A Systematic Approach for Optimizing Complex Mining Tasks on Multiple Databases

Ruoming Jin Gagan Agrawal
Department of Computer Science and Engineering
The Ohio State University
Columbus, OH, 43220
{jinr,agrawal}@cse.ohio-state.edu

Abstract

Many real world applications involve not just a single dataset, but a view of multiple datasets. These datasets may be collected from different sources and/or at different time instances. In such scenarios, comparing patterns or features from different datasets and understanding their relationships can be an extremely important part of the KDD process. This paper considers the problem of optimizing a mining task over multiple datasets, when it has been expressed using a high-level interface. Specifically, we make the following contributions: 1) We present an SQL-based mechanism for querying frequent patterns across multiple datasets, and establish an algebra for these queries. 2) We develop a systematic method for enumerating query plans and present several algorithms for finding optimized query plan which reduce execution costs. 3) We evaluate our algorithms on real and synthetic datasets, and show up to an order of magnitude performance improvements.

1 Introduction

Within the last decade, data mining has emerged as an important component of databases and information systems. A large body of research exists on algorithms for a variety of data mining tasks, targeting a variety of applications, data types, and execution environments.

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) [16, 19]. 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 [14, 18, 25, 36] and implementing mining algorithms in a database system [9, 30]. Logic and algebra based methods have also been proposed to model the mining process [7, 13, 20]. The subfield of constraint association mining allows mining of *interesting* association rules by taking of a variety of constraint conditions as input [6, 22, 27, 31].

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 [8]. 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, reasonable cost models are not available for a given mining operator.

In this paper, 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 estab-

lish 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 [15].

To summarize, this paper 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 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 [28]. 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 [28]:

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.

Dataset A_1		Dataset A_2	
TransID	Items	TransID	Items
1	{1, 2, 5}	1	{1, 2, 4, 5}
2	{2, 4}	2	{2, 3, 5}
3	{1, 2, 5}	3	{1, 2, 5 }
4	{1, 3, 4 }	4	{1, 2, 3}
5	{2, 3, 4}	5	{1, 3 }
6	{1, 3,4}	6	{3, 4}
7	{1, 2 }		
8	{1, 2, 3, 4, 5 }		

Table 1. Datasets A_1 and A_2

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 [24, 17]. 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.

In order to simplify our discussion, we will focus on frequent itemset mining tasks only in the rest of this paper. Because the down-closure property is applicable to other patterns as well, our work can be easily adapted to the tasks involving such patterns.

3 SQL Extensions and Algebra for Mining Across Multiple Datasets

In this section, we first introduce an SQL based mechanism for querying frequent itemsets across multiple datasets (Subsection 3.1). Then, we establish an algebra for expressing the information required to answer such a mining query (Subsection 3.2). Finally, we discuss the mapping from a mining query in its SQL format to an algebra expression (Subsection 3.3).

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.

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 1. The set of distinct items in the two datasets, $Item$, is $\{1, 2, 3, 4, 5\}$. Table 2 contains a portion of the F table for the datasets A_1 and A_2.

I	A_1	A_2
{1}	6/8	4/6
{2}	6/8	4/6
{3}	4/8	4/6
{4}	6/8	2/6
{5}	3/8	3/6
{1, 2}	4/8	3/6
{1, 3}	3/8	2/6
:	:	:
{1, 2, 3, 4, 5}	1/8	0

Table 2. 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 Q1:
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 AND F.D ≥ 0.05)
   OR (F.C ≥ 0.1 AND F.D ≥ 0.1 AND
   (F.A ≥ 0.05 OR F.B ≥ 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).

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 3 shows the results of SF operator on the datasets A_1 and A_2 (shown in Table 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* [33]. 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 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.*

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 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.

$SF(A_1, 0.5)$		$SF(A_2, 0.4)$	
I	A_1	I	A_2
{1}	6/8	{1}	4/6
{2}	6/8	{2}	4/6
{3}	4/8	{3}	4/6
{4}	6/8	{5}	3/6
{1,2}	4/8	{1,2}	3/6
{3,4}	4/8		

Table 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
{1}	6/8	4/6	{1}	6/8	4/6
{2}	6/8	4/6	{2}	6/8	4/6
{3}	4/8	4/6	{3}	4/8	4/6
{1,2}	4/8	3/6	{4}	6/8	$\circ$
			{5}	$\circ$	3/6
			{1,2}	4/8	3/6
			{3,4}	4/8	$\circ$

Table 4. Intersection and Union Operation

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.

$$(SF(A, 0.1) \sqcap SF(B, 0.1)) \sqcap SF(D, 0.05) \Rightarrow F_1$$
$$\sqcup (SF(A, 0.05) \sqcap SF(C, 0.1) \sqcap SF(D, 0.1)) \Rightarrow F_2$$
$$\sqcup (SF(B, 0.0.5) \sqcap SF(C, 0.1) \sqcap SF(D, 0.1)) \Rightarrow F_3$$

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.

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.

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 both 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*.

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.

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 5, and the last in Section 6.

5 Query Plan Generation

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.

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

Table 5. M Table for the 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 6. M Table for the query $\mathcal{Q}$

As an example, the M table for the query $\mathcal{Q}_1$ has 4 rows and 3 columns and is shown in Table 5.

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 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 6 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 non-colored (white). The operators, SF, CF, and GF, can color a number of non-empty cells black (shaded). The query evaluation process is complete when all non-empty cells are colored black.

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 non-colored table M of the query $\mathcal{Q}$. Table 7 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 black. In our example, the first operator, $SF(A, 0.05)$, will turn the cells $M_{1,1}$, $M_{1,2}$, and $M_{1,4}$ black.

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 black cells, and let S be the set of columns where these black 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 black. In our running example, the third operator

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

picks the black 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 $M_{2,2}$ black.

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

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 black. There are different costs associated with each of them. The next subsection addresses the problem of finding efficient query evaluation plans.

	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 7. Colored M Table for the query $\mathcal{Q}$

5.2 New Query Plans

For a given M table with m rows and t columns, the total number of possible query plans using only SF and CF operators is $O((\sum_{i=1}^{i=m} j_i)! \times 2^{\sum_{i=1}^{i=m} j_i})$, 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 proceeding it. Though this is somewhat similar to the *Join* optimization problem in the traditional databases [3], 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 partition 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.

5.2.1 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 black 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 black cell. In the *second* phase, we use the $CF(A_j, \alpha, X)$ operators to color 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 black 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* [10].

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. Therefore, 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.

Approach for Phase Two: We can use either of the two greedy algorithms, *Algorithm 1* and *Algorithm 2*, which are listed in the Figure 1. 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.

Input: table M after phase-one coloring
Algorithm 1
Find datasets whose corresponding rows has non-colored cells;
For each row, find the lowest support level among non-colored 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
Find the rows having the non-colored cells with highest support among the non-colored cells in the entire table;
Invoke the CF operator to color these cells in these rows;
Repeat the above steps until all cells are colored.

Figure 1. 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. Using GF operator for Phase One

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

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

The second algorithm gives the following query plan.

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

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

5.2.2 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, uses the GF operator in a efficient way. It results in the following query plan for our example query:

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

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 6.1, we introduce these two requirements, and the algebra extensions to capture these requirements. In Subsection 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.

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

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:

$$
\begin{aligned}
& (A \geq 0.1 \wedge B \geq 0.1 \wedge C < 0.05 \wedge D < 0.05) \\
\vee & (C \geq 0.1 \wedge D \geq 0.1 \wedge A < 0.05 \wedge B < 0.05)
\end{aligned}
$$

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

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

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 $F_{\mathcal{C}'}$. 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:

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

	Query Conditions	CSET
Q_1	$A \geq \alpha_1 \wedge B \geq \alpha_1 \wedge C \geq \alpha_2 \wedge D \geq \alpha_2$	
Q_2	$(A \geq \alpha_1 \wedge B < \alpha_2) \vee (B \geq \alpha_1 \wedge A < \alpha_1)$	{A}
Q_3	$(A \geq \alpha_1 \wedge B \geq \alpha_1 \wedge C < \alpha_2 \wedge D < \alpha_2) \vee$ $(C \geq \alpha_1 \wedge D \geq \alpha_1 \wedge A < \alpha_2 \wedge B < \alpha_2)$	{C}
Q_4	$(A \geq \alpha_1 \vee B \geq \alpha_1 \vee C \geq \alpha_1) \wedge D < \alpha_2$	{D}
Q_5	$A \geq \alpha_1 \wedge B \geq \alpha_1 \wedge C \geq \alpha_1 \wedge D < \alpha_2$	{D}
Q_6	$(A \geq \alpha_1 \wedge B < \alpha_2 \wedge C < \alpha_2 \wedge D < \alpha_2) \vee$ $(B \geq \alpha_1 \wedge A < \alpha_2 \wedge C < \alpha_2 \wedge D < \alpha_2) \vee$ $(C \geq \alpha_1 \wedge A < \alpha_2 \wedge B < \alpha_2 \wedge D < \alpha_2) \vee$	{D}

Table 8. Test Queries for Our Experiments

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

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

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 6) used in Section 5.

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.
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.

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

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 9. Performance (in seconds) on IPUMS 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 10. Performance (in seconds) on DARPA datasets

of the well-known Apriori algorithm [5] 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.

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.

7.2 Datasets

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

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 11. 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 12. Performance (in seconds) on QUEST datasets with query parameters $\alpha_1 = 0.25\%$ and $\alpha_2 = 0.08\%$

DARPA's Intrusion Detection: The second group of datasets is derived from the first three weeks of *tcpdump* data from the DARPA data sets [28]. 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 [2]. 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 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.

7.3 Test Queries

Our experiments use six different queries, which are listed in the Table 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 mo-

tivating example on finding the signature itemsets for network intrusion, presented in Section 2. The neptune, smurf, satan, and normal datasets are labeled as the datasets A, B, C, and D, respectively. Further, in the Table 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).

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 9 presents the running time for the first three queries on IPUMS datasets. Table 10 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 11 and 12. 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.

8 Related Work

Much research has been conducted to provide database support for mining operations. This includes extending the database query languages to support mining tasks [14, 18, 25], implementing data mining algorithms on a relational database system [30, 9], and applying user-defined functions (UDFs) to express data mining tasks [36]. However, all of these efforts focus on mining a single dataset with relatively simple conditions. Similarly, constraint frequent itemset mining has been used to guide the user to discover useful information and speedup mining process on a single dataset [6, 22, 26, 27, 31]. In particular, the algorithms for constraint frequent itemset mining 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.

Raedt and his colleagues have studied the generalized inductive query evaluation problem [21, 23]. 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* [32]. 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 [12, 35] 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 [4, 11, 34]. 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.

9 Conclusions

The work presented in this paper 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.

Our future work will concentrate on several issues that remain open. First, providing cost functions for mining operators is an open and important issue. The cost of a mining operator depends on many factors, including the mining algorithm, the size of the datasets, and many other characteristics of the transaction datasets, such as *density* [29]. We are currently working on methods to estimate such costs. Second, our techniques will need some modifications for dealing with *evolving* datasets, or where there is a temporal order between the datasets. Finally, research is needed to incorporate other conditions, such as those defined in constraint itemsets mining, and apply other mining operators, such as the *maximal* frequent itemset operator.

References

[1] Integrated public use microdata series. http://http://www.ipums.umn.edu/usa/index.html.

[2] 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.

[3] A. V. Aho, C. Beeri, and J. D. Ullman. The theory of joins in relational databases. *ACM Trans. Database Syst.*, 4(3), 1979.

[4] Stephen D. Bay and Michael J. Pazzani. Detecting group differences: Mining contrast sets. *Data Min. Knowl. Discov.*, 5(3):213–246, 2001.

[5] Christan Borgelt. Apriori implementation. http://fuzzy.cs.Uni-Magdeburg.de/ borgelt/Software. Version 4.08.

[6] Cristian Bucila, Johannes Gehrke, Daniel Kifer, and Walker White. Dualminer: a dual-pruning algorithm for itemsets with constraints. In *KDD Conference Proceedings*, pages 42–51, 2002.

[7] T. Calders and J. Wijsen. On monotone data mining languages. In *Proc. of International Workshop on Database Programming Languages (DBPL)*, pages 119–132, 2001.

[8] Surajit Chaudhuri. An overview of query optimization in relational systems. In *PODS Conference Proceedings*, 1998.

[9] Surajit Chaudhuri, Usama M. Fayyad, and Jeff Bernhardt. Scalable classification over sql databases. In *ICDE Conference Proceedings*, pages 470–479. IEEE Computer Society, 1999.

[10] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. McGraw Hill, 1990.

[11] Guozhu Dong and Jinyan Li. Efficient mining of emerging patterns: discovering trends and differences. In *KDD Conference Proceedings*, pages 43–52, 1999.

[12] Sašo Džeroski. Multi-relational data mining: an introduction. *SIGKDD Explor. Newsl.*, 5(1):1–16, 2003.

[13] 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)*.

[14] 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.

[15] 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.

[16] Jiawei Han, Laks V. S. Lakshmanan, and Raymond T. Ng. Constraint-based, multidimensional data mining. *Computer*, 32(8):46–50, 1999.

[17] 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.

[18] T. Imielinski and A. Virmani. Msql: a query language for database mining. In *Data Mining and Knowledge Discovery*, pages 3:393–408, 1999.

[19] Tomasz Imielinski and Heikki Mannila. A database perspective on knowledge discovery. *Commun. ACM*, 39(11):58–64, 1996.

[20] T. Johnson, Laks V. S. Lakshmanan, and Raymond T. Ng. The 3w model and algebra for unified data mining. In *VLDB Conference Proceedings*, 2002.

[21] Stefan Kramer, Luc De Raedt, and Christoph Helma. Molecular feature mining in hiv data. In *KDD Conference Proceedings*, pages 136–143, 2001.

[22] Laks V. S. Lakshmanan, Raymond Ng, Jiawei Han, and Alex Pang. Optimization of constrained frequent set queries with 2-variable constraints. In *SIGMOD Conference Proceedings*, pages 157–168, 1999.

[23] 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.

[24] 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.

[25] R. Meo, G. Psaila, and S. Ceri. A new sql-like operator for mining association rules. In *VLDB Conference Proceedings*, pages 122–133, Bombay, India, 1996.

[26] Rosa Meo, Marco Botta, and Roberto Esposito. Query rewriting in itemset mining. In *Proc. of the 6th International Conference On Flexible Query Answering Systems*, pages 111–124, June 2004.

[27] Raymond T. Ng, Laks V. S. Lakshmanan, Jiawei Han, and Alex Pang. Exploratory mining and pruning optimizations of constrained associations rules. In *SIGMOD Conference Proceedings*, pages 13–24, 1998.

[28] M. Otey, S. Parthasarathy, A. Ghoting, G. Li, S. Narravula, and D. Panda. Towards nic-based intrusion detection. In *KDD Conference Proceedings*, pages 723–728, 2003.

[29] P. Palmerini, S. Orlando, and R. Perego. Statistical properties of transactional databases. In *SAC '04: Proceedings of the 2004 ACM symposium on Applied computing*, 2004.

[30] S. Sarawagi, S. Thomas, and R. Agrawal. Integrating association rule mining with relational database systems: Alternatives and implications. In *SIGMOD Conference Proceedings*, 1998.

[31] Ramakrishnan Srikant, Quoc Vu, and Rakesh Agrawal. Mining association rules with item constraints. In David Heckerman, Heikki Mannila, Daryl Pregibon, and Ramasamy Uthurusamy, editors, *KDD Conference Proceedings*, pages 67–73, 1997.

[32] 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 *SIGMOD Conference Proceedings*, pages 1–12, 1998.

[33] J.D. Ullman and J. Widom. *A First Course in Database Systems*. Prentice Hall, Upper Saddle River, New Jersey, second edition, 2002.

[34] Geoffrey I. Webb, Shane Butler, and Douglas Newlands. On detecting differences between groups. In *KDD Conference Proceedings*, pages 256–265, 2003.

[35] X. Yin, J. Han, J. Yang, and P. S. Yu. Crossmine: Efficient classification across multiple database relations. In *ICDE Conference Proceedings*, Boston,MA, March 2004.

[36] 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.