Simultaneous Optimization of Complex Mining Tasks with a Knowledgeable Cache

Ruoming Jin
Department of Computer Science and Engineering
Ohio State University, Columbus OH 43210
jinr@cse.ohio-state.edu

Kaushik Sinha
Department of Computer Science and Engineering
Ohio State University, Columbus OH 43210
sinhak@cse.ohio-state.edu

Gagan Agrawal
Department of Computer Science and Engineering
Ohio State University, Columbus OH 43210
agrawal@cse.ohio-state.edu

ABSTRACT

With an increasing use of data mining tools and techniques, we envision that a Knowledge Discovery and Data Mining System (KDDMS) will have to support and optimize for the following scenarios: 1) *Sequence of Queries:* A user may analyze one or more datasets by issuing a sequence of related complex mining queries, and 2) *Multiple Simultaneous Queries:* Several users may be analyzing a set of datasets concurrently, and may issue related complex queries.

This paper presents a systematic mechanism to optimize for the above cases, targetting the class of mining queries involving frequent pattern mining on one or multiple datasets. We present a system architecture and propose new algorithms for this purpose. 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.

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.

1. INTRODUCTION

The iterative and exploratory nature of knowledge discovery or data mining has increasingly become a bottleneck in the KDD process, especially in view of the need for interactive response to the users. One of the relatively recent developments has been the focus on *constraint mining*. Constraint mining involves queries with constraint conditions, which help reduce the execution time of mining algorithms, and also help guide the KDD process. Many researchers have focused on designing languages or models to express these *complex* mining queries and developing new algorithms to evaluate each *single* complex query efficiently [19, 16].

Despite such developments, evaluation of mining queries is still relatively slow, and users often cannot perform data mining in an interactive manner. As the amount of data available for analysis in both scientific and commercial domains is increasing dramatically, efficiency in the data mining process is likely to become the crucial issue. With an increasing use of data mining tools and techniques, we envision that a Knowledge Discovery and Data Mining System (KDDMS) will have to support and optimize for the following scenarios:

- *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 [12], and similarly, techniques such as *reducing common subexpressions* [25, 24] 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 [18]. 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 [19]. 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 [13]. 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 paper, we focus on the problem of efficiently evaluating an important class of complex mining queries in a query intensive environment, where one needs to optimize multiple simultaneous queries, as well as a sequence of related queries. The class of complex mining queries we target are the ones involving frequent pattern mining on one or multiple datasets. 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 paper 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.

I	A	B	C	D
{1}	0.1	0.1	0.05	0.2
{2}	0.3	0.8	0.4	0.1
{3}	0.5	0.01	0.1	0.2
⋮	⋮	⋮	⋮	⋮
{1,2,3}	0.08	0	0.05	0.05
{1,2,4}	0.1	0.1	0	0.1
⋮	⋮	⋮	⋮	⋮

Table 1: F Table for Datasets A, B, C and D

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 paper is organized as follows. In Section 2, we will briefly introduce our target class of queries, which involve frequent pattern mining on multiple datasets. We describe three different representations of these queries, which are using SQL, an Algebra, and a new format, which we call the M-table. In Section 3, we present our framework to deal with both multiple simultaneous queries and a sequence of queries. In Section 4, 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 5, we present our optimization algorithms. In Section 6, we discuss the major implementation issues for our system, and present our experimental results. We compare our work with related research efforts in Section 7, and conclude in Section 8. Finally, in the Appendix, we discuss how our method can be generalized to frequent structure mining.

2. FREQUENT PATTERN MINING ON MULTIPLE DATASETS: SQL EXTENSIONS, ALGEBRA, AND M-TABLE

Frequent pattern mining focuses on discovering frequently appearing sub-structures in datasets. The structures explored include itemsets, sequences, sub-trees, sub-graphs, and other topological structures [29, 3, 33, 4]. Frequent pattern mining has emerged as a very useful class of techniques for analyzing datasets from a variety of domains, including retail transactions, DNA sequences, chemical compounds, XML documents, among others.

An important class of frequent pattern mining tasks involve discovering interesting patterns from *multiple* distinct datasets. For example, a manager of a nation-wide store will like to know what itemsets are frequent in stores in New York and New Jersey, but very infrequent in stores in California. Similarly, biology researchers are interested in sequences that are frequent in human gene but infrequent in chicken gene, and/or, the sequences are frequent in both the species.

In this section, we briefly describe the major issues in expressing as well as evaluating frequent pattern mining tasks on multiple datasets. Also, in order to simplify our discussion, we will focus on frequent itemset mining tasks. The key ideas in extending our work to other frequent patterns or structures, such as sequences, subtrees, and subgraphs, are presented in the Appendix.

2.1 SQL and M-Table for Mining Multiple Datasets

Assume we have four transaction datasets A, B, C, D, and *Item* is the set of all the items appearing in the four datasets. To extract the interesting patterns from these datasets, a *Frequency* table F is defined with a schema $Frequency(I, A, B, C, D)$. The column with attribute $F.I$ stores all possible itemsets, i.e, the power-set of $Item$. The columns with attribute $F.A$, $F.B$, $F.C$, $F.D$ store the frequency of the itemsets in the four datasets A, B, C, D, respectively. Table 1 contains a portion of this F table.

```
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 1: Query $\mathcal{Q}_1$

Note that the F table only serves as a virtual table or a logical view. A frequent pattern mining task on multiple datasets is expressed as a SQL query to partially materialize this table. The query $\mathcal{Q}_1$ in Figure 1(a) is an example. Here, we want to find the itemsets that are either frequent with support level 0.1 in both A and B, or frequent (with support level 0.1) in both C and D, and also frequent in either A or B (with support level 0.2).

Consider any query whose constraint condition (the WHERE clause) does not contain any *negative* condition, i.e., a condition which states that support in a certain dataset is below a specified threshold. Clearly, the constraint condition of such a query can be expressed in a tabular format, where 1) each row of the table represents a dataset, 2) each column corresponds to a conjunctive-clause (involving only the AND operation) in the disjunctive normal form (DNF) of the constraint condition, and 3) a cell at i-th row and j-th column will have the value α if the j-th conjunctive-clause requires that the support in the i-th dataset is at least α. The table thus computed is referred to as an M-table. For example, Figure 1(b) illustrates the M-table for the query $\mathcal{Q}_1$.

An M-table provides a systematic way to describe the information required to answer a query involving multiple datasets. It turns out that M-table can be used to 1) generate efficient query plans for a given query, 2) detect common computations across multiple queries, and 3) summarize the results obtained from multiple queries in a cache. Thus, our presentation in the rest of this paper will be based on M-table representation of the queries.

In the rest of this section, we focus on two issues. In the next subsection, we show how we can formally express the conditions captured by an M-table in terms of a basic mining operator, which is the frequent itemset operator on a single dataset. Finally, in Subsection 2.3, we show that an M-table can also be used to represent a class of queries involving *negative* conditions.

2.2 Algebra for Evaluating Frequent Mining Tasks

In this subsection, we introduce an algebra to express the information required to answer a mining query over multiple datasets. This algebra contains only one mining operator SF and two operations, intersection ($\sqcap$) and union ($\sqcup$). Formally, they are as follows:

The frequent itemset mining operator $SF(A_j, \alpha)$ 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 dataset A_j.

Intersection ($F_1 \sqcap F_2$): Let F_1 and F_2 be two tables whose first column contains a collection of itemsets, and other columns contain the corresponding frequency (possibly empty) in different datasets. The intersection operation ($F_1 \sqcap F_2$) 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.

Union ($F_1 \sqcup F_2$): The union operation ($F_1 \sqcup F_2$) 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.

Given an M-Table, the information required for answering the corresponding query can be described as follows. Each nonempty cell, $M_{i,j}$, maps to a SF operator, $SF(A_i, M_{i,j})$. The SF operators in the same column are connected by the intersection ($\sqcap$) operation, and the expressions corresponding to each column are connected by the union ($\sqcup$) operation. The resulting expression is referred as the *necessary information* for the query. For example, the necessary information of $\mathcal{Q}_1$ is shown in Figure 1(c).

2.3 Admissible Conditions and M-Table

In this subsection, we consider a broader class of queries, which could involve negative predicates as well. We establish that under certain restrictions, they can be represented through an M-table as well.

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. 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.

Through a set of transformations described in a related publication [14], we are able to achieve the following:

LEMMA 1. *The query constraints of an admissible query can be expressed as an M-table.*

Note that in such cases, the necessary information corresponding to the M-table may represent a superset of the results of the query. In such cases, a selection operation can be used to obtain the results of the query. As an example, Figure 2(a) shows the query $\mathcal{Q}_2$. The M-table representing its query constraints is illustrated in Figure 2(b), and the corresponding necessary information is in Figure 2(c). We can see that the first conjunctive-clause $F.A \geq 0.1$ AND $F.B \geq 0.1$ AND $F.D < 0.05$ has been mapped to the first two columns of the M-table. Particularly, in the necessary information format (F_2), we use $SF(A, 0.1) \sqcap SF(B, 0.1)$ to intersect with $SF(D, 0.05)$. Note that such repetition of $SF(A, 0.1) \sqcap SF(B, 01)$ is for minimizing the necessary information.

```
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.C < 0.05)
  OR (F.C ≥ 0.1 AND F.D ≥ 0.1 AND
  NOT(F.A ≥ 0.05 OR F.B ≥ 0.05))
```

(a) SQL query for query $\mathcal{Q}_2$

	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

(b) M Table for the query $\mathcal{Q}_2$

$$(SF(A, 0.1) \sqcap SF(B, 0.1)) \Rightarrow F_1$$
$$\sqcup(SF(A, 0.1) \sqcap SF(B, 0.1) \sqcap SF(C, 0.05)) \Rightarrow F_2$$
$$\sqcup(SF(C, 0.1) \sqcap SF(D, 0.1)) \Rightarrow F_3$$
$$\sqcup(SF(C, 0.1) \sqcap SF(D, 0.1) \sqcap SF(A, 0.05)) \Rightarrow F_4$$
$$\sqcup(SF(C, 0.1) \sqcap SF(D, 0.1) \sqcap SF(B, 0.05)) \Rightarrow F_5$$

(c) Necessary Information for query $\mathcal{Q}_2$

Figure 2: Query $\mathcal{Q}_2$

3. 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.

```
Q1: SELECT F.I, F.A, F.B, F.X
    FROM Frequency(I, A, B, X) F
    WHERE F.A ≥ 0.2 AND F.B ≥ 0.1 AND F.X < 0.1

Q2: SELECT F.I, F.A, F.B, F.Y, F.Z
    FROM Frequency(I, A, B, Y, Z) F
    WHERE (F.A ≥ 0.1 AND F.B ≥ 0.1
      AND F.Y ≥ 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 query $\mathcal{Q}$ (described in the previous section), before it receives the queries $\mathcal{Q}_1$ and $\mathcal{Q}_2$. 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. 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,

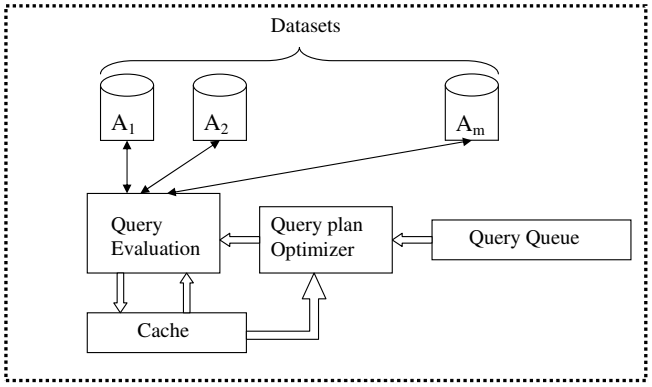

Figure 3: System Framework

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 paper 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 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 [18], 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 4, we focus on the properties of the M-table which enable the above optimizations. In Section 5, we discuss the detailed optimizations and cache management.

4. PROPERTIES OF M-TABLE FOR QUERY OPTIMIZATION

	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 2: M-Tables with Containment Relationships

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.

4.1 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 1. *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 $M_1[i] \geq M_2[i]$.*

If M_1 is contained in M_2, we denote this as $M_1 \subseteq M_2$. For example, in Table 2, 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 2. *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 2. *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 3. *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 5, 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 4. *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 2, 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 3. *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.

4.2 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 5. *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 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.

	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: Merge Operation for M-Tables

5. MULTIPLE QUERY OPTIMIZATION APPROACH

In this section, we present our optimization algorithms which are based on M-tables. Specifically, in Subsection 5.1, we first review how the query plan for a single query is generated from an M-table. In Subsection 5.2, we study how each mining operator can be mapped to the M-table and how the redundant mining operators can be detected. In Subsection 5.3, we discuss how local plans from several queries can be optimized together. In Subsection 5.4, we introduce another approach for optimizing multiple queries, which involves merging multiple queries into one query, and then generating a global query plan. Subsection 5.5 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.

5.1 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. 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 [14]. For simplicity, we will only use CF and SF in this paper. 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 4 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

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

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 [14]. 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 4. 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 4: Algorithm-CF for Query Plan Generation

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

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

	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 5: M-Tables of different mining operators

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

5.2 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 $\mathcal{Q}$.

$$\begin{aligned} O_1 &: SF(A, 0.1); \\ O_2 &: SF(C, 0.1); \\ O_3 &: CF(B, 0.05, (SF(A, 0.1) \sqcup SF(C, 0.1))^I); \\ O_4 &: CF(D, 0.1, CF^I(B, 0.1, SF^I(C, 0.1)); \end{aligned}$$

Table 5 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 2 3 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.

5.3 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 4.1, 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 computa-

tions. 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.

```
Input: local query plans Q_1, ··· , Q_n
S = {Q_1, ··· Q_n};
While (S ≠ ∅) Do
  Foreach Q_i ∈ 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 5: Greedy Algorithm to Remove Containment in Multiple Query Plans

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 5. 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 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.

5.4 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 subcondition 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.

$$\begin{aligned}
& SF(B, 0.15), SF(Z, 0.2); \\
& CF(A, 0.1, SF(B, 0.15)^I); \\
& CF(Y, 0.01, ((SF(A, 0.1) \sqcap SF(B, 0.15)) \sqcup SF(Z, 0.2))^I); \\
& CF(Z, 0.01, ((SF(A, 0.1) \sqcap SF(B, 0.15) \sqcap SF(C, 0.1))^I);
\end{aligned}$$

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 6.

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

$$\begin{aligned}
& SF(A, 0.1), SF(Z, 0.2); \\
& CF(A, 0.1, SF(B, 0.1)^I); \\
& CF(X, 0.1, ((SF(A, 0.2) \sqcap SF(B, 0.1))^I)); \\
& CF(Y, 0.01, ((SF(A, 0.1) \sqcap SF(B, 0.15)) \sqcup SF(Z, 0.2))^I); \\
& CF(Z, 0.01, ((SF(A, 0.1) \sqcap SF(B, 0.15) \sqcap SF(C, 0.1))^I);
\end{aligned}$$

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.

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

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 7: M Table for the Cache

5.5 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.

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 7.

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

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 7. The pre-coloring of the merged M-table for queries $\mathcal{Q}_1$ and $\mathcal{Q}'_2$ is shown as Table 8.

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 5.3), 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 5.4), 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:

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

6. 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. 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 ?

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

6.1 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.

6.2 Datasets

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

	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 9: Test Query Templates for Our Experiments

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.

IBM's Quest: The second 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. Each of four datasets contains 1,000,000 transactions.

6.3 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 9 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%.

6.4 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 5.3. 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 5.4. 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.

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 10: Group-I 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	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 11: Group-I Results on Real (IPUMS) Datasets (All Execution Times in Seconds)

6.5 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 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 10 and 11 show the *Group-1* experimental results. Tables 12 and 13 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 SQ, the average speedups of LQ for all batch size in Tables 10, 11, 12, and 13 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 10, 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.

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 12: 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 13: Group-II Results from Real (IPUMS) Datasets (All Execution Times in Seconds)

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 14: Caching Effects: IPUMS(in Seconds)

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 10, 11, 12, and 13, 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 10, 11, 12, and 13, 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 14 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 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.

7. 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 [8, 16, 19, 21, 27]. More recently, Yan *et. al* have studied the use of connectivity constraints to mine frequent graphs [31]. 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 [15, 17, 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.

Our research is also different from the work on *Query flocks* [28]. 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 [7, 10, 22, 32] 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 [6, 9, 30]. 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 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 [18]. Hipp and Guntzer have proposed to use pre-computation of frequent itemsets of certain support levels to answer constraint itemset mining queries [13]. Goethals and Bussche have developed methods to support an interactive data mining session [11]. 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 [25, 20, 24, 26]. 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 [34]. 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 [5]. However, their methods are mainly based on the spatial properties of the queries and cannot applied to the mining tasks we have focused on.

8. CONCLUSIONS

The work presented in this paper 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 paper, 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.

9. 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] 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.

[4] Rakesh Agrawal and Ramakrishnan Srikant. Mining sequential patterns. In *Proceedings of the Eleventh International Conference on Data Engineering*, 1995.

[5] 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.

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

[7] Hendrik Blockeel and Micheˇle Sebag. Scalability and efficiency in multi-relational data mining. *SIGKDD Explor. Newsl.*, 5(1):17–30, 2003.

[8] 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.

[9] 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.

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

[11] 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.

[12] Alon Y. Halevy. Answering queries using views: A survey. *The VLDB Journal*, 10(4), 2001.

[13] Jochen Hipp and Ulrich Guntzer. 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.

[14] 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.

[15] 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.

[16] 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.

[17] 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.

[18] Biswadeep Nag, Prasad M. Deshpande, and David J. DeWitt. Using a knowledge cache for interactive discovery of association rules. In *Proceedings of the fifth ACM SIGKDD international conference on Knowledge discovery and data mining*, 1999.

[19] 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.

[20] Jooseok Park and Arie Segev. Using common subexpressions to optimize multiple queries. In *Proceedings of the Fourth International Conference on Data Engineering*, 1988.

[21] 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.

[22] 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.

[23] 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.

[24] Prasan Roy, S. Seshadri, S. Sudarshan, and Siddhesh Bhobe. Efficient and extensible algorithms for multi query optimization. *SIGMOD Rec.*, 29(2), 2000.

[25] Timos K. Sellis. Multiple-query optimization. *ACM Trans. Database Syst.*, 13(1), 1988.

[26] Kyuseok Shim, Timos Sellis, and Dana Nau. Improvements on a heuristic algorithm for multiple-query optimization. *Data Knowl. Eng.*, 12(2), 1994.

[27] 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.

[28] 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.

[29] Takashi Washio and Hiroshi Motoda. State of the art of graph-based data mining. *SIGKDD Explor. Newsl.*, 5(1):59–68, 2003.

[30] 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.

[31] Xifeng Yan, X. Jasmine Zhou, and Jiawei Han. Mining closed relational graphs with connectivity constraints. In *ICDE*, 2005.

[32] 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.

[33] 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.

[34] Yihong Zhao, Prasad M. Deshpande, Jeffrey F. Naughton, and Amit Shukla. Simultaneous optimization and evaluation of multiple dimensional queries. *SIGMOD Rec.*, 27(2), 1998.

APPENDIX

A. 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 [29, 3, 33, 4]. 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 ?

A.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.Ai1, F.Ai2, ···, F.Ais
  FROM Frequency(T, A1, ···, Am) 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.

A.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 paper 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.

A.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 straight-forward. 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.