find the remaining frequent itemsets with the K-$Way\ join$ algorithm. In the K_{th} pass, we need to generate a set of candidate itemsets C_k from frequent itemsets F_{k-1} of the previous pass. And then generate a set of frequent itemsets F_k using K-$Way\ join$.

Based on the above illustration, we have two algorithms of hybrid as follows.

The two hybrid algorithms are different in line 4-7. The former one works well when the dataset is sparse since the number of transactions that support length k itemsets sharply decreases when k increases. For very dense datasets, however, this decrease is not obvious as k increases. Moreover, it may generate very long patterns. In this situation, the later one works well.

Algorithm 8 *Hybrid*(1)

Input: A transaction table T, a minimum support threshold ξ and stop level k
Output: A frequent pattern table PAT
Procedure:

```
1. generate frequent 1-itemset table;
2. create frequent k-itemset table $F_k$;
3. for each frequent item $i$
4.     construct the projection table $PT$ with $Propad$ till the level $k$;
5. build frequent k-itemset table $F_k$;
6. generate table $C_{k+1}$;
7. build remaining frequent patterns with K-Way join;
```

Algorithm 9 *Hybrid*(2)

Input: A transaction table T, a minimum support threshold ξ and stop level k
Output: A frequent pattern table PAT
Procedure:

```
1. generate frequent 1-itemset table;
2. create frequent k-itemset table $F_k$;
3. for each frequent item $i$
4.     construct the projection table $PT$ with $Propad$ till the level $k$;
5.     build frequent k-itemset table $F_k$;
6.     generate table $C_{k+1}$;
7.     build remaining frequent patterns with K-Way join;
```

We can see that the hybrid approach is sensitive to level k. Basically, the significant portions of the total cost in the whole procedure are the cost of the second pass and the third pass. This can be seen through Figure 5.8. Therefore, we set the stop level is level 2 or level 3 according to the character of the data sets. For example, the stop level can be set to 3 when mining long frequent patterns.

5.3 Experimental Evaluation and Performance Study

To evaluate the efficiency and scalability of *Propad* and *Hybrid*, we have performed an extensive performance study. In this section, we report our experimental results on the performance of *Propad* and *Hybrid* in comparison with *K-Way join* based on *Apriori*-like and *EFP* based on *FP-tree* proposed in Chapter 4. It shows that algorithm based on *Propad* outperforms *K-Way join* and *EFP*. The *hybrid* approach from *Propad* and *K-Way join* can get efficient performance on sparse datasets or very dense datasets when the value of level k is well selected.

All the experiment were performed on Version 8 of IBM DB2 EEE installed on Linux operation system with Pentium IV 2.0Ghz.

The performance measure was the execution time of the algorithm on the datasets with different support threshold.

5.3.1 Data Set

We report experimental results on four synthetic data sets and one real data set. They are respectively T5I5D10K, T25I10D10K, T25I20D100K, T10I4D100K, and *Connect*4.

The first four datasets have described in Chapter 4. T25I20D100K is relatively dense and contains abundant mixtures of short and long itemsets. T10I4D100K is very sparse. The frequent itemsets are short and not numerous. Here we have chosen the dataset T10I4D100K, because for this dataset, the experiment runs for 10 passes and we want to see how these approaches perform when mining long pattern.

To test the capability of our approach on dense datasets with long patterns, we use

Datasets	*Numbers of transactions*	*Numbers of items*	*Avg. transaction length*
$T5I5D10K$	$10,000$	$1,000$	5
$T25I10D10K$	$10,000$	$1,000$	25
$T25I20D100K$	$100,000$	$1,000$	25
$T10I4D100K$	$100,000$	$1,000$	10
$Connect4$	$67,557$	130	43

Table 5.5: Description of the generated datasets

the real data set $Connect4$ from the UC-Irvine Machine Learning Database Repository. It is compiled from the $Connect4$ game state information. The total number of transaction is 67557, which each transactions is with 43 items. It is a dense dataset with a lot of long frequent itemsets. Table 5.5 summarizes the parameters associated with the datasets.

5.3.2 Comparison of Different Approaches

For these experiment, we built (tid, item), (item, tid) index on the data table T and (item) index on the frequent itemsets table F. The goal was to let the optimizer choose the best plan possible. We didn't build indexes on table *PT*s and other temporary tables built during the whole mining procedure, since these tables are modified frequently during the whole procedure. Each modification to a table must be reflected in all the indexes that are defined on the table, that carries a certain cost. The run time used here means the total execution time, that is the period between input and output.

In this subsection, we describe *Propad* algorithm performance compared with *K-Way join* and *EFP* algorithm proposed in Chapter 4.

Figure 5.6 shows the run time of *Propad*, *Propad_O* (Optimized *Propod*), *K-Way join* and *EFP* on T10I4D100K. Clearly, *Propad* and *Propad_O* win the other two approaches, and the gaps become larger as the support threshold goes down. *Propad_O* optimizes *Propad* through enumerating all combination of frequent itemsets on a special PT, in which all frequent items associated with item i co-occur in the same transaction. So that the search for frequent patterns associated with item

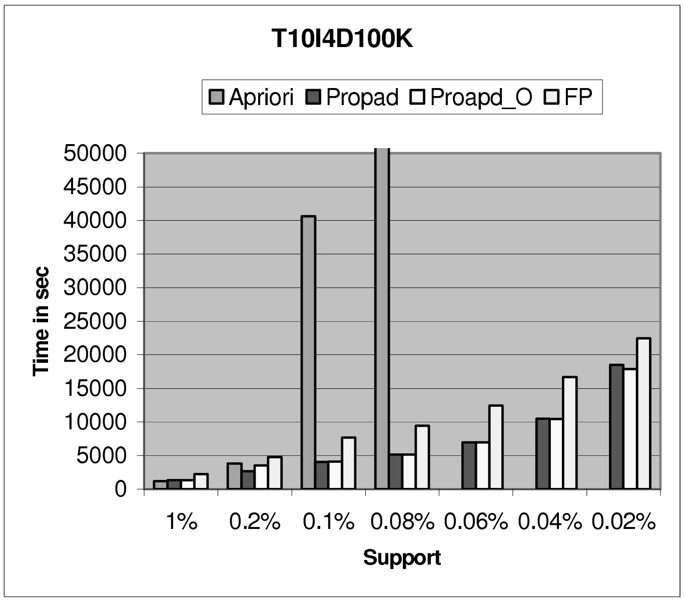


Figure 5.6: Comparison over dataset T10I4D100K

i terminates earlier. However, the *Propad_O* approach does not gain the significant performance improvement as we expected since it still takes time to test whether one *PT* is kind of this special *PT* or not.

K-Way join works well in such sparse data set since most of the candidates that *Apriori* generates turn out to be frequent patterns. However candidates generation and frequent count are still the major cost of *Apriori*.

EFP has a similar performance as *K-Way join* and sometime is even slightly worse. This is because when the data set is sparse, table *FP* can not compass data as effectively as what is does on dense data sets. Constructing table *FP*s over sparse data sets recursively has its overhead.

To test the performance of these approaches over dense data sets, we use the synthetic data set generator described in [AS94] to generate a data set T25I20D100K. It is a relatively dense data set and contains an abundant mixtures of short and long frequent patterns.

Figure 5.7 shows the run time of the three approaches on this data set. When the support threshold is high, most patterns are of short lengths, *K-Way join* and *EFP* have similar performance. When the support threshold becomes low, the number

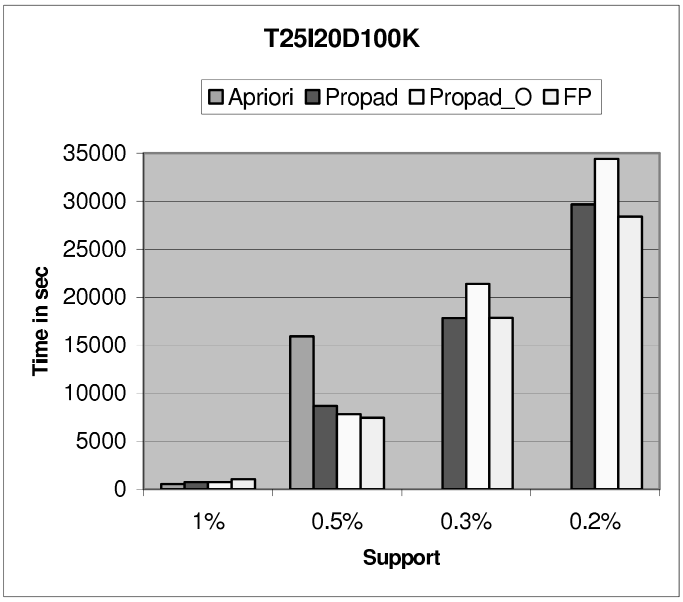


Figure 5.7: Comparison over dataset T25I20D100K

of frequent patterns goes up and most items are frequent. Then, the advantages of *Propad* over other two approaches becomes obvious.

Propad approach can get competitive performance out of SQL-based *FP* implementation. An important reason for superior performance of *Propad* over SQL-based *FP* is that it avoids materializing table *FP* and *CFP* (conditional FP), that is the main time-consuming part of the algorithm.

Figure 5.8 shows the run time of *K-Way join* approach to generate different level frequent patterns on data set T10I4D100K. Form the figure, we can see that the time required for support counting at higher passes is not very significant where the length of the largest frequent itemset is small. This is because there is a great reduction in the size of the candidate itemset C_k. However, for datasets with long patterns, joining k-copies of input table for support counting at higher passes is quite significant though the cardinality of the C_k decreases with the increase in the number of passes. However, of all the passes, second pass (for short patterns) or third pass (for long patterns) is the most time consuming. In general, because of the immense size of C_2 or C_3, the cost of support counting for C_2 or C_3 is very high. In addition, for candidate sets of length 2, as all the subsets of length 1 are known to be frequent,

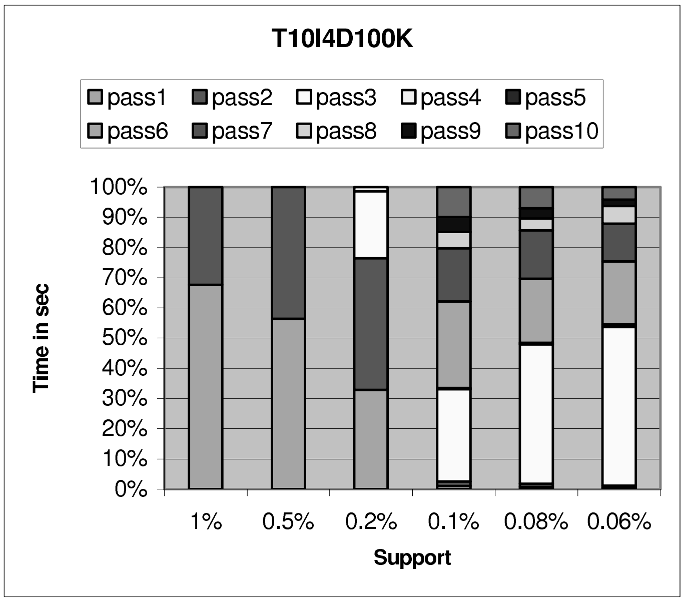


Figure 5.8: *K-Way join* over dataset T10I4D100K

there is no gain from pruning during the candidate generation.

In sparse data set T10I4D100K with long patterns, constructing projection table of each frequent item recursively is high overhead. We compared *Propad* with $Hybrid(1)$ with cut level being 3 on this data set. Figure 5.9 shows the run time taken by these two approaches.

The result on mining the real dataset Connect-4, which contains 67,557 transactions with 43 items in each transaction, is shown in Figure 5.10. The figure supports the idea that $Hybrid(2)$ runs faster than *Propad* on very long itemsets. From the figure, however, one can see that EFP is scalable even when there are many long patterns. In such dataset, neither *Propad* nor $Hybrid(2)$ are comparable to the performance of EFP. To deal with such a very dense data set, the main costs in both *Propad* and $Hybrid(2)$ are recursively projection. In contrast, the compactness of a table FP is very high since many transactions share the prefix paths of it. The recursive construction of conditional FP tables is limited by the maximal length of the transactions.

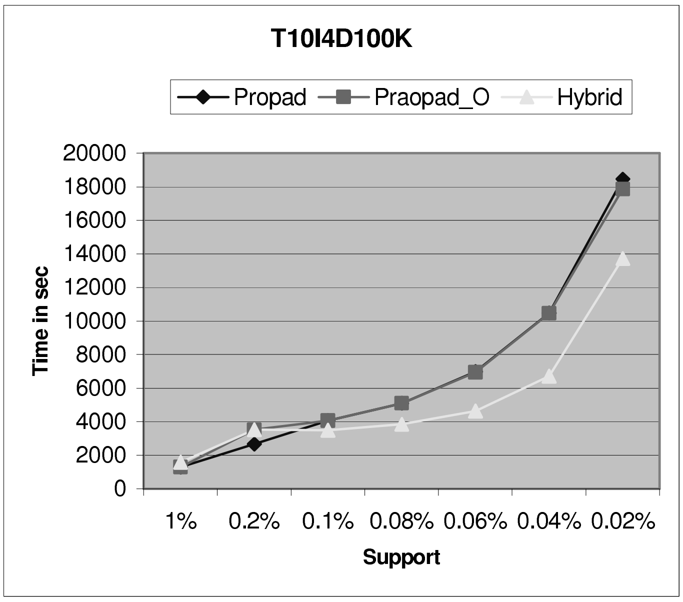


Figure 5.9: Comparison between *Propad* and *Hybrid*(1) over dataset T10I4D100K

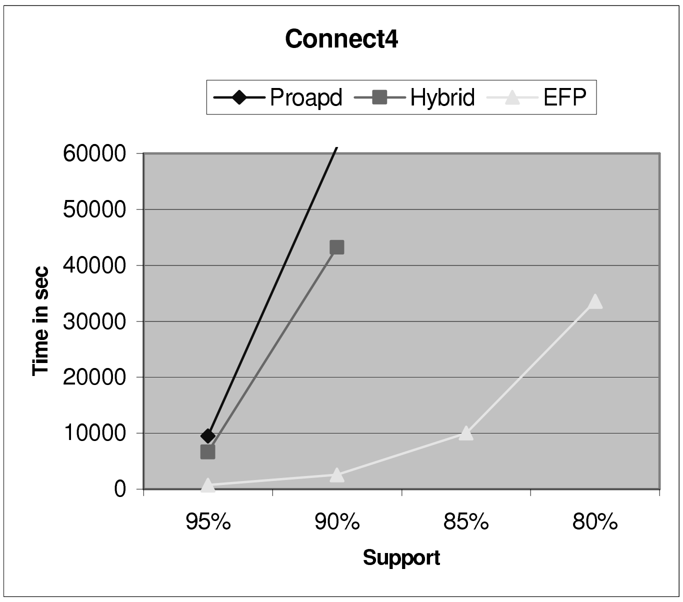


Figure 5.10: Comparison over dataset Connect4

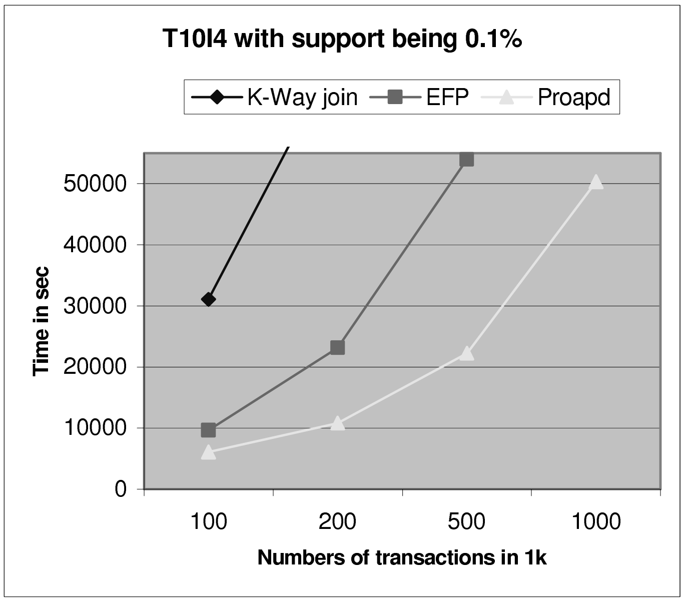


Figure 5.11: Scalability with the number of transactions in T10I4

5.3.3 Scale-up Study

We experimented with several synthetic datasets to study the scale-up behavior of *Propad* against the number of transactions and the transaction size. A set of synthetic datasets are generated using the same parameters of T10I4 and T25I20. Figure 5.12 and Figure 5.13 show the scalability of *Propad* and *EFP* and *K-Way join* with respect to the number of transactions increased from 100k to 1M.

For data sets with large transactions, *Apriori* has to generate tremendous number of candidates, that is a very costly process. To deal with large datasets, the construction of an *FP* table may consume large main memory. The performance of *EFP* decreases with the number of transactions goes up. From the figures, one can see taht the *Propad* is more efficient and scalable at mining very large databases. It can be seen that the execution times scale quite linearly and both the datasets exhibit similar scale up behavior.

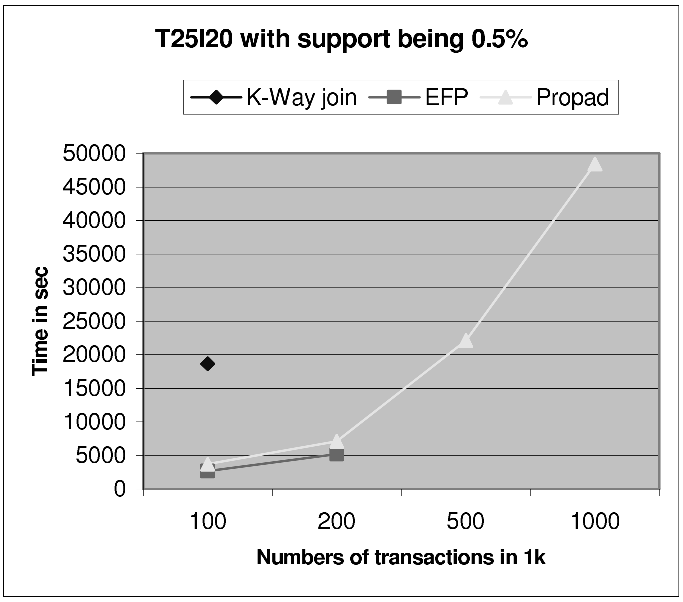


Figure 5.12: Scalability with the number of transactions in T25I20

5.4 Conclusion

In this chapter, we propose an efficient SQL based algorithm to mine frequent itemsets from databases. Rather than *Apriori*-like method it adopts the divide-and-conquer strategy and projects the transaction table into a set of frequent item-related projected tables. Experimental study shows that our *Propad* algorithm can get higher performance than *K-Way join* based on *Apriori*-like on all data sets.

A major distinct of *Propad* from the previous proposed method *EFP* is that *Propad* avoid materializing a good number of table *FP*s and *ConFP*s. It is essentially a frequent pattern growth approach since it partitions its search space according to both patterns and data based on a divide and conquer methodology. In that case, *Propad* has better performance than *EFP*.

We next implement a hybrid approach from *Propad* and *K-Way join*, that can achieve the efficiency and scalability when the value of level k is well selected.

There remain lots of further investigations. We plan to implement our SQL based frequent pattern mining approach on parallel RDBMS, and to check how efficiently our approach can be parallelized and speeded up using parallel database system. We try to find the best cutting level for combination of *Propad* and *Apriori* to make our approach more efficient on both sparse and dense datasets at all levels of support threshold.

Chapter 6

Parallelization

One typical problem is that real-world databases tend to be very large. The need to handle large amounts of data implies a lot of computational power, memory and disk I/O, which can only by provided by parallel computers. In this chapter, we present an overview of parallelization for frequent pattern mining and our implementation using parallel database systems.

6.1 Parallel Algorithms

Researchers expect parallelism to relieve current frequent pattern mining methods from the sequential bottleneck, providing scalability to massive data sets and improving response time. Achieving good performance on today's multiprocessor system is not trivial. The main challenges include synchronization and communication minimization, workload balancing, finding good data layout and data decomposition, and disk I/O minimization. In this section, we will give a brief introduction of the most influential parallel algorithms based on both *Apriori* and *FP-growth* that were proposed during the last decade. All of those are based on their sequential counterparts and exploit parallelism through parallel processing.

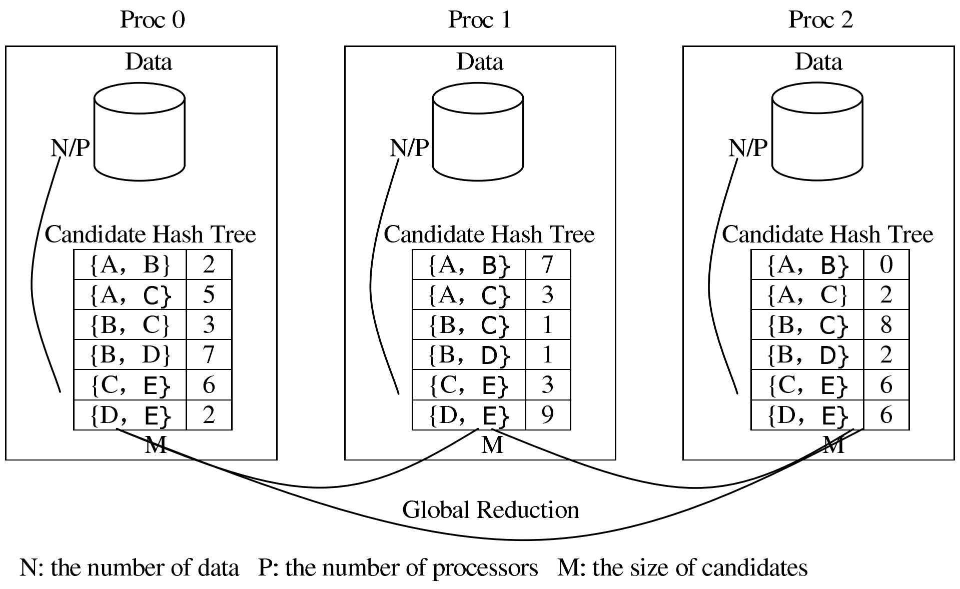


Figure 6.1: *Count Distribution* algorithm

6.1.1 Parallel *Apriori*-like Algorithms

Agrawal et al. [AIS93] have proposed three parallel algorithms based on *Apriori*: *Count Distribution* (**CD**), *Date Distribution* (**DD**), and *Candidate Distribution*. They are implemented on a 32-node IBM SP2 DMM. The CD algorithm is a simple parallelization of *Apriori* and achieves parallelism by partition data as shown in Figure 6.1. All processors generate the entire candidate hash tree from F_{K-1}. Each processor can thus independently count partial supports of the candidates from its local database partition. Next, the algorithm does a sum reduction to obtain global counts by exchanging local counts with all other processors. Once the global F_k has been determined, each processor builds the entire candidate C_{k+1} in parallel, and repeat the process until all frequent itemsets are found. This algorithm minimizes the communication since only the counts are exchanged among the processors. It, however, doesn't use the aggregate system memory effectively because the algorithm replicates the entire hash tree on each processor.

The DD algorithm is designed to minimize computational redundancy and maximize use of the total system memory by generating disjoin candidate sets on each

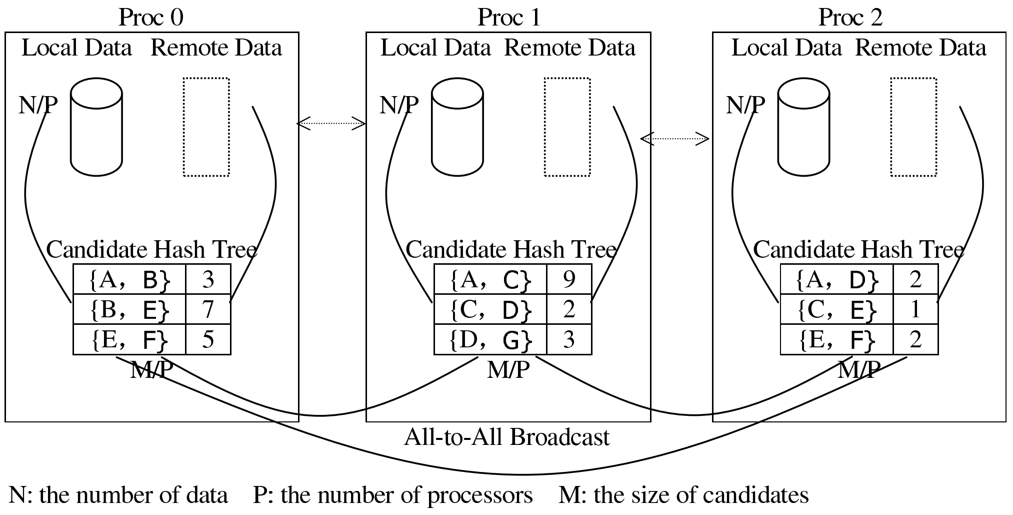


Figure 6.2: *Date Distribution* algorithm

processor as shown in Figure 6.2. However, each node must scan the entire database to examine its candidates. Thus this algorithm suffers from high communication overhead and performs poorly when compared to CD.

The *Candidate Distribute* algorithm partitions the candidates across nodes. The partitioning use a heuristic based on support so that each processor gets an equal amount of work. It attempts to minimize communication by selectively replicating the database so that a processor can generate global counts independently. It is done after a fixed number of passes of the standard data distribution algorithm. The choice of the redistribution pass involves a tradeoff between duplication and poor load balancing. *Candidate Distribute* performs worse than CD because it pays the cost of redistributing the database while scanning the local database partition repeatedly.

The **PDM** algorithm proposed by Park et al. [PCY95a, PCY95b] is based on **DHP**. In PDM, candidates are generated in parallel. Each processor generates its own local set, which is exchange through an all-to-all broadcast to construct the global candidate sets. Next, PDM obtains the local counts of k-itemsets and approximate counts of $k+1$-itemsets with a hash table for all candidates and exchange them among the processors to determine the globally frequent itemsets. Because the 2-itemset hash table can be very large, directly exchanging the counts through all-to-all broadcast can be expensive. The PDM uses an optimized method that exchanges only the cells that are guaranteed to be frequent. However, this method requires two rounds of communication. PDM implements the parallelization of hash table generation, but it is done at the cost of an all-to-all broadcast to construct the entire candidate sets. The communication costs might render this parallelization ineffective.

Shintani et al. proposed three *Apriori*-based parallel algorithms [SK96]: **Non-Partition**, **Simple-Partition**, and **Hash-Partition**. Their target machine was a 64-node Fujutsu AP1000DDV DMM. Non-Partition is essentially the same as CD, except that the sum reduction occurs on the one master processor. Simple-partition is the same as DD. Hash-partition is similar to Candidate Distribution. Unlike Candidate Distribution, it does not selectively replicate the database for counting. Each processor generates a k-subset for every local transaction, calculates the destination processor and communicate that subset to the processor. The home processor is responsible for incrementing the counts using the local database and any message sent by other processors.

Shintani et al. also proposed a variant of **Hash-Partition** called **HPA-ELD**. The motivation is that even though we might partition candidates equally across processors, some candidates are more frequent than others. In that case, their home processor will consequently be loaded. HPA-ELD addresses this by replicating the extremely frequent itemsets on all processors and processing them using the NPA scheme. Thus, no subsets are communicated for these candidates. Local counts are obtained by a sum reduction for their global support. The experiments confirmed that HPA-ELD outperforms the other approaches.

Intelligent Data Distribution (**IDD**) and *Hybrid Distribution* (**HD**) proposed by Han et al. were based on Data Distribution [HKK97]. IDD uses a liner-time, ring-based, all-to-all broadcast of communication. Compare with DD, the IDD avoid the competition problem of communication. It switches to Count Distribution once the candidates fit in memory. Instead of a round-robin candidate partitioning, it performs a single-item, prefix-based partitioning.

The HD combines *Count Distribution* and *Intelligent Distribution*. It partitions the P processors into G equal-size groups, where each group is considered as a superprocessor. CD is used among the G superprocessors, while the P/G processors in a group use IDD. HD reduces database communication costs by $1/G$ and tries to keep processors busy, especially during later iterations. Han et al. showed that HD has the same performance as CD, while it can handle much large databases.

Common Candidate Partition Database(**CCPD**) proposed by Zaki et al. in [ZOPL96] and *Asynchronous Parallel Mining* (**APM**) proposed by Cheung et al. in [CHX98] were implemented on share memory systems. CCPD uses a data parallel approach. The database is logically partitioned into equal-size chunks, and all the processors synchronously process a global or common-candidate hash tree. To build a hash tree in parallel, CCPD associates a lock with each leaf node. When a processor wants to insert a candidate into the tree, it starts at the root, and successively hashes on the items until it reaches a leaf. It then requires the lock and inserts into the candidate. With this locking mechanism, each processor can insert itemsets in different parts of the hash tree in parallel. However, hash tree has the inferior data layout. It might lead to false counting when sharing the common hash tree. Some optimization were proposed to address these problems. Such as, a hash-tree balancing and memory placement optimization.

APM is based on DIC. It uses FDM's global-pruning technique to decrease the size of candidate 2-itemsets. This pruning is most effective when there is huge skew among the partitions. At the first iteration, APM logically divides the database into many small, equal-sized virtual partitions. Then it gathers the local counts of the items in each partition and clusters them to generate a small set of candidate 2-itemsets. APM now prepares to apply DIC in parallel. The database is divided into homogeneous partitions. Each processor independently applies DIC to its local partition. There is a shared prefix tree among all processors, which is built asynchronously. These algorithms based on share memory systems has several limitations such as high I/O overload, disk competition and inferior data placement.

A hierarchical system has both distributed and share-memory components. Zaki et al. proposed four algorithms-**ParEclat**, **ParMaxEclat**, **ParClique**, and **ParMaxClique** that implemented on hierarchical systems [ZPOL97b]. All four algorithms have a similar parallelization and differ only in search strategy and equivalence class-decomposition technique. ParEclat and ParMaxEclat use prefix-based classes, bottom-up and hybrid search respectively. ParClique and ParMacClique use smaller clique-based classes, with bottom-up and hybrid search respectively. Each of these algorithms consists of three main phases: initialization phase, which performs computation and data partitioning; the asynchronous phase, where each processor independently generates frequent itemsets; the reduction phase, which aggregates final results.

6.1.2 Parallel *FP-growth* Algorithms

Zaiane et al. proposed Multiple Local Frequent Pattern Tree (**MLFPT**) [ZEHL01] based on *FP-growth* algorithm. It implemented on a shared memory and shared hard drive architectures. The MLFPT consists of two main stages. Stage one is the construction of the parallel frequent pattern trees and stage two is the mining of these data structures. In order to enumerate the frequent items efficiently, the datasets are divided among the available processors. Each processor is given an approximately equal number of transactions and enumerates the local occurrences of the items appearing in the transaction at hand. Next, a global count is done in parallel, where each processor is allocated an equal number of items to sum their local supports into global count. Then, each processor builds its own frequent pattern tree in parallel as shown in Figure 6.3. The mining process starts with a bottom up traversal of the nodes on the MLFPT structure, where each processor mines fairly equal amounts of nodes. The MLFPT algorithm overcomes the major drawbacks of parallel algorithms derived from *Apriori*, in particular the need for k I/O passes over the data. The experiments showed that with I/O adjusted, the MLFPT could achieve an encouraging many-fold speedup improvement.

Pramudiono et al. reported parallel execution of *FP-growth* on shared nothing environment [PK03a, PK03b]. The parallelization of *FP-tree* is difficult, especially

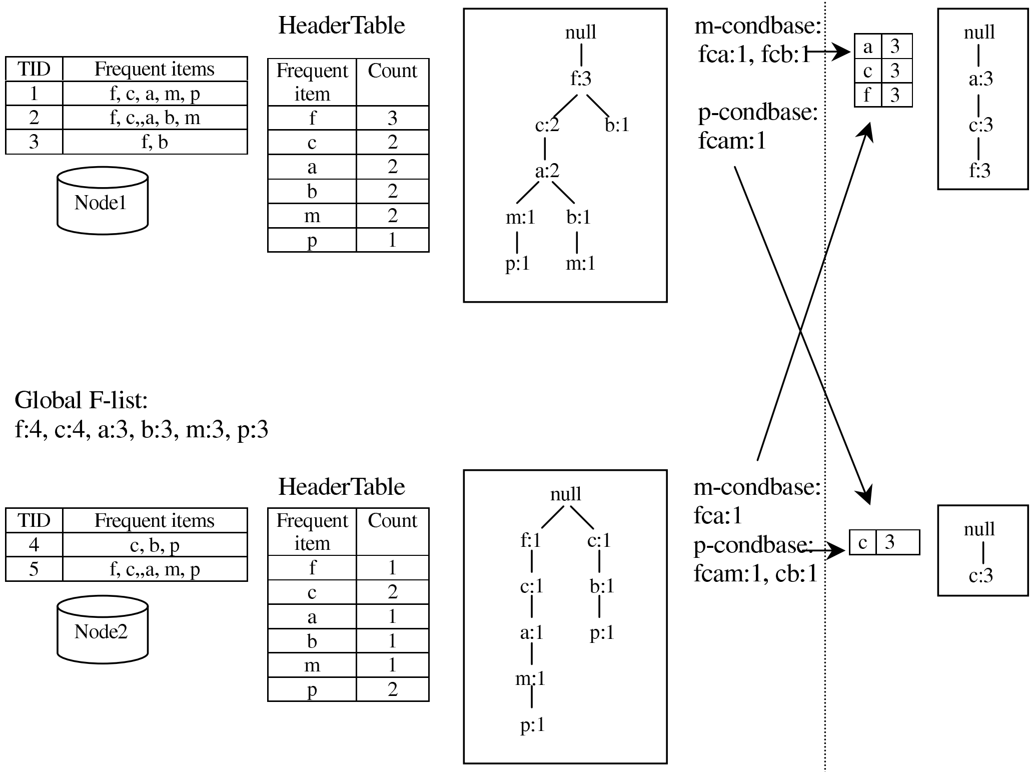


Figure 6.3: Parallel *FP-growth* algorithm on shared nothing systems

for shared nothing machines. In [PK03b], the database is distributed evenly among nodes. After the first scan of transaction database, a process exchanges the support count of all items to determine globally frequent items. Then each code builds F-list since it also has global count. At the second database scan, each node builds its local *FP-tree* from local transaction database with respect to the global F-list. Instead of processing conditional pattern base locally, each node accumulates a complete conditional pattern base and process it independently until the completion before receiving other conditional pattern base. A novel notion of *path depth* is introduced to break down the granularity of parallel processing of conditional pattern base. To reduce the cost of conditional base exchanging, serval optimizations were made. For example, ordering of conditional base processing, destination node on each round, appropriate buffer, and background processor. The experiments showed that the algorithm can achieve reasonably good speedup ratio.

Parallel system	*data parallelism Apriori*-based	*data FP-growth*	*task parallelism Apriori*-based	*task FP-growth*
shared-nothing	*PDM* *CD* *NPA* *FDM* *FPM*	*PFP*	*DD* *CandDist* *SPA* *HPA* *HPA-ELD* *IDD* *HD*	
shared-memory	*CCPD*	*MLFPT*	*PCCD* *APM*	
hierarchical			*ParEclat* *ParMaxEclat* *ParClique* *ParMaxClique*	

Table 6.1: Parallel frequent pattern mining algorithms

Table 6.1 illustrates parallel frequent pattern mining algorithms.

6.2 Frequent Pattern Mining on Parallel Database Systems

There are a verity of frequent pattern mining algorithms constructed to run in parallel, taking advantage of parallel architecture using specific programming routines.

Particularly, a parallel database system can be used to provide performance improvement, although their engines are not specifically optimized to handle data mining applications. The advantages of the implementation based on parallel database systems are as follows.

- Simpler implementation.

  The declarative, set-oriented nature of relational database query languages paves the way for the automatic exploitation of data parallelism. There is no need to use parallel routines such ad MPI libraries. The parallel DBMS is responsible itself for parallelizing queries that are issued against it. The SQL implementation can be easily parallelized when written with due care.

- Simplified data management.

  A DBMS itself has its own tools to manage data.

- Higher scalability.

  It's possible to mine datasets that are considerably larger than main memory. If data does not fit in memory, the database itself is responsible for handling information, paging and swapping when necessary.

- Opportunity for database fragmentation.

  Database fragmentation provides many advantages related to reduce number of page accesses necessary for reading data. Irrelevant data could be just ignored according on the filter specified by SQL statements. Moreover, some DBMSs can process each partition in parallel, using different execution plans when applicable.

- Portability across a wide range of parallel computer architectures.

  Virtually all commercial-available parallel database systems offer an SQL or SQL-like interface. This extends the benefit of portability achieved by using RDBMS to a wide range of parallel computer architectures, including both share-memory and share-nothing systems.

- Compatibility with parallel data warehouse systems.

  Note that current data warehouses are typically stored on a RDBMS, often running on parallel database systems for efficiency and scalability reasons. Compatibility with existing business applications is very important in practice.

However, there are few reports available about how the parallelization affects the performance of complex queries required by frequent itemset mining. In this section, we examine how efficiently frequent itemset mining with SQL can be paralleled and speeded up using parallel database systems.

6.2.1 Parallel Relational Database Systems

This subsection presents an overview of parallel relational database systems.

There are two major types of parallelism, namely temporal and spatial parallelism [HB84]. Temporal parallelism or pipelining refers to the execution of a task as a cascade of sub-tasks. Spatial parallelism refers to the simultaneous execution of tasks by several processing units.

A parallel database is a database in which multiple actions can take place at the same time in order to get your job done faster or to accomplish more work per unit time. By putting multiple physical processors to work on the same SQL statement, it is possible to scan large amount of data very quickly and to dramatically reduce the time needed to process the statement.

There are three main approaches to making multiple computers work on the same problem at the same time. Parallel processing hardware implementations are often categorized according to the particular resources which are shared. The approaches are described as follows.

- Shared nothing parallel (massively parallel, MPP) architecture

  A shared nothing architecture means each computer has its own CPU, memory, and disk. The computers are connected together with a high speed interconnect as shown in Figure 6.4. Each node can independently process its own data, and then passes the nodes' partial result back to the coordinator node. The coordinator combines all the results from all the nodes into the final result set. The nodes do not have to be independent computers. Multiple partitions can live on a single computer. This architecture takes advantage of high-performance, low-cost commodity processors and memory, has a very good flexibility and can easily be scaled to handle very large databases by adding more processors. MPPs are good for read-only databases and decision support applications. Moreover, failure is local: if one node fails, the others stay up. The major drawback of this architecture is that load balancing is difficult. Note that if a data subset is very frequently accessed, its corresponding node(s) will be a bottleneck. Hence, data placement is a crucial issue here. In addition, more overhead is required

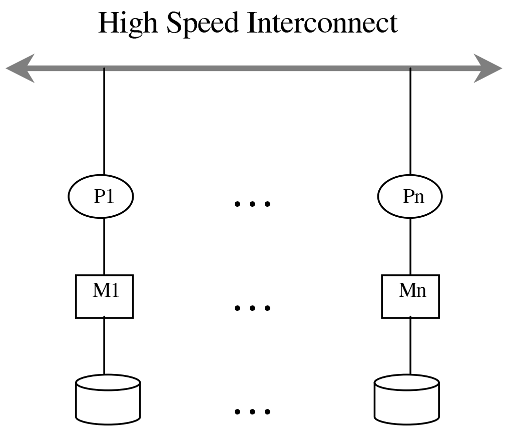


Figure 6.4: Shared-nothing architecture

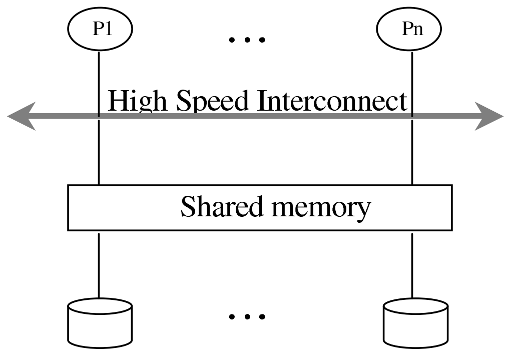


Figure 6.5: Shared-memory architecture

for a process working on a disk belonging to another node.

- Shared memory (shared everything) parallel architecture

  A shared memory architecture is also known as a symmetric multiprocessor (SMP) architecture, in which multiple processors share common memory and disks as shown in Figure 6.5. Load balancing can be automatically done and parallelization of database operation is easy, since shared memory is an inter-processor communication model more flexible than distributed memory. A disadvantage of shared memory systems for parallel processing is that scalability is limited by bus bandwidth and latency, and by available memory.

- Shared disk (distributed lock) parallel architecture

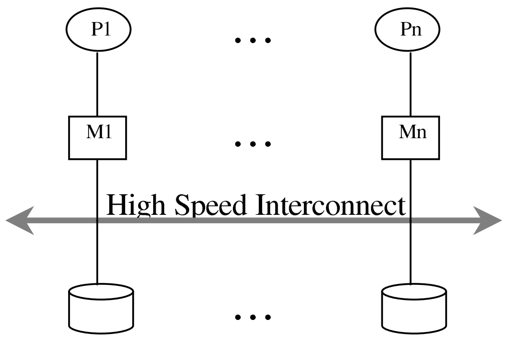


Figure 6.6: Shared-disk architecture

In a shared disk architecture, all the disks containing data are accessible by all nodes. Each node consists of one or more CPUs and associated memory. Memory is not shared between nodes as shown in Figure 6.6. Disk sharing architecture requires suitable lock management techniques to control the update concurrence control. This architecture permit high availability. All data is accessible even if one node dies. It has better scalability, in comparison with shared memory architecture. These systems have the concept of ”one database” and multiple access points. In other words, we can say it is ”multiple instances and single database”. There is no issue such as data skew as the data is located and accessed at a common location, which is an advantage over shared nothing systems. The disadvantage of shared disk systems is that inter-node synchronization is required. If the workload is not partitioned well, there may be high synchronization overhead. It is fundamentally flawed because as the number of nodes increases, the cluster drowns in lock requires.

In practice, hybrid architectures may combine advantages of different architectures [NZT96, BCV93]. An interesting possibility is to have a shared nothing architecture in which each node is a shared memory multiprocessor system as shown in Figure 6.7. This architecture has the benefit of scalability associated with shared nothing and the benefit of easy parallelization associated with shared memory.

There are two types of query parallelism that parallel databases can support to the processing of an SQL statement.

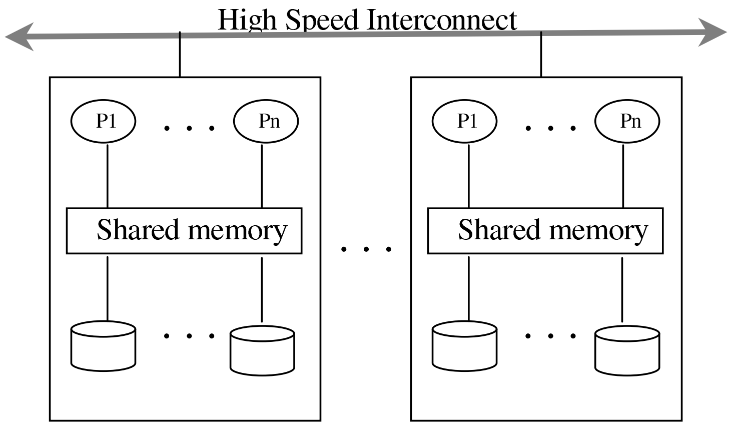


Figure 6.7: Hybrid architecture

- Inter-Query.

  This type of parallelism refers to the ability of multiple applications to query a database at the same time. Each of those queries executes independently of others, but database can execute all of them at the same time.

- Intra-Query.

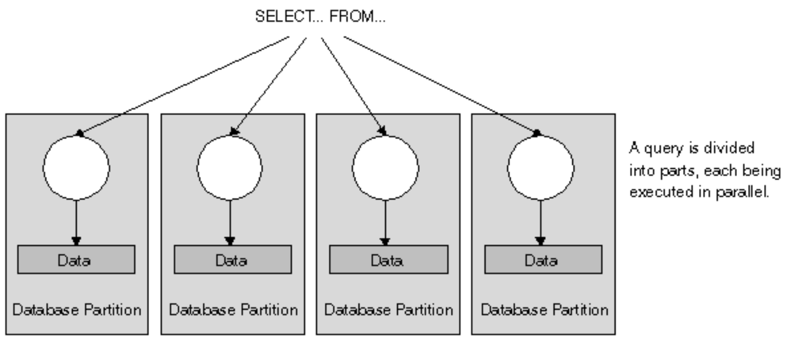
  This type of parallelism refers to the processing of parts of a single query at the same time using either intra-partition parallelism or inter-partition parallelism or both.

  – Inter-Partition Parallelism.

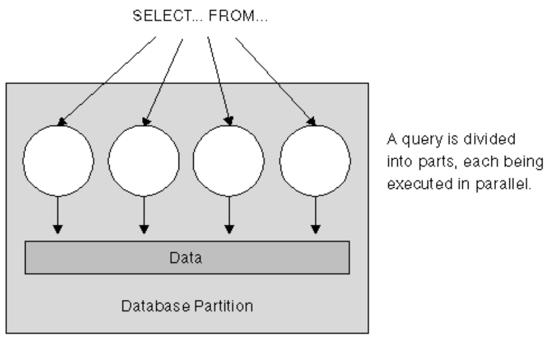
    Inter-partition refers to the ability to break up a SQL statement into a number of subset queries across multiple partitions of a partitioned database, on one machine or multiple machines. Each subset query is run in parallel. Figure 6.8 (a) shows a query that is broken into four pieces that can be run in parallel, with the results return more quickly than if a query were run in serial fashion on a single partition.

  – Intra-Partition Parallelism.

    Intra-partition refers to the ability to break up a single SQL statement into separate tasks, like scan, join or sort. These separate tasks are performed


SELECT... FROM...
Data
Database Partition
A query is divided into parts, each being executed in parallel.

(a) *Inter-Partition* parallelism


SELECT... FROM...
Data
Database Partition
A query is divided into parts, each being executed in parallel.

(b) *Intra-Partition* parallelism

Figure 6.8: Inter-Partition and Intra-Partition parallelism

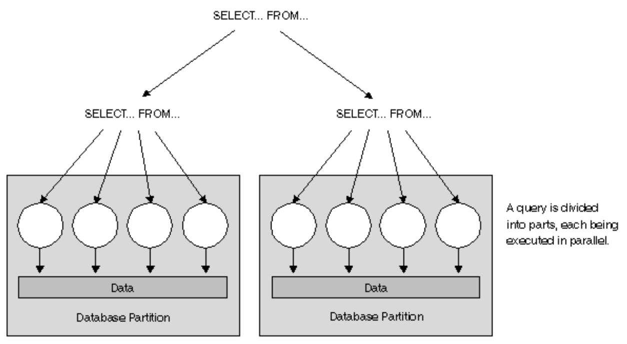


Figure 6.9: Simultaneous Inter-Partition and Intra-Partition Parallelism

in parallel. Figure 6.8 (b) shows a query that is broken into four pieces that can be run in parallel. The pieces are copies of each other. To utilize intra-partition parallelism, you must configure the database appropriately. You can choose the degree of parallelism or let the system do it for you. The degree of parallelism represents the number of pieces of a query running in parallel.

Inter-partition parallelism and intra-partition parallelism can be used at the same time. This combination provides two dimension of parallelism, resulting in an even more dramatic increase in the speed at which queries are processed as shown in Figure 6.9.

6.2.2 SQL Queries in *Apriori* and *Propad*

When data resides in a RDBMS, using SQL to work with the data increase the reliability and portability of an applications. In the case of RDBMS supporting parallelizable queries, the SQL implementation can be easily parallelized. However, there are few reports available about how the parallelization affects the performance of complex queries required by frequent itemset mining. In this section, we examined how efficiently frequent itemset mining with SQL can be paralleled and speeded up using parallel database systems.

```
insert into $C_k$ select $I_1.item_1, \ldots, I_1.item_{k-1}, I_2.item_{k-1}$
from        $F_{k-1}$ $I_1$, $F_{k-1}$ $I_2$
where       $I_1.item_1 = I_2.item_1$ and
                        $\vdots$
            $I_1.item_{k-2} = I_2.item_{k-2}$ and
            $I_1.item_{k-1} < I_2.item_{k-1}$


insert into $F_k$ select $item_1, \ldots, item_k$, count(*)
from        $C_k$, T $t_1, \ldots$ T $t_k$
where       $t_1$.item = $C_k.item_1$ and
                        $\vdots$
            $t_k$.item = $C_k.item_k$ and
            $t_1$.tid = $t_2$.tid and
                        $\vdots$
            $t_{k-1}$.tid = $t_k$.tid
group by    $item_1, item_2 \ldots item_k$
having      count(*) $\geq minsupp$
```

Figure 6.10: *K-Way join*

6.2.2.1 SQL Query Using *Apriori* Algorithm

K-Way Joins approach is reported the best algorithm overall compared to the other approaches based on SQL-92 in [STA98]. The SQL queries used to generate candidates and count the support of candidates are illustrated in Figure 6.10.

These statements do not suit well in a partitioned database. Since as the processing of mining goes on, the statements have many large tables and a wide variety of tables and columns involves in joins. In such situations it can be difficult to choose the tables' partitioning key such that all significant queries can be executed without a heavy inter-partitioned communication.

```
insert into $TEMP$ $(id, item)$ select $t_1.id, t_1.item$
from        $PT_{k-1}$ $t_1$, $PT_{k-1}$ $t_2$
where       $t_1.id = t_2.tid$ and
            $t_2.item = i$ and
            $t_1.item > i$


insert into $F$ $(item, count)$ select $item$, count(*)
from        $TEMP$
group by    $item$


insert into $PT\_k$ select $t.id, t.item$
from        $TEMP$ as t, $F$
where       $t.item = F.item$
having      count(*) $\geq$ $minsupp$
```

Figure 6.11: PT generation in SQL

6.2.2.2 SQL Query Using *Propad* Algorithm

The transaction table T is partitioned uniformly by hashing algorithm corresponding to transaction TID among processing nodes. In the first pass we simply gather the count of each item. Items that satisfy the minimum support are insert into the frequent itemset table F that takes the form (*item*, *count*). Transactions that match large itemsets are preserved in the transformed transaction table TF that has the same schema as transaction table T.

In next pass, for each frequent item i we construct its respective projected transaction table PT_i. Frequent 1-items are regarded as the prefixes, frequent 2-patterns are gained by simply combining the prefixes and their local frequent itemsets. we successively construct the projected transaction table $PT_{i,j}$ and gain its local frequent items. One projected transaction table is filtered if each transaction in the table only maintains items that contribute to the further construction of descendants.

The SQL queries used to create the frequent itemsets table and the projected transaction table are illustrated in Figure 6.11.

6.2.3 Parallel *Ppropad*

The *Ppropad* approach we propose consists of two main stages.

Stage one is the construction of the transformed transaction table TF that includes all frequent 1-items. In order to enumerate the frequent items efficiently, the transaction data is partitioned uniformly correspond to transaction TID among the available processors. In a partitioned database, this can be done automatically. The statements for generating TF from the transaction table T are showed as follows. The table F stores all frequent 1-itemsets.

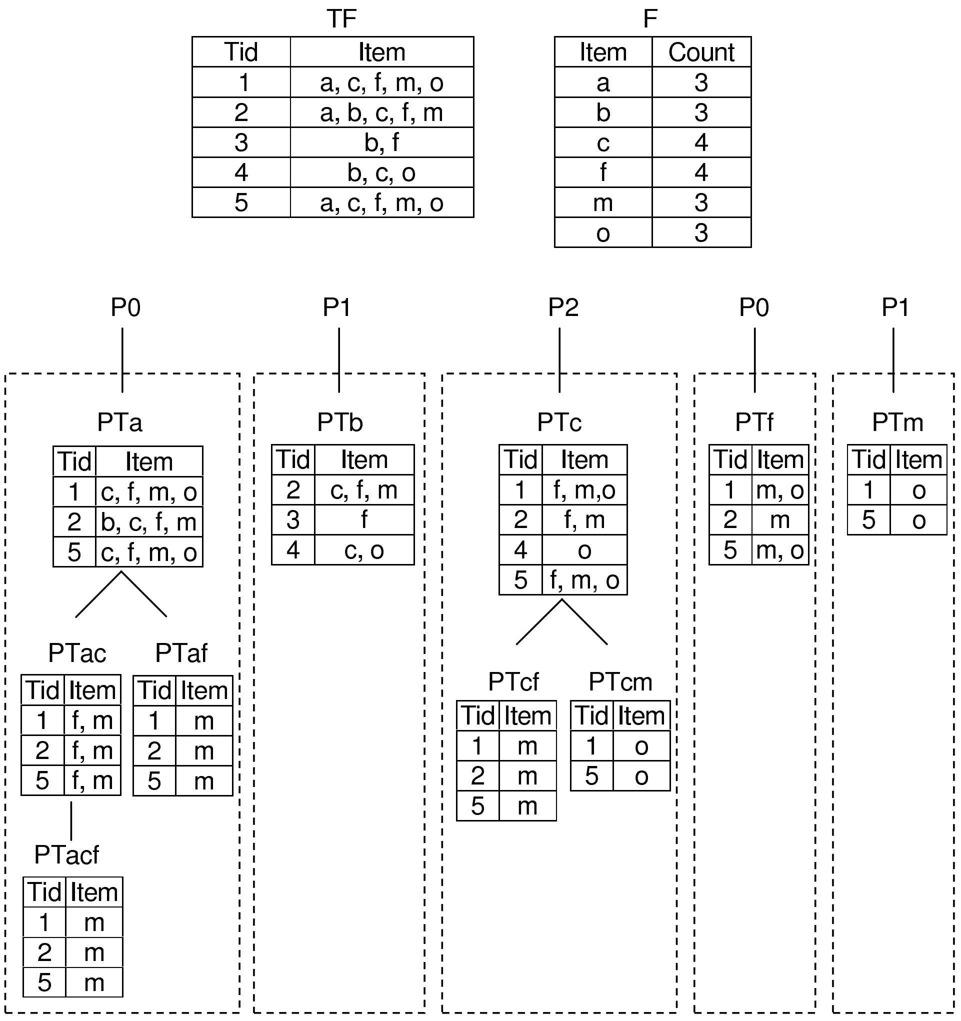
```
insert into F select item, count(*)
from        T
group by    item
having      count(*) ≥ minsupp
```

```
insert into TF select T.id, T.item
from        T, F
where       T.item = F.item
```

These two statements are suitable for a partitioned database. The former allows each partition to count its rows and send the subtotal to the coordination partition for computing the overall count, and the communication costs is negligible compared with the work done within each partition. The later involves in a large table joining with a small table. The large table is partitioned, and the small one is replicated on all partitions, enabling the joins to be collocated.

Stage two is the actually mining of the table by projecting. In the *Ppropad* approach, the projecting process is facilitated by depth first approach. Since the processing of the projection of one frequent itemset is independent from those of others, it is natural to consider it as the execution unit for the parallel processing.

Following the order of frequent items found at stage one, the *Ppropad* divides complete set of frequent patterns into subsets associated with frequent items without



TF

Tid	Item
1	a, c, f, m, o
2	a, b, c, f, m
3	b, f
4	b, c, o
5	a, c, f, m, o

F

Item	Count
a	3
b	3
c	4
f	4
m	3
o	3

PTa

Tid	Item
1	c, f, m, o
2	b, c, f, m
5	c, f, m, o

PTac

Tid	Item
1	f, m
2	f, m
5	f, m

PTaf

Tid	Item
1	m
2	m
5	m

PTacf

Tid	Item
1	m
2	m
5	m

PTb

Tid	Item
2	c, f, m
3	f
4	c, o

PTc

Tid	Item
1	f, m,o
2	f, m
4	o
5	f, m, o

PTcf

Tid	Item
1	m
2	m
5	m

PTcm

Tid	Item
1	o
5	o

PTf

Tid	Item
1	m, o
2	m
5	m, o

PTm

Tid	Item
1	o
5	o

Figure 6.12: Parallel *Propad*

overlap. In that case, for each frequent item, the processing of mining can be done on each node. We divide the frequent items among the available nodes using a round robin fashion. Each node is given an approximately equal number of items to read and analyze. As a result, the items is spilt in p equal size, supposed p is the number of available nodes. Each node locally constructs the projected transaction tables associated with the items in hand until the the search for frequent patterns associated with the items terminates. The basic idea of stage two is illustrated in the following example. Let the number of available nodes be 3, as listed in Figure 6.12.

The algorithm *Parallel PROjection PAttern Discovery*, abbreviated as *Ppropad*, as follows. The number of iterations in the *for* loop is one less than the number of frequent 1-itemsets.

Algorithm 10 *Ppropad*

```
Input: A transaction table $T$ and a minimum support threshold $\xi$
Output: A frequent pattern table $PAT$
Procedure:
1. $pass\_num := 0$;
2. $prefix := null$;
3. get the transformed transaction table $TF$ by removing infrequent items from $T$;
4. insert the frequent 1-items into $PAT$;
5. divide frequent 1-items among the available nodes;
6. for each node
7.     for each frequent item $i$
8.         $prefix := i$;
9.         call findFP$(prefix, 1)$;


findFP$(prefix, k)$
if $PT_k$ has at least one frequent item
  combine $prefix$ with frequent item sets and insert them into $PAT$;
  if $PT_k$ is not be filtered
    if $k + 1 > pass\_num$
      create table $PT_{k+1}$;
      $pass\_num = k + 1$;
    else clear table $PT_{k+1}$;
    construct $PT_{k+1}$ by projection;
    for each frequent item $j$ in $PT_{k+1}$
        $prefix := prefix + j$;
        findFP $(prefix, k + 1)$;
```

6.3 Experimental Evaluation and Performance Study

6.3.1 Parallel Execution Environment

In our experiment we built a parallel RDBMS: IBM DB2 UDB EEE version 8.1 on multiple nodes. DB2 UDB EEE takes advantages of shared-nothing clustering. It can optimize its dynamic partitioning for either symmetric multiprocessor (SMP)-style parallelism with a low overhead, single copy of EEE or across a cluster using a share-nothing architecture with multiple instances of EEE.

We configure DB2 EEE to execute in a shared-nothing architecture that each node has exclusive access to its own disk and memory. Four nodes were employed in our experiments. Each node runs the Linux operation system on Intel Xeon 2.80Ghz with 1G of main memory.

DB2 UDB EEE supports parallel queries by using intelligent database partitioning. In parallel configuration, DB2 UDB EEE distributes the data and database functions on multiple nodes using a hashing algorithm. Data scans, joins, sorts, load balancing, table reorganization, data load, index creation, index access, backup and restore are all performed simultaneously on all hosts in the DB2 cluster. DB2 has predictably scalable performance to accommodate more users and more data.

6.3.2 Data Set

We report experimental results on two synthetic data sets. They are respectively T25I20D100K, T10I4D100K. All the datasets have described in Chapter 4. Transaction data is partitioned uniformly by hashing algorithm corresponds to transaction TID among processing nodes.

6.3.3 Performance Comparison

In this subsection, we describe our algorithm performance compared with K-Way join. Figure 6.13 (a) shows the execution time for T10I4D100 with the minimum support of 0.1% and 0.06% on each degree of parallelization. We can drive that *Propad* is faster than K-Way join as the minimum support threshold decreases. This is because for datasets with long patterns, joining k-copies of input table for support counting at higher passes is quite significant though the cardinality of the C_k decreases with the increase in the number of passes. The speedup ration is shown in Figure 6.13 (b).

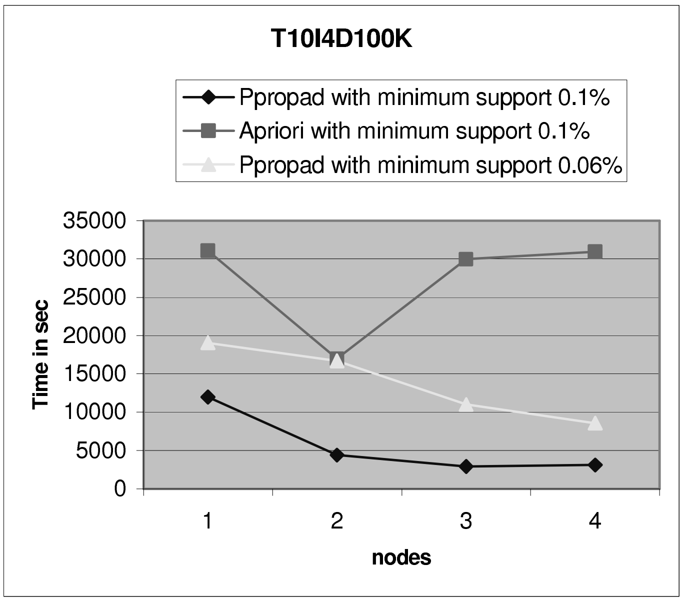
Figure 6.14 shows the execution time and speedup ration for T25I20D100K with the minimum support of 0.2% and 0.1% on each degree of parallelization. The speedup ratio shown in Figure 6.14 (b) seems to decrease with 4 processing nodes. It might be caused by the communication overhead because more coordination and more overhead are required for a process working on a disk belonging to another node with a shared nothing system.

From the results we can see that the *Ppropad* approach has better parallelization than K-Way join. This is because for K-Way join approach with many large tables and a wide variety of tables and columns involved in joins, it can be difficult or impossible to choose the table's partitioning key such that all significant queries can be executed without heavy inter-partition communication. While, *Ppropad* approach avoids complex joins between tables that poorly suited for partitioning, as well as explores task parallelism.

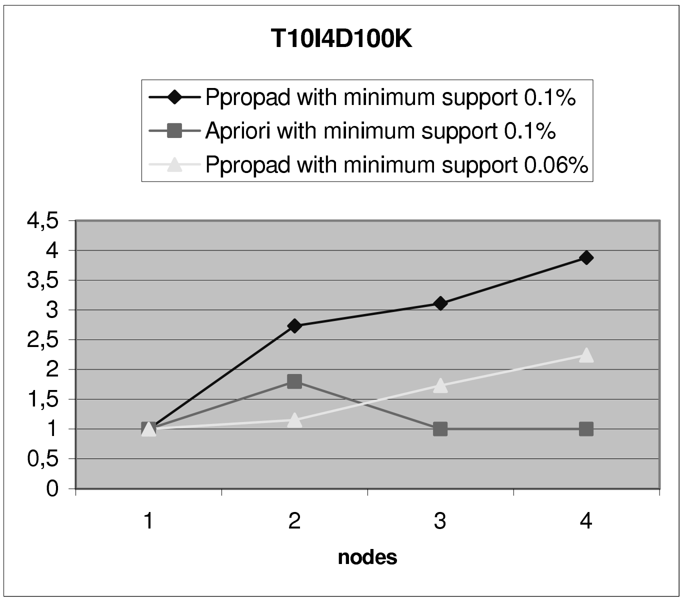
6.4 Conclusion

The ability to perform data mining using SQL queries will benefit data warehouses with the better integration with commercial RDBMS. Particularly, unlike taking advantage of parallel architecture using specific programming routines, a parallel database system can be used to provide performance improvement easily and flexibly since parallel execution of SQL comes at no extra costs, although their engines are not specifically optimized to handle data mining applications. It also allows easier porting codes among different systems.

In this chapter, we have configured a parallel DB2 database to execute in a shared-nothing architecture. In such an environment, databases are partitioned across multiple machines. This partitioning enables the RDBMS to perform complex parallel data operations.

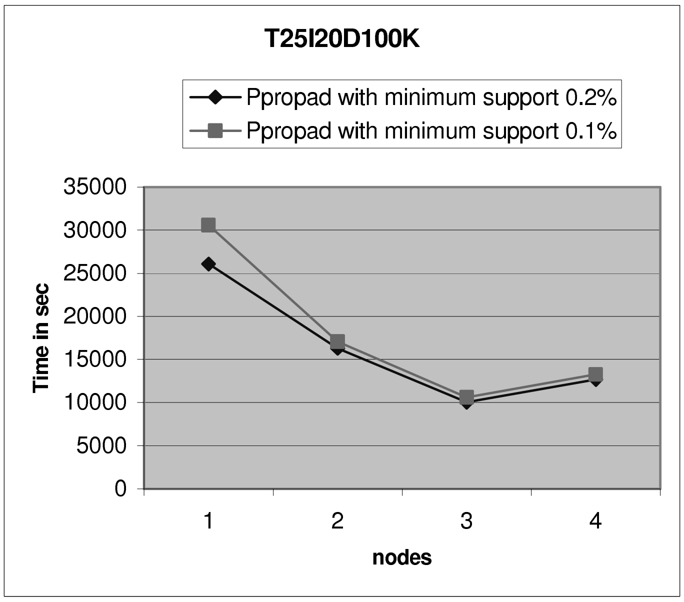


(a)

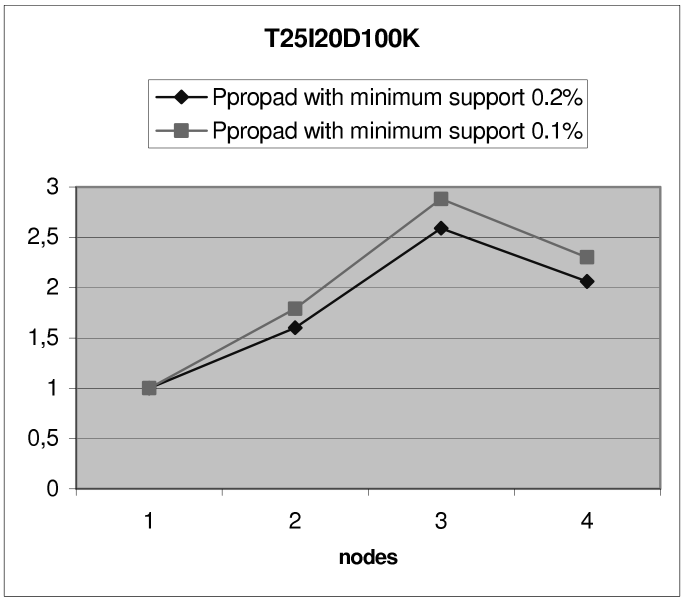


(b)

Figure 6.13: Execution time (top) Speedup ration (bottom)



(a)



(b)

Figure 6.14: Execution time (top) Speedup ration (bottom)

We implemented the parallelization of SQL based algorithm, *Ppropad*, to mine frequent itemsets from databases. Rather than *Apriori*-like method it adopts the divide-and-conquer strategy and projects the transaction table into a set of frequent item-related projected tables. As shown in our performance study, the *Ppropad* algorithm can get better speedup ratio than K-Way join based on *Apriori*-like on all data sets, that means it is parallelized well.

There remain lots of further investigations. It's obvious to achieve good parallelization, we have to consider the granularity of the execution unit or parallel task. In particular, although the trivial approach uses hashing of frequent items partition among the nodes, the time to process each projection could vary. The load balancing is a problem when the extreme skew exists in data. We would like to examine how to absorb such skew.

We also plan to do our parallel SQL based frequent pattern mining approach using more large transaction data. In addition, we'd like to investigate the effect of intra parallelism under SMP environment.

Chapter 7

Conclusions and Future Work

Scalable data mining in large databases is one of today's real challenges to database research area. The integration of data mining with database systems is an essential component for any successful large-scale data mining application. A fundamental component in data mining tasks is finding frequent patterns in a given dataset. In this thesis, we focus on the problem of efficiently mining frequent patterns with the help of SQL, and develop a new class of approaches.

This chapter is organized as follows. We first summarize the thesis in Section 7.1. Section 7.2 discuss some future research directions.

7.1 Summary

Data mining on large relational databases has gained popularity and its significance is well recognized. However, the performance of SQL based data mining is known to fall behind specialized implementation since the prohibitive nature of the cost associated with extracting knowledge, as well as the lack of suitable declarative query language support.

Most of the previous studies adopt an *Apriori*-like candidate set generation-and-test approach. However, candidate set generation is still costly, especially when there exist prolific patterns and/or long patterns. In this thesis, we investigate and propose a class of approaches based on SQL for the problem of finding frequent patterns from a transaction table. The major contributions of my thesis work are summarized as follows.

- We propose a relational presentation of *FP-tree*, which is an extended prefix-tree for storing compressed, complete information about frequent patterns, and develop two methods, *FP* and *EFP* to construct *FP-tree* table. Based on the *FP-tree* table, we develop an efficient pattern growth method for mining the complete set of frequent patterns. Comparison with SQL-based *Apriori* approach, *FP* and *EFP* have efficient performance gain with following techniques:
  - Table *FP* is usually substantially smaller than the transaction table T and is to be used in the subsequent mining processes. Thus it saves the costly join with the original transaction table.
  - Pattern growth method avoids costly candidate generation and test by successively concatenating frequent 1-itemset found in the (*ConFP*) *FP*. In this context, the major operations of mining are count accumulation and prefix path count adjustment which is usually much less costly than candidate generation and pattern matching operations performed in *Apriori*-like algorithms.
- One major cost of *FP* and *EFP* is that it has to materialize a large number of table *FP*s and *ConFP*s. To overcome this disadvantage, we propose an efficient SQL based method, *Propad*, to mine frequent itemsets. It is essentially a frequent pattern growth approach since it partitions its search space according to both patterns and data based on a divide and conquer methodology. Experimental study shows that *Propad* algorithm can get higher performance than *K-Way join* based on *Apriori*-like on all data sets.

  We next implement a hybrid approach from *Propad* and *K-Way join*. Our study shows that *Hybrid* can achieve the efficiency and scalability when the value of level k is well selected.

- One typical problem is that real-world databases tend to be very large. Parallelism can be expected to relieve current frequent pattern mining methods from the sequential bottleneck, providing scalability to massive data sets and improving response time. At first glance it seems that parallel processing introduces an additional complexity. However, this drawback may be avoided in our architectural framework, as follows. A parallel database system is used to provide performance improvement. The parallel DBMS is responsible itself for parallelizing queries that are issued against it. The SQL implementation can be easily parallelized when written with due care.

  We report the parallelization of *Propad* to mine frequent patterns on a parallel RDBMS (IBM DB2 UDB EEE). As shown in our performance study, good speedup ratio can be achieved, that means it is parallelized well.

7.2 Future Research Directions

Future work might involve in exploring many related problems, extensions and applications. Some of them are listed as follows.

- Investigation of the parallelization under more nodes environment and other architectures such as shared-memory parallel. Based on this, data mining middleware for parallel frequent pattern mining would be proposed.

- There are some kinds of knowledge to be mined. For example, closed association rules, sequential pattern mining and multi-dimensional sequential patterns.

  In classical frequent pattern mining, the common framework is to use a *min_sup* threshold to ensure the generation of the correct and complete set of frequent patterns. However, without specific knowledge, setting *min_sup* is quite subtle. On the other hand, frequent pattern mining often leads to the generation of a large number of patterns. Instead of mining the complete set of frequent itemsets and their associations, association mining only needs to find frequent closed itemsets and their corresponding rules [PBTL99, PHM00, ZH02]. Since a closed pattern covers all of its subpatterns with the same support, one just need to mine the set of closed patterns without losing information. The important implication is that mining frequent closed itemsets has the same power as mining the complete set of frequent itemsets, but it substantially reduces the redundant rules generated and increases both effectiveness and efficiency of mining. Mining top-k frequent closed patterns is another particular interesting problem [HWLT02], where k is a user-defined number of frequent closed patterns to be mined, top-k refers to the k most frequent closed patterns.

Sequential pattern mining, which finds the set of frequent subsequences in sequence database, is an important data mining task and has broad applications. Multi-dimensional sequential pattern mining integrates multi-dimensional analysis and sequential data mining [PHJ+01]. Some methods has been proposed since last few years. There is no study based on SQL available. The extension and implementation of our approaches for mining frequent closed itemsets and sequential patterns are very interesting for future research.

Mining frequent patterns when data reside in more than one table. The term Multi-Relational Data Mining has been introduced by Knobbe et al. in [KBSW99], to describe the problem of finding interesting patterns about sets of tuples belonging to a user selected table, named target table. Several studies based on *Apriori* has been proposed. However, running the *Apriori* algorithm on the join of tables of a database can fail to produce all existing rules or may produce rules whose support and confidence do not properly reflect the knowledge embedded in the data [Cri02]. Actually, the entity-relationship model of a database provides important information concerning the relations between entities, this information can be used by data mining. How to take advantage of this information represents an interesting direction for future research.

- Building SQL-aware data mining systems. There is a need to look for generic scalable implementation over SQL systems for each class of data mining algorithms, rather than for specific scalable algorithm in each class. In addition, flexible and efficient visualization support is necessary to SQL-aware data mining systems.

7.2.1 Final Thoughts

Data mining is that: "An information extraction activity whose goal is to discover hidden facts contained in data bases".

The word "discover" is related to the fact that the most valuable information is not previously known. however, the mining techniques may help to confirm any suspected behavior of the system in a particular context. By mining, we can see the patterns hidden behind the data more accurately, more systematically, more efficiently.

Bibliography

[AAP00] R. Agarwal, C. Aggarwal, and V.V.V. Prasad. Depth first generation of long patterns. In *Proc. of 6th ACM SIGKDD Int. Conf. on Knowledge Discovery and Data Mining)*, pages 108–118, Boston, MA, USA, August 2000. ACM Press.

[AAP01] R. Agarwal, C. Aggarwal, and V.V.V. Prasad. A tree projection algorithm for generation of frequent itemsets. *Parallel and Distributed Computing*, 61(3):350–371, 2001.

[AGI$^+$92] R. Agarwal, S. Ghosh, T. Imielinskie, , B. Iyer, and A. Swami. An interval classifier for database mining applications. In *Proc. of 18th Int. Conf. on Very Large Data Bases (VLDB'92)*, pages 560–573, Vancouver, Canada, August 1992. Morgan Kaufmann Publishers Inc.

[AIS93] R. Agarwal, T. Imielinski, and A. Swam. Mining association rules between sets of items in large databases. In *Proc. 1993 ACM-SIGMOD Int. Conf. Management of Data (SIGMOD'93)*, pages 207–216, Washington, D.C., USA, May 1993. ACM Press.

[AMS$^+$96] R. Agrawal, H. Mannila, R. Srikant, H. Toivonen, and A. I. Verkamo. Fast discovery of association rules. *Advances in knowledge discovery and data mining*, pages 307–328, 1996.

[AS] R. Agrawal and R. Srikant. Quest synthetic data generator. In *IMB Almaden Research Center*, San Jose, California, USA.

[AS94] R. Agarwal and R. Srikant. Fast algorithm for mining association rules in large databases. In *Proc. 1994 Int. Conf. on Very Large Data Bases (VLDB'94)*, pages 487–499, Santiago, Chile, September 1994. Morgan Kaufmann Publishers Inc.

[AS95] R. Agarwal and R. Srikant. Mining sequential patterns. In *Proc. 1995 Int. Conf. Data Engineering (ICDE'95)*, pages 3–14, Taipei, Taiwan, March 1995. IEEE Computer Society Press.

[AS96] R. Agarwal and R. Shim. Developing tightly-coupled data mining application on a relational database system. In *Proc.of the 2nd Int. Conf. on Knowledge Discovery in Database and Data Mining (KDD-96)*, pages 287–290, Portland, Oregon, August 1996. ACM Press.

[AW97] C. Apte and S. Weiss. Data mining with decision trees and decision rules. *FGCS Joural, Special Issue on Data Mining*, 13(2-3):197–210, 1997.

[Bay98] R. J. Bayardo. Efficient mining long patterns from databases. In *Proc. of 1998 ACM-SIGMOD Int. Conf. on Management of Data*, pages 85–93, Seattle, Washington, USA, June 1998. ACM Press.

[BBMM04] M. Botta, J. F. Boulicaut, C. Masson, and R. Meo. Query languages supporting descriptive rule mining: a comparative study. In *Database Support for Data Mining Applications*, volume 2682 of *Lecture Notes in Computer Science*, pages 27–54. Springer, 2004.

[BCV93] B. Bergsten, M. Couprie, and P. Valduriez. Overview of parallel architectures for databases. *The Computer Journal*, 36(8):734–740, 1993.

[Ber02] P. Berkhin. Survey on clustering data mining techniques. Technical report, Accrue Software, San Jose, CA, 2002.

[BHM98] B.Liu, W. Hsu, and Y. Ma. Integrating classification and association rule mining. In *Knowledge Discovery and Data Mining*, pages 80–86, 1998.

[BMS97] S. Brin, R. Motwani, and C. Silverstein. Beyond market baskets: Generalizing association rules to correlations. In *SIGMOD 1997: Proceedings of the 1997 ACM SIGMOD Int. Conf. on Management of Data*, pages 265–276, Tucson, Arizona, USA, May 1997. ACM Press.

[BMUT97] S. Brin, R. Motwani, J.D. Ullman, and S. Tsur. Dynamic itemset counting and implication rules for market basket analysis. In *SIGMOD 1997: Proceedings of the 1997 ACM SIGMOD Int. Conf. on Management of data*, pages 255–264, Tucson, Arizona, USA, May 1997. ACM Press.

[Bor03] C. Borgelt. Efficient implementations of apriori and eclat. In *1st Workshop of Frequent Item Set Mining Implementations (FIMI 2003)*, Melbourne, FL, USA, November 2003.

[BR99] K. Beyer and R. Ramakrishnan. Bottom-up computation of sparse and Iceberg CUBE. In *SIGMOD '99: Proceedings of the 1999 ACM SIGMOD international conference on Management of data*, pages 359–370, Philadelphia, Pennsylvania, United States, 1999. ACM Press.

[CHX98] D. W. Cheung, K. Hu, and S. Xia. Asynchronous parallel algorithm for mining association rules on a shared-memory multi-processors. In *Proc. of the tenth annual ACM Symposium on Parallel Algorithms and architectures*, pages 279–288, Puerto Vallarta, Mexico, June/July 1998. ACM Press.

[Cri02] L. Cristofor. *Mining Rules in Single-table and Multiple-table Databases.* PhD thesis, University of Massachusetts Boston, Boston, USA, 2002.

[DL99] G. Dong and J. Li. Efficient mining of emerging patterns: Discovering trends and differences. In *Knowledge Discovery and Data Mining*, pages 43–52, 1999.

[DS99] B. Dunkel and N. Soparkar. Data organization and access for efficient data mining. In *Proc. of the 25th Int. Conf. on Data Engineering*, pages 522–529, Sydney, Australia, 1999. IEEE Computer Society.

[FPSM91] W. J. Frawley, G. Piatetstly-Shapiro, and C. J. Matheus. Knowledge discovery in databases: An overview. *Knowledge Discovery in Databases*, pages 1–30, 1991.

[GG02] R. Grossman and Y. Guo. Data mining tasks and methods: parallel methods for scaling data mining algorithms to large data sets. *Handbook of data mining and knowledge discovery*, pages 433–442, 2002.

[Goe02] B. Goethals. *Efficient frequent pattern mining.* PhD thesis, University of Limburg, Belgium, 2002.

[Goe03] B. Goethals. Survey on frequent pattern mining. Manuscript, 2003.

[HB84] K. HWang and F.A. Briggs. *Computer Architecture and Parallel Processing.* McGraw-Hill, Inc., 1984.

[HDY99] J. Han, G. Dong, and Y. Yin. Efficient mining of partial periodic patterns in time series database. In *Fifteenth Int. Conf. on Data Engineering*, pages 106–115, Sydney, Australia, March 1999. IEEE Computer Society.

[HFK$^+$96] J. Han, Y. Fu, K. Koperski, W. Wang, and O. Zaiane. Dmql: A data mining query language on data mining and knowledge discovery. In *Proc. 1996 ACM-SIGMOD workshop on Research Issues on Data Mining and Knowledge Discovery*, Montreal, Canada, May 1996.

[HFW$^+$96] J. Han, Y. Fu, W.Wang, J. Chiang, W. Gong, K. Koperski, D. Li, Y. Lu, A. Rajan, N. Stefanovic, B. Xia, and O. R. Zaiane. DBMiner: A system for mining knowledge in large relational databases. In *Proc. 1996 Int. Conf. on Data Mining and Knowledge Discovery (KDD'96)*, pages 250–255, Portland, Oregon, August 1996.

[HGN00a] J. Hipp, U. Güntzer, and G. Nakhaeizadeh. Algorithms for association rule mining - a general survey and comparison. *SIGKDD Explorations*, 2(1):58–64, July 2000.

[HGN00b] J. Hipp, U. Güntzer, and G. Nakhaeizadeh. Mining association rules: Deriving a superior algorithm by analysing today's approaches. In *Proc. of the The Fourth European Conference on Principles and Practice of Knowledge Discovery in Databases (PKDD'00)*, Lyon, France, September 2000. Springer.

[HK00] J. Han and M. Kamber. Data mining concepts and techniques. *The Morgan Kaufmann Series in Data Management Systems*, 2000.

[HKK97] E. Han, G. Karypis, and V. Kumar. Scalable parallel data mining for association rules. In *Proc. of 1997 ACM-SIGMOD Int. Conf. on Management of Data*, pages 277–288, Tucson, Arizona, USA, May 1997. ACM Press.

[HPY00] J. Han, J. Pei, and Y. Yin. Mining frequent patterns without candidate generation. In *Proc. 2000 ACM-SIGMOD Int. Conf. Management of Data (SIGMOD'00)*, pages 1–12, Dallas, Texas, USA, May 2000. ACM Press.

[HPYM04] J. Han, J. Pei, Y. Yin, and R. Mao. Mining frequent patterns without candidate generation: A freuqnt pattern-tree approach. *Data Mining and Knowledge Discovery*, 8(1):53–87, January 2004.

[HS95] M. Houtsma and A. Swami. Set-oriented data mining in relational databases. *Data Knowledge Engineering*, 17(3):245–262, 1995.

[HWLT02] J. Han, J. Wang, Y. Lu, and P. Tzvetkov. Mining top-k frequent closed patterns without minimum support. In *Proc. of the 2002 IEEE Int. Conf. on Data Mining (ICDM'02)*, page 211, Maebashi TERRSA, Maebashi City, Japan, December 2002. IEEE Computer Society.

[IM96] T. Imielinski and H. Mannila. A database perspective on knowledge discovery. *Communication of ACM*, 39(11):58–64, 1996.

[Imi96] T. Imielinski. From file mining to database mining. In *ACM-SIGMOD'96 Data Mining Workshop Advance Program and Registration Information*, pages 35–39, Montreal, Canada, May 1996.

[IV99] T. Imielinski and A. Virmani. Msql: A query language for database mining. *Data Mining and Knowledge Discovery*, 3(4):373–408, December 1999.

[IVA96] T. Imielinski, A. Virmani, and A. Abdulghani. Discovery board application programming interface and query language for database mining. In *Proc. of the 2nd Int. Conf. on Knowledge Discovery and Data Mining*, pages 20–26, Oregon, Portland, August 1996. AAAAI Press.

[JL96] G. H. John and P. Langley. Static versus dynamic sampling for data mining. In Evangelos Simoudis, Jiawei Han, and Usama M. Fayyad, editors, *Proc. 2nd Int. Conf. Knowledge Discovery and Data Mining KDD'96*, pages 367–370, Portland, Oregon, August 1996. AAAI Press.

[KBSW99] A. J. Knobbe, H. Blockeel, A. Siebes, and D. Wallen. Multi-relational data mining. Technical Report INS-R9908, March, 1999.

[KHC97] M. Kamber, J. Han, and J. Chiang. Metarule-guided mining of multi-dimensional association rules using data cubes. In *Proc. of the Third Int. Conf. on Knowledge Discovery and Data Mining (KDD'97)*, pages 207–210, Newport Beach, California, USA, August 1997. AAAAI Press.

[KI91] R. Krishnamurthy and T. Imielinski. Practitioner problems in need of database research: Research directions in knowledge discovery. *SIGMOD RECORD*, 20(3), September 1991.

[KMR$^+$94] M. Klemettinen, H. Mannila, P. Ronkainen, H. Toivonen, and A. I. Verkamo. Finding interesting rules from large sets of discovered association rules. In Nabil R. Adam, Bharat K. Bhargava, and Yelena Yesha, editors, *Third International Conference on Information and Knowledge*

Management (CIKM'94), pages 401–407, Gaithersburg, Maryland, USA, Nov./Dec. 1994. ACM Press.

[KP03] W. A. Kosters and W. Pijls. Apriori: A depth first implementation. In *FIMI'03: the first Workshop on Frequent Itemset Mining Implementations*, Melbourne, Florida, USA, December 2003.

[LLXY03] G. Liu, H. Lu, Y. Xu, and J. X. Yu. Ascending frequency ordered prefix-tree: Efficient mining of frequent patterns. In *Proc. 8th Int. Conf. on Database Systems for Advanced Applications (DASFAA 2003)*, pages 65–72, Kyoto, Japan, March 2003.

[LPWH02] J. Liu, Y. Pan, K. Wang, and J. Han. Mining frequent item sets by opportunistic projection. In *Proc. 2002 SIGKDD Int. Conf. Knowledge Discovery and Data Mining (SIGKDD'02)*, pages 23–26, Alberta, CA, July 2002. ACM Press.

[LSW97] B. Lent, A. N. Swami, and J. Widom. Clustering association rules. In *ICDE '97: Proceedings of the Thirteenth Int. Conf. on Data Engineering*, pages 220–231, Birmingham, U.K, April 1997. IEEE Computer Society.

[MAR96] M. Metha, R. Agrawal, and J. Rissanen. Sliq: A fast scalable classifier for data mining. In *Proc. of the fifth Int. Conf. on Extending Database Technolgy (EDBT'96)*, pages 18–32, Avignon, France, March 1996. Springer-Verlag.

[MPC96] R. Meo, G. Psaila, , and S. Ceri. A new sql like operator for mining association rules. In *Proc. Of the 22nd Int. Conf. on Very Large Databases (VLDB'96)*, pages 122–133, Bombay, India, September 1996. Morgan Kaufmann Publishers Inc.

[MTV94] H. Mannila, H. Toivonen, and A. I. Verkamo. Efficient algorithms for discovering association rules. In Usama M. Fayyad and Ramasamy Uthurusamy, editors, *AAAI Workshop on Knowledge Discovery in Databases (KDD-94)*, pages 181–192, Seattle, Washington, 1994. AAAI Press.

[MTV97] H. Mannila, H. Toivonen, and A. I. Verkamo. Discovery of frequent episodes in event sequences. *Data Mining and Knowledge Discovery*, 1(3):259–289, 1997.

[NZT96] M.G. Norman, T. Zurek, and P. Thanish. Much ado about shared-nothing. *SIGMOD Records*, 25(3):16–21, September 1996.

[OPP01] S. Orlando, P. Palmerini, and R. Perego. Enhancing the apriori algorithm for frequent set counting. In *DaWaK '01: Proceedings of the Third International Conference on Data Warehousing and Knowledge Discovery*, pages 71–82, Munich, Germany, September 2001. Springer-Verlag.

[OPPS02] S. Orlando, P. Palmerini, R. Perego, and F. Silvestri. Adaptive and resource-aware mining of frequent sets. In *Proc. 2002 IEEE International Conference on Data Mining (ICDM'02)*, page 338, Maebashi City, Japan, December 2002. IEEE Computer Society.

[PB99] B. Pijls and J. C. Bioch. Mining frequent itemsets in memory-resident databases. In *Proc. of the Eleventh Belgium-Netherlands Conf. on Artifical Intelligence (BNAIC 1999)*, pages 75–82, Belgium, Netherland, November 1999.

[PBTL99] N. Pasquier, Y. Bastide, R. Taouil, and L. Lakhal. Discovering frequent closed itemsets for association rules. In *ICDT'99: Proceeding of the 7th International Conference on Database Theory*, pages 398–416, Jerusalem, Israel, January 1999. Springer-Verlag.

[PCY95a] J.S. Park, M.S. Chen, and P.S. Yu. An effective hash-based algorithm for mining association rules. In *Proc. 1995 ACM-SIGMOD Int. Conf. Management of Data (SIGMOD'95)*, pages 175–186, San Jose, CA, May 1995. ACM Press.

[PCY95b] J.S. Park, M.S. Chen, and P.S. Yu. Efficient parallel data mining for association rules. In *Proc. ACM Int. Conf. Information and Knowledge*

Management, pages 31–36, Baltimore, Maryland, USA, Nov./Dec. 1995. ACM Press.

[PH00] J. Pei and J. Han. Can we push more constraints into frequent pattern mining? In *KDD'00: Proceedings of the sixth ACM SIGKDD int. conf. on Knowledge discovery and data mining*, pages 350–354, Boston, Massachusetts, United States, 2000. ACM Press.

[PH02] J. Pei and J. Han. Constrained frequent pattern mining: a pattern-growth view. *ACM SIGKDD Explorations Newsletter*, 4(1):31–39, 2002.

[PHJ$^+$01] H. Pinto, J. Han, J.Pei, K. Wang, Q. Chen, and U. Dayal. Multi-dimensional sequential pattern mining. In *CIKM'01: Proceedings of the tenth international conference on Information and knowledge management*, pages 81–88, Atlanta, Georgia, USA, November 2001. ACM Press.

[PHL$^+$01] J. Pei, J. Han, H. Lu, S. Nishio, S. Tang, and D. Yang. H-mine: Hyper-structure mining of frequent patterns in large databases. In *Proc. 2001 Int. Conf. on Data Mining (ICDM'01)*, San Jose, CA, November 2001. IEEE Computer Society.

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

[PK03a] I. Pramudiono and M. Kitsuregawa. Parallel fp-growth on pc cluster. In *Proc. Advances in Knowledge Discovery and Data Mining: 7th Pacific-Asia Conference, PAKDD 2003*, pages 467–473, Seoul, Korea, April 2003. Springer-Verlag.

[PK03b] I. Pramudiono and M. Kitsuregawa. Shared nothing parallel execution of fp-growth. In *DEWS2003*, March 2003.

[PSTK99] I. Pramudiono, T. Shintani, T. Tamura, and M. Kitsuregawa. Parallel sql based associaton rule mining on large scale pc cluster: performance comparision with directly coded c implementation. In *Proc. Of Third Pacific-Asia Conf. on Knowledge Discovery and Data Mining*, pages 94–98, Beijing, China, April 1999. Springer-Verlag.

[RCIC99] K. Rajamani, A. Cox, B. Iyer, and A. Chadha. Efficient mining for association rules with relational database systems. In *Proc. of the Int. Database Engineering and Applications Symposium (IDEAS'99)*, Montreal, Canad, August 1999.

[RMZ02] G. Ramesh, W. Maniatty, and M. J. Zaki. Indexing and data access methods for database mining. In *ACM SIGMOD Workshop on Research Issues on Data Mining and Knowledge Discovery (DMKD'02)*, Madison, Wisconsin, USA, June 2002.

[RS00] R. Rastogi and K. Shim. Public: A decision tree classifier that integrates building and pruning. *Data Mining Knowledge Discovery*, 4(4):315–344, 2000.

[SA96] R. Srikant and R. Agrawal. Mining sequential patterns: Generalization and performance improvements. In *Proc. Of Fifth Int. Conf. on Extending Database Technology (EDBT)*, pages 3–17, Avigon, France, March 1996. Springer-Verlag.

[SA97] R. Srikant and R. Agrawal. Mining generalized association rules. *Future Generation Computer Systems*, 13(2–3):161–180, 1997.

[SAM96] J. Shafer, R. Agrawal, and M. Mehta. Sprint: A scalable parallel classifier for data mining. In *Proc. Of 22nd Int. Conf. on Very Large Databases*, pages 544–555, Bombay, India, September 1996. Morgan Kaufmann.

[SK96] T. Shintani and M. Kitsuregawa. Hash based parallel algorithms for mining association rules. In *Int. Conf. on Parallel and Distributed Information Systems (PDIS)*, pages 19–30, Miami Beach, Florida, USA, December 1996. IEEE Computer Society.

[SON95] A. Savasere, E. Omiecinski, and S. Navathe. An effective algorithm for mining association rules in large databases. In *Proc. 1995 Int. Conf. Very Large Data Bases (VLDB'95)*, pages 432–443, Zurich, Switzerland, September 1995. Morgan Kaufmann.

[SS05] X. Shang and K. Sattler. Depth-first frequent itemset mining in relational databases. In *Proc. ACM Symposium on Applied Computing SAC 2005*, New Mexico, USA, 2005.

[SSG04a] X. Shang, K. Sattler, and I. Geist. Efficient frequent pattern mining in relational databases. In *Workshop on Knowledge Discovery in Databases (AKKD 2004)*, Berlin, Germany, 2004.

[SSG04b] X. Shang, K. Sattler, and I. Geist. Sql based frequent pattern mining with fp-growth. In *Proc. 2004 the 15th Int. Conf. on Applications of Declarative Programming and Knowledge Management and 18th Workshop on Logic Programming*, Berlin, Germany, March 2004.

[SSG04c] X. Shang, K. Sattler, and I. Geist. Sql based frequent pattern mining without candidate generation. In *Proc. 2004 ACM Symposium on Applied Computing (SAC'04)*, pages 618–619, Nicosia, Cyprus, March 2004. ACM Press.

[SSG05] X. Shang, K. Sattler, and I. Geist. Sql based frequent pattern mining with fp-growth. *Applications of Declarative Programming and Knowledge Management - LNAI 3392*, 2005.

[STA98] S. Sarawagi, S. Thomas, and R. Agrawal. Integrating association rule mining with relational database systems: Alternatives and implications.

In *Proc. 1998 ACM-SIGMOD Int. Conf. Management of Data (SIGMOD'98)*, pages 345–354, Seattle, WA, June 1998. ACM Press.

[TL01] M. C. Tseng and W. Y. Lin. Mining generalized association rules with multiple minimum supports. In *Proc. of Int. Conf. on Data Warehousing and Knowledge Discovery*, pages 11–20, Munich, Germany, September 2001. ACM Press.

[Toi96] H. Toivonen. Sampling large databases for association rules. In *Proc. 1996 Int. Conf. Very Large Data Bases (VLDB'96)*, pages 134–145, Bombay, India, September 1996. Morgan Kaufmann Publishers Inc.

[Vir98] A. Virmani. *Second generation data mining.* PhD thesis, Rutgers University, New Jersey, USA, 1998.

[WK91] S. M. Weiss and C. A. Kulikowski. Computer systems that learn: Classification and prediction methods from statistics. *Neural Nets, Machine Learning, and Expert Systems*, 1991.

[YPK00] T. Yoshizawa, I. Pramudiono, and M. Kitsuregawa. Sql based association rule mining using commercial rdbms (ibm db2 udb eee). In *DaWaK 2000: Proceedings of the Second International Conference on Data Warehousing and Knowledge Discovery*, pages 301–306, London, UK, 2000. Springer-Verlag.

[Zak00] M.J. Zaki. Scalable algorithms for association mining. *IEEE Transactions on Knowledge and Data Engineering*, pages 372–390, May/June 2000.

[ZEHL01] O.R. Zaiane, M. El-Hajj, and P. Lu. Fast parallel association rule mining without candidacy generation. In *Proc. of the 2001 IEEE International Conference on Data Mining*, pages 665–668, Washington, DC, USA, 2001.

[ZG03] M.J. Zaki and K. Gouda. Fast vertical mining using diffsets. In *Proc. 9th Int. Conf. ACM-SIGKDD Int. Conf. Knowledge Discovery and Data Mining*, pages 326–335, Washington, DC, USA, 2003. ACM Press.

[ZH02] M. J. Zaki and C. Hsiao. Charm: An efficient algorithm for closed itemset mining. In *Proceedings of the Second SIAM International Conference on Data Mining*, pages 457–473, Arlington, VA, USA, April 2002.

[ZKM01] Z. Zheng, R. Kohavi, and L. Mason. Real world performance of association rule algorithms. In *KDD '01: Proceedings of the seventh ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 401–406, San Francisco, California, 2001. ACM Press.

[ZOPL96] M.J. Zaki, M. Ogihara, S. Parthasarathy, and W. Li. Parallel data mining for association rules on shared-memory multi-processors. In *Conf. on High Performance Networking and Computing*, page 43, Pittsburgh, Pennsylvania, USA, 1996. ACM Press.

[ZPLO96] M. J. Zaki, S. Parthasarathy, W. Li, and M. Ogihara. Evaluation of sampling for data mining of association rules. Technical Report TR617, 1996.

[ZPOL97a] M.J. Zaki, S. Parthasarathy, M. Ogihara, and W. Li. New algorithms for fast discovery of association rules. Technical Report TR651, 1997.

[ZPOL97b] M.J. Zaki, S. Parthasarathy, M. Ogihara, and W. Li. Parallel algorithms for discovery of association rules. *Data Mining and Knowledge Discovery*, 1(4):343–373, 1997.