NEW TECHNIQUES FOR EFFICIENTLY DISCOVERING FREQUENT PATTERNS

DISSERTATION

Presented in Partial Fulfillment of the Requirements for

the Degree Doctor of Philosophy in the

Graduate School of The Ohio State University

By

Ruoming Jin, B.E., M.E., M.S.

* * * * *

The Ohio State University

2005

Dissertation Committee:

Gagan Agrawal, Adviser

Srinivasan Parthasarathy

Hakan Ferhatosmanoglu

Approved by

Adviser

Graduate Program in
Computer Science and
Engineering

ABSTRACT

Because of its theoretical and practical importance, the field of frequent pattern mining has been and remain to be one of the most active research area in KDD. In this dissertation, we study three different problems in frequent pattern mining, *mining multiple datasets*, *mining streaming data*, and *mining large-scale structures from graph datasets*. Our study has not only extended the breadth of frequent pattern mining, but also brought new techniques and algorithms into this field. Specifically, our contributions are as follows.

1. *Mining Multiple Datasets:* We develop a systematic approach to generate efficient query plans for a single mining query across multiple datasets. We also propose methods to simultaneously optimize multiple such queries and utilize the past mining results in a *query-intensive* KDD environment. Our experimental results have shown a speedup up to two-order of magnitude comparing with the naive methods without these optimizations.

2. *Mining Frequent Itemsets over Streaming Data:* We propose a new algorithm *StreamMining* to discover the frequent itemsets over streaming data. In a single pass, StreamMining will guarantee to find a superset of frequent itemsets, but false positive may occur. If the second pass is allowed, StreamMining will be able to remove the false positive and find the exact frequent itemsets. Our detailed evaluation using both synthetic and real datasets has shown our one-pass algorithm is very accurate in practice, and is also very memory efficient.

3. *Mining Frequent Large-Scale Structures from Graph Datasets:* We develop a new framework to discover the frequent large-scale structures from graph datasets. This framework is derived from a mathematical concept, *topological minor*. In this framework, we propose a new algorithm *TSMiner*, which efficiently enumerates all the frequent large-scale structures in a graph dataset, and a new approach called *relabeling function* to perform constraint mining. We apply our framework to protein structure data and discover meaningful topological structures. Finally, we demonstrate the viability and scalability of the proposed algorithms on both real and synthetic datasets.

To my parents,

Shiwei Jin and Jiuyun Zhu

ACKNOWLEDGMENTS

Foremost, I wish to thank my thesis adviser, Professor Gagan Agrawal. His vision and insight have led me into the field of data mining, and introduced me many interesting research problems. Some of them eventually became the topics of this dissertation. His patience, kindness, and encouragement have helped me to go through many difficulty times. His advice and research experience have helped to shape my research skills. Without him, this dissertation would not be possible.

I am also grateful to Professor Srinivasan Parthasarathy, who advised me and helped me in many aspects of my research. I thank him for spending his valuable time to discussing problems with me, carefully reading some of my papers, and providing valuable feedback. I have greatly benefited from discussions with him. I will also give a special thank to Professor Hakan Ferhatosmanoglu for providing useful feedback to improve the quality of this dissertation.

I would also like to thank my collegues, Kaushik Sinha, Chao Wang, and Dmitrii Polshakov. They have been collaborating with me and giving me great help in completing this dissertation. Specifically, Kaushik has helped to implement the system prototype and help with experimental evaluations for Chapter 3. Chao has helped with running experiments and generating figures for Chapter 6. Dmitrii has provided the protein datasets and analyzed the experimental results for Chapter 6.

I am also obliged to my friends and my collegues at the Ohio State University who have given all kinds of help and supports. They are: Wenbin Ma, Zhuyu Liu, Matthew Otey, Leo Glimcher, Wei Du, Xiaogang Li, Hui Yang, Liang Chen, and Xuan Zhang. Thank you very much!

Finally, I would like to thank my parents, my brother, and all my relatives in China for supporting me through all these years. Your love is the strength to keep me moving forward!

VITA

March 21, 1974	Born - ChangChun, JiLin Province, China
1996	B.E. Computer Engineering
1999	M.E. Computer Engineering
2001	M.S. Computer Science
2003-present	Graduate Research Associate, The Ohio State University.

PUBLICATIONS

Research Publications

1. *Fast and Exact Out-of-Core and Distributed K-Means Clustering*, Ruoming Jin, Anjan Goswami, and Gagan Agrawal, invited for publication in Knowledge and Information System (KAIS journal).

2. *Communication and Memory Optimal Parallel Data Cube Construction*, Ruoming Jin, Karthik Vaidyanathan, Ge Yang, and Gagan Agrawal, accepted in the IEEE transactions on Parallel and Distributed Systems (TPDS).

3. *A Methodology for Detailed Performance Modeling of Reduction Computations on SMP Machines*, Ruoming Jin and Gagan Agrawal, accepted in the special issue of "Performance Evaluation: An International Journal" on Performance Modeling and Evaluation of High-Performance Parallel and Distributed Systems.

4. *Shared Memory Parallelization of Data Mining Algorithms: Techniques, Programming Interface, and Performance*, Ruoming Jin, Ge Yang, and Gagan Agrawal, in the IEEE Transactions on Knowledge & Data Engineering (TKDE), Vol. 17, No. 1, January, 2005

5. *Implementing Data Cube Construction Using a Cluster Middleware: Algorithms, Implementation Experience, and Performance Evaluation*, Ge Yang, Ruoming Jin, and Gagan Agrawal, in Future Generation Computer Systems (FGCS), v. 19, i. 4, p. 533 - 550, 2003 .

6. *Research on Static Prediction and Visual Analysis of Program Execution Time*, Changai Sun, Maozhong Jin, Chao Liu, and Ruoming Jin, Journal of Software (Chinese), Vol. 14, No. 1, 2003, p: 68-75.

7. *Testing Technology of Real-time and Embedded Software*, Changai Sun, Ruoming Jin, Chao Liu, and Maozhong Jin, Journal of Mini Micro Systems (Chinese), Vol. 21, No. 9, 2000, p: 920-924.

FIELDS OF STUDY

Major Field: Computer Science and Engineering

Studies in Data Intensive Computing: Prof. Gagan Agrawal

TABLE OF CONTENTS

Page

LIST OF TABLES

LIST OF FIGURES

Figure **Page**

CHAPTER 1

INTRODUCTION

Since its introduction in [6], frequent pattern mining has received a great deal of attention. In the first several years, the research focused on mining frequent itemsets, or sequences from transaction datasets. More recently, many researchers have started to working with more complex structured datasets, such as protein, chemical compounds, and XML datasets. Mining frequent trees and graphs from such datasets has reinvigorated the field of frequent pattern mining. In over a decade, frequent pattern mining has been and still is one of the most popular research topics in KDD.

The significance of finding frequent patterns from datasets is two-folds. On one hand, frequent patterns can effectively summarize the underlying datasets, and provide key insights into the data. In many cases, such frequent patterns can even help the domain experts to gain knowledge of hidden mechanisms, which the data may represent. On the other hand, frequent pattern mining serves as the basic tool for many other data mining tasks, including association rule mining, classification, clustering, and change detection, among others [59, 120, 53, 63].

1.1 What is Frequent Pattern Mining?

Let the dataset D be a collection of objects, i.e. $D = \{o_1, o_2, \cdots, o_{|D|}\}$. Let P be the set of all possible (interesting) patterns occurring in D. Usually, we can define the containing ($\subset$) relation over P such that P satisfies the down-closure property: if $p \in P$, and for any $q \subset p$, $q \in P$. Further, we can define a counting function $g : P \times O \rightarrow N$, where O is the set of objects, and N is the set of nonnegative integers. Given parameters $p \in P$, and $o \in O$, $g(p, o)$ returns the number of times p occurs in o. The support of a pattern $p \in P$ in the dataset D is defined as

$$supp(p) = \sum_{j=0}^{j=|D|} I(g(p, o_j))$$

where, I is an *indicator* function: if $g(p, o_j) > 0$, $I(g(p, o_j)) = 1$; otherwise, $I(g(p, o_j)) = 0$. Given a support level θ, the frequent patterns of P in D is the set of patterns in P which have support greater than or equal to the θ. Note that the counting function usually has the following property: given two patterns p and q, if $q \subset p$, then for any o, $g(q, o) \geq g(p, o)$.

Given the above conditions, we have the well-known *down-closure* property for frequent pattern mining: *if p is a frequent pattern, then for any pattern $q \subset p$, q is also a frequent pattern*. Also, in most of the cases, the set P describes a class or a type of patterns, and usually will not explicitly be given for a frequent pattern mining problem.

Let us look at two typical types of frequent pattern mining problems: frequent itemsets and mining frequent patterns from graph datasets.

Frequent Itemsets Mining: In this setting, the objects in the dataset D are transactions or sets of items. Let *Item* be the set of all possible items in the dataset D. Then the dataset D can be represented as $D = \{I_1, \cdots, I_{|D|}\}$, where $I_j \subseteq Item, \forall j, 1 \leq j \leq |D|$. The set of all possible patterns P is the power-set of *Item*. Note that the set of all possible objects O is

the same as P in this setting. The counting function g is defined upon on the set containing ($\subseteq$) relationship. In other words, if the itemset p is contained in I_j ($p \subseteq I_j$), the function $g(p, I_j)$ returns 1; otherwise, it returns 0.

For instance, given a dataset D={{A,B,D,E}, {B,C,E},{A,B,E}, {A,B,C}, {A,C}, {B,C}}, and a support level $\theta = 50\%$, the frequent patterns are $\{A\}, \{B\}, \{C\}$, and $\{B, C\}$.

Mining Frequent Patterns from Graph Datasets (Graph Mining): In this setting, the dataset D is a collection of (labeled) graphs, i.e. $D = \{G_1, G_2, \cdots, G_{|D|}\}$, where $D_j, 1 \leq j \leq |D|$ are (labeled) graphs. Given a set of vertex labels, l_V and a set of edge labels l_E, two types of frequent mined patterns are subtrees or subgraphs which are labeled by l_V and l_E. The counting function g is defined upon the subgraph isomorphism test. In other words, the function $g(p, G_j)$ returns the number of distinct subgraphs (subtrees) of G_j which are isomorphism to the subgraph (or subtree) p.

Clearly, the down-closure property holds for both frequent itemset mining and graph mining. The main factors in frequent pattern mining can be largely determined by the complexity of objects in the dataset, and/or the complexity of the targeting patterns, the size of the dataset, and/or the number of datasets targeting. In the following section, we will give an overview of the existing research to deal with these issues.

1.2 Background

Frequent Pattern Mining Algorithms: One of the main efforts in this field has been to develop efficient mining algorithms. Depending on the complexity of objects in the dataset and the complexity of the targeting patterns, the mining algorithms can vary quite differently. However, the search strategies of these algorithms can be largely into two categories:

the level-wise approach and depth-first search (DFS) approach. For example, for frequent itemsets mining, Apriori [5] uses the first method, Eclat [117] and FP-Tree [48] uses the latter method; for graph mining, FSG [67] belongs the first category, gSpan citeYan02, FFSM [53], and Gaston [78] belongs the second category.

Mining Maximal, Closed Patterns, or Pattern with Constraint Conditions: One of the main issues in frequent pattern mining is the size of frequent patterns can often be very large. Such huge number of patterns can not only slow the mining algorithms, but also are very hard for data miners to analyze. Therefore, researchers have designed a cluster of methods to reduce the number of pattern being generated. In particular, maximal frequent patterns (MFP) are those patterns that are frequent but none of their supersets are frequent. Closed frequent patterns (CFP) are patterns that are frequent but have higher frequency than all of their supersets. (If pattern p is a subset of pattern q, q is called the superset of p.) Another approach allows users to specify a subset of frequent patterns being generated through constraint conditions. Many mining algorithms have been developed to mine MFP, CFP, and frequent patterns with constraint conditions. Similarly, to enumerate all frequent pattern, these algorithms perform either in the level-wise fashion [90, 84, 76] or the DFS fashion [42, 17, 54, 108, 85].

Mining Very Large Datasets (Scalability): If the datasets are too large to fit in the main memory, many mining algorithm will become very slow. This issue is often referred to as the scalability issue, and several methods have been proposed to scale mining algorithms. One type of methods use the small partitions of the datasets [95], or the sampling [100] to find the potential frequent itemsets, then scan the entire datasets to count the frequency of these patterns. Another type of methods design disk-resident data structured [31] or indexing [104] to facilitate the mining process.

Database Support for Frequent Pattern Mining: Implementing frequent pattern mining as a type of query in the database systems allow the users to perform the mining tasks easily. Several research groups have proposed extensions of the database query languages to support mining tasks, especially for frequent pattern mining [47, 57, 74]. Sarawagi and Agrawal [93] have studied implementing Apriori association mining algorithm on a database system. ATLas [112] applies user-defined functions (UDFs) to express the frequent pattern mining tasks.

Mining Contrast Patterns Given two datasets, the difference between their frequent patterns or the set of patterns having very different frequency (*high contrast set*) can be very useful to summarize or represent the difference between two datasets. A number of researchers have developed efficient algorithms for mining the difference or *contrast sets* between two datasets [12, 29, 106].

There are many other research topics in this field, such as summarizing or pruning frequent patterns to reduce the resulting patterns [61, 2], mining frequent patterns in parallel computers or distributed environments [4, 21, 116], privacy-aware frequent pattern mining [32, 103, 89], and the theoretical foundation of frequent pattern mining [15, 114, 110].

1.3 Thesis Contribution

In this thesis, we study the frequent pattern mining from three different perspectives, namely, the number of datasets, streaming data, and complex topological patterns from graph datasets.

1.3.1 Mining Multiple Datasets

Problems: In many real world situations, such as in data warehouse and scientific discovery, users usually have a view of multiple datasets collected from different data source or at

different time. In such scenarios, comparing the patterns from multiple datasets and understanding their relationships can be an extremely important part of the data mining process. Note that this problem can be looked as a generalization of finding the contrast or changing patterns, where only two datasets are targeted.

The problems that we are interested in mining multiple datasets are as follows. Given a single mining task on multiple datasets, what are the key optimization techniques? In other words, how we can evaluate such a mining task efficiently? Further, consider a mining intensive environment: a user analyze one or more datasets by issuing a sequence of related complex mining queries, and several users may be analyzing a set of datasets concurrently, and may issue related complex queries. In such an environment, how can we evaluate mining queries efficiently?

Our Contributions: To deal with the first problem, we transform the problem of mining frequent patterns across multiple datasets into a query evaluation problem. Then, we present several heuristic algorithms for finding efficient query plans. Our query optimization techniques demonstrate up to an order of magnitude performance gains as compared to the naive execution on both real and synthetic datasets.

To deal with the second problem, we have developed a novel system architecture. In particular, we have proposed new algorithms to perform multiple-query optimization for frequent pattern mining queries which involve multiple datasets. We have also designed a knowledgeable cache which can store the past query results from queries, and enable the use of these results to further optimize multiple queries. We have demonstrated a speedup of up to a factor of 9 on top of the optimized query plans for single query evaluation.

1.3.2 Mining Frequent Itemsets over Data Stream

Problem: Recently, a new data analysis model, streaming data, has received a lot of attention. In this model, the data arrives continuously and may not be stored onto the disks. Therefore, a mining algorithm can only scan the datasets once and get the mining results. The streaming data model can be looked as an extreme case of very large datasets. Several algorithms have been proposed to deal with this challenge. However, they either require out-of-core data structure, or potentially miss some frequent patterns [70, 113].

Our Contribution: We propose a new algorithm *StreamMining* with a parameter of support θ and a desired accuracy ϵ: In a single pass, StreamMining will find a superset of frequent itemsets with support θ, and each itemset in the superset will appear more than the frequency $(1-\epsilon)\theta$. If a second pass allowed, StreamMining will guarantee to find the exactly all frequent itemsets by eliminating the false positive.

StreamMining has been the first *in-core* algorithm, meaning computation can be performed without using disks, for frequent itemsets mining over data streams. In addition, the detailed evaluation using both synthetic and real datasets has shown that our one pass algorithm is very accurate in practice, and is very memory efficient.

1.3.3 Mining Frequent Topological Structures from Graph Datasets

Problem: Many objects, such as chemical compounds, proteins, web-logs, and XML can be represented by graphs. In this problem, we study mining frequent large-scale structures from graph datasets. Such frequent patterns are very important and useful in many real world applications, such as biology, social networks, and telecommunication.

Our Contribution: The main contribution of our work is a framework to mine frequent large-scale structures from graphs. In particular, we develop an efficient vertical mining

algorithm to mine such patterns, and propose a new approach to summarize and control the discovery of constrained patterns. We also study the scalability and quality of the proposed framework on several real and synthetic datasets, and demonstrate the use of the framework for discovering novel and meaningful motifs in membrane protein structures.

1.4 Organization of the thesis

The rest of thesis is organized as follows. In Chapter 2, we will introduce the problem of mining multiple datasets, and show the optimization techniques to evaluate a single mining task. In Chapter 3, we study the approach to optimize multiple such mining tasks together, and develop a knowledgeable cache to utilize the past mining results. Our new algorithm to mine frequent itemsets from streaming data will be discussed in Chapter 4. We will present our framework to mine frequent large-scale structures from graph datasets in Chapter 5. Finally, in Chapter 6, we will conclude the thesis and discuss the future work.

CHAPTER 2

A SYSTEMATIC APPROACH FOR OPTIMIZING COMPLEX MINING TASKS ON MULTIPLE DATABASES

2.1 Introduction

It has been well recognized that data mining is an interactive and iterative process, i.e., a data miner cannot expect to get interesting patterns and knowledge by a single execution of one algorithm. In order to support this process, one of the long-term goals of data mining research has been to build a *Knowledge Discovery and Data Mining System* (KDDMS) [24, 49, 58]. The vision is that such a system will provide an integrated and user-friendly environment for efficient execution of data mining tasks or queries. Along this line, much research has been conducted to provide database support for mining operations. This includes the work on query language extensions [47, 57, 74, 112] and implementing mining algorithms in a database system [20, 93]. Logic and algebra based methods have also been proposed to model the mining process [18, 37, 64]. The subfield of constraint association mining allows mining of *interesting* association rules by taking of a variety of constraint conditions as input [16, 68, 77, 98].

In the above research projects, the focus has typically been on mining a *single* dataset. However, in many situations, such as in a data warehouse, the user usually has a view

of multiple datasets collected from different sources. In such scenarios, comparing the patterns from different datasets and understanding their relationships can be an extremely important part of the KDD process. This, however, requires support for complex queries on multiple datasets in a KDDMS.

Such support involves significant and new optimization challenges. Suppose a user needs to find patterns that frequent with a certain support in both A and B. While this can be answered by taking intersection of the results from both A and B, this is likely to be very expensive. Instead, we can compute patterns frequent in either of the two datasets, and then simply find which of these are frequent in the other dataset. However, this leads to two different evaluation plans, corresponding to using the dataset A and B, respectively, for the initial evaluation. The two evaluation plans can have different costs, depending upon the nature of the datasets A and B. Furthermore, as the number of datasets and the complexity of the query condition increases, the number of possible evaluation plans can also grow.

Thus, there is a need for techniques for enumerating different query plans and choosing the one with the least cost, similar to what have been developed for traditional database queries [19]. However, compared with query optimization in traditional databases, the problem we consider is quite different in the following ways. First, the basic operators in our algebra are *mining operators*, which are more complex than the relational algebra operations. Second, the search space of query plans can be very large in our case. Third, the cost associated with a given mining operator is very hard to estimate.

In this chapter, we start with a simple mechanism for specifying mining queries across multiple datasets. Then, by representing these queries through an algebra, and developing a set of transformation and optimization techniques, we establish an approach for optimizing these queries. Our work is specifically in the context of frequent pattern mining.

Algorithms for frequent pattern mining have formed the basis for a number of other mining problems, including association mining, correlations mining, and mining sequential and emerging patterns [48].

To summarize, this chapter makes the following contributions:

1. We present an SQL based mechanism and establish an algebra for querying frequent patterns across multiple datasets.

2. We introduce several new operators and develop a number of transformations on this algebra to enable aggressive optimizations.

3. We present several heuristic algorithms for finding efficient query plans.

4. We evaluate our query optimization techniques on both real and synthetic datasets, and demonstrate up to an order of magnitude performance gains as compared to the naive execution.

2.2 Motivating Examples

To further motivate and facilitate our study, we consider different scenarios and list many examples of the kind of queries our framework targets.

Mining the Data Warehouse for a Nation-wide Store: Consider a store that has three branches, in New Jersey, New York, and California, respectively. Each of them maintains a database with last one week's retail transactions. To understand how the geographical factors impact shopping patterns, queries of the following type are likely to be asked:

Q1: Find the itemsets that are frequent with support level 0.1% in *any* of the stores.

Q2: Find the itemsets that are frequent with support level 0.1% in *each* store.

Q3: Find the itemsets that are frequent with support level 0.05% in both the stores on east coast, but are very infrequent (support less than 0.01%) in the west coast store.

Finding Signature Itemsets for Network Intrusion: In a signature detection system, frequent itemsets can serve as the patterns to signal well-known attacks [79]. Suppose a *tcpdump* dataset contains the TCP packet information of several different network intrusion attacks. We can split the available data into several datasets, with one dataset corresponding to each intrusion type and a *normal* dataset corresponding to the situation when no intrusion is occurring. Queries of the following type have been used to capture the signature patterns [79]:

Q4: Find the itemsets that are frequent with a support level 80% in *either* of the intrusion datasets, but are very infrequent (support less than 50%) in the normal dataset.

Q5: Find the itemsets that are frequent with a support level 70% in *each* of the intrusion datasets, but are very infrequent (support less than 60%) in the normal dataset.

Q6: Find the itemsets that are frequent with a support level 85% in *one* of the intrusion datasets, but are very infrequent (support less than 65%) in all other datasets.

Besides frequent items, mining other frequent patterns, including subgraphs, subtrees, or topological patterns, is also very useful in many domains. Examples of domain where such patterns have been shown to be useful are study of chemical compounds, protein tertiary structure analysis, motifs discovery, among others [73, 53]. Again, comparing patterns across multiple datasets is important in each of these areas. For example, a biologist may be interested in finding sequences that are frequent in a human gene, but infrequent in chicken gene, and/or, the sequences are frequent in both the species.

Dataset A_1		Dataset A_2	
TransID	Items	TransID	Items
1	{A,B,E}	1	{A B D E}
2	{B,D}	2	{B C E}
3	{A, B, E}	3	{A, B, E }
4	{A,C, D }	4	{A, B, C}
5	{B,C,D}	5	{A, C }
6	{A,C ,D}	6	{C, D}
7	{A, B }		
8	{A, B, C, D, E }		

Table 2.1: Datasets A_1 and A_2

In order to simplify our discussion, we will focus on frequent itemset mining tasks in the rest of this chapter. In Section 2.8, we discuss how our method can be generalized to frequent structure mining.

2.3 SQL Extensions and Algebra for Mining Across Multiple Datasets

In this section, we first introduce an SQL based mechanism for quering frequent itemsets across multiple datasets (Subsection 2.3.1). Then, we establish an algebra for expressing the information required to answer such a mining query (Subsection 2.3.2). Finally, we discuss the mapping from a mining query in its SQL format to an algebra expression (Subsection 2.3.3).

2.3.1 SQL Extensions

Let $\{A_1, A_2, \cdots, A_m\}$ be the set of datasets we are targeting. Each of these comprises transactions, which are set of items. The datasets are also *homogeneous*, i.e, an item has an identical name across different datasets. Let $Item$ be the set of all the possible items in all datasets.

I	A_1	A_2
{A}	6/8	4/6
{B}	6/8	4/6
{C}	4/8	4/6
{D}	6/8	2/6
{E}	3/8	3/6
{A,B}	4/8	3/6
{A,C}	3/8	2/6
:	:	:
{A,B,C,D,E }	1/8	0

Table 2.2: F Table for the Datasets A_1 and A_2

We define the following schema,

$$Frequency(I, A_1, A_2, \cdots, A_m)$$

For a table F of this schema, the column with attribute $F.I$ stores all possible itemsets, i.e, the power-set of $Item$. The column with attribute $F.A_i$ stores the frequency of the itemsets in the dataset A_i. For example, consider two transaction datasets A_1 and A_2, as shown in Table 2.1. The set of distinct items in the two datasets, $Item$, is $\{A, B, C, D, E\}$. Table 2.2 contains a portion of the F table for the datasets A_1 and A_2.

Such a table can only be used as a *virtual* table or a logical view, as the total number of itemsets is likely to be too large for the table F to be materialized and stored. In our SQL extensions, a frequent itemset mining task on multiple datasets is expressed as an SQL query to partially materialize this table. The following query $\mathcal{Q}_1$ is an example.

Query $\mathcal{Q}_1$:

SELECT $F.I, F.A, F.B, F.C, F.D$

FROM *Frequency*(I, A, B, C, D) F

WHERE $(F.A \geq 0.1$ AND $F.B \geq 0.1$ AND $F.D \geq 0.05)$

OR ($F.C \geq 0.1$ AND $F.D \geq 0.1$ AND

($F.A \geq 0.05$ OR $F.B \geq 0.05$))

Here, we want to find the itemsets that are either frequent with support level 0.1 in both A and B, and frequent in D with support level 0.05, or frequent (with support level 0.1) in both C and D, and also frequent in either A or B (with support level 0.05).

2.3.2 Basic Algebra for Queries

Our algebra contains only one mining operator SF and two operations, intersection ($\sqcap$) and union ($\sqcup$). We begin with the definition of a *view* of the F table. A view of the F table is a table with a subset of the rows and columns of the F table, which always contains the column of the attributes I, and the exact frequency of an itemset can be replaced by a *Null* value (denoted as $\circ$).

Given this, we define the basic mining operator SF to generate above simple views (containing only two columns) of F table.

The frequent itemset mining operator $SF(A_j, \alpha)$ computes the frequent itemset from a single dataset A_j with support level α. It returns a two-column table, where the first column contains itemsets in A_j which have the support level α, and the second column contains their corresponding frequency in the dataset A_j.

Table 2.3 shows the results of SF operator on the datasets A_1 and A_2 (shown in Table 2.1) with support level 0.5 and 0.4, respectively.

Next, we define the two operations that can combine the views of the F table. Let F_1 and F_2 be two views of the F table. Let F_1^I and F_2^I be the projections of F_1 and F_2 on the attribute I.

Intersection ($F_1 \sqcap F_2$) returns a table whose first column contains the itemsets appearing in the first columns of *both* F_1 and F_2, and other columns contain frequency information for these itemsets in the datasets appearing in F_1 and F_2. Formally, $F_1 \sqcap F_2$ is defined as

$$(F_1^I \cap F_2^I) \bowtie_I F_1 \bowtie_I F_2$$

Note that $\bowtie$ is the standard database join operation (over the attribute I), with one important difference. Any column that is common between F_1 and F_2 is *merged*. In merging the columns, an actual count is preferred over a $\circ$ (Null) value.

Union ($F_1 \sqcup F_2$) returns a table whose first column contains the itemsets appearing in the first columns of *either* F_1 or F_2, and other columns contain the frequency of these itemsets in the datasets appearing in F_1 or F_2. Formally, $F_1 \sqcup F_2$ is defined as

$$(F_1^I \cup F_2^I) \overset{O}{\bowtie}_I F_1 \overset{O}{\bowtie}_I F_2$$

Note that we take an *outerjoin* [102]. Null is inserted for entries for which values are not available from either F_1 or F_2.

Note that the results of the two operations are still views of the F table. Table 2.4 provides examples for each of these two operations.

Based upon the definitions of the above operations, we can easily prove the following:

Lemma 1 *The operations, intersection* $(\sqcap)$ *and union* $(\sqcup)$*, satisfy the associative, commutative, and distributive properties.*

$SF(A_1, 0.5)$		$SF(A_2, 0.4)$		$\overline{SF(A_2, 0.4)}$		$P(X, A_1)$	
I	A_1	I	A_2	I	A_2	I	A_1
{A}	6/8	{A}	4/6	{D}	∘	{A }	6/8
{B}	6/8	{B}	4/6	{A,C}	∘	{B}	6/8
{C}	4/8	{C}	4/6	{A,D}	∘	{E}	3/8
{D}	6/8	{E}	3/6	{A,E}		{A,B}	4/8
{A,B}	4/8	{A,B}	3/6	:	:	{A,E}	3/8
{C,D}	4/8			{A,B,C, D,E }	∘	{B,E}	3/8
						{A,B,E}	3/8

Table 2.3: Basic Operators on F Table

$SF(A_1, 0.5) \sqcap SF(A_2, 0.4)$			$SF(A_1, 0.5) \sqcup SF(A_2, 0.4)$		
I	A_1	A_2	I	A_1	A_2
{A}	6/8	4/6	{A}	6/8	4/6
{B}	6/8	4/6	{B}	6/8	4/6
{C}	4/8	4/6	{C}	4/8	4/6
{A,B}	4/8	3/6	{D}	6/8	∘
			{E}	∘	3/6
			{A,B}	4/8	3/6
			{C,D}	4/8	∘

Table 2.4: Intersection and Union Operation

2.3.3 Mapping from SQL Queries to Basic Algebra

In the following, we discuss how a restricted class of queries can be directly modeled using the above operator and operations. This class of queries involves constraint conditions (the WHERE clauses) which do not contain any *negative* predicates, i.e., a condition which states that support in a certain dataset is below a specified threshold. We call this class of queries *positive queries*. In Section 2.6, we will discuss how a more general class of mining queries, which could involve *negative* conditions as well, can be expressed by this algebra as well.

Let us consider a positive query Q with the condition C. Clearly, the condition C can be restated in the DNF form, with conjunctive clauses $C_1, \ldots, C_k$. Formally,

$$C \ = \ C_1 \vee \ldots \vee C_k, \ \ C_i \ = \ p_{i1} \wedge \ldots \wedge p_{im}, \ 1 \leq i \leq k$$

where, $p_{ij} = F.A_{ij} \geq \alpha$ is a *positive* predicate, i.e., a condition which states that support in a certain dataset (A_{ij}) is greater than or equal to a specified threshold (α). The corresponding *basic algebra expression* is as follows. We replace p_{ij} by the operator $SF(A_{ij}, \alpha)$. We can represent the query by

$$F_Q = F_{C_1} \sqcup \cdots \sqcup F_{C_k}$$

where, in each F_{C_i}, the corresponding SF operator is connected using intersection operations. Therefore, for query $\mathcal{Q}_1$, its corresponding basic algebra expression $F_{\mathcal{Q}_1}$ is as follows.

$$(SF(A, 0.1) \sqcap SF(B, 0.1)) \sqcap SF(D, 0.05) \Rightarrow F_1$$

$$\sqcup(SF(A, 0.05) \sqcap SF(C, 0.1) \sqcap SF(D, 0.1)) \Rightarrow F_2$$

$$\sqcup(SF(B, 0.0.5) \sqcap SF(C, 0.1) \sqcap SF(D, 0.1)) \Rightarrow F_3$$

2.4 Query Optimization Overview

This section gives an overview of the challenges in query optimization. The first important observation is that the costs of the mining operators, such as SF, are typically much higher than those of union and intersection operations. Therefore, we need to focus on mining operators in our optimization process.

Let us consider the naive evaluation of the basic algebra expression $F_{\mathcal{Q}_1}$ for the query $\mathcal{Q}_1$ stated in the previous section. We need to invoke the SF operator 7 times, including mining frequent itemsets on datasets A, B, and D with two different supports 0.1 and 0.05, and on dataset C with support 0.1. The important observation here is that in such a naive evaluation, a large fraction of the computation is either *repetitive* or *unnecessary*. By *repetitive* computation, we imply finding the frequency of an itemset on a dataset more than once, because of different mining operators. For example, the computation of $SF(A, 0.1)$ is repetitive. This is because $SF(A, 0.05)$ is also evaluated and $SF(A, 0.1) \subseteq SF(A, 0.05)$. By *unnecessary* computation, we imply finding the frequency of the itemsets which do not appear in the generated view of the basic algebra expression. For example, the computation of frequency for each itemset in the set $SF^I(A, 0.1) - F^I_{\mathcal{Q}_1}$ on the dataset A is unnecessary.

2.4.1 Challenges in Mining Query Optimization

In view of the above example, the main challenges in optimizing evaluation of a given query can be summarized as follows.

New Mining Operators: As discussed above, to reduce the cost of evaluating a basic algebra expression, we need to reduce *repetitive* and *unnecessary* computations. In particular,

in the basic algebra, there is no easy way to remove *unnecessary* computations. Therefore, new mining operators are needed to address this problem. Particularly, we will use *constraint* and *group* mining operators in our work.

Query Plan Enumeration: Assume we have new mining operators. Now, the problem is how to use them in an effective manner. For a given complicated mining query, a number of different sequences of mining operators can be used to evaluate this query. Clearly, if we can enumerate the different query plans, we can use a cost model to find the one with the least cost. However, enumerating query plans for a given mining query is a very different problem than the one for traditional database queries.

Algorithms for Finding Optimized Query Plans: The challenge of finding optimized query plans is two-folds. On one hand, the search space of possible query plans can be very large for a complicated query. Therefore, even if the costs associated with the different query plans are known, we still need efficient algorithms to find the one with the least cost. At the same time, the cost of a query plan is very hard to estimate. Though this cost can be stated as the sum of the costs for each individual mining operator in the plan, the cost of a mining operator can depend on the mining operators preceding it. Therefore, precise cost models are almost impossible, and we find to find good heuristics.

In the following two subsections, we introduce the tools we use to address the problem of repetitive and unnecessary computations. These are, the new mining operators, and using *containing* relations.

2.4.2 New Operators

To reduce the unnecessary computation, two new operators, CF and GF, are introduced.

1. Frequent itemset mining operator with constraints $CF(A_j, \alpha, X)$ finds the itemsets that are frequent in the dataset A_j with support α and also appears in the set X. X is a set of itemsets that satisfies the *down-closure property*, i.e., if an itemset is frequent, then all its subsets are also frequent. This operator also reports the frequency of these itemsets in A_j. Formally, $CF(A_j, \alpha, X)$ computes the following view of the F table:

$$X \sqcap SF(A_j, \alpha)$$

The typical scenario where this operator helps remove unnecessary computation is as follows. Suppose the frequent itemset operator intersects with some view of the F table, such that the projection of this view on the attribute I is X. This operator *pushes* the set X into the frequent itemset generation procedure, i.e., X serves as the search space for the frequent itemset generation. Thus, the unnecessary computation for the itemsets that are not in X can be saved.

2. Group frequent itemset mining operator $GF(Y)$, where $Y = \{< A_1, \alpha_1 >, \cdots, < A_u, \alpha_u >\}$, finds the itemsets that are frequent in each dataset A_i with support α_i, and reports their frequency in each of these datasets. Formally, $GF(Y)$ computes the following view of the F table:

$$SF(A_1, \alpha_1) \sqcap \cdots \sqcap SF(A_u, \alpha_u)$$

The idea behind this operator is as follows. The frequency count for all datasets in Y is carried out in parallel. Thus, all supersets of an itemset that is determined to be infrequent in any of the datasets is *pruned*.

We use the following example to illustrate the use of these operators. Consider the following view of the F table (we need to find the itemsets with support 0.1 that are frequent

in A and are also either frequent in B or in C),

$$(SF(A, 0.1) \sqcap SF(B, 0.1)) \sqcup (SF(A, 0.1) \sqcap SF(C, 0.1))$$

Applying the CF operator, we can evaluate $SF(A, 0.1)$ first, and then intersect it with

$$(CF(B, 0.1, SF^I(A, 0.1)) \sqcup CF(C, 0.1, SF^I(A, 0.1)))$$

Here, we first find the frequent itemsets in A, and then among them, find those are either frequent in B or in C. Compared with the naive method where we find the frequent itemsets on each dataset and them perform intersection, the cost of finding frequent itemsets in B and C but infrequent in A is saved. Formally, this evaluation reduces the *unnecessary* costs of $SF^I(B, 0.1) - (SF(A, 0.1) \sqcap SF(B, 0.1))^I$ on the dataset B and $SF^I(C, 0.1) - (SF(A, 0.1) \sqcap SF(C, 0.1))^I$ on the dataset C. However, the cost of finding itemsets which are frequent in A but infrequent in either B and C ($(SF^I(A, 0.1) - (SF(B, 0.1) \sqcup (SF(C, 0.1))^I)$ is still unnecessary.

Applying the GF operator, this view can be evaluated as

$$GF(\{< A, 0.1 >, < B, 0.1 >\}) \sqcup GF(\{< A, 0.1 >, < C, 0.1 >\})$$

Here, we first find the itemsets which are frequent in both A and B, and then we find the itemsets which are frequent in both A and C. No unnecessary computation is involved now. However, the itemsets that are frequent in A, but also frequent in both B and C, are generated twice. Specifically, the computation of the itemsets in the set $(SF(A, 0.1) \sqcap SF(B, 0.1) \sqcap SF(C, 0.1))^I$ for dataset A has now become *repetitive*.

2.4.3 Containing Relation

An important tool to remove repetitive computation is based on the *containing relation* for the sets of frequent itemsets. The containing relation is as follows: $\beta \leq \alpha$, $SF(A_j, \beta)$

contains all the frequent itemsets in $SF(A_j, \alpha)$. Therefore, if the first one is available, invocation of the second can be avoided. Instead, a relatively inexpensive selection operator, denoted as σ, can be applied. Formally, for $\beta \leq \alpha$, we have,

$$SF(A_j, \alpha) = \sigma_{A_j \geq \alpha}(SF(A_j, \beta))$$

This containing relations can be also extended to the our two new operators, CF and GF.

Let us revisit the query $\mathcal{Q}_1$. In view of this relation, at most one invocation of the mining operator SF on each dataset is required. Thus, we only need four invocations of the SF operator, i.e., mining frequent itemsets on datasets A, B, and D with support 0.05, and on dataset C with support 0.1. This method, which removes all repetitive computation due to SF operator, but does not use CF and GF operators, is referred to as the *Optimization RR* (Remove Repetition). It should be noted that though the repetitive computation due to SF operator is removed here, much unnecessary computation is still involved.

2.4.4 Overview of Query Plan Generation

The discussion in the previous two subsections focused on removing unnecessary and repetitive computations, respectively. Each was considered independently. In generating an efficient plan for evaluating a query, it is important to consider both. As our example has shown, removing unnecessary computation can introduce repetitive computation, and vice-versa. Clearly, this makes query optimization a challenging task. In many cases, removing both unnecessary and repetitive computation for a query evaluation is not possible.

In the next two sections, we present a systematic approach for finding efficient query plans. Our approach includes the following three key elements:

M table Formulation: The basic algebra expression of a given query is encoded into an M table. In the M table, each column represents a conjunctive-clause in the condition,

and each row represents a dataset. Each cell in the table contains a predicate that appears in the condition and needs to be evaluated. Further, the query evaluation process can be depicted as a coloring scheme of the M table. Therefore, M table provides an intuitive way to enumerate possible query plans.

Query Plan Generation: The efficient query plans are generated with the help of the coloring scheme of the M table. We partition the query plan into two phases. The first phase contains the mining operators that are *independent* of the mining results generated from the mining operators evaluated before it. The second phase contains the mining operators that are *dependent* on these results. Such partition allows us to derive good heuristics to reduce the evaluation costs.

Transformations: Consider a query containing the *negative* predicates. To optimize such queries, we will use a set of transformations. To express such queries in our algebra, we introduce two additional mining operators. Then, we will show how these mining operators can be removed, and therefore, the basic algebra expression is constructed. Among the above three issues, we discuss the first two in Section 2.5, and the last in Section 2.6.

2.5 Query Plan Generation

2.5.1 A Unified Query Evaluation Scheme

This subsection describes a general representation, the M-table, for query evaluation based on the basic algebra expression of a given query. As we will show, such a scheme provides an intuitive way to describe the possible query plans.

Definition 1 *Assume the basic algebra expression of a query Q is*

$$F_Q = F_1 \sqcup \cdots \sqcup F_t$$

where, each F_i involves intersection among one or more SF operators. Let m be the number of distinct datasets that appear in F. Then, the M-table for the basic algebra expression of this query is a table with m rows and t columns, where the row i corresponds to the dataset A_i, and the column j corresponds to the clause F_j. If $SF(A_i, \alpha)$ appears in F_j, the cell at j-th column and i-th row will have α, i.e., $M_{i,j} = \alpha$. Otherwise, the cell $M_{i,j}$ is empty.

As an example, the M table for the query $\mathcal{Q}_1$ has 4 rows and 3 columns and is shown in Table 2.1.

```
SELECT F.I, F.A, F.B, F.C, F.D
FROM Frequency(I, A, B, C, D) F
WHERE (F.A ≥ 0.1 AND F.B ≥ 0.1)
  OR (F.C ≥ 0.1 AND F.D ≥ 0.1 AND
  (F.A ≥ 0.2 OR F.B ≥ 0.2))
```

(a) SQL query for query $\mathcal{Q}_1$

	F_1	F_2	F_3
A	0.1	0.2	
B	0.1		0.2
C		0.1	0.1
D		0.1	0.1

(b) M Table for the query $\mathcal{Q}_1$

$$(SF(A, 0.1) \sqcap SF(B, 0.1)) \Rightarrow F_1$$
$$\sqcup(SF(A, 0.2) \sqcap SF(C, 0.1) \sqcap SF(D, 0.1)) \Rightarrow F_2$$
$$\sqcup(SF(B, 0.2) \sqcap SF(C, 0.1) \sqcap SF(D, 0.1)) \Rightarrow F_3$$

(c) Necessary Information for the query $\mathcal{Q}_1$

Figure 2.1: Query $\mathcal{Q}_1$

	F_1	F_2	F_3	F_4	F_5
A	0.1	0.1		0.05	
B	0.1	0.1			0.05
C	0	0	0.1	0.1	0.1
D		0.05	0.1	0.1	0.1

Table 2.5: M Table for the query $\mathcal{Q}$

	F_1	F_2	F_3	F_4	F_5
A	0.1	0.1		0.05	
B	0.1	0.1			0.05
C			0.1	0.1	0.1
D		0.05	0.1	0.1	0.1

Table 2.6: Colored M Table for the query $\mathcal{Q}$

Note that the mapping between the M tables and the basic algebra expressions is *one-to-one*. It is important to note that the M table representation can be used to answer more complex queries, which could have negative predicates as well. This is discussed in Subsection 2.6.2.

Now, we focus on query plan generation using the M-table and the operators we have defined so far. To facilitate our discussion, we will use the M table in Table 2.5 as our running example. One of the most important features of M table is that it can capture the evaluation process for a query by using a simple coloring scheme. Initially, all the cells are black. The operators, SF, CF, and GF, can color a number of non-empty cells red. The query evaluation process is complete when all non-empty cells are colored red.

As a running example, consider applying $SF(A, 0.05)$, $CF(B, 0.1, SF^I(A, 0.1))$,and $GF(\{C, 0.1\}, \{D, 0.1\})$ consecutively on an initially black-colored table M of the query $\mathcal{Q}$. Table 3.3 shows the resulting colored table. We now define how each operator colors the table.

Frequent mining operator $SF(A_i, \alpha)$**:** An invocation of the frequent mining operator on the dataset A_i, with support α, will turn each non-empty cell at row i who is greater than or equal to α red. In our example, the first operator, $SF(A, 0.05)$, will turn the cells $M_{1,1}$, $M_{1,2}$, and $M_{1,4}$ red.

Frequent mining operator with constraint $CF(A_i, \alpha, X)$**:** The coloring impacted by this operator is dependent on the current coloring of the table M. Let X be the set of frequent itemsets defined by all the red cells, and let S be the set of columns where these red cells appear. Then, by applying this operator on dataset A_i with support α, all cells on row i whose column is in the set S, and whose value is greater than or equal to α, will turn red. In our running example, the third operator

$$CF(B, 0.1, SF^I(A, 0.1))$$

picks the red cells $M_{1,1}$ and $M_{1,2}$ by the parameter

$$X = SF^I(A, 0.1)$$

The set S includes the first two columns. Therefore, this operator turns the cells $M_{2,1}$ and $M_{2,2}$ red.

Group frequent itemset mining operator $GF(Y)$**:** The parameter $Y = \{< A_1, \alpha_1 > , \cdots, < A_u, \alpha_u >\}$, specifies the support level α_i for the dataset A_i. Let the dataset $A'_k, 1 \leq k \leq u$ correspond to the row ik. Let S_i be the set of columns whose cells at row ik are less than or equal to the correspond α_i. Let $S = S_{i1,j1} \cap \cdots \cap S_{iu,ju}$. Invoking this

operator will turn every cell in the row defined by $\{i1, \cdots, iu\} \times S$ red. In our example, the operator $GF(\{C, 0.1\}, \{D, 0.1\})$, will turn the cells the right-bottom rectangle defined by $\{3, 4\} \times \{3, 4, 5\}$ red.

By the above formulation, the query evaluation problem has been converted into the problem of coloring the table M. The possible query plans can be intuitively captured in this framework. Note that different operators can be used, and in different order, to color the entire table red. There are different costs associated with each of them. The next subsection addresses the problem of finding efficient query evaluation plans.

2.5.2 New Query Plans

For a given M table with m rows and t columns, the total number of possilbe query plans using only SF and CF operators can be up to $(\sum_{i=1}^{i=m} j_i)! \times 2^{\sum_{i=1}^{i=m} j_i}$, where j_i is the number of different support levels in the row i. Clearly, using the GF operator will make this number even higher. Furthermore, another difficulty in this optimization process is that it is very hard to associate cost functions for the three operators. We are not aware any research on predicting the running time for a specific mining algorithm on a given dataset. The costs of CF operator depends on the mining results from the operators preceeding it. Though this is somewhat similar to the *Join* optimization problem in the traditional databases [9], the cost from such a mining operator is even harder to estimate.

To deal with these challenges, we use a set of heuristics and greedy algorithms to help find efficient query plans. Specifically, a basic idea of our approach is to paritition the query plan into two phases. The first phase contains only the mining operators that are *independent* of the mining results generated from the mining operators evaluated before it. The second phase contains the mining operators that are *dependent* on these results. In

other words, only SF and GF can be used in the first phase, and CF can be used in the second phase. Such partition allows us to derive good heuristics to reduce the evaluation costs.

In the following, we first present two algorithms that are based upon the use of the SF and CF operators. Then, we describe another algorithm that further exploits the GF operator.

Using Constraint Based Operator

The constraint based mining operator $CF(A_j, \alpha, X)$ helps reduce the computational cost as follows. At any stage p, suppose that we need to color the cell $M_{i,j}$. As long as another red cell is available in the same column, CF operator can be used.

The algorithms we present here are based upon aggressively using the CF operator. The goals of each phase in a query plan is as follows. In the *first* phase, we use the $SF(A_j, \alpha)$ operators so that each column has at least one red cell. In the *second* phase, we use the $CF(A_j, \alpha, X)$ operators to compute all other non-empty cells in the table.

Approach for Phase One: To understand the complexity of optimizing the cost for this phase, let us assume that we know the cost for the operator $SF(A_j, \alpha)$. Our goal is to find the set of operations which has the least cost for coloring all columns of the table. This problem can be generalized and formulated as follows. For a set $S = \{S_1, \cdots, S_n\}$, $S_1 \cup \cdots \cup S_n = \{1, \cdots, m\}$, where each set S_i has a cost function and corresponds to a set of columns whose cells can be turned red by a SF mining operator. we need to find the a subset of S who can cover $\{1, \cdots, t\}$ with the least cost. This is a generalized *set-covering problem*, and is *NP-hard* [23].

Note, in our case, each row only needs at most one invocation of the SF operator, due to the *containing relation*. Clearly, the search space in this phase is much smaller than the entire search space for a query plan. Therfore, we can enumerate the coloring schemes and find the one with the minimal cost in $O(j_1 \times \cdots \times j_m) = O(t^m)$ time complexity. Here, m and t are the number of rows and columns respectively in table M, and j_i is the number of different support levels in the row i. In practice, the above enumeration can be done without a very high cost.

However, the problem still is that precise cost functions are not available. The heuristic approach we use is based on the observation that no *repetitive* computation due to the SF operator is involved in the phase one. So, we can solely focus on reducing the *unnecessary* computations. A natural heuristic for minimizing unnecessary computation is through the support level. For a single dataset, higher support level for the SF operator implies lower unnecessary computation. We use this in our implementation.

Input: table M after phase-one coloring
Algorithm 1
Find datasets whose corresponding rows has black cells;
For each row, find the lowest support level among black cells;
On each row, we invoke the CF operator with the lowest support level. Across the rows, this operator is invoked in the decreasing order of support level used for the CF operator.

Algorithm 2
Remove all the red cells from each chain set $S_{i,r}$;
Find the non-empty chain set with the highest support and invoke the CF operator to color the set;
Remove all new red cells from the chain set;
Repeat the above steps until all cells are colored.

Figure 2.2: Algorithms for Phase Two

Input: table M without coloring
Algorithm 3
Build a collection of candidate sets by running the enumeration algorithm for $SF(A_j, \alpha)$ operator;
For the candidate set S, let $SF(A_j, \alpha) \in S$
If there exists another mining operator $SF(A_k, \alpha')$ in S colors same columns as $SF(A_j, \alpha)$, transform $SF(A_j, \alpha)$ into $GF(\{< A_j, \alpha >, < A_k, \alpha' >\})$.
Repeat the above step to see if any more set can be aggregated into a GF operation;
Select a set from these transformed candidate sets based on some heuristic, e.g., the average size of the parameter set Y for the GF operation.

Figure 2.3: Using GF operator for Phase One

Approach for Phase Two: We can use either of the two greedy algorithms, *Algorithm 1* and *Algorithm 2*, which are listed in the Figure 2.2. The first algorithm tries to reduce the *repetitive* computation by invoking CF operator for each dataset at most once. Therefore, frequency of any itemset will be counted at most two times for a dataset: one from the SF operator in the phase one and second from the CF operator in the phase two. However, much unnecessary computation is involved since CF operator always picks the lowest support level for each dataset. The second algorithm targets the *unnecessary* computation, since for each support level, CF operator will use the smallest possible set X to constraint the itemset generation. However, much repetitive computation can be generated, since an itemset can be computed several times for a dataset.

Let us consider the query $\mathcal{Q}$. Combining phase one and phase two, the first algorithm gives the following query plan.

$Phase1:$ $SF(A, 0.1), SF(C, 0.1);$

$$Phase2: \quad CF(A, 0.05, SF(C, 0.1)^I);$$
$$CF(B, 0.05, (SF(A, 0.1) \sqcup SF(C, 0.1))^I);$$
$$CF(D, 0.05, ((SF(A, 0.1) \sqcap SF(B, 0.1)) \sqcup SF(C, 0.1))^I)$$
$$CF(C, 0, (SF(A, 0.1) \sqcap SF(B, 0.1))^I);$$

The second algorithm gives the following query plan.

$$Phase1: \quad SF(A, 0.1), SF(C, 0.1);$$
$$Phase2: \quad CF(B, 0.1, SF(A, 0.1)^I);$$
$$CF(D, 0.1, SF(C, 0.1)^I);$$
$$CF(A, 0.05, (SF(C, 0.1) \sqcap SF(D, 0.1))^I);$$
$$CF(B, 0.05, (SF(C, 0.1) \sqcap SF(D, 0.1))^I);$$
$$CF(D, 0.05, (SF(A, 0.1) \sqcap SF(B, 0.1))^I);$$
$$CF(C, 0, ((SF(A, 0.1) \sqcap SF(B, 0.1))^I);$$

We can see that both query plans can reduce the costs by aggressively utilizing the available information and the CF operator.

Using the Group Operator

The group mining operator GF can help remove some unnecessary computation due to SF operator. In the above example, suppose that $SF(A, 0.1) \sqcap SF(B, 0.1)$ and $SF(C, 0.1) \sqcap SF(D, 0.1)$ are generated in phase one. In this way, each column is also covered, and the *unnecessary* computation of set $SF^I(A, 0.1) - (SF(A, 0.1) \sqcap SF(B, 0.1))^I$ on dataset A is also saved.

The use of GF operator only changes the phase one, i.e, our method for coloring at least cell in each column. Instead of finding $SF(A_j, \alpha)$ operations to cover each column, we now need to find GF operations to meet the same goal. *Algorithm 3*, described in Figure 2.3, uses the GF operator in a efficient way. It results in the following query plan for our example query:

$$
\begin{aligned}
Phase1: \quad & GF(\{<A, 0.1>, <B, 0.1>\}); \\
& GF(\{<C, 0.1>, <D, 0.1>\}); \\
Phase2: \quad & CF(A, 0.05, (SF(C, 0.1) \sqcap SF(D, 0.1))^I); \\
& CF(B, 0.05, (SF(C, 0.1) \sqcap SF(D, 0.1))^I); \\
& CF(D, 0.05, (SF(A, 0.1) \sqcap SF(B, 0.1))^I); \\
& CF(C, 0, ((SF(A, 0.1) \sqcap SF(B, 0.1))^I);
\end{aligned}
$$

2.6 Generalized Queries and Transformations

In this section, we describe how the approach presented in the previous two sections can be applied to a more general class of queries. Specifically, we consider two additional requirements for a mining query. The first is to allow *negative* predicates in the query. The second is to allow users to specify conditions related with the *Null* values in the materialized views. In Subsection 2.6.1, we introduce these two requirements, and the algebra extensions to capture these requirements. In Subsection 2.6.2, we describe how we can transform the extended algebra expression into the basic algebra expression, and thus use the M-table and the algorithms from the previous section for query optimization.

2.6.1 Generalized Queries

Admissible Queries: We initially define a class of queries we consider *admissible* queries. For a given query, we transform the constraints into the disjunctive normal form (DNF), $C = C_1 \vee C_2 \vee \cdots C_k$ where, C_i is a *conjunctive-clause*, i.e., it involves AND operation on one or more predicates.

Definition 2 *A query is considered* admissible *if each conjunctive-clause in the DNF format contains at least one* positive *predicate, i.e.,* $F.A_i \geq \alpha$.

For example, a query involving the following condition is not admissible.

$F.A_1 < 0.1$ `OR` `(`$F.A_2 \geq 0.2$ `AND` $F.A_3 < 0.05$`)`

This is because the first conjunctive-clause, $F.A_1 < 0.1$, contains only a negative predicate. The significance of the admissible condition is that we are able to transform such a query into a basic algebra expression (Subsection 2.6.2).

Counting Requirements: The views generated from a basic algebra expression can contain *Null* values. In some cases, a user may be interested in removing the Null values in the final query answers. We introduce a new notation, g, for this purpose. In the Select clause of original query, replacing A_i by $g(A_i)$ denotes that the null value needs to removed, i.e. the actual frequency information is required. For simplicity, we denote the set of datasets having g in the Select clause as $CSET$. We call this function g as counting requirement since this can directly map into a counting operator discussed below.

Algebra Extensions: The two additional operators to help map an admissible query with negative predicate and counting requirement are as follows.

The negative frequent itemset mining operator $\overline{SF(A_j, \alpha)}$ computes itemsets in A_j with support level less than α. Formally, assuming 2^{Item} to be the power-set of $Item$, and

$SF^I(A_j, \alpha)$ is the projection of $SF(A_j, \alpha)$ on the column of attributes I, we have

$$\overline{SF(A_j, \alpha)} = (2^{Item} \ - \ SF^I(A_j, \alpha)) \times \{\circ\}$$

The counting operator $P(X, A_j)$ counts the frequency for each itemset in the set X on dataset A_j. To simplify its evaluation, this operator is only defined on a set X that satisfies the *down-closure* property.

Mapping to Extended Algebra: Consider mapping a admissible query with negative predicates and/or counting requirements. For the DNF format of the query condition (SELECT clauses), we replace the negative predicates with their corresponding infrequent itemsets mining operator. Further, we map the datasets with g functions to the counting operator. Therefore, we can build the extended algebra expression F_C for a given query Q with the condition C. Let $R(Q)$ is the final answering set for Q.

$$R(Q) \ = \ F_C \ \bowtie_I P(\widehat{F_C^I}, A_{i1}) \bowtie_I \cdots \bowtie_I P(\widehat{F_C^I}, A_{it})$$

where, $CSET = \{A_{i1}, A_{i2}, \cdots, A_{it}\}$, and $\widehat{F_C^I}$ is the minimal extension of F_C^I which satisfies the down-closure property.

2.6.2 Transformations for Query Optimization

In the following, we introduce two transformations which can remove the the negative frequent itemsets operator $\overline{SF}$ and the counting operator P from $R(Q)$, and replace them by $F(A_j, \alpha)$ operators.

To facilitate our discussion, we use the following query, denoted by $\mathcal{Q}$, as a running example.

SELECT $F.I, F.A, F.B, g(F.C), F.D$

FROM *Frequency*(I, A, B, C, D) F

WHERE ($F.A \geq 0.1$ AND $F.B \geq 0.1$ AND

NOT($F.C \geq 0.05$ OR $F.D \geq 0.05$))

OR ($F.C \geq 0.1$ AND $F.D \geq 0.1$ AND

NOT($F.A \geq 0.05$ OR $F.B \geq 0.05$))

The query involves finding the itemsets which are frequent with support level 0.1 in both the datasets A and B, but infrequent (support less than 0.05) in the datasets C and D, or vise versa. The DNF form of the condition $\mathcal{C}$ is:

$$(A \geq 0.1 \wedge B \geq 0.1 \wedge C < 0.05 \wedge D < 0.05)$$

$$\vee (C \geq 0.1 \wedge D \geq 0.1 \wedge A < 0.05 \wedge B < 0.05)$$

$F_{\mathcal{C}}$ can be expressed as:

$$(SF(A, 0.1) \sqcap SF(B, 0.1) \sqcap \overline{SF(C, 0.05)} \sqcap \overline{SF(D, 0.05)})$$

$$\sqcup (SF(C, 0.1) \sqcap SF(D, 0.1) \sqcap \overline{SF(A, 0.05)} \sqcap \overline{SF(B, 0.05)})$$

The answering set of this query can be expressed as

$$R(\mathcal{Q}) = F_{\mathcal{C}} \bowtie_I P(\widehat{F^I_{\mathcal{C}}}, C)$$

Now, we introduce the two transformations to remove the counting operator and negative mining operator.

Transformation 1: (Removing Counting Operator) This transformation takes three steps. In the first step, for any dataset $A_j \in CSET$, which suggests that a counting operator P might be needed, we add the boolean clause $A_j \geq 0$ into every conjunctive-clause in the DNF format of condition C. Thus, we generate a new condition, denoted as C'. Clearly, in this new condition, two boolean clauses on the same dataset may appear in a single

conjunctive-clause. In the second step, we remove these redundant boolean clauses by the following rule. If the boolean clause besides the new one is positive, the new one is removed, and if the boolean clause besides the new one is negative, the negative boolean clause is removed. Finally, we construct $F_{C'}$ corresponding to condition C' after the second step, and apply the selection operator with condition C to get $R(Q)$. Formally,

$$R(Q) = \sigma_C(F_{C'})$$

Let us illustrate this transformation on our running example. The set $CSET$ includes only the dataset C. In the first step, the new condition $\mathcal{C}'$ is

$$(A \geq 0.1 \wedge B \geq 0.1 \wedge C < 0.05 \wedge D < 0.05 \wedge C \geq 0) \vee$$

$$(C \geq 0.1 \wedge D \geq 0.1 \wedge A < 0.05 \wedge B < 0.05 \wedge C \geq 0))$$

In the second step, the condition $\mathcal{C}'$ becomes:

$$(A \geq 0.1 \wedge B \geq 0.1 \wedge C \geq 0 \wedge D < 0.05) \vee$$

$$(C \geq 0.1 \wedge D \geq 0.1 \wedge A < 0.05 \wedge B < 0.05))$$

In the final step, we construct $F_{\mathcal{C}'}$,

$$F_{\mathcal{C}'} = (SF(A, 0.1) \sqcap SF(B, 0.1) \sqcap SF(C, 0) \sqcap \overline{SF(D, 0.05)}$$

$$\sqcup (SF(C, 0.1) \sqcap SF(D, 0.1) \sqcap \overline{SF(A, 0.05)} \sqcap \overline{SF(B, 0.05)})$$

The answering set $R(\mathcal{Q})$ becomes $\sigma_{\mathcal{C}}(F_{\mathcal{C}'})$.

Transformation 2: (Removing Negative Frequent Itemset Operator) This transformation is based upon the following Lemma.

Lemma 2 *Let C be any condition, and F_C is the set satisfying this condition, then we have*

$$F_C \sqcap \overline{SF(A_j, \alpha)} = \sigma_{C \wedge (A_j < \alpha)}(F_C \sqcup (F_C \sqcap SF(A_j, \alpha)))$$

Note that the $NULL(\circ)$ value is treated as 0. The detailed proof is omitted here, but the correctness of this lemma can be observed from the fact that

$$F_C \sqcap \overline{SF(A_j, \alpha)} \subseteq F_C \sqcup (F_C \sqcap SF(A_j, \alpha))$$

This lemma suggests that the negative frequent itemset operator can be removed by applying the *union*($\sqcap$), *intersection*($\sqcup$), and selection operator.

By applying Lemma 2, all the negative frequent itemset operator can be removed from $F_{C'}$. Let

$$F_{C_j} = SF(A_{i1}, \alpha_1) \sqcap \cdots SF(A_{iu}, \alpha_u) \sqcap$$
$$\overline{SF(A_{i(u+1)}, \alpha_{u+1})} \sqcap \cdots \sqcap \overline{SF(A_{is}, \alpha_s)}$$

We denote $F_{C_j^+}$ to contain only the sets of frequent itemsets for C_j, such as

$$F_{C_j^+} = SF(A_{i1}, \alpha_1) \sqcap \cdots \sqcap SF(A_{iu}, \alpha_u)$$

Therefore, we have the following equality:

$$F_{C_j} = \sigma_{C_j}(F_{C_j^+} \sqcup (F_{C_j^+} \sqcap SF(A_{i(u+1)}, \alpha_{u+1})) \sqcup$$
$$\cdots \sqcup (F_{C_j^+} \sqcap SF(A_{is}, \alpha_s)))$$

Further, we can see that for each F_{C_j}, the selection operator (σ) can be removed because of the outside selection operator. In sum, this transformation removes all the negative frequent itemset mining operator, such as $\overline{SF(A_j, \alpha)}$, in $F_{C'}$ by applying this equality and removing the selection operator for each conjunctive clause C_j.

After these two transformations, the entire computation cost to evaluate the query Q has been shifted to compute $F_{C'}$. To simplify the discussion, we treat computing $F_{C'}$ as an

instance of this generalized problem of evaluating expression F_Q, where, $F_Q = F_1 \sqcup \cdots \sqcup F_t$, and,

$$F_j = SF(A_{j1}, \alpha_{j1}) \sqcap \cdots \sqcap SF(A_{jh}, \alpha_{jh})$$

Therefore, in our example, we have

$$\begin{aligned} F_{\mathcal{Q}} = F_{\mathcal{C}'} = (SF(A, 0.1) \sqcap SF(B, 0.1) \sqcap SF(C, 0)) &\Rightarrow F_1 \\ \sqcup (SF(A, 0.1) \sqcap SF(B, 0.1) \sqcap SF(C, 0) \sqcap SF(D, 0.05)) &\Rightarrow F_2 \\ \sqcup (SF(C, 0.1) \sqcap SF(D, 0.1)) &\Rightarrow F_3 \\ \sqcup (SF(C, 0.1) \sqcap SF(D, 0.1) \sqcap SF(A, 0.05)) &\Rightarrow F_4 \\ \sqcup (SF(C, 0.1) \sqcap SF(D, 0.1) \sqcap SF(B, 0.05)) &\Rightarrow F_5 \end{aligned}$$

Clearly, F_Q uses only the SF operator and two operations defined in the basic algebra. For a given query Q, the expression using only the basic algebra and generated through the above two transformations is the basic algebra expression of Q. Finally, we can see the M table corresponding to $F_{\mathcal{Q}}$ is the table (Table 2.5) used in Section 2.5.

2.7 Experimental Evaluation

This section reports a series of experiments we conducted to demonstrate the efficacy of the optimization and transformation techniques we have developed. Particularly, we were interested in the following questions:

1. What are the performance gains from the use of new mining operators, CF and GF, and what are the key factors impacting the level of gain.

	Query Conditions
Q_1	$A \geq \alpha_1 \wedge B \geq \alpha_2$
Q_2	$(A \geq \alpha_1 \vee B \geq \alpha_2) \wedge C < \beta_1$
Q_3	$(A \geq \alpha_1 \wedge B < \beta_1) \vee (B \geq \alpha_2 \wedge A < \beta_2)$
Q_4	$(A \geq \alpha_1 \wedge B \geq \alpha_2 \wedge C < \beta_1 \wedge D < \beta_1) \vee$ $(C \geq \alpha_1 \wedge D \geq \alpha_2 \wedge A < \beta_1 \wedge B < \beta_2)$
Q_5	$A \geq \alpha_1 \wedge B \geq \alpha_1 \wedge C \geq \alpha_1 \wedge D < \beta_1$
Q_6	$(A \geq \alpha_1 \wedge B < \beta_1 \wedge C < \alpha_2 \wedge D < \beta_2) \vee$ $(B \geq \alpha_3 \wedge A < \beta_3 \wedge C < \alpha_4 \wedge D < \beta_4) \vee$ $(C \geq \alpha_5 \wedge A < \beta_5 \wedge B < \alpha_6 \wedge D < \beta_6) \vee$

Table 2.7: Test Query Templates for Our Experiments

2. Compared with the naive evaluation method, what performance gains are obtained from the of different optimizations, and new query plans generated using the three algorithms we have presented.

Initially, we briefly describe how the three new operators we introduced were implemented.

2.7.1 Implementation of Operators

The operators used in our query evaluation are the *frequent mining operator*, the *counting operator*, the *frequent itemset with constraints operator*, and the *group frequent itemset operator*. For our experimental study, Borgelt's implementation of the well-known Apriori algorithm [14] is used as the frequent mining operator. The other three operators were derived from it as follows:

Counting operator $P(X, A_j)$: Initially, the set of itemsets X is organized as a prefix tree, where each node corresponds to an itemset. Then, a single pass on the dataset A_j is taken to project each transaction onto the prefix tree, using a depth-first traversal.

Query	Naive	ORR	CF-1	CF-2	GF-1
$Q_1(60\%, 40\%)$	397		168		158
$Q_2(60\%, 40\%)$	626	352	158		
$Q_3(60\%, 40\%)$	914	619	236	386	277
$Q_1(50\%, 35\%)$	1024		279		265
$Q_2(50\%, 35\%)$	1381	687	265		
$Q_3(50\%, 35\%)$	2206	1558	394	484	471

Table 2.8: Performance (in seconds) on IPUMS datasets

Frequent itemset mining operator with constraints: $CF(A_j, \alpha, X)$**:** Initially, the set of itemsets X is put into a hash table. The processing of CF is similar to the frequent itemset mining operator, with one exception in the candidate generation stage. While placing an itemset in the candidate set, not only all its subsets need to be frequent, but the itemset needs to be in the hash table as well.

Group frequent itemset mining operator $GF(Y)$**:** The parameter $Y = \{< A_1, \alpha_1 >, \cdots, < A_u, \alpha_u >\}$, specifies the support level α_i for the dataset A_i. There are three differences between the implementation of this operator and the implementation of the common frequent mining operator. First, each node representing an itemset in the prefix tree has one count field for each dataset in Y. Second, the counts for each dataset are updated independently. Finally, in the candidate generation stage, an itemset is treated as a candidate set if all of its subsets are frequent in every dataset in Y.

2.7.2 Datasets

Our experiments were conducted using three groups of data, each of them comprising four different datasets.

Query	Naive	ORR	CF-1	GF-1
$Q_4(85\%, 55\%)$	1229	1085		
$Q_5(85\%, 55\%)$	1178	1032	301	218
$Q_6(85\%, 55\%)$	2502	1350	1304	
$Q_4(60\%, 40\%)$	1525	1361		
$Q_5(60\%, 40\%)$	1313	1152	491	216
$Q_6(60\%, 40\%)$	2634	1392	1470	

Table 2.9: Performance (in seconds) on DARPA datasets

IPUMS: The first group of datasets is derived from the *IPUMS* 1990-5% census micro-data, which provides information about individuals and households [1]. The four datasets each comprises 50,000 records, corresponding to New York, New Jersey, California, and Washington states, respectively. Every record in the datasets has 57 attributes. After discretizing the numerical attributes, the datasets have a total of 2,886 distinct items.

DARPA's Intrusion Detection: The second group of datasets is derived from the first three weeks of *tcpdump* data from the DARPA data sets [79]. The three datasets include the data for three most frequently occurring intrusions, *Neptune*, *Smurf*, and *Satan*. The first two are Denial of Service attacks (DOS) and the last one is a type of Probe. Further, an additional dataset includes the data of the *normal* situation (i.e., without intrusion). Each transaction in the datasets has 40 attributes, corresponding to the fields in the TCP packets. After discretizing the numerical attributes, there are a total of 343 distinct itemsets. The neptune, smurf, satan, and normal datasets contain 107,201, 280,790, 1,589, and 97,277 records, respectively.

IBM's Quest: The third group of datasets represents the market basket scenario, and is derived from IBM Quest's synthetic datasets [5]. The first two datasets, dataset-1 and dataset-2, are generated from the $T20.I8.N2000$ dataset by some perturbation. Here, the

Query	Naive	ORR	CF-1	CF-2	GF-1
Q_1	3825		727		338
Q_2	7048	3384	1138		
Q_3	10369	7617	1344	1462	977
Q_4	2828	1395			
Q_5	2753	1324	693		283
Q_6	10105	7368	1815		

Table 2.10: Performance (in seconds) on QUEST datasets with query parameters $\alpha_1 = 0.3\%$ and $\alpha_2 = 0.1\%$

Query	Naive	ORR	CF-1	CF-2	GF-1
Q_1	5120		971		351
Q_2	9016	4379	1599		
Q_3	13285	9764	1743	1827	1042
Q_4	3823	2039			
Q_5	3662	1876	904		364
Q_6	13034	9394	2511		

Table 2.11: Performance (in seconds) on QUEST datasets with query parameters $\alpha_1 = 0.25\%$ and $\alpha_2 = 0.08\%$

number of items per transactions is 20, the average size of large itemsets is 8, and the number of distinct items is 2000. For perturbation, we randomly change a group of items to other items with some probability. The other two datasets, dataset-3 and dataset-4, are similarly generated from the $T20.I10.N2000$ dataset. There are a total of 1943 distinct items in the four datasets, and each of them contains 1,000,000 transactions.

2.7.3 Test Queries

Our experiments use six different queries, which are listed in the Table 3.8. The first three queries, Q_1, Q_2, and Q_3, are applicable on IPUMS datasets, and the New York, New

Jersey, California, and Washington datasets are labeled as the datasets A, B, C, and D, respectively. The other three queries, Q_4, Q_5, and Q_6, correspond to the queries in the motivating example on finding the signature itemsets for network intrusion, presented in Section 2.2. The neptune, smurf, satan, and normal datasets are labeled as the datasets A, B, C, and D, respectively. Further, in the Table 3.8, the $CSET$ is specified. Finally, each query requires two different support levels, α_1 and α_2. The evaluation using the IBM Quest dataset used all six queries.

In our experiments, up to five different query plans were implemented for each query. The exact number depended upon the applicability of specific optimization strategies on the given query. The five query plans are as follows:

1. `Naive`: using the naive evaluation method.

2. `ORR`: applying *Optimization RR* and using *Transformation* 1 to remove the negative predicate.

3. `CF-1`: applying the constraint frequent itemset mining operator CF and using the *Algorithm 1*.

4. `CF-2`: applying the constraint frequent itemset mining operator CF and using the *Algorithm 2*.

5. `GF-1`: applying the group frequent itemset mining operator GF and using the *Algorithm 3* (in Phase 1, and *Algorithm 1* in Phase 2).

2.7.4 Experimental Results

This subsection reports the results we obtained. All experiments are performed on a 933MHZ Pentium III machine with 512 MB main memory.

Table 2.8 presents the running time for the first three queries on IPUMS datasets. Table 2.9 shows the results from the other three queries, Q_4, Q_5, and Q_6, on DARPA datasets. Also, all six queries were used with the QUEST synthetic datasets, and the results are presented in Tables 2.10 and 2.11. Each query is executed with two different pairs of support levels.

The queries Q_1 and Q_5 mainly show how the CF and GF operators can reduce the evaluation cost. The CF operator amounts to an average of more than 3 times speedup on both real and synthetic datasets. The speedups are higher with Query Q_1 than query Q_5, since the CF operator is applied three times in Q_1 and only two times in Q_5. Further, the GF operator performs better than CF operator for both the queries, and gains an average of 4 times the speedup on the real datasets, and up to 14 times speedup on the synthetic datasets.

The queries Q_2, Q_3, Q_5, and Q_6 benefit from the *Optimization RR* and are able to use the CF operator. The `ORR` versions can achieve up to two times the speedup in these cases, and `CF-1` always performs better than `ORR`. The query plan `CF-1` can achieve an additional speedup of more than 5. Further, in all test cases, the versions `CF-1` perform a little better than the version `CF-2`. This suggests that in the phase two, reducing the repetitive computation is more important. At last, the query Q_4 can be optimized by removing the negative predicate, but the CF and GF operators cannot be applied.

The results from the query Q_6 give rise to the following question: *"Why does the* `GF-1` *query plan perform better than the* `CF-1` *plan on QUEST datasets, and* `CF-1` *performs better than* `GF-1` *on IPUMS datasets"*. A related issue is that depending on the datasets and queries, the performance gains from the CF and GF operators can vary significantly. For example, the difference in speedup varies from 3 to 14 in our experiments. By further

analyzing the detailed cost of each query, we believe that one of the key factors impacting the performance gains from both CF and GF operators is the ratio of the size of the intersection set with size of the set generated directly from the common frequent itemset mining operator. The less the ratio is, the more gain we can get from the GF operator by reducing the unnecessary computation and lesser repetitive computation is introduced. For example, in the query $Q_1(50\%, 35\%)$ on IPUMS datasets, the size of intersection set is 19 times smaller than the total size of the four sets of frequent itemsets. However, in query Q_1 on QUEST synthetic datasets, the size of the intersection set is more than 1000 times smaller than the total size of the four sets of frequent itemsets.

To summarize, the new query plans `CF-1` and `GF-1` do result in improved performance, provided they are applicable on a given query. In our experiments, they show an improvement ranging from a factor of 2 to 15. Moreover, the size of intersection set is a significant factor impacting the performance gains from the use of CF and GF operators.

2.8 Mining Generalized Patterns on Multiple Datasets

In the past several years, the field of frequent pattern mining has gone beyond frequent itemset mining. Algorithms have been developed to mine a very rich class of patterns or structures, including sequential pattens, sub-graphs, sub-trees, and other topological structures [105, 7, 118, 8]. Also, in order to discover interesting patterns, comparing and analyzing interesting patterns from *multiple* datasets is often required. We refer to the patterns mined by algorithms for frequent pattern mining (besides itemsets) as *complex patterns*.

In this section, we briefly outline how our framework and techniques for optimizing operations for frequent itemset mining can be extended to handle complex patterns. Specifically, we focus on the following three questions. First, can our SQL extensions and Algebra be used for operations on complex patterns? Second, can our mining operators, the M-table representation, and query plan generation algorithms still be used to optimize queries on the complex patterns? Third, what are the key implementation issues in handling complex patterns in our system?

2.8.1 SQL and Algebra for Mining Complex Patterns on Multiple Datasets

Let $\{A_1, A_2, \cdots, A_m\}$ be the set of datasets, which contain complex patterns that we are interested in analyzing and comparing. The datasets are *homogeneous*, in the sense that the same item, or the same vertex/edge label, has the same name across different datasets. Let T be the set of all possible patterns in all datasets. We can then define the following schema,

$$Frequency(T, A_1, A_2, \cdots, A_m)$$

For a table F of this schema, the column with attribute $F.T$ stores all possible patterns, and the column with attribute $F.A_i$ holds the frequency of the patterns at their corresponding rows on the dataset A_i. Note that itemset mining has become a special case of this definition, where the first column stores all the frequent itemsets ($F.I$).

As was the case for itemset mining, the table F usually cannot be materialized because of the large number of potential patterns. It only serves as a virtual table or a logical view. Similar to mining itemsets, an SQL query will be used to partially materialize the virtual frequency table F, which has the following format.

SELECT $F.T, F.A_{i1}, F.A_{i2}, \cdots, F.A_{is}$

FROM *Frequency*$(T, A_1, \cdots, A_m)$ F

WHERE *Condition* C

where, $\{A_{i1}, \cdots, A_{is}\} \subseteq \{A_1, \cdots, A_m\}$, and *Condition* is defined the same as in itemset mining.

To deal with the complex patterns, we can define the basic operator as $SFT(A_j, \alpha)$, which mines frequent complex patterns on the dataset A_j with support level α. The basic operations ($\sqcup$ and $\sqcap$) will remain the same. Therefore, the above SQL queries can be translated into the algebra format and then be normalized to the standard form.

2.8.2 New Operators and Query Plans

Recall that in mining itemsets on multiple datasets, the standard form of a query is mapped to the M-table format. M-table captures the relationships among the basic operators and operations. Using the M-table representation, we can explore the search space of query plans and find the efficient ones. However, efficient query plans often rely on the additional mining operators, such as the CF operator. Therefore, the main challenge for complex pattern mining using the approach presented in this chapter is, *"Can new mining operators similar to CF be defined for complex pattern mining? "*

We have an affirmative answer to this question. The reason is that the new frequent complex pattern mining algorithms are all based on the *down-closure* property, i.e., if a complex pattern is frequent, then all its sub-patterns are also frequent. Therefore, new frequent pattern mining operator CFT can be defined in very similar ways to the operator CF.

Frequent complex pattern mining operator with constraints

$CFT(A_j, \alpha, X)$ finds the complex patterns that are frequent in the dataset A_j with support

α and also appears in the set X. X is a set of complex patterns that satisfies the down-closure property. This operator also reports the frequency of these patterns in A_j. Formally, $CFT(A_j, \alpha, X)$ computes the following view of the F table:

$$X \sqcap SFT(A_j, \alpha)$$

The efficiency of this operator comes from the fact that by deeply pushing the set X into the frequent pattern generation procedure, where X can serve as the search space for the frequent pattern generating, the extra computation for the itemsets not in X can be saved.

2.8.3 Implementation

There are two key issues in extending our system to work with the complex patterns. The first issue is that we need efficient implementations of the new operator CFT for different patterns. The second issue is to efficiently cache complex patterns in our knowledgeable cache.

Implementation of the operator CFT (or other similar operators) is fairly straightforward. They can be implemented based on the frequent pattern mining operator (SFT), for which algorithms and their implementations are available. For example, consider implementing the frequent complex pattern mining operator with constraints, $CFT(A_j, \alpha, X)$. We can put the set of complex patterns X into a hash table. Then, in either vertical mining or level-wise mining approach (for SFT), as we we try to generate a possible candidate complex pattern, we will first test if the candidate pattern appears in the hash table. If it is not in the set X, we will simply prune this candidate.

Similar to itemset mining, a prefix-tree like data structure can be used to cache mining results from complex patterns. The reason is as follows. First, the results of these mining operators satisfy the down-closure property. Further, since our cache is the union of all

mining results, it also satisfies the down-closure property. Therefore, all complex patterns can actually be organized in a prefix-tree data structure. This prefix tree can be either stored in the main memory or in the secondary memory. The basic operations on the cache can also be easily implemented.

2.9 Related Work

Much research has been conducted to provide database support for mining operations. Han, Meo, Imielinski, and their colleagues have proposed extensions of the database query languages to support mining tasks [47, 57, 74]. Sarawagi and Agrawal [93] and Chaudhuri and his colleagues [20] have studied implementing Apriori association mining algorithm and decision tree construction, respectively, on a database system. ATLas [112] applies user-defined functions (UDFs) to express data mining tasks. However, all of these efforts focus on mining a single dataset with relatively simple conditions.

A number of constraint frequent itemset mining algorithms have been developed to use additional conditions and prune the search space [16, 68, 77, 98]. However, these algorithms cannot efficiently answer our queries, since the conditions in our queries corresponds to a set of (in)frequent itemsets. These cannot be directly used to reduce the search space with their methods. We have developed a systematic approach for finding efficient query plans answering these queries.

Raedt and his colleagues have studied the generalized inductive query evaluation problem [66, 69]. Although their queries target multiple datasets, they focus on the algorithmic aspects to apply version space tree and answer the queries with the generalized monotone

and anti-monotone predicates. In comparison, we are interested in answering queries involving frequency predicates more efficiently. We have developed a table based approach to generate efficient query plans.

Our research is also different from the work on *Query flocks* [101]. While they target complex query conditions, they allow only a single predicate involving frequency, and on a single dataset. The work on multi-relational data mining [30, 87, 111] has focused on designing efficient algorithms to mine a single dataset materialized as a multi-relation in a database system.

Finally, a number of researchers have developed techniques for mining the difference or *contrast sets* between the datasets [12, 29, 106]. Their goal is to develop efficient algorithms for finding such a difference, and they have primarily focused on analyzing two datasets at a time. In comparison, we have provided a general framework for allowing the users to compare and analyze the patterns in multiple datasets. Moreover, because our techniques can be a part of a query optimization scheme, the users need not be aware of the new algorithms or techniques which can speedup their tasks.

2.10 Conclusions

The work presented in this chapter is driven by two basic observations. First, analyzing and comparing patterns across multiple datasets is critical for many applications of data mining. Second, it is desirable to provide support for such tasks as part of a database or a data warehouse, without requiring the users to be aware of specific algorithms that could optimize their queries.

We have presented a systematic approach for expressing and optimizing frequent itemset queries that involve complex conditions across multiple datasets. Specifically, we have

proposed an SQL-based mechanism and have established an algebra for such queries. We have developed a number of new optimizations, new operators, transformations, and heuristic algorithms for finding query plans with reduced execution costs. Our experiments have demonstrated up to an order of magnitude performance gains on both real and synthetic datasets. Thus, we believe that our work has provided an important step towards building an integrated, powerful, and efficient KDDMS.

CHAPTER 3

SIMULTANEOUS OPTIMIZATION OF COMPLEX MINING TASKS WITH A KNOWLEDGEABLE CACHE

3.1 Introduction

In the last chapter, we studied how to evaluate a single mining query on multiple datasets efficiently. However, considering the iterative and exploratory nature of knowledge discovery or data mining, especially in view of the need for interactive response to the users, new techniques are needed to further improve the evaluation performance.

In this chapter, we study how to evaluate mining queries in a *query intensive* environment. Specifically, we envision the following scenarios that a Knowledge Discovery and Data Mining System (KDDMS) will have to support and optimize for:

- *Sequence of Queries:* A user may analyze one or more datasets by issuing a sequence of related complex mining queries. This may be due to the iterative and exploratory nature of the process, where the mining parameters and constraints are modified till desired insights are gained from the dataset(s).

- *Multiple Simultaneous Queries:* Several users may be analyzing a set of datasets concurrently, and may issue related complex queries.

The need for supporting and optimizing such scenarios has been well recognized in database and OLAP systems. *Views* have been used to optimize a sequence of database operations [45], and similarly, techniques such as *reducing common subexpressions* [96, 91] have been used. However, because the nature of the mining operations is very different from nature of database and OLAP operations, these techniques cannot apply to a KDDMS system.

Some efforts have been made towards addressing these issues for mining environments. Nag *et al.* have studied how a *knowledgeable* cache can be used to help to perform interactive discovery of association rules [75]. They maintain a cache to record (in)frequent itemsets with their support levels, and then modify the frequent itemset mining algorithm to utilize the itemsets in the cache. The focus of their research is on frequent itemset mining without complex mining conditions. Ng *et al.* have studied constraint association rule mining [77]. In their method, multiple queries can be merged as a single query for evaluation. Hipp and Guntzer have argued that execution of data mining queries with constraints can be very expensive [51]. Therefore, they have proposed to use pre-computation of frequent itemsets of certain support levels to answer constraint itemset mining queries.

The above efforts have two important limitations. First, sequence of queries and multiple simultaneous queries have not been studied together. Second, the techniques involving the use of knowledgeable cache have been restricted to deal with simple data mining queries.

In this chapter, we focus on the problem of efficiently evaluating frequent pattern mining queries on multiple datasets in a query intensive environment, where one needs to optimize multiple simultaneous queries, as well as a sequence of related queries. Particularly, we show how multiple simultaneous queries can be optimized, and how the results from

past mining queries can be utilized to evaluate the current ones. Due to the complexity and characteristics of such queries, simultaneous optimization of multiple queries and caching of their query results is challenging, and quite different from the existing work in this area.

Overall, this chapter makes the following contributions:

1. We present a novel system architecture to deal with a query intensive environment that needs to support and optimize both multiple simultaneous queries and a sequence of queries.

2. We propose new algorithms to perform multiple-query optimization for frequent pattern mining on multiple datasets.

3. We show the design of a knowledgeable cache which can store the past query results from queries on multiple datasets. We present algorithms which enable the use of the results stored in such a cache to further optimize multiple queries.

4. We have implemented and evaluated our system with both real and synthetic datasets. Our experimental results show that our techniques can achieve a speedup of up to a factor of 9, compared with the systems which do not support caching or optimize for multiple queries.

The rest of the chapter is organized as follows. In Section 3.2, we present our framework to deal with both multiple simultaneous queries and a sequence of queries. In Section 3.3, we discuss the important properties of the M-table, which form the basis for our multiple query optimizations and caching of query results. In Section 3.4, we present our optimization algorithms. In Section 3.5, we discuss the major implementation issues for our system, and present our experimental results. We compare our work with related research efforts in Section 4.5, and conclude in Section 4.6.

3.2 System Architecture and Optimization Overview

Let us envision a KDDMS system in which there are multiple datasets and multiple users. If different users issue queries each of which involves multiple datasets, it is quite likely that the queries could have a significant overlap.

For example, consider the following two queries, $\mathcal{Q}_1$ and $\mathcal{Q}_2$, which are issued simultaneously.

$\mathcal{Q}_1$: SELECT $F.I, F.A, F.B, F.X$
FROM *Frequency*(I, A, B, X) F
WHERE $F.A \geq 0.2$ AND $F.B \geq 0.1$ AND $F.X < 0.1$

$\mathcal{Q}_2$: SELECT $F.I, F.A, F.B, F.Y, F.Z$
FROM *Frequency*(I, A, B, Y, Z) F
WHERE ($F.A \geq 0.1$ AND $F.B \geq 0.1$
AND $F.Y \geq 0.1$ AND $F.Z < 0.01$)
OR ($F.Z > 0.2$ AND $F.Y < 0.01$)

These two queries overlaps on the datasets A and B. The question for us is, *"How can we exploit the overlap in the two queries to generate query plans that are more efficient than the independently generated query plans for each query?"*.

Furthermore, we consider the following possibility. As we had described earlier, it is very likely that a single user issues a sequence of related queries. For example, the system might have evaluated the following query $\mathcal{Q}$, before it receives the queries $\mathcal{Q}_1$ and $\mathcal{Q}_2$.

$\mathcal{Q}$: SELECT $F.I, F.A, F.B, F.C, F.D$

FROM *Frequency*(I, A, B, C, D) F

WHERE ($F.A \geq 0.1$ AND $F.B \geq 0.1$)

OR ($F.C \geq 0.1$ AND $F.D \geq 0.1$ AND

($F.A \geq 0.2$ OR $F.B \geq 0.2$))

In such a case, we have the following two additional questions: *"How can we effectively store the results from the recent queries in a cache?"*, and, *"How can we efficiently utilize such cached results to speedup computation of new queries?"*.

Before discussing how we address these issues, we describe our system architecture. This architecture is shown in Figure 3.1. Our system primarily contains four components, a *Query queue*, a *Query plan optimizer*, a *Query evaluation engine*, and a *Cache*. The queries issued by the users of the system are initially stored in the query queue. The query plan optimizer receives all the queries appearing in the queue, and then generates efficient query plans for all of them, simultaneously. In the process, the query plan optimizer utilizes the information in the cache, which maintains the results from a set of recent queries. The query evaluation engine evaluates the queries, based on the query plan that uses the mining operators and the operations defined in the Algebra. This component is also responsible for retrieving the necessary information from the cache. Finally, the query evaluation engine updates the cache, based upon the results of the current queries.

As we discussed above, we have two major goals, which are simultaneous optimization of multiple queries, and maintaining and exploiting a cache to optimize for a sequence of queries. In this section, we give a brief overview of our work. The rest of this chapter provides a more detailed account.

1. Simultaneous optimization of multiple queries: The basic idea here is to reduce the common computations appearing in different queries. This is similar to what is done for

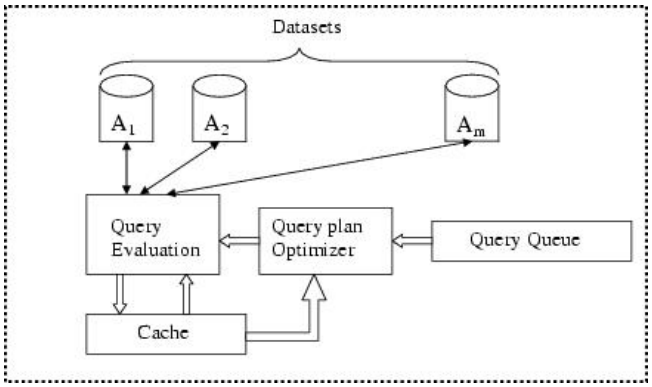


Figure 3.1: System Framework

database queries. However, our method for detecting and optimizing the common computations is quite different from the traditional database approach. Our method is based on M-table. Each mining operator in the query plan is mapped to an M-table representation. The *containment* relationships on the M-table are defined to capture the common or overlapping computations. Further, different M-tables can be merged together into one large table and a *global* query plan can be generated for the large M-table.

Based on the characteristics of the M-table, we propose two different approaches. The first approach utilizes the containment relationship of the M-tables to detect the overlapping computations across multiple queries. Here, each mining query will generate its own query evaluation plan. Then, we will detect and merge the common computations among different evaluation plans. The second approach involves merging the M-tables of different queries into a single M-table, and then generating an efficient global query plan.

2. Knowledgeable cache: Our cache stores the results of each mining operator. Compared to the previous effort on the use of a cache for supporting knowledge discovery [75], an

interesting aspect of our cache is as follows. It not only stores the itemsets with their frequency, but also maintains a high-level *knowledge* or summary of the information being stored. Therefore, when a new query comes in, the cache can systematically determine which part of the query can be directly answered from the cache. Such knowledge is maintained through the use of M-table. We show how we can use the M-table to summarize, update, and utilize the information in the cache.

In the next two sections, we provide a detailed account of these two issues. Specifically, in Section 3.3, we focus on the properties of the M-table which enable the above optimizations. In Section 3.4, we discuss the detailed optimizations and cache management.

3.3 Properties of M-Table for Query Optimization

In this section, we study the properties and operations of M-table, which form the basis for optimizing multiple mining queries and caching their results.

3.3.1 Containment Relationships of M-Tables

We begin with a set of containment relationships defined on the M-tables. These relationships provide a simple mechanism to detect common computations among different queries.

For the next two definitions, we assume we have two M-tables, M_1 and M_2, with the same number of rows (n), and the same row in the two tables corresponds to the same dataset.

Definition 3 *If M_1 and M_2 are both single-column, M_1 is contained in M_2 if for each corresponding pair of cells, $M_1[i]$ and $M_2[i], 1 \leq i \leq n$, either both the cells are empty, or both the cells are non-empty and $M_1[i] \geq M_2[i]$.*

	M_1	M_2	M_3	M_4	M_5
A	0.1	0.08	0.1		0.2
B	0.1	0.05	0.05	0.05	0.05
C				0.1	0.1
D					

Table 3.1: M-Tables with Containment Relationships

If M_1 is contained in M_2, we denote this as $M_1 \subseteq M_2$. For example, in Table 3.1, we have $M_1 \subseteq M_2$.

The intuition behind this definition is as follows. If the desired support levels are higher for the column M_1, then the answer set for the query corresponding to M_1 is a subset of the answer set for the query corresponding to M_2. Thus, the former can be computed from the latter by relatively inexpensive selection operations.

Definition 4 *If M_1 and M_2 are multi-column M-tables, M_1 is contained in M_2 if each column in M_1 is contained by some column in M_2.*

Again, the intuition behind the definition is the same. If each column in M_1 is contained by some column in M_2, the answer set for the query corresponding to M_1 can be obtained by the answer set for the query corresponding to M_2, using relatively inexpensive selection operations.

Given these definitions and the mapping between mining operators and M-tables, we have the following lemma.

Lemma 3 *Consider two mining queries Q_1 and Q_2, and let their associated M-tables be denoted as M_1 and M_2, respectively. If the M-table M_1 is contained in M_2, i.e., $M_1 \subseteq M_2$,*

the necessary information of Q_1 can be derived from the necessary information of Q_2 by a selection operation (σ).

This lemma helps us detect the common computations among queries.

Next, we study a more generalized containment relationship among M-tables, which is based on the cells of M-tables. The motivation for this is as follows. In many cases, the results of a query cannot be completely answered by one or more of the past queries, but part of its result can be derived from them. This containment helps answer these questions.

To facilitate our discussion, we first define the following inequalities for empty cells. Let e be the empty cell and let r be a positive (non-zero) threshold. Then, our discussion assumes the following inequalities, $e \geq e$, $r \geq e$, $0 \geq e, and, e \geq 0$.

For the following definition, we again assume that we have two M-tables, M_1 and M_2, with the same number of rows (n), and the same row in the two tables corresponds to the same dataset.

Definition 5 *Consider a cell c, which is at the row i in the column C_1 of the M-table M_1. This cell is contained in M_2 if there exists a column in M_2, denoted as C_2, such that: 1) $C_1[i]$ and $C_2[i]$ are both non-empty, and 2) $C_1[j] \geq C_2[j], \forall j, 1 \leq j \leq n$.*

We denote such containment as $c \subseteq M_2$. Intuitively, c is contained in M_2 if we can use the corresponding cell in the column C_2 to color the cell c. The reason we require $C_1[j] \geq C_2[j]$, for each pair of corresponding cells in the two columns, is that we need information in C_2 to be a superset of the information required for the cell c.

As an example, in Table 3.4, the cell at the row three in the single-column M-table corresponding to O_4, denoted as $O_4[3]$, is contained in the M-table for O_2. Formally, we say, $O_4[3] \subseteq O_2$.

Based upon the above definition, we have the following definition to relate one M-table to a set of M-tables.

Definition 6 *An M-table, M', is* cell-contained *in the group of M-tables, $M_1, \cdots, M_k$, if each non-empty cell in M' is contained by at least one M-table in the set $M_1, \cdots, M_k$.*

Formally, we denote this as

$$M' \subseteq_c \{M_1, \cdots, M_k\}$$

As an example, in Table 3.1, we have $M_5 \subseteq_c \{M_3, M_4\}$.

Given this definition, we have the following lemma to detect if the necessary information of a query can be derived from a group of other queries.

Lemma 4 *Let Q' be a query with an M-table, M', and let $Q_1, \cdots, Q_k$ be a group of queries with the corresponding M-tables $M_1, \cdots, M_k$, respectively. If M' is cell-contained in $M_1, \cdots, M_k$, then the necessary information of Q' can be derived from the necessary information of $Q_1, \cdots, Q_k$.*

Our discussion in this subsection has so far assumed that the M-tables have the same number of rows, and the same row in each table corresponds to the same dataset. However, this is not a serious limitation. If two M-tables do not satisfy this condition, we can *align* them to meet this condition. Briefly, this alignment procedure is as follows. First, we take a union of the two sets of datasets. Then, we extend the two M-tables to have the same number of rows, corresponding to the union of the set of datasets. This will involve adding rows where each cell will be empty. Finally, we shuffle the rows in the two M-tables to let each row represent the same dataset.

	M_1 ($\mathcal{Q}_1$)	
A	0.2	0.2
B	0.1	0.1
X		0.1

	M_2 ($\mathcal{Q}_2$)			
A	0.1	0.1		
B	0.1	0.1		
Y	0.1	0.1		0.01
Z		0.01	0.2	0.2

	$M_1 \oplus M_2$					
A	0.2	0.2	0.1	0.1		
B	0.1	0.1	0.1	0.1		
X		0.1				
Y			0.1	0.1		0.01
Z				0.01	0.2	0.2

Table 3.2: Merge Operation for M-Tables

3.3.2 The Merge Operation for M-Tables

We now define the merge operation for the M-Tables. This operation helps in replacing multiple queries by a single large query, and also helps maintain a high-level summary of the contents of the cache. Again, our definition assumes that the M-tables being merged have been *aligned*, i.e., they have the same number of rows and the same row in each table corresponds to the same dataset.

Definition 7 *The merge operation, denoted as $\oplus$, on two M-tables, M_1 and M_2, results in a table with the same rows, and a set of columns that is the union of the set of columns in M_1 and M_2.*

As an example, Table 3.2 shows the merged table, $M_1 \oplus M_2$, where, M_1 and M_2 are M-tables for the queries $\mathcal{Q}_1$ and $\mathcal{Q}_2$, respectively.

Clearly, the original tables are *contained* in the merged table, that is

$$M_1, M_2 \subseteq M_1 \oplus M_2$$

The implication of the above observation is as follows. For two M-tables M_1 and M_2, corresponding to the queries, Q_1 and Q_2, respectively, the answering set of both Q_1 and Q_2 can be derived from the result of the merged M table, $M_1 \oplus M_2$. This fact will be used to process multiple queries, as well as to update the knowledgeable cache with different mining operators.

3.4 Multiple Query Optimization Approach

In this section, we present our optimization algorithms which are based on M-tables. Specifically, in Subsection 3.4.1, we first review how the query plan for a single query is generated from an M-table. In Subsection 3.4.2, we study how each mining operator can be mapped to the M-table and how the redundant mining operators can be detected. In Subsection 3.4.3, we discuss how local plans from several queries can be optimized together. In Subsection 3.4.4, we introduce another approach for optimizing multiple queries, which involves merging multiple queries into one query, and then generating a global query plan. Subsection 3.4.5 focuses on how M-table can be used to summarize and update the cache, and how the cache can help us reduce the evaluation costs.

3.4.1 Single Query Plan Generation

We begin with introducing a new mining operator CF. We introduce this operator because using only the SF operator to evaluate queries can be very expensive.

Frequent itemset mining operator with constraints $CF(A_j, \alpha, X)$ finds the itemsets that are frequent in the dataset A_j with support α and also appears in the set X. X is a set of itemsets that satisfies the *down-closure* property, i.e., if an itemset is frequent, then all its subsets are also frequent. This operator also reports the frequency of these itemsets in A_j.

	F_1	F_2	F_3	F_4	F_5
A	0.1	0.1		0.05	
B	0.1	0.1			0.05
C			0.1	0.1	0.1
D		0.05	0.1	0.1	0.1

Table 3.3: Colored M Table for the query $\mathcal{Q}$

Formally, $CF(A_j, \alpha, X)$ computes the following view of the F table:

$$X \sqcap SF(A_j, \alpha)$$

Note that we can also define and use other mining operators to speedup the evaluation process [63]. For simplicity, we will only use CF and SF in this chapter. Our overall approach can be easily extended to include other mining operators as well.

Now, we focus on query plan generation using the M-table. One of the important features of M table is it can capture the evaluation process for a query by a simple coloring scheme. This coloring scheme is as follows. Initially, all the cells are black. Each invocation of a mining operator (like SF and CF) can color a number of non-empty cells red. This implies that the information corresponding to these cells has been computed. The query evaluation process is complete when all non-empty cells are colored red.

As a running example, consider applying $SF(A, 0.05)$, $SF(C, 0.1)$, $CF(B, 0.1, SF^I(A, 0.1))$, and $CF(D, 0.1, SF^I(C, 0.1))$ consecutively on an initially black-colored table M of the query $\mathcal{Q}$. Table 3.3 shows the resulting colored table (unshaded for black-colored, and

shaded for red-colored). In the following, we look at how the SF and CF operators color the table.

Frequent mining operator $SF(A_i, \alpha)$: An invocation of the frequent mining operator on the dataset A_i, with support α, will turn each non-empty cell at row i who is greater than or equal to α red. In our example, the first operator, $SF(A, 0.05)$, will turn the cells $M_{1,1}$, $M_{1,2}$, and $M_{1,4}$ red, and the second operator, $SF(C, 0.1)$, will turn the cells $M_{3,3}$, $M_{3,4}$, and $M_{3,5}$ red.

Frequent mining operator with constraint $CF(A_i, \alpha, X)$: The coloring impacted by this operator is dependent on the current coloring of the table M. Let X be the set of frequent itemsets defined by all the red cells, and let S be the set of columns where these red cells appear. Then, by applying this operator on dataset A_i with support α, all cells on row i whose column is in the set S, and whose value is less than or equal to α, will turn red.

In our running example, the third operator $CF(B, 0.1, SF^I(A, 0.1))$
picks the red cells $M_{1,1}$ and $M_{1,2}$ by the parameter

$$X = SF^I(A, 0.1)$$

The set S includes the first two columns. Therefore, this operator turns the cells $M_{2,1}$ and $M_{2,2}$ red. Similarly, the fourth operator turns the cells $M_{4,3}$, $M_{4,4}$, and $M_{4,5}$ red.

By the above formulation, the query evaluation problem has been converted into the problem of coloring the table M. Different operators can be used, and in different order, to color the entire table red. Generating optimal query plan is *NP-hard*, and a number of heuristic algorithms have been developed to find efficient query plans [63]. Here, we will only discuss one of the algorithms, the *Algorithm-CF*, which uses SF and CF operators to optimize the query evaluation. *Algorithm-CF* splits the evaluation into two phases. In the

first phase, we use the $SF(A_j, \alpha)$ operators so that each column has at least one red cell. In the *second* phase, we use the $CF(A_j, \alpha, X)$ operators to compute all other non-empty cells in the table.

The sketch of *Algorithm-CF* is listed is in Figure 3.2. It involves minimizing costs for each of the two phases. Since precise cost functions for each operator are not available, a simple heuristic based on the support level is used to estimate the cost. In general, for a single dataset, higher support level for the the SF operator implies lower computation.

Input: table M without coloring
Phase 1
Enumeration of possible SF operators to find the least cost
to cover at least one cell red for each column
Phase 2
Find datasets whose corresponding rows have black cells;
For each row, find the lowest support level among the black cells;
On each row, invoke the CF operator with
the lowest support level:
Across the rows, invoke the operator in the decreasing order
of support level used for the CF operator.

Figure 3.2: Algorithm-CF for Query Plan Generation

Algorithm-CF will generate the following query plan for the query $\mathcal{Q}$.

$$Phase1: \quad SF(A, 0.1), SF(C, 0.1);$$

$$Phase2: \quad CF(A, 0.05, SF(C, 0.1)^I);$$

$$CF(B, 0.05, (SF(A, 0.1) \sqcup SF(C, 0.1))^I);$$

$$CF(D, 0.05, ((SF(A, 0.1) \sqcap SF(B, 0.1)) \sqcup SF(C, 0.1))^I);$$

Note that F^I returns the first column of table F, i.e. the set of itemsets recored in F.

3.4.2 Mapping Mining Operators to M-Tables

Each mining operator in a query plan can be uniquely mapped to an M-table. This mapping plays an important role in multiple query optimization and cache management. This is because common computations among the mining operators can be easily captured using M-table, and similarly, the result of each mining operator can be uniformly expressed using M-tables.

We had earlier described how the two operators, SF and CF, contribute to the coloring of the table, and help generate query plans. Since part of our goal is to use an M-table to capture the cache, we define rules to map each different mining operator in a query plan to a unique M-table.

Frequent mining operator $SF(A_j, \alpha)$: An invocation of this operator on dataset A_j and support α will generate a single column M-table whose row j is α, and other rows are empty.

Frequent mining operator with constraint $CF(A_j, \alpha, X)$: Recall that the CF mining operator is used to color a set of columns, denoted as S, who have at least one cell to be colored red, and the cell at the row j for each column in S is black. Then, the M-table generated by the CF operator is composed of these columns in the set S, with the following exception. The cells which are still black after the CF mining operator will become empty in this new M-table.

Consider the following *incomplete* query plan for the query $\mathcal{Q}$.

$$O_1 : SF(A, 0.1);$$

	O_1	O_2	O_3		O_4
A	0.1		0.1		
B			0.05	0.05	0.05
C		0.1		0.1	0.1
D					0.1

Table 3.4: M-Tables of different mining operators

$$O_2 : SF(C, 0.1);$$

$$O_3 : CF(B, 0.05, (SF(A, 0.1) \sqcup SF(C, 0.1))^I);$$

$$O_4 : CF(D, 0.1, CF^I(B, 0.1, SF^I(C, 0.1));$$

Table 3.4 shows the corresponding M-tables for the mining operators in the above query plan.

The significance of associating an M-table with each mining operator is that the common computation among mining operators can be treated the same way as the query results. In particular, Lemmas 3 4 can be modified to apply to mining operators, instead of mining queries. In next subsection, we will use such methods to reduce the redundant computations among different query plans.

3.4.3 Optimizing Local Plans

To optimize multiple simultaneous queries, this approach generates local query plans for each query, and then tries to remove the common computations among the query plans. The common computations are categorized into two groups. In the first group, a mining operator in a query plan can be derived from another mining operator in one of the other query plans. In the second group, a mining operator in a query plan can be derived from

a group of mining operators which are in other query plans, or are in the same query plan but scheduled before this operator. As discussed in Subsection 3.3.1, we can detect these common computations by the containment relationship defined on the M-tables.

The difficulty of this approach is that different query evaluation order will result in different ways to remove the common computations. For example, assume one query plan has the mining operator,

$$CF(X, 0.1, (SF(A, 0.1) \sqcup SF(B, 0.2))^I)$$

and another query plan includes

$$CF(A, 0.1, SF(X, 0.1)^I), CF(B, 0.2, SF(X, 0.1)^I)$$

Since the two sets of mining operators are equivalent, depending on which query plan is evaluated first, we have different ways to eliminate the common computations. Note that in order to simplify the above problem, we are not considering combining local plans together into a global plan. This will be the topic of the next subsection.

To find the evaluation order for n queries to achieve the maximal savings from removing the common computations, a simple enumeration method will have the time complexity $O(n!)$. If n is large, this method is very expensive. Therefore, we propose a greedy algorithm, which is sketched in Figure 3.3. This greedy algorithm utilizes the following property. If a query plan, Q, is scheduled after a set of query plans, S, then the contained mining operators in Q do not depend on how the contained mining operators are removed within the set S. This is based on the transitive property of the containment relationships. To utilize this property, our algorithm finds the query plan which has the maximal savings when it is scheduled as the last one. Such a plan is then scheduled last, and then the order of the remaining operations is determined. Note that since the exact savings cannot typically

Input: local query plans $Q_1, \cdots, Q_n$
$S = \{Q_1, \cdots Q_n\}$;
While ($S \neq \emptyset$) **Do**
  Foreach $Q_i \in S$
    Eliminate Containment:
      If any mining operator in Q_i is contained in $S - \{Q_i\}$
    Eliminate Cell-Containment:
      If any mining operator in Q_i is cell-contained in mining operators in $S - \{Q_i\}$ or in Q_i but scheduled before this operator
    Find the savings from the above eliminations;
  Let Q_j in S have the maximal savings;
  Eliminate the contained mining operators from Q_j;
  Scheduled Q_j after $S - Q_j$;
  $S = S - \{Q_j\}$.

Figure 3.3: Greedy Algorithm to Remove Containment in Multiple Query Plans

be determined, we use simple heuristics, such as the number of mining operators, as the cost function.

Consider applying the greedy algorithm on the query plans of query $\mathcal{Q}_1$ and $\mathcal{Q}_2$, which are as follows:

$$\begin{aligned}
\mathcal{Q}_1: \quad & SF(A, 0.2); \\
& CF(B, 0.1, SF(A, 0.2)^I); \\
& CF(X, 0.1, (SF(A, 0.2) \sqcap SF(B, 0.1))^I); \\
\mathcal{Q}_2: \quad & SF(A, 0.1), SF(Z, 0.2); \\
& CF(B, 0.1, SF(A, 0.1)^I); \\
& CF(Y, 0.01, ((SF(A, 0.1) \sqcap SF(B, 0.1)) \sqcup SF(Z, 0.2))^I); \\
& CF(Z, 0.01, ((SF(A, 0.1) \sqcap SF(B, 0.1) \sqcap SF(C, 0.1))^I);
\end{aligned}$$

The algorithm will schedule the query $\mathcal{Q}_2$ before $\mathcal{Q}_1$, and the first two mining operators in the query plan of $\mathcal{Q}_1$ will be eliminated.

3.4.4 Global Query Plans

A drawback of the above approach is that it is very sensitive to the local plans, and often cannot find efficient query plans. For example, consider the new query $\mathcal{Q}_2'$ which is created by replacing the sub-condition in the query $\mathcal{Q}_2$, $F.B \geq 0.1$ by $F.B \geq 0.15$. The query plan for $\mathcal{Q}_2'$ is as follows.

$$SF(B, 0.15), SF(Z, 0.2);$$

$$CF(A, 0.1, SF(B, 0.15)^I);$$

$$CF(Y, 0.01, ((SF(A, 0.1) \sqcap SF(B, 0.15)) \sqcup SF(Z, 0.2))^I);$$

$$CF(Z, 0.01, ((SF(A, 0.1) \sqcap SF(B, 0.15) \sqcap SF(C, 0.1))^I);$$

If we are evaluating queries $\mathcal{Q}_1$ and $\mathcal{Q}_2'$ together, the above approach can not find any common computations between the two query plans, and the mining operators will be invoked 8 times.

However, the M-table format of queries enables us to perform more aggressive optimizations. This new approach does not depend on the local query plans. Instead, this approach combines the local M-tables from different queries into a single large M-table by the merge operation ($\oplus$). Then, it generates a global query plan based on this merged M-table. Consider the merged M-tables for query $\mathcal{Q}_1$ and $\mathcal{Q}_2'$ in Table 3.5.

We can have the following global query plan which needs only 6 mining operators.

$$SF(A, 0.1), SF(Z, 0.2);$$

$$CF(A, 0.1, SF(B, 0.1)^I);$$

$$CF(X, 0.1, ((SF(A, 0.2) \sqcap SF(B, 0.1))^I));$$

$$CF(Y, 0.01, ((SF(A, 0.1) \sqcap SF(B, 0.15)) \sqcup SF(Z, 0.2))^I);$$

A	0.2	0.2	0.1	0.1		
B	0.1	0.1	0.15	0.15		
X		0.1				
Y			0.1	0.1		0.01
Z				0.01	0.2	0.2

Table 3.5: Merged M-Table for Query $\mathcal{Q}_1$ and $\mathcal{Q}_2'$

$$CF(Z, 0.01, ((SF(A, 0.1) \sqcap SF(B, 0.15) \sqcap SF(C, 0.1))^I);$$

Compared with the first approach, this global query plan replaces the four mining operators $SF(B, 0.15)$, $SF(A, 0.2)$, $CF(A, 0.1, SF(B, 0.15)^I)$, $SF(B, 0.1, SF(A, 0.2)^I)$ by two mining operators, $SF(A, 0.1)$, $CF(A, 0.1, SF(B, 0.1)^I)$. This is likely to be more efficient.

3.4.5 Knowledgeable Cache Management and Utilization

We now discuss how the M-table can be used for summarizing our cache. Assume in our system, there are a total of p distinct datasets. Then, our cache can use an M-table with p rows, where each row corresponds to a dataset, to represent the past evaluation results that are stored in the cache. The set of columns of the M-table are dynamically changed after each invocation of a mining operator.

This update procedure is quite simple. Earlier, we had described how each mining operator in a query plan is mapped to an M-table. After invocation of a mining operator, besides inserting the mining results in the cache, the M-table for the mining operator will be merged with the M-table that summarized the cache earlier.

A	0.1		0.05	0.1		0.1	
B				0.05	0.05	0.1	
C		0.1	0.1		0.1		0.1
D						0.05	0.05
E							

Table 3.6: M Table for the Cache

A	0.2	0.2	0.1	0.1		
B	0.1	0.1	0.15	0.15		
X		0.1				
Y			0.1	0.1		0.01
Z				0.01	0.2	0.2

Table 3.7: Pre-Colored M-Table for Query $\mathcal{Q}_1$ and $\mathcal{Q}'_2$

Consider the query plan for the query $\mathcal{Q}$ described earlier, and assume the cache is empty initially. Then, the M-table of the cache after the evaluation of this query plan is shown in Table 3.6.

The high-level knowledge of our cache can be used to answer which part of a new query can be answered directly from the cache. Further, to help with the query plan generation, this information is represented by pre-coloring the M-table for the new queries. This is done by using the generalized containment relationship of M-tables based on cells. For each non-empty cell in the M-table for queries, we search the M-table of the cache to see if a column *contains* it. If such a column exists, the cell will be turned red. As an example, assume we have a cache with an M-table shown in Table 3.6. The pre-coloring of the merged M-table for queries $\mathcal{Q}_1$ and $\mathcal{Q}'_2$ is shown as Table 3.7.

After such pre-coloring, less cells need to be colored, and more efficient query plans can be generated. For the first approach to optimize multiple queries (Subsection 3.4.3), different local query plans are generated from the pre-colored M-tables, and then the common computations among them are removed. For the second approach (Subsection 3.4.4), a global query plan is generated from the pre-colored merged M-tables. For queries $\mathcal{Q}_1$ and $\mathcal{Q}_2'$, both approaches will generate the following query plan:

$$SF(Z, 0.2);$$
$$CF(X, 0.1, (SF(A, 0.2) \sqcap SF(B, 0.1))^I));$$
$$CF(Y, 0.01, ((SF(A, 0.1) \sqcap SF(B, 0.15)) \sqcup SF(Z, 0.2))^I);$$
$$CF(Z, 0.01, ((SF(A, 0.1) \sqcap SF(B, 0.15) \sqcap SF(C, 0.1))^I);$$

3.5 System Implementation and Experimental Evaluation

This section reports a series of experiments we conducted to demonstrate the efficacy of the optimization techniques we have developed. [1] Particularly, we were interested in the following questions:

1. What are the performance gains from two different approaches to simultaneously optimize multiple mining queries ?

2. What are the performance gains from the knowledgeable cache, and/or from pre-computation of frequent itemsets with certain threshold ?

[1] Thanks for Kaushik Sinha's help with the implementaion and experimental evaluation.

Initially, we briefly describe how we have implemented our cache, and the datasets and the queries used for our experiments.

3.5.1 Cache Implementation

In our current implementation of the cache, we use a memory-based hash-tree like data structure to maintain the itemsets with their frequency counts. For each dataset, we maintain an independent hash-tree. We define three primitives to access the cache. These are: the *add* operation, which adds a set of itemsets and their frequency in the cache, the *get* operation, which takes as parameter the support level α, and gets the set of itemsets with support level higher than or equal to α from the cache, and finally, the *remove* operation, which removes the itemsets whose support is lower than the given parameter for the specified dataset.

3.5.2 Datasets

Our experiments were conducted on two groups of datasets, each of them comprising four distinct datasets:

IPUMS: The first group of datasets is derived from the *IPUMS* 1990-5% census microdata, which provides information about individuals and households [1]. The four datasets each comprises 50,000 records, corresponding to New York, New Jersey, California, and Washington states, respectively. Every record in the datasets has 57 attributes.

IBM's Quest: The second group of datasets represents the market basket scenario, and is derived from IBM Quest's synthetic datasets [5]. The first two datasets, dataset-1 and dataset-2, are generated from the $T20.I8.N2000$ dataset by some perturbation. Here, the number of items per transactions is 20, the average size of large itemsets is 8, and the number of distinct items is 2000. For perturbation, we randomly change a group of items to

	Query Conditions
Q_1	$A \geq \alpha_1 \wedge B \geq \alpha_2$
Q_2	$(A \geq \alpha_1 \vee B \geq \alpha_2) \wedge C < \beta_1$
Q_3	$(A \geq \alpha_1 \wedge B < \beta_1) \vee (B \geq \alpha_2 \wedge A < \beta_2)$
Q_4	$(A \geq \alpha_1 \wedge B \geq \alpha_2 \wedge C < \beta_1 \wedge D < \beta_1) \vee$ $(C \geq \alpha_1 \wedge D \geq \alpha_2 \wedge A < \beta_1 \wedge B < \beta_2)$
Q_5	$A \geq \alpha_1 \wedge B \geq \alpha_1 \wedge C \geq \alpha_1 \wedge D < \beta_1$
Q_6	$(A \geq \alpha_1 \wedge B < \beta_1 \wedge C < \alpha_2 \wedge D < \beta_2) \vee$ $(B \geq \alpha_3 \wedge A < \beta_3 \wedge C < \alpha_4 \wedge D < \beta_4) \vee$ $(C \geq \alpha_5 \wedge A < \beta_5 \wedge B < \alpha_6 \wedge D < \beta_6) \vee$

Table 3.8: Test Query Templates for Our Experiments

other items with some probability. The other two datasets, dataset-3 and dataset-4, are similarly generated from the $T20.I10.N2000$ dataset. Each of four datasets contains 1,000,000 transactions.

3.5.3 Test Queries

We use a collection of query templates involving a different number of datasets, ranging from one to four. Each template involves several different thresholds. For convenience, the thresholds are classified into two groups. A threshold is positive if it is in a positive predicate, and negative if it is in a negative predicate. Table 3.8 illustrates several templates used in our experiment, where we use α and β to represent the positive and negative thresholds, respectively. To generate a query from these query templates, we assign values to each threshold in the query template. For IPUMPS datasets, a positive threshold ranges from 50% to 90%, and a negative threshold from 30% to 60%. For Quest datasets, a positive threshold ranges from 0.1% to 0.9%, and a negative threshold from 0.05% to 0.2%.

3.5.4 Experimental Settings

In our experiments, we evaluate three methods to deal with multiple mining queries. The first is the *naive* method, which generates efficient query plan for each single query, without considering their common computations. The second method is as described in Subsection 3.4.3. It tries to remove the common computations among the local query plans and greedily selects an evaluation order. The third method is as described in Subsection 3.4.4. It merges the local queries into one single query by using the M table format, and then generates an efficient global query plan. For our discussion, we denote them as SQ (single query plan), LQ (local query plan), and GQ (global query plan), respectively. In each of these methods, we use the *Algorithm-CF* to generate our query plans.

We also consider the following experimental settings to study the impact of pre-computation and caching.

Setting-I: No pre-computation and caching,

Setting-II: Use pre-computation only,

Setting-III: Use Caching only, and

Setting-IV: Use both pre-computation and caching.

Note that in our experiments, we do not consider cache replacement. This is a topic for future research.

3.5.5 Experimental Results

In the following, we first report two groups of experimental results. The first group, (*Group-1*), assumes that queries are issued in a random fashion. Specifically, we randomly

Batch Size	Setting-I			Setting-II		Setting-III		Setting-IV	
	SQ	LQ	GQ	LQ	GQ	LQ	GQ	LQ	GQ
2	391	362	234	288	207	160	149	149	125
3	587	462	254	398	231	239	224	224	162
4	783	607	305	502	274	319	299	299	192
6	1175	798	339	684	316	479	448	448	232

Table 3.9: Group-I Results on Synthetic (Quest) Datasets (All Execution Times in Seconds)

generate 24 queries from the query templates, and put them in the query queue. Our system will evaluate them in a batch fashion, where the batch size varies from 2 to 6. The second group, (*Group-2*), emulates a *mining session*. Each mining session is defined as a sequence of queries with the same query template but different thresholds. This simulates the situation in which a user issues a sequence of related queries, in order to find the desired results. Specifically, we randomly pick 24 query templates, and then randomly generate 6 queries from each template. In our experiment, we vary the batch size to evaluate the total of 144 queries generated in this fashion. Each batch contains 2, 3, 4, or 6 queries from different mining sessions.

Tables 3.9 and 3.10 show the *Group-1* experimental results. Tables 3.11 and 3.12 show the *Group-2* experimental results. Each table contains four different experimental settings: Setting-I, Setting-II, Setting-III, and Setting-IV, as described above. The number in the table represents the average evaluation time for each batch of queries. Note that in each of these these tables, for pre-computation, we select the frequent itemsets with support level 0.5% for the Quest datasets, and with support level 60% for the IPUMS datasets.

From these tables, we can see that GQ (global query plan) always performs better than LQ (local query plan). In the Setting-I (no pre-computation and caching), compared with

Batch Size	Setting-I			Setting-II		Setting-III		Setting-IV	
	SQ	LQ	GQ	LQ	GQ	LQ	GQ	LQ	GQ
2	83	71	58	44	40	29	25	24	21
3	126	101	80	66	59	43	32	36	26
4	167	112	79	71	65	58	40	47	41
6	250	171	109	121	88	88	55	72	48

Table 3.10: Group-I Results on Real (IPUMS) Datasets (All Execution Times in Seconds)

Batch Size	Setting-I			Setting-II		Setting-III		Setting-IV	
	SQ	LQ	GQ	LQ	GQ	LQ	GQ	LQ	GQ
2	412	365	263	273	202	84	53	81	51
3	619	523	301	389	237	126	77	120	75
4	825	638	333	497	250	168	90	163	87
6	1238	815	364	662	282	251	141	248	135

Table 3.11: Group-II Results on Synthetic (Quest) Datasets: (All Execution Times in Seconds)

Batch Size	Setting-I			Setting-II		Setting-III		Setting-IV	
	SQ	LQ	GQ	LQ	GQ	LQ	GQ	LQ	GQ
2	88	76	69	57	52	13	11	10	10
3	132	98	89	78	72	20	17	16	14
4	175	118	102	93	82	26	20	22	18
6	260	146	123	126	111	39	29	32	29

Table 3.12: Group-II Results from Real (IPUMS) Datasets (All Execution Times in Seconds)

SQ, the average speedups of LQ for all batch size in Tables 3.9, 3.10, 3.11, and 3.12 are 1.3, 1.3, 1.3, and 1.4, respectively. GQ gains an average speedup of 2.5, 1.9, 2.4, and 1.7, respectively. Also, as the batch size becomes larger, the gains from GQ and LQ also become larger. For example, when the batch size is 6 in Table 3.9, the speedups of LQ and GQ are 1.5 and 3.5, respectively. This is because with a larger number of queries in a batch, more common computations can be removed. This observation also validates the effectiveness of our methods to optimize multiple queries.

From the experimental results, we can see that pre-computation and caching also help reduce the evaluation costs. In our experiments, Setting-IV which combines pre-computation and caching is always the best. Setting-III (purely caching) is also quite effective, and delivers a speedup quite close to Setting-IV. Compared with Setting-I (no caching and pre-computation), Setting-II (Pre-computation) achieves an average speedup of 1.2, 1.4, 1.3,and 1.2, in Tables 3.9, 3.10, 3.11, and 3.12, respectively; The gains from the Setting-III amount to a factor of 1.8, 2.2, 3.8,and 5.0, respectively. Finally, the Setting-IV achieves the highest gains, with an average speedup of 1.9, 2.6, 4.0, and 5.9, respectively.

In the Setting-IV, caching and pre-computation maximize the gains for the both local and global query plans. Specifically, compared with SQ in the Setting-I, the average speedups of LQ in the Setting-IV are 2.6, 3.5, 5.1, and 8.3, in the Tables 3.9, 3.10, 3.11, and 3.12, respectively. GQ obtains an average speedup of 4.0, 4.5, 8.8, and 9.2, respectively.

An interesting property of caching is if there is no cache replacement, as is the case in our system, it reduces the average query evaluation time as more queries are being evaluated. Table 3.13 shows this caching effect. Here, global query plans are used. Each row of the table corresponds to a different batch size, ranging from 1 to 6. The columns in the

Batch Size	24 Queries	48 Queries	96 Queries	144 Queries
1	13	11	10	6
2	25	21	17	13
3	32	33	26	18
4	40	38	31	24
6	55	52	44	34

Table 3.13: Caching Effects: IPUMS(in Seconds)

table correspond to the number of queries being evaluated. We issued four sets of queries, with a total of 24, 48, 96, and 144 queries, respectively, to the query queue in our system. We can see that as more queries are processed by the system which is using the cache, the average of the batch processing time is reduced. Specifically, the average evaluation time has reduced from 9.2 seconds per query when there are 24 queries, to only 5.0 seconds per query when there are a total of 144 queries.

3.6 Related Work

This section compares our work with related research efforts.

A number of constraint frequent itemset mining algorithms have been developed, with the goal of using additional conditions and pruning the search space [16, 68, 77, 86, 98]. More recently, Yan *et. al* have studied the use of connectivity constraints to mine frequent graphs [33]. However, these algorithms cannot efficiently answer our target class of queries, since the conditions in our queries correspond to a set of (in)frequent patterns. Moreover, they have not considered the multiple query optimization problem.

Raedt and his colleagues have studied the generalized inductive query evaluation problem [66, 69, 88]. Although their queries target multiple datasets, they focus on the algorithmic aspects to apply version space tree and answer the queries with the generalized monotone and anti-monotone predicates. In comparison, we are interested in answering queries involving frequency predicates more efficiently.

Our research is also different from the work on *Query flocks* [101]. While they target complex query conditions, they allow only a single predicate involving frequency, and on a single dataset. The work on multi-relational data mining [13, 30, 87, 111] has focused on designing efficient algorithms to mine a single dataset materialized as a multi-relation in a database system.

A number of researchers have developed techniques for mining the difference or *contrast sets* between the datasets [12, 29, 106]. Their goal is to develop efficient algorithms for finding such a difference, and they have primarily focused on analyzing two datasets at a time. In comparison, we have provided a general framework for allowing the users to compare and analyze the patterns in multiple datasets.

As discussed in Section 5.1, some efforts have been made toward addressing the issues arising from sequence of queries and multiple simultaneous queries in mining environments. Nag *et al.* have studied how a knowledgeable cache can be used to help perform interactive discovery of association rules [75]. Hipp and Guntzer have proposed to use pre-computation of frequent itemsets of certain support levels to answer constraint itemset mining queries [51]. Goethals and Bussche have developed methods to support an interactive data mining session [40]. Compared with our work, these efforts have not addressed both of the issues, sequence of queries and multiple simultaneous queries, together, and the knowledgeable cache is restricted to simple data mining queries.

Multiple query optimization has been widely studied in database systems [96, 82, 91, 97]. Here, the focus has been on finding efficient query plans by dealing with the trade-offs between materialization and re-computation of common subexpressions. Zhao *et al.* have studied simultaneous optimization of a restricted kind of queries, called *multi-dimensional* queries [121]. The main differences between their study and our approach is that we assume that common computations will always be materialized, and we have developed an efficient way to detect and utilize such common computations.

Andrade *et al* have studied how to simultaneously optimize a group of related scientific data processing queries [10]. However, their methods are mainly based on the spatial properties of the queries and cannot applied to the mining tasks we have focused on.

3.7 Conclusions

The work presented in this chapter is driven by the need to efficiently process a large number of data mining queries, which are being issued by a number of users. To speedup the evaluation of queries in such a scenario, we need to not only evaluate each single query efficiently, but also need to optimize multiple queries simultaneously. Furthermore, we need to be able to utilize mining results from past queries in a systematic fashion.

In this chapter, we have presented a novel system architecture to deal with such a query intensive environment. We have proposed new algorithms to perform multiple-query optimization for frequent pattern mining queries which involve multiple datasets. We also designed a knowledgeable cache which can store the past query results from queries, and enable the use of these results to further optimize multiple queries. Finally, we have implemented and evaluated our system with both real and synthetic datasets. Our experimental results have demonstrated a speedup of up to a factor of 9.

CHAPTER 4

AN ALGORITHM FOR IN-CORE FREQUENT ITEMSET MINING ON STREAMING DATA

4.1 Introduction

Frequent itemset mining is a core data mining operation and has been extensively studied over the last decade [3, 41, 46, 48, 115, 122]. Algorithms for frequent itemset mining form the basis for algorithms for a number of other mining problems, including association mining, correlations mining, and mining sequential and emerging patterns [48].

Algorithms for frequent itemset mining have typically been developed for datasets stored in persistent storage and involve two or more passes over the dataset. Recently, there has been much interest in data arriving in the form of continuous and infinite data streams. In a streaming environment, a mining algorithm must take only a single pass over the data [11]. Such algorithms can only guarantee an approximate result.

In this chapter, we present a new approach for frequent itemset mining. Our work has two main contributions:

In-core Mining in Streaming Environment: We present a single pass algorithm for frequent itemset mining in a streaming environment. Our algorithm has provable deterministic bounds on accuracy. Unlike the only other existing work in this area that we are

familiar with [71], our algorithm does not require any out-of-core summary structure. We believe that this is a very desirable property, since stream mining algorithms may need to be executed in small and mobile devices, which do not have attached disks for storing an out-of-core summary structure.

Memory Efficient Accurate Mining: A key limitation of the existing work on frequent itemset mining has been the high memory requirements when the number of distinct items is large and/or the support level desired is quite low. Our single pass algorithm has a property that it does not produce *false negatives*, i.e., all frequent itemsets with desired support level are reported. The false positives reported by our algorithm can be easily removed through a second pass on the dataset. Our two pass algorithm provides high memory efficiency, while not compromising accuracy in any way.

Our work derives from the recent work by Karp *et al.* on determining frequent items (or 1-itemsets) [65]. They present a two pass algorithm for this purpose, which requires only $(1/\theta)$ memory, where θ is the desired support or frequency level. Their first pass computes a superset of frequent items, and the second pass eliminates any false positives. Our work addresses three major challenges in applying their ideas for frequent itemset mining in a streaming environment. First, we have developed a method for finding frequent k-itemsets, while still keeping the memory requirements limited. Second, we have developed a way to have a bound on the superset computed after the first pass. Third, we have developed a new data structure and a number of other implementation optimizations to support efficient execution.

We have carried out a detailed evaluation using both synthetic and real datasets. Our results can be summarized as follows.

- Our one pass algorithm is very accurate in practice.

- Our algorithm is very memory efficient. For example, using the T10.I4.N10K dataset and a support level of 1%, we can consistently handle 4 million to 20 million transactions with less than 2.5 MB main memory. In comparison, Manku and Motwani's algorithm [71] requires an out-of-core data-structures on top of a 44 MB buffer to process 1 million transactions.

- The algorithm can handle large number of distinct items and small support levels using a reasonable amount of memory. For example, a dataset with 100,000 distinct items and a support level of 0.05% could be handled with less than 200 MB main memory, a factor of 5 improvement over apriori.

Last year, a workshop on frequent itemset implementations was organized and a detailed evaluation of algorithms was carried out [41]. The focus in this workshop was on in-core datasets. In comparison, we are focusing on streaming data and the cases where large number of distinct itemsets or very low support levels can lead to high memory requirements for most algorithms.

The rest of the chapter is organized as follows. In Section 4.2, we introduce the method from Karp *et al.* to find frequent items, and explain the basic ideas to extend this method for mining frequent itemsets. In Section 4.3, we present our new algorithm and its theoretical properties. Implementation and detailed experimental evaluation is discussed in Section 4.4. We compare our work with related research efforts in Section 4.5 and conclude in Section 4.6.

4.2 Basic Ideas

This section describes the basic ideas that lead to our new algorithm. Initially, we discuss a new approach for finding frequent items from Karp *et al.* [65]. We then discuss

the challenges in extending this idea to frequent itemset mining, and finally outline our ideas for addressing these issues.

4.2.1 Finding Frequent Items

Our work is derived from the recent work by Karp, Papadimitriou and Shenker on finding frequent elements (or 1-itemset) [65]. Formally, given a sequence of length N and a threshold θ ($0 < \theta < 1$), the goal of their work is to determine the elements that occur with frequency greater than $N\theta$.

A trivial algorithm for this will involve counting the frequency of all distinct elements, and checking if any of them has the desired frequency. If there are n distinct elements, this will require $O(n)$ memory.

Their approach requires only $O(1/\theta)$ memory. Their approach can be viewed as a generalization of the following simple algorithm for finding the *majority element* in a sequence. A majority element is an element that appears more than half the time in an entire sequence. We find two distinct elements and eliminate them from the sequence. We repeat this process until only one distinct element remains in the sequence. If a majority element exists in the sequence, it will be left after this elimination. At the same time, any element remaining in the sequence is not necessarily the majority element. We can take another pass over the original sequence and check if the frequency of the remaining element is greater than $N/2$.

The idea can be generalized to an arbitrary θ. We can proceed as follows. We pick any $1/\theta$ distinct elements in the sequence and eliminate them together. This can be repeated until no more than $1/\theta$ distinct elements remain in the sequence. It can be claimed that any element appearing more than $N\theta$ times will be left in the sequence. The reason is that the elimination can only be performed at most $N/(1/\theta) = N\theta$ times. During each

```
FindingFrequentItems(Sequence S, θ)
 global Set P; // Set of Potentially
 P ← ∅;          // Frequent Items
 foreach (s ∈ S) // each item in S
      if s ∈ P
           s.count++;
        else
           P ← {s} ∪ P;
           s.count = 1;
           if |P| ≥ ⌈1/θ⌉
                foreach (p ∈ P)
                     p.count − −;
                     if p.count = 0
                          P ← P − {p};
 Output(P);
```

Figure 4.1: Karp *et al.* Algorithm for Frequent Items

such elimination, any distinct element is removed at most once. Hence, for each distinct element, the total number of eliminations during the entire process is at most $N\theta$. Any element appearing more than $N\theta$ times will remain in the sequence. Note, however, the elements left in the sequence do not necessarily appear with frequency greater than $N\theta$. Thus, this approach will provide a superset of the elements which occur more than $N\theta$ times.

Such processing can be performed to take only a single pass on the sequence, as we show in Figure 4.1. $\mathcal{P}$ is the set of potentially frequent items. We maintain a $count$ for each item in the set $\mathcal{P}$. This set is initially empty. As we process a new item from a sequence, we check if it is in the set $\mathcal{P}$. If yes, its count is incremented, otherwise, it is inserted with a count of 1. When the size of the set $\mathcal{P}$ becomes larger than $\lceil 1/\theta \rceil$, we decrement the

count of each item in $\mathcal{P}$, and eliminate any item whose count has now become 0. This processing is equivalent to the eliminations we described earlier. Note that this algorithm requires only $\Omega(1/\theta)$ space. It computes a superset of frequent items. To find the precise set of frequent items, another pass can be taken on the sequence, and the frequency of all remaining elements can be counted.

4.2.2 Issues In Frequent Itemset Mining

In this chapter, we build a frequent itemset mining algorithm using the above basic idea. There are three main challenges when we apply this idea to mining frequent itemsets, which we summarize below.

1. *Dealing with Transaction Sequences:* The algorithm from Karp *et al.* assumes that a sequence is comprised of elements, i.e., each transaction in the sequence only contains one-items. In frequent itemset mining, each transaction has a number of items, and the length of every transaction can also be different.

2. *Dealing with k-itemsets:* Karp *et al.*'s algorithm only finds the frequent items, or 1-itemsets. In a frequent itemset mining algorithm, we need to find all k-itemsets, $k \geq 1$, in a single pass.

   Note that their algorithm can be directly extended to find i-itemsets in the case where each transaction has a fixed length, l. This can be done by eliminating a group of $(1/\theta) \times \binom{l}{i}$ different i-itemsets together. This, however, requires $\Omega((1/\theta) \times \binom{l}{i})$ space, which becomes extremely high when l and i are large. Furthermore, in our problem, we have to find all i-itemsets, $i \geq 1$, in a single pass.

3. *Providing an Accuracy Bound:* Karp *et al.*'s algorithm can provably find a superset of the frequent items. However, no accuracy bound is provided for the item(set)s in the superset, which we call the potential frequent item(set)s. For example, even if an item appears just a single time, it can still possibly appear in the superset reported by the algorithm. In frequent itemset mining, we will like to improve above result, and provide a bound on the frequency of the itemsets that are reported by the algorithm.

4.2.3 Key Ideas

We now outline how we can address the three challenges we listed above.

Dealing with k-itemsets in a Stream of Transactions: Compared with the problem of finding frequent items, the challenges in finding frequent itemsets from a transaction sequence mainly arise due to the large number of potential frequent itemsets. This also results in high memory costs. As we stated previously, a direct application of the idea from Karp *et al.* will require $\Omega((1/\theta) \times \binom{l}{i})$ space to find potential frequent i-itemsets, where l is the length of each transaction. This approach is prohibitively expensive when l and i are large, but can be feasible when i is small, such as 2 or 3.

Recall that most of the existing work on frequent itemset mining uses the *apriori* property [3], i.e., an i-itemset can be frequent only if all subsets of this itemset are frequent. One of the drawbacks of this approach has been the large number of 2-itemsets, especially when the number of distinct items is large, and θ is small.

Our idea is to use a *hybrid* approach to mine frequent itemsets from a transaction stream. We use the idea from Karp *et al.* to determine the potential frequent 2-itemsets. Then, we use the set of potential frequent 2-itemsets and the apriori property to generate the potential

i-itemsets, for $i > 2$. This approach finds a set of potential frequent itemsets, which is guaranteed to contain all the *true* frequent itemsets, in a single pass of the stream.

Also, if a second pass of the data stream is allowed, we can eliminate all the *false* frequent itemsets from our result set. The second pass is very easy to implement, and in the rest of our discussion, we will only focus on the first pass of our algorithm.

Bounding False Positives: In order to have a accuracy bound, we propose the following criteria for the reported potential frequent itemsets after the first pass. Besides reporting all items or itemsets that occur with frequency more than $N\theta$, we want to report only the items or itemsets which appear with frequency at least $N\theta(1-\epsilon)$, where $0 < \epsilon \leq 1$. This criteria is similar to the one proposed by Manku and Motwani [71].

We can achieve this goal by modifying the algorithm as shown in Figure 4.2. In the *first* step, we invoke the algorithm from Karp *et al.* with the frequency level $\theta\epsilon$. This will report a superset of items occurring with frequency more than $N\theta\epsilon$. We also record the number of eliminations, c, that occur in this step. Clearly, c is bounded by $N\theta\epsilon$. In the *second* step, we remove all items whose reported frequency is less than $N\theta - c \geq N\theta(1-\epsilon)$.

We have two claims about the above process: 1) it reports all items that occur with frequency more than $N\theta$, and 2) it only reports items which appear with frequency more than $N\theta(1-\epsilon)$. The reason for this is as follows. Consider any element that appears with frequency $N\theta$. After the first step, it will be reported in the superset with a frequency greater than $c, c \leq N\theta\epsilon$. Therefore, it will remain in the set after the second step also. Similarly, consider any item that appears with frequency less than $N\theta(1-\epsilon)$. If this item is present in the superset reported after the first step, it will be removed during the second step since $N\theta - c \geq N\theta(1-\epsilon)$. This idea can be used for frequent itemset mining also.

```
FindingFrequentItemsBounded(Sequence S, θ, ε)
 global Set P;
 P ← ∅;
 c ← 0; // Number of Elimination
 foreach (s ∈ S)
      if s ∈ P
           s.count++;
        else
           P ← {s} ∪ P;
           s.count = 1;
           if |P| ≥ ⌈1/(θε)⌉
                c++; // Count Eliminations
                foreach (p ∈ P)
                     p.count--;
                     if p.count = 0
                          P ← P − {p};
 foreach (p ∈ P)
      if p.count ≤ (Nθ − c)
           P ← P − {p};
 Output(P);
```

Figure 4.2: Improving Algorithm with An Accuracy Bound

In the next Section, we introduce our algorithm for mining frequent itemsets from streaming data based on the above two ideas.

4.3 Algorithm

In this section, we introduce our new algorithm in three steps. In Subsection 4.3.1, we describe an algorithm for mining frequent itemsets from a data stream, which assumes that each transaction has the same length. In Subsection 4.3.2, we extend this algorithm to provide an accuracy bound on the potential frequent itemsets computed after one pass.

In Subsection 4.3.3, we further extend the algorithm to deal with transactions of variable length.

Before detailing each algorithm, we first introduce some terminology. We are mining a stream of transactions $\mathcal{D}$. Each transaction t in this stream comprises a set of *items*, and has the length $|t|$. Let the number of transactions in $\mathcal{D}$ be $|\mathcal{D}|$. Each algorithm takes the support level θ as one parameter. An itemset in $\mathcal{D}$ to be considered frequent should occur more than $\theta|\mathcal{D}|$ times.

To store and manipulate the candidate frequent itemsets during any stage of every algorithm, a lattice $\mathcal{L}$ is maintained.

$$\mathcal{L} \;=\; \mathcal{L}_1 \cup \mathcal{L}_2 \cup \ldots \cup \mathcal{L}_k$$

where, k is largest frequent itemset, and $\mathcal{L}_i, 1 \leq i \leq k$ comprises the potential frequent i-itemsets. Note that in mining frequent itemsets, the size of the set $\mathcal{L}_1$, which is bound by the number of distinct items in the dataset, is typically not very large. Therefore, in order to simplify our discussion, we will not consider $\mathcal{L}_1$ in the following algorithms, and assume we can find the exact frequent 1-itemsets in the stream $\mathcal{D}$. Also, we will directly extend the idea from Karp *et al.* to find the potential frequent 2-itemsets.

As we stated in the previous section, we deal with all k-itemsets, $k > 2$, using the apriori property. To facilitate this, we keep a buffer $\mathcal{T}$ in each algorithm to store the recently received transactions. The buffer will be accessed several times to find the potential frequent k-itemsets, $k > 2$.

4.3.1 Mining Frequent Itemsets from Fixed Length Transactions

The algorithm we present here mines frequent itemsets from a stream, under the assumption that each transaction has the same length $|t|$. The algorithm has two interleaved

```
StreamMining-Fixed(Stream D, θ )
 global Lattice L;
 local Buffer T;
 local Transaction t;
 L ← ∅; T ← ∅;
 f ← |t| * (|t| − 1)/2;
 foreach (t ∈ D)
      T ← T ∪ {t};
      Update(t, L, 2);
      if |L_2| ≥ ⌈1/θ⌉ · f
           ReducFreq(L, 2);
           {* Deal with k − itemsets, k > 2 *}
           i ← 2;
           while L_i ≠ ∅
                i++;
                foreach (t ∈ T)
                     Update(t, L, i);
                ReducFreq(L, i);
           T ← ∅;
 Output(L);
```

Figure 4.3: StreamMining-Fixed: Algorithm Assuming Fixed Length Transactions

phases. The *first* phase deals with 2-itemsets, and the *second* phase deals with k-itemsets, $k > 2$. The main algorithm and the associated subroutines are shown in Figures 4.3 and 5.4, respectively. Note that the two subroutines, *Update* and *ReducFreq*, are used by all the algorithms discussed in this section.

The first phase extends the Karp *et al.*'s algorithm to deal with 2-itemsets. As we stated previously, the algorithm maintains a buffer $\mathcal{T}$ which stores the recently received transactions. Initially, the buffer is empty. When a new transaction t arrives, we put it in $\mathcal{T}$. Next, we call the *Update* routine to increment counts in $\mathcal{L}_2$. This routine simply updates

```
Update(Transaction t, Lattice ℒ, i )
 for all i subsets s of t
      if s ∈ ℒ_i
           s.count++;
        else if i ≤ 2
           ℒ_i.insert(s);
        else if all i − 1 subsets of s ∈ L_{i−1}
           ℒ_i.insert(s);

ReducFreq(Lattice ℒ, i)
 foreach i itemsets s ∈ ℒ_i
      s.count − −;
      if s.count = 0
           ℒ_i.delete(s);
```

Figure 4.4: Subroutines Description

the count of 2-itemsets that are already in $\mathcal{L}_2$. Other 2-itemsets that are in the transaction t are inserted in the sets $\mathcal{L}_2$.

When the size of $\mathcal{L}_2$ is beyond the *threshold*, $\lceil 1/\theta \rceil f$, where f is the number of 2-itemsets per transaction, we call the procedure *ReducFreq* to reduce the count of each 2-itemsets in $\mathcal{L}_2$, and the itemsets whose count becomes zero are deleted. Invoking *ReducFreq* on $\mathcal{L}_2$ triggers the *second* phase.

The second phase of the algorithm deals with all k-itemsets, $k > 2$. This process is carried out level-wise, i.e, it proceeds from 3-itemsets to the largest potential frequent itemsets. For each transaction in the buffer $\mathcal{T}$, we enumerate all i-subsets. For any i-subset that is already in $\mathcal{L}$, the process will be the same as for a 2-itemset, i.e, we will simply increment the count. However, an i-subset that is not in $\mathcal{L}$ will be inserted in $\mathcal{L}$ only if all of its $i-1$ subsets are in $\mathcal{L}$ as well. Thus, we use the *apriori* property.

After updating i-itemsets in $\mathcal{L}$, we will invoke the *ReducFreq* routine. Thus, the itemsets whose count is only 1 will be deleted from the lattice. This procedure will continue until there are no frequent k-itemsets in $\mathcal{L}$. At the end of this, we clear the buffer, and start processing new transactions in the stream. This will restart the first phase of our algorithm to deal with 2-itemsets.

We next discuss the correctness and the memory costs of our algorithm. Let $\mathcal{L}_i^\theta$ be the set of frequent i-itemsets with support level θ in $\mathcal{D}$, and $\mathcal{L}_i$ be the set of potential frequent i-itemsets provided by this algorithm.

Theorem 1 *In using the algorithm StreamMining-Fixed on a set of transactions with a fixed length, for any* $k \geq 2$, $\mathcal{L}_k^\theta \subseteq \mathcal{L}_k$.

To prove this Theorem, we first consider the following lemma for 2-itemsets computed by the Algorithm StreamMining-Fixed.

Lemma 5 $\mathcal{L}_2^\theta \subseteq \mathcal{L}_2$.

Proof: The lemma directly follows the fact that *ReducFreq* is called at most $\lfloor \theta |\mathcal{D}| \rfloor$ times in the *foreach* loop. This can be observed from the following inequality

$$\lfloor \theta |\mathcal{D}| \rfloor \geq |\mathcal{D}| / \lceil 1/\theta \rceil$$

□

Proof:(Theorem 1): Now, we prove this theorem inductively. The base case, $k = 2$, has been shown in the Lemma 13. Assume that the property holds true for $\mathcal{L}_{k-1}$. To show that it is valid for $\mathcal{L}_k$, we take two steps. First, assume that we do not check for subsets. Then, any k-itemsets which appears in the buffer will be inserted in the lattice. In this case, it is easy to see that the property holds for $\mathcal{L}_k$. Now let us see why checking for subsets

does not change the final results. Assume there is an k-itemset s appearing in a transaction t, and one of its $k-1$ subset is not in $\mathcal{L}_{k-1}$. We can deduce that s must appear only once in the buffer, and it is not included in any other transactions besides t. Thus, we can see that s, if included in the lattice, would have been eliminated by the next invocation of *ReducFreq*. □

Lemma 6 *In using the algorithm StreamMining-Fixed on a set of transactions with a fixed length, the size of $\mathcal{L}_2$ is bounded by $(\lceil 1/\theta \rceil + 1)\binom{|t|}{2}$.*

Proof: This is based on the following observation. After processing each transaction, we will check if the size of $\mathcal{L}_2$ is larger than $(\lceil 1/\theta \rceil)\binom{|t|}{2}$. Also note that each transaction can add at most $\binom{|t|}{2}$ new 2-itemsets into the set $\mathcal{L}_2$. □

Theorem 1 implies that any frequent k-itemset is guaranteed to be in the output of our algorithm. Lemma 6 provides an estimate of the memory costs for $\mathcal{L}_2$.

4.3.2 Providing an Accuracy Bound

We now extend the algorithm from the previous subsection to provide a bound on the accuracy of the reported results. As described in Subsection 4.2.3, the bound is described by an user-defined parameter, ϵ, where $0 < \epsilon \leq 1$. Based on this parameter, the algorithm ensures that the frequent itemsets reported do occur more than $(1 - \epsilon)\theta|\mathcal{D}|$ times in the dataset.

The basic idea for achieving such a bound on frequent items computation was illustrated in Figure 4.2. We can extend this idea to finding frequent itemsets. Our new algorithm is described in Figure 4.5. Note that we still assume that each transaction has the same length.

This algorithm provides the new bound on accuracy in two steps. In the *first* step, we invoke the algorithm in Figure 4.3 with the frequency level $\theta\epsilon$. This will report a superset of

```
StreamMining-Bounded(Stream D, θ, ε )
 global Lattice L;
 local Buffer T;
 local Transaction t;
 L ← ∅; T ← ∅;
 f ← |t| * (|t| − 1)/2;
 c ← 0; // Number of ReducFreq Invocations
 foreach (t ∈ D)
      T ← T ∪ {t};
      Update(t, L, 1);
      Update(t, L, 2);
      if |L_2| ≥ ⌈1/θε⌉ · f
           ReducFreq(L, 2);
           c++;
           i ← 2;
           while L_i ≠ ∅
                i++;
                foreach (t ∈ T)
                     Update(t, L, i);
                ReducFreq(L, i);
           T ← ∅;
 foreach s ∈ L
      if s.count ≤ θ|D| − c
           L_i.delete(s);
 Output(L);
```

Figure 4.5: StreamMining-Bounded: Algorithm with a Bound on Accuracy

itemsets occurring with frequency more than $N\theta\epsilon$. We record the number of invocations of *ReducFreq*, c, in the first step. Clearly, c is bounded by $N\theta\epsilon$. In the *second* step, we remove all items whose reported frequency is less than $N\theta - c \geq N\theta(1-\epsilon)$. This is achieved by the last *foreach* loop.

The new algorithm has the following property: 1) if an itemset has frequency more than θ, it will be reported. 2) if an itemset is reported as a potential frequent itemset, it must have a frequency more than $\theta(1-\epsilon)$. Theorem 2 formally states this property.

Theorem 2 *In using the algorithm StreamMining-Bounded on a set of transactions with a fixed length, for any* $k \geq 2$, $\mathcal{L}_k^\theta \subseteq \mathcal{L}_k \subseteq \mathcal{L}_k^{(1-\epsilon)\theta}$.

To prove Theorem 2, we first prove the following lemmas for the algorithm StreamMining-Bounded.

Lemma 7 $\mathcal{L}_2^\theta \subseteq \mathcal{L}_2$.

Proof: The proof has two parts. First, we can see that *ReducFreq* is called at most $\lfloor \theta\epsilon|\mathcal{D}| \rfloor$ times in the *foreach* loop. Thus, any 2-itemset that appears more than $\theta\epsilon|\mathcal{D}|$ times will stay in the set $\mathcal{L}_2$. Then, after the *while* loop, the total invocations of *ReducFreq* will be at most $\theta|\mathcal{D}|$. Therefore, we have $\mathcal{L}_2^\theta \subseteq \mathcal{L}_2$.

Lemma 8 *For any* 2*-itemset* $s \in \mathcal{L}_2$, $s \in \mathcal{L}_2^{(1-\epsilon)\theta}$. *In other words,* s *will appear more than* $(1-\epsilon)\theta|\mathcal{D}|$ *times in* $\mathcal{D}$.

Proof: Note that *ReducFreq* is called c times, where $c \leq \theta\epsilon|\mathcal{D}|$. Suppose there is an itemset appearing with a frequency less than $(1-\epsilon)\theta|\mathcal{D}|$ in the stream $|\mathcal{D}|$. It will be removed from the $\mathcal{L}_2$ in the last *foreach* loop because $\theta|\mathcal{D}| - c \geq (1-\epsilon)\theta|\mathcal{D}|$. □

Putting Lemmas 7 and 8 together, we have following result. □

Lemma 9

$$\mathcal{L}_2^{\theta} \subseteq \mathcal{L}_2 \subseteq \mathcal{L}_2^{(1-\epsilon)\theta}$$

Proof:(Theorem 2) This follows from applying an induction similar to the one in the proof of Theorem 1, and using Lemma 9 as the base case. □

Note that the number of invocations of *ReducFreq*, c, is usually much smaller than $N\theta\epsilon$ after processing a data stream. Therefore, an interesting property of this approach is that it produces a very small number of false frequent itemsets, even with relatively large ϵ. The experiments in Section 4.4 also support this observation.

The following lemma claims that the memory cost of $\mathcal{L}_2$ is increased by a factor proportional to $1/\epsilon$.

Lemma 10 *In using the algorithm StreamMining-Bounded on a set of transactions with a fixed length, the size of* $\mathcal{L}_2$ *is bounded by* $(\lceil 1/\theta\epsilon \rceil + 1)\binom{|t|}{2}$.

4.3.3 Dealing with Variable Length Transactions

In this subsection, we present our final algorithm, which improves upon the algorithm from the previous subsection by dealing with variable length transactions. The algorithm is referred to as *StreamMining* and is illustrated in Figure 4.6.

When each transaction has a different length, the number of 2-itemsets in each transaction also becomes different. Therefore, we cannot simply maintain f, the number of 2-itemsets per transaction, as a constant. Instead, we maintain f as a weighted average of the number of 2-itemsets that each transaction processed so far. This weighted average is computed by giving higher weightage to the recent transactions. The details are shown in the pseudo-code for the routine *TwoItemsetPerTransaction*.

StreamMining(Stream $\mathcal{D}$, θ, ϵ)
global $Lattice\ \mathcal{L}$;
local $Buffer\ \mathcal{T}$;
local $Transaction\ t$;
$\mathcal{L} \leftarrow \emptyset$; $\mathcal{T} \leftarrow \emptyset$;
$f \leftarrow 0$; // $Average\ 2-itemset\ per\ transaction$
$c \leftarrow 0$;
foreach $(t \in \mathcal{D})$
    $\mathcal{T} \leftarrow \mathcal{T} \cup \{t\}$;
    $Update(t, \mathcal{L}, 1)$;
    $Update(t, \mathcal{L}, 2)$;
    $f \leftarrow TwoItemsetPerTransaction(t)$;
    if $|\mathcal{L}_2| \geq \lceil 1/\theta\epsilon \rceil \cdot f$
        $ReducFreq(\mathcal{L}, 2)$;
        $c++$;
        $i \leftarrow 2$;
        while $\mathcal{L}_i \neq \emptyset$
            $i++$;
            foreach $(t \in \mathcal{T})$
                $Update(t, \mathcal{L}, i)$;
            $ReducFreq(\mathcal{L}, i)$;
        $\mathcal{T} \leftarrow \emptyset$;
foreach $s \in \mathcal{L}$
    if $s.count \leq \theta|D| - c$
        $\mathcal{L}_i.delete(s)$;
$Output(\mathcal{L})$;

TwoItemsetPerTransaction(Transaction t)
global X; // $Number\ of\ 2\ Itemset$
global N; // $Number\ of\ Transactions$
local f;
$N++$;
$X \leftarrow X + \begin{pmatrix} |t| \\ 2 \end{pmatrix}$;
$f \leftarrow \lceil X/N \rceil$;
if $|\mathcal{L}_2| \geq \lceil 1/\theta\epsilon \rceil \cdot f$
    $N \leftarrow N - \lceil 1/\theta\epsilon \rceil$;
    $X \leftarrow X - \lceil 1/\theta\epsilon \rceil \cdot f$;
$return\ f$;

Figure 4.6: StreamMining: Final Algorithm

To motivate the need for taking such a weighted average, consider the natural alternative, which will be maintaining f as the average number of 2-itemsets that each transaction seen so far has. This will not work correctly. For example, suppose there are 3 transactions, which have the length 2, 2, and 3, respectively, and θ is 0.5. The first two transactions will have a total of two 2-itemsets, and the third one has 6 2-itemsets. We will preform an elimination when the number of different 2-itemsets is larger than or equal to $(1/\theta) \times f$. When the first two transactions arrive, an elimination will happen (assuming that the two 2-itemsets are different). When the third one arrives, the average number of 2-itemsets is less than 3, so another elimination will be performed. Unfortunately, a frequent 2-itemset that appears in both transactions 1 and 3 will be deleted in this way.

In our approach, the number of invocations of *ReducFreq*, c, is less than $|\mathcal{D}|(\theta\epsilon)$, where $|\mathcal{D}|$ is the number of transactions processed so far in the algorithm. Lemma 11 formalizes this.

Lemma 11 *$c < |\mathcal{D}|(\theta\epsilon)$ is an invariant in the algorithm StreamMining.*

Proof: This can be proved inductively. Initially, $c = 1$ after the condition $|\mathcal{L}_2| \geq \lceil 1/\theta\epsilon \rceil \times f$ become true, where f is simply the average 2-itemsets of the transactions processed so far. Clearly, we need at least $\lceil 1/\theta\epsilon \rceil$ transactions to achieve this condition.

Then, assuming $c \geq 1$ is true for the claim, we look at $c + 1$. Note that we maintain N to be the $|\mathcal{D}| - c \times \lceil 1/\theta\epsilon \rceil$ after the invocation of *ReducFreq*. Therefore, following this assumption, N is larger than 0 at this point. Further, we maintain $|\mathcal{L}_2|$ as the number of *different* 2-itemsets in the set $\mathcal{L}_2$, and let Y be the total number of 2-itemsets stored in $\mathcal{L}_2$, which counts the repetition of the same 2-itemsets. Clearly, $X \geq \mathrm{Y} \geq |\mathcal{L}_2|$. Thus, when $|\mathcal{L}_2| \geq \lceil 1/\theta\epsilon \rceil \cdot X/N$ become true, $N \geq \lceil 1/\theta\epsilon \rceil$. This suggests that $|\mathcal{D}| = N + c \times \lceil 1/\theta\epsilon \rceil > (c+1)\lceil 1/\theta\epsilon \rceil$. Therefore, $c + 1 < |\mathcal{D}|(\theta\epsilon)$.

Note that by using the Lemma 11, we can deduce that the property of the Theorem 2 still holds for mining a stream of transaction with variable transaction lengths. Formally,

Theorem 3 *In using the algorithm StreamMining on a stream of transactions with variable lengths, for any* $k \geq 2$, $\mathcal{L}_k^{\theta} \subseteq \mathcal{L}_k \subseteq \mathcal{L}_k^{(1-\epsilon)\theta}$.

An interesting property of our method is that in the situation where each transaction has the same length, our final algorithm, *StreamMining* will work in the same fashion as the algorithm previously shown in Figure 4.5.

Note, however, that unlike the case with fixed length transactions, the size of $\mathcal{L}_2$ cannot be bound by a closed formula. Also, in all the algorithms discussed in this section, the size of sets $\mathcal{L}_k$, $k > 2$ also cannot be bound in any way. Our algorithms use the apriori property to reduce their sizes. In the next section, we evaluate the memory cost of our algorithm experimentally.

4.4 Implementation and Experimental Results

In this section, we will first briefly introduce the important implementation issues, and then we will focus on evaluating our new algorithm using a number of synthetic and real datasets.

4.4.1 Implementation issues

One of the main difficulties to implement our algorithm is to maintain the set of potential frequent itemsets, $\mathcal{L}$, efficiently. As discussed in Section 4.3, our algorithm requires frequently invoking *Update* and *ReducFreq*. This means a data structure to support efficient insertion, deletion, and search operation. However, the traditional data structure, such as *prefix tree* and *hash tree*, cannot meet these requirements. We have developed a new data

structure, which we refer to as *TreeHash*. Essentially, this data structure stores a prefix tree using hash tables. It has the benefit of easy deletion that a hash table allows, but it is also compact like a prefix tree. We omit the details of this data structure, interested readers can look at [62]. Some other techniques are also used to speed-up of our algorithm, and are described in [62].

4.4.2 Experimental Evaluation

In our experiment, we are interested in a number of different aspects of our algorithm.

- Comparing the execution time and memory requirements of our one pass and two pass algorithm with those of apriori and fp-tree based algorithms.
- Evaluating the execution time and memory requirements of our new algorithms with increasing dataset size and decreasing support levels.
- Evaluating the accuracy of our algorithm with different levels of ϵ.
- Demonstrating the ability of our algorithm to handle very large number of distinct items and very low support levels.

For comparing our algorithm against the Apriori algorithm, we used a well-known public distribution from Borgelt [14]. Earlier versions of this code have been incorporated in a commercial data mining tool called Clementine. For comparisons with FP-tree based approach, the implementation we used is from Goethals [39]. All our experiments were conducted on a 933 MHz Pentium III machine with 512 MB main memory.

4.4.3 Synthetic Datasets

The synthetic datasets we used were generated using a tool from IBM [5]. Datasets generated from this tool have been widely used for evaluating frequent itemset and association mining implementations.

Initially, we focus on two datasets where conventional offline algorithm have performed well. We show that our algorithm can still be competitive, while allowing high accuracy on streaming data. Later, we show our algorithms ability to handle very large number of distinct itemsets and very low support levels.

The first dataset we used is T10I4.N10K. The number of distinct itemsets is 10,000, the average number of items per transaction is 10, and the average size of large itemsets is 4. We used three different versions of our algorithm. `Stream-e1` uses 1 as the value of ϵ and does not provide any theoretical bound on the accuracy. `Stream-e.75` uses .75 as the value of ϵ to provide a theoretical bound on the accuracy. `Stream+` is the two pass implementation that gives the accurate set of frequent itemsets and their frequency counts.

Figure 4.7 shows the execution times of apriori, fp-tree and our three versions as the support threshold is varied from 0.1% to 1.0%. The number of transactions is 12 million. Because of high memory requirements, fp-tree could not be executed with support levels lower than 0.4%. This limitation of the fp-tree appraoch has been identified by other experimental studies also [31]. Up to the support level of 0.4%, the execution times of all versions is quite similar. However, apriori's execution time increases rapidly when the support level is less than 0.4%. As expected, `Stream-e1` has the lowest execution time among all of our versions. The use of .75 as the value of ϵ increases the execution time by up to 25%. If a second pass is used, the total execution time is increased by up to 50%.

Figure 4.8 compares the memory requirements. Because the memory requirements of `Stream+` are identical to those of `Stream-e1`, this version is not shown separately in our memory requirements charts. The important property of our algorithm is that the memory requirements do not increase significantly as the support level is decreased.

Accuracy of an algorithm is defined as the fraction of reported frequent itemsets that are actually frequent. Obviously, the accuracy of apriori, fp-tree and `Stream+` is always 100%. With 12 million transactions, `Stream-e1` and `Stream-e.75` give accuracy of 100% with thresholds at 1%, .8%, .6%, and .4%. With thresholds of .2% amd .1%, `Stream-e1` has an accuracy of 95.8% and 97.8%, respectively. However, in both these cases, with .75 as the value of ϵ, the accuracy again becomes 100%.

Figures 4.9 and 4.10 examine the execution times as the dataset is increased. The threshold is kept at .4% and .1%, respectively. Because of the high memory requirements of fp-tree, our algorithm is only compared against apriori. When the support level is 0.1%, our algorithm is up to an order of magnitude faster. The relative difference is smaller when the support level is 0.4%, but even our two pass version is faster than apriori. Even as the dataset size is varied, our one pass algorithms always give an accuracy of 100% when the threshold is .4%. With the threshold at .1%, the accuracy of `Stream-e.75` is again 100% in all cases. The accuracy of `Stream-e1` varies between 94.3% and 98.6%.

Figure 4.11 focuses on memory requirements with support levels of .4% and .1%. At the support level of .4%, apriori's memory requirements are lower than our versions. However, with threshold at .1%, our versions require less than half the memory. Moreover, it is important to note that with 10,000 distinct items and a support level of .1%, the total memory requirements are only around 17 MB. Thus, our algorithm is well suited for mining streaming data using a small device with only a limited memory.

The second dataset we use is T15.I6.N10K. We repeated the same set of experiments using this dataset. The results are shown in Figures 4.12, 4.13, 4.14, 4.15, and 4.16, respectively. The key difference between this dataset and the previous dataset is the the length of each transaction and each frequent itemset is higher. Because our algorithm needs

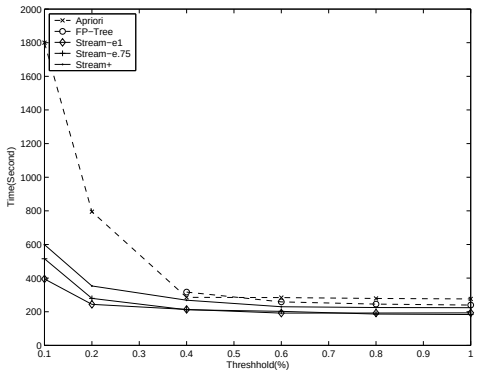


Figure 4.7: Execution Time with Changing Support Level (T10.I4.N10K Dataset)

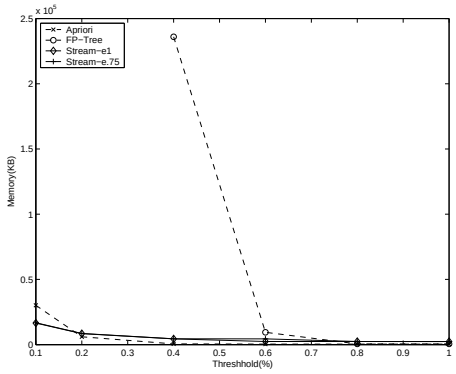


Figure 4.8: Memory Requirements with Changing Support Level (T10.I4.N10K Dataset)

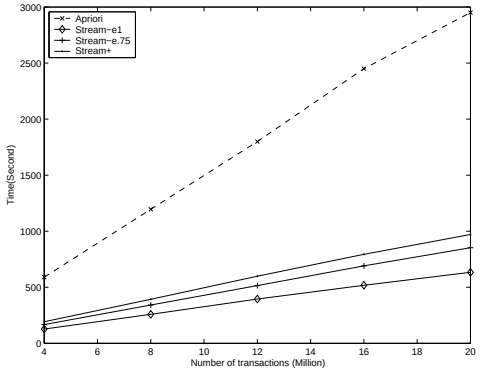


Figure 4.9: Execution Time with Increasing Dataset Size (threshold=0.1%, T10.I4.N10K Dataset)

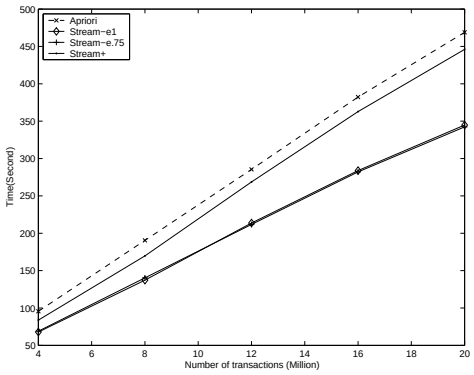


Figure 4.10: Execution Time with Increasing Dataset Size (threshold=0.4%, T10.I4.N10K Dataset)

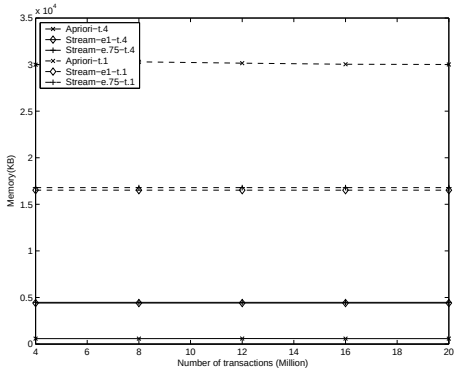


Figure 4.11: Memory Requirements with Increasing Dataset Size (T10.I4.N10K Dataset)

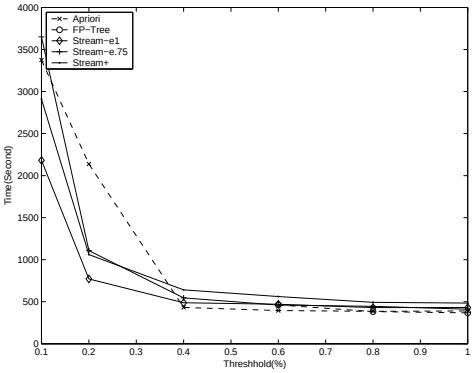


Figure 4.12: Execution Time with Changing Support Level (T15.I6.N10K Dataset)

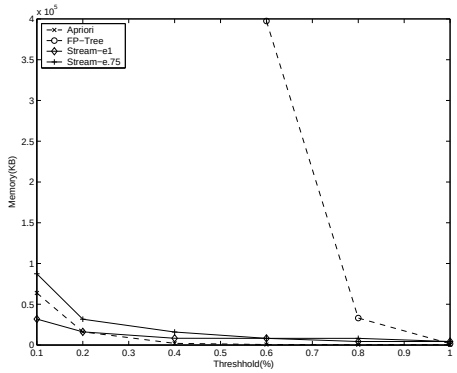


Figure 4.13: Memory Requirements with Changing Support Level (T15.I6.N10K Dataset)

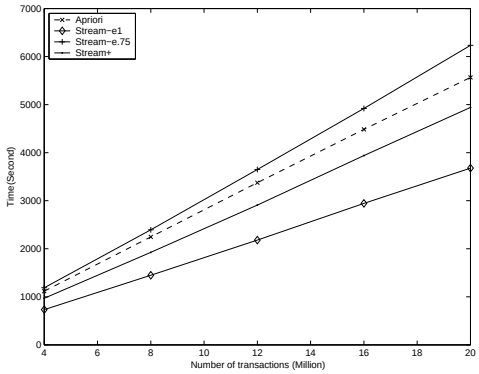


Figure 4.14: Execution Time with Increasing Dataset Size (threshold=0.1%, T15.I6.N10K Dataset)

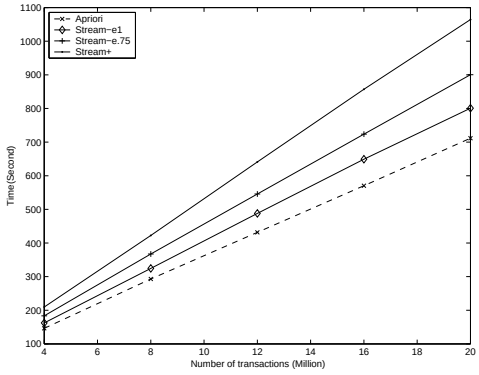


Figure 4.15: Execution Time with Increasing Dataset Size (threshold=0.4%, T15.I6.N10K) Dataset

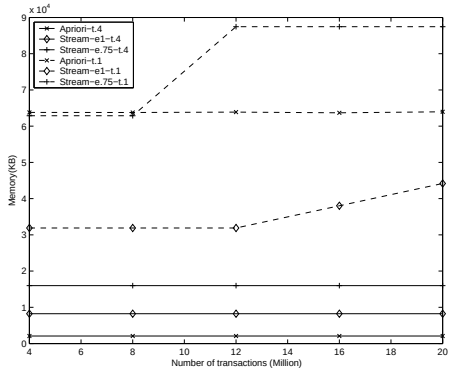


Figure 4.16: Memory Requirements with Increasing Dataset Size (T15.I6.N10K Dataset)

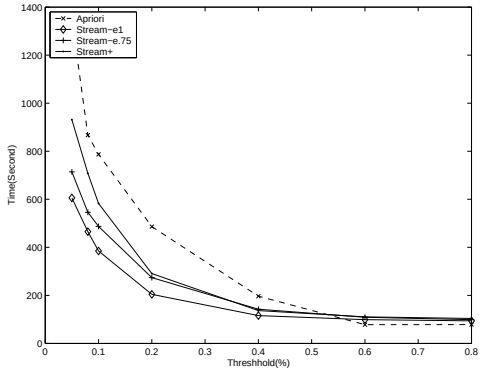


Figure 4.17: Execution Time with Changing Support Level (T25.I4.N100K Dataset)

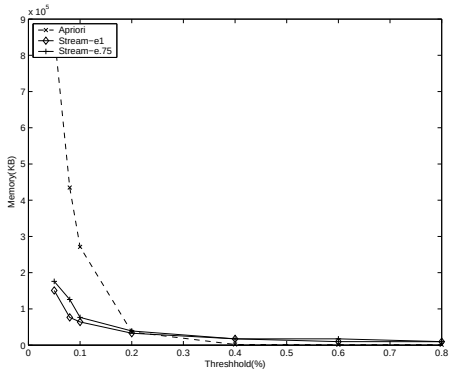


Figure 4.18: Memory Requirements with Changing Support Level (T25.I4.N100K Dataset)

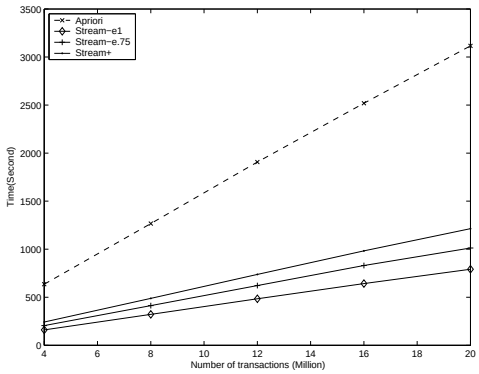


Figure 4.19: Execution Time with Increasing Dataset Size (threshold=0.08%, T10.I4.N10K Dataset)

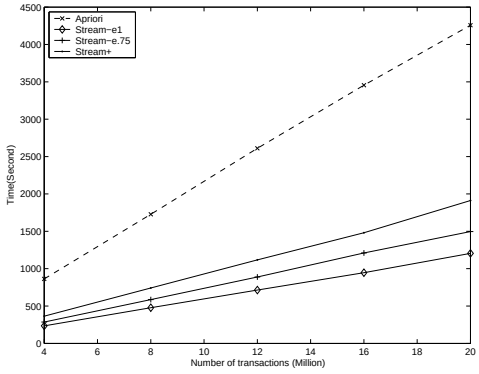


Figure 4.20: Execution Time with Increasing Dataset Size (threshold=0.05%, T10.I4.N10K Dataset)

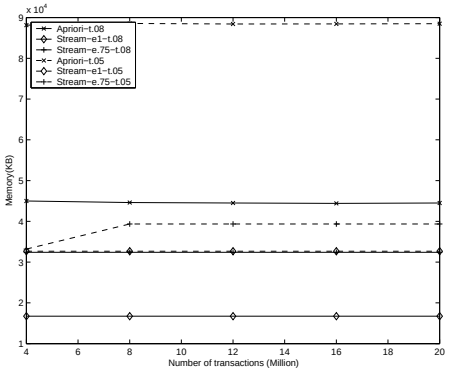


Figure 4.21: Memory Requirements with Increasing Dataset Size (T10.I4.N10K Dataset)

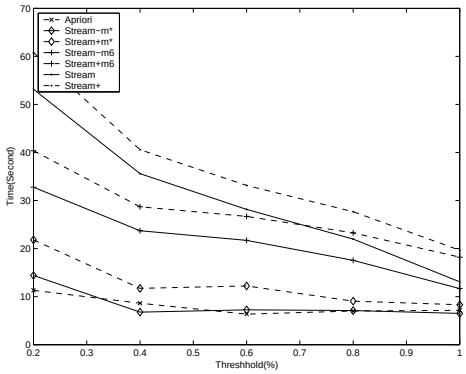


Figure 4.22: Execution Time with Changing Support Level (BMS-WebView-1 Dataset)

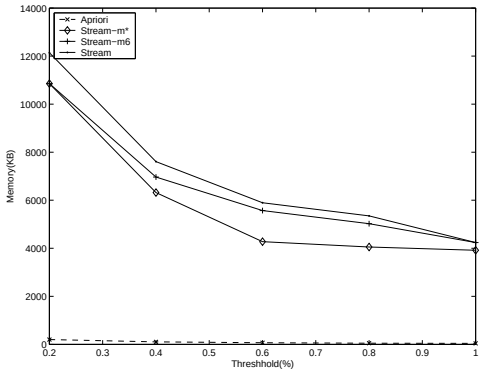


Figure 4.23: Memory Requirements with Changing Support Level (BMS-WebView-1 Dataset)

to generate fairly accurate results after one pass, its ability to prune large itemsets is limited. As a result, our algorithm does not always outperform apriori with this dataset. However, our algorithm does maintain very high accuracy of results after one pass on the dataset. With 4 million, 8 million, 12 million, 16 million, or 20 million transactions, and with support levels of 1%, .8%, .6%, or .4%, our `Stream-e1` always produces 100% accuracy. With support levels of .2% and .1%, the accuracy is still above 94%. The accuracy of Stream-e.75 is always above 99% with these support levels.

When the support level is .1%, `Stream-e1` is always significantly faster than apriori. `Stream-e.75` is also faster than apriori, but the difference is less significant. `Stream+` is actually slower than apriori. With the support level of .1%, `Stream-e1` also always requires less memory than apriori. When the threshold is .4%, apriori is faster than all of our versions.

As stated earlier, besides providing reasonably accurate results in one pass, the key benefit of our algorithm is its ability to handle very large number of distinct items and/or very low support levels. To demonstrate this, we first used the T25.I4.N100K dataset, which has 100,000 distinct items. The number of transactions was 12 million. Note that the size of each transaction is also quite large. Even in this case, the accuracy from `Stream-e1` is above 99.5% and the accuracy from `Stream-e.75` is above 99.8%. The execution times and memory requirements from this dataset are shown in Figures 4.17 and 4.18, respectively. With support levels below .4%, all of our versions are significantly faster than apriori. With support levels of .1% and .05%, the memory requirements are also drastically lower than those of apriori.

Next, we focus on the case when support levels are very low. The dataset we use is T10.I4.N10K. We consider support levels of .05% and .08%. The accuracy achieved is still

very good. `Stream-e1` has an accuracy of 97% or better, and `Stream-e.75` has an accuracy of 99.8% or better. The execution times are presented in Figures 4.19 and 4.20 and the memory requirements are shown in Figure 4.21. All of our versions are significantly better both in terms of execution time and memory requirements.

4.4.4 Real Dataset

The real dataset we use is the BMS-WebView-1 dataset which contains several months of click-stream data from one e-commerce website. A portion of it has been used in the KDD-Cup 2000 competition and also used by Zhang *et al.* [122] to evaluate traditional offline association mining algorithms.

The characteristics of the BMS-WebView-1 dataset are quite different from the IBM Quest synthetic datasets. The original dataset has 59,602 transactions and contains 497 distinct items. The maximum transaction size is 267, while the average transaction size is just 2.5. For our experiments, we duplicated and randomized the original dataset to obtain 1 million transactions.

Because of the small size of the dataset and the small number of distinct items, we did not expect to outperform apriori on this dataset. However, we have still compared the performance with apriori to show that the algorithm can give accurate results in one pass, and can still be competitive.

In our experiments, we use $\epsilon = 0.6$. Further, we provide another parameter m to represent the maximal frequent itemsets we are interested in. This is because if we have some additional knowledge about the length of the maximal frequent itemsets, the performance of our implementation can be improved. In this dataset, as the support level is 0.2%, 0.4%, 0.6%, 0.8% or 1%, the maximal frequent itemsets is 2, 3, 3, 4, and 6, respectively. For the

online checking optimization we had described earlier, the threshold we define is 10, i.e, two transactions in the buffer will not have a common subset which contains more than 10 items. Since we have only less than 500 distinct items, we maintain all of the 2-itemsets as an array in the main memory.

Figure 4.22 compares the execution time. `Stream-m*` refers to *StreamMining* with some knowledge of maximal frequent itemsets. For support level of 0.2%, we had $m = 4$, and for others, we had $m = 3$. `Stream-m6` refers to the version using $m = 6$ in all cases. `Stream` refers to *StreamMining* having no knowledge about the maximal frequent itemsets. `Stream+m*`, `Stream+m6` and `Stream+` refer to the corresponding two pass versions.

The three versions have very similar results for accuracy. For threshold levels between 1% and 0.4%, they achieve 100% accuracy. For the threshold of 0.2%, the accuracy is nearly 99%.

We can see that the performance of `Stream-m*` is quite similar to apriori. For the `Stream-m6` and `Stream`, we can see as the additional information on maximal frequent itemsets is reduced, the algorithm performance becomes less competitive. For the two-pass algorithm, we can see that the second pass just adds a fairly small and constant time.

Figure 4.23 compares the memory cost of apriori and *StreamMining*. Because the number of frequent itemset is relatively small, the memory cost of apriori is very low. Although the cost of `StreamMining` is almost two orders higher than that of apriori, we can see the absolute memory cost is just 11MB. It comes mostly from the initial hash table and the 2-itemset array.

4.5 Related Work

As stated through-out, our work has two implications. First, we have presented a one pass algorithm for approximate frequent itemset mining on streaming data. Second, we have presented a more memory efficient algorithm for two pass accurate frequent itemset mining. In this section, we compare our work with related research efforts in each of the areas.

Processing of streaming data has received a lot of attention within the last couple of years [11, 27, 35, 38]. Within the area of data mining, significant work has been done on the problem of classification [28, 56] and clustering [43]. More recently, attention has been paid to the area of frequent itemset mining [36, 71].

The work closest to our work on handling streaming data is by Manku and Motwani [71]. They have also presented a one pass algorithm that does not allow false negatives, and has a provable bound on false positives. They achieve this through a very different approach, called *lossy counting*. The differences in the two approaches are in space requirements. For finding frequent items, the approach we use takes $O(1/\theta)$ space. Their approach requires $O((1/\theta)log(\theta N))$ space, where θ is the desired support level and N is the length of the stream. Therefore, for frequent itemset mining, they require an out-of-core data structure. In comparison, we do not need any such structure. On the T10.I4.N10K dataset used in their chapter as well, we see that with 1 million transactions and a support level of 1%, their algorithm requires an out-of-core data-structures on top of even a 44 MB buffer. For datasets ranging from 4 million to 20 million transactions, our algorithm only requires 2.5 MB main memory based summary. In addition, we believe that there a number of advantages of an algorithm that does not require an out-of-core summary structure. Mining on streaming data may often be performed in mobile, hand-held,

or sensor devices, where processors do not have attached disks. It is also well known that additional disk activity increases the power requirements, and battery life is an important issue in mobile, hand-held, or sensor devices. Also, while their algorithm is shown to be currently computation-bound, the disparity between processor speeds and disk speeds continues to grow rapidly. Thus, we can expect a clear advantage from an algorithm that does not require frequent disk accesses.

Giannella *et al.* have developed a technique for dynamically updating frequent patterns on streaming data [36]. They create a variation of FP-tree, called FP-stream, for time-sensitive mining of frequent patterns. Because this approach gives additional weightage to recent transactions, it can efficiently answer time-sensitive queries, which we do not consider. However, for queries involving queries on an entire data stream, their approach is not efficient.

As our experimental results have shown, the memory requirements of our approach are significantly lower than those of FP-tree. However, we have not considered time-sensitive queries.

Recently, Yu and his colleagues proposed a new approach to mine frequent itemsets, which allows both *false negatives* and *false positives* [113]. Their approach is based on the Chernoff Bound. In comparison, our algorithm finds a superset of frequent itemsets, and therefore, only allows false positives. Further, if a second pass is allowed, our algorithm can also eliminate false positives.

Now, we compare our work with accurate frequent itemset mining algorithms, which require two or more passes. The classical work in this area is the Apriori algorithm [5, 3]. The basic idea in this algorithm has been extended by several others [117, 81]. Our experimental comparison has shown advantages of our approach when the number of distinct

itemsets is large and/or the support level desired is very low. Several algorithms since then have required only two passes. This includes the FP-tree based approach by Han and co-workers [48]. Again, as our experimental results have shown, the memory requirements for maintaining the frequent patterns summary increase rapidly when the support levels are low. Other two pass algorithms for association mining include those from Savarese *et al.* [94] and Toivonen [100]. In each of these cases, the two pass algorithm does not extend to a one pass algorithm with any guarantees on accuracy. Hidber has developed a technique which guarantees that the results after the first pass do not include any false negatives, but produces a large number of false positives [50]. A detailed comparison of frequent itemset mining algorithms has been done by Goethals and Zaki, as part of the FIMI workshop [41]. Our focus has been on the cases when the number of distinct itemsets is very large or the support levels are very low, which were not the emphasis of their evaluation.

4.6 Conclusions

In this chapter, we have developed a new approach for frequent itemset mining. We have developed a new one pass algorithm for streaming environment, which has deterministic bounds on the accuracy. Particularly, it is the first such algorithm which does not require any out-of-core memory structure and is very memory efficient in practice. We have developed a new data structure and several other optimizations to support this algorithm.

Our detailed experimental evaluation has shown the following. First, our one pass algorithm is very accurate in practice. Though a tighter theoretical bound on accuracy can be achieved by increasing memory requirements, it was not really required in practice. Second, the memory efficiency of our one and two pass algorithms allowed us to deal with large number of distinct items and/or very low support levels. For other cases, where

traditional multi-pass approaches have worked well in the past, our algorithms are still quite competitive. One exception is datasets with the average length of an itemset is quite large. In such case, some additional knowledge of maximal frequent itemsets helps efficiency of our algorithms.

CHAPTER 5

DISCOVERING FREQUENT TOPOLOGICAL STRUCTURES FROM GRAPH DATASETS

5.1 Introduction

Recently, there has been a lot of interest in mining frequent patterns from *structured datasets*, such as chemical compounds, proteins, web-logs, and XML datasets. Such patterns can effectively summarize the data, provide key insights and often serve as a preprocessing step for further analysis. Since, such datasets can often be modeled as graphs, a majority of research in this area has focused on developing efficient algorithms for mining frequently occurring (connected) subgraphs [60, 67, 107, 78].

However, in many real world applications, such as biology, social networks, and telecommunication, *large-scale structures*, which provide high-level topological information of graphs, may be equally or more important than discovering the basic components. For instance, the discovery of non-local or tertiary structural information is an important problem in protein structure analysis. Similarly, in the analysis of social or communication networks, the direct connection between a pair of nodes is often not the focus, instead, the patterns where several nodes are connected through a set of independent paths are of greater

interest. Such frequent large-scale structures can be very hard to discover using current frequent subgraph mining approaches. This is not only because the subgraphs sharing these kind of structures can be infrequent (i.e. the traditional anti-monotone property leveraged by most such algorithms does not hold), but also because the individual subgraphs are not adequately abstracted or represented.

As an example of a large-scale structure we are focusing on, consider mining a protein dataset where each protein is represented as a graph. The vertexes of each graph are protein secondary structures, and an edge is associated with two protein secondary structures if their distance in the three-dimensional space is within a certain range. A frequent large-scale topological structure in such a dataset can be as follows: three α-helices that are not direct neighbors of each other, but form a triangle in the three-dimensional space. Specifically, in the graphs for different proteins, each pair of above α-helices is connected through independent paths formed by other secondary structures, possibly including α-helices, β-sheets, or loops. The triangle information can be useful for understanding the functionalities of these proteins. For instance, two DNA-binding regulatory proteins (1ALI and 1E31), though seemingly different from the local-structure perspective, share such a α-helices triangle, and perform similar functionalities [34]. In fact, both belong to the class of zinc finger proteins. However, because this kind of structure is hidden under the pairwise relationship, it is very unlikely to be identified using the existing frequent subgraph mining approaches. In particular, even if some subgraphs which embed the three α-helices may appear to be frequent, the triangle structure can easily be missed.

The main contribution of this chapter is a framework to mine frequent large-scale structures from graphs. Our work is inspired by a well-established mathematical concept, *topological minor* [26]. A topological minor of a graph is an abstraction that focuses on its

structural information. Intuitively, such an abstraction is achieved by replacing or contracting *independent paths* in a subgraph with individual edges.

An important notion in our framework is that of a *relabeling function*. Since often real datasets can be best represented as labeled graphs when we replace independent paths in a subgraph with edges, the information labels on such paths are lost. However, in many applications, summarized information about the contracted paths can be useful to *categorize* these topological structures. For example, we may prefer to distinguish the α-helix triangles of different sizes, and the length of each independent path connecting these α-helices can help to provide such measurement. Our framework supports this notion through user-defined *relabeling functions* to recover some degree of information loss from the contracted paths. Such a function maps an entire labeled path to a single edge label. In other words, an edge label carries the desired information about its corresponding contracted path. For instance, in the above example, the relabeling function can use the length of each contracted path as their corresponding edge labels. An additional benefit of the relabeling function is that it can be used to support the mining of *constrained* topological structures.

To summarize, the main contributions of this chapter are as follows:

1. We introduce a novel framework for discovering frequent topological structures from graph datasets based on a vertical mining approach.

2. We study the basic properties of relabeling functions, and demonstrate their use for summarization and discovery of constrained topological structures. Our algorithms push the constraints deep into the mining process maximizing performance gains.

3. We evaluate the scalability and quality of the proposed framework on several real and synthetic datasets. We also demonstrate the use of the framework for discovering novel and meaningful motifs in membrane protein structures.

To the best of our knowledge, our work is the first to focus on the problem of mining frequent (large-scale) topological structures. Overall, our framework is also very flexible. It can be used for *approximate pattern mining*, where the support for a frequent pattern does not depend on the exact matches, but instead relies on some form of a *fuzzy matching* [52, 73]. The topological structures together with relabeling functions provide a powerful mechanism to express various forms of fuzzy matches.

5.2 Topological Minors and Topological Structures

We begin with some basic notations. Let $G = (V, E)$ be a graph, where V is the set of vertices, and E is the set of edges, and $E \subseteq V \times V$. The vertex set of a graph G is referred to as $V(G)$, and its edge set as $E(G)$. A *path* P in a graph G is a sequence of vertices $v_1, v_2, \cdots, v_k$, where $v_i \in V(G)$ and $v_i, v_{i+1} \in E(G)$. The vertices v_1 and v_k are linked by P and are called its *ends*, and $v_2, v_3, \cdots, v_{k-1}$ are the *inner* vertices of P. A path is *simple* if its vertices are all distinct, and we only consider simple paths in this chapter. Also, we define the number of inner vertices in a path as its *length*. In particular, a group of paths are *independent* if none of the paths have an inner vertex on another path. For simplicity, we call a path intersecting with other paths only at its ends as an *independent path*. Note that the independent paths are the key tools to study topological structures of a graph.

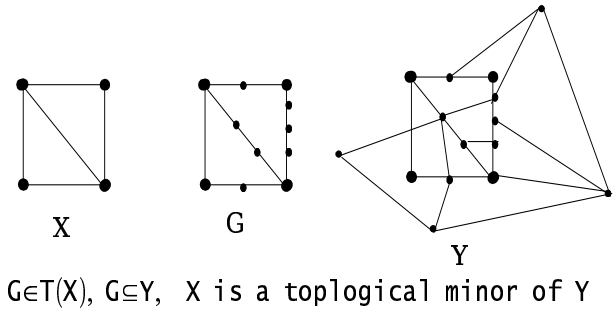


Figure 5.1: Topological Minor

5.2.1 Topological Minors

Informally, a *topological minor* of a graph is obtained by contracting the independent paths of one of its subgraphs into edges. For example, in Figure 5.1, X is a topological minor of Y since X can be obtained by contracting the independent paths of G, which is a subgraph of Y. Clearly, contracting independent paths helps simplify a (sub)graph without compromising its topological information [26].

The formal definition of the *topological minor* of a graph is as follows. A *subdivision operation* of a graph X, is to replace the edges of X with independent paths. A *subdivision graph* of X is a graph obtained by performing a subdivision-operation of X. For example, in Figure 5.1, the graph G is a subdivision graph of X. Note that the subdivision operation is basically an "inverse" of the path contraction operation. Further, the topological space of X, $T(X)$, is the collection of all its subdivisions graphs. If X has a subdivision graph G ($G \in T(X)$) and G is a subgraph of another graph Y, then X is a *topological minor* of Y. The vertices of X which corresponds to the original vertices of Y are called *branch vertices*.

5.2.2 Topological Structures

Topological structures of a graph are derived from topological minors. Given two parameters, l and $h, 0 \leq l \leq h$, an (l, h)-*subdivision* of a graph X, involves replacing *all* edges of X with independent paths whose lengths are between l and h. An (l, h) *subdivision graph* of X is a graph obtained by performing an (l, h)-subdivision operation of X. For example, in Figure 5.1, G is a $(0, 3)$-subdivision graph of X. Similarly, we can define the (l, h)-topological space of X, $T_{l,h}(X)$, to be the collection of all its (l, h)-subdivisions graphs. If X has an (l, h)-subdivision graph G ($G \in T_{l,h}(X)$) and G is a subgraph of another graph Y, then X is a (l, h)-*topological minor*, or a topological structure of Y. Therefore, in Figure 5.1, X is a $(0, 3)$-topological minor of Y.

The purpose of introducing the definition of topological structures of a graph is to control the compression ratio between a graph and its subdivision graph. In other words, when later we discover the frequent topological patterns from a graph dataset, the embeddings (subgraphs) that can contribute to the support of such a topological structure should be in a controllable size. Specifically, the following lemma describes the size difference between a graph and its subdivision graph in terms of vertex and edge number.

Lemma 12 *If a graph G is obtained by a (l, h)-subdivision operation of X, the number of vertices of G, ($|V(G)|$), and the number of edges of G, ($|E(G)|$), are bounded as follows:*

$$|V(X)| + |E(X)| \times l \ \leq \ |V(G)| \ \leq \ |V(X)| + |E(X)| \times h$$

$$|E(X)| \times (l+1) \ \leq \ |E(G)| \leq |E(X)| \times (h+1)$$

The following two lemmas also describe important properties of topological structures of a graph, and their proofs directly follow the above definitions.

Lemma 13 *Assume X is a (l_1, h_1)-topological minor of G, then for any l and h, where $l \leq l_1$ and $h_1 \leq h$, X is (l, h)-topological minor.*

Lemma 14 *The number of graphs in the (l, h)-topological space of X ($|T_{l,h}(X)|$) is bounded by $(h - l + 1)^{|E(X)|}$.*

In the following, we will mainly focus on the topological structures ((l, h)-topological minors) of a graph.

5.2.3 Labeled Graphs

So far, our discussion has focused on unlabeled graphs. Data miners are often more interested in *labeled* graphs. In the following, we extend the concept of topological structures on labeled graphs. Note that unlabeled graphs can be treated as a special case of labeled graphs, where all the vertices and edges have the same label.

We begin with the informal discussion of the topological structures on a labeled graph. Intuitively, the way to simplify a labeled graph is to remove all the inner vertices and edges of its independent labeled paths, and then connect their remaining labeled ends with an unlabeled edge. Later, in Section 5.4, we will study how to use *relabeling functions* to add labels to these edges. Clearly, the main difference between the topological structures on labeled graphs and on unlabeled graphs is that the vertex labels for the ends of contracted paths are still preserved. Similarly, in an unlabeled graph, such simplification maintains the important topological information from the original graph.

To facilitate our formal discussion of topological structures on labeled graphs, we first define a labeled graph. Let $G = (V, E)$ be an unlabeled graph. Let L_v and L_e be two sets of labels. A vertex labeling function, $l_v : V \rightarrow L_v$, will assign a vertex v with a vertex label $l_v(v) \in L_v$. Similarly, an edge labeling function, $l_e : V \rightarrow L_e$, will assign an edge e with

an edge label $l_e(e) \in L_e$. We refer to a graph G labeled by l_v and l_e as a *labeled graph*. A graph G only labeled by the vertex labeling function (l_v) is called a *vertex labeled graph*, and similarly, a graph G only labeled by the edge labeling function (l_e) is referred to as an *edge labeled graph*.

To simplify our discussion, we will mainly focus on the vertex labeled graphs. For example, all the graphs in Figure 5.2 are vertex labeled graphs. Note that our results and methods can be easily extended to (edge) labeled graphs.

Given two parameters, l and h, the main difference between an (l, h)-topological minor on labeled graph and unlabeled graph is the subdivision operation. An (l, h)-*subdivision operation* of a vertex labeled graph X, involves replacing all edges of X with independent paths satisfying the following conditions: 1) the path lengths are between l and h, 2) the vertices (and edges) in the paths are labeled, and 3) the ends of these paths share the same vertex label as the corresponding ends of their original edges.

The other concepts, including the (l, h)-subdivision graph, the (l, h)-topological space, and (l, h)-topological minors, are the same as in unlabeled graphs. Therefore, in Figure 5.2, the vertex labeled graph G_a is a $(1, 1)$-topological minor of the graph G_1, and a $(1, 2)$-topological minor to the graph G_2 and G_3.

Assume we have a collection of graphs, denoted as D. Given two parameters l and h, and a graph G, the number of graphs in D which have G as a (l, h)-topological minor (also topological structure) is referred to as the *support* of G.

Definition 8 *Given a collection of graphs, two parameters l and h, and a threshold θ, a (l, h)-topological minor whose support is greater than or equal to θ is called a frequent topological structure.*

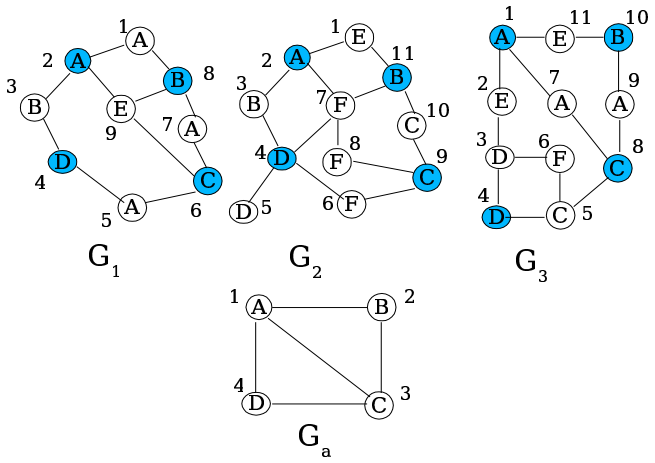


Figure 5.2: Running Example

For example, in Figure 5.2, for $l = 1$ and $h = 2$, the support of the graph G_a is 3 in the dataset composing of G_1, G_2, G_3, however, for $l = 0$ and $h = 1$, the support of the graph G_a is only 1.

5.3 Algorithm for Mining Topological Structures

Frequent topological structure mining is a generalization of frequent graph mining. Specifically, frequent sub-graphs for a vertex-labeled graph dataset can be mined as a special case of frequent topological structures: the $(0, 0)$-topological minors. It should also be noted that frequent topological structures are also graphs. Therefore, mining frequent topological structures shares some similarities with mining frequent graphs.

However, mining frequent topological structures is also quite different from graph mining. Given two parameters l and h, the support of a topological structure G depends on the definition of (l, h)-topological minor. Specifically, if G is a (l, h)-topological minor of a graph D_i in the graph dataset, we need to know if there is a subgraph H of D_i and H is a

(l, h)-subdivision graph of G. This potentially involves not only the subgraph isomorphism testing, but also the (l, h)-subdivision operation. In particular, counting support of topological structures is one of key issues in efficiently mining frequent topological structures.

In the following, we first present our approach to efficiently counting the support for a topological structure (Subsection 5.3.1). Then, we show how we perform a depth-first search to enumerate all the frequent patterns using the counting approach (Subsection 5.3.2).

5.3.1 Counting Support for Topological Structures

As mentioned before, compared with frequent subgraph mining, one of the main challenges for our mining algorithm is the need to handle the subdivision operation (path contraction) in addition to the subgraph isomorphism testing. To tackle this problem, we use an incremental approach. Consider a topological structure G' that can be extended from another topological structure G by adding a new edge e, denoted as $G' = G \cup \{e\}$. To test if G' is a topological structure of a graph H, our approach utilizes the information derived from G. In particular, such reuse is based on a uniform representation for a topological structure G and its corresponding subgraph in H. In the following, we first establish such representation, and then discuss the details of how we count the support of a topological structure.

Decomposition-based Representation Given l and h, let G be an (l, h)-topological minor of H. This implies that there exists a subgraph Y of H, where Y is a (l, h)-subdivision graph of G by a subdivision operation. To facilitate our discussion, we denote the subgraph

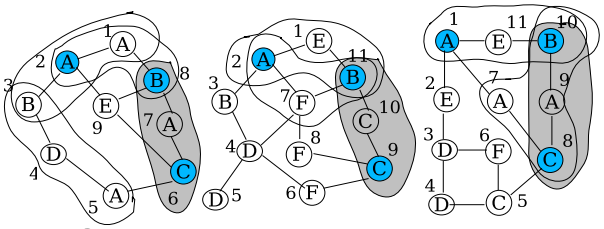
Y together with an (l, h)-subdivision operation as an *occurrence* of G. Here, Y is isomorphic to the graph obtained by performing the subdivision operation on G. In the following, we consider how we can express the occurrences of G explicitly.

We first decompose G as a collection of edges, i.e., $G = \{e_1\} \cup \{e_2\} \cdots \cup \{e_k\}$. Based on the definition of the subdivision operation, each edge e_i corresponds to an independent path in Y, denoted as $\vec{e_i}$. Therefore, we can also decompose Y as a collection of independent paths, i.e., $\{\vec{e_1}\} \cup \{\vec{e_2}\} \cdots \cup \{\vec{e_k}\}$. We denote this decomposition as $\vec{Y}$. Clearly, the above decomposition of Y can be used to represent an occurrence of G in H. For example, in Figure 5.3(a), we have $\{(2, 1, 8), (8, 7, 6)\}$ of G_1 to be an occurrence of the topological structure, $G' = \{(A, B), (B, C)\}$.

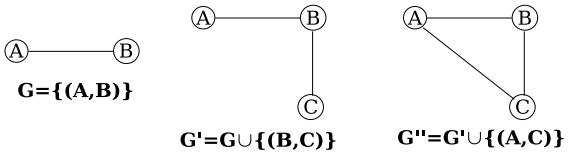
The decomposition can be further represented in a very concise format. Consider $G \cup \{e\}$ which is also a (l, h)-topological minor of H. Let $S_{G,H} = \{\vec{Y_1}, \vec{Y_2}, \cdots, \vec{Y_m}\}$ be all the occurrences of G in H. We have the following lemma.

Lemma 15 *The occurrences of $G \cup \{e\}$ can be represented as $\vec{Y_1'} \cup \{\vec{e}\}, \cdots, \vec{Y_n'} \cup \{\vec{e}\}$, where $Y_i' \in S_{G,H}, 1 \leq i \leq m$. Y_i' is called the parent occurrence of $\vec{Y_i'} \cup \{\vec{e}\}$.*

Given a topological structure G', we can decompose it as $G \cup \{e\}$, where G is called a parent of G'. For example, in Figure 5.3(b), we have $G' = G \cup \{(B, C)\}$, where $G = \{(A, B)\}$. Lemma 15 suggests that occurrences of G' can be partially represented by the occurrences of its parent. Naturally, for each topological structure, we can build an *occurrence list* to concisely record all of its occurrences in the graph dataset by using the occurrence list of its parent. Note that a topological structure can have many parents. However, we only need one of its parents to build its occurrence list. The question of which one of these parents is chosen will be addressed in Subsection 5.3.2).



(a) Occurrence Decomposition



(b) Topological Structure Decomposition

G ={(A,B)}	G'= G∪{(B,C)}	G''= G'∪{(A,C)}
0:{G_1, 0, (2,1,8)}	0:{G_1,0,(8,7,6)}	0:{G_1,0,(2,9,6)}
1:{G_1,0, (2,9,8)}	1:{G_1,0,(8,9,6)}	
2:{G_1,0, (1,2,3)}	2:{G_1,1,(8,7,6)}	
3:{G_1,0, (5,4,3)}		
4:{G_2,0, (2, 1, 11)}	3: (G_2,4,(11,10,9)}	1:{G_2,3,(2,7,8,9)}
5:{G_2,0, (2,7, 11)}	4: (G_2,5,(11,10,9)}	
6:{G_3,0, (1,11,10)}	5:(G_3,6,(10,9,8)}	2:{G_3,5,(1,7,8)}
7:{G_3,0, (7,8,9,10)}		
...	...	...

(c) Occurrence Lists

Figure 5.3: Decomposition and Occurrence Lists

The concise representation of each occurrence in the occurrence list for a topological structure $G \cup \{e\}$ is as follows. Each occurrence has a unique ID in the occurrence list, and the detailed information is a triple, (α, β, δ). Here, α is the index of the graph in the dataset D where this occurrence appears, β is the occurrence ID of this occurrence's parent, and δ is an independent path, $\vec{e}$, corresponding to the edge e. For instance, Figure 5.3(c), illustrates a portion of the occurrence lists for three $(1, 2)$-topological structures, G, G', and G''.

Building the Occurrence Lists Clearly, the support of a topological structure can be easily derived from its occurrence list. Therefore, the problem of efficiently counting the support of a potential frequent topological structure becomes the one of how we build its occurrence list efficiently. However, the straightforward solution can be very costly. For example, suppose we already have the occurrence list for G and try to build the occurrence lists for $G \cup \{e\}$ and $G \cup \{e'\}$, where e and e' are adjacent to the same vertex v in G. The straightforward method will build the occurrence lists for them independently. Specifically, for each of them, we need to go through all the occurrences of G to find out *all* the independent paths corresponding to edge e or e' (path contraction). This, however, involves a lot of repetitive work, since each time we have to find *all* the independent paths starting from the branch vertex corresponding to v in each occurrence. Note that the similar problem also needs to be addressed in frequent subgraph mining algorithms. However, it is even more costly in our algorithm because of the high cost of finding independent paths.

In order to build the occurrence lists efficiently for the topological structures, we try to minimize the number of times the *finding independent paths* operation needs to be invoked.

We also build occurrence lists in parallel when we invoke such an operation. To formally discuss our approach, we first introduce some notation.

Let us consider generating new frequent topological structures by extending an existing frequent topological structure G with a new edge. We classify these new edges in two categories: *inner* edges or *outer* edges. An inner edge connects two dis-adjacent vertices in the graph G, and an outer edge adds a new vertex into $V(G)$, and connects an existing vertex in $V(G)$ with this new vertex. For a topological structure G, we denote $[G]_{inner}$ to be the set of all inner edges of G, and $[G]_{outer}$ to be the set of all outer edges of G. We use $[G]_{io}$ to represent the union of $[G]_{inner}$ and $[G]_{outer}$. The significance of these two sets $[G]_{outer}$ and $[G]_{inner}$ is that they record all the potential extensions of G. Finally, for an extended graph $G \cup \{e\}$ from G, we denote its occurrence list as $e.occurrencelist$ or $(G \cup \{e\}).occurrencelist$.

The basic idea of our approach is as follows. For each topological structure G, we will maintain the occurrence list for each extended graph $G \cup \{e\}$ where $e \in [G]_{io}$. We will show an optimization in next subsection to reduce the number of recorded occurrence lists. Here, we consider how we can build these lists for $G \cup \{e\}$. If e is an inner edge, we can have $[G \cup \{e\}]_{io} \subseteq [G]_{io}$. Therefore, we need to simply copy the occurrence lists for the edges in $[G]_{io}$. Note that this is not a real copy since not all occurrences for $G \cup \{e'\}$, $e' \neq e, e' \in [G]_{io}$ can be extended to $G \cup \{e\} \cup \{e'\}$. Essentially, this copy is a *Join* operation, which will be discussed later. Further, if e is an outer edge, the new vertex generated by e will be likely to bring some new outer edges. Also, the existing outer edges of G may become inner edges for $G \cup \{e\}$. In this case, we will not only need to copy these occurrence lists from G, but also need to build the occurrence lists for all the new outer edges adjacent to the new vertex.

```
global Set Visited, Set PathSet;
IndependentPath(Graph G, Embedding emb, Vertex s)
 Visited ← ExtractEmbeddingVertex(emb);
 PathSet ← ∅;
 RecursivePath(G, s, {s}); {*p.from = s*}
 {* return all the independent paths corresponding to an
  outer edge, bounded by l and h, and staring from s *}
 returnPathSet;

RecursivePath(Graph G, Vertex v, Path p)
 if (|p| − 2 > h){* No. of inner vertices *}
      return;
 Visited ← Visited ∪ {v};
 foreach (v' : (v', v) ∈ G and (v' ∉ Visited)
      p ← p ∪ {v'}; {*p.to = v'*}
      if (|p| − 2 ≥ l)
           PathSet ← PathSet ∪ {p};
      RecursivePath(G, v', p);
      p ← p − {v'};
 Visited ← Visited − {v};
```

Figure 5.4: Enumerate Independent Paths

Finding Independent Paths The sketch of the algorithm for finding all independent paths for an occurrence $\vec{Y}$ starting from a branch vertex s is illustrated in Figure 5.4. Let G be the graph where this occurrence $\vec{Y}$ appears. We perform a depth-first search (DFS) to enumerate these paths. There are two important issues we need to deal with. The first involves maintaining the *independent* property, and the second involves bounding the length of each path, specifically, the number of inner vertices, between l and h. To deal with the first issue, we color the vertices in the occurrence of G (in *IndependentPath*). Then, as we traverse the graph G starting from the branch vertex s, we keep coloring the visited vertices. If we meet any colored vertex, we need to trace back since the path has become

not independent (the *foreach* loop in *RecursivePath*). When we found an independent path (the number of inner vertices) bounded by l and h, we will record this path. Finally, our traversal will trace back when the length of path is greater than the upper bound h. Note that the tracing back operation is associated with uncoloring the visited vertex.

Operation Description In the following, we formally introduce the two key operations mentioned earlier, which are the *Join* operation and the *ExtendOuterEdge* operation. The two operations are sketched in Figure 5.5. Assume G is generated by adding an outer edge e on its parent. The procedure *ExtendOuterEdges* will scan the entire list of occurrences of G (the first *foreach* loop in *ExtendOuterEdges*). For each occurrence, let $p.to$ be its branch vertex corresponding to the newly added vertex for G. This procedure will find all the independent paths beginning from this branch vertex (the second *foreach* loop in *ExtendOuterEdges*). Specifically, such functionality is achieved by the subroutine *IndependentPath* just introduced. Each independent path generated above corresponds to a new outer edge for the topological structure G, and the occurrence lists for these new outer edges are built by adding these independent paths (implemented by *insertOccurrence*). Finally, *ExtendOuterEdges* will return all the new edges which are frequent with respect to the given support level.

A new topological structure, $G \cup \{e\}$, will inherit more information from its parent G through the procedure *Join*. The *Join* operation will filter the occurrence lists for each edge in $[G]_{io}$ to generate all the inner edges. It will also filter all the outer edges adjacent with the vertices in $V(G)$ for $G \cup \{e\}$ (implemented by the nested *foreach* loops in *Join*). The essential part of the *Join* operation is to test if, after extending the new edge e, the paths

in the occurrences are still independent. This is done by the routine (*Independent* invoked from *Join*. For brevity, the deailts of its implementation are omitted.

Correctness One of the key properties of the topological structure is that all the paths corresponding to the edges in the subdivision graph are independent. In our algorithm, we explicitly maintain the paths corresponding to the edges for a topological structure G, by two operations, *ExtendOuterEdges* and *Join*. Therefore, the correctness of our algorithm depends on whether these paths in an occurrence are independent. Formally, assume that a graph-topological structure G is generated from the following edge sequence: $\{e_0\}, \{e_1\}, \cdots, \{e_k\}$. In our algorithm, an occurrence of G can be represented by the union of the corresponding paths, i.e., $\{\vec{e_0}\}, \{\vec{e_1}\}, \cdots, \{\vec{e_k}\}$. The following lemma states that the independence property is maintained for these edges. Therefore, it implies that our algorithm can correctly generate topological structures for a graph, and henceforth, correctly discover frequent topological structures.

Lemma 16 *The paths in any occurrence of G, i.e.,*
$\{\vec{e_0}\}, \{\vec{e_1}\}, \cdots, \{\vec{e_k}\}$*, are independent.*

Proof: By induction. □

5.3.2 Vertical Mining Approach

Our approach mines frequent topological structures in two phases. In the first phase, we mine all the frequent topological structures which are trees, and are referred to as *frequent tree-topological structures*. In the second phase, for each tree-topological structure T, we mine *frequent graph-topological structures* which have T as their spanning tree. The tree-topological structures are graphs without cycles, and the graph-topological structures are

graphs with at least one cycle. Note that the two-phase procedure has been proposed and used for efficiently mining frequent subgraphs also [107, 55].

In the first phase of our algorithm, a candidate frequent tree-topological structure can be generated by looking at edges in $[G]_{outer}$. In the second phase, a candidate frequent graph-topological structure can be generated through $[G]_{inner}$. Finally, if a topological structure G' is generated by adding a new edge e on G, $e \in [G]_{io}$, we call G as the *parent* graph of G'. Note that the above treatment is very similar to the algorithms in mining (connected) subgraphs since the frequent topological structures are also graphs.

A difficulty in enumerating frequent topological structures is that one frequent topological structure can be derived from different parent graphs, i.e. $G_1 \cup \{e_1\} = G_2 \cup \{e_2\}$, where $G_1 \neq G_2$. Clearly, an efficient mining algorithm needs to avoid generating duplicate frequent topological structures. This requires efficient topological structure isomorphism tests. This is why we use a two-phase procedure to enumerate frequent tree and graph topological structures separately. Basically, linear-time algorithms exist for enumerating tree topological structures, and therefore, our first phase can efficiently deal with tree-isomorphism. The complicated cases which require graph isomorphism testing arise only in the second phase.

Our algorithm is sketched in Figure 5.6. The mining procedure *VTreeTS* corresponds to the first phase, and the mining procedure *VGraphTS* corresponds to the second phase. To generate frequent tree-topological structures, for each tree T, we use the mechanisms introduced by Nijssen [78] to determine which edges in $[T]_{outer}$ are *valid* extensions. The valid extensions can also help to enumerate all frequent tree-topological structures without replication. Specifically, the procedure *ValidExtension* (invoked by *VTreeTS* in the *foreach* loop) provides the above mechanism. The frequent graph-topological structures are

enumerated by adding a subset of inner edges in $[T]_{inner}$ to each frequent tree-topological structure T. In our algorithm, the procedure *CanonicalExtension* (invoked by *VGraphTS* in the *foreach* loop) applies hashing and graph isomorphism test (*nauty* [72]) to avoid duplicating graph-topological structures.

The dominant computational time of our algorithm is in maintaining the edge sets, $[G]_{outer}$ and $[G]_{inner}$, for each topological structure G. Note that when G is a graph-topological structure, we only need to maintain its inner edge set. Our algorithm maintains them in an incremental manner. For a new tree-topological structure, $T \cup \{e\}$, it can inherit some of the inner and outer edges in $[T]_{io}$ through a *Join* operation (the *foreach* loop in *VTreeTS*). However, the new vertex (because of e) in the graph $T \cup \{e\}$ brings new outer edges, which do not appear in $[T]_{outer}$. In our algorithm, the procedure *ExtendOuterEdges* (invoked by *VTreeTS*) generates these new outer edges. For a new graph-topological structure, $G \cup \{e\}$, it only needs to inherit inner edges from its parent's inner edge set $[G]_{inner}$ through the *Join* operation (the *foreach* loop in *VGraphTS*).

5.4 Mining Topological Structures with Relabeling Functions

As discussed before, topological structures of a subgraph are extracted through compressing the inner vertices and edges of their independent paths into corresponding unlabeled edges. Two paths that have a different set of inner vertices and edges can be treated as the same, as long as the labels of their ends are the same. However, in many applications, the labels for inner vertices (and the inner edges) can provide important additional information. In order to reflect such information in the topological structures, we allow users to define a *relabeling function*, which assign labels to the edge in topological structures corresponding to the path that has been contracted.

In this section, we first formally introduce relabeling functions and briefly discuss their efficient implementation in the mining process. Then, we discuss how we can use relabeling functions to perform *constraint* topological structure mining. Finally, we relate relabeling functions with *approximate pattern mining*, and present how our framework can handle *fuzzy chains* in molecular fragments [73].

5.4.1 Relabeling Functions and Their Implementation

Consider a path $p = (v_0, v_1, \cdots, v_k)$. Normally, when it is contracted in a topological structure, the only information left is its ends, v_0 and v_k, with their vertex labels. Relabeling functions can preserve important additional information from these contracted paths, in the form of labels for the corresponding edges in the topological structure.

Formally, a relabeling function $f : \mathcal{P} \rightarrow \mathcal{L}$ can be defined as a map from the set of all possible paths $\mathcal{P}$ to the new edge-label set for the topological structure $\mathcal{L}$. To facilitate our discussion, the set $\mathcal{L}$ always contains a *null* symbol, $\emptyset$. Note that a given path p can usually be expressed in two different formats, p and $\overline{p}$, where $\overline{p}$ is the reverse of p, i.e. $\overline{p} = (v_k, \cdots, v_1, v_0)$. Clearly, not any map between $\mathcal{P}$ and $\mathcal{L}$ is valid, because they have to be consistent with respect to both p and $\overline{p}$. Therefore, a valid relabeling function f needs to satisfy the *reverse symmetric* property, i.e. $f(p) = f(\overline{p})$, for a given path $p \in \mathcal{P}$.

A common type of relabeling functions is derived from the length of each independent path. For example, we can use the length of a contracted path to label its corresponding edge. Formally, for a given path $p = (v_0, v_1, \cdots, v_k)$, $f(p) = k - 1$. Clearly, it satisfies the reverse symmetric property. Note that in this way, the edges in the topological structures become labeled. In order to efficiently mine frequent topological structures utilizing these relabeling functions, we need to push relabeling deeply into the support counting process.

In our mining algorithm, the *ExtendOuterEdge* scans these independent paths generated by the routine *IndependentPath*, and contracts these paths into corresponding edges (*e* $\leftarrow$ *Edge(p.from,p.to)* in Figure 5.5, p is an independent path). To implement a relabeling function, we need to compute a new label using the relabeling function f, $f(p)$, where p is an independent path, and then use it to label the corresponding edge, *Edge(p.from,p.to)*. In particular, if it is the null symbol $\emptyset$, we simply remove this path. Otherwise, we put this path into the occurrence list for the contracted edges with this new label $f(p)$.

5.4.2 Mining Topological Structures with Constraint Conditions

In this subsection, we study a specific type of relabeling function: *constraint conditions*. Such constraint conditions can help data miners focus only on certain types of independent paths to be contracted. In this way, for the edges in a frequent topological structure, the user can have an idea of what kind of paths (subgraphs) are contributing to them. In the following, we consider a powerful mechanism to specify such constraint conditions, which is based on regular expressions. For example, the following expression

$$\{A\} : A|B|C^2|E : \{B\}$$

requires that an independent path in the graphs starting from a vertex with label A, ending with a vertex with label B, either have length one with the inner vertices labeled as A,B,E or have length two with the inner vertices both labeled with C.

Such constraint conditions can be transformed into a table format: a table C with $|L_v|$ rows and $|L_v|$ columns, where L_v is the set of all the vertex labels. (The details of the transformation procedure is omitted for simplicity.) Each row and each column corresponds a label in L_v; and each cell has a regular expression. A cell C_{l_i,l_j} specifies a path starting with a label l_i, ending with a label l_j, and with the inner path labeled as C_{l_i,l_j}, can be

contracted into an edge $\{l_i, l_j\}$. Specifically, $C_{l_i,l_j} \subseteq L_v^l \cup \cdots \cup L_v^h$ since the length of each contracted path needs to be bounded by l and h. For example, Figure 5.7 illustrates such a table for the vertex label set $\{A, B, C, D, E\}$ in the table format. Note that an empty set ($\emptyset$) in a cell C_{l_i,l_j} suggests no path can be contracted as $\{l_i, l_j\}$; the symbol (?) represents the set L_v. Finally, the table also satisfies the *reverse symmetric* property: $C_{l_i,l_j} = \overline{C_{l_j,l_i}}$.

Mathematically, we can treat a regular-expression based condition as a type of relabeling function. Specifically, we can define the new edge-label set $\mathcal{L}$ of topological structures as $\mathcal{L} = \{1, \emptyset\}$. The symbol 1 represents that a path is acceptable by the constraint condition, and the symbol $\emptyset$ corresponds to the rejection of a path. Therefore, for a given path $p = (v_0, v_1, \cdots, v_k)$, if it satisfies the constraint condition, the relabeling function returns 1, otherwise, it returns $\emptyset$ (in other words, this path is simply removed). The detailed implementation is as follows. Basically, for each candidate path, we will use its ends to find the corresponding regular expression in the constraint table. To facilitate processing, we will map the regular expressions in the constraint table into DFAs (Deterministic Finite Automaton). Then, we will test if the path is accepted or rejected by the DFA. If it is rejected by the DFA, we will simply remove this path.

5.4.3 Mining Fuzzy Chains using Relabeling Functions

In the following, we study how we can use topological structure together with relabeling functions to implement one type of *approximate pattern mining*, which is mining fuzzy chains in chemical compounds [73].

We begin with the definitions of fuzzy chains. A chain in a chemical compound satisfies the following conditions: 1) every vertex corresponding to an atom in a chain has the same type, 2) every vertex in the chain must have exactly two edges (labeled with *single bound*

type) to other vertex, and 3) a chain always consists of the maximal possible number of atoms satisfying the first two conditions and must have a minimum length of one. For a biologist or a chemist, two chains are equivalent if both chains have the same atom type and the lengths of the two chains can be different and are bounded by user-defined ranges. Since such chains do not need an exact match, we call them *fuzzy chains*.

Let us consider the length of the fuzzy chains to be between two and four atoms (the common case). Then, the frequent chemical fragments with such fuzzy chains can be mined in our framework as $(0, 4)$-topological minors with the following relabeling function. For a given independent path, 1) if the path has no inner vertex, use the original edge label as the edge label for the new edge, 2) if the path has a number of inner vertices between 2 and 4, we check the following conditions for the path to see if it satisfies the chain condition, and return the atom type in the chain to label the new edge for the true case, and 3) remove the path in other conditions.

The method discussed in Subsection 5.4.1 can be used to implement this relabeling function.

5.5 Case study: Membrane Protein Structure Analysis

Discovery of lipids binding sites has been long known as a very challenging, but important, task for the biologists [80]. In this study, we use our new tool to search potential *protein-lipid binding sites* in an important class of proteins - membrane proteins, which are believed to account for approximately 20-30% of all protein sequences.

The dataset we use is derived from the protein data bank (PDB). [2] We use a set of six membrane proteins known to bind with cardiolipins (CL): 1KB1, 1KQF, 1M3X, 1OKC,

[2]Thanks for dmitrii polshakov's help with providing the dataset and analyzing the experimental results.

Parameters			No. of Large Topological Structures		
Support	l	h	Path	Tree	Graph
6	0	4	11 ($\|V\| = 3$)	0	1 ($\|E\| = 3, \|V\| = 3$)
5	0	3	1 ($\|V\| = 5$)	4 ($\|V\| = 4$)	4 ($\|V\| = 3, \|E\| = 3$)
5	1	2	17 ($\|V\| = 3$)	0	1 ($\|V\| = 3, \|E\| = 3$)
4	0	0	0 ($\|V\| > 2$)	0	0
4	0	1	11($\|V\| \geq 4$)	5 ($\|V\| \geq 4$)	2 ($\|V\| = 4, \|E\| = 4$)
4	1	2	27($\|V\| \geq 4$)	2 ($\|V\| \geq 4$)	1 ($\|V\| = 4, \|E\| = 4$)
4	0	2	24($\|V\| \geq 5$)	10 ($\|V\| \geq 5$)	10 ($\|V\| \geq 4, \|E\| \geq 4$)
3	0	0	1 ($\|V\| = 6$)	1 ($\|V\| = 6$)	0
3	0	1	20 ($\|V\| \geq 8$)	34 ($\|V\| = 9$)	19 ($\|V\| \geq 9, \|E\| \geq 9$)
3	1	2	12 ($\|V\| = 7$)	19 ($\|V\| = 8$)	20 ($\|V\| \geq 7, \|E\| \geq 7$)

Table 5.1: Number of Large Patterns Discovered by *TSMiner*

1V54, and 1OGV. Amino acids as nodes in the graph (20 labels) and edges between nodes are drawn if two amino acids are within 3.5 Å . There are known to be 20 naturally occurring amino acids and these serve as node labels. In order to find the structural motifs that can serve as binding site for a CL head group, we used only the relevant parts of proteins that are known to be local to CL molecule. Such a structure typically contains around $\sim 30 - 35$ amino acids (number of nodes per graph). Note that several membrane proteins we use contain more than one CL molecule. Therefore, the total number of CL binding regions that we used to find protein-lipid binding sites is 10 (number of graphs).

Table 5.1 summarizes the results on mining this dataset using our tool. Note that *TSMiner* at $l = 0$ and $h = 0$ is simply a connected subgraph mining tool (same results as with Gaston). For this parameter setting, one can only find patterns till the support level is 3, and the largest one found contains at most 6 vertexes. However, upon varying the value of the parameters, we find large triangles with support 5 and 6, along with large rectangles, and topological structures containing 5 or more vertexes. At support 3, with relaxed l and

h, we found a number of large topological structures, containing more than 9 vertexes, and 9 edges. Figure 5.8 shows two such large topological structures discovered by our toolkit. The topological structures consist largely of polar (N, T, S), charged (K) and aromatic (W) residues which is in agreement with recent advances in the understanding of such proteins within the biophysics community[80]. The structure we find is larger than any known motifs for CL binding sites in such proteins and also seems to partially span the membrane bridging components of the protein which seems quite novel according to domain experts.

5.6 Experimental Results

In this section, we will study the performance of our new algorithm, *TSMiner*, focusing on the following three issues: the scalability of the algorithm, how the parameters, l, h, and the support level θ, affect the performance, and how the relabeling functions affect performance. [3] We have implemented *TSMiner* in C++. The evaluation studies were conducted on a 2.66 GHz Pentium 4 machine with 1GB main memory, running Linux Mandrake 10.1.

5.6.1 Datasets Description

Our experiments used both synthetic and real datasets, containing vertex labeled graphs, i.e., the edge labels were not considered.

Synthetic Datasets: The synthetic datasets were generated from the graph generator provided by Kuramochi and Karypis at the University of Minnesota. Though this generator was originally designed for evaluating frequent subgraph mining algorithms, we have used it to study the performance and scalability of the algorithm for mining frequent topological structures. In our experiments, the following parameters were used to generate datasets:

[3]Thanks for Chao Wang's help with the experimental evaluation of *TSMiner*.

1) $|D|$, the total number of graphs to be generated, 2) $|T|$, the average number of edges for the generated graphs, 3) $|L|$, the total number of potentially frequent subgraphs, 4) $|I|$, the average number of edges in each potentially frequent subgraph, and 5) $|V|$, the total number of available labels for the vertices. In our experiments, we fixed $T = 20$, $L = 200$, $I = 5$, and we vary V, the total of vertex labels, to be between 5 and 20.

Chemical Compound Dataset from PTE: This dataset was originally used for the Predictive Toxicology Evaluation Challenge [99]. It contains a total of 340 chemical compounds. For each compound, the atoms correspond to the vertices of the graph, and the bonds between the atoms are mapped to the edges of the graph. Overall, the entire dataset contains a total of 66 vertex labels. For simplicity, we refer this dataset as *Chemical340*.

5.6.2 Performance Evaluation

Scalability: For the scalability study, we rely on the synthetic datasets. Figure 5.9 shows the performance of *TSMiner* under different conditions. In Figure 5.9(a) and (c), we vary the support threshold from high to low, and run our algorithm on datasets containing $10,000$ graphs. As we would expect, as the support level reduces, the running time increases. Also, we can observe that as h increases (l kept the same), the running time increases. This is also expected as the number of (potential) frequent topological patterns increases as we relax the condition on the length of the independent paths. From Figures 5.9(b) and (d), we see that *TSMiner* scales reasonably well (close to linear) as we increase the size of the dataset. Note that the *TSMiner* with parameters $l = 0$, $h = 0$ is essentially a frequent connected subgraph mining tool for vertex labeled graphs. For such cases, we did a comparison with the state-of-art subgraph mining tool *gSpan* [107]. Our results show that our implementation is slower by a factor of 1.6. We believe this is a reasonable result,

given that we offer additional functionality and do not specifically optimize for subgraph mining.

Number of Patterns and Running Time with respect to l and h In this study, we are interested in the number of patterns being generated by our new algorithm and its running time respect to the parameters l and h. Figure 5.10 presents the experimental results on the real dataset Chemical340. Figure 5.10 (a) shows the number of path, tree, and graph topological structures discovered by *TSMiner* at a support of 200. The primary observations of note here are: when using traditional graph mining algorithms ($l = 0$ and $h = 0$ in our tool) no frequent graph patterns are found; upon increasing the value of h to 1, 2, and 3, we are able to identify frequent graph structures; and finally from Figure 5.10(b) we can see that as the value of h is increased the running running time of our tool increases as it has to evaluate more candidate patterns and the cost for generating each pattern increases (the independent paths become longer). Figure 5.10 (c) and (d) show the total number of patterns being discovered and the running time of TSMiner at different support levels, as we increase h and keep l to be 1.

The Effect of Relabeling Functions In this study, we focus on how relabeling functions impact the performance of our algorithm. We study two types of relabeling functions. The first uses the length of the contracted path to relabel the corresponding edge, and is referred to as *length-relabeling* (see Subsection 5.4.1). The second involves constraining each path with a regular expression, or DFA, and is referred to as *DFA-relabeling* (see Subsection 5.4.2).

Figure 5.11(a) and (b) shows the number of patterns being generated by TSMiner without length-relabeling and the corresponding running time. The result is quite interesting.

Using relabeling, *more frequent patterns are being generated, however, the running time decreases significantly*. Basically, as we relax the condition for the length of independent path for a given topological structure, many occurrences with independent paths of different length maps to it. As we perform length-relabeling, the topological structures will be further categorized based on the size of its occurrences. This reduces the number of occurrences, as the condition for a subgraph being the subdivision graph of a topological structure becomes stricter. Therefore, such a relabeling function can improve the performance of *TSMiner*.

Figure 5.11(c) and (d) show the number of patterns being generated by *TSMiner* without DFA-relabeling and the corresponding running times. The constraint conditions are generated as follows. We first randomly generate a group of 100 DFAs to describe the conditions of an independent path. In particular, we use a parameter r to control how likely it is that an independent path can be accepted. In our experiment, for the independent paths having length 1, 2, and 3, their possibilities to be accepted were 0.5, 0.25, and 0.125, respectively. Then, each cell in the constraint condition table (defined in Subsection 5.4.2) is randomly assigned with a generated DFA. As shown in the figures, the DFA-relabeling reduces the number of frequent topological patterns being generated, as well as the running time.

5.7 Related Work

The early efforts on discovering useful patterns from graph datasets include the SUBDUE system [22] and WARMER algorithm [25]. The SUBDUE system relies on the Minimal Description Length (MDL) principle and a greedy strategy to find a subset of frequently occurring subgraphs. The WARMER algorithm combines Inductive Logical Programming

(ILP) with Apriori's level-wise search strategy [7] to find a wide class of frequent substructures. However, it is well known that ILP-based approaches are still quite expensive computationally, and do not scale very well to large datasets.

Recently, frequent *subgraph* mining approach has received much attention. This approach enumerates all frequent patterns defined by a class of subgraphs. The AGM algorithm [60] was the first to be proposed in this category. It can find all frequent *induced* subgraphs in a graph dataset. A subgraph G_s of G is *induced* if the subgraph G_s contains all edges in G connecting its vertices. The more recent efforts focus on discovering all frequent *connected* subgraphs. Several efficient algorithms, such as FSG [67], gSpan [107], FFSM [53], and Gaston [78], have been proposed to mine these kind of patterns. Two different types of search strategies are used in these algorithms: apriori's level-wise strategy and Eclat's [117] depth first search strategy. The experimental results show in most of the cases, the latter is more computationally efficient, and the former is more memory efficient. The framework proposed in this chapter enumerates a more *generalized* pattern in a graph dataset. The connected subgraph mining is a special case for this new type of topological structure mining. To efficiently enumerate these kind of patterns, our new algorithm, TSMiner, also uses Eclat's DFS strategy. However, the critical difference is that the new algorithm has to use with the topological minor test, which is more complicated than the subgraph isomorphism test.

Hofer *et al.* [52], as well as Parthasarathy and Coatney[83], make the observation that in many real world applications, a *fuzzy* match is needed, and not an exact match. As we demonstrate in our work, such fuzziness can be handled in our framework through the design of suitable relabeling functions.

To reduce the computational costs associated with enumerating frequent subgraphs, researchers have looked at generating *closed* [109], *maximal* [55] and *free-tree based* [92] frequent subgraph patterns. Such concepts can be naturally extended to handle frequent topological patterns as well. Further, several researchers have studied how to find efficient patterns in the tree dataset, such as an XML dataset [119]. Frequent topological patterns could be defined on tree datasets as well, and our algorithm is clearly capable of enumerating such patterns.

5.8 Conclusions

In this chapter, we have presented a novel framework for mining topological patterns in graph datasets. Based on the well known notion of a topological minor, we have designed efficient algorithms for mining such patterns. Additionally, our framework supports the notion of a user-defined relabeling function, which can be used to specify constraints and fuzzy matching criteria. We have demonstrated the effectiveness and scalability of the proposed algorithms on real and synthetic datasets. We have also reported on a case study where the framework has been used to identify topological structures from membrane protein structure data.

```
ExtendOuterEdges(Graph T)
 {* T is a Topological Structure *}
 E ← ∅;
 foreach (occ ∈ T.occurrencelist)
      G ← Graph(D, occ.tid);
      foreach (path p ∈ IndependentPath(G, occ, occ.δ.to))
           e ← Edge(p.from, p.to);
           if (e ∉ E)
                E ← E ∪ {e};
           InsertOccurrence(e, G, p);
 foreach (e ∈ E)
      if ( not Frequent(T ∪ {e}))
           E ← E − e;
 returnE;


Join(EdgeSet E1, Edge e2)
 E ← ∅;
 foreach (e1 ∈ E1)
      e.occurrencelist ← ∅;
      foreach ((l1, l2) : l1 ∈ e1.occurrencelist and
                          l2 ∈ e2.occurrencelist and
                          l1.parentID == l2.parentID)
           if (Independent(l1.path, l2.path))
                InsertOccurrence(e, l1.path);
      if (Frequent(e))
           E ← E ∪ {e};
 returnE;
```

Figure 5.5: Support Counting Procedures for Mining Topological Structures

VTSMining(Dataset D, Support θ, Bound l, h)
$\{* \; Find \; Frequent \; Single - Edge \; Topological \; Structures *\}$
$E \leftarrow FrequentEdgeTS(D, \theta, l, h);$
foreach $(e \in E)$
$VTreeTS(e);$

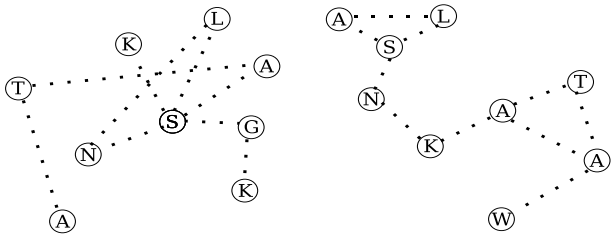
VTreeTS(Tree T)
$\{* \; New \; Outer \; Edges \; of \; T *\}$
$E \leftarrow ExtendOuterEdges(T);$
$[T]_{outer} \leftarrow [T]_{outer} \cup E;$
$\{* Tree \; Topological \; Structure \; Growing *\}$
foreach $(e : e \in [T]_{outer}$ **and**
$ValidExtension(T \cup \{e\}))$
$T_e \leftarrow T \cup \{e\};$
$[T_e]_{io} \leftarrow Join([T]_{io}, e);$
$VTreeTS(T_e);$
$\{* Enumerating \; Graph \; Topological \; Structures *\}$
$VGraphTS(T);$

VGraphTS(Graph G)
foreach $(e : e \in [G]_{inner}$ **and**
$CanonicalExtension(G \cup \{e\})$
$G_e \leftarrow G \cup \{e\};$
$[G_e]_{inner} \leftarrow Join([G]_{inner}, e);$
$VGraphTS(G_e);$

Figure 5.6: Algorithm Framework for Mining Topological Structures

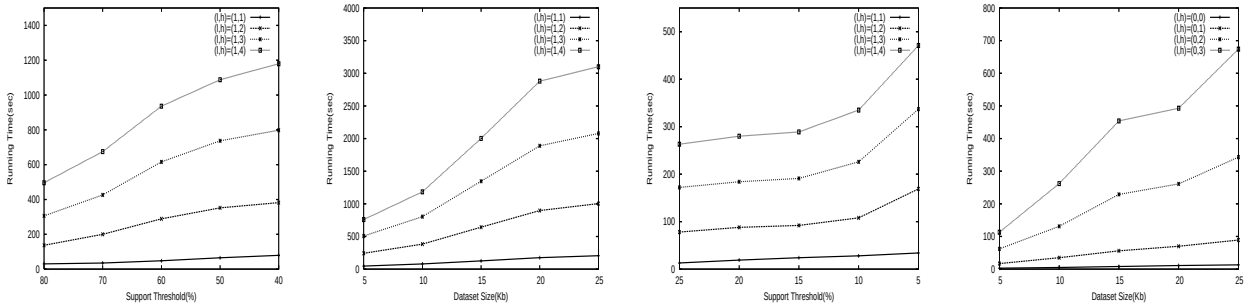
	A	B	C	D	E
A	(A|B)^1|D^2	A|B|C^2|E	?^{1}|?^{2}	$\emptyset$	D|E|GF
B	A|B|C^2|E	(A|B)^1|D^2	?^{1}|?^{2}	AB|BC|D	?^{1}|?^{2}
C	?^{1}|?^{2}	?^{1}|?^{2}	(A|B)^1|D^2	BB	?^{1}|?^{2}
D	$\emptyset$	BB	BA|CB|D	(A|B)^1|D^2	?^{1}|?^{2}
E	D|E|FG	?^{1}|?^{2}	?^{1}|?^{2}	?^{1}|?^{2}	(A|B)^1|D^2

Figure 5.7: Constraint Condition Table



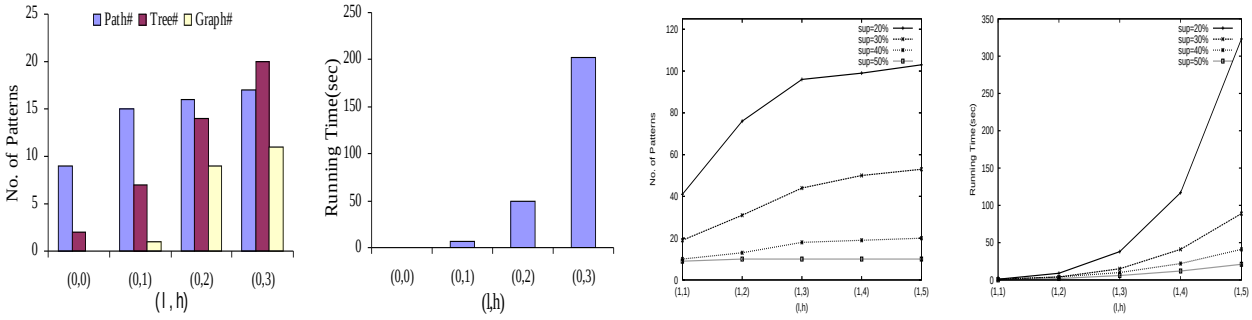
(a) frequent motif found in 1kqf (b) frequent motif found in 1m3x

Figure 5.8: Frequent Topological Structures Discovered by *TSMiner*



(a) (b) (c) (d)

Figure 5.9: (a) Varying Support(D10kV5) (b) Varying Dataset Size(D*kV5, Sup=40%)) (c)Varying Support (D10kV20) (d) Varying Dataset Size (D*kV20, Sup=20%)



(a) (b) (c) (d)

Figure 5.10: Chemical340 (a)No. of Patterns(Support=200) (b)Running Time(Support=200) (c)No. of Patterns(Varying Support) (d)Running Time(Varying Support)

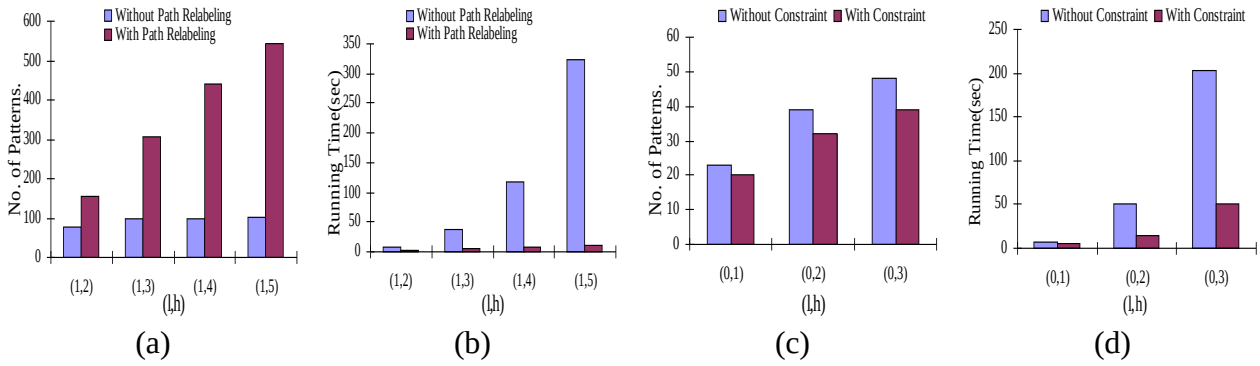


Figure 5.11: Relabeling with the Path Length on Chemical340 (Support=200) (a) No. of Patterns (b) Running Time; DFA Constraints on Chemical340 (Support=200) (c)No. of Patterns (d)Running Time

CHAPTER 6

CONTRIBUTIONS AND FUTURE WORK

In this thesis, we have studied three different problems in the field of frequent pattern mining, *mining multiple datasets*, *mining data streams*, and *mining graph datasets*. Our research has introduced new techniques and new algorithms to solve these problems, and also demonstrated their efficiency through detailed experimental evaluation as well as their applicability to real world problems. Specifically, our thesis contributions can be summarized as follows.

1. *Database Optimization for Mining Multiple Datasets:* We have modeled the problem of mining frequent patterns across multiple datasets to a query evaluation problem, where a set of simple mining algorithm servers as the basic operators. In particular, we built an intuitive model – M*-table* to formulate the query evaluation process and facilitate the query plan generation. Using the M-table, we have developed a set of greedy algorithms to generate efficient query plans for a single mining query. We were also able to utilize M-table to simultaneously optimize multiple queries and summarize/reuse the mining results of past mining queries. Our experimental results shown approximately an order of two speedup comparing with the naive method

without our optimization (more than an order of speedup from single query optimization and close to an order speedup from multiple query optimization together with a knowledgeable cache.)

2. *A New Algorithm for Mining Frequent Itemsets on Streaming Data:* Our new algorithm is inspired from Karp *et. al.*'s simple algorithm for finding frequent items from a sequence of items (the length of itemsets is equal to one). We have not only derived *StreamMining* to handle the frequent itemsets mining, but also developed a new bound to improve the quality of the mining results. Our detailed experimental evaluation has shown *StreamMining* is very accurate in practice, and is very memory efficient.

3. *A New Framework for Mining Topological Patterns on Graph Datasets:* Inspired by a mathematical concept *Topological Minor*, we have developed a new framework to mine *large-scale* topological patterns from graph datasets. This framework contains a new algorithm *VTSMining* to enumerate such patterns in a DFS fashion, and a new approach called *relabeling function* to perform constraint pattern mining. Our new framework has been successfully applied to a protein dataset and find active motifs which seem novel according to domain experts. The performance of our new framework has been validated on both real and synthetic datasets.

Many interesting problems have been raised in our research. Even though we are making some progress to tackle them, some of them remain open.

1. *Cost-based Mining Query Plan Generation:* Our current methods rely on heuristic and greedy algorithms to generate efficient mining query plans. However, in many situations, such methods are likely to fail. For example, the *support level* serves

as the main heuristic to estimate the cost of mining operators. Clearly, many other factors, such as the number of transactions, and the density of the datasets, also play very significant roles in determining the performance of the mining operation. Therefore, in order to generate the optimized mining-query plans, we need a cost-based query plan generation approach. This is similar to the traditional database query optimization. In particular, two difficult problems need to be answered for such an approach: 1) how should we associate the costs with different mining operators? 2) how to use dynamic programming to find the optimized ones?

2. *Approximate the Number of (Maximal) Frequent Itemsets:* One of the problem closely related with the cost estimation of a mining operator is to estimate the cardinality of its result sets - the number of (maximal) frequent itemsets. In other words, can we estimate the number of (maximal) frequent itemsets without enumerating them? The computational complexity of computing the exact number of frequent itemsets have been proved to be #P-hard [44], and computing the exact number of maximal itemsets are #P-complete [110]. However, this does not exclude the possibility that efficient polynomial-time algorithms exists to estimate the number of (maximal) frequent itemsets. Clearly, this problem has both theoretical and practical importance.

3. *Mining and Maintaining Maximal Frequent Itemsets over Data Streams:* Even several one-pass methods (including ours) have been proposed in mining frequent itemsets over data streams, the computational complexity of these algorithms are still very high and practically may not be efficient enough to handle very fast data streams. One way to reduce the computational cost is to only mine and maintain the maximal frequent itemsets (MFI). However, the difficulty of this problem is that if we

just maintain the information for MFI, it will be very hard to find a good estimate of the counts of the interior itemsets once the border of frequent itemsets shrinks. For example, assume itemset {a,b,c} is a current maximal frequent itemset. Now, after processing the new chunk, we know it becomes infrequent. Without loss of generality, we assume that {a,b}, {b,c} and {a,c} become potentially maximal frequent itemsets. However, because we do not record any information about these itemsets, it will be very hard to provide a reasonable estimation of how frequent these itemsets are.

To address this difficulty, we propose to maintain a concise frequency contour over frequent itemsets. In other words, we maintain the several MFI sets for different support levels. We call them as *contour sets*. Therefore, once an itemset does not appear in contour sets, it will be falling between two different MFI sets. In this case, we will take the greater support level between the two MFI sets as a frequency estimation. However, some technical issues arise when transforming this idea into an efficient algorithm, such as how to efficiently query an itemset that does not appear in the contour sets. and how many MFI sets should be built as well as how to build them efficiently.

4. *Efficiently Mining Frequent Large-Scale Structures from Structured Datasets:* As one of the first works to address mining frequent large-scale structures, we found many interesting problems which need further pay attention to. For example, we need to consider the *combinatorial-explosion* problem in enumerating frequent large-scale structures. This is referred to as the fact that if a subgraph is frequent, then any large-scale structure derived from such a subgraph is also frequent. Considering a *line-graph* with 5 vertices and 4 edges is frequent, the possible number of large-scale

structures can be derived from such a graph is 2^5 if the vertices have different labels, and the parameters of l and h to be 0 and 4, respectively. Therefore, we need a good strategy to deal with such combinatorial explosion problem. Another problem is that in our current research, we use the model based on the topological minor to mine large-scale structures. Therefore, a natural question to ask is whether such a model is general enough to describe other types of large-scale structures. We believe new models, consequently new algorithms, might be necessary to mine new families of frequent large-scale structures.

We are planing to work on these problems in our future research.

BIBLIOGRAPHY

[1] Integrated public use microdata series. http://http://www.ipums.umn.edu/usa/index.html.

[2] Foto Afrati, Aristides Gionis, and Heikki Mannila. Approximating a collection of frequent sets. In *KDD '04: Proceedings of the 2004 ACM SIGKDD international conference on Knowledge discovery and data mining*, 2004.

[3] R. Agrawal, H. Mannila, R. Srikant, H. Toivonent, and A. Inkeri Verkamo. Fast discovery of association rules. In U. Fayyad and et al, editors, *Advances in Knowledge Discovery and Data Mining*, pages 307–328. AAAI Press, Menlo Park, CA, 1996.

[4] R. Agrawal and J. Shafer. Parallel mining of association rules. *IEEE Trans. on Knowledge and Data Engg.*, 8(6):962–969, December 1996.

[5] R. Agrawal and R. Srikant. Fast algorithms for mining association rules. In *Proc. of Int. conf. Very Large DataBases (VLDB'94)*, pages 487–499, Santiago,Chile, September 1994.

[6] Rakesh Agrawal, Tomasz Imielinski, and Arun Swami. Mining association rules between sets of items in large databases. In *Proceedings of the 1993 ACM SIGMOD Conference*, pages 207–216, May 1993.

[7] Rakesh Agrawal and Ramakrishnan Srikant. Fast algorithms for mining association rules in large databases. In *Proceedings of the 20th International Conference on Very Large Data Bases*, 1994.

[8] Rakesh Agrawal and Ramakrishnan Srikant. Mining sequential patterns. In *Proceedings of the Eleventh International Conference on Data Engineering*, 1995.

[9] A. V. Aho, C. Beeri, and J. D. Ullman. The theory of joins in relational databases. *ACM Trans. Database Syst.*, 4(3), 1979.

[10] H. Andrade, T. Kurc, A. Sussman, and J. Saltz. Multiple query optimization for data analysis applications on clusters of smps. In *In Proceedings of the 2nd International Symposium on Cluster Computing and the Grid (CCGRID)*, 2002.

[11] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom. Models and Issues in Data Stream Systems. In *Proceedings of the 2002 ACM Symposium on Principles of Database Systems (PODS 2002) (Invited Paper)*. ACM Press, June 2002.

[12] Stephen D. Bay and Michael J. Pazzani. Detecting group differences: Mining contrast sets. *Data Min. Knowl. Discov.*, 5(3):213–246, 2001.

[13] Hendrik Blockeel and Michěle Sebag. Scalability and efficiency in multi-relational data mining. *SIGKDD Explor. Newsl.*, 5(1):17–30, 2003.

[14] Christan Borgelt. Apriori implementation. http://fuzzy.cs.Uni-Magdeburg.de/ borgelt/Software. Version 4.08.

[15] Endre Boros, Vladimir Gurvich, Leonid Khachiyan, and Kazuhisa Makino. On the complexity of generating maximal frequent and minimal infrequent sets. In *Symposium on Theoretical Aspects of Computer Science*, pages 133–141, 2002.

[16] Cristian Bucila, Johannes Gehrke, Daniel Kifer, and Walker White. Dualminer: a dual-pruning algorithm for itemsets with constraints. In *Proceedings of the eighth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 42–51, 2002.

[17] Doug Burdick, Manuel Calimlim, and Johannes Gehrke. Mafia: A maximal frequent itemset algorithm for transactional databases. In *Proceedings of 17th ICDE*, April 2001.

[18] T. Calders and J. Wijsen. On monotone data mining languages. In *Proc. of International Workshop on Database Programming Languages (DBPL)*, pages 119–132, 2001.

[19] Surajit Chaudhuri. An overview of query optimization in relational systems. In *PODS '98: Proceedings of the seventeenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, 1998.

[20] Surajit Chaudhuri, Usama M. Fayyad, and Jeff Bernhardt. Scalable classification over sql databases. In *Proceedings of the 15th International Conference on Data Engineering, 23-26 March 1999, Sydney, Austrialia*, pages 470–479. IEEE Computer Society, 1999.

[21] D. Cheung, J. Han, V. Ng, A. Fu, and Y.Fu. A fast distributed algorithm for mining association rules. In *4th Intl. Conf. Parallel and Distributed Info. Systems*, December 1996.

[22] Diane J. Cook and Lawrence B. Holder. Substructure discovery using minimum description length and background knowledge. *Journal of Artificial Intelligence Research*, 1:231–255, 1994.

[23] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. McGraw Hill, 1990.

[24] Luc De Raedt. A perspective on inductive databases. *SIGKDD Explor. Newsl.*, 4(2):69–77, 2002.

[25] L. Dehaspe, H. Toivonen, and R. D. King. Finding frequent substructures in chemical compounds. In R. Agrawal, P. Stolorz, and G. Piatetsky-Shapiro, editors, *4th International Conference on Knowledge Discovery and Data Mining*, pages 30–36. AAAI Press., 1998.

[26] Reinhard Diestel. *Graph Theory*. Springer-Verlag, 2000.

[27] A. Dobra, J. Gehrke, M. Garofalakis, and R. Rastogi. Processing complex aggregate queries over data streams. In *Proc. of the 2002 ACM SIGMOD Intl. Conf. on Management of Data*, June 2002.

[28] P. Domingos and G. Hulten. In *Proceedings of the ACM Conference on Knowledge and Data Discovery (SIGKDD), TITLE = Mining High-Speed Data Streams, YEAR = 2000, text = "P. domingos and G. Hulten. Mining High-Speed Data Streams. SIGKDD, 2000."*.

[29] Guozhu Dong and Jinyan Li. Efficient mining of emerging patterns: discovering trends and differences. In *Proceedings of the fifth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 43–52, 1999.

[30] Sašo Džeroski. Multi-relational data mining: an introduction. *SIGKDD Explor. Newsl.*, 5(1):1–16, 2003.

[31] Mohammad El-Haji and Osmar R. Zaiane. Inverted Matrix: Efficient Discovery of Frequent Items in Large Datasets in the Context of Interactive Mining. In *Proceedings of the ACM SIGKDD Conference on Knowledge Discovery and Data Mining*. ACM Press, 2003.

[32] Alexandre Evfimievski, Ramakrishnan Srikant, Rakesh Agrawal, and Johannes Gehrke. Privacy preserving mining of association rules. In *KDD '02: Proceedings of the eighth ACM SIGKDD international conference on Knowledge discovery and data mining*, 2002.

[33] feng Yan, X. Jasmine Zhou, and Jiawei Han. Mining closed relational graphs with connectivity constraints. In *ICDE*, 2005.

[34] Leonard P. Freedman, Keith R. Yamamoto, Ben F. Luisi, and Paul B Sigler. More fingers in hand. *Cell*, 54(4):444, 1988.

[35] J. Gehrke, F. Korn, and D. Srivastava. On computing correlated aggregates over continual data streams. In *In Proceedings of SIGMOD*, 2001.

[36] C. Giannella, Jiawei Han, Jian Pei, Xifeng Yan, and P. S. Yu. Mining Frequent Patterns in Data Streams at Multiple Time Granularities. In *Proceedings of the NSF Workshop on Next Generation Data Mining*, November 2002.

[37] F. Giannotti, G. Manco, D. Pedreschi, and F. Turini. Experiences with a logic-based knowledge discovery support environment. In *In Proc. 1999 ACM SIGMOD Workshop on Research Issues in Data Mining and Knowledge Discovery (DMKD 1999)*.

[38] Phillip B. Gibbons and S. Tirthapura. Estimating simple functions on the union of data streams. In *Proc. of the 2001 ACM Symp. on Parallel Algorithms and Architectures*, pages 281–291. ACM Press, August 2001.

[39] Bart Goethals. Fp-tree implementation. http://www.cs.helsinki.fi/u/goethals/software/index.html. Version Last Updated April 2003.

[40] Bart Goethals and Jan Van den Bussche. On supporting interactive association rule mining. In *Proceedings of the Second International Conference on Data Warehousing and Knowledge Discovery*, volume 1874 of *Lecture Notes in Computer Science*. Springer, 2000.

[41] Bart Goethals and Mohammed J. Zaki. Workshop Report on Workshop on Frequent Itemset Mining Implementations (FIMI). 2003.

[42] Karam Gouda and Mohammed Javeed Zaki. Efficiently mining maximal frequent itemsets. In *ICDM '01: Proceedings of the 2001 IEEE International Conference on Data Mining*, 2001.

[43] S. Guha, N. Mishra, R. Motwani, and L. O'callaghan. Clustering data streams. In *In Proceedings of Foundations of Computer Science*, 2000.

[44] Dimitrios Gunopulos, Roni Khardon, Heikki Mannila, Sanjeev Saluja, Hannu Toivonen, and Ram Sewak Sharma. Discovering all most specific sentences. *ACM Trans. Database Syst.*, 28(2), 2003.

[45] Alon Y. Halevy. Answering queries using views: A survey. *The VLDB Journal*, 10(4), 2001.

[46] E-H. Han, G. Karypis, and V. Kumar. Scalable parallel datamining for association rules. *IEEE Transactions on Data and Knowledge Engineering*, 12(3), May / June 2000.

[47] J. Han, Y. Fu, W. Wang, K. Koperski, and O. R. Zaiane. Dmql: A data mining query language for relational databases. In *In Proc. 1996 SIGMOD 96 Workshop on Research Issues on Data Mining and Knowledge Discovery (DMKD 96)*, pages 27–33, Montreal, Canada, Jun 1996.

[48] J. Han, J. Pei, and Y. Yin. Mining frequent patterns without candidate generation. In *Proceedings of the ACM SIGMOD Conference on Management of Data*, 2000.

[49] Jiawei Han, Laks V. S. Lakshmanan, and Raymond T. Ng. Constraint-based, multidimensional data mining. *Computer*, 32(8):46–50, 1999.

[50] C. Hidber. Online Association Rule Mining. In *Proceedings of ACM SIGMOD Conference on Management of Data*, pages 145–156. ACM Press, 1999.

[51] Jochen Hipp and Ulrich Güntzer. Is pushing constraints deeply into the mining algorithms really what we want?: an alternative approach for association rule mining. *SIGKDD Explor. Newsl.*, 4(1):50–55, 2002.

[52] H. Hofer, C. Borgelt, and M. R. Berthold. Large scale mining of molecular fragments with wildcards. In *Advances in Intelligent Data Analysis V*, pages 380–389, 2003.

[53] Jun Huan, Wei Wang, Deepak Bandyopadhyay, Jack Snoeyink, Jan Prins, and Alexander Tropsha. Mining protein family-specific residue packing patterns from protein structure graphs. In *Eighth International Conference on Research in Computational Molecular Biology (RECOMB)*, pages 308–315, 2004.

[54] Jun Huan, Wei Wang, Jan Prins, and Jiong Yang. Spin: mining maximal frequent subgraphs from graph databases. In *KDD '04: Proceedings of the 2004 ACM SIGKDD international conference on Knowledge discovery and data mining*, 2004.

[55] Jun Huan, Wei Wang, Jan Prins, and Jiong Yang. Spin: mining maximal frequent subgraphs from graph databases. In *KDD*, pages 581–586, 2004.

[56] G. Hulten, L. Spencer, and P. Domingos. Mining time-changing data streams. In *Proceedings of the ACM Conference on Knowledge and Data Discovery (SIGKDD)*, 2001.

[57] T. Imielinski and A. Virmani. Msql: a query language for database mining. In *Data Mining and Knowledge Discovery*, pages 3:393–408, 1999.

[58] Tomasz Imielinski and Heikki Mannila. A database perspective on knowledge discovery. *Commun. ACM*, 39(11):58–64, 1996.

[59] Akihiro Inokuchi, Takashi Washio, and Hiroshi Motoda. An apriori-based algorithm for mining frequent substructures from graph data. In *Principles of Knowledge Discovery and Data Mining (PKDD2000)*, pages 13–23, 2000.

[60] Akihiro Inokuchi, Takashi Washio, and Hiroshi Motoda. Complete mining of frequent patterns from graphs: Mining graph data. *Mach. Learn.*, 50(3):321–354, 2003.

[61] Szymon Jaroszewicz and Dan A. Simovici. Interestingness of frequent itemsets using bayesian networks as background knowledge. In *KDD '04: Proceedings of the 2004 ACM SIGKDD international conference on Knowledge discovery and data mining*, 2004.

[62] Ruoming Jin and Gagan Agrawal. An algorithm for in-core frequent itemset mining on streaming data. Technical Report OSU-CISRC-2/04-TR14, Ohio State University, 2004.

[63] Ruoming Jin and Gagan Agrawal. A systematic approach for optimizing complex mining tasks on multiple datasets. Technical report, Department of Computer Science and Engineering, OSU, 2004.

[64] T. Johnson, Laks V. S. Lakshmanan, and Raymond T. Ng. The 3w model and algebra for unified data mining. In *Proceedings of Interational Conference on Very Large DataBases (VLDB)*, 2002.

[65] Richard M. Karp, Christos H. Papadimitrious, and Scott Shanker. A Simple Algorithm for Finding Frequent Elements in Streams and Bags. Available from http://www.cs.berkeley.edu/ christos/iceberg.ps, 2002.

[66] Stefan Kramer, Luc De Raedt, and Christoph Helma. Molecular feature mining in hiv data. In *Proceedings of the seventh ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 136–143, 2001.

[67] Michihiro Kuramochi and George Karypis. Frequent subgraph discovery. In *ICDM '01: Proceedings of the 2001 IEEE International Conference on Data Mining*, pages 313–320, 2001.

[68] Laks V. S. Lakshmanan, Raymond Ng, Jiawei Han, and Alex Pang. Optimization of constrained frequent set queries with 2-variable constraints. In *Proceedings of the 1999 ACM SIGMOD international conference on Management of data*, pages 157–168, 1999.

[69] Sau Dan Lee and Luc De Raedt. An algebra for inductive query evaluation. In *Proc. The Third IEEE International Conference on Data Mining (ICDM'03)*, pages 147–154, Melbourne, Florida, USA, November 2003.

[70] G. S. Manku and R. Motwani. Approximate Frequency Counts Over Data Streams. In *Proceedings of International Conference on Very Large DataBases (VLDB)*, pages 346 – 357, September 2002.

[71] G. S. Manku and R. Motwani. Approximate Frequency Counts Over Data Streams. In *Proceedings of Conference on Very Large DataBases (VLDB)*, pages 346 – 357, 2002.

[72] Brendan McKay. Practical graph isomorphism. *Congr. Numer.*, 30:45–87, 1981.

[73] Thorsen Meinl, Christian Borgelt, Michael R. Berthold, and Michael Philippsen. Mining fragments with fuzzy chains in molecular databases. In *Second International Workshop on Mining Graphs, Trees and Sequences (MGTS2004)*, 2004.

[74] R. Meo, G. Psaila, and S. Ceri. A new sql-like operator for mining association rules. In *In Proc. of International Conference on Very Large Data Bases (VLDB)*, pages 122–133, Bombay, India, 1996.

[75] Biswadeep Nag, Prasad Deshpande, and David J. DeWitt. Using a knowledge cache for interactive discovery of association rules. In *Knowledge Discovery and Data Mining*, pages 244–253, 1999.

[76] Raymond T. Ng, Laks V. S. Lakshmanan, Jiawei Han, and Alex Pang. Exploratory mining and pruning optimizations of constrained associations rules. In *SIGMOD '98: Proceedings of the 1998 ACM SIGMOD international conference on Management of data*, 1998.

[77] Raymond T. Ng, Laks V. S. Lakshmanan, Jiawei Han, and Alex Pang. Exploratory mining and pruning optimizations of constrained associations rules. In *Proceedings of the 1998 ACM SIGMOD international conference on Management of data*, pages 13–24, 1998.

[78] Siegfried Nijssen and Joost N. Kok. A quickstart in frequent structure mining can make a difference. In *KDD*, pages 647–652, 2004.

[79] M. Otey, S. Parthasarathy, A. Ghoting, G. Li, S. Narravula, and D. Panda. Towards nic-based intrusion detection. In *Proceedings of the ninth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 723–728, 2003.

[80] H Palsdottir and C Hunte. Lipids in membrane protein structures. *BBA*, 1666:2–18, 2004.

[81] J. S. Park, M. Chen, and P. S. Yu. An effecitive hash based algorithm for mining association rules. In *ACM SIGMOD Intl. Conf. Management of Data*, May 1995.

[82] Jooseok Park and Arie Segev. Using common subexpressions to optimize multiple queries. In *Proceedings of the Fourth International Conference on Data Engineering*, 1988.

[83] S. Parthasarathy and M. Coatney. Efficient discovery of common substructures in macromolecules. *IEEE International Conference on Data Mining*, pages 362–369, 2002.

[84] Nicolas Pasquier, Yves Bastide, Rafik Taouil, and Lotfi Lakhal. Discovering frequent closed itemsets for association rules. In *ICDT '99: Proceeding of the 7th International Conference on Database Theory*, 1999.

[85] Jian Pei and Jiawei Han. Constrained frequent pattern mining: a pattern-growth view. *SIGKDD Explor. Newsl.*, 4(1):31–39, 2002.

[86] Jian Pei, Jiawei Han, and Laks V. S. Lakshmanan. Mining frequent item sets with convertible constraints. In *Proceedings of the 17th International Conference on Data Engineering*, pages 433–442, 2001.

[87] Chang-Shing Perng, Haixun Wang, Sheng Ma, and Joseph L. Hellerstein. Discovery in multi-attribute data with user-defined constraints. *SIGKDD Explor. Newsl.*, 4(1):56–64, 2002.

[88] Luc De Raedt, Manfred Jaeger, Sau Dan Lee, and Heikki Mannila. A theory of inductive query answering (extended abstract). In *Proc. The 2002 IEEE International Conference on Data Mining (ICDM'02)*, pages 123–130, Maebashi, Japan, December 2002.

[89] S. Rizvi and J. Haritsa. Maintaining data privacy in association rule mining. In *Proceedings of the 28th Conference on Very Large Data Base (VLDB'02),*, 2002.

[90] Jr. Roberto J. Bayardo. Efficiently mining long patterns from databases. In *SIGMOD '98: Proceedings of the 1998 ACM SIGMOD international conference on Management of data*, 1998.

[91] Prasan Roy, S. Seshadri, S. Sudarshan, and Siddhesh Bhobe. Efficient and extensible algorithms for multi query optimization. *SIGMOD Rec.*, 29(2), 2000.

[92] Ulrich Ruckert and Stefan Kramer. Frequent free tree discovery in graph data. In *SAC '04: Proceedings of the 2004 ACM symposium on Applied computing*, pages 564–570, 2004.

[93] S. Sarawagi, S. Thomas, and R. Agrawal. Integrating association rule mining with relational database systems: Alternatives and implications. In *Proceedings of the 1998 ACM SIGMOD international conference on Management of data*, 1998.

[94] A. Savasere, E. Omiecinski, and S.Navathe. An efficient algorithm for mining association rules in large databases. In *21th VLDB Conf.*, 1995.

[95] Ashoka Savasere, Edward Omiecinski, and Shamkant B. Navathe. An efficient algorithm for mining association rules in large databases. In *VLDB '95: Proceedings of the 21th International Conference on Very Large Data Bases*, 1995.

[96] Timos K. Sellis. Multiple-query optimization. *ACM Trans. Database Syst.*, 13(1), 1988.

[97] Kyuseok Shim, Timos Sellis, and Dana Nau. Improvements on a heuristic algorithm for multiple-query optimization. *Data Knowl. Eng.*, 12(2), 1994.

[98] Ramakrishnan Srikant, Quoc Vu, and Rakesh Agrawal. Mining association rules with item constraints. In David Heckerman, Heikki Mannila, Daryl Pregibon, and Ramasamy Uthurusamy, editors, *Proc. 3rd Int. Conf. Knowledge Discovery and Data Mining, KDD*, pages 67–73, 1997.

[99] A. Srinivasan, R.D. King, S.H. Muggleton, and M. Sternberg. The predictive toxicology evaluation challenge. In *the Fifteenth International Joint Conference on Artificial Intelligence (IJCAI-97)*, pages 1–6. Morgan-Kaufmann, 1997.

[100] H. Toivonen. Sampling large databases for association rules. In *22nd VLDB Conf.*, 1996.

[101] Dick Tsur, Jeffrey D. Ullman, Serge Abiteboul, Chris Clifton, Rajeev Motwani, Svetlozar Nestorov, and Arnon Rosenthal. Query flocks: a generalization of association-rule mining. In *Proceedings of the 1998 ACM SIGMOD international conference on Management of data*, pages 1–12, 1998.

[102] J.D. Ullman and J. Widom. *A First Course in Database Systems*. Prentice Hall, Upper Saddle River, New Jersey, second edition, 2002.

[103] Jaideep Vaidya and Chris Clifton. Privacy preserving association rule mining in vertically partitioned data. In *KDD '02: Proceedings of the eighth ACM SIGKDD international conference on Knowledge discovery and data mining*, 2002.

[104] Chen Wang, Wei Wang, Jian Pei, Yongtai Zhu, and Baile Shi. Scalable mining of large disk-based graph databases. In *KDD*, pages 316–325, 2004.

[105] Takashi Washio and Hiroshi Motoda. State of the art of graph-based data mining. *SIGKDD Explor. Newsl.*, 5(1):59–68, 2003.

[106] Geoffrey I. Webb, Shane Butler, and Douglas Newlands. On detecting differences between groups. In *Proceedings of the ninth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 256–265, 2003.

[107] Xifeng Yan and Jiawei Han. gspan: Graph-based substructure pattern mining. In *ICDM '02: Proceedings of the 2002 IEEE International Conference on Data Mining (ICDM'02)*, page 721, 2002.

[108] Xifeng Yan and Jiawei Han. Closegraph: mining closed frequent graph patterns. In *KDD '03: Proceedings of the ninth ACM SIGKDD international conference on Knowledge discovery and data mining*, 2003.

[109] Xifeng Yan and Jiawei Han. Closegraph: mining closed frequent graph patterns. In *KDD '03: Proceedings of the ninth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 286–295, 2003.

[110] Guizhen Yang. The complexity of mining maximal frequent itemsets and maximal frequent patterns. In *KDD '04: Proceedings of the 2004 ACM SIGKDD international conference on Knowledge discovery and data mining*, 2004.

[111] X. Yin, J. Han, J. Yang, and P. S. Yu. Crossmine: Efficient classification across multiple database relations. In *Proc. 2004 Int. Conf. on Data Engineering (ICDE'04)*, Boston,MA, March 2004.

[112] Y.N.Law, C.R.Luo, H.Wang, and C.Zaniol. Atlas: a turing complete extension of sql for data mining applications and streams. In *Posters of the 2003 ACM SIGMOD international conference on Management of data*, 2003.

[113] Jeffrey Xu Yu, Zhihong Chong, Hongjun Lu, and Aoying Zhou. False positive or false negative: Mining frequent itemsets from high speed transactional data streams. In *Proceedings of the 28th International Conference on Very Large Data Bases (VLDB)*, Toronto, Canada, Aug 2004.

[114] M. Zaki and M. Ogihara. Theoretical foundations of association rules. In *Proceedings of 3 rd SIGMOD'98 Workshop on Research Issues in Data Mining and Knowledge Discovery (DMKD'98)*, Seattle, Washington, USA, June 1998.

[115] M.J. Zaki. Parallel and distributed association mining: A survey. *IEEE Concurrency (Special Issue on Data Mining)*, 1999.

[116] M.J. Zaki, M. Ogihara, S. Parthasarathy, and W.Li. Parallel data mining for association rules on shared-memory multi-processors.esso. In *Supercomputing'96*, November 1996.

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

[118] Mohammed J. Zaki. Efficiently mining frequent trees in a forest. In *Proceedings of the eighth ACM SIGKDD international conference on Knowledge discovery and data mining*, 2002.

[119] Mohammed J. Zaki. Efficiently mining frequent trees in a forest. In *KDD '02: Proceedings of the eighth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 71–80, 2002.

[120] Mohammed J. Zaki and Charu C. Aggarwal. Xrules: an effective structural classifier for xml data. In *KDD '03: Proceedings of the ninth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 316–325, 2003.

[121] Yihong Zhao, Prasad M. Deshpande, Jeffrey F. Naughton, and Amit Shukla. Simultaneous optimization and evaluation of multiple dimensional queries. *SIGMOD Rec.*, 27(2), 1998.

[122] Z. Zheng, R. Kohavi, and L. Mason. Real World Performance of Association Rule Algorithms. In *Proceedings of the ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, pages 401–406. ACM Press, August 2001.