1-1-2007

Complete and equivalent query rewriting using views.

Minghao Li
University of Windsor

Recommended Citation

Li, Minghao, "Complete and equivalent query rewriting using views." (2007). *Electronic Theses and Dissertations*. 6993.
https://scholar.uwindsor.ca/etd/6993

COMPLETE AND EQUIVALENT QUERY REWRITING USING VIEWS

by

Minghao Li

A Thesis
submitted to the Faculty of Graduate Studies and Research
through Computer Science
in Partial Fulfillment of the Requirements for
the degree of Master of Science at the
University of Windsor
Windsor, Ontario, Canada
2007

Library and
Archives Canada

Published Heritage
Branch

395 Wellington Street
Ottawa ON K1A 0N4
Canada

Bibliothèque et
Archives Canada

Direction du
Patrimoine de l'édition

395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence
ISBN: 978-0-494-35018-8
Our file Notre référence
ISBN: 978-0-494-35018-8

Abstract

Query rewriting using views is a technique for answering a query using a set of views instead of accessing the database relations directly. There are two categories of rewritings, i.e., equivalent rewriting using materialized views applied in query optimization, and maximally contained rewriting used in data integration. Although maximally contained rewriting is acceptable in data integration, there are cases where an equivalent rewriting is desired. More importantly, the maximally contained rewriting is a union of all the contained queries, many of which are redundant. This thesis gives an efficient algorithm to find a complete and equivalent rewriting that is a single conjunctive query. We proved that the algorithm is guaranteed to find all the complete and equivalent rewritings, and that the produced rewriting is guarantee to be equivalent without additional containment checking. We showed that our algorithm is much faster than other algorithms by complexity analysis and experiments.

Keywords: query rewriting, data integration, containment mapping.

Acknowledgements

I would first like to thank my supervisor, Dr. Jianguo Lu, for his constant encouragement and guidance. He has walked me through all the stages of my research work and thesis writing. Without his support, the work presented in this thesis would not have become a reality.

I would extend my appreciation to Calisto Zuzarte, Xiaoyan Qian and Wenbin Ma from IBM Toronto Lab, for their enlightening discussion and help when I studied DB2 Query Rewriting in IBM CAS Toronto. I would also like to thank Mr. Kenneth Cheung for his excellent work on a preliminary version of query translating and rewriting system.

Most of all, I would like to express my deepest gratitude to my dear fiancée Ms. Juan Liu. It is her unconditional love and support that helps me overcome all the difficulties and finish this work eventually. My thanks would also go to my parents for their loving considerations and great confidence in me all through these years.

Many others, too numerous to mention, provided encouragement and support as I carried out my research. I would like to take this opportunity to thank them all.

Table of Contents

List of Tables

List of Figures

Chapter 1
Introduction

1.1 Application Background

Query rewriting using views is also known as answering query using views [1] or query folding [2]. Informally speaking, the problem can be described as following: given a query on a database schema, and a set of views over the same schema, how can the query be rewritten so that it partially or completely refers to the set of views? Query rewriting plays a very important role in many database management applications, such as query optimization [3, 4], data integration [5, 6, 7, 8], and data warehouse [9, 10, 11].

In query optimization, some views are created to perform the frequently executed query operations, and the results of these views are stored in disk for future uses. These views are called materialized views. Given a query posed over database relations, if it can be answered using materialized views, then the evaluation process of this query will become more efficient, because some computations are pre-performed by the materialized views. Therefore, from the query optimization point of view, query rewriting is to generate logical plan that is equivalent to the original query using materialized views.

In the context of data integration, query rewriting using views is also unavoidably involved into the process of answering queries. Some well-known data integration systems, such as Information Manifold [5] and Infomaster [12], provide a uniform query interface to a number of heterogeneous data sources. In order to do so, the data integration system first defines a global schema according to a particular application, and then describes all data sources as views over the global schema. Users pose queries over the global schema as well. Global schema, however, is a set of virtual relations, which means there is no actual data stored in them. In order to answer those queries, the system has to find rewritings of the queries that only refer to data source descriptions (views). The rewriting result might be a union of several queries that are contained in the original query.

This thesis focuses on the discussion of query rewriting algorithms for data integration applications. Thus in the next section, we will talk about data integration architecture in more detail.

1.2 Data Integration

1.2.1 Introduction

Along with the booming of World Wide Web, there is tons of information on the Internet. Apart from the Web, there is also a large number of public databases and many other types of information sources that can be accessed through the Internet. However, how can we make efficient use of all those information? Data integration system is one of the solutions to this problem.

Data integration system provides uniform access to various distributed heterogeneous information sources. The information sources can include traditional database systems, structured data like XML files, or even semi-structured data like information in HTML pages. Data integration system frees the users from finding information from many different sources and then manually combining them together. Taking advantages of data integration architecture, users are able to pose queries against a uniform schema, without knowing the type of data sources that is used behind.

In previous researches, several information integration systems have been proposed.

- Information manifold, a project of AT&T Lab [5].
- TSIMMIS, a cooperative project between Stanford University and IBM [6].
- Infomaster, a practical integration system from Stanford University [12].
- SIMS, a service and information management system for decision support [13].

1.2.2 Data Source Modeling

Data integration system is also known as mediator system. The two approaches to define mediator and information descriptor are as follows.

1. **Global-As-View (GAV)** approach defines the mediator schema as views in terms of the schema of information sources. The mediator schema is considered as a set of query patterns. A query is defined in terms of mediator schema. A query can be answered by the mediator if it matches one of those patterns, otherwise it cannot be handled. When a data source is added into or removed from the mediator, the mediator schema has to be reformulated. Some known data integration systems are based on GAV approach, for example, TSIMMIS, a cooperative project between Stanford University and IBM [6]. TSIMMIS uses GAV approach although it adopts a different rule language.

2. **Local-As-View (LAV)** approach is the opposite of GAV. The mediator schema in LAV is a global virtual schema designed according to the need of specific application. It is called virtual schema because there is no data actually stored in the mediator schema. Both user queries and source descriptions are defined in terms of the global schema.. In order to answer a query, the system first rewrites the query using source description views, and then retrieves the best result from the rewritings.

   Since the virtual mediator schema contains no data, the rewritten query must only refer to data descriptions (views). Because some data sources might be incomplete, the system allows a rewritten query not equivalent to the original query, but to be able to retrieve the maximal answer set based on all available data sources.

Adding or deleting an information source is just creating or removing a source descriptor. Therefore, the data integration system based on LAV approach has excellent extendibility and is suitable for Internet based applications. Information Manifold system from AT&T Lab [5] and Infomaster system from Stanford University [12] are two famous data integration systems that use LAV approach.

In these two systems, query rewriting plays an essential role. In the following sections, we will introduce some existing query rewriting algorithms used in data integration applications.

1.3 Query Rewriting Using Views

Query rewriting using views is a technique that offers a way to answer a query referring to pre-defined views instead of database relations. According to different criteria, query rewritings can be categorized into:

1. Complete Rewriting vs. Partial Rewriting
   - Complete Rewriting is a rewriting of query that only refers to views.
   - Partial Rewriting is a rewriting of query that refers to both views and database relations.

2. Equivalent Rewriting vs. Maximally Contained Rewriting

- Equivalent Rewriting is a rewriting having the same answer set as the original query.
- Contained rewriting is a rewriting whose answer set is contained by the answer set of the original query.
- Maximally contained rewriting is a contained rewriting whose answer set contains the answer set of any other contained rewriting of the query.

Different applications require different type of query rewritings. Query optimization application looks for an equivalent rewriting that has the shortest evaluation time, while data integration application needs a rewriting that only refers to views and has the same answering set as the original query. Therefore, complete and equivalent rewriting, also known as CE rewriting is the best solution for data integration application. However, some queries might not have equivalent rewriting based on all available views, and then complete maximally contained rewriting can work as an alternative solution.

Many algorithms have been proposed to solve the query rewriting problem for data integration applications. These algorithms generally fall into two classes: bucket-based algorithm and inverse rule-based algorithm. The primary bucket algorithm was first proposed by A.Y. Levy in [5]. More recently, some other bucket-based algorithms, such as MiniCon algorithm [14], Shared Variable Bucket (SVB) algorithm [15] and

CoreCover algorithm [16] were developed to improve the performance of primary bucket algorithm.

The idea of inverse rule was first proposed by X. Qian in [2], and presently the inverse rule algorithm was formally discussed in [17]. We are going to review these algorithms in section 2.3.

1.4 Thesis motivation and contribution

Existing complete query rewriting algorithms [5] [14] [15] [16] are designed to produce complete maximally contained rewritings. The result of those algorithms is a union of all complete contained rewritings that can be found. In application where there is a large number of views available for query rewriting, there might exist a single conjunctive query that is equivalent to the original query. This equivalent rewriting itself is sufficient to answer the query. However, existing complete query algorithms include all the other contained conjunctive queries into the result, which in fact are redundant and make the evaluation of the rewriting inefficient.

In order to find the complete and equivalent conjunctive rewriting, we propose a TCM (Tail Containment Mapping) Bucket algorithm that expands the bucket algorithm to generate complete and equivalent rewriting. TCM bucket algorithm prevents inappropriate views from being added to the buckets generated in the first stage of the bucket algorithm, so that the rewritings produced in the combination stage are automatically CE rewritings without any extra containment checking.

In the experiments of this thesis, we implement MiniCon algorithm, an advanced bucket based algorithm, with our TCM checking rules and compare this combination with some other algorithms. All testing data we use in our experiments, containing hundreds of relations, views and queries, are extracted from real e-commerce projects, rather than generated by ourselves.

1.5 Thesis overview

The main body of this thesis is organized as follows. Chapter 2 introduces the relevant notations, gives the formal definition of the query rewriting problem and some important concepts, and briefly goes over two previous query rewriting algorithms in the last section. In chapter 3, we first discuss the motivation and benefit of answering query using complete equivalent rewriting, and then we present our TCM bucket algorithm. Chapter 4 provides the results and analyses of the experiments that we conduct. Chapter 5 provides a summary of this thesis and discusses possible future work.

Chapter 2
Overview of Query Rewriting Using Views

In this chapter, we will first introduce some concepts that are related to query rewriting using views and are used throughout this thesis. Secondly, we will describe some different kinds of query rewritings surveyed in [1]. And at last, we will discuss some related works that have been already done under this topic.

2.1 Query Language

All the queries and views discussed in this thesis are SPJ queries, whose relational algebra expressions only contain project, select and join operations. We also assume that no queries contain redundant subgoal, and that neither query nor view contains comparison predicates.

We use Datalog to define all the queries and views in this paper. Datalog [18] is a powerful query language that is widely used to express queries and views in [16, 15, 14, 19]. Using Datalog expression, a SPJ query Q can be conveyed to a rule in the following form:

$$Q(\bar{X}) : -r_1(\bar{X}_1), \cdots, r_n(\bar{X}_n)$$

where Q and $r_1, \cdots, r_n$ are predicate names, and $\bar{X}$, $\bar{X}_1, \cdots, \bar{X}_n$ are tuples of variables or constants. Q($\bar{X}$) is the head of query, denoted by H(Q). Variables in $\bar{X}$ are called distinguished variables or head variables. The variables in $\bar{X}_1, \cdots, \bar{X}_n$ but not in $\bar{X}$ are called existential variables. To be safe, distinguished variables must also appear in subgoals of relations or views. $r_1(\bar{X}_1), \cdots, r_n(\bar{X}_n)$ are called ***source subgoals***, which refer to database relations or views. And we use Subgoal(Q) to denote all source relations of a query Q.

In a datalog clause, we always assume that the ith variable of a subgoal r(X) refers to the ith attribute of the relation table that r(X) refers to. And the fact that the same variable appears in two source subgoals implicates that there is a join predicate on this variable between the two relations or views referred by these two subgoals. Such variables are called join variables. A Datalog clause may also include subgoals of arithmetic comparison predicates, such as <, =, >. These subgoals are called ***comparison subgoals***. Variables in a comparison subgoal must also appear in subgoals of relations or views.

Example 2.1 *Here we give an example of Datalog notation based on a university schema as in Figure 2-1[1]. We will use this schema throughout this thesis. Based on this schema, suppose there is a query that asks for the name of a student and the course this student registered. The SQL expression for this query is as follows:*

SELECT Student.sname, Register.cnum

```
FROM      Student, Register
WHERE     Student.snum=Register.snum.
```

Prof(pname,area)	**Advise**(pname,snum)
Course(cnum,ctitle)	**Teach**(pname,cnum,term)
Student(snum,sname,major)	**Register**(snum,cnum,term)

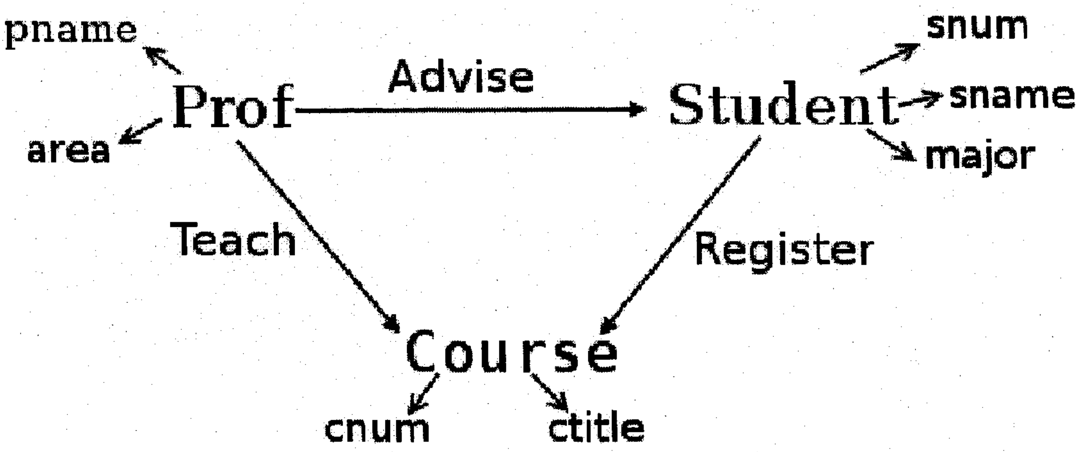


Figure 2-1: University schema and entity/relation diagram

The Datalog expression for this query is:

*Q(sname,cnum):–Student(**snum**,sname,major), Register(**snum**,cnum,term).*

Variable snum appears in both subgoals implies that there is a join condition between student and register on variable snum.

2.2 Query Rewriting

2.2.1 Query Containment and Equivalence

The problem of conjunctive query containment was first studied in [20]. Query containment is used to decide the containing relationship between two queries, and is important to check the correctness of a query rewriting. The formal definition of query containment and equivalence is as below [1].

Definition 2.1 (Query containment and equivalence)

A query Q_1 is said to be contained in another query Q_2, denoted as $Q_1 \subseteq Q_2$, if for any database D, the set of tuples computed for Q_1 is a subset of those computed for Q_2, i.e., $Q_1(D) \subseteq Q_2(D)$. The two queries are said to be equivalent, denoted as $Q_1 \equiv Q_2$, if $Q_1 \subseteq Q_2$ and $Q_2 \subseteq Q_1$.

2.2.2 Containment Mapping

Generally speaking, whether a query Q_1 is contained in query Q_2 is undecidable. For conjunctive queries that are considered in this thesis, the containment relationship is decidable [21]. Containment mapping [20] provides a method to decide containment relationship between conjunctive queries.

A containment mapping reflects a homomorphism between the variables of two queries.

It maps a variable in ith position of a source subgoal in query Q_1 to a variable in ith position of a subgoal referring to the same relation in query Q_2. A formal definition of containment mapping is given below [18]:

Definition 2.2 (Containment Mapping)

Given two queries Q_1 and Q_2 over database schema S, and a mapping φ from variables of Q_2 to variables of Q_1. Mapping φ is a containment mapping from Q2 to Q1, if

1. *The head of Q_2 maps to the head of Q_1; and*
2. *Every subgoal of Q_2 is mapped to a subgoal of Q_1.*

Intuitively, a containment mapping from Q_2 to Q_1 guarantees that all join relationships in query Q_2 are retained in query Q_1. In addition, query Q_1 may have more subgoals than query Q_2. Thus, comparing to Q_2, query Q_1 has same or even stricter conditions to select tuples. There is a well-known and important conclusion based on containment mapping as below [18].

Theorem 2.1 Given conjunctive queries Q_1 and Q_2, $Q_1 \subseteq Q_2$ if and only if there is a containment mapping from Q_2 to Q_1.

Example 2.2 *Suppose we have three queries as below:*

Q_1(pna1,cnu1,snu1):–Teach(pna1,cnu1,ter1), Register(snu1,cnu1,ter1')

Student(snu1,sna1,maj1)

Q_2(pna2,cnu2,sna2):–Teach(pna2,cnu2,ter2), Register(snu2,cnu2,ter2')

Q_3(pna3,cnu3,sna3):–Teach(pna3,cnu3,ter3), Register(snu3,cnu3,ter3)

We can form a mapping φ_{21} from Q_2 to Q_1 as following:

φ_{21}: {pna2→pna1, cnu2→cnu1, ter2→ter1, snu2→snu1 ter2'→ter1'}

φ_{21} satisfies the conditions of containment mapping, so Q1 is contained by Q2. For query Q_3, the only possible mapping φ_{31} we can construct from Q_3 to Q_1 is shown below.

φ_{31}: {pna3→pna1, cnu3→cnu1, ter3→ter1, snu3→snu1 ter3→ter1'}

However, it is not a containment mapping because variable ter3 of Q_3 is mapped to two different variables ter1 and ter1' of Q_1. As a result, Q_1 is not contained by Q_3.

Theorem 2.1 provides a basic way to check whether a conjunctive query Q_1 contains another query Q_2. Finding containment mapping from Q_1 to Q_2 is an NP-complete problem [20]. In the worst case, its time complexity is $O(m^n)$, where n is the number of subgoals in Q_1 and m is the number of subgoals in Q_2. In practice, the size of queries or views is usually not large; hence, the execution time for containment checking is acceptable. More important, the performance of an algorithm could be significantly improved if the use of containment checking is avoided or reduced.

2.2.3 Query Rewriting Using Views

The problem of query rewriting using views, a.k.a. answering query using views, is about finding a method to retrieve the resulting tuples of a query from some predefined views instead of accessing the database relations. The formal definitions of two kinds of query rewritings are shown below.

Definition 2.3 (Equivalent Rewriting) [1]

Given a query Q referring to database relations of a schema, and a set of views V = $V_1, \cdots, V_m$ *over the same schema, the query Q' is an equivalent rewriting of Q using V if:*

- *Q' refers to one or more views in V; and*
- *Q' is equivalent to Q.*

Definition 2.4 (Maximally-Contained Rewriting) [1]

Given a query Q referring to database relations of a schema, and a set of views V = $V_1, \cdots, V_m$ *over the same schema, Q' is a rewriting of Q using views from V, then*

1) *Q' is a* ***contained rewriting*** *of Q, if* $Q' \subseteq Q$.
2) *Q' is a* ***maximally-contained rewriting*** *of Q if:*
   - $Q' \subseteq Q$*; and*
   - *There is no another rewriting Q" of Q using the same query language, such that* $Q' \subset Q'' \subseteq Q$*, i.e., for any database D,* $Q'(D) \subset Q''(D) \subseteq Q(D)$.

Definition 2.5 (Complete and Equivalent Rewriting) [1]

Given a query Q referring to database relations of a schema, and a set of views V = $V_1, \cdots, V_m$ *over the same schema, Q' is a rewriting of Q using views from V, then*

1) *Q' is a* ***complete rewriting*** *of Q if Q' only refers to views from V.*

2) *Q' is a* ***complete and equivalent rewriting*** *of Q if:*

- *Q' is complete rewriting; and*
- *Q' ≡ Q.*

In this thesis, we abbreviate the complete and equivalent rewiring to CE rewriting.

Here are some examples of different types of query rewritings.

Example 2.3 *Suppose that we have a query over the schema defined above which asks for student names who are taught by professors in "Database" area in winter term.*

Q(sname):–Register(sname,cnum,term), Teach(pname,cnum, term, year),

Prof(pname,area), area="Database", term="win".

Besides the database relations, we have two views V_1 and V_2. V_1 shows the professors in Database area and the courses they teach. V_2 shows the registration information in winter term.

V_1(pname,cnum):–Prof(pname,area), Teach(pname,cnum,term),

area="Database".

V2(sname,cnum):- Register(sname,cnum,term), term="win".

There are three rewritings of query Q using V_1 and V_2 shown below. We can see both Q'_1 and Q'_2 are partial rewritings, while Q'_3 is a complete rewriting of Q.

Q'_1(sname):–V_1(pname,cnum), Register(sname,cnum,term), term="win".

Q'_2(sname):–V_2(sname,cnum), Teach(pname,cnum,term),

Prof(pname,area), area="Database".

$Q'_3(sname)$:–$V_1(pname,cnum)$, $V_2(sname,cnum)$.

Query rewriting is used in various database applications. Different applications require different kinds of query rewritings. For the purpose of query optimization, equivalent rewriting is required, but both complete and partial rewritings are acceptable. While in data integration applications, the rewriting must be complete rewriting, because data relations in this case are virtual relations and actual data sources are described as views. When equivalent rewriting cannot be approached, maximally-contained rewriting is accepted then. In this thesis, we only discuss the problem of finding complete rewriting.

2.2.4 Query Expansion

For a query Q over a database schema referring to not only relations but also views, the expansion of Q is a query Q_X constructed by substituting the views in the body of Q with their definitions and renaming the variables to maintain the equivalence with the original query Q.

Suppose the original query is as below:

$Q(X)$:– $r_1(X_1), \ldots, r_n(X_n)$, $V(Y)$.

and we have the definition of view V as follows:

$V(Z) :- p_1(Z_1), \ldots, p_m(Z_m).$

then the expansion of Q is of the form:

$Q_X(X) :- r_1(X_1), \ldots, r_n(X_n), p_1(V_1), \ldots, p_m(V_m).$

Q_X is created in two steps:

1. Replace the subgoal V(Y) in Q with the body of the definition of view V.
2. Rename all variables from V definition, $Z_1, \ldots, Z_m$, using the following rules:
   (a) If variable z_i is a distinguished variable of V, i.e., $z_i \in Z$, rename z_i as a corresponding variable in Y from the original query.
   (b) If variable z_i is an existential variable of V, i.e., $z_i \notin Z$, rename z_i as a new variable of Q which is not in $X_1 \cup \ldots \cup X_n$.

Example 2.4 *Consider the following query and views:*

Q(pname,cnum,sname):–V₁(pname,area,snum), V₂(pname,area,cnum),

Student(snum,sname,majo).

V₁(pna1,are1,snu1) :– Advise(pna1,snu1), Prof(pna1,are1.)

V₂(pna2,are2,cnu2) :– Prof(pna2,are2), Teach(pna2,cnu2,ter2).

The expansion of Q is as below:

Q_X(pname,cnum,sname):–Advise(pname,snum), Prof(pname,area),

Prof(pname,area), Teach(pname,cnum,ter2)

Student(snum,sname,major).

2.2.5 Rewriting Containment Verification

Given a query Q on a database schema S, and a rewriting Q' of Q referring to some views over the same schema, it is not applicable to compare Q and Q' directly, but taking advantage of Q'_X, it can be decided whether Q' is contained or equivalent to Q.

The rewriting Q' referring to V is computed using view instance I. While in Q'_X, all subgoals referring to views are substituted by the definitions of these views, so that Q'_{exp} refers to database relations. Based on our assumption that the instance of any view contains all tuples that satisfy its definition, we can infer that Q' and Q'_X yield the same set of resulting tuples, i.e., $Q' \equiv Q'_X$.

For a rewriting Q' of query Q, both Q'_X and Q refer to schema S, therefore according to Theorem 2.1, it can be decided whether Q'_X is contained or equivalent to Q by finding containment mapping between Q and Q'_X. Furthermore, it can be inferred whether Q' is contained or equivalent to Q using corollary 2.1.

Corollary 2.1 *Given a database schema S, a set of views V is defined on S. Suppose Q is a conjunctive query on S, and Q' is a rewriting of Q referring to V.*

1) $Q' \subseteq Q$ *iff* $Q'_X \subseteq Q$.
2) $Q' \equiv Q$ *iff* $Q'_X \equiv Q$

2.3 Previous Algorithms for Query Rewriting

Two types of rewriting algorithms have been proposed for data integration applications in previous researches: bucket based algorithm and inverse rule based algorithm. The primary bucket algorithm was first proposed by paper [5] for Information Manifold system. In the later research, several improved version of bucket algorithm were developed, such as MiniCon algorithm [14], shared variable bucket algorithm (SVB) [15] and CoreCover algorithm [16]. The basic idea about inverse rule was proposed by paper [2], and was then developed into the inverse rule algorithm in [17]. Other inverse rule based algorithms were proposed in [22], [19]. Both bucket based algorithms and inverse rule based algorithms aim to find maximally contained rewriting, so that output of these algorithms are a union of all contained rewritings that can be found.

Before continuing the thesis, it may be necessary to introduce the bucket algorithm in more detail.

Bucket Algorithm

The input of bucket algorithm is a conjunctive query Q referring to database relations and a set of views V: $V_1, \cdots, V_n$. Both the query Q and view V_i are non-recursive SPJ (project-select-join) queries. The output of bucket algorithm is a union of conjunctive rewritings that are contained in the original query Q.

With the input Q and V, bucket algorithm proceeds in following two steps.

1. bucket constructing

For each source subgoal r $\in$ subgoal(Q), create a bucket, and put the views $V_i \in V$, which can satisfy the following condition, into the bucket of subgoal r.

- View V_i has a subgoal r_{Vi}, representing the same relation to r, and there exists a partial containment mapping ψ from r to r_{Vi} .

When a view is added to a bucket, the view variables in the domain of ψ are renamed as the corresponding variables of Q, while other variables are renamed as new variables (primed).

2. making combination and containment checking

First, make a Cartesian product within all buckets, and for each tuple of the result of Cartesian product, create a conjunct query Q' by joining all elements in the tuple. Then proceed a containment checking to decide if Q' $\subseteq$ Q. If so, add Q' to the output union.

Example 2.5 *Suppose we have the query and views as following.*

Q(pname,sname):–Register(sname,cnum,term), Teach(pname,cnum,term),

Prof(pname,area).

V_1(sna1,cnu1,ter1):–Register(sna1,cnu1,ter1), Course(cnu1,cti1).

V_2(pna2,cnu2,ter2):–Prof(pna2,are2), Teach(pna2,cnu2,ter2).

V_3(pna3,sna3):–Teach(pna3,cnu3,ter3), Register(sna3,cnu3,ter3).

After the first step, a set of buckets (in Table 2-1) is created.

Register(sname, cnum, term)	Teache(pname, cnum, term)	Prof(pname, area)
V_1(sname, cnum, term)	V_2(pname, cnum,term)	V_2(pname, cnu2, ter2)
V_3(pna3, sname)	V_3(pname, sna3)	

Table 2-1: Bucket table for bucket algorithm

In the second step, a Cartesian product is performed within all buckets, and the combinations of views shown in Table 2-2 are retrieved. For each tuple in the combination table, a conjunctive query Q' is constructed by joining all elements of the tuple and containment checking is imposed to test if $Q' \subseteq Q$. For our example, two contained rewritings are found at the end as below:

V_1(sname, cnum, term)	V_2(pname, cnum,term)	V_2(pname, cnu2, ter2)
V_1(sname, cnum, term)	V_3(pname, sna3)	V_2(pname, cnu2, ter2)
V_3(pna3, sname)	V_2(pname, cnum,term)	V_2(pname, cnu2, ter2)
V_3(pna3, sname)	V_3(pname, sna3)	V_2(pname, cnu2, ter2)

Table 2-2: Table of view combinations

Q'_1(pname,sname):– V_1(sname,cnum,term), V_2(pname,cnum,term).

Q'_2(pname,sname):– V_3(pname,sname), V_2(pname,cnu2,ter2).

The final output of bucket algorithm is a union of Q'_1 and Q'_2, i.e., $Q'_1 \cup Q'_2$.

Chapter 3
Finding Complete and Equivalent Rewriting

In the last chapter, we have introduced some algorithms of answering query using views. The objective of those algorithms is to find maximally contained rewriting. In this chapter, we will present our TCM (Tail Containment Mapping) Bucket algorithm, which is designed to find equivalent and complete rewritings for a conjunctive query. We will first explain the motivation of our research in section 3.1. Then, we will discuss two solutions for finding complete and equivalent (CE) rewritings. In section 3.3, we will propose our TCM Bucket algorithm, and the analysis of its time complexity will be given in the last section.

3.1 Motivation

In the context of data integration, most of query rewriting algorithms [5][14] [15] [16] are designed to produce maximally contained rewriting. The output of those algorithms is a union of all contained rewritings they can find. We denote such maximally contained rewriting as Q'_M, $Q'_M = Q'_1 \cup, \ldots, \cup Q'_n$, where Q'_i, $1 \leq i \leq n$, is a conjunctive contained rewriting of Q using views.

However, among all those contained rewritings $Q'_1, \ldots, Q'_n$, there might be an equivalent single conjunctive rewriting, say it is Q'_i, $Q'_i \equiv Q$, and i is in {1, ..., n}. Obviously, answering Q using Q'_i will be much more efficient than using Q'_M. Since Q'_i is already equivalent to Q, there is no need to evaluate all other contained rewritings in $Q'_1, \ldots, Q'_n$.

In application where a large number of views are available for query rewriting, the number of contained rewritings for a query could also be very large. Among those contained rewritings, there usually exist CE rewritings. For example, in our experiment in section 4.2.1, when the number of views is 450, 158 queries have 3273 contained rewritings in total. On average, a query Q has 20 contained rewritings. 116 out of these 158(73%) queries have CE rewritings. Suppose a query Q has 20 contained rewritings, and one of those rewritings, say Q'_E is equivalent to Q, then using Q'_E instead of Q'_M to answer query Q will have following significant benefits:

- Save the time on generating other 19 contained rewritings.
- Avoid evaluating all other 19 contained rewritings.
- Save the time on making a union of the results of all rewritings.

Suppose it takes the same time to evaluate every single conjunctive contained rewriting, then using Q'_E to answer Q will be at least 20 times faster than using Q'_M.

Therefore, there is a need to develop a query rewriting algorithm that produces only CE rewritings for data integration applications. We have explored two approaches to find CE rewriting. One is expanding equivalent rewriting algorithms, and the other one is utilizing complete rewriting algorithms.

3.2 Expanding equivalent rewriting algorithms

Equivalent query rewriting algorithms [23, 24] are designed for the purpose of query optimization. The result of those algorithms are required to be equivalent rewritings, but not necessary to be complete.

In an equivalent rewriting algorithm, the rewriting is achieved by a sequence of substitutions. Each substitution replaces one or more subgoals of the original query with that of the given views. In order to obtain an equivalent rewriting in the end, the algorithm guarantees the result of each substitution to be equivalent to the original query. In general, it requires that every substitution is a safe substitution [23], which means all subgoals and predicates of the view must be able to be mapped to their counterparts of the original query.

Due to the sequential application of the substitutions and the strict restriction on safe substitution, in certain cases such algorithm cannot find any complete rewriting even if there is one. For example, there is a query that has one subgoal referring to a relation R and two views refer to R. In this case, one view is used and the relation R is removed, hence the second view can never be used and some possible CE rewritings may be missed.

Example 3.1 *Given one query Q and two views V_1 and V_2 as below:*

$Q(a, d)$:– $A(a, b)$, $B(b, c)$, $C(c, d)$.

$V_1(a, b, c)$:– $A(a, b)$, $B(b, c)$.

$V_2(b, c, d)$:– $B(b, c)$, $C(c, d)$.

Obviously, V_1 can substitute subgoals A and B in Q, and V_2 can substitute subgoals B and C in Q. Because both two substitutions are safe substitutions, after the first iteration, an equivalent rewriting algorithm will generate two partial equivalent rewritings:

$Q'_1(a, d)$:– $V_1(a, b, c)$, $C(c, d)$.

$Q'_2(a, d)$:– $A(a, b)$, $V_2(b, c, d)$.

Nevertheless, in the next iteration, we find out that subgoal C in Q'_1 cannot be safe substituted by V_2. Because subgoal B has been replaced by V_1 already, B in V_2 is not able to be mapped to any subgoal in Q'_1. Likewise, situation exists between Q_2 and V_1. As a result, the algorithm stops, and Q'_1 and Q'_2 are the final result. However, the algorithm misses two CE rewritings:

$Q'_3(a, d)$:– $V_1(a, b, c')$, $V_2(b, c, d)$.

$Q'_4(a, d)$:– $V_1(a, b', c)$, $V_2(b, c, d)$.

In order to make use of equivalent query rewriting algorithms to find CE rewritings, we have to adjust the query to be rewritten, in every iteration of substitution, by adding some extra subgoals and predicates, so that we can continuously conduct safe substitutions until all source subgoals in the query are replaced by views. However, adding extra subgoals will add unnecessary overhead, and make the algorithm very complicated. Equivalent query rewriting algorithm, therefore, is not a proper start point for CE rewritings.

3.3 Expanding Complete Rewriting Algorithms

Complete rewriting algorithms are designed for data integration applications. Accordingly, the result of this kind of algorithms must be complete rewriting. There are two major types of complete rewriting algorithms, bucket based algorithm [5, 14, 15] and inverse rule based algorithm [2, 17]. Complete rewriting algorithms can always find maximally contained rewritings. It implies that if there exist equivalent rewritings of the original query based on the given views, these algorithms can find at least one of them. The only problem is that such algorithm does not generate any single conjunctive query as its result; instead, it produces a union of all contained conjunctive queries.

A naive approach of expanding bucket algorithm to get CE rewritings is to properly remove contained conjunctive queries from the resulting contained rewritings of a complete rewriting algorithm. For each contained rewriting, we can conduct a containment checking with query Q. If the conjunctive rewriting contains Q, it is a CE rewriting, otherwise it is not equivalent to Q and will be eliminated.

Example 3.2 *Suppose we have three relations, A(a,b,c), B(c,d) and C(d,e). A query and four views are as in Table 3-1.*

Q(a, e) :- A(a, b, c), B(c, d), C(d, e).	
V$_1$(a, c) :- A(a, b, c).	V$_3$(a, c) :- A(a, d, c), B(c, d).
V$_2$(c, e) :- B(c, d), C(d, e).	V$_4$(c, e, f) :- B(c, d), C(d, e), D(e, f).

Table 3-1: Query and views for example 3.1

In the first step, we apply a complete rewriting algorithm, i.e. MiniCon algorithm [14], and get a set of all possible contained rewritings. The rewritings and their expansions are shown in Table 3-2.

Q$_1$’(a, e) :- V$_1$(a, c), V$_2$(c, e). Q’$_{1\text{-exp}}$(a, e) :- A(a, b, c), B(c, d), C(d, e).
Q$_2$’(a, e) :- V$_3$(a, c), V$_2$(c, e). Q’$_{2\text{-exp}}$(a, e) :- A(a, d’, c), B(c, d’), B(c, d”), C(d”, e).
Q$_3$’(a, e) :- V$_1$(a, c), V$_4$(c, e, f). Q’$_{3\text{-exp}}$(a, e) :- A(a, b, c), B(c, d), C(d, e), D(e, f).

$Q_4'(a, e)$:- $V_3(a, c)$, $V_4(c, e, f)$.

$Q'_{4\text{-exp}}(a, e)$:- A(a, d', c), B(c, d'), B(c, d"), C(d", e), D(e, f).

Table 3-2: Contained rewritings and expansions for example3.1

In the second step, we impose a containment checking on each produced contained rewriting respectively, to test if it contains Q. If the contained rewriting contains Q, it is a CE rewriting, otherwise it is not equivalent to Q and will be eliminated. Among the four contained rewritings above, the CE rewriting is:

$Q'_1(a, e)$:- $V_1(a, c)$, $V_2(c, e)$.

In general, there may be a large number of contained rewritings that need to be removed when the size of the query and the size of the view set are large. Some of the produced rewritings are eliminated for the same reason. For example, both eliminated rewritings Q'_3 and Q'_4 refer to view V_4. Hence, if we can remove V_4 before generating the rewriting, the algorithm will be much more efficient.

A closer inspection on the example reveals that there are two cases when a view should not be used in rewriting. One is when the view contains an extra table as in V_4. For example, V_4 introduces a subgoal D that can not be mapped to any subgoal in Q. Another case is when a view introduced extra constraints on variables, even though there are no extra tables introduced, such as the case in V_3 where there is another join condition on the second attributes of A and B. In this case, variable d' in expansions of Q'_2 and Q'_4, which is introduced by V_3, can not be mapped to any variable in Q. Hence, Q'_2 and Q'_4 are not equivalent to Q.

Based on the observations above, we can see that view V_3 and V_4 should not be used to generate CE rewriting of Q. Therefore, we should eliminate V_3 and V_4 before rewritings are constructed, hence prevent non-equivalent rewriting Q'_2, Q'_3 and Q'_4 from being generated, and avoid afterward containment checking.

3.4 TCM Bucket Algorithm

We now propose an algorithm that expands the bucket algorithm, a complete rewriting algorithm, to generate CE rewriting, which is much efficient than the naïve approach in section 2.3. The main idea is to prevent inappropriate views from being added to the buckets in the first stage of the bucket algorithm, so that the rewritings produced in the combination stage are automatically CE rewriting without any extra containment checking.

3.4.1 Definitions

Before going into the detail of TCM bucket algorithm, we first introduce some definitions and terms used in the algorithm.

Definition 3.1 (Tail Containment Mapping)

Given queries V and Q, and a mapping φ from the variables in V to the variables in Q. φ is called a TCM mapping if each subgoal of V can be mapped to one of the subgoals in Q.

Example 3.3 *Suppose there are four relations and 4 queries as in Table 3-3.*

Relations	Queries
A(a, b, c)	$Q_1(x_1, u_1)$:- $A_1(x_1, y_1, z_1)$, $B_1(z_1, y_1)$, $C_1(z_1, u_1)$.
B(d, e)	$Q_2(x_2, t_2)$:- $A_2(x_2, y_2, z_2)$, $B_2(z_2, t_2)$.
C(f, g)	$Q_3(y_3, z_3)$:- $B_3(z_3, y_3)$, $C_3(z_3, y_3)$.
D(h, i)	$Q_4(z_4, s_4)$:- $C_4(z_4, u_4)$, $D_4(u_4, s_4)$.

Table 3-3: Queries in example for tail containment mappings

Suppose subgoals A_i, B_i, C_i and D_i refer to relations A, B, C and D respectively, then we can construct three mappings from Q_2, Q_3 and Q_4 to Q_1 as shown in Table 3-4.

φ_{21} : Q_2 --> Q_1				
V	x_2	y_2	z_2	t_2

$\varphi_{21}(V)$	x_1	y_1	z_1	y_1
$\varphi_{31} : Q_3$ --> Q_1				
V	z_3	y_3		
$\varphi_{31}(V)$	z_1	y_1		
$\varphi_{41} : Q_4$--> Q_1				
V	z_4	u_4		
$\varphi_{41}(V)$	z_1	u_1		

Table 3-4: all possible mappings in example 3.3

In mapping φ_{31}, y_3 in Q_3 is mapped to y_1 in Q_1, and z_3 in Q_3 is mapped to z_1 in Q_1. However, there is no subgoal $C(z_1, y_1)$ in Q_1, so that subgoal C_3 in Q_3 can not be mapped to any subgoal in Q1, which implies that φ_{31} is not a tail containment mapping. For φ_{41}, because it does not cover subgoal D_4 in Q_4, it is not a tail containment mapping either. While φ_{21} holds all conditions in definition 3.1, φ_{21} is the only tail containment mapping in Example 3.3.

Similar to containment mapping [18], tail containment mapping also requires that all relations referred by Q_1 are also referred by Q_2, and all join relations in Q_1 are retained in Q_2. These two conditions guarantee that query Q_2 has the same as or stricter conditions than the query Q_1 does.

The difference between tail containment mapping and containment mapping is that tail containment mapping does not require that all head variables of Q_2 can be mapped to head variables of Q_1. Intuitively, a tail containment mapping from Q_2 to Q_1 represents that without considering projection to head variables, tuples that satisfy Q_1 also satisfy Q_2.

In order to verify TCM mapping, we first introduce some definitions and theorems.

Definition 3.2 (Variable Range)

Given a query Q, for any variable $x \in Q$, the range of x in Q, denoted by R(x,Q), is the set of relation attributes to which x refers.

For example, there are two relations A(a,b,c) and B(d,e), and a query Q(z) :- A(x,y,z), B(z,u). Then the range of variable x in Q is {A.a}, i.e. R(x,Q) = {A.a}, and the range of z in Q is {A.c,B.d}, i.e. R(y,Q) = {A.c,B.d}.

The range of variable *x* in query *Q* reflects the specific join relationship in query *Q*. If *R(x,Q)* contains only one attribute, it means there is no join on that attribute. If *R(x,Q)* has more than one attributes, it implicates that in query *Q*, all those attributes are joined together.

Definition 3.3 *Given subgoal $S_1(x_1,.., x_n)$ from query Q_1, and $S_2(y_1, ...,y_n)$ from query Q_2, S_1 can be mapped to S_2, if*

1. *S_1 and S_2 refer to the same database relation, and*
2. *The range of any variable of S_1 is a subset of the range of corresponding variable in S_2, i.e. $R(x_i, Q_1) \subseteq R(y_i, Q_2)$, $1 \leq i \leq n$.*

Intuitively, the fact that subgoal S_1 can be mapped to S_2 means all the join predicates applied on S_1 in query Q_1 can be mapped to some join predicates on S_2 in query Q_2. This is very important to tell whether a tuple satisfying Q_2 can also satisfy Q_1, i.e., there is a TCM mapping from Q_1 to Q_2.

3.4.2 Expanding Bucket Algorithm with TCM

We now propose an algorithm that expands the bucket algorithm to generate CE rewriting, which is much efficient than the naive approach in section 3.3. The main idea is to refine the buckets generated in the first stage of the bucket algorithm, so that the rewritings produced in the combination stage are automatically CE rewriting without any extra containment checking.

We illustrate the TCM bucket algorithm with the following example.

Example 3.4 Given a query Q(x, r):-A(x, y), A(y, z), B(z, r)., and four views as below:

V_1(y, z, r):-A(y, z), B(z, r).	V_2(x, y, z):-A_1(x, y), A_2(y, z).
V_3(y, z):-A(y, z), B(z, y).	V_4(z, r, w):-B(z, r), C(r, w).

Table 3-5: Views in example 3.4

After the first stage of the bucket algorithm, the generated buckets are as below:

A(x, y)	A(y, z)	B(z, r)
V_1(x, y, r’)	V_1(y, z, r”)	V_1(y’, z, r)
V_2(x, y, z’)	V_2(x’, y, z)	V_3(y’, z)
V_3(y, z’)	V_3(y, z)	V_4(z, r, w’)

Table 3-6: Buckets after the first stage

Some views will always cause non-equivalent rewritings. Theorem 3.1 below can be used to decide which views should never be used to generating CE rewritings.

Theorem 3.1 If Q’ is a CE rewriting of query Q, each view referred by Q’ must have a TCM mapping to Q.

Proof: Since Q’ is equivalent to Q, there must be a containment mapping φ from the expansion of Q’, noted by Q'_{exp}, to Q. Therefore, according to definition 3.1, we have a conclusion:

Any subgoal of Q'_{exp} can be mapped to a subgoal of query Q. .... (1)

Suppose Q’ is in the form of Q’(X):- $V_1(X_1)$, · · · ,$V_n(X_n)$. We assume that view

$V_i(\bar{X}_i)$:- $S_1(\bar{Y}_1), \cdots, S_m(\bar{Y}_m)$ does not have TCM mapping to Q. So there must be a subgoal of view V_i, say $S(\bar{Y})$ that can not be mapped any subgoal of Q.

Suppose in the Q'_{exp}, subgoal $S(y_1,\ldots,y_n)$ of V_i is changed to $S(z_1,\ldots,z_n)$, where z_i is renamed from y_i, $1 \leq i \leq n$. There are two cases of z_i:

a) Variable z_i is a fresh new variable in Q'_{exp}, which means z_i does not appear in any views other than V_i. Then $R(z_i, Q'_{exp}) = R(y_i, V_i)$.

b) Variable z_i is a variable in Q, so that $R(z_i, Q'_{exp}) = R(y_i, V_i) \cup \ldots \cup R(x_n, V_n)$, where x_n is the variable in view V_n that is also renamed to z_i. Then we have $R(_{zi}, Q'_{exp}) \subseteq R(y_i, V_i)$.

Based on a) and b), for any variable y_i of subgoal $S(y_1,\ldots,y_n)$, $R(z_i, Q'_{exp}) \subseteq R(y_i, V_i)$, where z_i is the corresponding variable in subgoal $S(z_1,\ldots,z_n)$. Thus we can draw another conclusion:

A subgoal $S(y_1,\ldots,y_n)$ in view V_i can be mapped to the corresponding subgoal $S(z_1,\ldots,z_n)$ in Q'_{exp}. ……….(2)

While according to conclusion (1), subgoal $S(z_1,\ldots,z_n)$ can be mapped to a subgoal, say it is S_q, of Q. Then it is easy to infer that $S(y_1,\ldots,y_n)$ can also be mapped to S_q, which is contradict to the assumption that $S(\bar{Y})$ can not be mapped to any subgoal of Q. It means that any view V_i must have a TCM mapping to Q.

Q.E.D.

Intuitively speaking, Theorem 3.1 proves that views having no TCM mapping to a query Q can never be used to generate CE rewritings of Q, no matter how they are combined with other views. Based on the theorem 3.1, we develop the first rule of our algorithm,

Rule 1: If a view does not have a TCM mapping to the given query, remove it before running the bucket algorithm.

In Example 3.4, view V_3 and V_4 do not have TCM mapping to query Q. For V_3, because there is an extra join between subgoal A and B, none of them can be mapped to a subgoal of Q. For V_4, the extra subgoal C can not be mapped to any subgoal of Q. Therefore, after applying Rule 1, the buckets are as below:

A(x, y)	A(y, z)	B(z, r)
V_1(x, y, r')	V_1(y, z, r")	V_1(y', z, r)
V_2(x, y, z')	V_2(x', y, z)	

Table 3-7: Buckets after applying Rule 1.

However, only rule 1 is not enough to ensure that the resulting rewritings are all CE rewritings. For the combination {V_1(x, y, r'), V_1(y, z, r"), V_1(y', z, r)} from the bucket in Table 3-7, a rewriting can be formed as below:

Q_1'(x, r):- V_1(x, y, r'), V_1(y, z, r''), V_1(y', z, r).

After optimization, Q_1' becomes:

Q'$_1$(x, r):- V_1(x, y, r'), V_1(y, z, r).

Q'$_{1\text{-exp}}$(x, r):-A(x, y), B(y, r'), A(y, z), B(z, r).

Q'$_1$ is a properly contained rewriting, i.e. Q'$_1$$\subset$Q, because in the expansion of Q'$_1$, subgoal B(y, r') cannot be mapped to any subgoal of Q.

In order to prevent generating a contained rewriting like Q'$_1$, we need to refine the TCM mapping to Bucket-TCM mapping as below.

Definition 3.4 *(Bucket-TCM mapping)*

Given a bucket (or subgoal) $S(x_1, ..., x_n)$ in query Q, and a subgoal $S(y_1,...,y_n)$ in a view V. A mapping φ is a Bucket-TCM mapping from V to Q wrt $S(x_1, ..., x_n)$ if

- *$S(y_1,...,y_n)$ is mapped to the subgoal $S(x_1, ..., x_n)$; and*
- *All other subgoals in V are mapped to some subgoals in Q.*

Different from the TCM mapping, Bucket-TCM mapping is relevant to one particular subgoal. Through this definition, we are aware of that V_1 does not have a Bucket-TCM mapping to Q wrt the bucket S(x, y), hence V_1 cannot be added to this bucket.

Generalizing from this observation, we developed the following theorem:

Theorem 3.2 Given a rewriting Q':-V_1, · · · , V_n. of query Q:-S_1, …, S_n., the rewriting is generated using a bucket-based algorithm. Each view V_i, $1 \leq i \leq n$, is selected from a bucket of the S_i. Q' is a CE rewriting iff each V_i has a Bucket-TCM mapping to Q wrt S_i.

Proof :

Part 1 → (If Q' is a CE rewriting, each V_i has a Bucket-TCM mapping to Q wrt S_i.)

First, we assume that Q' is equivalent to Q, and Q' refers to a view V that does not have a Bucket-TCM mapping to Q wrt any $A_{i,}$ $1 \leq i \leq n$.

Because Q' is generated from bucket algorithm, V_i must be selected from a bucket, say it is B_i. Assume that B_i is of query subgoal S_i, according to the bucket algorithm, V_i must have a subgoal $S(x_1,\ldots,x_n)$ that refers to the same database table as $S_i(y_1,\ldots,y_n)$ does. And based on the assumption given at the beginning of this proof,

$S(x_1,\ldots,x_n)$ can not be mapped to $S_i(y_1,\ldots,y_n)$ ......(1)

Suppose in the $Q'_{exp,}$ subgoal $S(y_1,\ldots,y_n)$ is changed to $S(z_1,\ldots,z_n)$, and $S(x_1,\ldots,x_n)$ can be mapped to $S(z_1,\ldots,z_n)$. And because Q' is an equivalent rewriting, there is a containment mapping from Q'_{exp} to Q. Therefore $S(z_1,\ldots,z_n)$ can be mapped to a subgoal $S(t_1,\ldots,t_n)$ of query Q. Furthermore, we have:

$S(x_1,\ldots,x_n)$ can be mapped to $S(t_1,\ldots,t_n)$ ....(2)

From (1), (2), it is obvious that $S_i(y_1,\ldots,y_n)$ and $S(t_1,\ldots,t_n)$ are different subgoals of query Q.

On the other hand, because Q' is an equivalent rewriting of query, the bucket algorithm ensures that there is a containment mapping for Q to Q'_{exp} that maps subgoal $S(y_1,\ldots,y_n)$ to $S(z_1,\ldots,z_n)$, which means $Si(y_1,\ldots,y_n)$ can be mapped to $S(z_1,\ldots,z_n)$. Because $S(z_1,\ldots,z_n)$ can be mapped to query subgoal $S(t_1,\ldots,t_n)$, the conclusion can be drawn that $S_i(y_1,\ldots,y_n)$ can be mapped to $S(t_1,\ldots,t_n)$. As $S_i(y_1,\ldots,y_n)$ and $S(t_1,\ldots,t_n)$ are different subgoals of Q, it can be proved that $S_i(y_1,\ldots,y_n)$ is redundant in query Q. This is contradictory to the earlier assumption in this paper that no queries contain redundant subgoals.

In conclusion, the assumption at the beginning of this proof is not true.

Part 2 ← (If each V_i has a Bucket-TCM mapping to Q wrt S_i, Q' is a CE rewriting.)

Because the bucket algorithm guarantees that Q' is a complete rewriting and $Q' \subseteq Q$, we only need to prove $Q \subseteq Q'$.

Because there is TCM mapping from V_i to Q, upon Theorem 3.1, any subgoal $S(x_1,\ldots,x_m)$ of V_i can be mapped to a subgoal $S(y_1,\ldots,y_m)$ of Q, which means for any x_j, $1 \leq j \leq m$, $R(x_j, V_i) \subseteq R(y_j, Q)$.

Suppose in the Q'_{exp}, subgoal $S(x_1,\ldots,x_m)$ is changed to $S(z_1,\ldots,z_m)$, and variable z_j, $1 \leq j \leq m$, is renamed from x_j using following rule:

a) If x_j is not a distinguished variable from subgoal S_{Vi}, z_j is a fresh variable in Q'_{exp} which does not appear in subgoals from views other than V_i. So $R(z_j, Q'_{exp}) = R(x_j, V_i) \subseteq R(y_j, Q)$.

b) If x_j is a distinguished variable from subgoal S_{Vi} , x_j is renamed to y_j , i.e. $z_j = y_j$. According to the bucket algorithm, all variables that are mapped to y_j are renamed to z_j. So $R(z_j, Q'_{exp}) = R(x_j, V_i) \cup \ldots \cup R(t, V_t)$. Because S_{Vi} can be mapped to S_{qi}, $R(x_j, V_i) \subseteq R(y_j, Q)$, same as all other variables that are renamed to zj. So $R(z_j, Q'_{exp}) \subseteq R(y_j, Q)$.

Based on the two situations a) and b) above, any subgoal of V_i in Q'_{exp} can be mapped to a subgoal of query Q. Similarly, all other subgoals in Q'_{exp} can also be mapped to subgoals in query Q. Based on Definition 3.1, there is a TCM mapping from Q'_{exp} to Q.

In addition, the bucket algorithm guarantees that all the head variables of Q'_{exp} have 1:1 map to the head variables of query Q. Overall, a conclusion can be drawn that there is a containment mapping from Q'_{exp} to Q, which proves $Q \subseteq Q'$.

Q.E.D

Based on Theorem 3.2, we develop the second rule for our algorithm:

Rule 2: When a view V is added to a bucket A of query Q, V must have a Bucket-TCM mapping to Q wrt A.

For the buckets in Table 3-7, the only view that does not satisfy rule 2 is view $V_1(x, y, r')$ in the bucket of A (x, y). It can be verified that subgoal A(y, z) of V_1 can only be mapped to subgoal A(y, z) of Q, but not A(x, y) of Q. Based on rule 2, V_1 should not be added to the bucket of A(x, y). After rule 2 is applied, the buckets in Table 3-7 are changed as below:

$A_1(x, y)$	$A_2(y, z)$	B(z, r)
$V_2(x, y, z')$	$V_1(y, z, r'')$ $V_2(x', y, z)$	$V_1(y', z, r)$

Table 3-8: Buckets after applying rule 2

In the combination stage of the bucket algorithm, with the buckets in Table 3-8, two rewritings are generated as below. It can be verified that both Q'_2 and Q'_3 are CE rewritings.

$Q'_2(x, r)$:-$V_2(x, y, z')$, $V_1(y, z, r)$.

$Q'_{2\text{-exp}}(x, r)$:-$A_1(x, y)$, $A_2(y, z')$, A(y, z), B(z, r).

$Q'_3(x, r)$:-$V_2(x, y, z)$, $V_1(y', z, r)$.

$Q'_{3\text{-exp}}(x, r)$:-$A_1(x, y)$, $A_2(y, z)$, A(y', z), B(z, r).

Theorem 3.1 and 3.2 prove that by applying rule 1 and 2 in the first stage of the standard bucket algorithm, the algorithm only produces CE rewriting. We refer to this expanded bucket algorithm as TCM Bucket algorithm. Many rewritings that can be generated using standard bucket algorithm will not be produced by TCM bucket algorithm.

According to Theorem 3.2, none of those rewritings are equivalent to the original query. In another word, all CE rewritings that the standard bucket algorithm can create are also in the output of our TCM Bucket algorithm. Because the bucket algorithm can always produce maximally contained rewriting, if there exist CE rewritings, the bucket algorithm is guaranteed to find one. The same conclusion applies to TCM bucket algorithm.

3.4.3 TCM Bucket Algorithm Implementation

The TCM bucket algorithm expands the bucket algorithm, so that all produced rewritings are automatically CE rewritings without extra containment checking. Based on the first stage of bucket algorithm, TCM bucket algorithm needs to apply an extra TCM mapping to test which view should be put into which bucket.

First, we need to implement a procedure to find TCM mapping from a view to the given query Q. Based on the definition of tail containment mapping, the procedure to verify that there is a TCM mapping from V to Q falls into two steps.

In the first step, the algorithm attempts to find out all possible mappings from variables of V to variables of Q that cover all subgoals of view V. It first creates a matching set for each subgoal of V. Every matching set contains all subgoals of Q that refer to the same database relation as the corresponding view subgoal does. If any matching set is empty, which means the view has an extra subgoal that does not exist in query Q, the verification fails immediately, because in that case, there would be no tail containment mapping from Q to V.

Here we use the query Q and view V_1 in Example 3 to illustrate the procedure of finding TCM mappings.

Q(x, r):-A(x, y), A(y, z), B(z, r).

V_1(y, z, r):-A(y, z), B(z, r).

The matching set of the two subgoals A(y, z) and B(z, r) are as in Table 3-9,

A(y, z)	A(x, y), A(y, z)
B(z, r)	B(z, r)

Table 3-9: Matching sets for subgoals of view V1

Then, the procedure selects one subgoal from each matching set, and constructs a mapping from the view subgoals to those query subgoals. As for the example above, two mappings can be created as in Table 3-10.

V_1	A(y, z)	B(z, r)	y	z	r
$\varphi_1(V_1)$	A(x, y)	B(z, r)	x	y	r
$\varphi_2(V_1)$	A(y, z)	B(z, r)	y	z	r

Table 3-10: Constructed mappings

In the second step, we check all mappings found in the first step, to see if there exists a tail containment mapping. For view V_1, φ_1 is not a tail containment mapping, because φ_1 maps subgoal B(z, r) to B(y, r) which is not included in query Q. Mapping φ_2 satisfies all conditions required to be a tail containment mapping.

To apply rule 1, TCM mapping verification procedure is used to test each view. If a view V does not have any TCM mapping to Q, V will not be added to any bucket.

To apply rule 2, we make use of the TCM mappings found in the previous step. If there is a Bucket-TCM mapping from V to *Q* wrt the query subgoal of the bucket, the head of V will be renamed according to the mapping and added into the bucket.

The whole TCM bucket algorithm is described as in Figure 3-1.

```
/* TCM bucket algorithm*/
Begin
    /* The first stage*/

    For every view V in the bucket set B
        Call TCM mapping verification procedure
        If there is NO TCM mapping from V to Q
            Continue
        Else
            For every bucket Bi of query subgoal Si
                If has a Bucket-TCM mapping to Q wrt Si
                    Rename head variables of V
                    Add V into Bi
                Endif
            Endfor
        Endif
    Endfor

    /* The second stage*/
    Execute the combination stage algorithm of the
    bucket algorithm with bucket sets B to generate
    CE rewritings.
End
```

Figure 3-1: TCM bucket algorithm

In fact, the two rules in TCM bucket algorithm can also be applied to other bucket based algorithms, such as MiniCon, SVB Bucket, etc. Since all bucket based algorithms consist of bucket construction stage and combination stage, it is very easy to apply the two rules to bucket construction stage of any bucket based algorithm to make the algorithm capable of automatically generating CE rewriting without extra containment checking. In our experiment system, we integrate TCM and MiniCon to TCM-MiniCon algorithm, which is much more efficient than the TCM bucket algorithm.

3.4.4 Time Complexity

First, we analyze the time cost of finding TCM mappings from a view to a query. Given a query Q, assume that, within all relation tables referred by Q, there are *p* relation tables referred by multiple query subgoals, and maximally one relation is referred by *q* subgoals, then there could be q^p possible mappings for every view in the worst case. In addition, suppose m is the maximal number of subgoals in a view, t is the maximal number of variables in each subgoal, then the time to verify if a mapping is a TCM mapping is O(mt). Therefore the time complexity of finding TCM mapping from a view to query Q is $O(mtq^p)$.

In the first stage of TCM bucket algorithm, for each available view, we need to find TCM mappings from the view to the query, and put the view into different buckets according to the TCM mappings. Therefore if there are totally M views available, the time complexity of the first stage is $O(Mmtq^p)$.

The analysis above shows that the major performance problem is that the number of possible mappings grows exponentially in terms of q and p. However, in most real applications, queries that have many different subgoals referring to a same relation table are very rare. As a result, we can simply assume that q^p will never be greater than a constant G. Therefore, we can regard the time complexity of the second stage of TCM bucket algorithm as O(Mmt).

Suppose that the number of views in all buckets is M', the time complexity of the second stage of bucket algorithm is $O((nmM')^n)$ [14]. Therefore, the time complexity of the whole TCM bucket algorithm is $O(Mmt + (nmM')^n)$.

In the worst case where all views are good to generate CE rewriting, i.e., M=M', the time complexity of TCM bucket algorithm is $O((nmM')^n)$, which is as same as that of bucket algorithm. However, in most practical cases, applying TCM testing will dramatically decrease M'. It is the reason why in our experiments, TCM bucket algorithm always runs much faster than the bucket algorithm.

Chapter 4
Experiments

In this chapter, we will describe several experiments we have conducted to support our theory and algorithm. In order to reflect real situation of data integration applications, all data we use in our experiments are extracted from a real e-commerce system, rather than created by ourselves.

We will review the design, testing data and environment of our experiments in the first section. Then the result and analysis of our experiments will be presented. In the last section, we will briefly describe the implementation of the application of our experiments.

4.1 Overview

4.1.1 Experiment Design

In order to find CE rewritings precisely and efficiently, we expand the MiniCon algorithm with our TCM checking rules. We choose MiniCon algorithm, rather than other bucket-based algorithms, because of the following two reasons:

1. MiniCon is proved to be able to find out all possible complete rewritings.

2. In most cases, MiniCon algorithm outperforms basic bucket and inverse-rule algorithm

In our experiment system, given a database schema, a set of predefined views, and a query posed over the schema, we implement three methods to rewrite the query.

Method 1

Expand the MiniCon algorithm with our TCM checking rules to generate CE rewritings. We refer to this method as *TCM-MiniCon algorithm.*

Method 2

Expand the MiniCon algorithm with the naïve approach to generate CE rewritings. We refer to this method as *E-MiniCon algorithm.*

Method 3

MiniCon algorithm is directly applied to find out all contained rewritings, and then a maximally contained rewriting can be retrieved by simply making a union of all found contained rewritings. For this method, we simply refer to it as *MiniCon algorithm.*

Three experiments are designed to compare these three methods to validate the necessity, effectivity and efficiency of using TCM-MiniCon algorithm:

1. Compare the results of TCM-MiniCon and MiniCon algorithms to show that how using CE rewriting other than using maximally contained rewriting can benefit the evaluation of a query when a large number views can be used for rewriting.
2. Compare TCM-MiniCon algorithm with E-MiniCon algorithm in the number of

rewritings that can be finally produced to prove that, considering the result, TCM-MiniCon algorithm is as effective as E-MiniCon algorithm.

3. Compare TCM-MiniCon with E-MiniCon and MiniCon in the running time of rewriting a certain number of queries to testify the efficiency of TCM-MiniCon algorithm.

4.1.2 Experiments Data

The data used in our experiments are extracted from a real e-commerce system [25, 26]. The database schema of this system includes 389 relations, and on average, each relation contains 8 columns. There are 450 views defined over 212 relations in this schema. And 30 views, 6% of all available views, have more than 1 subgoal. The reason why most views only have one subgoal is because that every view here is converted from a certain member function of an EJB bean, which represents a single relation.

The queries we use in our experiments are randomly selected from the queries in the real system that satisfy the following conditions:

- Queries require distinct tuples.
- Queries are conjunctive and contain no aggregation predicates.

We totally collected 200 such queries, and on average, each query has 3 subgoals. These queries might refer to any relation in the database schema. Nevertheless, the views we have only cover 55% of all 389 relations, thus there might be some queries that do not have completely rewriting using the views available.

As the number of subgoals in views and queries is relatively small, the most important variable throughout the experiments is the number of views available. All the resulting data are averaged over multiple runs with the same parameters.

4.1.3 Experiment Environment

All programs in our experiments are developed using Java JSDK1.5.0 and run on a Pentium 4 2.0 GHz running Ubuntu Linux 6.06 with 1024 MB RAM.

4.2 Experiment Results

4.2.1 Necessity Validation

In this experiment, we rewrite all 200 queries using TCM-MiniCon algorithm and MiniCon algorithm respectively. By comparing the outputs of these two algorithms, we will realize how CE rewriting benefits the evaluation of a query when the number of views available increases.

1. The number of queries having rewritings

Firstly, it is necessary to know how many of the 200 testing queries have contained rewritings and how many of them have CE rewritings. Given a certain number of views, MiniCon algorithm is used to generate contained rewritings, while TCM-MiniCon algorithm is called to find CE rewritings. The results are displayed in Table 4-1and Figure 4-1.

Number of views	50	100	150	200	250	300	350	400	450
Queries having contained rewritings	25	35	73	102	112	154	154	158	158
Queries having CE rewritings	19	30	60	89	90	99	112	116	116

Table 4-1: The number of queries having contained or CE rewritings

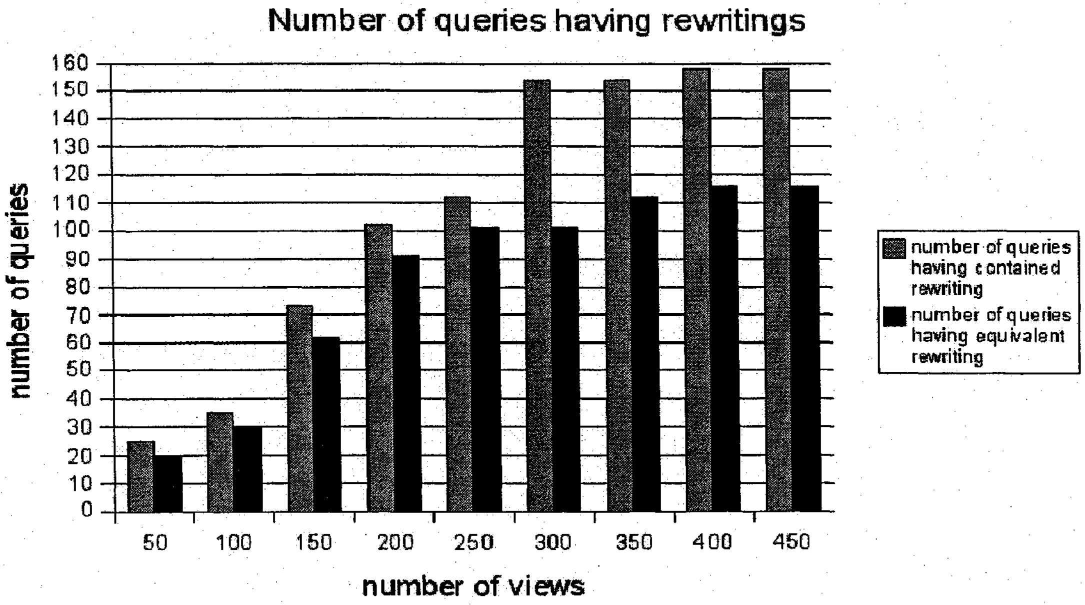


Figure 4-1: Number of queries having contained or CE rewritings

As we mentioned in the previous chapter of this thesis, given a certain number of views, not all queries have complete rewriting. As shown in Figure 4-1, both the number of queries having contained rewritings and the number of queries having CE rewriting greatly go up when the number of views increases. And in most cases, a major portion of the queries that have contained rewritings also have CE rewritings. For example, when 450 views are available, 158 queries (79% of all 200 testing queries) have contained rewritings, and 116 queries (58% of the same 200 queries) have CE rewritings. It means that 55% queries can be answered using CE rewritings, 24% queries have to be answered using maximally contained rewritings, and the rest 21% can not be answered using this set of views.

2. Query evaluation cost using different rewritings

Secondly, we compare the cost of answering a query using CE rewritings with the cost using maximally contained rewritings. Because we do not have a real database, we are not able to compare the real evaluation time of answering a query using these two types of rewritings. Therefore, we assume that every contained rewriting has the same evaluation time. Then, in order to compare the cost of answering a query using the two rewritings, we can simply compare the number of contained rewritings needed to be computed in each of these two methods.

Table 4-2 and Figure 4-2 show the average number of produced contained rewritings for each query that can be answered using maximally contained rewritings.

Number of views	50	100	150	200	250	300	350	400	450
Number of queries having contained rewritings	25	35	73	102	112	154	154	158	158
Total number of contained rewritings	51	90	202	371	454	1352	1813	2148	3273
Average number of contained rewritings per query	2	2.5	2.8	3.5	4	8.7	11.8	13.6	20.7

Table 4-2: Average number of contained rewritings per query

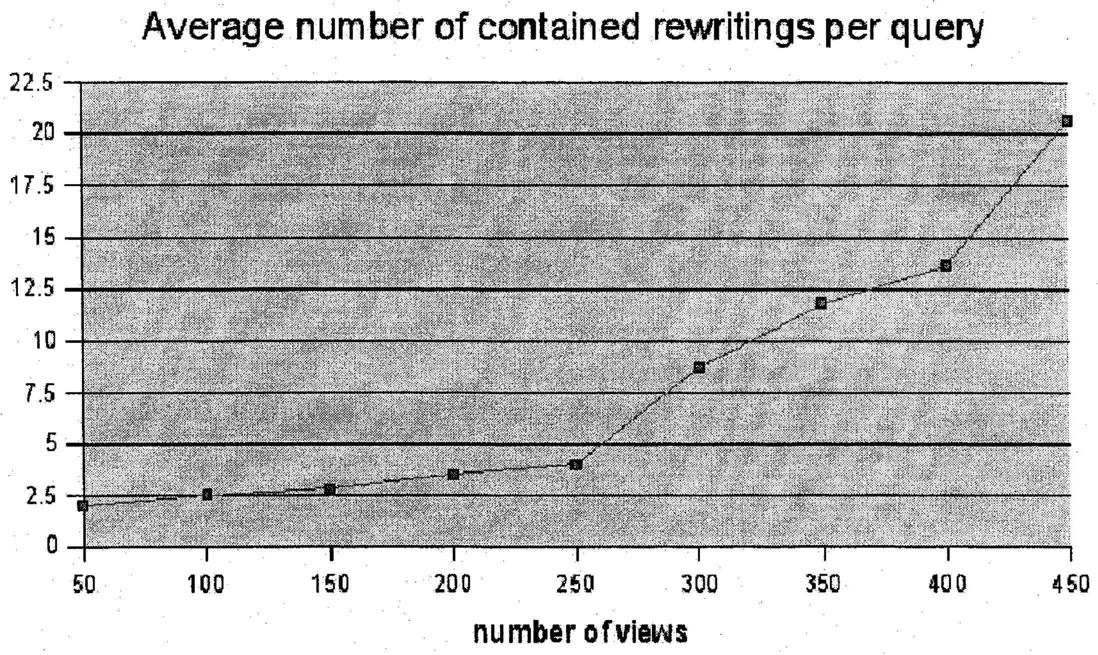


Figure 4-2: Average number of contained rewritings per query

From Figure 4-2, we notice that the average number of contained rewritings increases sharply as the set of views expands. When 450 views are available, there are 158 queries having contained rewritings, and on average, each of those queries has 20 contained rewritings. Consequently, to answer one of these queries using maximally contained rewritings now becomes to compute 20 contained rewritings and make a union of all the results.

As in Table 4-1, among those 158 queries, 116 queries have CE rewritings. These queries can be answered using a single CE rewriting. Using CE rewriting not only retrieves the same answering set as using maximally contained rewriting, but also saves the time to evaluate other 19 contained rewritings and to make the union. In another word, in this situation, answering a query using CE rewriting is at least 19 times faster than using maximally contained rewriting.

Overall, through the necessity validation experiment introduced in this section we observe that

(1) when the set of available views grows, both the number of queries having contained rewritings and the number of queries having CE rewritings greatly increase; meanwhile

(2) As the number of available views grows, the average number of contained rewritings a query may have dramatically rises, and accordingly so does the cost of answering query using maximally contained rewriting.

It is concluded that when there is a large number of views to be used in rewriting, answering query using CE rewriting instead of maximally contained rewriting is an essential way to speed up the process of query evaluation.

4.2.2 Correctness Validation

To validate the correctness of TCM–MiniCon algorithm, we rewrite all of the 200 queries using both TCM-MiniCon and MiniCon, and compare the results of these two algorithms in the number of queries having CE rewritings and the total number of CE rewritings found for all 200 queries.

Table 4-3 and Figure 4-3 give the number of queries having CE rewritings out of all 200 queries.

Number of views	50	100	150	200	250	300	350	400	450
TCM-MINICON	19	30	60	89	90	99	112	116	116
E-MiniCon	19	30	60	89	90	99	112	116	116

Table 4-3: The number of queries having CE rewritings

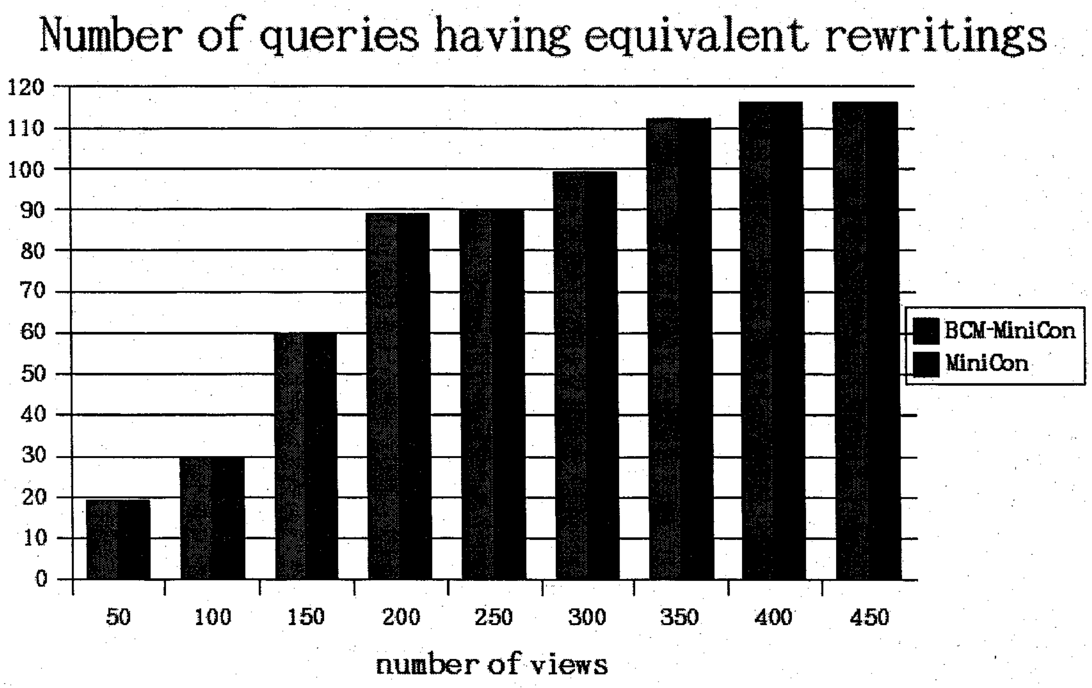


Figure 4-3: The number of queries having CE rewritings

Table 4-4 and Figure 4-4 give the total number of CE rewritings that can be found for all 200 queries.

Number of views	50	100	150	200	250	300	350	400	450
TCM-MINICON	45	80	130	176	219	239	351	362	404
E-MiniCon	45	80	130	176	219	239	351	362	404

Table 4-4: The total number of CE rewritings for all testing queries

We notice that TCM-MiniCon algorithm comes up with the same number of queries having CE rewritings and the same total number of CE rewritings for all 200 queries as E-MiniCon algorithm does. In conclusion, this experiment demonstrates that our

TCM-MiniCon algorithm can definitely compare to E-MiniCon algorithm in effectivity, because they always retrieve the same result.

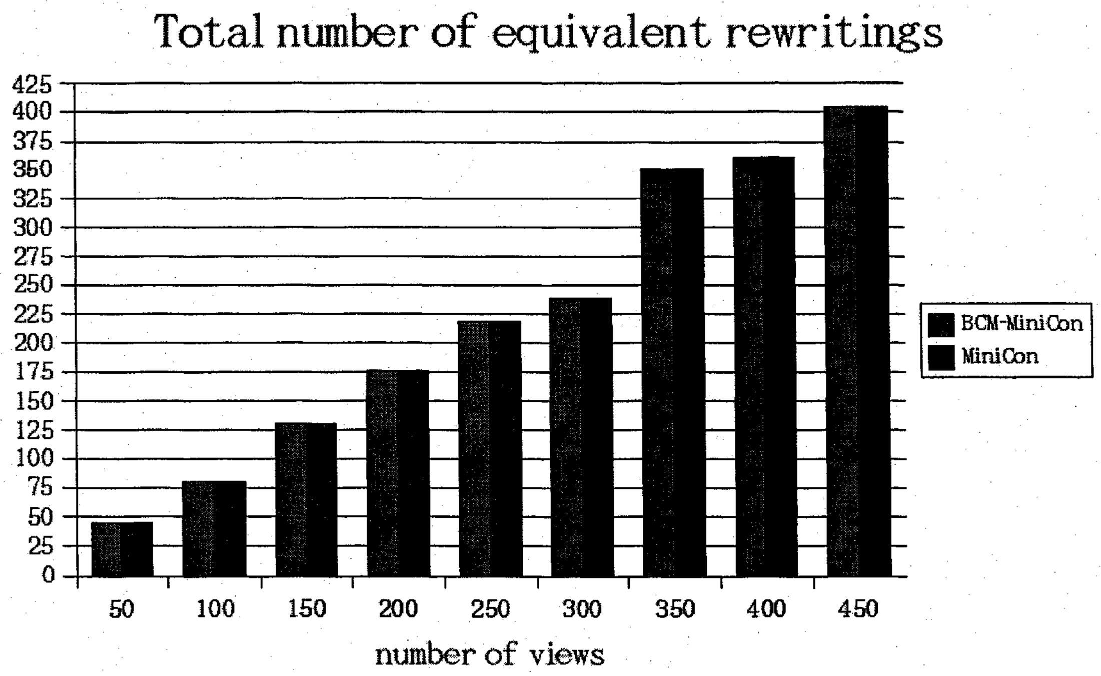


Figure 4-4: The total number of CE rewritings for all testing queries

4.2.3 Efficiency Validation

In this experiment, we rewrite all 200 queries using MiniCon, E-MiniCon and TCM-MiniCon respectively, and compare the running time of these three algorithms.

Number of views	50	100	150	200	250	300	350	400	450
E-MiniCon	748	1255	1871	2703	3263	5753	7575	9809	16032
MiniCon	671	1183	1808	2456	2752	4456	5583	7408	12156
TCM-MiniCon	445	527	788	1084	1154	1257	1655	1742	1932

Table 4-5: Rewriting time for 200 queries(in ms)

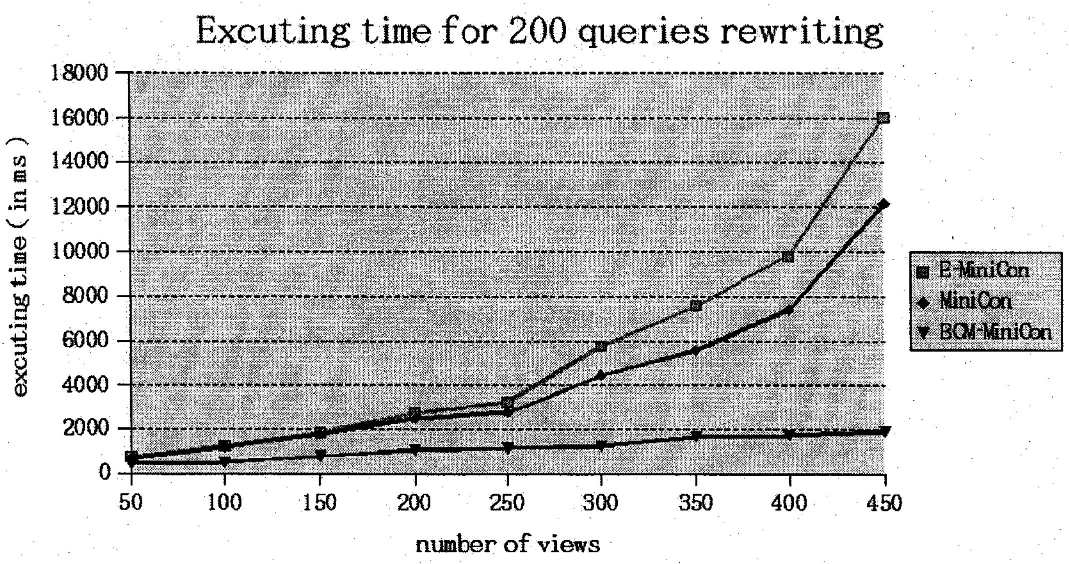


Figure 4-5: Rewriting time for 200 queries

From Table 4-5 and Figure 4-5, we have three observations:

1. With the same set of available views, TCM-MiniCon algorithm always costs less time than MiniCon and E-MiniCon to finish rewriting all 200 queries.
2. When the number of views grows, the performance superior of TCM-MiniCon to the other two algorithms dramatically increases.
3. In term of executing time, MiniCon algorithm also outperforms E-MiniCon algorithm in all cases.

In order to understand why TCM-MiniCon algorithm runs much faster than E-MiniCon and MiniCon, we first recall the time complexity of these three algorithms. Suppose M is the total number of all valid views that can be used to rewrite the given query, m is the maximal number of subgoals in a view, t is the is the maximal number of variables in each subgoal, and n is the number of subgoals in the query, then the time complexity of MiniCon algorithm is $O(nmM)^n$ [14]. E-MiniCon algorithm first calls MiniCon algorithm, and then processes the rewritings generated by MiniCon, it thereby always consumes more time then MiniCon.

On the other hand, TCM-MiniCon first calls TCM verification algorithm, whose time complexity is O(Mmt) (see section 3.3.2), to filter out all unsuitable views, then applies the second stage of MiniCon algorithm to directly generate CE rewritings. Suppose M' is the number of the views available in the second stage, then the time complexity of the whole TCM-MiniCon algorithm should be $O(Mmt + (nmM')^n)$.

Because in practical applications, m, n, and t are usually fairly small, M and M' will be the only critical factors that affect these algorithms' performance. Table 4-6 and Figure 4-6 show the number of all available views before and after applying TCM algorithm. The number of valid views varies according to different queries. Therefore, we record the average value for all 200 queries.

Number of views	50	100	150	200	250	300	350	400	450
TCM-MINICON	0.81	1.16	2.3	3.43	3.81	4.78	5.17	5.28	5.73
MiniCon	2.3	3.63	6.81	9.41	10.81	13.39	15.23	16.31	17.71

Table 4-6: The number of valid views in TCM-MiniCon and MiniCon algorithm

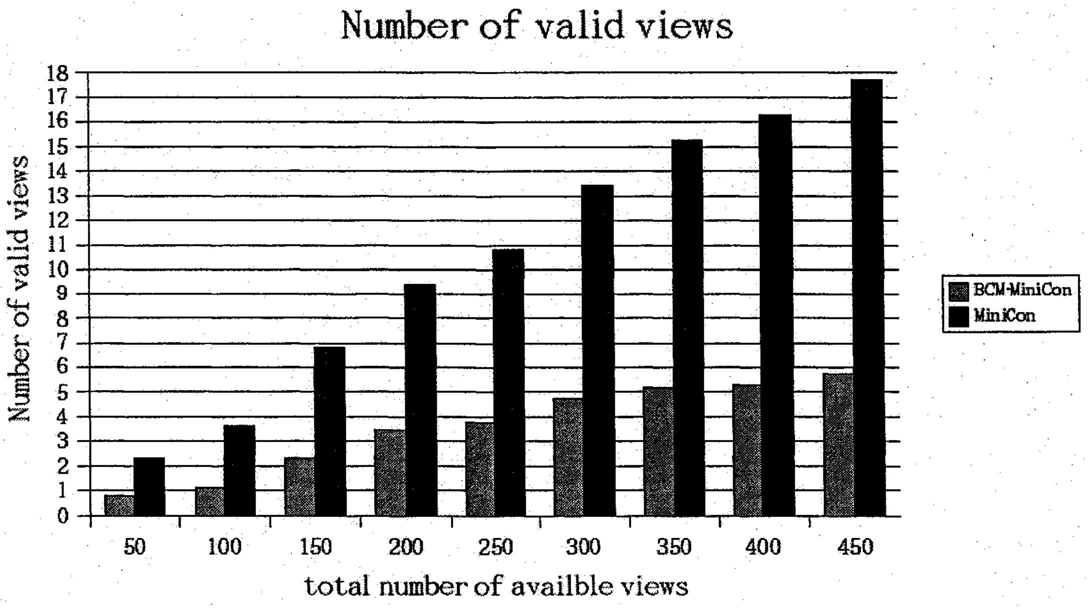


Figure 4-6: The difference of number of valid views between TCM-MiniCon and MiniCon algorithm

From Figure 4-6, we observe that for TCM-MiniCon algorithm, using TCM testing algorithm can reduce the size of valid views before the combination stage. It brings effect that is more significant when a larger number of views are available. This is because the time complexity of combination stage is exponential in size of the number of valid views, and the number of subgoals in the query, while TCM testing algorithm only has a linear time complexity in term of valid views. In conclusion, applying TCM before MiniCon can greatly improve the algorithm's performance in the application with large-scale view set and complex queries..

Here we also analyze the difference in executing time between MiniCon and E-MiniCon algorithm. E-MiniCon carries out the afterward containment checking approach. In order to get CE rewriting, it calls MiniCon algorithm first, then applies a containment checking on every single created contained rewriting. For a given query, there could be $O(nmM)^n$ contained rewritings generated by MiniCon algorithm. From the first experiment, we know that when 450 views are available for rewriting, there are 3273 contained rewritings created for 200 queries. Therefore, applying containment checking on each of these 3273 contained rewritings extremely slows down the E-MiniCon algorithm.

4.2.4 Summary

The observations of our experiments are summarized as below:

1. When the number of views is very large, there generally are many contained rewritings for a single query. In such circumstance, answering query using CE rewriting is much more efficient than using maximally contained rewritings.
2. In term of rewriting result, TCM-MiniCon is just as good as E-MiniCon. The two algorithms always have the exactly same output.

3. Speaking of performance, TCM-MiniCon is superior to both E-MiniCon and MiniCon. Applying TCM algorithm effectively improves the performance of the entire rewriting algorithm, especially in the environment with massive views and complex queries.

4.3 Implementation

Our experimental query rewriting system is developed using Java on Linux platform. All java class source codes are compiled and executed on the latest J2SE 1.5.0. During the development, we use Eclipse as the IDE(Integrated Development Environment), Apache Ant as package management software, and CVS as version control software.

Our system includes several libraries that provide classes and functions for query parsing, translating and rewriting. We also develop a command line interface and a GUI(Graphic User Interface) to test various query rewriting algorithms in different situations. Figure 4-7 shows all the packages in our system. Below is the briefly description of each package.

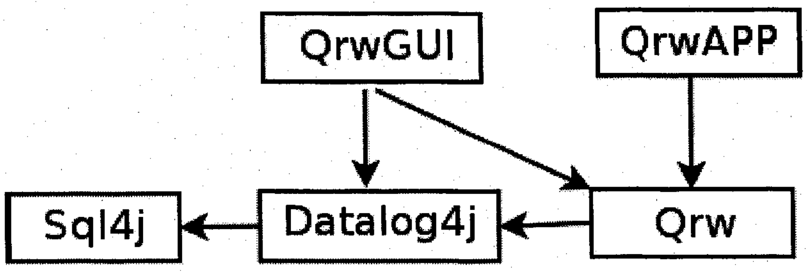


Figure 4-7: Package relationship in experiment rewriting system

- **Sql4j** is a SQL parser that can parse a SQL string and fill in java structures that represent SQL statements and expressions [27].
- **Datalog4j** defines the datalog class that is a semantic expression of queries. This package also provides functions to translate a query in SQL into a datalog expression [28].
- **Qrw** is the most critical class in our system. It contains an abstract class, *QRewriter*, defining a set of uniform interfaces for query rewriter. Besides, there are three subclasses derived from QRewriter, which are TCM-MiniCon, E-MiniCon, and MiniCon.
- QrwGUI provides graphic interface to show database environment (schemas and view definitions) and to rewrite a query using different algorithms. It is convenient to compare results of different rewriting algorithms for a single query. See Appendix C for the screen-shots of QrwGUI.

QrwGUI and QrwApp take a SQL query string as input and produce a set of CE rewritings or contained rewritings as output. In order to use QrwGUI and QrwAPP correctly, user must provide the definitions of database schema and views. They are defined using two XML files. The sample XML files for definitions of the database schema and views used in our experiment can be found in Appendix A.

Chapter 5
Conclusion

5.1 Summary

Based on the bucket algorithm, we propose TCM bucket algorithm to find CE rewritings of a query. It applies TCM mapping to test which view should be put into which bucket, so that each produced rewriting is automatically a CE rewriting without extra containment checking.

In order to filter out the inappropriate view from the buckets, we developed two rules:

Rule 1 If a view does not have a TCM mapping to the given query, remove it before running the bucket algorithm.

Rule 2 When adding a view V to a bucket S of query Q, V must have a Bucket-TCM mapping to Q wrt S.

We prove that with these two rules, all the rewriting generated by TCM bucket algorithm are CE rewriting without applying extra containment checking, and TCM bucket algorithm can always find a CE rewriting if there exits one. Moreover, rule 1 and 2 can be easily integrated with any bucket based algorithm, such as MiniCon algorithm and SVB algorithm, so that they are also able to generate CE rewritings.

A series of experiments have been carried out to validate the necessity, effectiveness and efficiency of our algorithm. All testing data we use in our experiments are extracted from a real data integration project, which includes 450 views and 200 queries. In our testing system, the experiments are undertaken upon the following three query rewriting methods.

1. MiniCon

   A popular maximally contained rewriting algorithm based on bucket algorithm

2. E-MiniCon

   The naïve extension of MiniCon algorithm. It obtains CE rewritings by removing properly contained rewritings from all rewritings generated by MiniCon algorithm.

3. TCM-MiniCon

   It integrates the two rules in TCM Bucket algorithm with MiniCon algorithm to produce CE rewritings

The result of these experiments shows three facts that support our algorithm.

1. When the number of views is very large, there generally are many contained rewritings for a single query. In such circumstance, answering query using CE rewriting is much more efficient than using maximally contained rewritings.
2. In term of rewriting result, TCM-MiniCon is just as good as E-MiniCon. The two algorithms always have the exactly same output.
3. Speaking of performance, TCM-MiniCon is superior to both E-MiniCon and

MiniCon algorithm. Applying TCM algorithm effectively improves the performance of the whole rewriting algorithm, especially in the environment with massive views and complex queries.

5.2 Future work

There are three aspects to improve our method of finding CE rewritings:

1. Consider the value domain of all variables when verify tail containment mapping so that TCM algorithm can handle queries and views having comparison predicates. Some of the ideas for handling comparison predicates have been mentioned in [29, 30, 31, 32].
2. Extend our algorithms to handle integrity constrains, such as inclusion dependencies. It will enable our algorithms to find out more CE rewritings. There are some papers, such as [33, 34, 35], discussing the problem about query rewriting with inclusion dependencies.
3. Our algorithm usually generates more than one CE rewritings of the given query, and randomly picks one as the final answer. For the further improvement, we need to introduce a cost model in evaluating views in TCM algorithm. With this cost model, TCM algorithm will be able to filter out more views, and guarantee that the complete rewriting algorithm can generate the best rewriting for the given query.

Appendices

Appendix A Rewriting Environment

A rewriting environment contains database schema and view definitions. They are defined in two XML files: schema.xml and views.xml. Below is an example for the University schema and views.

1. schema.xml

```
<?xml version=’1.0’?>
<Schema DatabaseName="ViewSurvey">
<TableName Name="Prof">
    <ColumnName Name="name" TypeName="CHARACTER"/>
    <ColumnName Name="area" TypeName="CHARACTER"/>
    <ColumnName Name="age" TypeName="CHARACTER"/>
    <PrimaryKey>
        <PrimaryKeyColumnName Name="name" SequenceName = "1"/>
    </PrimaryKey>
    <ForeignKey>
    </ForeignKey>
</TableName>
<TableName Name="Course">
    <ColumnName Name="c_number" TypeName="CHARACTER"/>
    <ColumnName Name="title" TypeName="CHARACTER"/>
    <PrimaryKey>
```

```
        <PrimaryKeyColumnName Name="c_number" SequenceName = "1"/>
    </PrimaryKey>
    <ForeignKey>
    </ForeignKey>
</TableName>
<TableName Name="Teaches">
    <ColumnName Name="prof" TypeName="CHARACTER"/>
    <ColumnName Name="c_number" TypeName="CHARACTER"/>
    <ColumnName Name="quarter" TypeName="CHARACTER"/>
    <ColumnName Name="evaluation" TypeName="CHARACTER"/>
    <PrimaryKey>
        <PrimaryKeyColumnName Name="c_number" SequenceName = "1"/>
        <PrimaryKeyColumnName Name="quarter" SequenceName = "2"/>
    </PrimaryKey>
    <ForeignKey>
        <ForeignKeyColumnName Name="prof">
              <ForeignKeyReferenceTableName Name="Prof"/>
              <ForeignKeyReferenceColumnName Name="name"/>
              <ForeignKeySequence Name="1"/>
        </ForeignKeyColumnName>
        <ForeignKeyColumnName Name="c_number">
              <ForeignKeyReferenceTableName Name="Course"/>
              <ForeignKeyReferenceColumnName Name="c_number"/>
              <ForeignKeySequence Name="1"/>
        </ForeignKeyColumnName>
        </ForeignKey>
</TableName>
<TableName Name="Registered">
    <ColumnName Name="student" TypeName="CHARACTER"/>
    <ColumnName Name="c_number" TypeName="CHARACTER"/>
    <ColumnName Name="quarter" TypeName="CHARACTER"/>
    <PrimaryKey>
        <PrimaryKeyColumnName Name="student" SequenceName = "1"/>
        <PrimaryKeyColumnName Name="c_number" SequenceName = "2"/>
        <PrimaryKeyColumnName Name="quarter" SequenceName = "3"/>
    </PrimaryKey>
        <ForeignKey>
            <ForeignKeyColumnName Name="c_number">
                   <ForeignKeyReferenceTableName Name="Course"/>
                   <ForeignKeyReferenceColumnName Name="c_number"/>
                   <ForeignKeySequence Name="1"/>
            </ForeignKeyColumnName>
            <ForeignKeyColumnName Name="student">
                   <ForeignKeyReferenceTableName Name="Major"/>
                   <ForeignKeyReferenceColumnName Name="student"/>
                   <ForeignKeySequence Name="1"/>
            </ForeignKeyColumnName>
            <ForeignKeyColumnName Name="c_number">
                   <ForeignKeyReferenceTableName Name="Teaches"/>
                   <ForeignKeyReferenceColumnName Name="c_number"/>
                   <ForeignKeySequence Name="1"/>
            </ForeignKeyColumnName>
            <ForeignKeyColumnName Name="quarter">
                   <ForeignKeyReferenceTableName Name="Teaches"/>
                   <ForeignKeyReferenceColumnName Name="quarter"/>
                   <ForeignKeySequence Name="2"/>
            </ForeignKeyColumnName>
        </ForeignKey>

</TableName>
<TableName Name="Major">
    <ColumnName Name="student" TypeName="CHARACTER"/>
    <ColumnName Name="dept" TypeName="CHARACTER"/>
    <PrimaryKey>
        <PrimaryKeyColumnName Name="student" SequenceName = "1"/>
    </PrimaryKey>
    <ForeignKey>
    </ForeignKey>
```

```
</TableName>
<TableName Name="Advises">
    <ColumnName Name="prof" TypeName="CHARACTER"/>
    <ColumnName Name="student" TypeName="CHARACTER"/>
    <PrimaryKey>
          <PrimaryKeyColumnName Name="prof" SequenceName = "1"/>
          <PrimaryKeyColumnName Name="student" SequenceName = "2"/>
    </PrimaryKey>
    <ForeignKey>
          <ForeignKeyColumnName Name="prof">
                <ForeignKeyReferenceTableName Name="Prof"/>
                <ForeignKeyReferenceColumnName Name="name"/>
                <ForeignKeySequence Name="1"/>
          </ForeignKeyColumnName>
          <ForeignKeyColumnName Name="student">
                <ForeignKeyReferenceTableName Name="Major"/>
                <ForeignKeyReferenceColumnName Name="student"/>
                <ForeignKeySequence Name="1"/>
          </ForeignKeyColumnName>
     </ForeignKey>
</TableName>
</Schema>
```

2. views.xml

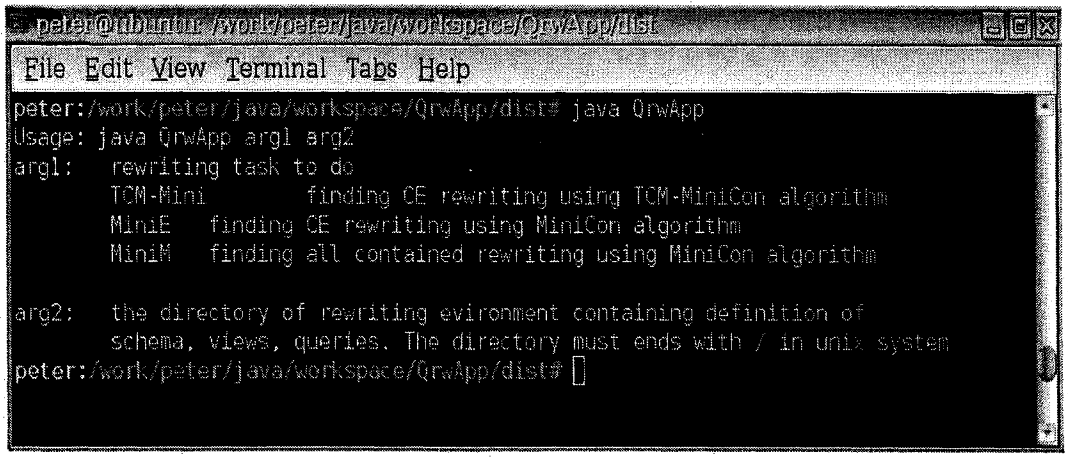
```
<?xml version="1.0"?>
<views>
<view name="V1"> <Query>
        SELECT Registered.student, Registered.c_number,
               Course.title
        FROM Registered, Course
        WHERE Registered.c_number=Course.c_number
              AND Course.title = 'DB'
  </Query>
```

```
</view>

<view name="V2"> <Query>
        SELECT Registered.student, Teaches.prof,
               Registered.c_number, Teaches.evaluation
        FROM Registered, Teaches
        WHERE Registered.c_number=Teaches.c_number
              AND Registered.quarter=Teaches.quarter
  </Query>
</view>
<view name="V3"> <Query>
        SELECT Registered.student, Registered.c_number
        FROM Registered
</Query>
</view>
<view name="V4"> <Query>
        SELECT Teaches.prof, Teaches.c_number, Course.title
        FROM Registered, Course, Teaches
        WHERE Registered.c_number=Teaches.c_number
        AND Registered.c_number=Course.c_number
</Query>
</view>
 <view name="V5"> <Query>
        SELECT Prof.name, Teaches.c_number, Teaches.quarter,
               Prof.area, Teaches.evaluation
        FROM Teaches, Prof
        WHERE Teaches.prof=Prof.name
        And Teaches.evaluation = 70
</Query>
</view>
</views>
```

Appendix B Demonstration of QrwApp

QrwApp is a command line interface of our query rewriting system. We implemented many different query rewriting algorithms in Qrw library. Taking advantage of Qrw library, QrwApp provides a way to find contained rewritings or equivalent rewritings of queries using different rewriting algorithms. Another significant feature of QrwApp is that it is able to handle queries in batches.

a) Usage of QrwApp


peter@ubuntu: /work/peter/java/workspace/QrwApp/dist
File Edit View Terminal Tabs Help

```
peter:/work/peter/java/workspace/QrwApp/dist# java QrwApp
Usage: java QrwApp arg1 arg2
arg1:   rewriting task to do
        TCM-Mini        finding CE rewriting using TCM-MiniCon algorithm
        MiniE   finding CE rewriting using MiniCon algorithm
        MiniM   finding all contained rewriting using MiniCon algorithm

arg2:   the directory of rewriting evironment containing definition of
        schema, views, queries. The directory must ends with / in unix system
peter:/work/peter/java/workspace/QrwApp/dist# 
```

b) Demonstration of QrwApp

Suppose we have two queries based on the schema defined in Appendix A as below:

Q1:

SELECT Registered.student, Registered.c_number, Registered.quarter, Course.title

FROM Registered, Course

WHERE Registered.c_number=Course.c_number

Q2:

SELECT Registered.student, Registered.c_number, Teaches.prof

FROM Teaches, Registered, Course

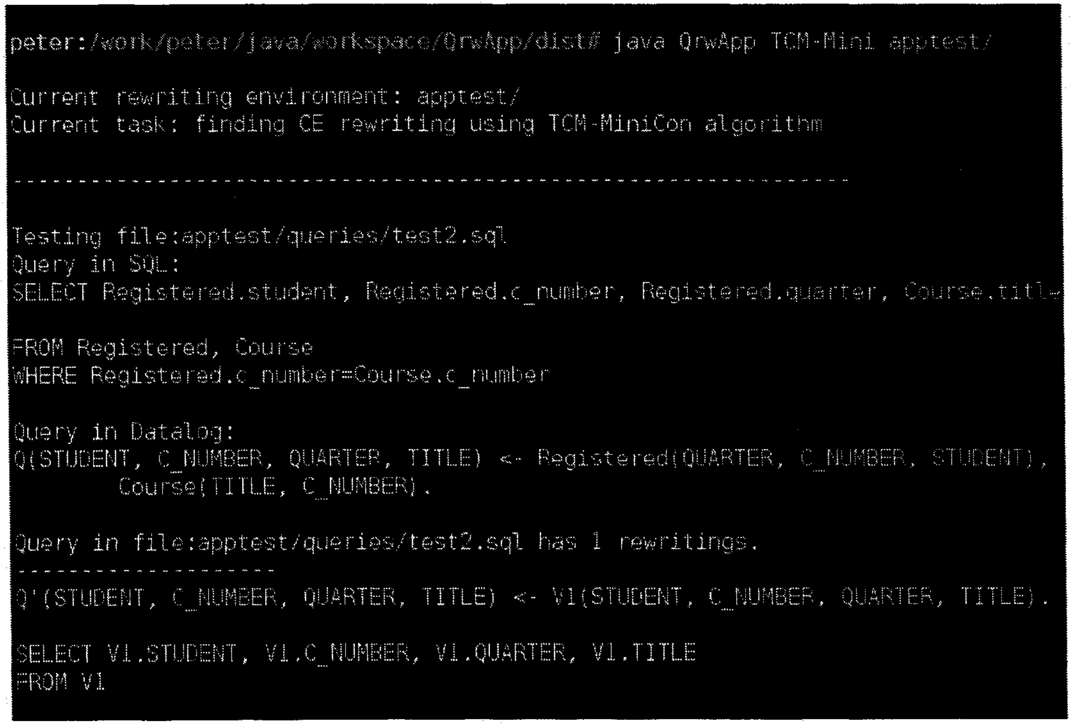
WHERE Teaches.c_number=Registered.c_number

AND Registered.quarter = Teaches.quarter

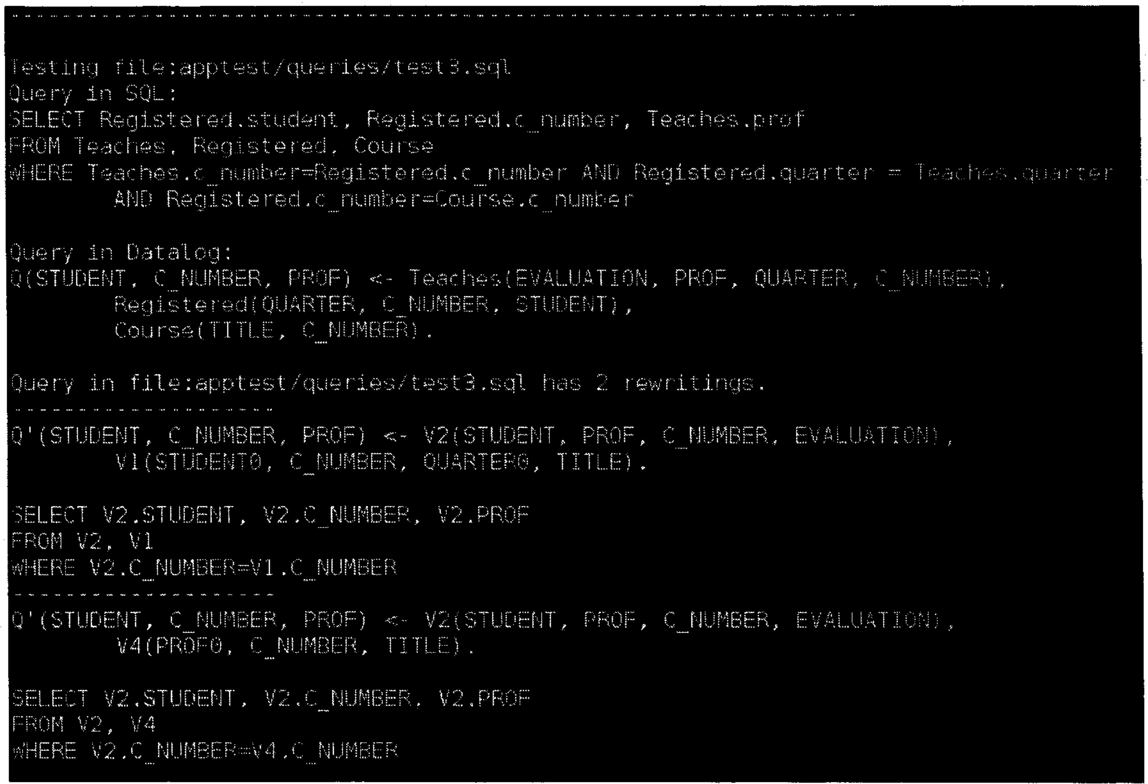
AND Registered.c_number=Course.c_number

Before running QrwApp, we put definition files of schemas, views and those two queries into one directory, for example aptest. Then we run QrwApp with TCM-MiniCon and E-MiniCon respectively.

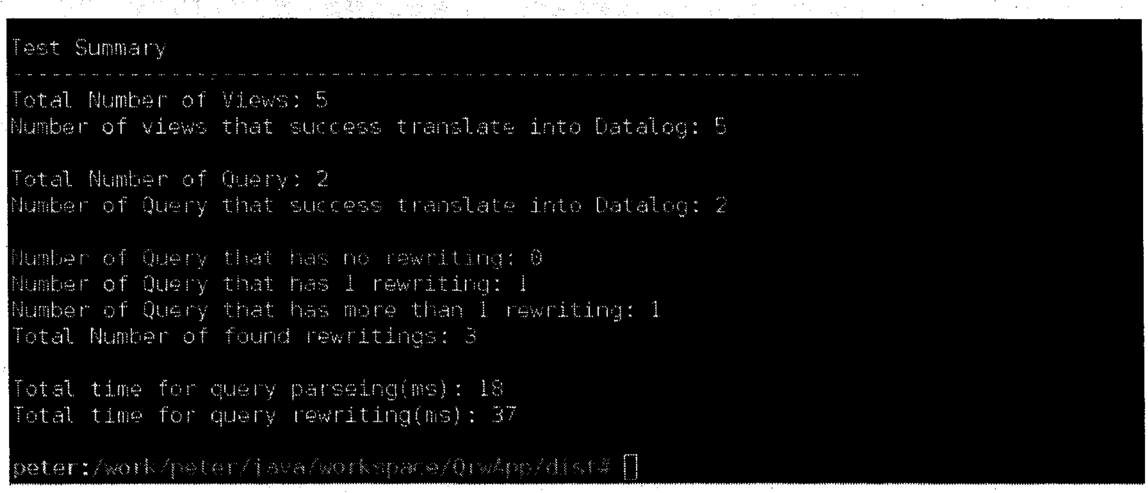
- **Result of running QrwApp using TCM-MiniCon**



```
peter:/work/peter/java/workspace/QrwApp/dist# java QrwApp TCM-Mini apptest/

Current rewriting environment: apptest/
Current task: finding CE rewriting using TCM-MiniCon algorithm

-----------------------------------------------------------------

Testing file:apptest/queries/test2.sql
Query in SQL:
SELECT Registered.student, Registered.c_number, Registered.quarter, Course.titl

FROM Registered, Course
WHERE Registered.c_number=Course.c_number

Query in Datalog:
Q(STUDENT, C_NUMBER, QUARTER, TITLE) <- Registered(QUARTER, C_NUMBER, STUDENT),
        Course(TITLE, C_NUMBER).

Query in file:apptest/queries/test2.sql has 1 rewritings.
--------------------
Q'(STUDENT, C_NUMBER, QUARTER, TITLE) <- V1(STUDENT, C_NUMBER, QUARTER, TITLE).

SELECT V1.STUDENT, V1.C_NUMBER, V1.QUARTER, V1.TITLE
FROM V1
```

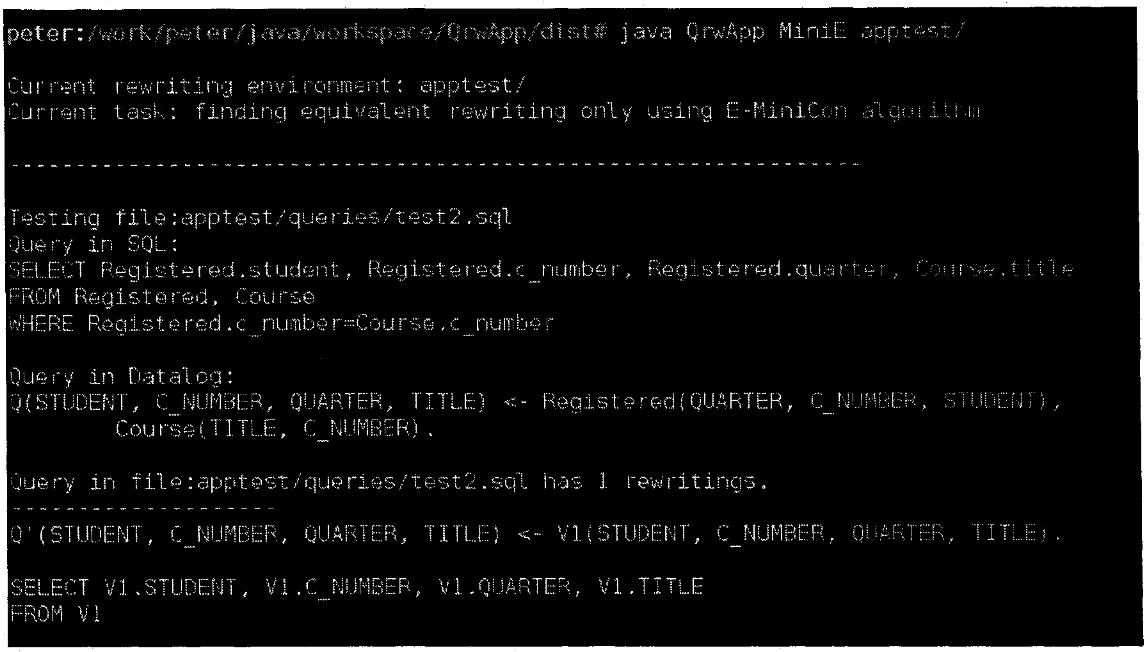


```
----------------------------------------------------------------

Testing file:apptest/queries/test3.sql
Query in SQL:
SELECT Registered.student, Registered.c_number, Teaches.prof
FROM Teaches, Registered, Course
WHERE Teaches.c_number=Registered.c_number AND Registered.quarter = Teaches.quarter
        AND Registered.c_number=Course.c_number

Query in Datalog:
Q(STUDENT, C_NUMBER, PROF) <- Teaches(EVALUATION, PROF, QUARTER, C_NUMBER),
        Registered(QUARTER, C_NUMBER, STUDENT),
        Course(TITLE, C_NUMBER).

Query in file:apptest/queries/test3.sql has 2 rewritings.
--------------------
Q'(STUDENT, C_NUMBER, PROF) <- V2(STUDENT, PROF, C_NUMBER, EVALUATION),
        V1(STUDENT0, C_NUMBER, QUARTER0, TITLE).

SELECT V2.STUDENT, V2.C_NUMBER, V2.PROF
FROM V2, V1
WHERE V2.C_NUMBER=V1.C_NUMBER
--------------------
Q'(STUDENT, C_NUMBER, PROF) <- V2(STUDENT, PROF, C_NUMBER, EVALUATION),
        V4(PROF0, C_NUMBER, TITLE).

SELECT V2.STUDENT, V2.C_NUMBER, V2.PROF
FROM V2, V4
WHERE V2.C_NUMBER=V4.C_NUMBER
```

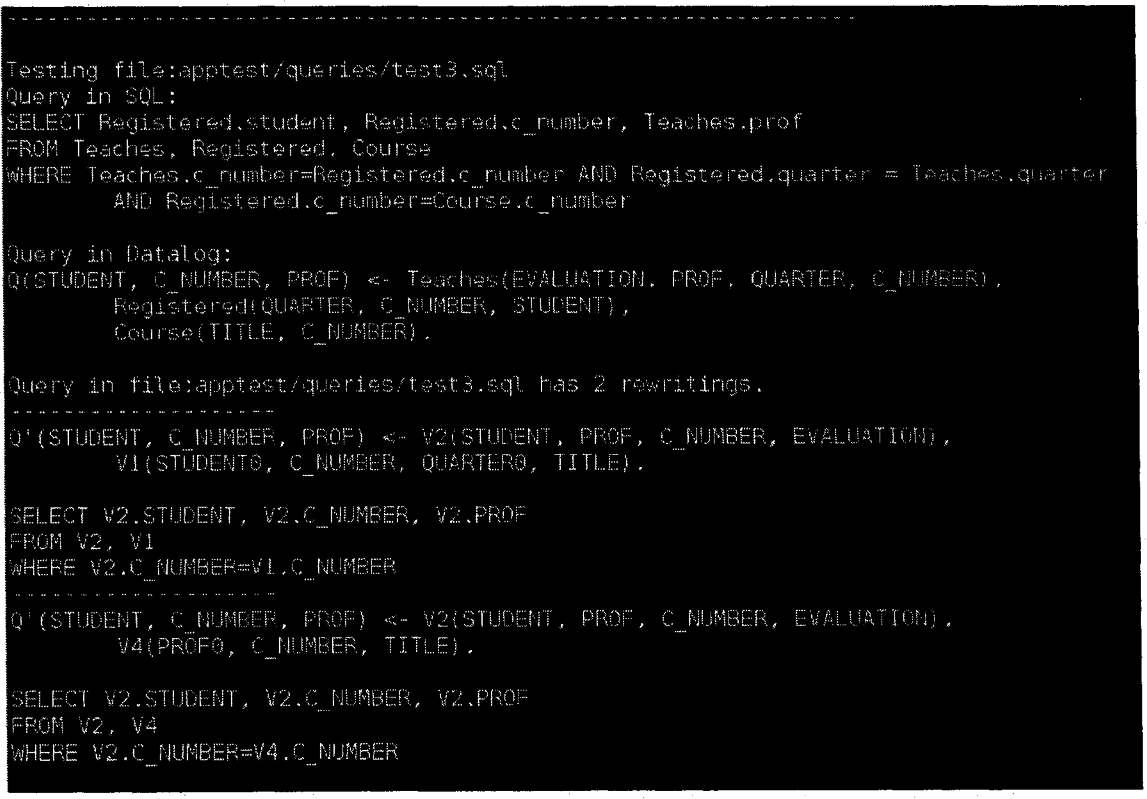


```
Test Summary
----------------------------------------------------------------
Total Number of Views: 5
Number of views that success translate into Datalog: 5

Total Number of Query: 2
Number of Query that success translate into Datalog: 2

Number of Query that has no rewriting: 0
Number of Query that has 1 rewriting: 1
Number of Query that has more than 1 rewriting: 1
Total Number of found rewritings: 3

Total time for query parseing(ms): 18
Total time for query rewriting(ms): 37

peter:/work/peter/java/workspace/QrwApp/dist# 
```

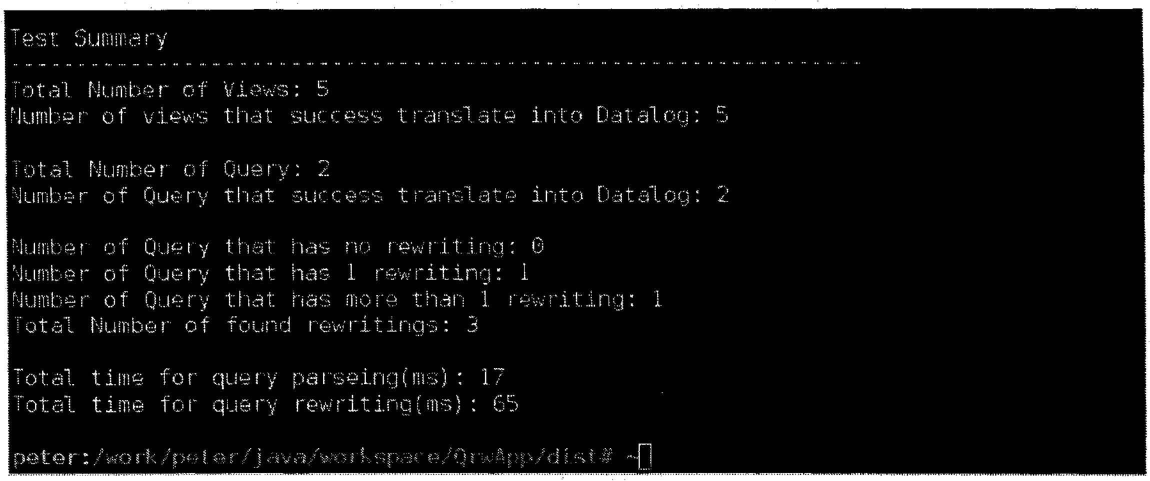
- **Result of running QrwApp using E-MiniCon**



```
peter:/work/peter/java/workspace/QrwApp/dist# java QrwApp MiniE apptest/

Current rewriting environment: apptest/
Current task: finding equivalent rewriting only using E-MiniCon algorithm

-----------------------------------------------------------------

Testing file:apptest/queries/test2.sql
Query in SQL:
SELECT Registered.student, Registered.c_number, Registered.quarter, Course.title
FROM Registered, Course
WHERE Registered.c_number=Course.c_number

Query in Datalog:
Q(STUDENT, C_NUMBER, QUARTER, TITLE) <- Registered(QUARTER, C_NUMBER, STUDENT),
        Course(TITLE, C_NUMBER).

Query in file:apptest/queries/test2.sql has 1 rewritings.
--------------------
Q'(STUDENT, C_NUMBER, QUARTER, TITLE) <- V1(STUDENT, C_NUMBER, QUARTER, TITLE).

SELECT V1.STUDENT, V1.C_NUMBER, V1.QUARTER, V1.TITLE
FROM V1
```

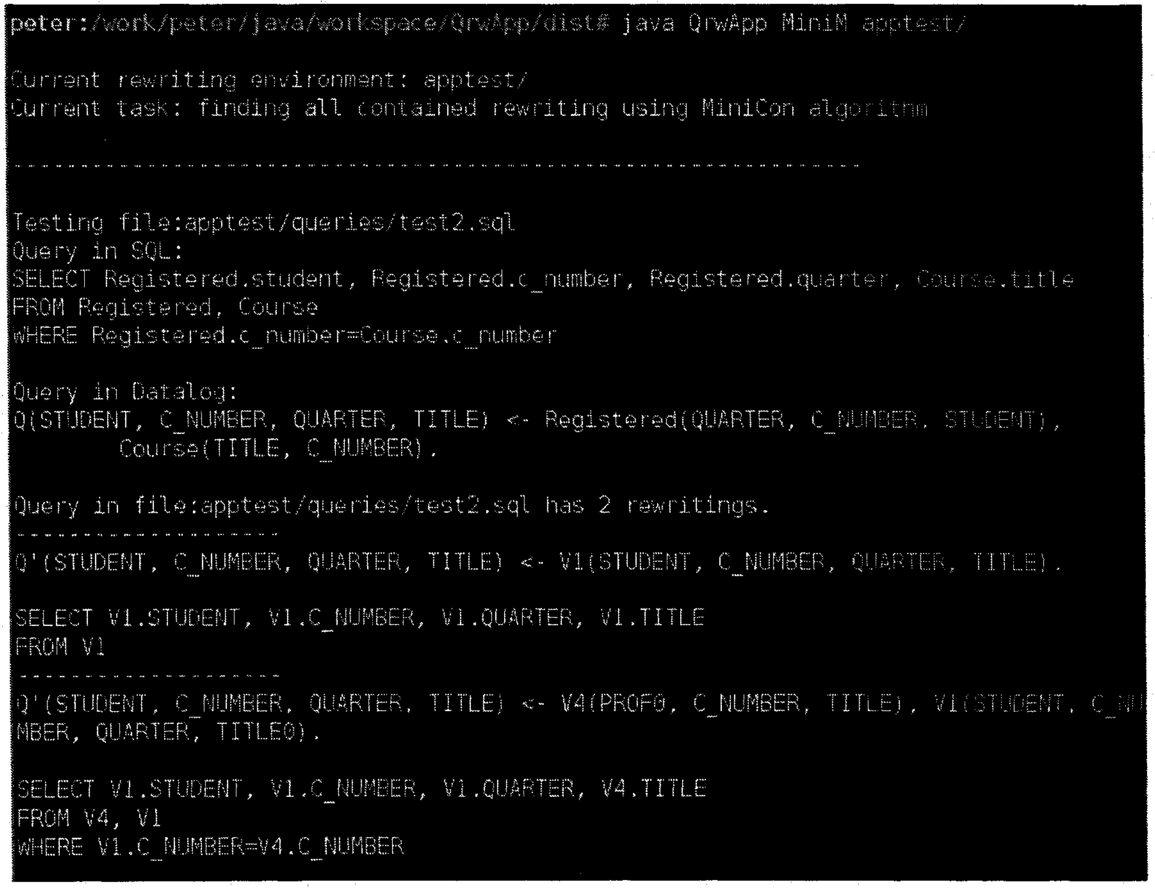


```
-----------------------------------------------------------------

Testing file:apptest/queries/test3.sql
Query in SQL:
SELECT Registered.student, Registered.c_number, Teaches.prof
FROM Teaches, Registered, Course
WHERE Teaches.c_number=Registered.c_number AND Registered.quarter = Teaches.quarter
        AND Registered.c_number=Course.c_number

Query in Datalog:
Q(STUDENT, C_NUMBER, PROF) <- Teaches(EVALUATION, PROF, QUARTER, C_NUMBER),
        Registered(QUARTER, C_NUMBER, STUDENT),
        Course(TITLE, C_NUMBER).

Query in file:apptest/queries/test3.sql has 2 rewritings.
--------------------
Q'(STUDENT, C_NUMBER, PROF) <- V2(STUDENT, PROF, C_NUMBER, EVALUATION),
        V1(STUDENT0, C_NUMBER, QUARTER0, TITLE).

SELECT V2.STUDENT, V2.C_NUMBER, V2.PROF
FROM V2, V1
WHERE V2.C_NUMBER=V1.C_NUMBER
--------------------
Q'(STUDENT, C_NUMBER, PROF) <- V2(STUDENT, PROF, C_NUMBER, EVALUATION),
        V4(PROF0, C_NUMBER, TITLE).

SELECT V2.STUDENT, V2.C_NUMBER, V2.PROF
FROM V2, V4
WHERE V2.C_NUMBER=V4.C_NUMBER
```

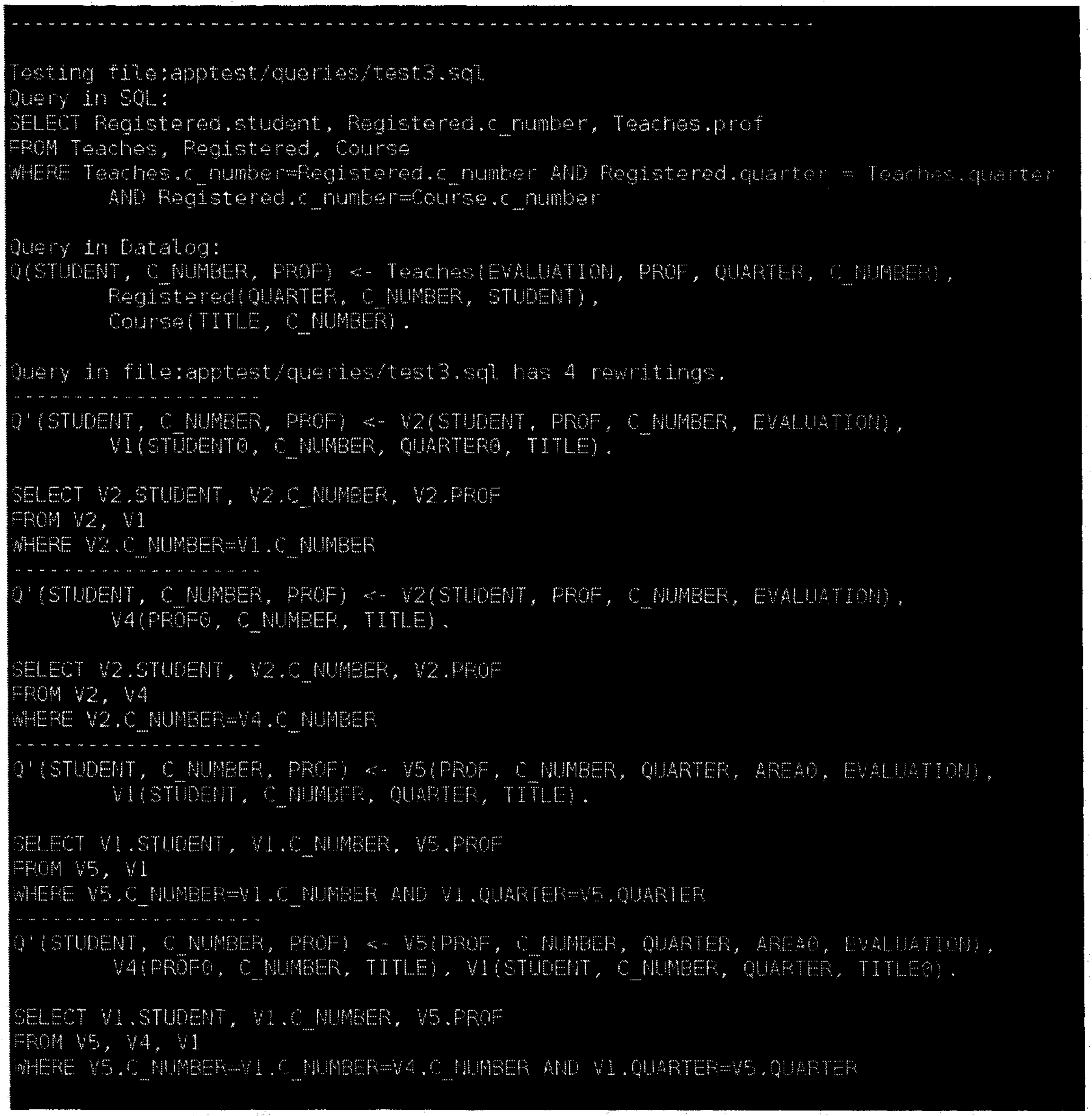


```
Test Summary
----------------------------------------------------------------
Total Number of Views: 5
Number of views that success translate into Datalog: 5

Total Number of Query: 2
Number of Query that success translate into Datalog: 2

Number of Query that has no rewriting: 0
Number of Query that has 1 rewriting: 1
Number of Query that has more than 1 rewriting: 1
Total Number of found rewritings: 3

Total time for query parseing(ms): 17
Total time for query rewriting(ms): 65

peter:/work/peter/java/workspace/QrwApp/dist# ~
```

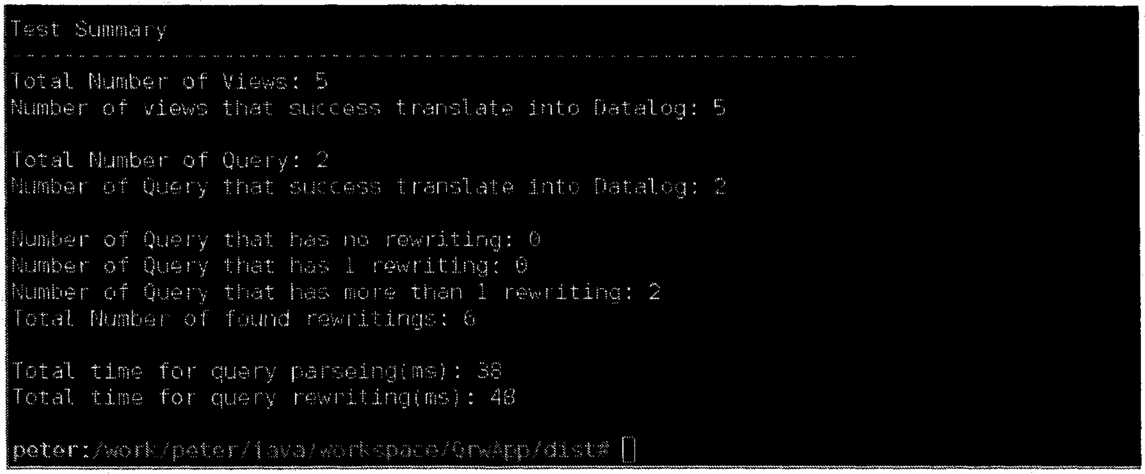
- **Result of running QrwApp using MiniCon**



```
peter:/work/peter/java/workspace/QrwApp/dist# java QrwApp MiniM apptest/

Current rewriting environment: apptest/
Current task: finding all contained rewriting using MiniCon algorithm

----------------------------------------------------------------


Testing file:apptest/queries/test2.sql
Query in SQL:
SELECT Registered.student, Registered.c_number, Registered.quarter, Course.title
FROM Registered, Course
WHERE Registered.c_number=Course.c_number

Query in Datalog:
Q(STUDENT, C_NUMBER, QUARTER, TITLE) <- Registered(QUARTER, C_NUMBER, STUDENT),
        Course(TITLE, C_NUMBER).

Query in file:apptest/queries/test2.sql has 2 rewritings.
--------------------
Q'(STUDENT, C_NUMBER, QUARTER, TITLE) <- V1(STUDENT, C_NUMBER, QUARTER, TITLE).

SELECT V1.STUDENT, V1.C_NUMBER, V1.QUARTER, V1.TITLE
FROM V1
--------------------
Q'(STUDENT, C_NUMBER, QUARTER, TITLE) <- V4(PROF0, C_NUMBER, TITLE), V1(STUDENT, C_NU
MBER, QUARTER, TITLE0).

SELECT V1.STUDENT, V1.C_NUMBER, V1.QUARTER, V4.TITLE
FROM V4, V1
WHERE V1.C_NUMBER=V4.C_NUMBER
```



```
----------------------------------------------------------------

Testing file:apptest/queries/test3.sql
Query in SQL:
SELECT Registered.student, Registered.c_number, Teaches.prof
FROM Teaches, Registered, Course
WHERE Teaches.c_number=Registered.c_number AND Registered.quarter = Teaches.quarter
        AND Registered.c_number=Course.c_number

Query in Datalog:
Q(STUDENT, C_NUMBER, PROF) <- Teaches(EVALUATION, PROF, QUARTER, C_NUMBER),
        Registered(QUARTER, C_NUMBER, STUDENT),
        Course(TITLE, C_NUMBER).

Query in file:apptest/queries/test3.sql has 4 rewritings.
--------------------
Q'(STUDENT, C_NUMBER, PROF) <- V2(STUDENT, PROF, C_NUMBER, EVALUATION),
        V1(STUDENT0, C_NUMBER, QUARTER0, TITLE).

SELECT V2.STUDENT, V2.C_NUMBER, V2.PROF
FROM V2, V1
WHERE V2.C_NUMBER=V1.C_NUMBER
--------------------
Q'(STUDENT, C_NUMBER, PROF) <- V2(STUDENT, PROF, C_NUMBER, EVALUATION),
        V4(PROF0, C_NUMBER, TITLE).

SELECT V2.STUDENT, V2.C_NUMBER, V2.PROF
FROM V2, V4
WHERE V2.C_NUMBER=V4.C_NUMBER
--------------------
Q'(STUDENT, C_NUMBER, PROF) <- V5(PROF, C_NUMBER, QUARTER, AREA0, EVALUATION),
        V1(STUDENT, C_NUMBER, QUARTER, TITLE).

SELECT V1.STUDENT, V1.C_NUMBER, V5.PROF
FROM V5, V1
WHERE V5.C_NUMBER=V1.C_NUMBER AND V1.QUARTER=V5.QUARTER
--------------------
Q'(STUDENT, C_NUMBER, PROF) <- V5(PROF, C_NUMBER, QUARTER, AREA0, EVALUATION),
        V4(PROF0, C_NUMBER, TITLE), V1(STUDENT, C_NUMBER, QUARTER, TITLE0).

SELECT V1.STUDENT, V1.C_NUMBER, V5.PROF
FROM V5, V4, V1
WHERE V5.C_NUMBER=V1.C_NUMBER=V4.C_NUMBER AND V1.QUARTER=V5.QUARTER
```

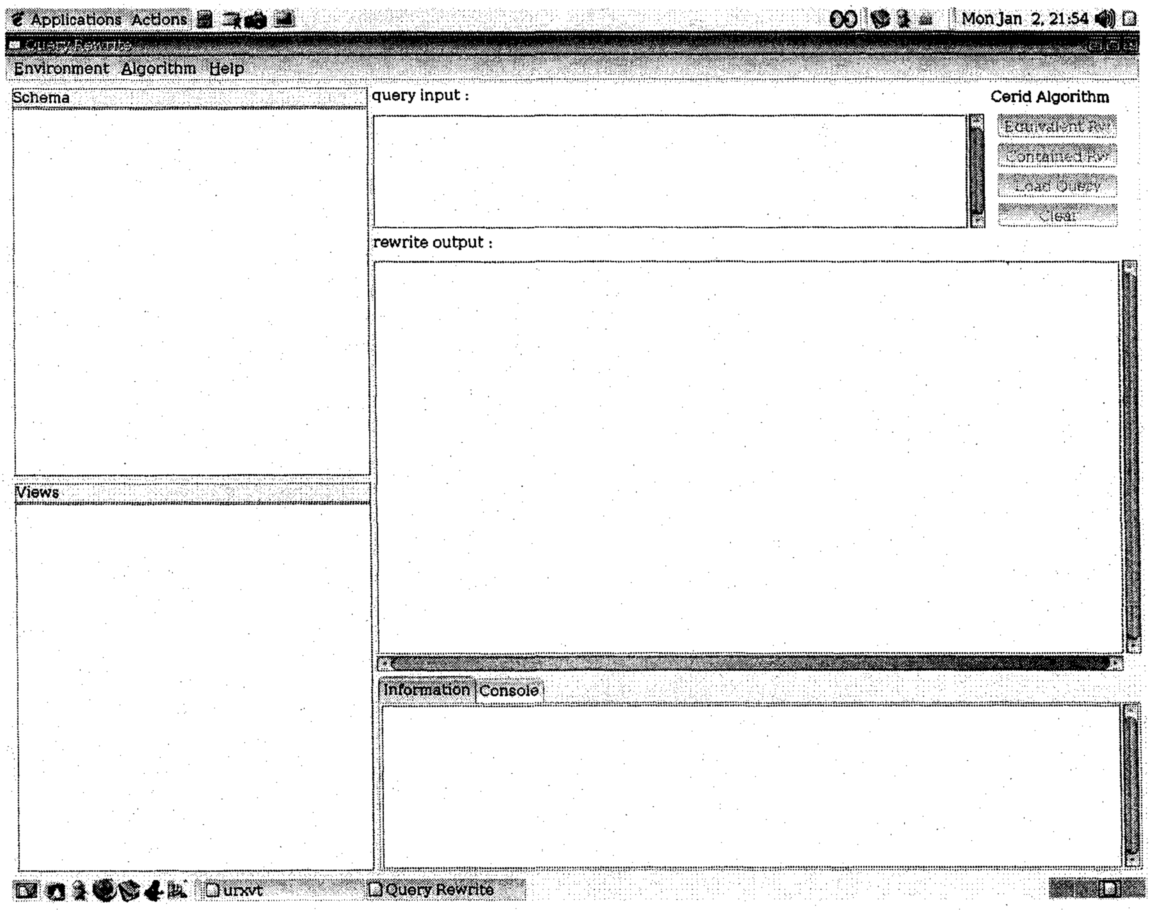


```
Test Summary
----------------------------------------------------------------
Total Number of Views: 5
Number of views that success translate into Datalog: 5

Total Number of Query: 2
Number of Query that success translate into Datalog: 2

Number of Query that has no rewriting: 0
Number of Query that has 1 rewriting: 0
Number of Query that has more than 1 rewriting: 2
Total Number of found rewritings: 6

Total time for query parseing(ms): 38
Total time for query rewriting(ms): 48

peter:/work/peter/java/workspace/QrwApp/dist# 
```

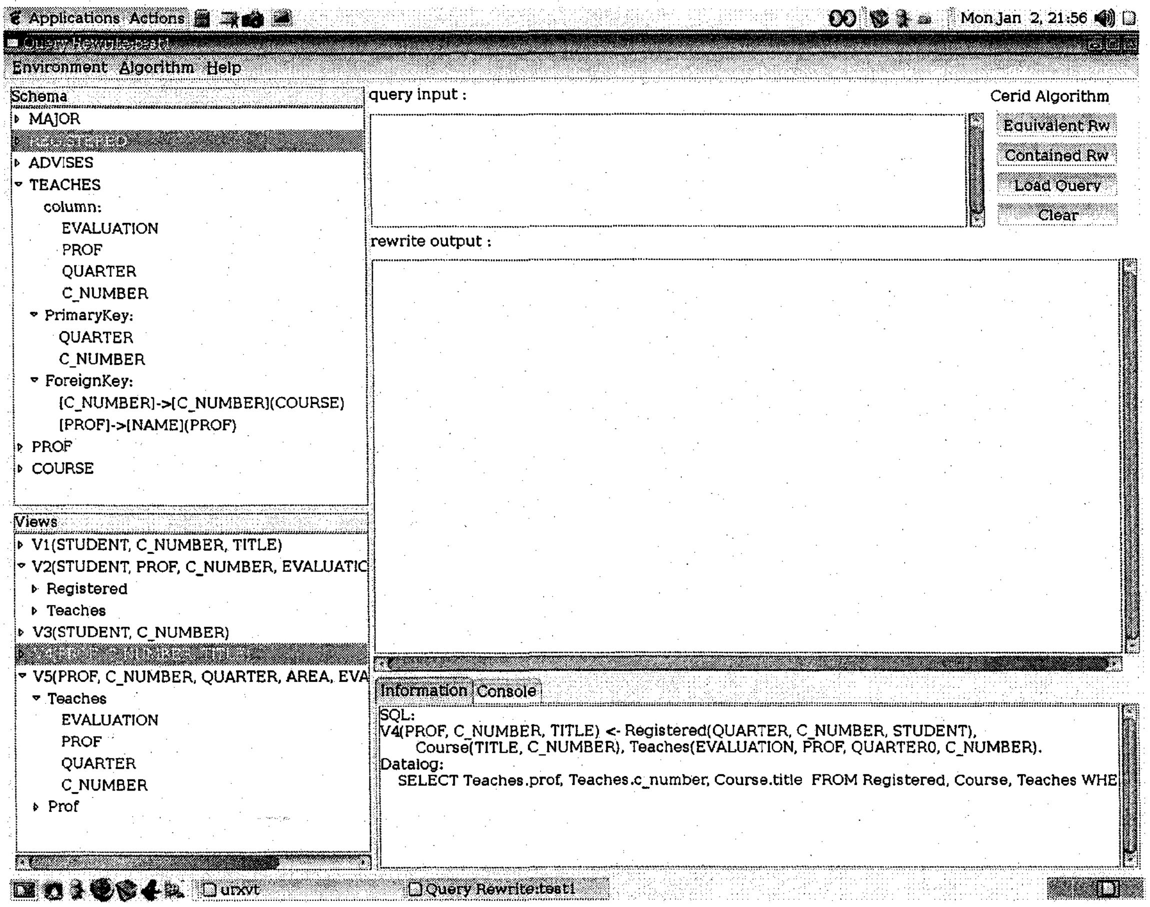
Appendix C Demonstration of QrwGUI

QrwGUI is GUI program to rewriting a single query using different rewriting algorithms. Here are some screenshots for QrwGUI.

1. QrwGui startup



2. QrwGui loads rewriting environment files

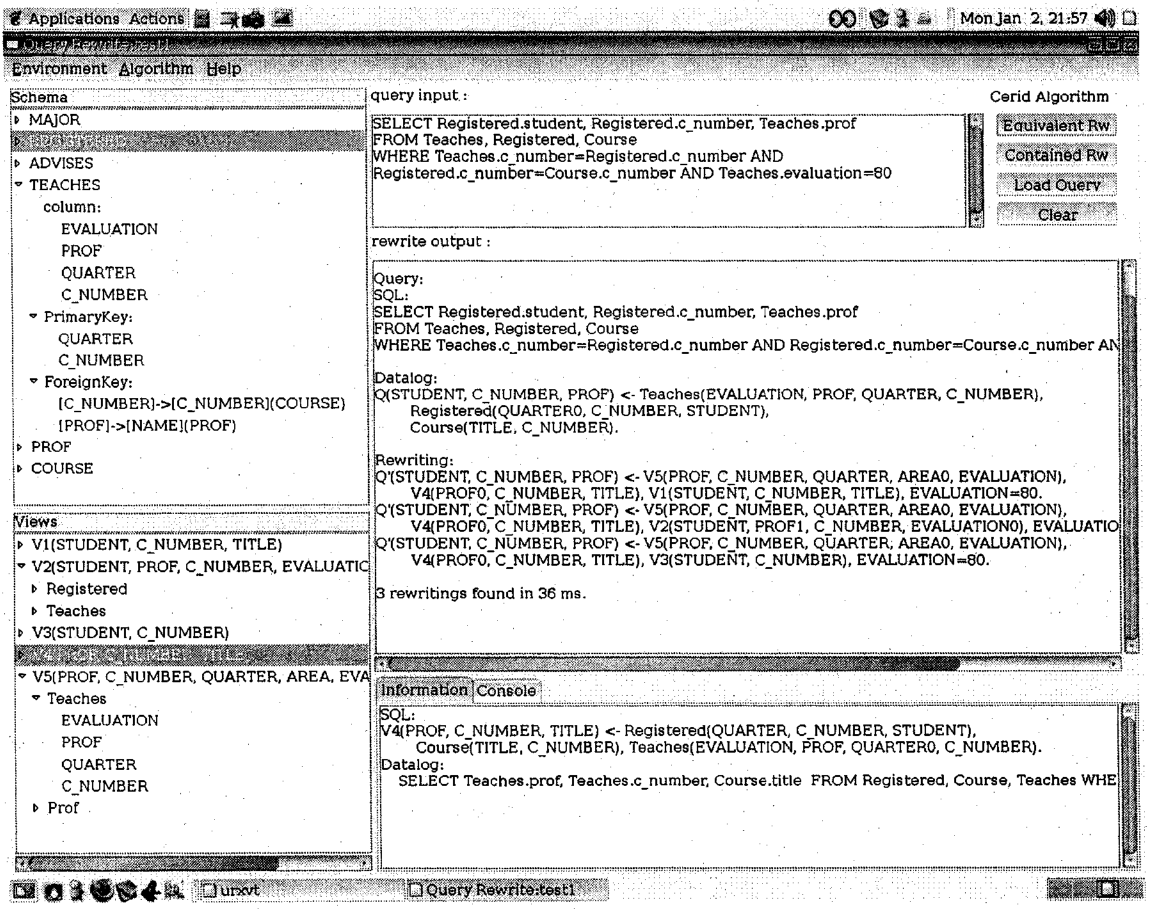


```
Schema
▹ MAJOR
▹ REGISTERED
▹ ADVISES
▿ TEACHES
    column:
      EVALUATION
      PROF
      QUARTER
      C_NUMBER
  ▿ PrimaryKey:
      QUARTER
      C_NUMBER
  ▿ ForeignKey:
      [C_NUMBER]->[C_NUMBER](COURSE)
      [PROF]->[NAME](PROF)
▹ PROF
▹ COURSE
```

```
Views
▹ V1(STUDENT, C_NUMBER, TITLE)
▿ V2(STUDENT, PROF, C_NUMBER, EVALUATIC
  ▹ Registered
  ▹ Teaches
▹ V3(STUDENT, C_NUMBER)
▹ V4(PROF, C_NUMBER, TITLE)
▿ V5(PROF, C_NUMBER, QUARTER, AREA, EVA
  ▿ Teaches
      EVALUATION
      PROF
      QUARTER
      C_NUMBER
  ▹ Prof
```

```
SQL:
V4(PROF, C_NUMBER, TITLE) <- Registered(QUARTER, C_NUMBER, STUDENT),
     Course(TITLE, C_NUMBER), Teaches(EVALUATION, PROF, QUARTER0, C_NUMBER).
Datalog:
  SELECT Teaches.prof, Teaches.c_number, Course.title  FROM Registered, Course, Teaches WHE
```

3. QrwGui shows rewriting resul



Schema

```
▸ MAJOR
▸ ADVISES
▾ TEACHES
    column:
      EVALUATION
      PROF
      QUARTER
      C_NUMBER
  ▾ PrimaryKey:
      QUARTER
      C_NUMBER
  ▾ ForeignKey:
      [C_NUMBER]->[C_NUMBER](COURSE)
      [PROF]->[NAME](PROF)
▸ PROF
▸ COURSE
```

Views

```
▸ V1(STUDENT, C_NUMBER, TITLE)
▾ V2(STUDENT, PROF, C_NUMBER, EVALUATIC
  ▸ Registered
  ▸ Teaches
▸ V3(STUDENT, C_NUMBER)
▾ V5(PROF, C_NUMBER, QUARTER, AREA, EVA
  ▾ Teaches
      EVALUATION
      PROF
      QUARTER
      C_NUMBER
  ▸ Prof
```

query input :

```
SELECT Registered.student, Registered.c_number, Teaches.prof
FROM Teaches, Registered, Course
WHERE Teaches.c_number=Registered.c_number AND
Registered.c_number=Course.c_number AND Teaches.evaluation=80
```

rewrite output :

```
Query:
SQL:
SELECT Registered.student, Registered.c_number, Teaches.prof
FROM Teaches, Registered, Course
WHERE Teaches.c_number=Registered.c_number AND Registered.c_number=Course.c_number AN

Datalog:
Q(STUDENT, C_NUMBER, PROF) <- Teaches(EVALUATION, PROF, QUARTER, C_NUMBER),
      Registered(QUARTER0, C_NUMBER, STUDENT),
      Course(TITLE, C_NUMBER).

Rewriting:
Q'(STUDENT, C_NUMBER, PROF) <- V5(PROF, C_NUMBER, QUARTER, AREA0, EVALUATION),
      V4(PROF0, C_NUMBER, TITLE), V1(STUDENT, C_NUMBER, TITLE), EVALUATION=80.
Q'(STUDENT, C_NUMBER, PROF) <- V5(PROF, C_NUMBER, QUARTER, AREA0, EVALUATION),
      V4(PROF0, C_NUMBER, TITLE), V2(STUDENT, PROF1, C_NUMBER, EVALUATION0), EVALUATIO
Q'(STUDENT, C_NUMBER, PROF) <- V5(PROF, C_NUMBER, QUARTER, AREA0, EVALUATION),
      V4(PROF0, C_NUMBER, TITLE), V3(STUDENT, C_NUMBER), EVALUATION=80.

3 rewritings found in 36 ms.
```

```
SQL:
V4(PROF, C_NUMBER, TITLE) <- Registered(QUARTER, C_NUMBER, STUDENT),
      Course(TITLE, C_NUMBER), Teaches(EVALUATION, PROF, QUARTER0, C_NUMBER).
Datalog:
  SELECT Teaches.prof, Teaches.c_number, Course.title  FROM Registered, Course, Teaches WHE
```

Bibliography

[1] Alon Y. Halevy. Answering queries using views: A survey. The VLDB Journal, 10(4):270–294, 2001.

[2] Xiaolei Qian. Query folding. In Proceedings of the 12th International Conference on Data Engineering(ICDE), pages 48–55, 1996.

[3] Surajit Chaudhuri, Ravi Krishnamurthy, Spyros Potamianos, and Kyuseak Shim. Optimizing queries with materialized views. In 11th Int. Conference on Data Engineering, 1995.

[4] Daniela Florescu, Alon Y. Levy, Dan Suciu, and Khaled Yagoub. Optimization of run-time management of data intensive web-sites. In VLDB '99: Proceedings of the 25th International Conference on Very Large Data Bases, pages 627–638, 1999.

[5] Alon Y. Levy, Anand Rajaraman, and Joann J. Ordille. Querying heterogeneous information sources using source descriptions. In Proceedings of the Twentysecond VLDB Conference, pages 251–262, 1996.

[6] Hector Garcia-Molina, Yannis Papakonstantinou, Dallan Quass, Anand Rajaraman, Yehoshua Sagiv, Je_re Ullman, and Jennifer Widom. The tsimmis project: integration of heterogeneous information sources. Intelligent Information Systems, 8(2):117–132, 1997.

[7] Chung T. Kwok and Daniel S. Weld. Planning to gather information. In 13th AAAI National Conf. on Artificial Intelligence, pages 32–39, 1996.

[8] Eric Lambrecht, Subbarao Kambhampati, and Senthil Gnanaprakasam. Optimizing recursive information-gathering plans. In IJCAI '99: Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence, pages 1204–1211, 1999.

[9] Venky Harinarayan, Anand Rajaraman, and Je_rey D. Ullman. Implementing data cubes e_ciently. In SIGMOD '96: Proceedings of the 1996 ACM SIGMOD international conference on Management of data, pages 205–216, 1996.

[10] Dimitri Theodoratos and Timos K. Sellis. Data warehouse configuration. In VLDB '97: Proceedings of the 23rd International Conference on Very Large Data Bases, pages 126–135, 1997.

[11] Jian Yang, Kamalakar Karlapalem, and Qing Li. Algorithms for materialized view design in data warehousing environment. In VLDB '97: Proceedings of the 23rd International Conference on Very Large Data Bases, pages 136–145, 1997.

[12] Oliver M. Duschka and Michael R. Genesereth. Query planning in infomaster. In SAC '97: Proceedings of the 1997 ACM symposium on Applied computing, pages 109–111, 1997.

[13] Yigal Arens, Chin Y. Chee, Chun-Nan Hsu, and Craig A. Knoblock. Retrieving and integrating data from multiple information sources. International Journal of Cooperative Information Systems, 2(2):127–158, 1993.

[14] Rachel Pottinger and Alon Halevy. Minicon: A scalable algorithm for answering queries using views. The VLDB Journal, 10(2-3):182–198, 2001.

[15] Prasenjit Mitra. An algorithm for answering queries e_ciently using views. In ADC '01: Proceedings of the 12th Australasian conference on Database technologies, pages 99–106, 2001.

[16] Foto N. Afrati, Chen Li, and Je_rey D. Ullman. Generating e_cient plans for queries using views. In SIGMOD '01: Proceedings of the 2001 ACM SIGMOD international conference on Management of data, pages 319–330, 2001.

[17] Oliver M. Duschka and Michael R. Genesereth. Answering recursive queries using views. In PODS '97: Proceedings of the 16th ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems, pages 109–116, 1997.

[18] Je_rey D. Ullman. Principles of Database and Knowledge-Base Systems: Volume II: The New Technologies. W. H. Freeman & Co., New York, NY, USA, 1990.

[19] Junhu Wang, Michael Maher, and Rodney Topor. Rewriting general conjunctive queries using views. In CRPITS '02: Proceedings of the thirteenth Australasian conference on Database technologies, pages 197–206, 2002.

[20] Ashok K. Chandra and Philip M. Merlin. Optimal implementation of conjunctive queries in relational data bases. In Proceedings of the ninth annual ACM symposium on Theory of computing(STOC), pages 77–90, 1977.

[21] O. Shmueli. Decidability and expressiveness aspects of logic queries. In PODS '87: Proceedings of the sixth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems, pages 237–249, 1987.

[22] Oliver M. Duschka and Alon Y. Levy. Recursive plans for information gathering. In 15th International Joint Conference on Artificial Intelligence, pages 778–784, Nagoya, Japan, 1997.

[23] Surajit Chaudhuri, Ravi Krishnamurthy, Spyros Potamianos, and Kyuseak Shim. Optimizing queries with materialized views. In 11th Int. Conference on Data Engineering, pages 190–200, Los Alamitos, CA, 1995. IEEE Computer Soc. Press.

[24] Markos Zaharioudakis, Roberta Cochrane, George Lapis, Hamid Pirahesh, and Monica Urata. Answering complex sql queries using automatic summary tables. In SIGMOD '00: Proceedings of the 2000 ACM SIGMOD international conference on Management of data, pages 105–116, 2000.

[25] Terry Lau, Jianguo Lu, John Mylopoulos, Kostas Kontogiannis, Migrating E-commerce Database Applications to an Enterprise Java Environment, Information Systems Frontiers, Vol. 5, No. 2 (2003), Kluwer Academic Publishers.

[26] Jianguo Lu, Reengineering Database Applications to EJB based architecture, CAiSE'02, 14th Conference on Advanced Information Systems Engineering, 2002.

[27] A SQL Parser written in Java, http://www.cs.toronto.edu/~jglu/sql4j/index.htm, 2003.

[28] Kenneth Cheung. A query rewriting system: implementation and applications. Master Thesis, University of Toronto, 2003.

[29] Chen Li, Foto Afrati, and Prasenjit Mitra. Answering queries using views with arithmetic comparisons. In PODS '02: Proceedings of the twenty-first ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems, pages 209–220, 2002.

[30] Anthony Klug. On conjunctive queries containing inequalities. J. ACM, 35(1):146–160, 1988.

[31] Phokion G. Kolaitis, David L. Martin, and Madhukar N. Thakur. On the complexity of the containment problem for conjunctive queries with built-in predicates. In PODS '98: Proceedings of the seventeenth ACM SIGACT-SIGMODSIGART symposium on Principles of database systems, pages 197–204, 1998.

[32] Alon Y. Levy and Yehoshua Sagiv. Queries independent of updates. In Proceedings of 19th VLDB Conference, pages 171–181, 1993.

[33] Jarek Gryz. Query rewriting using views in the presence of functional and inclusion dependencies. Inf. Syst., 24(7):597–612, 1999.

[34] Qingyuan Bai, Jun Hong, and Michael F. McTear. Query rewriting using views in the presence of inclusion dependencies. In WIDM '03: Proceedings of the 5th

ACM international workshop on Web information and data management, pages 134–138, 2003.

[35] Christoph Koch. Query rewriting with symmetric constraints. In FoIKS '02: Proceedings of the Second International Symposium on Foundations of Information and Knowledge Systems, pages 130–147, 2002.

Vita Auctoris

NAME:	Minghao Li
PLACE OF BIRTH:	Shaan Xi Province, China
YEAR OF BIRTH:	1977
EDUCATION:	The second High School, Zhengzhou, China
	1993-1996
	Renmin University of China, Beijing, China
	1996-2000 B.Sc
	University of Windsor, Windsor, Ontario
	2003-2007 M.Sc