EFFICIENT AND SCALABLE TECHNIQUES FOR MINIMIZATION AND REWRITING OF CONJUNCTIVE QUERIES

Ali Kiani Tallaei

A thesis

in

The Department

of

Computer Science and Software Engineering

Presented in Partial Fulfillment of the Requirements

For the Degree of Doctor of Philosophy

Concordia University

Montréal, Québec, Canada

August 2010

Your file Votre référence
ISBN: 978-0-494-71139-2
Our file Notre référence
ISBN: 978-0-494-71139-2

Abstract

Efficient and Scalable Techniques for Minimization and Rewriting of Conjunctive Queries

Ali Kiani Tallaei, Ph.D.

Concordia University, 2010

Query rewriting as an approach to query answering has been a challenging issue in database and information integration systems. In general, rewriting of a conjunctive query Q using a set of views in conjunctive form consists of two phases: (1) generating proper building blocks using the views, and (2) combining them to generate a union of conjunctive queries which is maximally contained in Q. While the problem of query rewriting is known to be exponential in the number of subgoals of Q, there is a demand for increased efficiency for practical queries. We revisit this problem for conjunctive queries, and show that Stirling numbers can be used to determine the optimal number of combinations in the second phase, and hence the number of rules in the generated union of conjunctive queries. Based on these numbers, we introduce the notion of combination patterns and develop a rewriting algorithm that uses these numeral patterns to break down the large combinatorial problem in the second phase into several smaller ones. The results of our numerous experiments indicate that the proposed rewriting technique outperforms existing techniques including Minicon

algorithm in terms of computation time, memory requirements, and scalability.

On a related context, we studied query minimization, motivated by the fact that queries with fewer or no redundant subgoals can be evaluated faster, in general. However, such redundancies are often present in automatically generated queries. We propose an algorithm that, given a conjunctive query, repeatedly identifies and eliminates all the redundant subgoals. We also illustrate its performance superiority over existing minimization algorithms.

It has been shown that query rewriting naturally generates queries with redundant subgoals. We also show that redundant subgoals in the input of query rewriting result in redundant rules in its output. Based on this, we investigate the impact of minimization as pre-processing and post-processing phases to query rewriting technique. Our experimental results using different synthetic data show that our query rewriting technique coupled with pre/post minimization phases produces the best quality of rewriting in a more efficient way compared to existing rewriting techniques, including the Treewise algorithm.

It has been shown that extending conjunctive queries with constraints adds to the complexity of query rewriting. Previous studies identified classes of conjunctive queries with constraints in the form of arithmetic comparisons for which the complexity of rewriting does not change. Such classes are said to satisfy homomorphism property. We identify new classes of conjunctive queries with linear arithmetic constraints that enjoy this property, and extend our query rewriting algorithm accordingly to support such queries.

Dedicated to my wife

Effat

Acknowledgments

I would like to express my sincere gratitude to my supervisor, Dr. Shiri, whose encouragement, dedication and constructive criticism has been a great motivation for me. Without his helps, I would not be able to finish this thesis. I am also thankful to him for the teaching opportunities and access to the database laboratory equipped with the latest technologies.

I am grateful to my teachers at the department of Computer Science and Software Engineering at Concordia University, specially Dr. Grahne and Dr. Haarslev for their constant supports and helpful discussions.

I owe many thanks to my committee members Dr. Bertossi, Dr. Al-Khalili, Dr. Haarslev, Dr. Grahne, and Dr. Shiri for their detailed reviews and excellent advices during the preparation of this thesis.

My Special thanks to my friend, Shirley Pai for discussions, suggestions, interesting ideas and also helps during preparation of this thesis.

I am grateful to my colleagues at the database lab at Concordia university, Mihail Halachev. Xi Deng, Nima Mohajerin, Ahmed Alasoud, Alex Thomo, Weisheng Lin, Zheng Zhi Hong, Ali Taghizadeh, Vidya Kadiyala and Keywan Hodaie.

Last but not least, I would like to thank Halina Monkiewicz, Pauline Dubois, Hirut Adugna, Edwina Bowen, Stan Swiercz, and the staff at department of Computer Science and Software Engineering at Concordia University.

Contents

List of Figures

List of Abbreviations and Symbols

AC Queries	Conjunctive Queries with Arithmetic Comparison
$B(n)$	Bell number of n, the number of partitions of a set of n elements
CLAC	Conjunctive Queries with Linear Arithmetic Constraints
CLSI	Closed Left Semi-Interval Comparisons
CQ	Conjunctive Queries
CQEC	Conjunctive Queries with Equality Constraints
CQEC+AC	Conjunctive Queries with Equality Constraint and Arithmetic Comparison
CQEC+LSI	Conjunctive Queries with Equality Constraint and LSI Comparison
CQEC+RSI	Conjunctive Queries with Equality Constraint and RSI Comparison
LSI	Left Semi-Interval
MCR	Maximally Contained Rewriting
OLSI	Open Left Semi-Interval
PI	Point-Inequalities
QC	Query Containment
QR	Query Rewriting
RSI	Right Semi-Interval

$S(n,k)$	The Stirling numbers of the second kind, the number of ways to partition a set of n elements into k nonempty subsets
SI	Semi-Interval
UCQ	Union of Conjunctive Queries

Chapter 1

Introduction

In this chapter, we motivate the problem of *Query Rewriting* in *data integration* as a technique for answering *Conjunctive Queries*. We then explain how this is related to the problem of *Query Minimization* and motivate investigating *Query Minimization*. We also recall the importance of *Constraints* in real life applications which motivates investigating efficient rewriting of *Conjunctive Queries with Constraints*.

1.1 Overview

Databases are essential part of many information systems. In general, a database records information about a specific application domain using a fixed representation of the concepts it models, called *schema*. A schema is a set of relations each of which is set of attributes that represents a concept (e.g., Department and Course in the domain of a university), or relationship between the concepts (e.g., relationship

between Department and Course).

Initially, querying databases was done locally. Also, communication between databases was difficult, mostly because it required a lot of manual work. Advances in hardware technologies and the falling cost of hardware made database solutions more attractive and affordable. This resulted in creation of many databases that have overlapping domains, i.e., domains that have common concepts, explained in the following example.

Example 1.1 *In the domain of a university, there are different units involved, including Office of the Registrar (OR), School of Graduate Studies (SGS), Human Resources (HR), department of Computer Science (CS), and International Student Office (ISO), each have their own database where some concepts are common. For instance, an international student who is enrolled in the PhD program in the department of Computer Science and works as a teacher assistant in that department would have records in the database of each of the above units. In fact, each database would present this student from a different point of view.*

We refer to each of these databases as an *Information Source.* The availability of information sources with overlapping domains motivates the idea of integrating the information sources. To see this, consider the information sources in Example 1.1, and the query "List all international students enrolled in a Computer Science program." While this query cannot be answered by the ISO database nor by the CS database alone, it can be answered by the integration of these two information sources.

For this, assume a new schema called ICSS (International Computer Science Student) is created to "integrate" the concept of student from ISO and CS. The relation(s) in ICSS represents a computer science student who is also an international student. Now, it is possible to query this integrated source to answer the above query.

The two major approaches to data integration are *Data Warehouses* and *Mediator-based Information Integration.* In this thesis, we only consider the mediator-based approach to integration. Data warehouses have been extensively studied from different point of views including design, optimization and maintenance [Inm96, JLVV03, CDL+01, GM05, TS97], and automation of schema management [BR00, MRB03, Ber01, PTU00]. Next, we review mediator-based information integration.

1.2 Mediator-based Information Integration

As shown in Figure 1.1, there are three layers in the mediator-based integration: the user, the mediator layer, and the information sources. The mediator includes *Metadata* and query processor. Metadata contains information about (1) the schema of integration (known as *global schema*), (2) the schemas of information sources (known as *local schemas*), and (3) the mappings between the local schemas and the global schema.

In mediator-based integration, the user query is based on the global schema, but since the relations in the global schema do not contain any physical data, the query cannot be executed at this level. The query processor in the mediator is responsible for

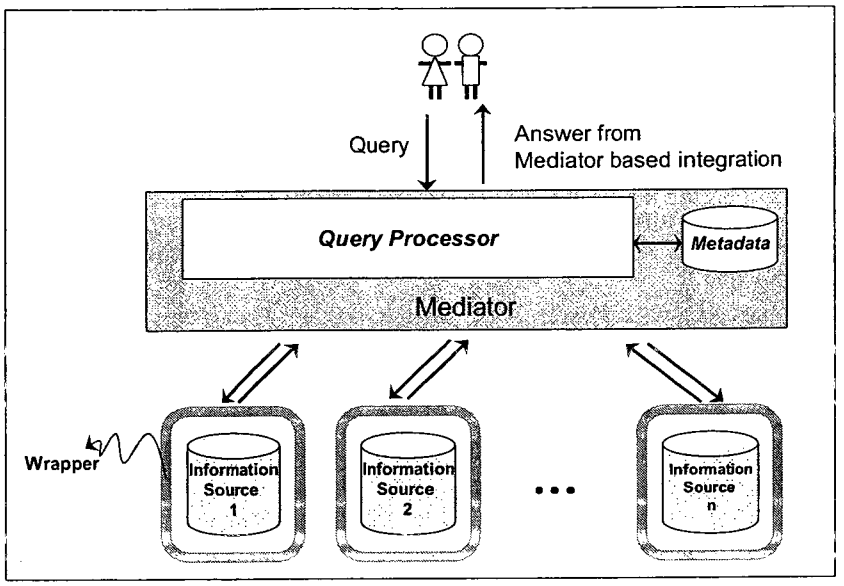

Figure 1.1: Architecture of Mediator-based System

exploiting a proper technique and using the available information sources to answer the query.

Next, we review query processing techniques for mediator-based integration.

1.3 Query Processing in Mediator-based Integration

In general, in mediator-based integration systems, there are two main approaches to query processing and finding answers to queries: (1) *Query Rewriting*, which uses view definitions to rewrite the input query expressed over global schema in terms of local schemas (relations in information sources), and then evaluates the rewriting over the source extensions. (2) *Query Answering Using Views*, which uses both view

definitions and extensions (i.e., relation instances) to find the query answers [AD98].

In this thesis, we focus on query rewriting as an approach to processing of conjunctive queries. Query rewriting has been the subject of numerous studies and several solutions have been proposed, namely Bucket algorithm [LRO96], Inverse-rules [DL00], MiniCon [PL00] and Treewise [MS08].

Query rewriting, in general, consists of two phases: (1) creating building blocks and (2) combining them to generate rewriting. In the first phase, views are analyzed to find out the subgoals they can be replaced with, and in the second phase, view heads are combined to generate queries that are contained in the input query [LMSS95].

We note that the second phase, which becomes a combinatorial problem, is exponential in the number of subgoals in the input query. Despite this, it seems that the above rewriting algorithms focus more on the first phase of rewriting. Considering the complexity of the second phase, it is important to investigate the second phase as well, as it affects the efficiency and scalability of the rewriting. Next, we explain the problem we investigate in this thesis.

1.4 Problem Statement

In this thesis, we investigate *performance and scalability* of query rewriting as an approach to query processing in the context of conjunctive queries. For that, we consider the following problems.

1. **Performance and Scalability of Query Rewriting**

   In the context of mediator-based integration, since the number of possible queries is not limited and information sources could leave and join dynamically, query rewriting cannot be done a priori. Thus, performance and scalability of query rewriting become important issues in practice. Efficient rewriting of conjunctive queries has been the subject of several studies and interesting solution techniques, including the Bucket algorithm [LRO96], Inverse-rules [DL00, Qia96, DG97a], MiniCon [PL00], and Treewise [MS08] have been proposed. [Lev01] provides a survey of these techniques. We revisit these solutions and obtain further improvement in terms of performance and scalability. More specifically, we investigate the combinatorial problem in the second phase of query rewriting for achieving an improved performance and scalability compared to the existing solutions.

2. **Redundancies in Queries**

   When the input query and views are expressed as conjunctive queries, the output rewriting is usually expressed as union of conjunctive queries (i.e., a rewriting is a set of rules each of which is a conjunctive query). We show that if the input query and views contain redundant subgoals, query rewriting generates redundant rules in the output. Also, it has been shown that even if input query and views do not have redundant subgoals, query rewriting usually generates rules that have redundant subgoals [LMSS95, PL00].

Thus, minimization of conjunctive queries [CM77], in general, and in the context of query rewriting, in particular, is an important problem. This problem has been studied [CM77, KS02, LMSS95, PL00] and some good solutions have been proposed. We revisit this problem, and investigate efficient query minimization and also its effect on both input and output of query rewriting.

3. **Efficient Rewriting of Queries with Constraints** Constraints in the form of arithmetic comparison appear frequently in the WHERE clause of SQL queries. It has been shown that the problems of containment and rewriting of queries with arithmetic comparison are more difficult to deal with than the corresponding problems in standard queries (queries with no inequality constraints) [Klu88, ALM04]. Several studies have identified classes of conjunctive queries for which the complexity of containment and rewriting remains the same as in the standard conjunctive queries [ALM04, ALM02, KS10]. Identifying such classes is an important problem in practice. As shown in [ALM02], given an input that contains query and views with constraints, if testing the membership of the input query and views to such classes of queries can be done in polynomial time, then query rewriting can be done more efficiently compared to the general case of queries with constraints.

1.5 Thesis Contributions

The contributions of this thesis are as follows.

1. Query rewriting consists of two phases: finding building blocks for rewriting and combining them to generate rewriting. We show that combining these building blocks is a partitioning problem, based on which we introduce a formula to determine the number combinations and hence, rules in the rewriting before generating it.

2. We propose a novel query rewriting technique for conjunctive queries that uses some numerical patterns to break a large combinatorial problem into several smaller problems and perform query rewriting efficiently. We experimentally show that our proposal outperforms the MiniCon [PL00] and Treewise [MS08] algorithms for different types of queries in terms of performance (two order of magnitude in some cases), scalability, and memory requirement [KS09].

3. Based on the approach used to form the building blocks of rewriting, we classify rewriting algorithms into bottom-up and top-down approaches, and show that the quality of rewriting (i.e., the number of subgoals in the output of the rewriting) in top-down approaches is better than in bottom-up approaches. However, there is an isomorphism between the results of the two approaches. This confirms that minimization can be applied to the output of bottom-up approach to obtain the same quality of a top-down approach.

4. In the context of conjunctive queries, the output of the rewriting is a union of conjunctive queries. It has been shown that usually redundant subgoals could be generated in the rules in the output of the rewriting [PL00]. We show

that if the input queries and views in rewriting are not minimized, not only redundant subgoals but also redundant rules will be generated in the output of the rewriting.

5. We propose a minimization algorithm that identifies the redundant subgoals in conjunctive queries, and experimentally show that our minimization technique is efficient and outperforms existing solutions. We exploit our minimization technique as pre-processing and post-processing phases of query rewriting, and conduct numerous experiments on query rewriting equipped with minimization for different types and sizes of queries. The results show that unlike the assumptions in previous studies, applying query minimization to the output of query rewriting is practical.

   Moreover, minimizing the input of query rewriting improves (in orders of magnitude, for some cases) (1) the performance of query rewriting and (2) the size of the generated rewriting.

6. Following the approach in [ALM04, ALM02] for containment and rewriting of conjunctive queries with constraints, we identify more classes of conjunctive queries with constraints for which the complexity of query containment remains in NP. We extend our pattern-based query rewriting algorithm to support such queries [KS10].

1.6 Thesis Outline

The rest of this thesis is organized as follows. In Chapter 2, we provide the background and definitions required for the discussions in the thesis, and review the existing query rewriting algorithms with the focus on "maximally contained rewriting." In Chapter 3, we introduce a formula in terms of Stirling numbers to determine the number of rules in the rewriting. Using this, we propose our pattern-based rewriting algorithm to generate "maximally contained rewriting" for standard conjunctive queries. Moreover, we define a new measure for the quality of rewriting based on the number of subgoals in the generated rewriting, and show the importance of query minimization on both input and output of query rewriting.

In chapter 4, we investigate query minimization for standard conjunctive queries, and introduce a novel algorithm for query minimization.

In chapter 5, we report the results of our experiments for evaluating the performance of the proposed minimization algorithm using different types and sizes of queries. We conduct another set of experiments to evaluate the performance, scalability and memory requirements of our pattern-based rewriting algorithm, and compare the results with the Minicon and Treewise algorithms. We also study the impact of the proposed query minimization algorithm on the output of the query rewriting algorithm and through numerous experiments, we show that the overhead of applying query minimization on the output of query rewriting is not considerable and this combination outperforms the Minicon and Treewise algorithms in terms of time and

"quality" (number of subgoals in the generated rewriting).

In chapter 6, we consider conjunctive queries with linear arithmetic constraints and identify classes of such queries for which the complexity of containment and rewriting is the same as the standard case (i.e., conjunctive queries with no inequality constraints). Moreover, we extend our query rewriting algorithm to support such queries.

In Chapter 7, we summarize the contributions of this thesis and list the future work.

1.7 Summary

In this chapter, we motivated this thesis by explaining the importance of query rewriting as an approach to query processing in the context of mediator-based integration, and identified a number of problems to be addressed in this thesis. They include performance and scalability of query rewriting, efficient minimization of input and output of query rewriting, and efficient query rewiring for queries with constraints.

Chapter 2

Background

In this chapter, we review the basic definitions and concepts required for the developments and discussions in this thesis. We consider *the Relational Model* for databases, and recall the concepts of *Query*, *View*, and *Conjunctive Query*. Moreover, we review the Containment and Equivalence for conjunctive queries. Finally, we consider query processing in mediator-based integration, and define view-based query processing for *LAV-based* and *GAV-based* integrations, as well as the concepts of *Maximally Contained Rewriting*. We then focus on query processing for LAV-based integrations, and review existing rewriting algorithms in this context.

2.1 Basic Concepts

2.1.1 Relational Model

The relational model, introduced by Codd [Cod70], provides a model for data based on *relations*. Every relation has a *schema* which is a set of n *attributes* each of which is associated with a domain. Each row in a relation is called a *tuple*. Each tuple consists of n values where the i^{th} value is associated with the i^{th} attribute, and is taken from the i^{th} domain. A relation can also be seen as a set of columns, where column i is identified by the i^{th} attribute.

Example 2.1 *The following figure illustrates an instance of relation Student. The schema includes attributes ID, Name, and GPA. The given instance contains three tuples.*

Student

ID	*Name*	*GPA*
S1	*Joe*	*3.4*
S2	*Jack*	*3.5*
S3	*Sally*	*3.5*

As explained earlier, a relation instance in the relational model, is a set of tuples. In our discussion, we do not consider relations with repeated tuples.

2.1.2 Query

One of the main goals of database systems is to answer queries. A query on a database is a request for information expressed in a specific language. When a user or an application poses a query, the database system processes the query, and returns the answer which is usually a collection of tuples. Most database management systems use SQL (Structured Query Language) as the query language.

Example 2.2 *The following SQL query, defined based on the relation in Example 2.1, lists ID and Name of the students whose GPA is 3.5.*

SELECT *ID, Name*

FROM *Student*

WHERE *GPA* = *3.5*

The result over the given instance of Student in Example 2.1 is as follows.

ID	*Name*
S2	*Jack*
S3	*Sally*

The following example shows a query over three relations.

Example 2.3 *Consider the following instances of relations Course and Enrolled as well as the relation Student from Example 2.1.*

Course

ID	*Description*	*Credits*
C1	*Discrete Math I*	*3*
C2	*Discrete Math II*	*3*
C3	*Data Structures*	*3*
C4	*Databases*	*3*
C5	*Object Oriented Programming*	*3*

Enrolled

SID	*CID*
S1	*C1*
S1	*C3*
S2	*C2*
S2	*C3*
S3	*C4*

The following query finds all the students with a GPA of 3.5, and lists pairs of student name and course description.

SELECT *Name, Description*

FROM *Student, Enrolled,Course*

WHERE *Student.ID=Enrolled.SID AND*

Course.ID=Enrolled.CID AND

Student.GPA = 3.5

This query has three clauses: Select, From and Where. A simplified procedure to find the answer to this query is as follows.

Step 1: *Perform a Cartesian product on the relations in the From clause: Student, Course and Enrolled.*

Step 2: *Select only the tuples of the result of step 1 for which the conditions in the*

Where clause are satisfied, and discard the rest of the tuples.

Step 3: *Project columns Name and Description in the Select clause, and discard the rest of the columns.*

Using the above instances of relations Student, Course and Enrolled, the answer to this query includes the following tuples.

Name	*Description*
Jack	*Discrete Math II*
Jack	*Data Structures*
Sally	*Databases*

2.1.3 Conjunctive Queries

While SQL is the de facto standard query language in practice, other languages such as *Conjunctive Queries*, *Relational Algebra*, or *Relational Calculus* [GUW08, SRV95] are used in theoretical developments.

Majority of SQL queries, as shown in Example 2.3, are of the form of Select-From-Where. In this thesis, we focus on conjunctive queries which are logical expressions of the Select-From-Where queries, defined as follows.

Definition 2.1 (Conjunctive Query) *A conjunctive query Q is a statement of the following form:*

$$Q: \quad h(\bar{X}) \text{ :- } g_1(\bar{X}_1), \cdots, g_k(\bar{X}_k), \alpha_1, \cdots, \alpha_n,$$

where $g_i(\bar{X}_i)$ is a subgoal consisting of a predicate name g_i and a list $\bar{X}_i$ of variables and constants. Each predicate g_i refers to a relation in the relational database and

every argument in g_i is either a variable or a constant. Let S be the set of all the variables and constants in Q. Every α_j is a built-in predicate in the form of $L_j\theta_j R_j$, where θ_j is a comparison operator from $\{<, \leq, >, \geq, =, \neq\}$, and L_j and R_j are expressions over S.

Q consists of two parts: (1) query head: $h(\bar{X})$, denoted as $Head(Q)$ and (2) query body: $g_1(\bar{X}_1), \cdots, g_k(\bar{X}_k), \alpha_1, \cdots, \alpha_n$, denoted as $Body(Q)$. We refer to the set of subgoals in $Body(Q)$ as $Core(Q)$, and to the set of built-in predicates as $Constraints(Q)$.

A conjunctive query is *safe*, if every head variable (variables in $\bar{X}$) also appears in a subgoal in the body. In our discussions, we only consider safe queries. Let D be a database instance. We use $Q(D)$ to refer to the extension of Q in D, i.e., the set of valuations for the variables of Q that make Q true in D.

A variable A in a subgoal in $Body(Q)$ is called *distinguished*, if it also appears in the query head. We use $Vars(Q)$ to refer to the variables in Q.

We say a conjunctive query is *standard* if its constraints are limited to the forms A=B or A=c, where A and B are variables from $Vars(Q)$, and c is a constant. Note that as shown in Example 2.4, a constraint in the form of A=B can be expressed using repeated variables in $Core(Q)$. Similarly, constraint A=c can be expressed using occurrence of constant c in $Core(Q)$. Therefore, a standard conjunctive query can be expressed in the following form:

Q : $\quad h(\bar{X})$:- $g_1(\bar{X}_1), \cdots, g_k(\bar{X}_k)$

where $g_i(\bar{X}_i)$ is a subgoal consisting of a predicate name g_i and a list $\bar{X}_i$ of variables

and constants.

Every Select-From-Where SQL query can be expressed as a conjunctive query, explained as follows. The Select clause in a SQL query corresponds to the query head in a conjunctive query, the From clause corresponds to the subgoals in the query body, and the Where clause is expressed by built-in predicates.

Example 2.4 *The query in Example 2.3 can be expressed as a conjunctive query as follows.*

$Q: \quad h(B, E) \text{ :- } Student(A, B, 3.5), Course(D, E), Enrolled(A, D).$

Here, variable A has appeared in $Student(A, B, 3.5)$ *and also in the subgoal* $Enrolled(A, D)$*, enforcing that the ID from Student matches with SID in Enrolled. Similarly, variable D has appeared in subgoals* $Course(C, D)$ *and* $Enrolled(A, D)$*, enforcing that the ID from Course matches with CID in Enrolled. Also, the constant 3.5 in subgoal* $Student(A, B, 3.5)$ *enforces that only students with a GPA of 3.5 are selected. Finally, variables B and E that correspond to student name and course description are projected in the query head.*

Throughout this thesis, we use small letters such as r, s, and t for relations, and use capital letters such as X, Y, and Z for variables.

2.1.4 View

A *view* in the relational model is a stored query which provides a higher level of abstraction over relations. A view can be used in queries similar to relations and for this reason, it is also called a virtual relation. Every time a view is queried, its stored query is executed and returns a set of tuples, called the *view extension* or *view instance.*

The following example defines a view V, and shows a query based on it.

Example 2.5 *Consider the relations Student, Course and Enrolled defined earlier. The following conjunctive query defines a view for the same pairs of student and course in Example 2.4.*

V : $v(B, E)$:- $Student(A, B, 3.5), Course(D, E), Enrolled(A, D)$.

Below is a query based on v, which lists the students who are enrolled in Discrete Math II and have a GPA of 3.5.

Q : $h(A)$:- $v(A, \text{'Discrete Math II'})$.

2.1.5 Views, Open World and Closed World Assumptions

In general, there are two assumptions about the views, called *Closed World* and *Open World* [AD98]. Next, we discuss the impact of these assumptions on query containment and hence query rewriting.

Let V be a view and I an extension of V. Then, under the closed world assumption (CWA), I stores all the tuples that satisfy the view definitions in V, i.e. $I = V(D)$,

for every database instance D. That is, view extensions under the CWA are *sound* and *complete* [AD98].

Under the open world assumption (OWA), the view extension I might store some but not all the tuples that satisfy V, i.e. $I \subseteq V(D)$, for some database D. That is, view extensions under OWA are sound but not necessarily complete [AD98].

Example 2.6 *Consider the views V_1 and V_2 defined as follows.*

V_1 : $v_1(A)$:- $r(A, B)$.

V_2 : $v_2(D)$:- $r(C, D)$.

Suppose the view extensions provided are V_1={<a>} and V_2={<b>}. Under CWA, V_1={<a>} implies that r has only tuples with a as the value of the first argument, and V_2={<b>} implies that r has only tuples with b as the value of the second argument. From this we get that under CWA, relation r has only one tuple, namely {<a,b>}.

Under OWA, however, we cannot identify the contents of the relation r. This is because under OWA, V_1={<a>} implies that r has some tuples with a as the value of the first argument, and V_2={<b>} implies that r has some (possibly different) tuples which has b as the value of the second argument.

Since the problems of query rewriting and query minimization are based on query containment, we next review containment of conjunctive queries under the *Open World Assumption* (OWA), often considered in related work.

2.1.6 Containment of Conjunctive Queries

In this section, first, we conceptually define containment of queries and then, review its syntactical characterization for conjunctive queries.

Definition 2.2 (Query Containment) *[CM77]*

Let Q_1 and Q_2 be two conjunctive queries. We say Q_2 is contained in Q_1, denoted $Q_2 \sqsubseteq Q_1$, if for every database instance D, $Q_2(D) \subseteq Q_1(D)$.

Chandra and Merlin [CM77] characterized the containment of conjunctive queries (see Theorem 2.1 below) based on the notion of *Containment Mapping*, defined as follows.

Definition 2.3 (Containment Mapping) *Let Q_1 and Q_2 be conjunctive queries. A mapping μ from variables in Q_1 to variables in Q_2 is a containment mapping if applying μ on Q_1 makes all subgoals in Q_1 a subset of subgoals in Q_2, and the head of Q_1 identical to the head of Q_2.*

Note that using a containment mapping, a variable X that is in position j in a subgoal $r(\bar{X}_i)$ of Q_1 can only be mapped to a variable Y or a constant C in position j in a subgoal $r(\bar{Y}_k)$ in Q_2 , denoted as X/Y or X/c, respectively. That is, a variable cannot be mapped to different variables or constants.

Theorem 2.1 (Containment of Conjunctive Queries [CM77]) *Let Q_1 and Q_2 be standard conjunctive queries defined as follows:*

$$Q_1: \quad h(\bar{X}) :\text{-} g_1(\bar{X}_1), \cdots, g_k(\bar{X}_k).$$

Q_2 : $h(\bar{Y})$:- $p_1(\bar{Y_1}), \cdots, p_l(\bar{Y_l})$.

Then, $Q_2 \sqsubseteq Q_1$ if and only if there exists a containment mapping μ from Q_1 to Q_2.

The following example illustrates details of query containment test.

Example 2.7 *Let $r(A, B)$ and $s(C, D, E)$ be relation schemas. Consider the following queries:*

Q_1 : $h(A, B)$:- $r(A, B)$, $s(B, D, E)$.

Q_2 : $h(A, B)$:- $r(A, B)$, $s(B, D, D)$.

In order to test containment of Q_2 in Q_1, we search for containment mappings from Q_1 to Q_2. Here, we find $\mu = \{A/A, B/B, D/D, E/D\}$ as the containment mapping from Q_1 to Q_2, applying which makes the head of Q_1 and Q_2 identical and make the body of Q_1 a subset of the body of Q_2, hence $Q2 \sqsubseteq Q_1$.

Chandra and Merlin characterized equivalence of standard conjunctive queries based on query containment as follows [CM77].

Definition 2.4 (Equivalence of Conjunctive Queries) *Let Q_1 and Q_2 be standard conjunctive queries. We say that Q_1 and Q_2 are equivalent, denoted $Q_1 \equiv Q_2$, if and only if $Q_1 \sqsubseteq Q_2$ and $Q_2 \sqsubseteq Q_1$.*

2.2 Query Rewriting

In this section, we review *query rewriting* as an approach to query processing in mediator-based information integration systems.

Definition 2.5 (Query Rewriting [Lev01]) *Let Q be a query and V be a set of views. Rewriting Q using V is the problem of generating a new query Q' that refers only to the views in V.*

We say a query Q refers to V if the subgoals in the query body are heads of the views in V. Query rewriting is also defined as a decision problem [LMSS95].

Definition 2.6 (Query Rewriting as a Decision Problem) *Let Q be a query and V be a set of views. Query rewriting is the problem of determining whether there exists a rewriting for Q that refers only to the views in V.*

Query rewriting has been the subject of numerous studies and a number of solutions have been proposed. [Hal00] provides a survey of the subject. [AD98] studies the complexity of the problem for standard conjunctive queries and conjunctive queries with constraints. [Lev01] reviews the proposed solutions for this problem.

In general, there are two main applications for query rewriting: query rewriting for answering queries and query rewriting for optimization, each of which has its own requirements. Several studies including [PL00], [KS09], [LMSS95], [LRO96] and [Mit01] proposed query rewriting algorithms in the context of standard conjunctive queries. [ALM02],[KS10], [PL00], and [KS05a] proposed query rewriting algorithms for conjunctive queries with constraints.

Query rewriting is also studied in other languages. [AGK99] considered queries with disjunction, [BLRR97] considered description logics, [CDLV99] considered regular path queries, and [GHQ95] and [CNS99] considered queries with aggregation.

[Dus97] and [DG97a] considered recursive queries, and [GM99] extended classical tableau representation for query rewriting. While the above studies focused on query rewriting for answering queries, [LRU96], [FRV96], [BDD+98], [CKPS95], [DG97b], and [FRV95] studied query rewriting as an approach to query optimization.

There are different types of rewriting among which we consider *Equivalent Rewriting* and *Maximally Contained Rewriting* (MCR, for short), defined as follows.

Definition 2.7 (Equivalent Rewriting [Lev01]) *Let Q be a query and V be a set of view definitions. The query Q' is an equivalent rewriting of Q using V if: (1) Q' refers only to the views in V, and (2) $Q' \cup V$ is equivalent to Q.*

Note that in order to determine equivalence or containment of query and rewriting, we need to consider view definitions, i.e., Q should be compared with $Q' \cup V$.

Although equivalent rewriting is desired in all applications of query rewriting, it is mainly considered in query optimization and maintaining physical data independence [Lev01]. In other words, it is not always possible to find equivalent rewriting [LMSS95]. When equivalent rewriting is not available, usually the next best option is maximally contained rewriting, defined as follows.

Definition 2.8 (Maximally Contained Rewriting [Lev01]) *Let Q be a query, V be a set of view definitions, and L be a query language. The query Q' is a maximally contained rewriting of Q using V with respect to L if: (1) Q' is a query in L that refers only the views in V, (2) $Q' \cup V$ is contained in Q, and (3) there is no rewriting Q_1 in L such that $(Q' \cup V) \sqsubseteq (Q_1 \cup V) \sqsubseteq Q$ and $Q_1 \cup V$ is not equivalent to $Q' \cup V$.*

It is easy to see that if an equivalent rewriting exists, it is indeed maximally contained.

Based on this, the goal of query rewriting would be one of the following.

1. *Equivalent Rewriting*: Evaluating such rewriting results in an answer that is both sound and complete. It is sound because it satisfies the criteria of the query, and it is complete because it is the same as the query answer.

2. *Maximally Contained Rewriting*: Evaluating such rewriting results in an answer that is sound but not necessarily complete. In other words, evaluating rewriting generates a subset of the query answer.

2.2.1 Rewriting Queries for Query Processing

Recall the architecture of the mediator-based information integration shown in Figure 1.1. In general, a query Q in such an integration cannot be executed as is. The reason is that the relations in the global schema do not contain data. Therefore, in order to evaluate a query Q, we could use query rewriting to expressed Q in terms of relations at the local schemas.

Since the type of integration defines the rewriting approach that can be used for query processing, we first review different types of integration from query processing point of view.

As explained in Chapter 1, the metadata in a mediator-based integration consists of (1) the global schema (which specifies the relations at the integration), (2) the

local schema (which specifies the relations at information sources level), and (3) the mapping between them which is usually in the form of view definitions. In general, there are the following approaches to define the mappings between the local schema and the global schema which also define the type of integration [Len02].

1. Global as View (GAV) approach: every relation at the global schema is defined as a view over the local schema.

2. Local as View (LAV) approach: every relation at the local schema is defined as a view over the global schema.

3. GLAV approach: a combination of the above approaches.

Depending on the integration type (i.e., LAV-based or GAV-based), different query rewriting techniques are applicable. Next, we review these techniques. We do not discuss GLAV-based integration in this thesis.

2.2.2 Query Rewriting in GAV-based Integration

In a GAV-based integration, every relation in the global schema is expressed as a view over the relations in the local schemas, therefore every query over the global schema refers only to views. Based on this, query answering in GAV-based integration could be done using *Query Unfolding* which, intuitively, replaces every subgoal (i.e., view $v_i(\bar{X}_j)$) in the query by the body of view v_i [Ull00]. The result of unfolding Q is an equivalent query Q' which refers only to the relations in the local schemas [Ull00].

That is, the subgoals in the body of Q' consists of the relations in the local schema. Therefore, evaluating Q' finds answers to Q.

Example 2.8 *Consider the following query and view.*

Q: $h(X)$:- $v(X, \text{'Discrete Math II'}), v(X, \text{'Databases'})$.

V: $v(B, E)$:- $Student(A, B, 3.5), Course(D, E), Enrolled(A, D)$.

The result of unfolding Q is an equivalent query defined as follows.

Q: $h(X)$:- $Student(A_1, X, 3.5), Course(D_1, \text{'Discrete Math II'}),$
$Enrolled(A_1, D_1), Student(A_2, X, 3.5),$
$Course(D_2, \text{'Databases'}), Enrolled(A_2, D_2)$.

This is explained as follows. For unfolding subgoal $v(X, \text{'Discrete Math II'})$, based on the view definition, the distinguished variable B in the view head is substituted by variable X, and the distinguished variable E is substituted by the constant 'Discrete Math II'. Non-distinguished variables, A and D, are substituted by new and unique variables A_1 and D_1, respectively. Similarly, for subgoal $v(X, \text{'Databases'})$, distinguished variables B and E are substituted by variable X and constant 'Databases', respectively, and non-distinguished variables, A and D are substituted by new and unique variables A_2 and D_2, respectively.

2.2.3 Query Rewriting in LAV-based Integration

Unlike GAV-based integrations, in a LAV-based integration, the relations in the global schema are not views. In fact, every relation in the local schemas is expressed as a

view over the relations in the global schema, therefore, query rewriting in a LAV-based integration is more involved compared to GAV-based integration.

In the context of LAV-based integration, finding equivalent rewriting is not always possible. The following example illustrates this point.

Example 2.9 *Let $r(A, B, C)$ be a relation schema. Consider the following query Q and the view V_1.*

Q : $h(X)$:- $r(X, Y, Z)$.

V_1 : $v_1(A, B)$:- $r(A, B, B)$.

Note that the view defines the local schema v_1 as a view over the global schema. It is easy to see that because of the restriction in the body of V_1 (imposed by repeated argument B), the best rewriting for Q based on V_1 is Q' below, which is not equivalent to Q.

Q' : $h(X)$:- $v_1(X, Y)$.

This is illustrated in more details using the following instance of r.

Let r = {<1,2,2>, <2,2,3>}. Then Q returns {<1>, <2>} as the answer, and v_1 contains a single tuple <1,2>. Based on this, Q' returns {<1>} as the answer which is not equivalent to the answer from Q.

It has been shown that when the input query and views are conjunctive queries, to express maximally contained rewriting, we need to use Union of Conjunctive Queries (UCQ) as the language of rewriting [AD98]. A UCQ query consists of a set of conjunctive queries all of which use the same query head. We refer to each query in such

a rewriting as a *rule*.

The following example shows a case in which the rewriting has more than one rule, illustrating the fact that the language of rewriting is union of conjunctive queries.

Example 2.10 *Let $r(A, B, C)$ and $s(A, B)$ be relation schemas. Consider the following query Q and the set of views $V = \{V_1, V_2\}$:*

Q : $h(X)$:- $r(X, Y, Z), s(Z, W)$.

V_1 : $v_1(A, C, D)$:- $r(A, B, C), s(D, C)$.

V_2 : $v_2(A, C)$:- $r(A, B, C)$.

The following query Q' is the maximally contained rewriting for Q.

Q' : $h(X)$:- $v_1(X, Z, D), v_1(A, W, Z)$.

$h(X)$:- $v_1(A, W, Z), v_2(X, Z)$.

Q' is contained in Q because each rule in Q' is contained in Q so the union of these rules is also contained in Q. Q' is maximally contained in Q because any combination of view specializations of V_1 and V_2 that generates a contained rule in Q is contained in one of the rules in Q'.

In the rest of our discussions, we focus on query rewriting for answering queries for LAV-based integration, therefore we consider only view-limited rewritings. We consider open world assumptions on views. Moreover, we consider conjunctive queries as the language of the input query and views, and union of conjunctive queries as the language of the rewriting output. This problem is summarized as follows.

- Problem: *Query Rewriting*

- Input: *A query Q and a set of views V*
- Output: *Maximally contained rewriting for Q using V*
- Input Language: *Standard conjunctive queries*
- Output Language: *Union of conjunctive queries*
- Assumptions: *Open world assumption*

It has been shown that the problem of query rewriting is NP-hard [LMSS95]. Moreover, it has been shown that in the context of standard conjunctive queries, the search for a maximally contained rewriting is finite and requires considering the possible conjunction of a maximum of n view heads, where n is the number of subgoals in the query [LMSS95]. This suggests a two phase process for query rewriting, (1) finding proper view heads, and (2) combining them to generate rewriting. Considering that the complexity of query rewriting is exponential (in the number of subgoals in the given query), it may suggest that query rewriting for large number of views is not practical. However, Minicon [PL00] showed that is not the case. In fact, Minicon improved the existing techniques of rewriting, and experimentally showed that not only it is faster than earlier algorithms such as Inverse-Rules [DL00, Qia96, DG97a] and Bucket algorithm [LRO96], but also it can scale up to large number of views.

While the exponential complexity of the rewriting is due to its second phase, existing algorithms including Minicon algorithm, mainly focus on the first phase. Next, we introduce our pattern-based query rewriting algorithm whose focus is more on the combinatorial problem in the second phase of the rewriting.

As mentioned earlier, query processing in the context of GAV-based integration can be done by query unfolding. Next, we consider only the LAV-based integrations, and study the existing query rewriting algorithms for this context.

2.3 Existing Query Rewriting Solutions

In this section, we review the major query rewriting algorithms for LAV-based integration, namely Bucket algorithm [LRO96], Inverse-rules [DL00, Qia96, DG97a], MiniCon [PL00] and Treewise [MS08]. The goal of all these algorithms is to generate maximally contained rewriting.

2.3.1 Bucket Algorithm

Bucket algorithm is a query rewriting algorithm that was introduced in the Information Manifold System [LRO96]. Given a query Q and set of views V, Bucket algorithm generates maximally contained rewriting for Q using V. The input query Q is of the following form.

$Q: \quad h(\bar{X}) :\text{-} p_1(\bar{X}_1), \cdots, p_m(\bar{X}_m),\ C$

where every p_i is a regular predicate, $\bar{X}_i$ is a list of variables, and C is a built-in predicate. Similarly, every input view in V is of the form:

$V_i: \quad v_i(\bar{Y}_i) :\text{-} q_1(\bar{Y}_1), \cdots, q_n(\bar{Y}_n),\ D.$

In the first phase, the algorithm builds a *bucket* for each predicate p_j in Q. Intuitively, a bucket is a data structure that contains a set of subgoals. To build a bucket for a subgoal p_j, the algorithm finds all views $v_i(\bar{Y}_i)$ whose body contain predicate with the same name as p_j, and for each such predicate $q_k = p_j$, it adds $v_i(\phi(\bar{Y}_i))$ to the bucket that corresponds to p_j. The mapping ϕ maps every distinguished variable y in $v_i(\bar{Y}_i)$ to a variable x in $p_j(\bar{X}_j)$ if y appears in $q_k(Y_k)$ and in the same position as x appears in $p_j(\bar{X}_j)$; otherwise, ϕ replaces y with a new fresh variable.

In the second phase, the algorithm considers all the possible combinations of predicates, one from each bucket, and checks whether each combination generates a contained query, or can be changed to a contained query if additional built-in constraints are added to it. Next, the Bucket algorithm minimizes each of these

contained queries by removing redundant subgoals, and returns the union of these contained rules as the maximally contained rewriting for Q using V.

The following example illustrates details of the Bucket algorithm. The query and views are taken from [PL00] based on the relation schemas $r_1(A, B)$ and $r_2(A, B)$.

Example 2.11 (Bucket algorithm) *Consider the query Q and the set of views V which includes $\{V_1, V_2, V_3\}$.*

Q : $h(X)$:- $r_1(X, Y), r_2(Y, Z)$

V_1 : $v_1(A)$:- $r_1(A, B)$

V_2 : $v_2(D)$:- $r_2(C, D)$

V_3 : $v_3(A, B, C)$:- $r_1(A, B), r_2(B, C)$

In the first phase, the Bucket algorithm generates the following two buckets, one for r_1 and the other for r_2 in Q.

$Bucket_{r_1} : \{v_1(X), v_3(X, Y, C)\}$.

$Bucket_{r_2} : \{v_2(Z), v_3(A, Y, Z)\}$.

For predicate $r_1(X, Y)$ in Q, we create $Bucket_{r_1}$. Since view V_1 contains predicate $r_1(A, B)$ which has the same name as $r_1(X, Y)$, we add $v_1(\phi_1(A)) = v_1(X)$ to $Bucket_{r_1}$. Note that A is a distinguished variable in V_1, and it appears in $r_1(A, B)$ in the same position as variable X in $r_1(X, Y)$. Next, we consider view V_3 for predicate $r_1(X, Y)$, and add $v_3(X, Y, C)$ to $Bucket_{r_1}$. Similarly, we create $Bucket_{r_2}$ which would include $v_2(X)$ and $v_3(A, Y, Z)$ based on views V_2 and V_3, respectively. This completes the first phase.

The second phase combines these buckets. This is done by taking the Cartesian product R of the elements in the buckets. For each combination r in R, we create a query Q'_i in which the query head is the head of Q, and the query body is the conjunction of the predicates in r. Each query Q'_i is called a candidate rule. This

yields the following candidate rules:

Q'_1 : $h(X)$:- $v_1(X), v_2(X)$.

Q'_2 : $h(X)$:- $v_1(X), v_3(A, Y, Z)$.

Q'_3 : $h(X)$:- $v_3(X, Y, C), v_2(Z)$.

Q'_4 : $h(X)$:- $v_3(X, Y, C), v_3(A, Y, Z)$.

Next, the containment of each candidate rule Q'_i in Q is verified. For Q'_1, if we unfold the views $v_1(X)$ and $v_2(X)$, we get the subgoals $r_1(X, Y_1)$ and $r_2(Y_2, X)$. Unlike in Q, since the variables Y_1 and Y_2 are not necessarily the same, Q'_1 is not contained in Q. For a similar reason, Q'_2 is not contained in Q. The other two rules are contained in Q, however, they contain redundant subgoals. In fact Q'_3 is contained in Q'_4 and Q'_4 itself can be minimized as follows:

Q'_4 : $h(X)$:- $v_3(X, Y, Z)$.

The main problem with the Bucket algorithm is the second phase, in which it tests the containment of each candidate rule in query Q. Since the number of combinations is, in general, exponential in the number of subgoals in Q (the cost of cartesian product is m^n where n is the number of subgoals in Q and m is the number of elements in every bucket), and the containment test of each candidate rule is NP-complete for conjunctive queries (which becomes even more expensive when constraints are also included), the Bucket algorithm is not a practical solution for generating maximally contained rewriting.

2.3.2 Inverse-Rules Algorithm

The Inverse-Rules algorithm used in InfoMaster System [DL00, Dus97] is a rewriting algorithm that inverses the resource definitions (views). Intuitively, given a LAV-based view, it generates GAV like views independent of any query. To illustrate this,

consider the following view V for which we define the inverse rules.

$V: \quad v(\bar{X}) \text{ :- } r_1(\bar{X}_1), \cdots, r_n(\bar{X}_n),$

For every subgoal $r_j(\bar{X}_j)$ in V, Inverse-Rules generates a rule IR_j, defined as follows.

$IR_j: \quad r_j(A_1, \ldots, A_m) \text{ :- } v(\bar{X})$

where A_i in $r_j(A_1, \ldots, A_m)$ is determined as follows.

$$A_i = \begin{cases} x & \text{if variable } x \text{ at position } i \text{ in } r_j(\bar{X}_j) \text{ is distinguished in } V. \\ f_{V_j}(\bar{X}_j) & \textit{otherwise} \end{cases}$$

where for every view, f_{V_j} is a function symbol.

Given a query Q and set of views V, Inverse-Rules algorithm uses inverse rules of the views in V to generate a query plan that is maximally contained in Q. Note that the output of the Inverse-Rules algorithm is a logic query because it contains function symbols. The following example illustrates details of the Inverse-Rules algorithm.

Example 2.12 *Consider the same query and views used in example 2.11. For every view, we define the inverse rules as follows.*

$IR_1: \quad r_1(A, f_{V_1}(A)) \text{ :- } V_1(A).$

$IR_2: \quad r_2(f_{V_2}(D), D) \text{ :- } V_2(D).$

$IR_3: \quad r_1(B, C) \text{ :- } V_3(B, C, F).$

$IR_4: \quad r_2(G, H) \text{ :- } V_3(E, G, H).$

Next, we use these rules together with the query Q to generate a query plan as follows.

$Q: \quad h(X) \text{ :- } r_1(X, Y), r_2(Y, Z)$

$r_1(A, f_{V_1}(A)) \text{ :- } V_1(A).$

$r_2(f_{V_2}(D), D) \text{ :- } V_2(D).$

$r_1(B, C)$:- $V_3(B, C, F)$.

$r_2(G, H)$:- $V_3(E, G, H)$.

Note that the result of Inverse-Rules is a logic query. In order to evaluate the query plan, we use a bottom-up evaluation which is guaranteed to terminate in finite steps [DL00].

2.3.3 MiniCon Algorithm

The MiniCon algorithm [PL00] was introduced to address the performance and scalability issues in the Bucket and Inverse-Rules algorithms. Minicon algorithm is similar to the Bucket algorithm in that it performs the rewriting in two phases by first forming the buckets and then combining them. However, it differs from the Bucket algorithm explained as follows. In the first phase, Minicon identifies the views whose subgoals correspond to subgoals in Q. For each view V_i and subgoal p in Q, it finds a partial mapping from p to a subgoal g in V_i, and then finds the minimal additional set of subgoals that needs to be mapped to subgoals in V_i, given that p will be mapped to g. This set of subgoals and mapping information is called *MiniCon Description* (MCD). The following definition, taken from [PL00], formally defines MCD.

Definition 2.9 (Minicon Description (MCD)) *A MCD, C for a query Q over a view V is a tuple of the form $(h_C, V(\bar{Y})_C, \phi_C, G_C)$, where h_C is a head homomorphism on V, $V(\bar{Y})_C$ is the result of applying h_C to V, i.e., $\bar{Y} = h_C(\bar{A})$, where $\bar{A}$ denote the head variables of V, ϕ_C is a partial mapping from $Vars(Q)$ to $h_C(Vars(V))$, and G_C is a subset of the subgoals in Q which are covered by some subgoal in $h_C(V)$ using the mapping ϕ_C.*

In the second phase, Minicon combines the MCDs to produce the rewritings. It is shown that only certain combinations of MCDs do not generate redundant rewriting

rule [PL00]. Let S be a set of MCDs. Then MCDs in S do not generate redundant rewriting if

1. No two MCDs in S cover the same subgoal, i.e., $G_{C_i} \cap G_{C_j} = \emptyset$, for all i and j, where C_i and C_j are two MCDs in S.

2. MCDs in S cover all the subgoals in the body of the query, i.e., $\cup_i\ G_{C_i} = subgoals(Q)$.

Intuitively, a MCD C represents the set of subgoals G_C of Q. To generate a rewriting using MCDs, we need to form the join criteria that exists in Q. For this, we consider the set J of variables that appear in the subgoals of G_C and also in the rest of subgoals in Q including $head(Q)$. In order to be able to build the join criteria of Q in rewriting, all variables in view that corresponds to variables in J should be distinguished.

Based on this, Minicon combines MCDs to generate rewriting. Note that the number of combinations to be considered in MiniCon is less than those considered in the Bucket algorithm. Moreover, it is shown that rewriting rules generated by MCDs are guaranteed to be contained in the query [PL00]. Therefore, unlike the Bucket algorithm, Minicon does not require to perform containment test for the generated rules. As a result, Minicon performs query rewriting more efficiently than the Bucket algorithm, however, the complexity remains $O(2^{n^2})$.

The following example shows details of the rewriting by the Minicon algorithm.

Example 2.13 (Minicon algorithm) *Consider the query Q and views V_1, V_2, and V_3.*

Q : $h(X)$:- $r_1(X,Y), r_2(Y,Z)$

V_1 : $v_1(A)$:- $r_1(A,B)$

V_2 : $v_2(D)$:- $r_2(C,D)$

V_3 : $v_3(A,B,C)$:- $r_1(A,B), r_2(B,C)$

We use Minicon algorithm to rewrite Q based on views. We start with subgoal $r_1(X,Y)$, and consider its variables. Variables X and Y in Q are distinguished and join variable respectively.

We consider view V_1 and try to create a MCD C_1 based on V_1. The mapping $\phi_1 = \{X/A, Y/B\}$ maps the subgoal $r_1(X,Y)$ in Q to the subgoal $r_1(A,B)$ in V_1. Therefore, we set $G_1 = \{r_1(X,Y)\}$. Variable X in $r_1(X,Y)$ is a distinguished variable and its corresponding variable in $r_1(A,B)$, A, is a distinguished variable in V_1. Similarly, variable Y in $r_1(X,Y)$ is a join variable, but its corresponding variable B in $r_1(A,B)$, is not distinguished in V_1. As a result, we need to add additional subgoals to G_1 so that B is no longer a join variable. To do that, we need to add $r_2(Y,Z)$ to G_1. However, since there is no subgoal r_2 in V_1, we cannot proceed and discard MCD C_1. As a result, view V_1 cannot be used to form any MCD. Similarly, we cannot use V_2 to form a MCD. For V_3, we start with subgoal r_1, i.e., $G_3^1 = \{r_1(X,Y)\}$, and define the partial mapping from Q to V_3, i.e., $\phi_3^1 = \{X/A, Y/B\}$. Since variables A and B in V_3 that correspond to variables X and Y in Q, respectively, are distinguished, we define MCD C_3^1 based on G_3^1 and ϕ_3^1, i.e., $C_3^1 = (\{A/A, B/B, C/C\}, v_3(X,Y,C), \{X/A, Y/B\}, r_1(X,Y)\})$. Next, considering subgoal $r_2(Y,Z)$, we define $G_3^2 = \{r_2(Y,Z)\}$ and correspondingly, define the partial mapping $\phi_3^2 = \{Y/B, Z/C\}$. Since variable B which corresponds to variable Y (a join variable in Q) is a distinguished variable in V_3, we define MCD C_3^2 based on G_3^2 and ϕ_3^2, i.e., $C_3^2 = (\{A/A, B/B, C/C\}, v_3(A,Y,Z), \{Y/B, Z/C\}, \{r_2(Y,Z)\})$. In the second phase, Minicon searches for combinations of MCDs to generate the rewriting rules. Here, C_3^1 and C_3^2 are the only MCD's, and together they satisfy the two conditions above for a contained rewriting. This yields the following rewriting of Q:

Q' : $h(X)$:- $V_3(X,Y,C), V_3(A,Y,Z)$.

2.3.4 Treewise Algorithm

The rewriting algorithms we discussed so far do not minimize the size (i.e., the number of subgoals) of the rules in the rewriting. It has been shown that the Bucket algorithm and Minicon naturally generate rewriting rules that include redundancies [PL00, LMSS95]. Even though [PL00] introduces a polynomial algorithm for reducing the number of subgoals in the rewriting, it does not minimize the number of subgoals in rules. The Treewise algorithm [MS08] is a rewriting technique that addresses this issue.

Unlike Minicon that uses the concept of MCD, which represents the minimum number of subgoals, Treewise uses the maximum number of subgoals to generate the building block for rewriting, which is called *tuple*.

In the first phase of the rewriting Treewise uses a hyper-graph model to represent query and views. The hyper-graph generated for a query consists of hyper-nodes of the query head as well as the hyper-nodes of the subgoals in the query body. Every hyper-node consists of some nodes each of which represents an occurrence of a variable in the query body. To generate the hyper-graph of a query Q, Treewise algorithm considers every occurrence of a variable X in Q, for which it creates a node. The nodes are labeled (i, j, k), where i is the index of the subgoal p that uses X, j is the position of X in the list of the variables of p, and k is the position of X in the query head ($k = 0$ means that X does not appear in the head, i.e., it is not a distinguished variable). Edges between nodes of the repeated variables are defined to represent the join criteria in the query.

Once the hyper-graphs for query and views are generated, it finds consistent partial mappings from the hyper-graph of query to hyper-graphs of views, and in the second phase, it combines the partial mappings properly to generate maximally contained rewriting, i.e. union of conjunctive queries.

A desired partial mapping μ in Treewise must satisfy the following conditions:

1. Head-unification: Tests if all distinguished variables in query are mapped to distinguished variables in view.

2. Join-recoverability: Consider two nodes n_1 and n_2 of an edge in the hyper-graph of the query. If n_1 (but not n_2) is in the domain of μ, then to satisfy this condition, $\mu(n_1)$ should be a distinguished node in the graph of view.

3. Partial-mapping consistency: Consider two nodes n_1 and n_2 of an edge in the graph of the query. If both n_1 and n_2 are in the domain of μ, then to satisfy this condition, $\mu(n_1)$ and $\mu(n_2)$ should be nodes of an edge in the graph of the view, or they should be distinguished variables in which case they should be equated by adding new edge in the hyper-graph of view.

4. Partial-mapping-maximality: As explained above, in order to satisfy partial-mapping consistency, we might need to add new edges to the hyper-graph. Partial-mapping-maximality condition tests that unnecessary constraints are not added to the rewriting.

The first three conditions above guarantee consistency of the mappings and the containment of the generated rewriting. The last condition ensures the maximality of the generated rewriting.

If a mapping μ satisfies all these conditions, a tuple is created for μ which contains the following information.

1. Mapping μ.

2. A copy of the head hyper-node from the hyper-graph of the corresponding view, V. Intuitively, this corresponds to the second element in a MCD (in the Minicon

algorithm) which is the result of applying head homomorphism to the head of view.

3. A copy of the hyper-nodes of the query to which edges are added during the mapping construction phase. Intuitively, this together with the next item correspond to the constraints introduced in the head homomorphism in the first element of a MCD of Minicon.

4. A copy of the set of hyper-nodes in Q that are connected.

5. The set of hyper-nodes of the query covered by μ.

In the second phase of rewriting, the Treewise algorithm searches for the combination of the tuples (generated in the first phase) that cover the query body. Every valid combination generates a rule of the rewriting.

Treewise algorithm is shown ([MS08]) to outperform the Minicon algorithm in most query types defined in [PL00].

The following example illustrates the detail of creating the hyper-graphs for queries and views in a rewriting problem.

Example 2.14 *Consider again the same query Q and views V_1, V_2, and V_3, reproduced below for convenience.*

Q : $h(X)$:- $r_1(X,Y), r_2(Y,Z)$

V_1 : $v_1(A)$:- $r_1(A,B)$

V_2 : $v_2(D)$:- $r_2(C,D)$

V_3 : $v_3(A,B,C)$:- $r_1(A,B), r_2(B,C)$

In the first phase of the query rewriting, Treewise algorithm creates the hyper-graphs for the query and views. To see the details of creating the hyper-graph, we consider query Q, and create its hyper-graph (see Figure 2.1.a). Since there are four

occurrences of variables in the body of Q, we create four nodes. For this, we first consider subgoal $r_1(X,Y)$. We create a node N_X for variable X in subgoal $r_1(X,Y)$, and labeled it $(0,1,1)$. This is explained as follows. Recall that every label is a triple (i,j,k). For variable X, i (the first position in the label) is 0 because subgoal $r_1(X,Y)$ is the first subgoal in Q (subgoal indexes in Treewise start from 0), j (the second position in the label) is 1 because X is the first variable in $r_1(X,Y)$, and k (the third position in the label) is 1 because X is in the first position in the query head. Similarly, we create a node N_{Y_1} for the occurrence of Y in $r_1(X,Y)$, and label it $(0,2,0)$ (the last position in the label is 0 because Y does not appear in the query head). As shown in Figure 2.1.a, these two nodes form a hyper-node for the subgoal $r_1(X,Y)$ (a line around these two nodes is drawn to identify this hyper-node). Next, we consider subgoal $r_2(Y,Z)$ and its variables. We create a node N_{Y_2} for the occurrence of Y in $r_2(Y,Z)$ and label it $(1,1,0)$. Similarly, we create a node N_Z for variable Z in $r_2(Y,Z)$ and label it $(1,2,0)$. These two nodes form a hyper-node for subgoal $r_2(Y,Z)$. In order to show the join criteria in the query (i.e., Y is a shared variable between subgaols $r_1(X,Y)$ and $r_2(Y,Z)$), an edge is created to connect the nodes for the two occurrences of Y, i.e., N_{Y_1} and N_{Y_2}.

The nodes in which the last position in the label is not zero (N_X, in this case) form the hyper-node for the query head, and together with the hyper-nodes in the body form the hyper-graph of the query. Figure 2.1.a, 2.1.b, 2.1.c, and 2.1.d illustrate the hyper-graphs generated for query Q and views V_1, V_2, and V_3 in Example 2.14.

Once the hyper-graphs are created, Treewise algorithm finds all the consistent partial mappings from the hyper-graph of Q to those of views, listed as follows.

$\mu_1 = \{(0,1,1) \rightarrow (0,1,1), (0,2,0) \rightarrow (0,2,0)\}$*: mapping the hyper-node of r_1 in the hyper-graph of Q to the hyper-node of r_1 in the hyper-graph of V_1.*

$\mu_2 = \{(1,1,0) \rightarrow (0,1,0), (1,2,0) \rightarrow (0,2,1)\}$*: mapping the hyper-node of r_2 in*

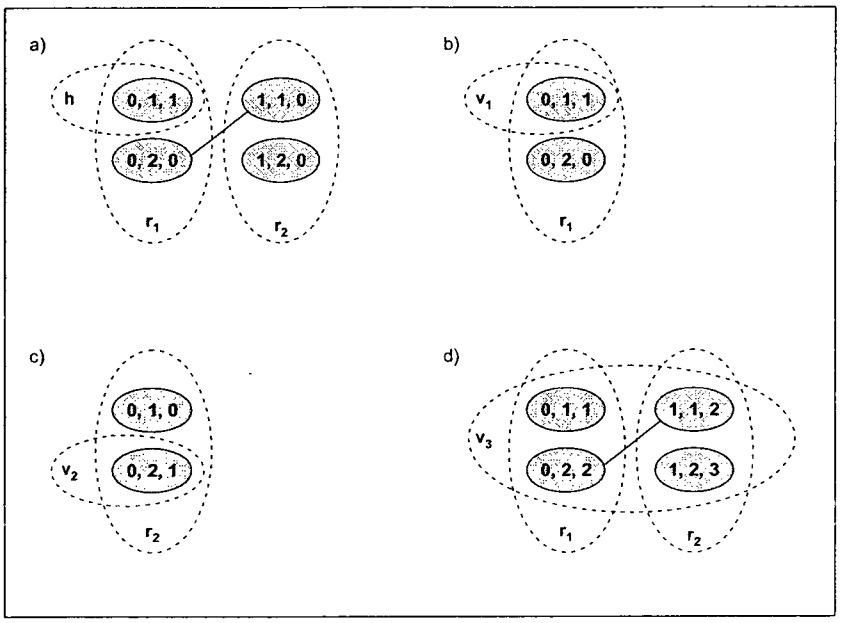

Figure 2.1: The hyper-graphs a, b, c, and d shown here represent query Q, and views V_1 , V_2, and V_3 in Example 2.14, respectively.

the hyper-graph of Q to the hyper-node of r_2 in the hyper-graph of V_2.

$\mu_3 = \{(0,1,1) \rightarrow (0,1,1), (0,2,0) \rightarrow (0,2,2), (1,1,0) \rightarrow (1,1,2), (1,2,0) \rightarrow (1,2,3)\}$*: mapping hyper-nodes of r_1 and r_2 in the hyper-graph of Q to the hyper-nodes of r_1 and r_2 in the hyper-graph of V_2.*

Next, we test each of the above mappings to see if conditions Head-unification, Join-recoverability, Partial-mapping consistency, and Partial-mapping-maximality are satisfied. We can see that only μ_3 satisfies the above conditions, and μ_1 and μ_2 should be discarded. This is explained as follows.

Considering mapping μ_1, since node $N_X = (0,1,1)$ is the only distinguished variable in the hyper-graph of Q, and it is mapped to $(0,1,1)$ which is also a distinguished variable in V_1, μ_1 satisfies Head-unification condition. However, since there is an edge connected to $N_Y = (0,2,0)$ in the hyper-graph of Q (i.e., Y is a join variable)

but the corresponding node, $(0, 2, 0)$ *does not have an edge and it does not correspond to a distinguished variable, then Join-recoverability condition is not satisfied. As a result, we discard* μ_1. *For a similar reason, we discard mapping* μ_2 *(the node* $(1, 1, 0)$ *in the hyper-graph of* Q *has an edge but it is mapped to a node that is not part of the head hyper-node in* V_2*).*

Mapping μ_3 *is the only consistent mapping, we test if its tuple is good for generating a rewriting. Since* μ_3 *covers all the subgoals of* Q*, its tuple alone generates the rewriting, as follows.*

Q' : $\quad h(X)$:- $V_3(X, Y, Z)$.

While the rewriting generated in Treewise and Minicon are equivalent, as we can see in the above examples, for the same input, the number of subgoals in a rewriting generated by Treewise is less than the number of subgoals in a rewriting by Minicon, in general. This is a major advantage of Treewise compared to Minicon. It has been shown that the performance of Treewise for most of the query types reported in [PL00] is better than the performance of Minicon [MS08].

2.4 Summary

In this chapter, we reviewed the basic definitions and terminologies required for the discussions in this thesis. These include query, view, conjunctive query, query containment, query processing using views, GAV-based and LAV-based integration, equivalent rewriting, maximally contained rewriting. Moreover, we reviewed major query rewriting algorithms in LAV-based integration, namely Minicon, Treewise, Inverse-rules, and Bucket algorithm. In the next chapter, we explain details of our pattern-based query rewriting algorithm.

Chapter 3

Query Rewriting for Standard Conjunctive Queries

In Chapter 1, we reviewed *Query Rewriting* as an approach to answering queries using views in the context of LAV-based data integration, and motivated performance and scalability of query rewriting as important problems in this context.

In this chapter, we study query rewriting to improve performance and scalability of the Minicon algorithm which was shown to outperform some of the existing rewriting techniques, namely the Bucket algorithm and the inverse-rules algorithm. We consider standard conjunctive queries as the input language, and union of conjunctive queries as the output language. Throughout our discussions, we consider the same assumptions as in Minicon [PL00], i.e, open-world assumption on views. The contributions of this chapter are:

1. We propose a formula which uses Stirling numbers to determine the size of rewriting, i.e., the number of rules in the output. This is important in practice because one can estimate time and memory required to perform the rewriting before actually generating it.

2. We propose a novel rewriting algorithm that uses numerical patterns to break a large combinatorial problem into several smaller ones based on which it generates rewriting efficiently.

3. We consider quality of rewriting as another important aspect in this problem, and define the number of subgoals in the rewriting as a measure for quality. We classify the rewriting algorithms into two classes: "bottom-up" and "top-down" approaches. We show that a top-down query rewriter generates rewriting with a better quality compared to a bottom-up query rewriter, i.e., there are fewer number of subgoals in the rewriting under the top-down approach.

4. We show that there is an isomorphism between the rules in results of bottom-up and top-down approaches. This confirms that applying query minimization to the result of a bottom-up approach can generate the same quality as in a top-down approach. We also show that redundant subgoals in the input of rewriting result in redundant rules in the output.

3.1 Pattern-based Query Rewriting

As discussed in Chapter 2, for a conjunctive query Q, a maximally contained rewriting Q' is a set of rules (conjunctive queries), each of which is contained in Q. In general, in order to generate maximally contained rewriting, one should consider all possible view heads (also referred to as head specialization), and combine them to generate contained rules (queries), and return the union of all such contained rules as the rewriting [LMSS95]. Our query rewriting algorithm follows this approach. In order to explain details of our query rewriting algorithm, we introduce a few new concepts including *Coverage* as follows [KS09].

Definition 3.1 (Coverage) *Given a query Q and a view V_i, a coverage C is a data structure of the form $C = \langle S, \phi, h, \delta \rangle$, where S is a subset of the subgoals in Q, ϕ is a mapping from subgoals in S to subgoals in V_i, h is the head of V_i, i.e., $v_i(\bar{X}_i)$, and δ contains sets of variables in S where variables in each set are mapped to the same variable under ϕ.*

The following example illustrates the components of a coverage.

Example 3.1 *Considering the following query Q and views V_1 and V_2.*

Q : $h(X,Y)$:- $r(X,Y)$, $s(Y,Z,W)$.

V_1 : $v_1(A,B)$:- $r(A,B)$.

V_2 : $v_2(B,D)$:- $s(B,D,D)$, $s(E,F,D)$.

The following coverages are generated based on V_1 and V_2.

$C_1 = \langle \{r(X,Y)\}, \{X/A, Y/B\}, v_1(A,B), \{\} \rangle$

$C_2 = \langle \{s(Y,Z,W)\}, \{Y/B, Z/D, W/D\}, v_2(B,D), \{\{W,Z\}\} \rangle$

$C_3 = \langle \{s(Y,Z,W)\}, \{Y/E, Z/F, W/D\}, v_2(N_1,D), \{\} \rangle$

Coverage C_1 is defined based on V_1, and uses the mapping $\phi_1 = \{X/A, Y/B\}$ to cover subgoal $r(X,Y)$. Since every variable in source is mapped to one variable in the target, the set δ_1 is empty. Coverage C_2 is defined based on V_2, and uses the mapping $\phi_2 = \{Y/B, Z/D, W/D\}$ to cover subgoal $s(Y,Z,W)$. Since variables Z and W are mapped to the same variable D, δ_2 includes the set $\{W,Z\}$ which means W and Z should be equated in any rule generated using C_2. Similarly, coverage C_3 is defined based on V_3, and uses $\phi_3 = \{Y/E, Z/F, W/D\}$ to cover subgoal $s(Y,Z,W)$.

Intuitively, coverage extends the concept of MCD used in Minicon in the sense that it does not need the minimal number of subgoals that can participate to generate a head specialization. In fact, we can use coverage to express both MCD in Minicon and Tuple in Treewise.

From the above definition, note that in general, ϕ is a non-injective and non-surjective mapping (e.g., ϕ_2 in C_2 in the above example). That is, under ϕ, (1) two or more variables from S could be mapped to the same variable in view V_i, and (2) there are some variables in V_i that are not image of any variable in S.

Before explaining how to build a coverage, let us explain how coverages are used to generate a rule in rewriting. In order to generate a contained rule for Q, we use a set M of coverages that together contain all the subgoals in Q but do not have common subgoals. We create a copy of Q and call it Q'.

For every coverage C_i in M, we define $\psi_i = \phi_i^{-1}$, the inverse of the mapping of C_i, and modify it to fix the following issues:

1. A variable A in the domain of ψ_i could be mapped to a set D of different variables in S. This is because it is possible that a several variables in S are mapped to the same variable in the view. For every such variable A, we fix this problem by unifying/equating all variables in D. In fact, δ_i in C_i, unifies such variables.

2. A variable A in the domain of ψ might not be mapped to any variable in S. This is because ϕ is not necessary an onto mapping. To fix this, we define a distinct new variable N and update ψ to map A to N.

Then, for each coverage C_i in M, we replace subgoals S (covered by C_i) in Q' by $\psi(h)$ which is called the view specialization based on C_i. Finally, we define $\Delta = \cup\ \delta_i$ for all δ_i taken from the coverages in M, and apply it on Q'. That is, for every set U of variables that should be unified, we pick a variable A from U and substitute in Q' every variable $B \in U$ by A. The resulting query Q' is contained in Q.

Example 3.2 *Consider the coverages generated for query and views in Example 3.1. Coverages C_1 and C_2 generate view specializations $v_1(X, Y)$ and $v_2(Y, Z)$, respectively.*

If combined, they generate the following rule that is contained in Q.

Q' : $h(X,Y)$:- $v_1(X,Y)$, $v_2(Y,Z)$.

To show that Q' is contained in Q, we can unfold Q' using view definitions and perform a containment test.

In order to make sure that the generated rules are contained in the input query Q, we ensure that view specializations (that are joined to form a rule) build the same join criteria in Q. In order to guarantee this, coverages should satisfy certain conditions. To elaborate on this, we define *Joint variables* and *Accessible Variables* as follows [KS09].

Definition 3.2 (Joint Variables) *Let Q be a conjunctive query, S be a subset of subgoals in the body of Q, and S' be the rest of all subgoals in Q including the head. We call the variables appearing in both S and S' as joint variables of S.*

As an example, consider the query Q in Example 3.1. Let $S = \{r(X,Y)\}$, then the joint variables of S are X and Y. The reason is that the set S' of all other subgoals in Q is $\{h(X,Y), s(Y,Z,W)\}$ with which S shares variables X and Y.

It is important to note that if a variable X is a joint variable, one of the following could happen: (1) X is a join variable appearing in two or more subgoals in the body, (2) X is a distinguished variable, or (3) X is both join and distinguished variable.

Definition 3.3 (Accessible Variable) *Given a coverage C, a variable X in subgoals covered by C is accessible if the variable Y in the view in C is distinguished, where ϕ is the mapping in C, and $Y = \phi(X)$.*

For instance, consider the coverage C_2 in Example 3.1. We can see that variable Z is an accessible variable because $D{=}\phi(Z)$, and D is a distinguished variable of V_2. Similarly, we can see that variable W is also accessible.

Next, consider coverage C_3 in Example 3.1. We can see that variable Y is not accessible in C_3. The reason is that in C_3, Y is mapped to variable E which is not a distinguished variable in V_2. Also, we can see that in C_3 variable Z is not accessible, but variable W is accessible.

Based on these, we recall the notion of "useful coverage" as follows.

Definition 3.4 (Useful Coverage [KS09]) *Let Q be a query and V_i be a view. A coverage C for a set S of subgoals of Q with respect to V_i is said to be useful if (1) the coverage mapping ϕ maps all the subgoals in S to V_i, and (2) every joint variable X of S is accessible in C.*

For instance, consider coverages C_1 and C_3 from Example 3.1. Coverage C_1 is a useful coverage because it covers subgoal $r(X, Y)$, whose joint variables, i.e., X and Y are both accessible in C_1. The reason is that X and Y are mapped to A and B, respectively, which are distinguished variables in V_1. However, coverage C_3 is not a useful coverage because it includes subgoal $s(Y, Z, W)$ but the joint variable of $s(Y, Z, W)$, i.e., Y is not accessible in C_3. The reason is that variable Y in C_3 is mapped to variable E which is not distinguished in V_2. In fact, since coverage C_3 is not a useful coverage, we discard it and do not consider it the second phase of rewriting.

We note that the criteria for a coverage to be useful is the same as the criteria for building MCDs in Minicon which guarantees containment of rewriting rule in Q. In the rest of this thesis, we only consider useful coverages.

Definition 3.5 (Basic Coverage) *A coverage C is called basic if removing any subgoal makes it non useful.*

Since MCDs contain the minimal number of subgoals, and satisfy the above conditions, every MCDs is a basic coverage.

3.1.1 Finding Coverages

In order to find coverages for query Q and views V, one could use *bottom-up approach* or *top-down approach*, defined as follows.

1. *Bottom-up approach*: This approach generates basic coverages. For this, we consider every subgoal s_i in Q and every view v_j in V where there exists a mapping ϕ from s_i to a subgoal in v_j. Let $S = \{s_i\}$, then we test if every joint variable A of S is accessible through v_j. If this is the case, we can create a coverage for S. Otherwise, we add to S all the subgoals in Q that are joined with the subgoals in S, update ϕ, and retest if every joint variables of S is accessible through v_j. We repeat this process until all the joint variables of S are accessible through v_j or there exists no more subgoal in Q to be added to S. Finally, a coverage C that uses ϕ and v_j to cover S is created if all the joint variables of S are accessible through v_j. In general, it is possible to find more than one mapping from S to subgoals in v_j each of which results in a different coverage.

   Intuitively, bottom-up approach starts with coverages that contain single subgoals and adds subgoal to each coverage until it becomes a useful coverage. This is why we refer to it as bottom-up approach. We say a rewriting algorithm is bottom-up if it uses bottom-up approach for finding the coverages.

2. *Top-down approach*: This approach generates non-basic coverages. For this, we consider a set S of all the subgoals in Q and every view v_j in V. We create a coverage C that covers S if the following two conditions hold: (1) there exists a mapping ϕ from S to v_j and (2) all joint variables of S are accessible through v_j.

   Subgoals are removed from S in all possible orders so that we find all smaller

sets of subgoals that satisfy the two conditions above. For each set, we create a coverage and continue removing subgoals from S even if it has contributed to a coverage. The reason is that we would like to find all possible coverages so that we do not miss any combination of coverages. During or after this process, all coverages that cannot contribute to a rule in the rewriting should be discarded. In order to identify such coverages, one needs to consider other coverages.

Intuitively, top-down approach starts with coverages that contain all subgoals and removes subgoals from them until no more subgoal could be removed. This is why we refer to it as top-down approach. We say a rewriting algorithm is top-down if it uses top-down approach for finding the coverages.

After forming the coverages, we "combine" coverages in phase 2 to get a rewriting rule. We say two coverages have overlap if the sets of subgoals they cover have non-empty intersection. It is shown that only combinations of coverages without overlap are useful for generating a rewriting rule [PL00].

The following example illustrates a case where coverages have overlap and there is no rewriting.

Example 3.3 *Let $r(A, B)$, $s(C, D)$, and $t(E, F)$ be relations. Consider the following query Q and views $V = \{V_1, V_2\}$:*

Q: $h(A)$:- $r(A, B)$, $s(B, D)$, $t(D, E)$.

V_1: $v_1(A)$:- $r(A, B)$, $s(B, D)$.

V_2: $v_2(B)$:- $s(B, D)$, $t(D, E)$.

Consider coverages C_1 (based on V_1) and C_2 (based on V_2) covering $\{r, s\}$ and $\{s, t\}$, respectively. They overlap on subgoal s, but there is no way to match s from C_1 with s from C_2, since we need both variables B and D accessible through both coverages, which is not the case here. In fact, if all the variables in s were accessible

through a view, then s alone would form a coverage. Since, we cannot match the two copies of s, there is no way of forming a combination that satisfies the join criteria in the query body.

It has been shown that in the context of conjunctive queries, in order to generate maximally contained rewriting, we need to use *Union of Conjunctive Queries* as the language of the generated rewriting [LMSS95]. To generate maximally contained rewriting, we need to find all rules that are contained in the input query (generated by combining coverages), and return the union of such rules. Following the same criteria for building MCD's, defined in [PL00], we build coverages. As shown in [PL00], every valid combination of MCD's (basic coverages) gives a rule that is contained in the input query, therefore finding all possible valid combinations results in a set of rules union of which is the maximally contained rewriting. This is shown in the following example.

Example 3.4 *Let $r(A, B)$, $s(C, D)$, and $t(E, F)$ be relations. Consider the following query and views:*

Q: $h(A)$:- $r(A, B)$, $s(B, D)$, $t(D, E)$.

V_1: $v_1(A, B, D)$:- $r(A, B)$, $s(B, D)$.

V_2: $v_2(B, D, E)$:- $s(B, D)$, $t(D, E)$.

Based on V_1, we generate coverages C_1 and C_2 that cover $\{r\}$ and $\{s\}$, respectively. Also based on V_2, we generate coverages C_3 and C_4 that cover $\{s\}$ and $\{t\}$, respectively. Note that Minicon would generate the same number of MCDs each of which would correspond to one of these coverages. We consider all possible valid combinations of coverages each of which results in a rule contained in Q. Based on this, the following is the maximally contained rewriting for Q. Note that the heads of the two rules in Q' are identical.

Q': $h(A)$:- $v_1(A, B, D_1)$, $v_1(A_1, B, D)$, $v_2(B, D, E)$.

$h(A)$:- $v_1(A, B, D_1)$, $v_2(B, D, E_1)$, $v_2(B_1, D, E)$.

The first rule is based on the coverages C_1, C_2 and C_4, and the second one is based on C_1, C_3 and C_4.

Due to its importance, we discuss the second phase of query rewriting that deals with proper combination of coverages, in a separate section.

3.2 Combining Non-overlapping Coverages: A Partitioning Problem

In this section, we focus on the combining phase of rewriting and introduce an efficient technique for finding non-overlapping combinations of coverages. Basically, this is where our algorithm outperforms Minicon.

We first review the approach used in Minicon. For that, consider a conjunctive query Q with n subgoals, and a set of views V. In order to generate the maximally contained rewriting for Q based on V, for every subgoal in Q, Minicon creates a bucket. Next, it finds all MCDs (coverages), and based on the subgoals each MCD C covers, places C in the corresponding buckets. Finally, Minicon performs a Cartesian product of the buckets to find all combinations of MCDs that do not overlap on any subgoal while covering all the subgoals in the body of Q. The problem here is that finding non-overlapping combinations of MCDs during this Cartesian product is very expensive.

Next, we investigate this problem, and propose our solution for finding non-overlapping combination of coverages in a more efficient way.

To find all possible combinations, we consider every bucket as a bit in a binary representation of n bits, hence every coverage consists of n bits. If a coverage C_j is

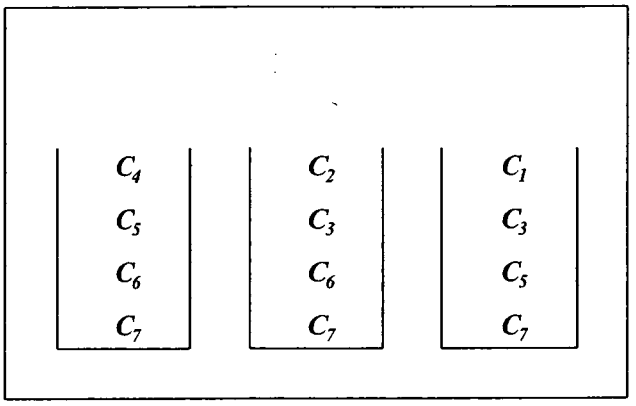

Figure 3.1: All possible occurrences of coverages for a query with 3 subgoals

placed in bucket B_i, the i^{th} bit in the sequence denoting coverage C_j is 1; otherwise it is 0. In other words, the sequence corresponding to a coverage indicates its presence in different buckets, which we consider in our technique as an identifier.

Note that the maximum number of sequences is $2^n - 1$, where n is the number of subgoals in the body of Q; the sequence with all zeros is excluded.

We define *occurrence classes* based on occurrence identifier. All coverages with the same identifier are members of the same occurrence class. Assuming that there is only one coverage from every occurrence class, Figure 3.1 shows all possible coverage occurrences for a query with 3 subgoals.

Now, to find a rewriting based on these coverages, we need to identify sets of coverages with no overlap which cover all the subgoals in the query. For this, we consider identifiers $ID(C_j)$ in binary representation, which satisfy the following two conditions on a given set S of coverages [KS09]:

$$(1) \bigvee_{C_j \in S} ID(C_j) = 2^n - 1,$$

$$(2)\forall\ S' \subseteq S, \bigwedge_{C_k \in S'} ID(C_k) = 0$$

The first condition ensures that the coverages in S cover all subgoals in Q. This is accomplished by taking the identifiers of all coverage in S, to which we then apply the (bitwise) logical OR operation (which amounts to adding the bits). If the result is equal to $2^n - 1$, the coverages in S cover all subgoals in Q. The second condition ensures that no subgoal overlap exists. When the result of the bitwise logical AND operation is 1, S is not a desired combination. Note that in the second condition, checking only pairs of coverages is inadequate, and we need to check the overlap for every subset.

3.2.1 Finding Occurrence Classes

A question at this point is: "Can we find occurrence classes that satisfy the two conditions above, i.e., cover all subgoals in the query and have no overlap?" The answer is positive since we can define this as a partitioning problem. Assume that there are n subgoals in a query Q, and the subgoals are indexed based on their positions in Q, with the right most subgoal indicated as 0, and the leftmost one as $n-1$. For every position i, we include integer 2^i in a set, called P. Now, the number of partitions of P is the number of valid combinations of occurrence classes for rewriting of the query. The following example lists all possible partitions for a query with 3 subgoals.

Example 3.5 *Let Q be a query with three subgoals. So, we have subgoal positions 0, 1, and 2, and hence $P = \{2^2, 2^1, 2^0\} = \{4, 2, 1\}$. There are 5 partitions of P listed as follows.*

1. $P_1 = \{\{4, 2, 1\}\}$

2. $P_2 = \{\{2, 1\}, \{\{4\}\}$

3. $P_3 = \{\{4, 2\}, \{1\}\}$

4. $P_4 = \{\{4, 1\}, \{2\}\}$

5. $P_5 = \{\{4\}, \{2\}, \{1\}\}$

It is easy to see that the partitions do not have overlap and their union forms the original set P (which confirms the definition of partitioning). Moreover, the sum of the numbers in each partition P_i gives an occurrence identifier. In this example, we have occurrence identifiers 1, 2, 3, 4, 5, 6, 7 from $\{1\}$, $\{2\}$, $\{2, 1\}$, $\{4\}$, $\{4, 1\}$, $\{4, 2\}$, *and* $\{4, 2, 1\}$, *respectively. That is, finding the proper combinations for rewriting a query is a partitioning problem [KS09].*

A partition of a set $S = \{2^0, \ldots, 2^{n-1}\}$ is a set of nonempty, pairwise disjoint subsets of S whose union is S. The number of partitions of a set of size n is called the n^{th} Bell number, denoted $B(n)$ [Rio58].

Now, we can determine the maximum number of rules in a rewriting of Q. If there is only one coverage from every occurrence class, then the number of rules in the rewriting would be $B(n)$, where n is the number of subgoals in the query.

Note that with n subgoals, we can generate all identifiers from 1 to $2^n - 1$. This is because we created the set P with numbers 2^0 to 2^{n-1} and if we consider the sum of different combinations of these numbers, we get 1 to $2^n - 1$ (which is the sum of all the numbers from 2^0 to 2^{n-1}).

In general, the number of coverages for an occurrence class is not necessarily 1 and can be any number. To define the maximum number of rules, we use Stirling numbers as follows.

Definition 3.6 (Stirling number [Rio58]) *The Stirling numbers of the second kind $S(n,k)$ indicate the number of ways to partition a set of n elements into k nonempty subsets. Bell numbers $B(n)$ can then be defined as:*

$$B(n) = \sum_{k=1}^{n} S(n,k)$$

Now, using Stirling numbers, we express the maximum number of rules in the rewriting of query Q with n subgoals. Suppose for every occurrence class, there are m coverages. Then, the maximum number of rules in the rewriting of Q would be:

$$\sum_{k=1}^{n} m^k S(n,k)$$

In fact, $S(n,k)$ gives the number of cases in which k different occurrence classes together cover the body of query Q without any overlap [KS09].

The above formula is important as it gives the maximum number of rules in a rewriting, which also helps analyzing the complexity of the query rewriting problem.

Also, Stirling numbers help to perform the combination phase of query rewriting more efficiently. Recall that for a query with n subgoals, we need to multiply n buckets in the combination phase and discard combinations that have overlapping coverages. Using Stirling numbers, we can break a large multiplication on the original buckets into a maximum of $B(n)$ sets of fewer and smaller buckets. That is, we create $S(n,k)$ buckets, where k is the number of occurrence classes in each partition, and copy the contents of the original buckets into these smaller buckets. We elaborate on this process in the next section.

3.2.2 Creating Buckets Using Patterns

Consider a query with 3 subgoals and coverages with occurrence identifiers 1, 2, 3, 4, 5, 6, and 7. The original set of buckets is illustrated in Figure 3.1. However, these buckets can be broken down into smaller buckets which results in efficient multiplication. For this purpose, we consider every possible partitioning (as listed in Example 3.5), get the sum of the numbers in each partition and create a set of buckets accordingly.

1. $\{\{7\}\}$: one bucket for occurrence class 7.
2. $\{\{3\},\{4\}\}$: two buckets, one for each of occurrence classes 4 and 3.
3. $\{\{6\},\{1\}\}$: two buckets, one for each of occurrence classes 6 and 1.
4. $\{\{5\},\{2\}\}$: two buckets, one for each of occurrence classes 5 and 2.
5. $\{\{4\},\{2\},\{1\}\}$: three buckets, one for each of occurrence classes 4, 2, and 1.

We next define the notion of *combination pattern* based on the occurrence identifiers in each partition [KS09]. Intuitively, every pattern indicates the number of buckets we need and the coverages (based on occurrence identifier) they should contain. The maximum number of patterns for a query with n subgoals is $B(n)$.

Definition 3.7 (Combination Pattern) *For a query with n subgoals, a set of positive integers $\mathbb{P}$ is a Combination Pattern, if it satisfies all the following properties:*

1. $\mathbb{P} \subseteq \{1, \ldots, 2^n - 1\}$
2. $\sum_{i \in \mathbb{P}} i = 2^n - 1$, *and*
3. $\forall\ i, j \in \mathbb{P},\ i \wedge j = 0$, *where $\wedge$ is the extended bitwise operation on integers.*

The following recursive procedure called *FindPatterns* finds all combination patterns for the numbers in a set N. In order to find the patterns for a query with n subgoals, we call *FindPatterns* with $A = 2^n - 1$ and $N = \{1, 2, 3, \ldots, 2^n - 1\}$. Note that the numbers in N are the available occurrence identifiers and in practice, not all occurrence identifiers are present. This, in general, results in a fast execution of *FindPatterns* algorithm.

Intuitively, given the set $\{1, 2, 3, 4, 5, 6, 7\}$, this algorithm is supposed to return $\{\{7\}, \{6, 1\}, \{5, 2\}, \{4, 3\}, \{4, 2, 1\}\}$ as the set of patterns where each number is an occurrence identifier.

```
Procedure FindPatterns (A, N, R)
Input:
   A is the sum of all occurrence identifiers
   N is the ascendingly ordered set of available occurrence
     identifiers to be used in the patterns
Output:
   R is the list of patterns built from the numbers in N for A

BEGIN
   IF the patterns for A is already computed THEN
      assign it to R, and exit.
   ELSE
      BEGIN
         1- R = {}
         2- current = the largest number in N that is at most A
         3- complement = A - current
         4- IF (complement = 0) THEN
            add {current} to R
         5- ELSE
            BEGIN
               6- FindPatterns(complement, N, P_comp)
               7- For every P_i ∈ P_comp and every m_i ∈ P_i,
```

```
                IF P_i ⊆ N AND m_i ∧ current = 0 THEN
                    add current to the beginning of P_comp and
                    add P_comp to R
            END
        8- Find the next number x in N such that A/2 ≤ x < current
        9- IF such a number x exists THEN
            assign x to current and go to step 3.
        10- ELSE
            Return R as the result for input A.
    END
END.
```

The following example illustrates the steps of this algorithm.

Example 3.6 *Assume a query Q with three subgoals for which we want to find all the combination patterns. Since $n = 3$, we have that $S = 2^n - 1 = 7$ and the set of possible identifiers is $S = \{1, 2, 3, 4, 5, 6, 7\}$. Accordingly, we call FindPatterns with arguments $(7, \{1, 2, 3, 4, 5, 6, 7\}, R)$. The steps of execution are as follows.*

1. *current $= 7$ and hence complement $= 0$. Thus based on step 4, $R = \{\{7\}\}$. Next, we find the new value for current which is 6, and go to step 3.*

2. *current $= 6$, so complement $= 1$. Then based on step 6, we compute P_1. For this, we call FindPattern$(1, \{1, 2, 3, 4, 5, 6, 7\}, P_1)$, which terminates in one iteration and returns $\{\{1\}\}$.*

   Based on step 7, since $\{1\} \subseteq \{1, 2, 3, 4, 5, 6, 7\}$, and since there is no bitwise overlap between 1 $\equiv$001 and 6$\equiv$110 , we add 6 to $\{1\}$ and add the result $\{6, 1\}$ to R. At this point $R = \{\{7\}, \{6, 1\}\}$. The next value for current is 5, and we continue at step 3.

3. *current $= 5$, so complement $= 2$. Then based on step 6, we compute P_2 for which we recursively call FindPattern$(2, \{1, 2, 3, 4, 5, 6, 7\}, P_2)$. In the first*

iteration, it assigns P_2=$\{\{2\}\}$. *However, in the second iteration where current is* 1, *and complement* = 1 *(2-1=1), since current and the elements in* $P_{comp} = P_1 = \{\{1\}\}$ *have bitwise overlap, we do not include the pattern* $\{1\}$ *to* P_2. *This process terminates in two iterations and returns* $\{\{2\}\}$ *for* P_2.

Based on step 7, since $\{2\} \subseteq \{1,2,3,4,5,6,7\}$, *and since there is no bitwise overlap between 2 and 5, we add 5 to* $\{2\}$ *and add the result* $\{5,2\}$ *to R. At this point* $R = \{\{7\},\{6,1\},\{5,2\}\}$. *The next value for current is 4, and the process continues at step 3.*

4. *current* = 4, *so complement* = 3. *Then based on step 6, we compute* P_3 *by calling* $FindPattern(3,\{1,2,3,4,5,6,7\},P_3)$. *In its first iteration, it assigns* $\{3\}$ *to* P_3, *and in the second iteration where current* = 2, *and complement* = 1 *(3-2=1), since current* = 2 *and the elements in* $P_{comp} = P_1 = \{\{1\}\}$ *have no bitwise overlap, we add 2 to every pattern in* P_1. *This process terminates in two iterations and returns* $\{\{3\},\{2,1\}\}$ *for* P_3.

   Based on step 7, since $\{3\} \subseteq \{1,2,3,4,5,6,7\}$, *and there is no bitwise overlap between 3 and 4, we include 4 in* $\{3\}$ *and add the result* $\{4,3\}$ *to R. Similarly, since* $\{2,1\} \subseteq \{1,2,3,4,5,6,7\}$, *and there is no bitwise overlap between 2 and 4 nor between 1 and 4, we insert 4 into* $\{2,1\}$ *and add the result* $\{4,2,1\}$ *to R. At this point* $R = \{\{7\},\{6,1\},\{5,2\},\{4,3\},\{4,2,1\}\}$. *The next value for current is 3, but the process terminates since every element in* $\{1,2,3,4,5,6,7\}$ *that is greater than* 7/2 *is considered.*

The complexity of finding combination patterns is exponential, as it is based on computing Bell number which is exponential. However, the combination patterns for queries with different number of subgoals can be identified a priori. The number of combination patterns for queries with 3, 4, 5, 10, and 15 subgoals are 5, 15, 52, 115975

and 1382958545, respectively. It is important to note that when some identifiers are not present, the number of patterns drop significantly. As explained earlier, our algorithm considers only the available identifiers, and performs quite well in practice.

The following figure compares the number of operations required in the second phase of rewriting for Minicon and Pattern-based algorithm for queries and views that generate all possible occurrence identifiers (i.e., the worst case).

Number of subgoals	$2^{n(n-1)}$ operations in Minicon	B(n) operations in Pattern-based
3	64	5
4	4,096	15
5	1,048,576	52
6	1,073,741,824	203
7	4,398,046,511,104	877
8	$(2^7)^8$	4140

Figure 3.2: Comparing Minicon and Pattern-based algorithm based on the number of operations required in the second phase for queries and views with all occurrence identifiers.

3.3 Query Rewriting Algorithm

In this section, we describe our query rewriting algorithm as well as possible approaches for each step. The basis for the correctness of our algorithm is that it finds all possible coverages (each of which generates a view specialization) and considers all possible combinations of such view specializations to ensure that the generated rewriting is maximally contained in the input query. This is discussed later in Section 3.3.3.

Algorithm *QueryRewriter*(Q, V, R)
Input:
Q: a query to be rewritten.
V: the set of views to be used in the rewriting.

Output:
R: a maximally contained rewriting for Q based on V.
BEGIN

Phase 1

1- For every view, find the basic coverages.

Phase 2

2- Combine coverages to generate rules each of which is contained in Q.
3- Assign the union of all the rules generated in step 2 to R, and return R as the rewriting output.
END.

The steps in the rewriting algorithm are explained as follows.

3.3.1 Finding Basic Coverages

In the first phase of our rewriting algorithm in which we find the coverages, we can consider two possible approaches, described as follows.

1. Subgoal based approach:

   In this approach (also used in Minicon [PL00]), in order to find the basic coverages, we consider a set S with a single subgoal sg_i in the query Q and its joint variables J_S. For every view V_j that includes sg_i, we ensure that all the variables A in J_S are accessible through V_j, that is image of A under the mapping is a distinguished variable in V_j. If this is the case, we create a new coverage

C_{ji} based on V_j and assign S to C_{ji}. Otherwise, we add more subgoals to S, update J_S accordingly, and inquire if V_j can be useful in forming a coverage.

Adding subgoals to S is based on the joint variables that are not accessible. If $A \in J_S$ is not accessible, we include all the subgoals in Q in which A has appeared. After adding the new subgoals to S, we recalculate joint variables J_S of S and repeat the test. This process terminates if a basic coverage for sg_i (and possibly some other subgoals) is found, or all possible subgoals are added to S, and there is at least one join variable that is not accessible.

2. View based approach:

   The idea in this approach (also used in [KS05a]) in forming the coverages is similar to the subgoal based approach, except for the order in which views are processed. To be precise, instead of considering one subgoal and checking all the views that cover it, we consider one view at a time, identify all the coverages it can generate and then move to another view. This approach seems to be more efficient since it checks every pair of view-subgoal only once, whereas the first approach requires some additional bookkeeping to avoid creating redundant coverages.

3.3.2 Combining Coverages

In the second phase in which we combine the coverages, each subgoal in the query is assigned a bucket and we place coverages in the corresponding buckets. Note that a coverage might appear in several buckets as it may cover several subgoals. As illustrated earlier in Example 3.3, coverages that have overlap over some subgoals should not participate in the same combination since otherwise they may introduce unnecessary restrictions.

In general, to find coverages that do not overlap and generate contained rules,

there are three possible approaches defined as follows [KS09].

1. Detection and Recovery- Simple Cartesian Product:

   In this approach which is used in the Bucket algorithm, to find all the rules, we perform the Cartesian product of the buckets, and eliminate rows that contain overlapping coverages. Assuming that query Q has n subgoals, there will be n buckets. A memory efficient way to perform the Cartesian product of the buckets is to define an index I_i for each bucket with an initial value of 0. At each iteration, the set of indexes $\{I_1, \ldots, I_n\}$ points to coverages in their respective buckets: $B_1, \ldots, B_n$. These coverages cover all the subgoals in Q and, if they do not have any overlap, they can yield a rule.

   For that, we copy the coverages into a set R, and create a query in which the head is the same as the head of Q, and the subgoals in the body are view specializations defined based on the coverages in R, as explained in Section 3.1.

   Recall that for every bucket P, we considered an index I_P with the initial value of 0. Intuitively, we use this set of indexes as a counter to go through all possible combinations. To find the next combination (a new iteration), we consider the index of the last bucket, $P = n$, and perform the following steps. We increment I_P (initially I_n) by 1 and if it is less than the size of the bucket B_P, we stop and define a new set of coverages based on $\{I_1, \ldots, I_n\}$. Otherwise, we set I_P=0, decrement P, and continue this process until we either find a new set of coverages, or P becomes zero. In other words, this process simulates a counter with n digits in base n. This ensures consideration of all combinations. Note here the possible inefficiency for generating many combinations and then discarding those that are not required.

2. Avoidance- Optimized Cartesian Product:

   This approach improves the performance of a simple Cartesian product (discussed above) by discarding redundant or useless combinations during the Cartesian product, explained as follows. At each iteration, before adding to R a coverage C_j from bucket B_i, we check all the subgoals already covered by coverages in R, and ensure there is no overlap. If there is an overlap, we discard the combination of coverages in R and avoid checking all combinations built based on indexes $\{I_1, \ldots, I_i\}$. For that, instead of incrementing I_P starting with P=n, we set P=i, where i is the index of the bucket B_i. Based on this, we skip the combinations that have the same overlapping coverages. Minicon uses this approach to perform the Cartesian product.

3. Prevention- Pattern based Cartesian Product:

   As proposed in this thesis, this approach is based on the idea of occurrence patterns. It assigns to query Q the number $2^n - 1$, where n is the number of subgoals in Q, and assigns an occurrence identifier to every coverage C_i that is based on the subgoals it contains and its position in the query body. For instance, in Example 3.3, occurrence identifiers assigned to C_1 and C_2 would be 6 ($= 2^2 + 2^1$), and 3 ($= 2^1 + 2^0$), respectively. Let N be the sorted list of all identifiers (with no duplicate identifier). Then calling $FindPatterns(2^n - 1, N, P)$ returns the list of patterns based on available coverages in P. For each combination pattern in P, we then create buckets and perform a simple Cartesian product. Since a combination pattern has no overlap, no further checking is required. For example, consider the coverages listed in Figure 3.1, for which we break down the original bucket structure into smaller buckets and perform 5 different Cartesian products (since there are 5 different pattern

combinations) listed as follows.

(a) $\mathbb{P}_1 = \{\{7\}\}$: No need to apply Cartesian product since there is a single bucket $[C_7]$ for C_7

(b) $\mathbb{P}_2 = \{\{3\},\{4\}\}$: Cartesian product of two buckets: $[C_3] \times [C_4]$

(c) $\mathbb{P}_3 = \{\{6\},\{1\}\}$:Cartesian product of two buckets: $[C_6] \times [C_1]$

(d) $\mathbb{P}_4 = \{\{5\},\{2\}\}$: Cartesian product of two buckets:$[C_5] \times [C_2]$

(e) $\mathbb{P}_5 = \{\{4\},\{2\},\{1\}\}$: Cartesian product of three buckets: $[C_4] \times [C_2] \times [C_1]$

Note that for a query with three subgoals, the Bucket algorithm (based on Detection and Recovery), and Minicon (based on Avoidance) need to perform a Cartesian product of size 4^3, whereas the cost in our pattern-based algorithm is reduced to 5 operations.

3.3.3 Correctness

In this section, we discuss the correctness of our rewriting algorithm in generating maximally contained rewriting on the basis of the Minicon algorithm. For this, we note that Minicon algorithm generates rewriting in two phases: (1) finding MCDs, and (2) combining them to generate the rewriting, and show that the output of each phase in our Pattern-based algorithm matches with the output of the corresponding phase in Minicon. Next, we compare the corresponding phases of Minicon and Pattern-based algorithms.

As mentioned earlier, coverage extends the notion of MCD. Looking at the definition of MCDs, we can see that a MCD has exactly the same properties of a basic coverage, i.e., all joint attributes are accessible through the view, and if any subgoal is removed from MCD the condition for joint variable is no longer satisfied.

Considering the second phase of rewriting, finding non-overlapping combination of coverages, we remark that the only difference between Minicon and Pattern-based algorithm is that Minicon takes an Avoidance approach while our Pattern-based uses Prevention. Regardless of their approach, as explained in section 3.3.2, the properties of the combinations found in these approaches are identical, i.e., no two coverages in a combination have common subgoal. Based on this, we note that they both generate the same set of rules for rewriting. This explains why our query rewriting algorithm generates maximally contained rewriting.

3.3.4 Complexity of Query Rewriting

In this section, we show that the upper bound for query rewriting could be expressed in terms of the Bell numbers ([Rio58]).

Theorem 3.1 *Let Q be a query and V be a set of views. Then the upper bound for the complexity of the problem of determining whether there exists a rewriting of Q that uses V is $O\Big((\frac{0.792n}{ln(n+1)})^n\Big)$, where n is the number of subgoals in the query.*

Proof *Query rewriting has two phases. In the first phase, it finds coverages for which it searches for mappings from subgoals in the query Q to those in the views. The complexity of this phase is exponential in the number of subgoals in Q. Assume a is the number of different predicates in Q, and b and c are the number of subgoals of each predicate in Q and V_i, respectively. Then the number of possible mappings from Q to V_i is $(c^b)^a$. The reason is that every subgoal in Q can be mapped to c subgoals in V_i, and since for every predicate p there are b subgoals of that predicate, the number of possible mappings for that predicate is c^b. Since the number of different predicates in Q is a, the total number of possible mappings would be $(c^b)^a = c^{ab}$.*

In the second phase of rewriting, we use a prevention approach for combining the coverages. Assuming that Q has n subgoals, the number of different sets of buckets

formed in the worst case is the n^{th} Bell number, $B(n)$. To determine whether a rewriting exists, every bucket has to have at least one coverage, and hence we assume the number of coverages in each bucket is 1. The cost of Cartesian products of these sets of buckets can be defined based on the number of rules generated which is $B(n)$. Since the growth of the Bell number is much faster than c^{ab}, the complexity of the second phase is more than the first phase, and hence we may ignore the complexity of the first phase.

As a result, the upper bound for testing whether there exists a rewriting of Q that uses V is the upper bound of the Bell numbers which has been shown to be $(\frac{0.792n}{ln(n+1)})^n$ [Rio58]. ■

Next, we classify the rewriting algorithms based on their approach to generate coverages and show its affect on the number of subgoals in the generated rewriting.

3.4 Classification of Approaches to Query Rewriting

In this section, we classify query rewriting algorithms into two approaches and compare these approaches based on the quality of the rewriting. For this, we first define the notion of *area* of a rewriting, that we use as a quality measure to identify the rewriting that is less expensive to execute.

Definition 3.8 (Area of rewriting) *The area of a rewriting is the total number of subgoals in the bodies of the rules it contains.*

Note that area of a rewriting is a language dependent issue, which in our case is the union of conjunctive queries.

As explained in Section 3.1.1, there are two possible approaches to build coverages, bottom-up and top-down approaches, which define the rewriting approach, explained as follows.

1. Top-Down: Used in [KS05a] and Treewise [MS08].

2. Bottom-Up: Used in Bucket algorithm [LRO96], Inverse Rule [Qia96, DG97a], Minicon [PL00], and our Pattern-based algorithm [KS09].

The differences between the bottom-up and the top-down approaches are as follows.

1. In the first phase of rewriting, to generate a coverage, a bottom-up algorithm usually starts with a single subgoal coverage and adds subgoals to it until it becomes a useful coverage; however, a top-down algorithm starts with a coverage containing all subgoals and reduces it through breaking it until it finds a set of useful coverages which requires no further breaking.

2. In the first phase of rewriting, a bottom-up algorithm can generate coverages without considering other views and coverages. However, a top-down algorithm needs to consider other coverages/views because it might need to break a coverage into smaller ones so that they can participate in a rewriting combined with coverages from other views. The reason is that if a top-down algorithm keeps only large coverages, they might not form all possible rewritings, and hence, we do not obtain a maximally contained rewriting.

3. When combining coverages (phase 2), since a top-down approach usually deals with larger coverages, the size of each rule in the rewriting would not be more than the corresponding rule in a rewriting in a bottom-up approach. In fact, the top-down approach can always find the rules with the least number of subgoals.

This is an advantage of top-down paying off the extra processing in the first phase of rewriting to check all views/coverages. In fact, bottom-up can also reach this least number of subgoals in the result of rewriting by exploiting query minimization as a post-processing phase. We discuss the query minimization and its application in query rewriting in Chapter 4.

As a result, both top-down and bottom-up approaches can generate rules that are minimal in size. Top-down determines this optimized rewriting when finding coverages in the first phase, while bottom-up finds it in a post-processing step after combining the coverages. A question at this point is whether there is any preference between the two approaches.

Example 3.7 *Consider the following query Q and views V_1 and V_2.*

Q : $h(X,W)$:- $r(X,Y), s(Y,Z), t(Z,W)$.

V_1 : $v_1(A,B,C)$:- $r(A,B), s(B,C)$.

V_2 : $v_2(D,E,F)$:- $s(D,E), t(E,F)$.

Bottom-up approach and top-down approach create different sets of coverages and form different buckets. As shown in Figure 3.3, bottom-up creates $\{C_1, C_2, C_3, C_4\}$ as the set of coverages, where C_1 covers subgoal $\{r\}$, C_2 covers $\{s\}$, C_3 covers $\{s\}$, and C_4 covers $\{t\}$.

As shown in Figure 3.4, top-down creates $\{C_1', C_2', C_3', C_4'\}$ as the set of coverages, where C_1' covers $\{r,s\}$, C_2' covers $\{r\}$, C_3' covers $\{s,t\}$, and C_4' covers $\{t\}$.

Although the rewritings generated in bottom-up and top-down approaches are equivalent, they may differ in the number of subgoals in the rule bodies.

To generate a rule in a rewriting, we combine the coverages so that they cover the entire query body. As shown in Figures 3.3 and 3.4, a bucket is defined for every subgoal in the query, and coverages are placed in these buckets according to the subgoals they cover.

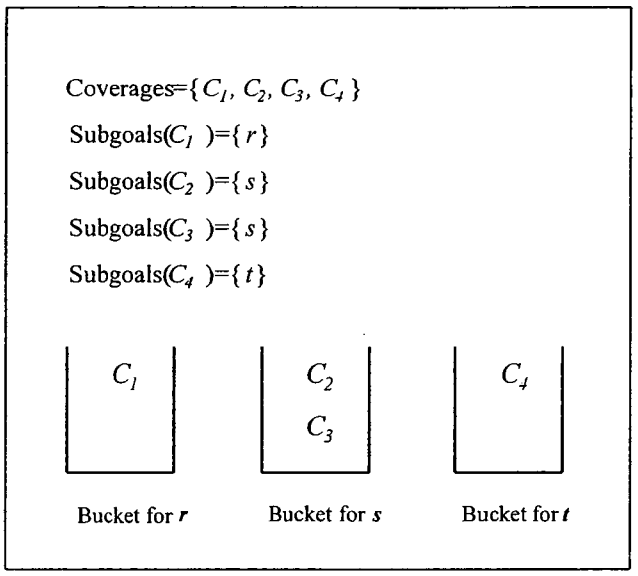

Figure 3.3: Coverages and Buckets in bottom-up for Example 3.7

Combining these buckets yields the following rewriting:

$$Q: \quad h(X,W) \text{ :- } V_1(X,Y,C), V_1(A,Y,Z), V_2(D,Z,W).$$

$$h(X,W) \text{ :- } V_1(X,Y,C), V_2(Y,Z,F), V_2(D,Z,W).$$

When comparing the two approaches, we can see that a top-down algorithm, in general, generates rewriting with smaller area compared to a bottom-up algorithm. For instance, the areas of the rewritings for Example 3.7 in Figures 3.3 and 3.4 are 4 and 6, respectively. When the size of a given database D is huge, this could result in a significant difference in the execution cost. So, the extra cost spent in top-down approach may pay off during evaluation of rewriting on D. Based on what we discussed so far, we can see that in general, bottom-up outperforms top-down in rewriting time. However, quality of the rewritings in top-down is usually better than that of bottom-up.

Next, we show that existence of redundant subgoals in the input of rewriting results in redundant rules in the output of rewriting. For that, consider the following

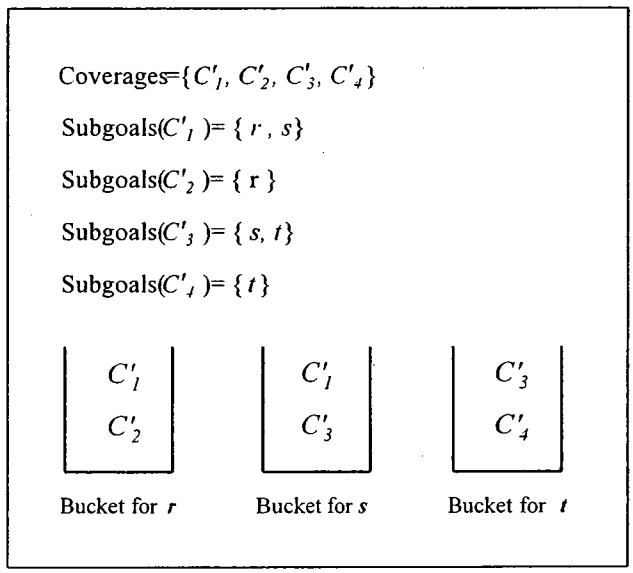

Figure 3.4: Coverages and Buckets in top-down for Example 3.7

Combining these buckets yields the following rewriting:

$$Q: \quad h(X,W) \text{ :- } V_1(X,Y,Z), V_2(D,Z,W).$$

$$h(X,W) \text{ :- } V_1(X,Y,C), V_2(Y,Z,W).$$

query Q_1 and views V_1 and V_2.

Q_1 : $h(X)$:- $r(X,Y), r(X,Z), s(Z,W)$.

V_1 : $v_1(A,B,C)$:- $r(A,B), s(B,C)$.

V_2 : $v_2(D,E)$:- $r(D,E)$.

We want to rewrite Q_1 using V_1 and V_2. In the first phase of the rewriting, we find the following coverages for the subgoals in Q_1.

$C_1 = \langle \{r(X,Y)\}, \{X/A, Y/B\}, v_1(A,B,C), \{\} \rangle$.

$C_2 = \langle \{r(X,Z)\}, \{X/A, Z/B\}, v_1(A,B,C), \{\} \rangle$.

$C_3 = \langle \{s(Z,W)\}, \{Z/B, W/C\}, v_1(A,B,C), \{\} \rangle$.

$C_4 = \langle \{r(X,Y)\}, \{X/D, Y/E\}, v_2(D,E), \{\} \rangle$.

$C_5 = \langle \{r(X,Z)\}, \{X/D, Z/E\}, v_2(D,E), \{\} \rangle$.

Coverages C_1 , C_2 and C_3 are based on V_1, and coverages C_4 and C_5 are based on V_2.

In the second phase of rewriting, we combine the coverages using simple Cartesian product approach. For that, we consider one bucket for each subgoal in Q_1, and assign the coverages to the corresponding buckets.

- Bucket B_1 for $r(X, Y)$ containing coverages C_1 and C_4, i.e., $B_1 = \{C_1, C_4\}$.
- Bucket B_2 for $r(X, Z)$ containing coverages C_2 and C_5, i.e., $B_2 = \{C_2, C_5\}$.
- Bucket B_3 for $s(Z, W)$ containing coverages C_3, i.e., $B_3 = \{C_3\}$.

The following combinations are the result of performing Cartesian product on these buckets.

1. $\{C_1, C_2, C_3\}$
2. $\{C_1, C_5, C_3\}$.
3. $\{C_4, C_2, C_3\}$.
4. $\{C_4, C_5, C_3\}$.

Each of the above combination results in a rule in the rewriting, in the order shown as follows.

$$
\begin{aligned}
Q': \quad & h(X) :\text{-} v_1(X, Y, N_1), v_1(X, Z, N_2), v_1(N_3, Z, W). \\
& h(X) :\text{-} v_1(X, Y, N_4), v_2(X, Z), v_1(N_5, Z, W). \\
& h(X) :\text{-} v_2(X, Y), v_1(X, Z, N_6), v_1(N_7, Z, W). \\
& h(X) :\text{-} v_2(X, Y), v_2(X, Z), v_1(N_8, Z, W).
\end{aligned}
$$

As we can see, there are 4 rules in the output of the rewriting. To see the importance of minimization in the quality of rewriting, next, we minimize the query and views in this example before performing the rewriting. Since views V_1 and V_2

do not have repeated subgoal so they are already minimized. Consider the following query Q_2 which has no repeated subgoals and equivalent to Q_1 (there are containment mapping from Q_1 to Q_2 and vice versa).

Q_2 : $h(X)$:- $r(X,Z), s(Z,W)$.

V_1 : $v_1(A,B,C)$:- $r(A,B), s(B,C)$.

V_2 : $v_2(D,E)$:- $r(D,E)$.

In the first phase of the rewriting of Q_2, we find the following coverages for the subgoals in Q_2 based on view V_1 and V_2.

$C_1 = \langle \{r(X,Z)\}, \{X/A, Z/B\}, v_1(A,B,C), \{\} \rangle$.

$C_2 = \langle \{s(Z,W)\}, \{Z/B, W/C\}, v_1(A,B,C), \{\} \rangle$.

$C_3 = \langle \{r(X,Z)\}, \{X/D, Z/E\}, v_2(D,E), \{\} \rangle$.

Coverages C_1 , C_2 are based on V_1, and coverage C_3 is based on V_2.

Next, we consider one bucket for each subgoal in Q_2, and assign the coverages to the corresponding buckets.

- Bucket B_1 for $r(X,Z)$ containing coverages C_1 and C_3, i.e., $B_1 = \{C_1, C_3\}$.
- Bucket B_2 for $s(Z,W)$ containing coverage C_2, i.e., $B_2 = \{C_2\}$.

The following combinations are the result of performing Cartesian product on these buckets.

1. $\{C_1, C_2\}$
2. $\{C_3, C_2\}$.

Each of the above combination results in a rule in the rewriting in the same order, shown as follows.

Q_2' : $h(X)$:- $v_1(X,Z,N_1), v_1(N_2,Z,W)$.

$h(X)$:- $v_2(X,Z), v_1(N_3,Z,W)$.

As we can see, the number of rules generated for the minimized input is less than the number of rules generated for the case that input had redundant subgoals.

We also note that while rewriting Q'_2 has the least number of rules, its area (=4 subgoals) is not optimal. Minimizing the rules in Q'_2 generates the optimal rewriting in which the area is 3 subgoals, shown as follows.

Q'_3 : $h(X) :- v_1(X, Z, W).$

$h(X) :- v_2(X, Z), v_1(N_3, Z, W).$

3.5 Summary

In this chapter, we studied query rewriting in the context of standard conjunctive queries. We investigated the two phases of rewriting, finding coverages and combining them, compared different approaches for each phase, and focused on the combining phase as the more expensive part of the rewriting. We proposed a pattern-based algorithm that uses prevention approach to combine coverages (i.e., preventing from having overlapping coverages in the buckets).

Furthermore, we proposed a formula to determine the number of rules in the rewriting based on the Stirling numbers. This is particularly of practical importance because given a query and set of views, one can know the number of rules in the rewriting before actually generating it, and hence plan available resources required for generating the rewriting.

It has been shown that query rewriting usually generates rules with redundant subgoals. In fact, it is possible that both the input query and views contain redundant subgoals, in which case, rewriting performs unnecessary computations. In the next chapter, we address these issues by studying minimization of conjunctive queries and exploiting it to minimize both input and output of the rewriting algorithm, and hence improve efficiency and scalability of our rewriting algorithm.

Chapter 4

Minimization of Conjunctive Queries

In this chapter, we revisit the problem of query minimization as both pre-processing and post-processing phases in query rewriting (i.e., applying minimization to both input and output of query rewriting). We investigate this in the context of standard conjunctive queries, and propose a novel query minimization algorithm that uses special endomorphisms together with some heuristics to identify and remove redundant subgoals in an iterative approach.

4.1 Introduction

Nowadays, there are many applications that generate queries automatically. Such queries may have redundant subgoals which add unnecessary overhead when processing the queries. Identifying redundant subgoals is a minimization problem which is based on containment ([CM77]). Hence finding efficient heuristics to identify such redundancies is of practical importance.

Some applications that generate queries with potentially redundant subgoals include view expansion, translating Xqueries to SQL, and query rewriting [FCS00, CR00, Lev01, KS09, KS05a]. The following example illustrates the importance of minimization after view expansion.

Example 4.1 *Consider the following query Q_1 and view definitions V_1 and V_2, over the base relations $r(A, B)$ and $s(A, B)$.*

Q_1: $h(X)$:- $v_1(X, Y), v_2(Y, X)$.

V_1: $v_1(A, B)$:- $r(A, A), s(A, B)$.

V_2: $v_2(A, B)$:- $r(A, B), s(B, B)$.

In order to evaluate Q_1, we expand it using the view definitions and generate query Q_2 which is equivalent to Q_1, as follows.

Q_2: $h(X)$:- $r(X, X), s(X, Y), r(Y, X), s(X, X)$.

Now, we can evaluate Q_2 because it includes only base relations. For that we need to perform three join operations, however, if we apply query minimization on Q_2, we get the following query Q_3 that is equivalent to Q_2 (and hence Q_1) but requires only one join operation for evaluation.

Q_3: $h(X)$:- $r(X, X), s(X, X)$.

Definition 4.1 (Query Minimization) *Given a standard conjunctive query Q, query minimization defines a conjunctive query Q' that is equivalent to Q and has the minimum number of subgoals in the body.*

Intuitively, given a query Q, query minimization removes from Q all *redundant subgoals*, defined as follows.

Definition 4.2 (Redundant Subgoal) *A subgoal g_i in the body of a conjunctive query Q is redundant if it can be removed from Q without changing the meaning of Q.*

Query minimization has been studied in different contexts including optimization of queries in relational databases [CM77, Ull89, Mai83, KS02, FCS00], Datalog [LS92], and query rewriting [LMSS95, PL00, Ull00, KS09].

Although query minimization is not a new problem, there has been little focus on its practical efficiency [KS02]. The reason is the complexity of the problem and the fact that there was not much space for improvement in queries as they were often composed manually by human experts and hence little or no opportunity for minimization. Today, this situation has changed, and many applications automatically generate queries which may include redundant subgoals.

4.2 Related Work

Chandra and Merlin ([CM77]) showed that for every conjunctive query Q, a minimal query exists and is unique up to renaming of variables. They also showed that finding a minimal query is exponential in the number of subgoals in the input query. Maier [Mai83] and Ullman [Ull89] showed that minimized query can be found by finding an endomorphism on the query itself. However, the algorithms provided are not efficient without heuristics as the search space is huge. Levy et al. [LMSS95] studied this problem in the context of query rewriting using views. They introduced a polynomial time algorithm to remove some but not all the redundant subgoals, resulting in a rewriting that is not minimal, in general. A similar version of this algorithm is used in [PL00].

Kunen and Suciu [KS02] studied this problem and introduced an algorithm based on a set of heuristics for query minimization and show its efficiency for large queries with hundreds of joins. To the best of our knowledge their work is the only work in the literature that studies the performance of query minimization and provides experimental results for large queries. Their solution approach defines a canonical

database for a given query Q, which basically includes a tuple $\bar{X}$ in relation r for every subgoal $r(\bar{X})$ in Q. At each step in an iterative process, they remove one tuple from the database and evaluate the query. If the query result did not change, it means that the tuple is redundant. This process repeats until all redundant subgoals are identified and removed. They showed that evaluating the query on the canonical database without any optimization is not efficient. To address this issue, they implement the following heuristics.

1. Local optimization on query plan: Convert the query into a tree (query plan) where the goal is to find a tree with the smallest width.

2. Randomization: Randomly repeat the first heuristics and choose the tree with the smallest width.

3. Early Termination: The algorithm may be terminated in the following cases.

   (a) If at any point in the query execution, an intermediate result is found to be empty, the end result must be empty too and hence the execution terminates.

   (b) In this algorithm, query is executed on a very similar database repeatedly and there is a high chance to see internal tables that are not changed from iteration to iteration. Based on this, the result of join operations can be cached, and if the input relations to each join are not changed, the result of the previous execution could be reused.

4. Table Pruning: Consider a join operation $E \bowtie F$. If there are rows in E which do not join with any row in F, or vice versa, then they can be safely discarded for the join.

5. Incremental Evaluation: Let $D = E \bowtie F$ be an internal table which records the

result of the join of E and F. If E and F are updated to E' and F', we can either calculate the join using $D' = E' \bowtie F'$, or we can compute the change in D, ΔD, as $\Delta D = (E \bowtie \Delta F) \cup (\Delta E \bowtie F) - (\Delta E \bowtie \Delta F)$. Although this requires three joins, but when ΔE and ΔF are small relations, computing ΔD might be faster than computing $E' \bowtie F'$.

It has been shown through experiments with different types of queries, that their minimization algorithm is efficient for queries with hundreds of joins and is scalable [KS02].

In the next section, we explain our query minimization technique.

4.3 Query Minimization

It has been shown that to minimize a conjunctive query Q one can use an endomorphism, i.e., a proper mapping from query Q to itself [Mai83, Ull89]. This is explained as follows.

(a) Applying "proper" *endomorphisms* on input query generates an equivalent query in which some occurrences of subgoals are identical.

(b) Removing redundant occurrences generates a shorter query which is equivalent to the original query.

(c) Iteratively, finding and applying such endomorphisms and removing all redundant subgoals would result in the minimized query.

The following example illustrates the steps of minimization in this approach.

Example 4.2 *Consider the following conjunctive query Q.*

Q: $h() :\text{-}\ r(A,A,B),\ r(D,D,A),\ r(D,C,F),\ r(C,D,G),\ r(C,E,G).$

To identify redundant subgoals, we take two steps: first, we find an endomorphism that generates two identical occurrences of the same subgoal and then test if applying the endomorphism maintains the equivalence. These steps are explained as follows.

1. *Finding an endomorphism. For this, let us take $r(\bar{X}_1) = r(D,D,A)$ and $r(\bar{X}_2) = r(C,D,G)$ and apply the mapping $\mu_1 = \bar{X}_2 \rightarrow \bar{X}_1 = \{C/D, D/D, G/A\}$ on Q as the endomorphism, where C/D in μ_1 means that variable C in $r(C,D,G)$ is mapped to variable D in $r(D,D,A)$. Applying μ_1 on Q results in the following query.*

   Q_1: $h() :\text{-}\ r(A,A,B),\ r(D,D,A),\ r(D,D,F),\ r(D,D,A),\ r(D,E,A).$

   Note that, in general, there are $n\,(n-1)$ possibilities for choosing pairs of $r(\bar{X}_1)$ and $r(\bar{X}_2)$, where n is the number of subgoals with identical predicate name. Using some heuristics we can do this faster. This is important because the minimization is iterative and the earlier the redundant subgoals are identified, the faster minimization terminates.

   Initially, Q_1 has two occurrences of subgoal $r(D,D,A)$, of which we remove one, as follows.

   Q_1: $h() :\text{-}\ r(A,A,B),\ r(D,D,A),\ r(D,D,F),\ r(D,E,A).$

2. *Checking equivalence $Q_1 \equiv Q$. Since we already found the containment mapping μ_1 from Q to Q_1, to show the equivalence, we only need a containment mapping from Q_1 to Q.*

   Since this is a special case of containment test, some heuristics could help determining the containment mapping faster. In this example, we find containment

mapping $\mu_2 = \{A/A, B/B, D/D, F/A, E/D\}$ *which means* $Q_1 \equiv Q$. *We can continue this process iteratively until no more subgoal is removed. The following query* Q' *is the minimized query equivalent to* Q, *which requires only one join to evaluate.*

Q': $\quad h() \text{ :- } r(A, A, B),\ r(D, D, A).$

Next, we review some background information and explain how to find endomorphisms that help minimize conjunctive queries. Moreover, we identify some heuristics to speed up the minimization process.

4.3.1 Identifying Redundant Subgoals

In this section, we propose an algorithm together with heuristics for identifying redundant subgoals. We first review and define the concepts we need in our discussion.

Recall that a conjunctive query is an expression of the form:

Q: $\quad h(\bar{X}) \text{ :- } r_1(\bar{X}_1), \ldots, r_n(\bar{X}_n),$

where $h(\bar{X})$ is the head, and predicate h does not appear in the body of Q. $\bar{X}_i$ is a list of variables, and $r_i(\bar{X}_i)$ is a subgoal in the body.

We assume each r_i is a base relation (and not a view). We use $r(\bar{X})[i]$ to denote the i^{th} variable in $r(\bar{X})$.

As mentioned in Example 4.2, our approach to query minimization is iterative and in each iteration we take two steps: (1) find an endomorphism, and (2) test if it maintain the equivalence.

It is important to note that repetition of variables in a subgoal $r(barX)$ imposes restriction on that subgoal. Intuitively, when searching for minimizing endomorphism, we cannot map subgoals that are more restricted to subgoals that are less restricted. This is because the resulting query would not be equivalent to the original query. For

this reason, and in order to compare subgoals based on the restrictions imposed by their repeated variables, we define the notion of *Pattern Sequence* as follows.

Definition 4.3 (Pattern Sequence) *Let Q be a conjunctive query Q, $r(\bar{X})$ be a subgoal in Q. For each variable A in a $r(\bar{X})$, the pattern sequence of A in $\bar{X}$, denoted $\mathbb{P}_A(r(\bar{X}))$, is a binary sequence that has a '1' at position i if $r(\bar{X})[i] = A$, and '0', otherwise.*

For simplicity, we may use $\mathbb{P}_A(\bar{X})$ instead of $\mathbb{P}_A(r(\bar{X}))$. For example, for subgoals $t(W,Y,Z)$, we have that $\mathbb{P}_W(r(W,Y,Z)) = \mathbb{P}_W(W,Y,Z) = (1,0,0)$.

We define a partial order $\leq$ on pattern sequences of subgoals as follows. Let S_g be the set of subgoals in Q with predicate name g. For every two subgoals $g(\bar{X}_1)$ and $g(\bar{X}_2)$ in S_g and variables A and B, we say that $\mathbb{P}_A(\bar{X}_1) \leq \mathbb{P}_B(\bar{X}_2)$ if condition $(g(\bar{X}_1)[i] = 1) \rightarrow (g(\bar{X}_2)[i] = 1)$ holds for all i. That is, for every position i in $g(\bar{X}_1)$ with '1', there is a '1' at position i in $g(\bar{X}_2)$. For example, for subgoals $t(W,Y,Z)$ and $t(X,Y,X)$, we have that $\mathbb{P}_W(W,Y,Z) = (1,0,0) \leq \mathbb{P}_X(X,Y,X) = (1,0,1)$. We also consider an extension of the logical *or* operator to a list. Formally, $\mathbb{P}_A(\bar{X}_1) \vee \mathbb{P}_A(\bar{X}_2)$ returns a binary sequence, which has a '1' at position i if the i^{th} position in $\mathbb{P}_A(\bar{X}_1)$ or in $\mathbb{P}_A(\bar{X}_2)$ is 1.

Definition 4.4 (Minimizing Substitution) *Given a query Q and a variable substitution θ, if by applying θ on Q we get two or more identical occurrences of the same subgoal, we can keep one occurrence and remove other occurrences. This results in a new query Q' that has fewer subgoals. If Q' is equivalent to Q, then Q' is a reduced equivalent of Q. We refer to such substitution θ as minimizing substitution.*

Note that such a minimizing substitution defines an endomorphism.

Lemma 4.1 *For a non-minimized conjunctive query Q, minimizing substitutions always exist.*

Proof *Consider queries Q and Q', where Q' is the minimized equivalent of Q. Since $Q' \sqsubseteq Q$, there is a containment mapping from Q to Q'. This means that there exists a containment mapping (i.e., variable substitution) from Q to Q' that can help finding a minimized equivalent query.* ■

Next, we show how to find a minimizing substitution that identifies a redundant subgoal and satisfies query equivalence.

The basic idea is as follows. We take a pair of subgoals in Q that have the same predicate name, define a mapping from one to the other as a substitution θ, apply θ on Q to get a new query Q' (which has at least one less subgoal), and test if $Q \equiv Q'$ holds. In case of equivalence, θ is a minimizing substitution.

Theorem 4.2 *Given a conjunctive query Q, finding and applying minimizing substitutions on Q in a finite number of iterations would result in a minimized query equivalent to Q.*

Proof *Assume that query Q' is the result of applying minimizing substitutions on a given query Q in which no further reduction of Q' is possible. If Q' is not the minimized query then based on Lemma 4.1 there is a substitution θ that can be applied on Q' to generate the minimized query. This contradicts with the assumption that Q' is minimized. Therefore Q' is the minimized query.*

Since every minimizing substitution is an endomorphism, and the number of endomorphisms for every query Q is finite, minimization algorithm terminates in finite steps. ■

4.3.2 Finding Minimizing Substitution

The goal of this section is to identify the conditions for finding minimizing substitutions. Consider a general form of a conjunctive query with repeated subgoals (and

potentially redundant ones), as follows.

Q: $\quad h(\bar{X}) \text{ :- } r(\bar{X}_1),\ r(\bar{X}_2),\ L$

where L indicates the rest of the subgoals in Q. Assume that the following query Q' is the result of applying a minimizing substitution $\theta : \bar{X}_1 \rightarrow \bar{X}_2$ on Q. Note that Q' has at least one subgoal less than Q. The general form of Q' is as follows.

Q': $\quad h(\bar{X}') \text{ :- } r(\bar{X}_2),\ L'.$

In order to test the equivalence of Q and Q', we test two containments: (1) $Q' \sqsubseteq Q$ and (2) $Q \sqsubseteq Q'$ ([CM77]), as follows.

1. $Q' \sqsubseteq Q$: To test this containment, we need to find a containment mapping from Q to Q'. We know that Q' is generated by applying θ on Q. In fact θ defines a mapping μ from Q to Q' as follows:

$$\mu(A) = \begin{cases} A\theta & \text{if variable } A \in \bar{X}_1 \\ A & \text{otherwise} \end{cases}$$

That is, μ is θ for those variables in Q that appear in $\bar{X}_1$, and is the identity function for the rest of the variables of Q.

Based on this, $Q' \sqsubseteq Q$ if μ is a valid containment mapping. For this, we define the following two conditions.

(a) Substitution $\theta = \bar{X}_1 \rightarrow \bar{X}_2$ is consistent, i.e., every variable is substituted by exactly one variable or a constant. This guarantees that μ is consistent. The following example illustrates a case where this condition does not hold.

Example 4.3 *Consider the query Q defined as follows.*

Q: $\quad h() \text{ :- } r(X, Y, X), r(A, B, C).$

Let $r(\bar{X}_1) = r(X, Y, X)$, *and* $r(\bar{X}_2) = r(A, B, C)$. *Then, substitution* $\theta = \{X/A, Y/B, X/C\}$ *is not consistent because* X *is mapped to two different variables,* A *and* C, *i.e.,* θ *is not a consistent substitution.*

In order to express this condition, we use pattern sequence and define Test I as follows.

Test I: $\forall\, A\ \exists\, B:\ \mathbb{P}_A(\bar{X}_1) \leq \mathbb{P}_B(\bar{X}_2)$.

Test I verifies that if variable A has appeared more than once in $\bar{X}_1$, then there is a variable B in $\bar{X}_2$ that appears at least in the same positions in $\bar{X}_2$. Intuitively, it checks if A is mapped to exactly one variable.

(b) Substitution θ should generate identical occurrences of $r(\bar{X}_2)$. In other words, according to the general form of Q', after applying θ, both $r(\bar{X}_1)$ and $r(\bar{X}_2)$ should be transformed to $r(\bar{X}_2)$. The following example shows a case where this condition is not satisfied.

Example 4.4 *Consider the following query* Q.

Q: $\quad h() \text{ :- } r(X, A, Z), r(A, B, C).$

Let $r(\bar{X}_1) = r(X, A, Z)$, *and* $r(\bar{X}_2) = (A, B, C)$.

Then, using $\theta = \{X/A, A/B, Z/C\}$, *we get the following query* Q':

Q': $\quad h() \text{ :- } r(A, B, C), r(B, B, C).$

We can see that Q' *is not in the proper form because* Q' *is supposed to include only* $r(A, B, C)$. *Therefore,* θ *is not a proper minimizing substitution and choosing* $r(\bar{X}_1) = r(X, A, Z)$ *and* $r(\bar{X}_2) = (A, B, C)$ *is not a good choice.*

In order to express this condition, we use pattern sequence and define Test

II as follows.

Test II: $\forall\, A \in \bar{X}_2 :\ \mathbb{P}_A(\bar{X}_1) \ \leq\ \mathbb{P}_A(\bar{X}_2).$

Test II verifies that all common variables of $r(\bar{X}_1)$ and $r(\bar{X}_2)$ remain unchanged after applying θ.

Next, we discuss the containment of Q in Q'.

2. $Q \sqsubseteq Q'$: To test this containment, we need a containment mapping from Q' to Q. For this, we find all possible partial mappings ρ_i's from Q' to Q, and combine them to form a containment mapping from Q' to Q. That is, we find the partial mappings for (a) $h(\bar{X}')$, (b) $r(\bar{X}_2)$, and (c) subgoals in L. Next, we discuss details of such partial mappings for each case.

   (a) Partial mapping for the head $h(\bar{X}')$:

   $$\rho_1 : h(\bar{X}') \ \rightarrow\ h(\bar{X}).$$

   Since there is only one mapping for the head, we define the consistency test for ρ_1 as follows.

   Test III: $\forall\, A\ \exists\, B :\ \mathbb{P}_A(\bar{X}') \ \leq\ \mathbb{P}_B(\bar{X})$

   Test III can confirm if ρ_1 could be used during the search for a containment mapping.

   (b) Partial mappings for the subgoal $r(\bar{X}_2)$:

   $$\rho_2 : r(\bar{X}_2) \ \rightarrow\ r(\bar{X}_1),$$
   $$\rho_3 : r(\bar{X}_2) \ \rightarrow\ r(\bar{X}_2),$$
   $$\rho_4 : r(\bar{X}_2) \ \rightarrow\ r(\bar{X}_3), \text{ where } r(\bar{X}_3) \text{ is a subgoal in } L.$$

   We do not define the consistency tests for ρ_2, ρ_3 and ρ_4 because we know

that at least ρ_3 (the identity mapping) is always satisfied.

(c) Partial mappings for subgoals L':

$\rho_5 : L' \rightarrow I$, where I is a set of subgoals in Q.

Note that since L contains a set of subgoals, unlike other partial mappings above that are determined, we actually need to find ρ_5.

To summarize, we list all the possible combinations of partial mappings. It is important to note that $Q \sqsubseteq Q'$ holds if one of the following three sets forms a containment mapping:

(1) $\mu_1 = \{\rho_1, \rho_2, \rho_5\}$

(2) $\mu_2 = \{\rho_1, \rho_3, \rho_5\}$

(3) $\mu_3 = \{\rho_1, \rho_4, \rho_5\}$.

If ρ_1 is not consistent, since it is a fixed partial mapping that is common in all these mappings, Test III fails and we conclude that $Q \not\sqsubseteq Q'$.

Next, we introduce some heuristics to test if one of the mappings μ_1, μ_2, or μ_3 is a containment mapping.

4.3.3 Proposed Heuristics

We need to find a partial mapping ρ_5 that is consistent with the rest of partial mappings in μ_1, μ_2, or μ_3. Since Q' is the result of applying an endomorphism on Q, certain subgoals could be mapped using the identity mapping. Therefore, to speed up the process, we exclude such subgoals in our search. For this, we define the notion of *Connected subgoals* as follows.

Definition 4.5 (Connected Subgoals) *Let Q be a conjunctive query, and $r(\bar{X})$ and $s(\bar{Y})$ be subgoals in Q. Subgoals $r(\bar{X})$ and $s(\bar{Y})$ are connected if they share some variables or there exists a subgoal $t(\bar{Z})$ in Q that is connected to $r(\bar{X})$ and $s(\bar{Y})$.*

The connected subgoals for a subgoal $r(\bar{X})$, denoted $\mathbb{C}_{r(\bar{X})}$, is the set of all subgoals in a query Q that are connected to $r(\bar{X})$.

In the search for the partial mappings in ρ_5, we focus only on the set of subgoals in Q' that are connected to $r(\bar{X}_1)$ through some join variables. Other subgoals can be mapped using the identity function. The following example illustrates this. Consider the query Q defined as follows.

Q: $\quad h(A)$:- $r(X,Y,Z), s(Z,W), r(A,B,C), s(C,D), t(D,E)$.

Let $r(\bar{X}_1) = r(X,Y,Z)$ and $r(\bar{X}_2) = (A,B,C)$. Then, applying $\theta = \{X/A, Y/B, Z/C\}$ on Q would generate the following query Q'.

Q': $\quad h(A)$:- $s(C,W), r(A,B,C), s(C,D), t(D,E)$.

Here, the partial mapping from Q' to Q for the subgoals $r(A,B,C)$, $s(C,D)$, and $t(D,E)$ in Q' is based on identity. The reason is that these subgoals are not joined directly or indirectly with $r(X,Y,Z)$. Finding such subgoals is polynomial and can reduce the search space significantly. In this example, after identifying the subgoals that use the identity mapping, the search is limited to $s(C,W)$. That is, we need to find a partial mapping that maps $s(C,W)$ in Q' to some subgoal in Q. The partial mapping for $s(C,W)$ is $\{C/C, W/D\}$, based on which the containment mapping from Q' to Q is $\{A/A, C/C, W/D, B/B, D/D, E/E\}$.

Based on this, we define Test IV that checks if μ_1, μ_2, or μ_3 form a containment mapping from Q' to Q.

Test IV: Let θ be a substitution that maps $\bar{X}_1$ to $\bar{X}_2$. To find a valid mapping that satisfies the containment test, we reduce the search space as follows.

1. *Find the Source: Compute* $\mathbb{C}_{r(\bar{X}_1)}$ *and apply the substitution* θ *on the result. We refer to the result as Source, i.e., Source=* $\mathbb{C}_{r(\bar{X}_1)}\theta$ which is a set of subgoals in Q'.

2. *Find the Target: Compute* $\mathbb{C}_{r(\bar{X}_2)}$. *We refer to the result as Target, i.e., Target=* $\mathbb{C}_{r(\bar{X}_2)}$ which is a set of subgoals in Q.

3. *Find a mapping from variables in Source to those in Target.*

Even though these steps might reduce the search space, but finding mapping in Test IV, in general, is exponential in the number of subgoals in L'. To speed up Test IV, we consider the following heuristics.

1. Before minimization, group the subgoals in the query body based on their predicate name, and sort groups ascendingly based on the number of variables.

2. During test IV, when searching for containment mapping from Q' to Q, the mappings μ_2, μ_1, and μ_3 are verified in that order. This is because, intuitively, μ_2 is less likely to have conflict compared to other mappings, and giving it a higher priority is often expected to result in early termination of the containment test.

3. During test IV, when we combine the partial mappings of different subgoals in Source to subgoals in Target, we use a breadth-first search approach.

It is important to note that testing containment of Q in Q' in the context of query minimization is a restricted form of the containment problem because Q' itself is the result of applying an endomorphism on Q. As shown in our experiments, exploiting these heuristics results in considerable speed up.

The ideas above are formally stated as a minimization algorithm, proposed next.

4.4 Our Proposed Algorithm

In this section, we introduce our minimization algorithm.

Algorithm QS (Input: Q, Output: Q')

Begin

1- Group subgoals based on predicate names and sort them based on the arities of the predicates.

2- For every group g, and for every pair of subgoals $r(\bar{X}_1)$ and $r(\bar{X}_2)$ in g, perform tests I, II, III, and IV in that order to verify if $\theta : \bar{X}_1 \rightarrow \bar{X}_2$ is a minimizing substitution. If yes, apply θ on the query.

3- Repeat step 2 until there is no minimizing substitution.

4- Return Q' as the minimized query.

End.

The following example illustrates the steps of our minimization algorithm.

Example 4.5 *Consider again the query Q in example 4.2.*

Q: $h() :\text{-}\ r(A, A, B), r(D, D, A), r(D, C, F), r(C, D, G), r(C, E, G).$

There is only one predicate in the body of Q, so there is only one group of subgoals. We start step 2 by taking pair $r(C, D, G)$ and $r(D, D, A)$ as $r(\bar{X}_1)$ and $r(\bar{X}_2)$, respectively, and check if $\theta = \{C/D, D/D, G/A\}$ is a minimizing substitution. For this, we perform tests I, II, III, and IV as follows.

Iteration #1

Test I *is satisfied because the following conditions hold.*

(1) $\mathbb{P}_C(r(C, D, G)) \leq \mathbb{P}_D(r(D, D, A))$

(2) $\mathbb{P}_D(r(C, D, G)) \leq \mathbb{P}_D(r(D, D, A))$

(3) $\mathbb{P}_G(r(C,D,G)) \leq \mathbb{P}_A(r(D,D,A))$

Test II *is satisfied because the following conditions hold.*

(1) $\mathbb{P}_D(r(C,D,G)) \leq \mathbb{P}_D(r(D,D,A))$

(2) $\mathbb{P}_A(r(C,D,G)) \leq \mathbb{P}_A(r(D,D,A))$

Test III *is satisfied because query head has no attribute.*

Test IV *is satisfied because* $\mu_2 = \{\rho_1, \rho_3, \rho_5\}$ *is a containment mapping from* Q' *to* Q*, explained as follows.*

1. *Find the source of mapping by applying* θ *on* $\mathbb{C}_{r(C,D,G)}$.
   $\mathbb{C}_{r(C,D,G)} = \{r(A,A,B), r(D,D,A), r(D,C,F), r(C,D,G),\ r(C,E,G)\}$. *Therefore, Source=* $\mathbb{C}_{r(C,D,G)}\theta = \{r(A,A,B), r(D,D,A), r(D,D,F),\ r(D,E,A)\}$.

2. *Target=* $\mathbb{C}_{r(D,D,A)} = \{r(A,A,B),\ r(D,D,A),\ r(D,C,F), r(C,D,G), r(C,E,G)\}$.

3. *We see that* $\mu_2 = \{\rho_1, \rho_2, \rho_5\} = \{A/A, B/B, D/D, F/A, E/D\}$ *can map subgoals in Source to subgoals in Target. So we conclude that* $r(C,D,G)$ *is redundant, and to remove it, we apply the mapping* $(C,D,G) \rightarrow (D,D,A)$ *on* Q*. This results in the following intermediate query with fewer subgoals in the body than* Q *which we use as the input query for the next iteration.*

Q: $\quad h() :\text{-}\ r(A,A,B),\ r(D,D,A),\ r(D,D,F),\ r(D,E,A).$

Iteration #2

In the second iteration, we consider $r(D,D,F)$ *and* $r(D,D,A)$ *as* $r(\bar{X}_1)$ *and* $r(\bar{X}_2)$*, respectively. We then check if* $\theta = \{D/D, F/A\}$ *is a minimizing substitution. For this, we perform tests I, II, III, and IV as follows.*

Test I *is satisfied because the following conditions hold.*

(1) $\mathbb{P}_D(r(D,D,F)) \leq \mathbb{P}_D(r(D,D,A))$

(2) $\mathbb{P}_F(r(D,D,F)) \leq \mathbb{P}_D(r(D,D,A))$

Test II *is satisfied because the following conditions hold.*

(1) $\mathbb{P}_D(r(D,D,F)) \leq \mathbb{P}_D(r(D,D,A))$

(2) $\mathbb{P}_A(r(D,D,F)) \leq \mathbb{P}_A(r(D,D,A))$

Test III *is satisfied because query head has no attribute.*

Test IV *is satisfied because* $\mu_2 = \{\rho_1, \rho_3, \rho_5\}$ *is a containment mapping from* Q' *to* Q*, explained as follows.*

1. $\mathbb{C}_{r(D,D,F)} = \{r(A,A,B), r(D,D,A), r(D,D,F), r(D,E,A)\}$. *Thus, Source=* $\mathbb{C}_{r(D,D,F)}\theta = \{r(A,A,B), r(D,D,A), r(D,E,A)\}$.

2. *Target=*$\mathbb{C}_{(D,D,A)} = \{r(A,A,B), r(D,D,A), r(D,D,F), r(D,E,A)\}$.

3. *Here, the breadth-first search finds the identity mapping in the first try that maps Source to Target.*

As a result, we conclude that $r(D,D,F)$ *is redundant. To remove it, we apply the mapping* $(D,D,F) \rightarrow (D,D,A)$ *on* Q*. The resulting query is as follows.*

Q: $h()$:- $r(A,A,B)$, $r(D,D,A)$, $r(D,E,A)$.

Iteration #3

Next, let us consider $r(D,E,A)$ *and* $r(A,A,B)$ *as* $r(\bar{X}_1)$ *and* $r(\bar{X}_2)$*, respectively. That is,* $\theta = \{D/A, E/A, A/B\}$*. Here, Test I succeeds but Test II fails because the following condition is not satisfied.*

$$\mathbb{P}_A(r(D,E,A)) = (0,0,1) \not\leq \mathbb{P}_A(r(A,A,B)) = (1,1,0).$$

As a result, θ *is not a minimizing substitution.*

Iteration #4

Next, we consider $r(D, E, A)$ *and* $r(D, D, A)$ *as* $r(\bar{X}_1)$ *and* $r(\bar{X}_2)$*, respectively. Accordingly,* $\theta = \{D/D, E/D, A/A\}$*. Here, Tests I, II, III and IV succeed. We illustrate the detail for Test IV as follows.*

1. $\mathbb{C}_{r(D,E,A)} = \{r(A, A, B),\ r(D, D, A),\ r(D, E, A)\}$*, therefore,*
   Source= $\mathbb{C}_{r(D,E,A)}\theta = \{r(D, D, A), r(A, A, B)\}$.

2. *Target=* $\mathbb{C}_{r(D,D,A)} = \{r(A, A, B),\ r(D, D, A),\ r(D, E, A)\}$.

3. *Here, the breadth-first search finds the identity mapping in the first try. As a result,* (D, E, A) *is redundant and we apply the mapping* $(D, E, A) \rightarrow (D, D, A)$ *on the query. This results in the following query.*

Iteration #5

Next, we consider $r(A, A, B)$ *and* $r(D, D, A)$ *as* $r(\bar{X}_1)$ *and* $r(\bar{X}_2)$*, respectively. Accordingly,* $\theta = \{A/D, B/A\}$*. Here, Test I succeeds but Test II fails so* θ *is not a minimizing substitution.*

Iteration #6

Next, we consider $r(D, D, A)$ *and* $r(A, A, B)$ *as* $r(\bar{X}_1)$ *and* $r(\bar{X}_2)$*, respectively. Accordingly,* $\theta = \{D/A, A/B\}$*. Here, Test I succeeds but Test II fails so* θ *is not a minimizing substitution. Since there is no more pair to consider, we conclude that the following is the minimized query.*

Q: $\quad h()$:- $r(A, A, B),\ r(D, D, A)$.

Next, we analyze the complexity of the proposed algorithm for identifying and removing redundant subgoals in a query.

4.5 Complexity

The following theorem specifies the complexity of our query minimization algorithm.

Theorem 4.3 *Minimization of a conjunctive query Q is exponential in the number of repeated variables in the body of Q.*

Proof *Let Q be a conjunctive query with n subgoals consisting of m groups of different subgoals with f_i as the number of subgoals in each group. That is, $n = \sum_{i=1}^{m} f_i$. Since, we need to consider every pair of repeated subgoals $r(\bar{X}_1)$ and $r(\bar{X}_2)$ in each group, the number of combinations to consider is $f_i(f_i - 1)$, and since there are m groups, there are $\sum_{i=1}^{m} f_i(f_i - 1)$ possibilities. Let us assume that every group i has f subgoals. Then we would have $mf(f-1)$ combinations, for each of which we need to perform tests I, II and III. That is $mf(f-1)[cost(I) + cost(II) + cost(III)]$, where $cost(c)$ is the cost of testing condition c. We will see that Cost(III) is exponential, while tests I and II are polynomial, so, we can ignore the latter two in our complexity analysis.*

Now, we consider cost of test III. Without loss of generality, assume that every subgoals has a variables. While performing test III, we have to find a containment mapping from the Source $\mathbb{C}_{\bar{X}_1}\theta$ to the Target $\mathbb{C}_{\bar{X}_2}$. Since the mapping should respect the join variables in both Source and Target, we can use the repeated variables (most of which are join variables) as a guideline. Every two occurrences of a variable A in the Source can be expressed as a constraint. For example, if Source contains subgoals $s_1 = r(A, B, C)$ and $s_2 = r(M, N, A)$, we can express it as the constraint $s_1[1] = s_2[3]$, i.e., first variable in s_1 is equal to the third variable in s_2. Suppose there are C constraints in the Source. We check if there exists a set of subgoals in the Target that satisfies all these C constraints. For this, we define C buckets, one for each constraint. In every bucket, we have the left hand side (LHS) and right hand side (RHS) subgoals. For example, for the constraint $s_1[1] = s_2[3]$, knowing that we have

f occurrences of subgoal r, we need to compare every single subgoal with other subgoals in this group including itself to check if the constraint is satisfied. That is, f subgoals for LHS and f subgoals for RHS which makes f^2 possible cases, and since we have C buckets, the total number of possibilities we have would be f^{2C}. Thus, the total cost for Test III is $mf(f-1)\ f^{2C}$, which is exponential in the number of repeated variables in $\mathbb{C}_{\bar{X}_1}\theta$. ■

4.6 Summary

Query minimization is not a new problem however, it has become more important because in the last decade the number of the applications that generate queries automatically has increased, many of which generate queries with redundancy. One such example is query rewriting.

In this chapter, we studied the problem of query minimization in the context of conjunctive queries and proposed a novel minimization algorithm. In the following chapter, we present the experiments we performed to evaluate the efficiency of our query rewriting algorithm with and without our query minimization algorithm and report the result.

Chapter 5

Experiments and Results

In this chapter, we present our experiments and results of performance evaluation of the proposed query minimization and rewriting algorithms.

We used two sets of input (conjunctive) queries, (1) "real" queries collected from papers related to query rewriting and (2) synthetic queries.

Since the number of queries in the first category is small, we used them mainly to confirm the correctness of our implementations. The second category contains different types of conjunctive queries including *Chain*, *Star*, *Duplicate Random*, *All-Range*, *Augmented Path*, *Augmented Ladder*, and *Snowflake* queries. Augmented path and ladder queries are special cases of Chain queries, and Snowflake queries are special case of Star queries.

For Chain, Star, Duplicate, and Random queries, we adopted a query generator from [PL00]. For the rest, we developed a query generator based on their description. Sample queries are made available at http://users.encs.concordia.ca/~ali_kian/samples/.

We developed an experiment platform with a user-friendly interface using which we could choose the type of experiments: query minimization or query rewriting. Moreover, using the interface we could choose the query type for each experiment and

set the values for the related parameters including number of variables in subgoals, number of subgoals in queries, number of repetition of each experiment, total number of iterations, number of subgoals to be added to input at each iteration, number of views to be added to input at each iteration, time limit, etc.

For query rewriting experiments, we designed the interface such that we could choose the desired approach for each phase of rewriting. We could also enable and/or disable query minimization on input and/or output of our query rewriting prototype.

In all cases, exploiting query generator was such that identical input would be provided to the competing algorithms.

For query minimization experiments, we compare our minimization algorithm with the Kunen-Suciu algorithm. For each query type, we consider proper parameters so that we can compare our results with those reported in [KS02].

For query rewriting experiments, we compare our pattern-based algorithm with Minicon and Treewise algorithms. For each query type, we consider proper parameters so that we can compare our results with those reported in [PL00] and [MS08]. In query rewriting experiments, we measure memory requirement, efficiency and scalability for three rewriting algorithms: Minicon, Treewise, and our pattern-based. Moreover, we exploit query minimization on both input and/or output of our query rewriting algorithm. In the next section, we briefly describe the types of queries we used in our experiments. In Section 5.2, we report the experiments and results for query minimization, and in Section 5.3, we explain the experiments and results for query rewriting.

5.1 Classes of Queries

For the synthetic data in our experiments, we used different classes of queries. Each class has a number of parameters using which we generated different instances of

the given class. For this, we adopted and used the query generator developed for Minicon [PL00] to generate Chain, Star, Duplicate and Random queries. We also extended the classes considered and created what we refer to as All-Range queries [KS09]. Moreover, to compare our minimization algorithm with the one introduced in [KS02], we developed query generator for Augmented Path queries, Augmented Ladder queries, and Snowflake Star queries used in [KS02].

5.1.1 Chain Queries

Each query in this class has subgoals that are chained by join variables. The following example shows a Chain query with 3 subgoals, and 3 distinguished variables in which variable C chains the first and the second subgoals, and variable E chains the second and third subgoals.

$Q: \quad h(A, B, F) :\text{-} r(A, B, C), s(C, D, E), t(E, F).$

The parameters to generate Chain queries include the number of subgoals, number of variables in every predicate, and number of distinguished variables in the query.

5.1.2 Star Queries

Star queries form a class of conjunctive queries in which a "central" subgoal is joined with all other subgoals in the query. The following example shows a Star query with 4 subgoals, and 3 distinguished variables in which u is the central subgoal joined with other subgoals in the query based on their first variable.

$Q: \quad h(A, B, G) :\text{-} r(A, B, C), s(D, E, F), t(G, H), u(A, D, G).$

A generic Star query can be defined based on the number of subgoals, number of variables in each subgoal, and number of distinguished variables.

5.1.3 Duplicate Queries

Since different occurrences of the same predicate may have significant impact on the performance of query rewriting, we generated and used Duplicate queries with different number of repetition of predicates. The following example shows a Duplicate query, in which predicate r occurs 3 times in the body.

$Q: \quad h(A,D) :\text{-} r(A,B,C), r(C,C,F), t(B,H), r(A,D,E).$

5.1.4 Random Queries

In a Random query, the join structure, number of variables and their occurrences are decided randomly. We could think of such a query as a generic case of Chain, Star, and Duplicate queries. The parameters of a generic Random query include the number of subgoals, number of variables in every predicate, and number of distinguished variables.

5.1.5 All-Range Queries

The coverages generated in query rewriting for the classes of queries mentioned above do not contain all possible cases. To push the algorithms to their limits, we also consider new classes of queries queries and views that generate coverages with all possible occurrence identifiers. For example, for a query with 3 subgoals, we have 7 possible occurrence identifiers. The All-Range queries can generate coverages with identifiers in the range of 1 to 2^{n-1}. Unlike the classes of queries described earlier, the definition of query and views in this class are closely related. The following example shows a query and views that generate all possible coverages (in this case 7 coverages) [KS09].

$Q: \quad h() :\text{-}\ r_0(A_{01}, A_{02}), r_1(A_{01}, A_{12}), r_2(A_{02}, A_{12}).$

V_1 : $v_1(A_{01}, A_{02})$:- $r_0(A_{01}, A_{02})$.

V_2 : $v_2(A_{01}, A_{12})$:- $r_1(A_{01}, A_{12})$.

V_3 : $v_3(A_{02}, A_{12})$:- $r_0(A_{01}, A_{02}), r_1(A_{01}, A_{12})$.

V_4 : $v_4(A_{02}, A_{12})$:- $r_2(A_{02}, A_{12})$.

V_5 : $v_5(A_{01}, A_{12})$:- $r_0(A_{01}, A_{02}), r_2(A_{02}, A_{12})$.

V_6 : $v_6(A_{01}, A_{02})$:- $r_1(A_{01}, A_{12}), r_2(A_{02}, A_{12})$.

V_7 : $v_7()$:- $r_0(A_{01}, A_{02}), r_1(A_{01}, A_{12}), r_2(A_{02}, A_{12})$.

In this example, each view V_i provides only one coverage. The index chosen for each view indicates the occurrence identifier for the related coverage. To generate such query and views, we need to know the number of subgoals in the query. Assuming that the query has n subgoals and that each view is responsible for generating one coverage, to generate all possible occurrence identifiers, we need $2^n - 1$ views [KS09].

First, we generate a query with n subgoals. In order to generate a coverage with the identifier $2^n - 1$, we need all subgoals in the query to be joined. Moreover, these subgoals should also appear in the related view where the join variables are not distinguished, otherwise, the view will not give a coverage containing all the subgoals but several smaller coverages whose identifiers are less than $2^n - 1$. So, there are the two important points to consider when generating All-Range queries, (1) every subgoal in the query is joined with every other subgoal, but uses different variables each time, and (2) distinguished variables in the views should be selected in such a way that the coverage cannot be broken into smaller ones.

In our example, we consider a query with $n = 3$ subgoals, so we have 3 join variables: A_{01} is the join variable between r_0 and r_1, A_{02} is the join variable between r_0 and r_2, and A_{12} is the join variable between r_1 and r_2.

When generating each view v_i, $1 \leq i \leq 2^n - 1$, to form the body of v_i, we pick subgoals from Q based on the binary representation of the view index (i.e.. i), and

add it to v_i. That is, if there is a 1 at the k^{th} position in the binary representation of i, then we add the subgoal at position k in Q to the body of v_i. To form the view head, we copy all the variables to the view head except the join variables of the subgoals copied in the view.

5.1.6 Augmented Path queries

An augmented path query is a special type of Chain query which consists of a path of length n with an additional dangling edge rooted at each node except the last. An augmented path query of length n has $2n$ subgoals. The following is an example of such a query.

$$Q: \quad h() \text{ :-} r(A_0, A_1), r(A_1, B_1), r(A_0, A_2), r(A_2, B_2), r(A_0, A_3), r(A_3, B_3).$$

Figure 5.1.(a) illustrates the path formed by variables in Q [KS02].

5.1.7 Augmented Ladder queries

Ladder queries are Chain queries in ladder-like structure with all edges pointing from upper left to lower right [KS02]. An augmented ladder query is also defined as a ladder query with additional dangling edges attached to each node.

For instance, Figure 5.1.(c) illustrates the path formed by variables in the following query [KS02].

$$\begin{aligned} Q: \quad h() \text{ :-} & r(B_1, A_F), r(A_F, A_0), r(A_0, A_L), r(A_L, B_2), \\ & r(B_3, A_F), r(A_F, A_1), r(A_1, A_L), r(A_L, B_4), \\ & r(B_5, A_F), r(A_F, A_2), r(A_2, A_L), r(A_L, B_6). \end{aligned}$$

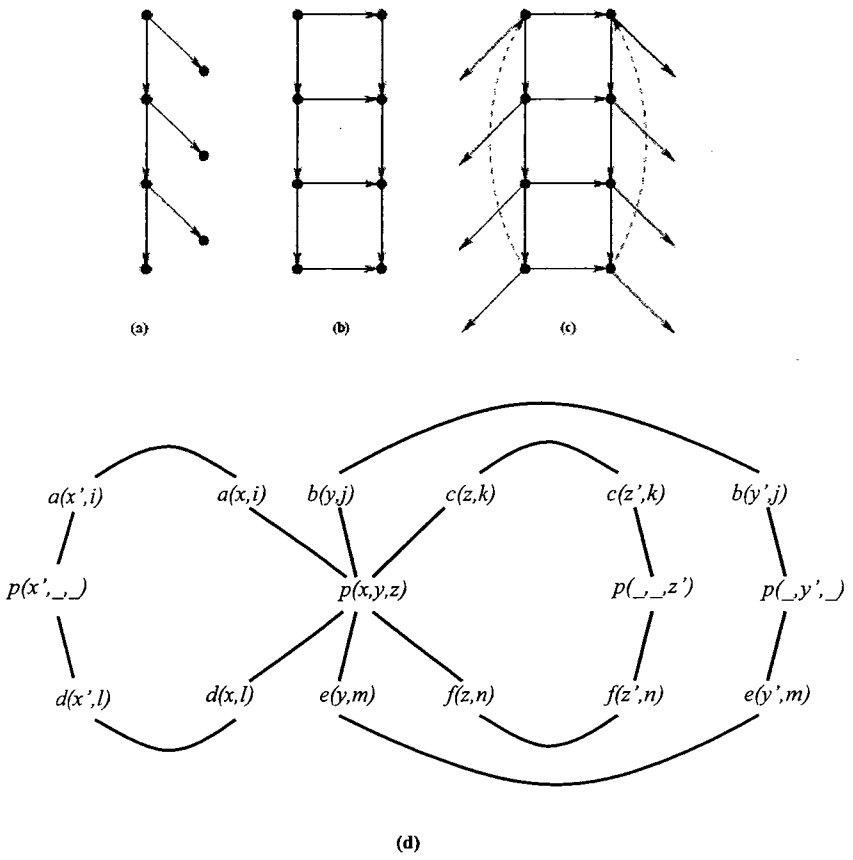

Figure 5.1: (a) Augmented Path Query, (b and c) Augmented Ladder query, and (d) Snowflake) query [KS02]

5.1.8 Snowflake queries

Snowflake queries found in data warehouse applications ([KS02]), are special cases of Star queries that have a central subgoal with each variable joined to a pair of side subgoals [KS02]. Side subgoals are joined through an extra copy of the central subgoal with its variables renamed [KS02]. An example of a snowflake query is illustrated in Figure 5.1.(d).

Next, we report the results of our experiments for different classes of queries. We first present the performance evaluation results for query minimization algorithm and then consider experiments and results for query rewriting algorithm.

5.2 Query Minimization

In order to evaluate the performance of our minimization algorithm, we developed in Java, a version of our algorithm and a version of the minimization algorithm reported in [KS02] to which we refer as Kunen-Suciu algorithm. We attempted to ensure the efficiency of our implementation of Kunen-Suciu algorithm matches with what they have reported for different types and sizes of queries in [KS02].

For evaluating and comparing our algorithm, we conducted extensive experiments for which we used a Pentium 4, 1.73 GHz desktop computer with 1GB RAM running MS Windows XP. In our experiments, the amount of heap used never exceeded 32MB.

For these experiments, we consider six types of queries. They include Chain, Duplicate, Star, Augmented Path, Augmented Ladder, and Snowflake Star queries.

To evaluate the efficiency and scalability of our query minimization algorithm, we created test data of various sizes and tried to push the algorithm to its limits by increasing the number of subgoals in the queries. Figures 5.2 and 5.3 show query minimization time in seconds for Chain and Duplicate queries, respectively. We can see that in both cases, our minimization algorithm outperforms Kunen-Suciu algorithm. These figures also show the effectiveness of the heuristics in our minimization algorithm. In these experiments, we increased the number of joins in the input query to more than 1000 joins.

As shown in Figure 5.3, we could process Duplicate queries with 1000 joins in about 1 second. This is due to the presence of many redundant subgoals in such queries resulting in a rapid decrease of query size. In our experiments, we observed

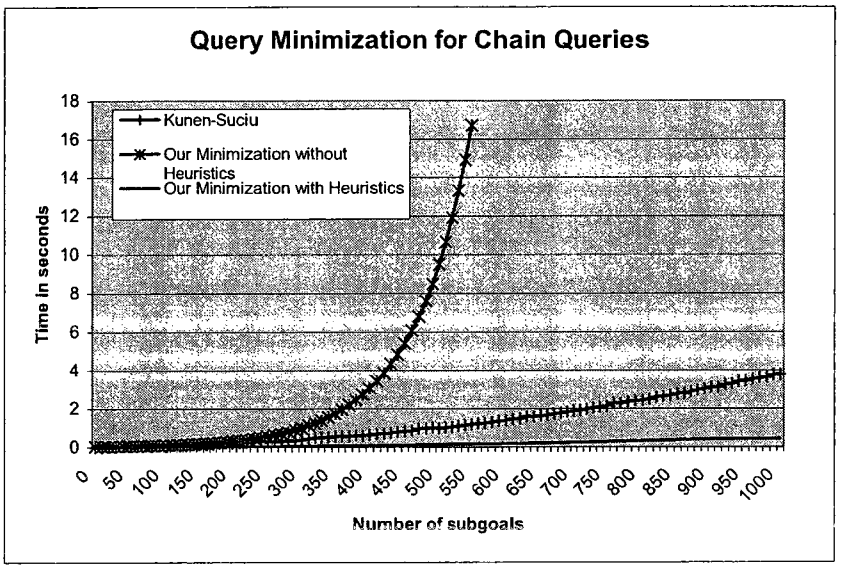

Figure 5.2: Average minimization time for Chain queries.

that the sooner a redundant subgoal is identified the faster minimization performs. This is an important role of the heuristics we used which help identify redundant subgoals in earlier in the process.

In our experiments, since Chain and Star queries usually do not contain as many redundant subgoals as Duplicate queries do, the minimization times for them were relatively higher compared to Duplicate queries.

Regarding Augmented Path queries, as reported in [KS02], Kunen-Suciu algorithm took about 57 seconds to process such queries with 1000 subgoals, whereas our minimization algorithm performed this in about 35 seconds. For Snowflake queries with central subgoals with arity 20, and 1 to 20 sets of side subgoals and complete side loops between side subgoals, Kunen-Suciu algorithm completed the task in around 28 seconds whereas ours completed this in less than 5 seconds. For Snowflake queries, we continued increasing the query size to more than 4500 subgoals for which our minimization algorithm took around 40 seconds. The minimization time reported

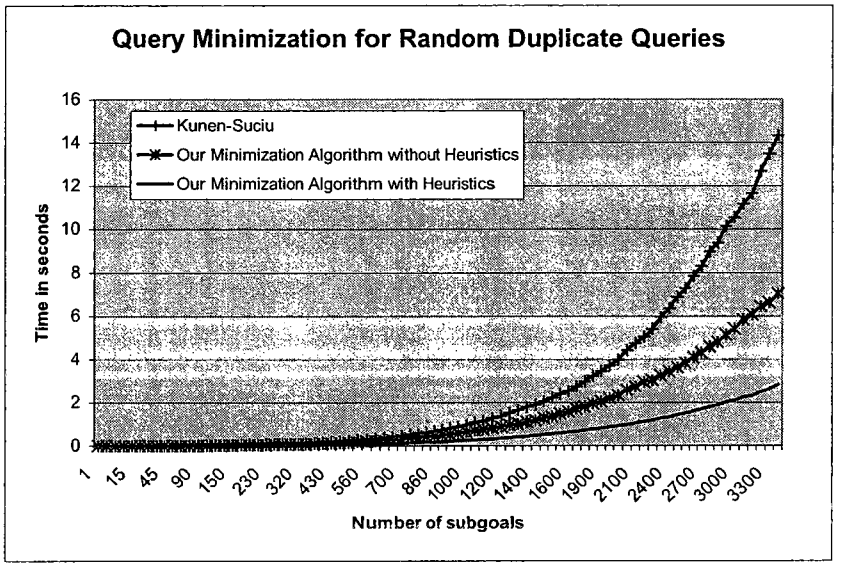

Figure 5.3: Average minimization time for Duplicate queries.

in [KS02] is close to the minimization time of our implementation of Kunen-Sucio algorithm that confirms fairness of our comparison. We need to mention that Kunen-Suciu used a Pentium 4, 1.7GHz, linux, C++ for their experiments whereas we used a Pentium 4, 1.73GHz, Windows XP, Java.

Figures 5.4 and 5.5 compare the results of our minimization algorithm with Kunen-Suciu algorithm for augmented ladder and snowflake queries. In each case, we show the impact of our heuristics on the performance too.

The improved performance of our algorithm compared to Kunen-Suciu ([KS02]), can be explained by noting that regardless of the query type, Kunen-Suciu algorithm needs to evaluate the query Q at least once on the canonical database which is defined based on Q. Considering a query with hundreds of subgoals, this evaluation is relatively expensive even if it is done only once. This, however, is not the case for our algorithm, since as the experiments show, in most cases our heuristics were able to identify a proper minimization substitution early in the process.

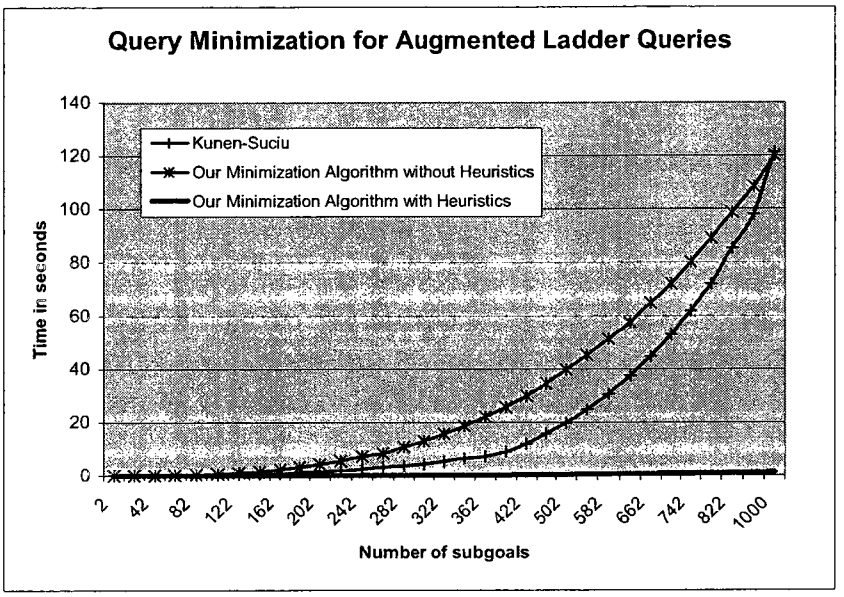

Figure 5.4: Average minimization time for Augmented Ladder queries.

5.2.1 Application in Query Rewriting

In order to test our minimization algorithm in situations closer to real life applications, we considered query rewriting as an application where query minimization can be used and performed numerous experiments. In these experiments, we ran the proposed query minimization technique on the input queries and also on the generated rewritings.

We used Chain, Duplicate, and Star queries. For each experiment, we randomly generated a query Q and a set of views V as the input to our rewriting algorithm, which generated rewriting Q'.

We measured the total time for query rewriting, area of rewriting (a measure which considers the number of subgoals in the body of the rewriting generated), scalability

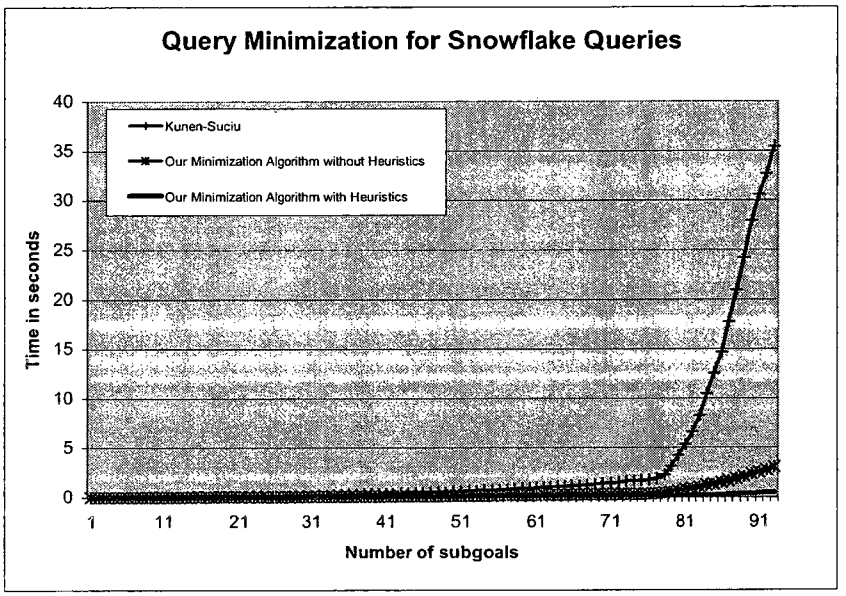

Figure 5.5: Average minimization time for Snowflake queries.

on the number of views, and minimization time. While types of queries and views, the number of repeated subgoals as well as the size of rewriting are important factors in performance of query minimization, the results of our extensive experiments indicate that applying minimization on Q as a pre-processing step to rewriting improves the performance in the above aspects, in general. Similarly, applying minimization on the output of rewriting improves the area significantly. Figures 5.6 and 5.7 show in logarithmic scale, the average area and average total time for rewriting of minimized versus non-minimized queries. The figures show the best case (minimized input and minimized rewriting), versus the worst case (no minimization). That is, the graph for Minimized/Minimized represents the case where both input and output of rewriting were minimized. Similarly, the graph for non-Minimized/non-Minimized represents the cases where minimization was not used.

In the queries generated for the experiments in Figures 5.6 and 5.7, the average

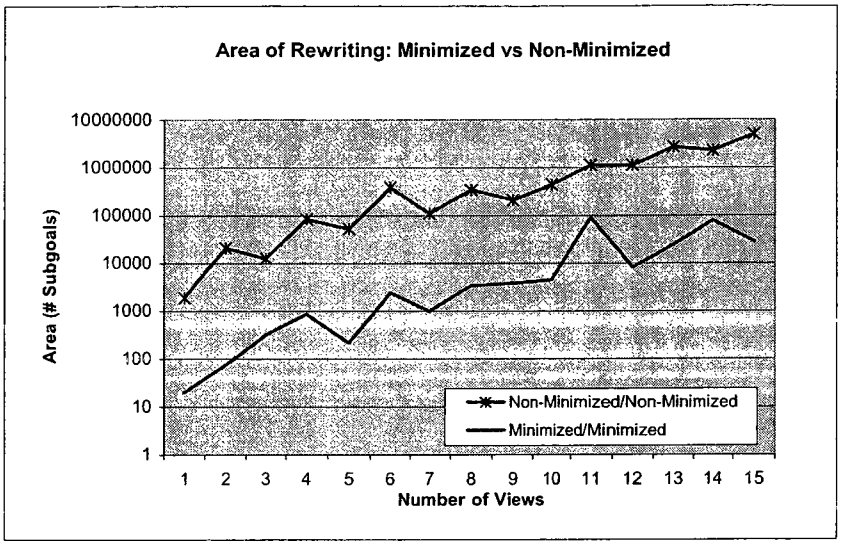

Figure 5.6: Area of rewriting for minimized vs non-minimized query.

number of predicates, variables, and distinguished variables are 15, 4, and 2, respectively. As shown in Figure 5.6, for an input of 10 views, the average area size (number of subgoals in the rewriting) is near 7000, whereas without minimization, it is more than 100,000 subgoals. This significant improvement on the area can be considered as indicative of the rewriting quality. Note that in these experiments as the number of views increases, the area of the rewriting does not necessarily increase.

Also, as shown in Figure 5.7, applying minimization to the input query Q improves the rewriting time and scalability as the number of views grow. For example, for 15 views, it took about 1 second, on average, to obtain a rewriting for minimized input, whereas it took more than 1.5 hours for a non-minimized query.

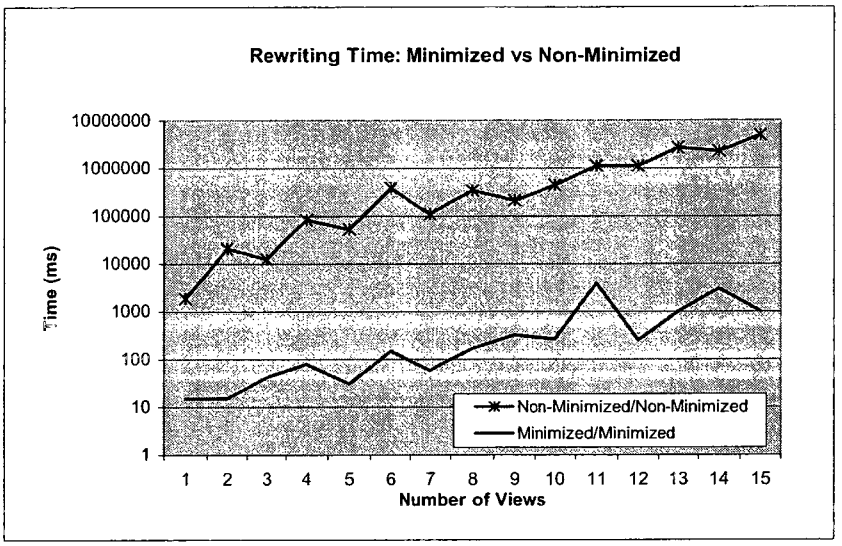

Figure 5.7: Total rewriting time for minimized vs non-minimized query.

5.3 Query Rewriting

In this section, we report the experiments and results for query rewriting. For the synthetic data, we considered different classes of queries. For every class of queries, input parameters to define instances of queries and views included query type, number of subgoals, number of variables in each subgoal, number of distinguished variables, and number of subgoal repetition. For every test, we repeated the experiments many times (100 times in most cases). We consider different measures including memory requirement, performance, scalability, and quality of rewriting, and compare the results with (two implementations of) Minicon [PL00], Treewise [MS08], and our pattern-based rewriting technique. The implementations of Treewise and an implementation of Minicon were provided to us by the authors of [MS08]. In their work, they showed the correctness of their implementation of Minicon algorithm and matched the performance with the results reported in [PL00]. We also developed our

version of Minicon algorithm in which we used view-based approach for finding atomic coverages and used exclusion set technique to perform the Cartesian products of the buckets used by Minicon to generate the rewriting. As discussed in Chapter 3, view-based approach generates a minimum number of coverages in a natural way whereas the subgoal-based approach (used in Minicon) requires some bookkeeping for that purpose. We refer to our view-based (VB) version of Minicon as VB-Minicon, and naturally witness improved performance compared to the original Minicon algorithm as shown in our experiments.

We also developed a prototype of pattern-based query rewriting technique. In the first phase of rewriting, pattern-based technique uses view-based approach to look for atomic coverages, and in the second phase, it follows the Prevention approach and considers patterns as a guide to perform the Cartesian product for combining coverages.

Since the Treewise algorithm naturally generates rewritings that are shorter in the number of subgoals in the body than Minicon (and have a better area), in order to make comparison fair, in all these experiments, we exploit our query minimization on the output of the rewriting, unless mentioned otherwise. Our goal is to compare the performance of the rewriting for the same inputs and the same output. Our results indicate that in most cases, our technique is much faster than other techniques and generates rewriting with smaller areas (see Section 5.3.3).

5.3.1 Memory Requirement

In this section, we report the memory requirements for the four algorithms Minicon, VB-Minicon, Treewise, and pattern-based query rewriting. In our experiments, we realized that rewriting of All-Range queries requires more memory compared to other types of queries. So, to push all these algorithms to their limits, we used All-Range

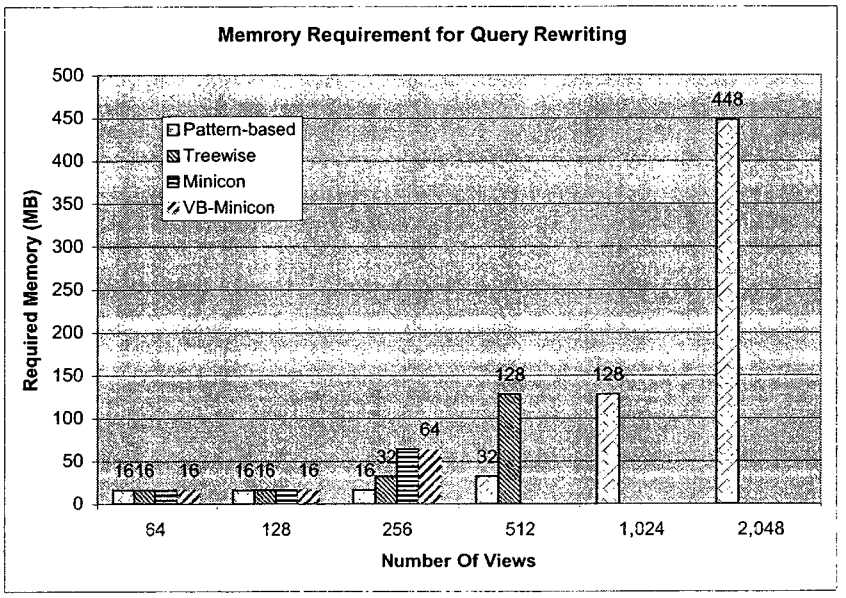

Figure 5.8: Comparison of memory requirement for Minicon, VB-Minicon, Treewise and pattern-based algorithms

queries and ran the experiments for different numbers of views.

As shown in Figure 5.8, pattern-based algorithm requires the least amount of memory. At each step, the same input was used for all these algorithms. For the inputs containing 63 and 127 views (with 203 and 877 rules generated in the rewriting), all the four algorithms completed the process using 16 MB of memory. For 255 views (4140 rules in the rewriting), both versions of Minicon required 64 MB, Treewise required 32 MB, but pattern-based required 16 MB as before. For 511 views (21147 rules in the rewriting), Minicon algorithms could not complete the task due to memory exception. For this case, Treewise and pattern-based completed the task using 128 MB and 32 MB, respectively. To explain why Minicon algorithms crashed, we examined the buckets structure. For 511 views (and a query with 9 subgoals),

we had 9 buckets each containing 256 coverages which means Minicon needed to perform 256^9 Cartesian products. Treewise could finish the task because of its top-down decomposition approach which helps prune away many unnecessary combinations. The case was easier for pattern-based because it basically broke this large Cartesian product into 21147 smaller Cartesian products. For 1023 views (115975 rules in the resulting rewriting), pattern-based was the only algorithm that could complete the task for which it used 128 MB of memory. We continued with 2047 views (678570 rules in rewriting) for which pattern-based completed the task using 448 MB of memory. The pattern-based algorithm could not complete the task for 4095 views where there were 4213597 rules in the resulting rewriting [KS09].

5.3.2 Efficiency and Scalability

To evaluate the efficiency and scalability of the algorithms, we considered Chain, Star, Duplicate, Random, and All-Range queries. For each case, we used the same input to the four algorithms and compared their performance in terms of the rewriting time.

We start with All-Range queries for which Figure 5.9 shows the rewriting time. As we can see, the rewriting time for up to 32 views is almost the same for all these algorithms, however, for larger inputs, our pattern-based algorithm outperforms others, significantly. For instance, for 127 views, pattern-based, Treewise, VB-Minicon and Minicon took 625 ms, 3875 ms, 187000 ms, and 134344 ms, respectively. For 255 views, pattern-based completed in 2031 ms- almost 10 times faster than Treewise (21641 ms). The Minicon algorithms could not finish this task. As the input size increased, the gap in the efficiency between pattern-based and Treewise increased. For example, for 511 views, pattern-based completed in about 8 seconds, whereas Treewise finished in 153 seconds.

Each view in the example of All-Range queries provided in section 5.1.5, has

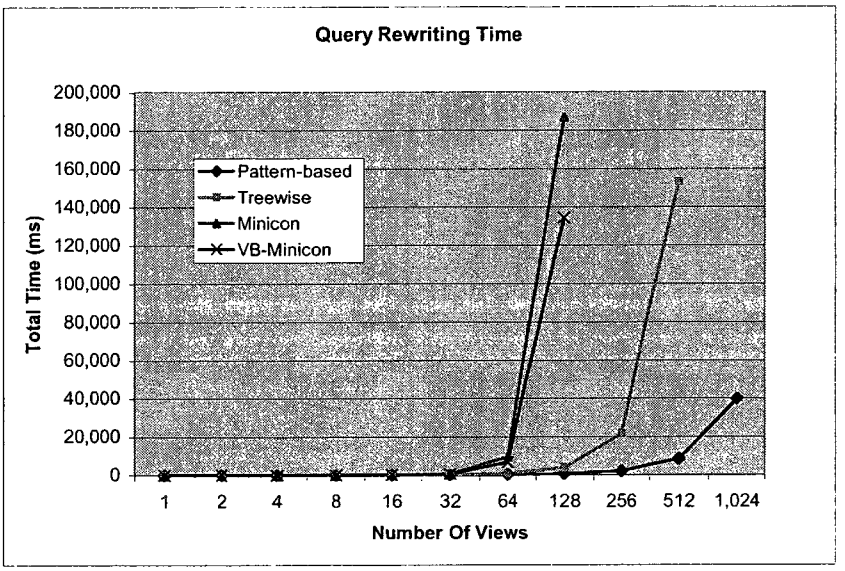

Figure 5.9: Rewriting time for All-Range queries with up to 10 subgoals

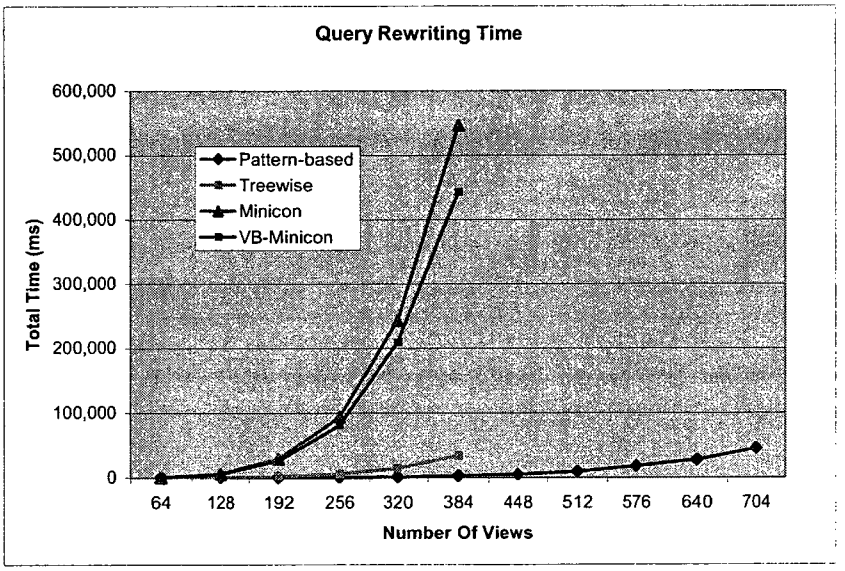

Figure 5.10: Rewriting time for All-Range queries with 6 subgoals and up to 20 repetitions of view types

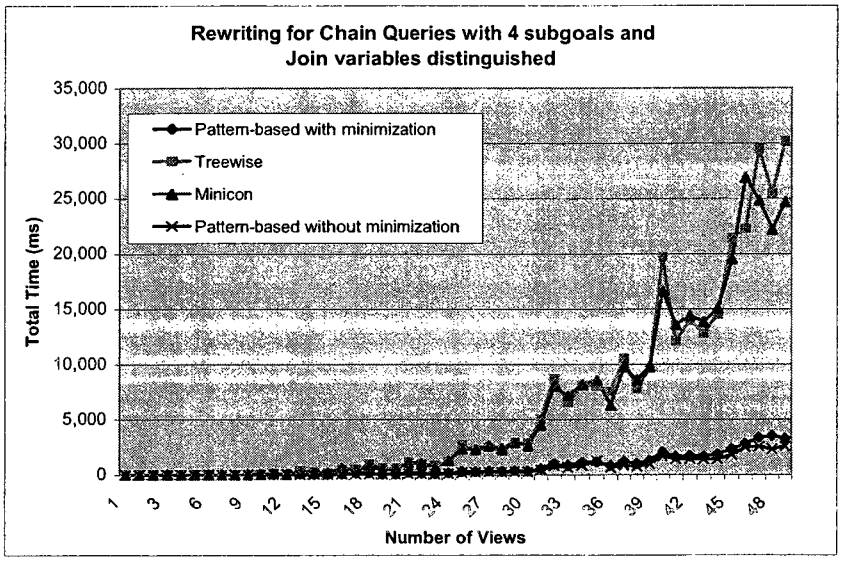

Figure 5.11: Rewriting time for Chain queries with 4 subgoals and all join variables distinguished

a unique combination of subgoals that results in coverages with unique identifiers. Figure 5.10 illustrates the rewriting time for All-Range queries which contain up to 20 instances of views with the same coverages. At each iteration in this experiment and during view generation, we increased the number of views by incrementing the number of repetitions of views with the same coverages.

So far, we illustrated the rewriting time for All-Range queries. In what follows, we focus on the same types of queries and input discussed in [PL00]. Since the two versions of Minicon are almost identical in terms of performance and memory requirements, in the rest of the discussion we refer to both versions as Minicon. Moreover, to show the overhead of the query minimization algorithm, we ran pattern-based with and without minimization and present the results. Next, we consider Chain queries. Figure 5.11 shows the rewriting time for Chain queries with four subgoals and all join variables distinguished. Pattern-based algorithm, with and

without minimization, outperform others. The results also show that the overhead of the minimization algorithm is not considerable.

Figure 5.12 shows the rewriting time for Chain queries with 8 subgoals and all variables distinguished. As we can see, the best performance is by pattern-based algorithm without minimization, however, compared to pattern-based with minimization, the overhead of minimization seems not considerable. In all runs of this test, pattern-based with minimization outperformed Treewise and Minicon algorithms significantly.

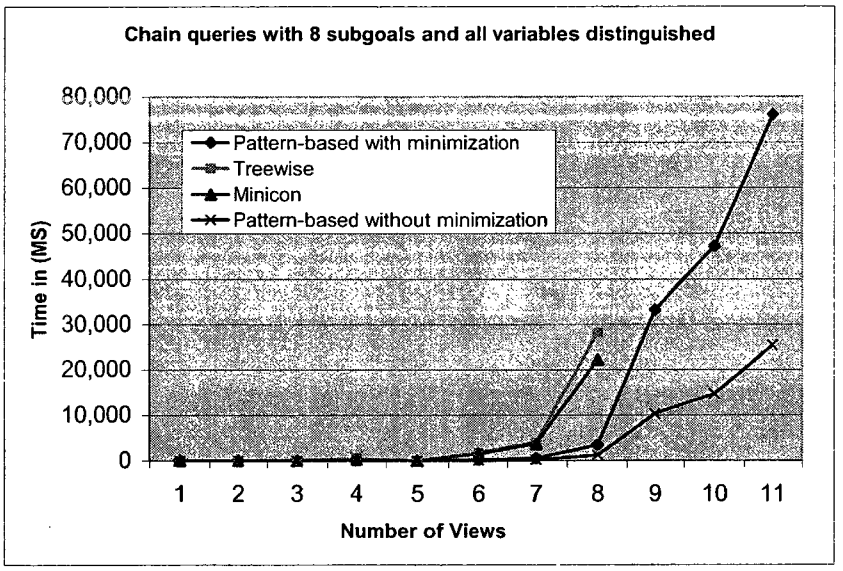

Figure 5.12: Rewriting time for Chain queries with 8 subgoals and all variables distinguished

Figure 5.13 shows the query rewriting time for Star queries with 10 subgoals and non-join variables distinguished. As we can see, pattern-based algorithm with and without minimization outperforms Treewise and Minicon, and Treewise performs slightly better than Minicon. We can also see that the difference in rewriting time between all these algorithms is less than 1 second. The reason is that there were few rules in the output of the rewriting of these queries.

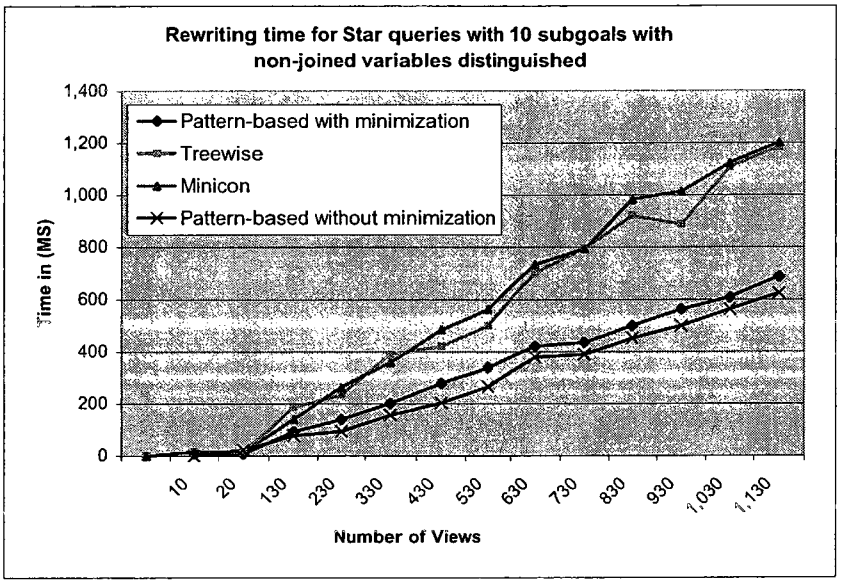

Figure 5.13: Rewriting time for Star queries with 10 subgoals and non-join variables distinguished

Figure 5.14 shows the rewriting time for Star queries with 10 subgoals and all join variables distinguished. For these experiments, we increased the number of views and measured the time, the number of rules in the output, and their areas. We observed that increasing the number of views does not always result in increasing the time. For this reason, we illustrate the rewriting time based on the number of rules in the output which clearly indicate that pattern-based algorithm with minimization outperforms Treewise and Minicon. For example, for 30 views, where there are 129024 rules in the rewriting, pattern-based finished in less than 10 seconds, whereas Minicon and Treewise needed at least 35 seconds.

To evaluate the performance for Duplicate queries, we considered queries with 12 subgoals, 5 duplicate subgoals and 4 distinguished variables. Figure 5.15, compares the rewriting time for Duplicate queries in which pattern-based algorithm outperforms the others. For this sets of experiments, we applied minimization on the input query

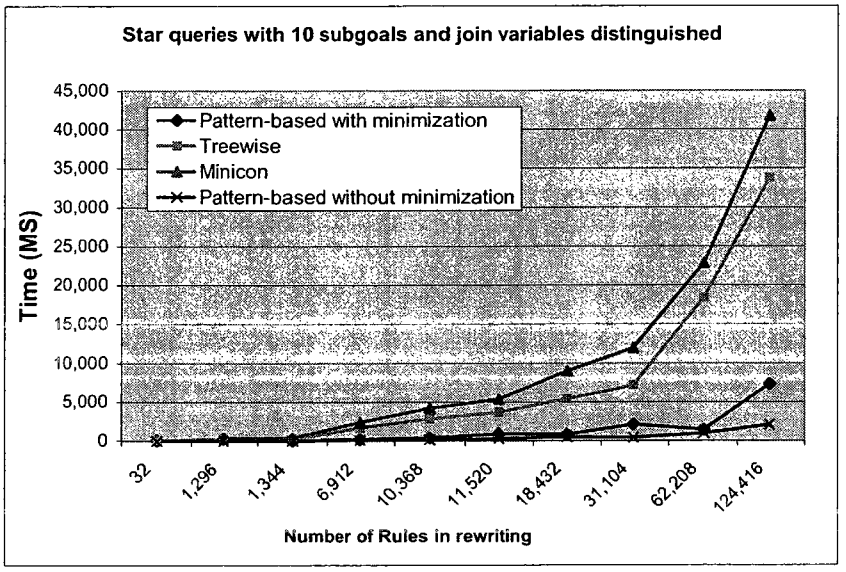

Figure 5.14: Rewriting time for queries with 10 subgoals and all join variables distinguished

as well. We observed that when there are some redundant subgoals in the input query and/or views, the rewriting time reduces significantly. As a result, increasing the number of views does not necessarily result in increase in the rewriting time. We thus show for this set of experiments only the rewriting time based on the number of rules generated.

For experiments on Random queries we considered queries with 10 subgoals each having a maximum of 5 variables and the rest of the parameters were decided randomly. Similar to the case of Duplicate queries, we enabled minimization on the input. Figure 5.16 shows the rewriting time and as can be seen, pattern-based algorithm shows to be superior to others, with the overhead of minimization being not considerable.

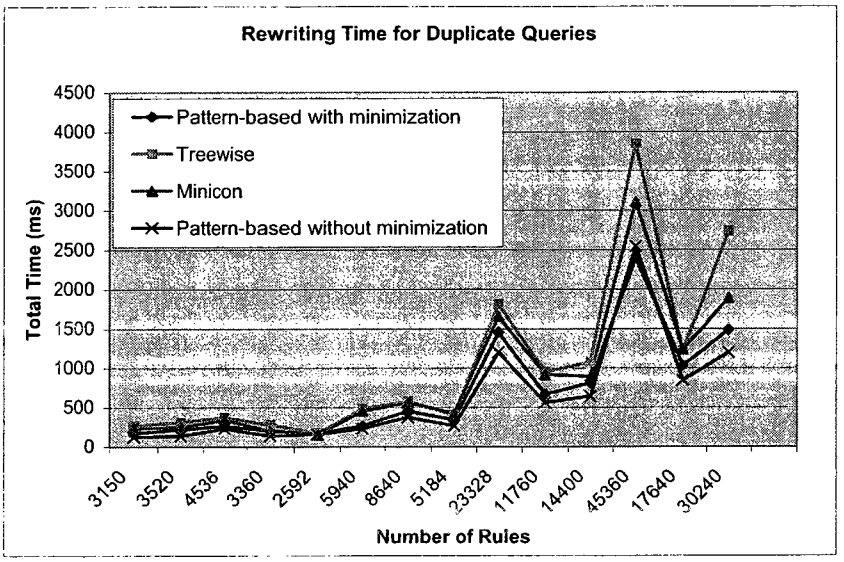

Figure 5.15: Rewriting time for Duplicate queries with 12 subgoals

5.3.3 Rewriting Quality

Figure 5.17 compares these rewriting algorithms based on the quality of the rewriting they generate, measured by area (the total number of subgoals in the output). The rewriting time for these experiments is illustrated in Figure 5.12 for Chain queries with 8 subgoals and all the variables distinguished. An important point here is that pattern-based with minimization was not only faster but also generated a better quality result. The quality for pattern-based without minimization and Minicon are identical as expected.

Figure 5.18 compares the area of the output for Star queries. The rewriting time for this experiment is illustrated in Figure 5.14. The interesting point here is that considering the last two runs, even though the number of rules increased from 124,416 to 129,024, the final area of the rewriting decreased in both pattern-based with minimization and Treewise. This is important as it shows the effectiveness of

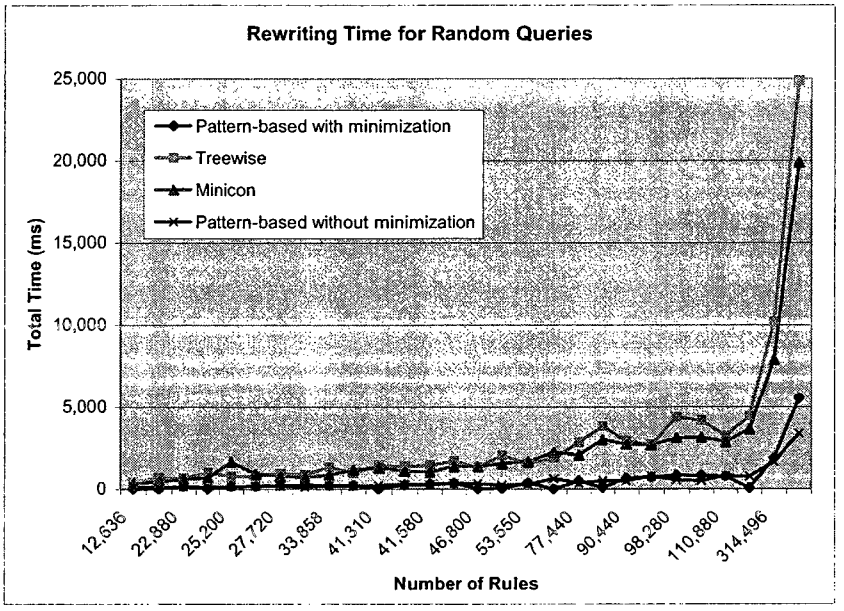

Figure 5.16: Rewriting time for Random queries with 10 subgoals

the minimization technique.

5.4 Summary

In this chapter, we presented the results of our experiments for evaluating performance of our query minimization algorithm for different types and sizes of queries, and showed its superiority over the existing minimization algorithms. Moreover, we experimentally compared our pattern-based query rewriting with and without our minimization technique against existing rewriting solutions and showed that even with the overhead of query minimization, our rewriting technique outperforms other techniques.

So far our focus has been on standard conjunctive queries. In the next chapter, we investigate the query rewriting problem in the context of conjunctive queries with

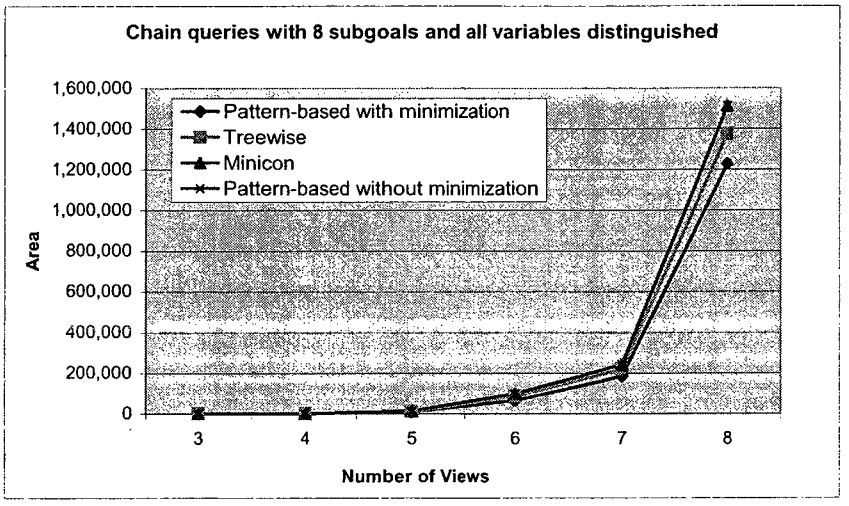

Figure 5.17: Area for queries with 8 subgoals and all variables distinguished

constraints.

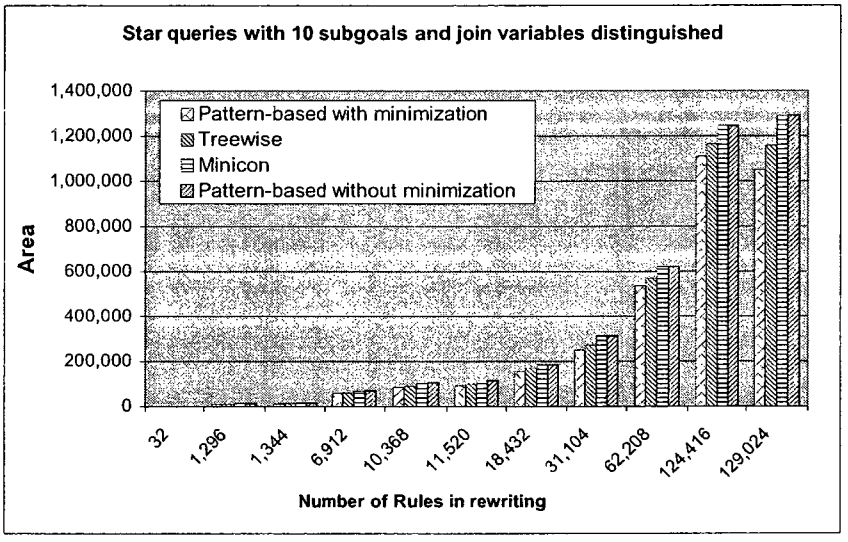

Figure 5.18: Rewriting area for queries with 10 subgoals and all join variables distinguished

Chapter 6

Query Rewriting for Conjunctive Queries with Constraints

In Chapter 3, we investigated the problem of query rewriting in the context of standard conjunctive queries. In this chapter, we extend the problem to conjunctive queries with constraints. It has been shown that, in general, adding constraints to conjunctive queries adds to the complexity of both query containment and query rewriting problems. However, there are certain classes of conjunctive queries with constraints for which the complexity of query containment remains in NP. Such queries are said to satisfy the homomorphism property [Klu88]. In this chapter, we identify new such classes of conjunctive queries with constraints, and extend our pattern-based rewriting algorithm to support such queries.

6.1 Introduction

As explained in Chapter 2, containment of standard conjunctive queries is characterized based on single containment mappings. While the complexity of containment of standard conjunctive queries is NP-complete, it has been shown that the complexity of

containment of conjunctive queries with constraints is Π_2^P-complete [Klu88, Mey92]. The reason is that for such queries, two or more containment mappings may team up and satisfy the containment requirement. In other words, to reject the containment of Q_1 in Q_2, we should consider and test all possible combinations of containment mappings. On the other hand, there are conjunctive queries with constraints that do not require such an expensive test. Since containment test for such queries is based on single containment mappings, they are said to have *homomorphism property*. That is, as in the standard case, to establish the containment for queries with homomorphism property, we need to find a single containment mapping that satisfies the containment requirement, and to reject the containment, we need to reject all single containment mappings.

Previous studies ([Klu88, GSUW94, ALM04]) identified classes of conjunctive queries with constraints that satisfy homomorphism property [Klu88]. In this chapter, we extend the result in [ALM04] and identify new classes of such queries that have homomorphism property.

In order to keep the complexity of the problem limited to query containment and not directly "influenced" by constraint solving, we restricted our attention to *Linear Arithmetic Constraints* which can be solved in polynomial time.

We refer to such conjunctive queries as Conjunctive queries with Linear Arithmetic Constraints, or CLAC for short [KS05a, KS05b]. CLAC queries appear frequently in database applications. For instance, consider the following typical SQL query on relation schemas $r(X, Y, Z)$ and $s(T, U, W)$.

Example 6.1 *[CLAC Query]*

SELECT X

FROM r, s

WHERE $X < 2Y$ AND $Z = T$ AND $U \leq W$;

This query can be expressed as the following CLAC query.

$$Q_1: \quad h(X) :\text{-} r(X,Y,Z), s(Z,U,W), X < 2Y,\ U \leq W$$

Next, we formally define the CLAC queries [KS10].

Definition 6.1 *A CLAC query Q is a conjunctive query of the form:*

$$Q: \quad h(\bar{X}) :\text{-} g_1(\bar{X}_1), \cdots, g_k(\bar{X}_k), \alpha_1, \cdots, \alpha_n,$$

where $h(\bar{X})$ is the query head, and every $g_i(\bar{X}_i)$ is a subgoal. Let S be the set of all the variables and constants in the body of Q. Every head variable and every argument in the subgoals $g_i(\bar{X}_i)$ is a linear arithmetic expressions over S. Every constraint α_j is of the form $E_l\ \theta\ E_r$, where E_l and E_r are linear arithmetic expressions over S, and θ is a comparison operator from $\{<, \leq, >, \geq, =, \neq\}$.

Example 6.2 *Let $r(X,Y,Z)$ and $s(T,U,W)$ be two relation schemas. The following query illustrates another example of CLAC query based on r and s.*

$$Q_2: \quad h(2X+1, W) :\text{-} r(X, Z+2, Z), s(5Z, U, W),\ U \leq W$$

Here, the set of variables and constants in Q_2 is $S = \{X, Z, U, W, 2, 1\}$, and the linear arithmetic expressions defined over S include $2X+1$, $Z+2$, $5Z$, and Z.

Similar to the standard conjunctive queries, containment and rewriting of CLAC queries are characterized using the notion of containment mapping. However, since containment mappings do not map expressions but only single variables, we cannot find mapping for general form of CLAC queries. For this, we define the notion of *Normalized* CLAC query.

Definition 6.2 (Normalized CLAC) *A CLAC query is normalized if every argument in the head and subgoals is a single variable.*

The procedure to normalize a CLAC query is as follows.

Definition 6.3 (CLAC Normalization) *Given a CLAC query Q, for every argument E in the head or subgoals in Q, if E is not a variable, replace it by a new variable A_{new}, and append the constraint A_{new}=E to the list of constraints in Q.*

The following example illustrates how to normalize a CLAC query.

Example 6.3 *Consider the CLAC query Q_2 in Example 6.2. To normalize it, we consider the arguments in head or subgoals that are not single variables. They are $\{2X+1, Z+2, 5Z\}$. We define A_1 for $2X+1$, replace $2X+1$ with A_1 in the head, and add constraint $A_1 = 2X+1$ to the query body. Similarly, we replace arguments $Z+2$ and $5Z$ with the new variables A_2 and A_3, respectively, and append constraints $A_2 = Z+2$ and $A_3 = 5Z$ to query body. The resulting query is as follows.*

$$Q_3: \quad h(A_1, W) :\text{-} r(X, A_2, Z), s(A_3, U, W), U \leq W, A_1{=}2X{+}1, A_2{=}Z{+}2, A_3{=}5Z.$$

In the rest of our discussion, we assume the CLAC queries are normalized, unless specified otherwise.

It is easy to see that the class of CLAC queries extends *conjunctive queries with arithmetic comparisons* which were studied in previous work on query containment and rewriting [Klu88, GSUW94, ALM02, KS05b]. In fact, conjunctive queries with arithmetic comparison (AC queries) include constraints in the forms $A\theta B$ and $A\theta c$, where A and B are variables, c is a constant, and $\theta \in \{<, \leq, >, \geq, =, \neq\}$.

The rest of this chapter is organized as follows. Next, we study the containment of queries with constraints. In section 6.3, we introduce new classes of conjunctive queries with homomorphism property and in section 6.4, we provide a rewriting algorithm for CLAC queries with homomorphism property. It is equally important to recognize classes of CLAC queries for which homomorphism property does not hold. We elaborate on this in Section 6.6 and identify such classes. Section 6.7 summarizes this chapter.

6.2 Containment of Queries with Constraints

Since query rewriting is based on the notion of containment, in this section, we study the problem of query containment in the context of CLAC queries, and then focus on rewriting of such queries. A containment mapping from query Q_1 to Q_2 is a function that maps variables in Q_1 to those in Q_2.

As shown in previous studies, the presence of constraints affects the requirements for query containment. To illustrate this point, we compare containment in the standard case (conjunctive queries with no constraints) [CM77] with containment in conjunctive queries with arithmetic comparisons [Klu88, IS99], and show that the results from [Klu88] carry over to CLAC queries.

Theorem 6.1 characterizes containment of conjunctive queries for the standard case [CM77], and Theorem 6.2 does this for conjunctive queries with arithmetic comparisons (*AC queries*)[Klu88].

Theorem 6.1 *Containment of Standard Conjunctive Queries [CM77]:*

Let Q_1 and Q_2 be conjunctive queries defined as follows:

Q_1 : $h(\bar{X}) :\text{-} g_1(\bar{X}_1), \cdots, g_k(\bar{X}_k)$

Q_2 : $h(\bar{Y}) :\text{-} p_1(\bar{Y}_1), \cdots, p_l(\bar{Y}_l)$

Q_2 is contained in Q_1 (denoted $Q_2 \sqsubseteq Q_1$) if and only if there exists a containment mapping from Q_1 to Q_2.

Theorem 6.2 *Containment of queries with arithmetic comparison[Klu88]:*

Let Q_1 and Q_2 be conjunctive queries defined as follows:

Q_1 : $h(\bar{X}) :\text{-} g_1(\bar{X}_1), \cdots, g_k(\bar{X}_k), \alpha_1, \cdots, \alpha_n$

Q_2 : $h(\bar{Y}) :\text{-} p_1(\bar{Y}_1), \cdots, p_l(\bar{Y}_l), \beta_1, \cdots, \beta_m$

where each α_i and β_j is of the form $A\theta B$ where A is a single variable and B is either a single variable or a constant. Let $C = \{\alpha_i | 1 \leq i \leq n\}$ and $D = \{\beta_j | 1 \leq j \leq m\}$ be the set of constraints in Q_1 and Q_2, respectively. Then, $Q_2 \sqsubseteq Q_1$ if and only if the implication $D \Rightarrow \mu_1(C) \vee \cdots \vee \mu_q(C)$ holds, where each μ_i is a containment mapping from Q_1 to Q_2.

It has been shown that the complexity of the containment test for standard case is NP-complete and for AC queries is Π_2^P-complete [Klu88, Mey92]. Note that the added complexity in containment test for AC queries compared to the standard case is due to the disjunction in the implication test defined in Theorem 6.2.

6.2.1 Containment of CLAC Queries

In this section, we show that Theorem 6.2 is also applicable to establish the containment for CLAC queries. The following Lemma 6.3 shows that every CLAC query can be transformed to an equivalent AC query with a set of auxiliary views [KS05b]. Using this, in Theorem 6.4, we show the requirements for containment of CLAC queries

are the same as AC queries [KS10].

Lemma 6.3 *Every CLAC query can be transformed to a conjunctive query with arithmetic comparisons using a set of auxiliary views.*

Proof *Let Q_1 be a CLAC query, defined as follows.*

$$Q_1: \quad h(\bar{X}) \text{ :- } g_1(\bar{X_1}), \cdots, g_k(\bar{X_k}), \alpha_1, \cdots, \alpha_n$$

where $\alpha_i = L_i \theta_i R_i$, in which L_i is the left hand side, R_i is the right hand side, and θ_i is a comparison operator.

For each constraint α_i in Q_1 and based on the subgoals in Q_1 that share some variable with L_i or R_i in α_i, we define an auxiliary view v_i, as follows.

$$V_i: \quad v_i(N_i, M_i, \bar{X_{i_1}}, \cdots, \bar{X_{i_n}}) \text{ :- } g_{i_1}(\bar{X_{i_1}}), \cdots, g_{i_n}(\bar{X_{i_n}}),$$
$$N_i = L_i,$$
$$M_i = R_i$$

where N_i and M_i in the head of v_i are two new variables defined based on L_i and R_i (appeared in α_i), and the rest of variables in the view are the variables/constants of the subgoals contributed to the definition of v_i, in the order in which the subgoals appear. Note that some variables might appear multiple times in the head of view, however, this is for ease of presentation of the proof and with a minor modification we can remove repeated variables from the head.

Also, note that not all the subgoals in the query body contribute to the definition of a constraint. For such subgoals, we define a view V_0. The difference between V_0 and other views is that in the head of V_0, we introduce no new variable.

$$V_0: \quad v_0(\bar{X_{0_1}}, \cdots, \bar{X_{0_n}}) \text{ :- } g_{0_1}(\bar{X_{0_1}}), \cdots, g_{0_n}(\bar{X_{0_n}})$$

Now, we define Q_2 using the set of views as follows:

$$\begin{aligned} Q_2: \quad h(\bar{X}) \text{ :- } & v_0(\bar{X_{0_1}}, \cdots, \bar{X_{0_p}}), \\ & v_1(N_1, M_1, \bar{X_{1_1}}, \cdots, \bar{X_{1_p}}),\ N_1\theta_1 M_1, \\ & \cdots, \\ & v_n(N_n, M_n, \bar{X_{n_1}}, \cdots, \bar{X_{n_p}}),\ N_n\theta_n M_n \end{aligned}$$

where θ_i is the comparison operator in α_i. Also, note that for each α_i in the original query Q, we considered a constraint $N_i\theta_i M_i$ in Q_2 which is in the form of arithmetic comparison.

It can be shown that $Q_2 \equiv Q_1$. For this, we unfold Q_2 using the definitions of the auxiliary views [Ull00]. After expansion, we get the same set of subgoals (some of which might be repeated) and the same set of arithmetic constraints (α_i's), because originally we defined every $N_i\theta_i M_i$ based on a constraint α_i. This means, we can transform every CLAC query to an AC query and a set of auxiliary views. ■

The following example illustrates details of the transformation process.

Example 6.4 *Transform the following CLAC query Q_1 to an AC query Q_1' and a set of auxiliary views.*

$$Q_1: \quad h(X) \text{ :- } r(X, Y, Z), s(Z, U, W), X + 2Y > Z,\ U \geq W$$

Here we have two constraints, each of which defines an auxiliary view.

1. *$(X + 2Y > Z)$ defines the auxiliary view V_1^1:*

$$\begin{aligned} V_1^1: \quad v_1^1(N_1, M_1, X, Y, Z, Z, U, W) \text{ :- } & r(X, Y, Z), s(Z, U, W), \\ & N_1 = X + 2Y, M_1 = Z \end{aligned}$$

2. *$(U \geq W)$ defines the auxiliary view V_2^1:*

$$V_2^1: \qquad v_2^1(N_2, M_2, Z, U, W) \text{ :- } s(Z, U, W), N_2 = U, M_2 = W$$

Note that since the variables in the first constraint come from subgoals r and s, both subgoals will contribute to the definition of v_1^1, i.e., the subgoals related to constraint $X + 2 > Y$ are $r(X, Y, Z)$ and $s(Z, U, W)$. However, since all the variables in the second constraint appear only in subgoal s, only s and its variables appear in the definition of v_2^1. Based on these two views, we define below an AC query Q_1' that is equivalent to Q_1.

$$Q_1': \quad h(X) \text{ :- } v_1^1(N_1, M_1, X, Y, Z, Z, U, W),\ N_1 > M_1,$$
$$v_2^1(N_2, M_2, Z, U, W),\ N_2 \geq M_2.$$

Unfolding the views will yield the same original query. Note that since the auxiliary views are conjunctive queries, we regard and treat each v_j^i in the body of Q_1' as a base relation.

Next, we define the containment for CLAC queries [KS10].

Theorem 6.4 *Let Q_1 and Q_2 be CLAC queries. Then $Q_2 \sqsubseteq Q_1$ iff the implication $D \Rightarrow \mu_1(C) \vee \cdots \vee \mu_q(C)$ holds, where C and D are linear arithmetic constraints in Q_1 and Q_2, respectively, and each μ_i is a containment mapping from Q_1 to Q_2.*

Proof *Based on Lemma 6.3, we create the following AC queries Q_1' and Q_2' which are equivalent to Q_1 and Q_2, respectively.*

$$Q_1': \quad h(\bar{X}) \text{ :- } v_0^1(\bar{X_{0_1}}, \cdots, \bar{X_{0_p}}), v_1^1(N_1^1, M_1^1, \bar{X_{1_1}}, \cdots, \bar{X_{1_p}}), \cdots,$$
$$v_n^1(N_n^1, M_n^1, \bar{X_{n_1}}, \cdots, \bar{X_{n_p}}), N_1^1\theta_1^1 M_1^1, \cdots, N_n^1\theta_n^1 M_n^1.$$

$$Q_2': \quad h(\bar{X}) \text{ :- } v_0^2(\bar{X_{0_1}}, \cdots, \bar{X_{0_q}}), v_1^2(N_1^2, M_1^2, \bar{X_{1_1}}, \cdots, \bar{X_{1_q}}), \cdots,$$
$$v_m^2(N_m^2, M_m^2, \bar{X_{m_1}}, \cdots, \bar{X_{m_q}}), N_1^2\theta_1^2 M_1^2, \cdots, N_m^2\theta_m^2 M_m^2.$$

It is easy to see that there is no containment mapping from Q_1' to Q_2', because their subgoals are distinct. In order to get the same set of containment mappings that exist from Q_1 to Q_2, we modify Q_2' while maintaining its equivalence to Q_2.

For this, we find all the containment mappings μ_j from Q_1 to Q_2. If there is no containment mapping, then the containment test fails. Otherwise, for every containment mapping μ_j, we apply μ_j on every subgoal of Q_1' (i.e., v_i^1) and append the new subgoal $\mu_j(v_i^1)$ to the body of Q_2'. We call the resulting query Q_2''. Note that $Q_2'' \equiv Q_2'$ because every newly added subgoal originates from a containment mapping from Q_1 to Q_2 whose target subgoals have already been in Q_2, and hence in Q_2'. Therefore, only some repeated subgoals are added to Q_2'', and $Q_2'' \equiv Q_2'$.

For instance, assume that u is the only containment mapping from Q_1 to Q_2. It maps subgoals in the body of Q_1 to a subset of subgoals in Q_2. Corresponding to this mapping, we append $v_i'' = \mu(v_i')$ to Q_2' and define Q_2''. If we unfold both Q_2' and Q_2'', we can see that unfolding Q_2'' might generate more subgoals all of which already appeared in the unfold of Q_2'.

Now, we consider the containment of Q_2'' in Q_1'. As these two queries are AC queries, we can apply Theorem 6.2. Note that we already have all the containment mappings from Q_1' to Q_2'', since we actually built Q_2'' based on such mappings, i.e., those from Q_1 to Q_2.

So, $Q_2'' \sqsubseteq Q_1'$ iff $D' \Rightarrow \mu_1(C') \vee \cdots \vee \mu_q(C')$, where D' is the set of constraints $N_i^2\theta_i^2M_i^2$ of Q_2'', and C' contains the constraints $N_j^1\theta_j^2M_j^2$ in Q_1'.

Now, if we replace $N_i^2\theta_i^2M_i^2$s and $N_j^2\theta_j^2M_j^2$s with their original definitions in the body of v_i^1s and v_j^2s, we get a new implication to be tested, i.e., $D \Rightarrow \mu_1(C) \vee \cdots \vee \mu_q(C)$, where D and C are the set of constraints in Q_2 and Q_1, respectively. ■

$$\begin{array}{ccc} Q_1 & \rightsquigarrow & Q_1' \\ \downarrow & & \downarrow \\ Q_2 & \rightsquigarrow & Q_2'' \end{array}$$

Figure 6.1: Transforming CLAC query into an AC query while maintaining the containment mapping

Figure 6.1 illustrates the steps of the above proof. Theorem 6.4 characterizes containment of CLAC queries. It confirms that containment of CLAC queries and AC queries have the same requirements and are of the same complexity. The following example illustrates details of Theorem 6.4.

Example 6.5 *Consider query Q_1 and Q_1' (its equivalent) from Example 6.4 together with query Q_2 defined as follows:*

$$Q_1: \quad h(X) \text{ :- } r(X,Y,Z), s(Z,U,W),\ X+2Y > Z,\ U \geq W.$$

$$Q_2: \quad h(X) \text{ :- } r(X,Y,Z), s(Z,U,W), s(Z,W,U),\ X > Z-10,\ 2Y > 0.$$

To test the containment of Q_2 in Q_1, based on Theorem 6.3, we first transform Q_1 and Q_2 to AC queries. The AC query equivalent to Q_1 is shown in Example 6.4, and the AC query equivalent to Q_2 is as follows:

$$V_1^2: \quad v_1^2(N_3, M_3, X, Y, Z, Z, U, W, Z, W, U) \text{ :- } r(X,Y,Z), s(Z,U,W), s(Z,W,U), N_3 = X, M_3 = Z-10.$$

$$V_1^2: \quad v_2^2(N_4, M_4, X, Y, Z) \text{ :- } r(X,Y,Z), N_4 = 2Y, M_4 = 0.$$

$$Q_2': \quad h(X) \text{ :- } v_1^2(N_3, M_3, X, Y, Z, Z, U, W, Z, W, U),\ N_3 > M_3, v_2^2(N_4, M_4, X, Y, Z),\ N_4 > M_4.$$

Next, we find the containment mappings from Q_1 to Q_2. Here, we get two sets of

containment mappings, shown as follows.

$\mu_1 = \{X/X, Y/Y, Z/Z, U/U, W/W\}$ *maps the body of* Q_1 *to subgoals* $r(X,Y,Z)$ *and* $s(Z,U,W)$, *and* $\mu_2 = \{X/X, Y/Y, Z/Z, U/W, W/U\}$ *maps the body of* Q_1 *to subgoals* $r(X,Y,Z)$ *and* $s(Z,W,U)$.

For every containment mapping μ_i, *we apply* μ_i *on every auxiliary view of the first query* $(\mu_i(v_j^1))$ *and append it to the body of* Q_2'. *That defines* Q_2''.

$$\begin{aligned} Q_2'': \quad h(X) :- & v_1^2(N_3, M_3, X, Y, Z, Z, U, W),\ N_3 > M_3 \\ & v_2^2(N_4, M_4, X, Y, Z),\ N_4 > M_4, \\ & v_1^1(N_1, M_1, X, Y, Z, Z, U, W, Z, W, U),\ v_2^1(N_2, M_2, Z, U, W), \\ & v_1^1(N_1', M_1', X, Y, Z, Z, W, U, Z, U, W),\ v_2^1(N_2', M_2', Z, W, U). \end{aligned}$$

We assume that the auxiliary views in the body of Q_2'' *and* Q_1' *are base relations. In Theorem 6.4, it is shown that* $Q_1 \equiv Q_1'$, *and* $Q_2 \equiv Q_2''$. *Therefore, if* $Q_2'' \sqsubseteq Q_1'$ *then* $Q_2 \sqsubseteq Q_1$. *Based on Theorem 6.2, we have to verify the following implication.*

$$\big[(N_3 > M_3) \wedge (N_4 > M_4)\big] \Rightarrow \Big[\big[(N_1 > M_1) \wedge (N_2 \geq M_2)\big] \vee \big[(N_1' > M_1') \wedge (N_2' \geq M_2')\big]\Big].$$

Now if we replace the variables by their definition, we get

$$\big[(X > Z - 10) \wedge (2Y > 0)\big] \Rightarrow \Big[\big[(X + 2Y > Z) \wedge (U \geq W)\big] \vee \big[(X + 2Y > Z) \wedge (W \geq U)\big]\Big].$$

This is nothing but the implication test for CLAC queries. It is easy to verify that in this example, Q_2 *is contained in* Q_1.

Note that to test containment of CLAC queries we do not need to transform them to AC queries.

So far, we showed that containment requirements for CLAC queries and AC queries are the same. Next, we explain the differences between the containment

of standard conjunctive queries and CLAC or AC queries.

Example 6.6 *Consider the following queries* Q_1 *and* Q_2*:*

Q_1: $h(X) :\text{-} p(X), r(Y,Z), Y \leq Z$

Q_2: $h(X) :\text{-} p(X), r(Y,Z), r(Z,Y)$

In order to test $Q_2 \sqsubseteq Q_1$*, we apply Theorem 6.2. For that, we find all the containment mappings from* Q_1 *to* Q_2*:* $\mu_1 = \{X/X, Y/Y, Z/Z\}$ *and* $\mu_2 = \{X/X, Y/Z, Z/Y\}$*, and accordingly, test the implication as follows.*

$D \Rightarrow \mu_1(C) \vee \cdots \vee \mu_q(C) \equiv$

$True \Rightarrow \mu_1(Y \leq Z) \vee \mu_2(Y \leq Z) \equiv$

$True \Rightarrow (Y \leq Z) \vee (Z \leq Y)$.

It is easy to see that no single term on the right hand side can be derived from the left hand side, however, the disjunction is always true, making the whole implication true.

Since testing the implication for disjunction is expensive, this example raises the question that are there classes of queries for which we do not need to consider the expensive test for disjunctions? Next, we introduce such queries and their syntactic characteristics.

6.2.2 Importance of Homomorphism Property

In previous section, we showed that the difference between the containment of standard conjunctive queries and CLAC queries is due to the disjunction in the implication test in Theorem 6.2. Previous studies ([Klu88, ALM04]) on containment of conjunctive queries with constraints identified classes of queries for which containment could be tested using a single containment mapping, hence the complexity of containment remains the same as in the standard conjunctive queries. This is defined using the notion of *homomorphism property.*

Definition 6.4 (Queries with Homomorphism property [Klu88]) *Let Q_1 and Q_2 be conjunctive queries. Q_1 and Q_2 have homomorphism property if containment of Q_2 in Q_1 could be tested using a single containment mapping.*

It is important to note that when homomorphism property holds, to reject the containment of Q_2 in Q_1, every containment mapping from Q_1 to Q_2 should be examined.

In the next section, we introduce two new classes of queries with homomorphism property which extend the class of AC queries with homomorphism property defined in [ALM04]. The desired characteristics of such a class H is that checking membership of a given pair of queries Q_1 and Q_2 in H is syntactically polynomial.

6.3 Classes of Queries with Homomorphism Property

In this section, we review extensions of conjunctive queries that enjoy the homomorphism property.

1. Conjunctive Queries with Equality Constraints (CQEC Queries) [KS05b, KS10]
2. Conjunctive Queries with Arithmetic Comparison (AC Queries) [KS10]
3. CQEC+AC Queries [KS10]

Next, we study these classes in details.

6.3.1 Conjunctive Queries with Equality Constraints

We define Conjunctive Queries with linear Equality Constraints (CQEC) as follows [KS05b].

Definition 6.5 (Conjunctive Queries with Linear Equality Constraints) *A CQEC query Q is a CLAC query in which comparison operator in the constraints is the equality (=). That is, every constraint α is of the form $E_l = E_r$, where E_l and E_r are linear expressions over the variables and constants in Q.*

Examples of such queries are as follows.

1. Q_1 : $h(X,T) \text{:-} r_1(X,H),\ r_2(X,N,R), T = H - R + 100N$

   Note that in this query, the head variable T is defined as an expression over the variables of the subgoals in the body.

2. Q_2 : $h(X) \text{:-} s_1(X,V_1,C_1,Y),\ s_2(X,V_2,C_2,Y_1), V_1 + C_1 = V_2 - C_2,$

   $Y_1 = Y + 1$

   In this query, the constraint part filters out the tuples for which the conditions $V_1 + C_1 = V_2 - C_2$ and $Y_1 = Y + 1$ are not satisfied.

As can be noted in the above examples, such constraints play two major roles in CQEC. A common major role is asserting a condition on the query. The other is defining a variable in the query head based on other variables. Next, we show that CQEC queries enjoy homomorphism property [KS10].

Theorem 6.5 (Containment of CQEC Queries) *Let Q_1 be a CQEC query and Q_2 any CLAC query. Then, homomorphism property holds for containment of Q_2 in Q_1.*

Proof *Applying Theorem 6.4, it is straightforward to transform Q_1 and Q_2 to conjunctive queries with arithmetic comparison in which Q_1 would have variable equality constraints (i.e., constraints in the form of $A = B$ where A and B are single variables). When testing implication, we can see that on the right hand side of the*

implication we will only have equality constraints that cannot form a "coupling" (a disjunction of terms on the right hand side where none of them alone is implied by the left hand side of the implication, however, together they are implied). As a result, either there exists a single term that satisfies the implication or the implication fails. This is nothing but homomorphism property. ■

6.3.2 Homomorphism Property and Conjunctive Queries with Arithmetic Comparison

Afrati et al. [ALM04] studied classes of queries with homomorphism property. These classes are special cases of AC queries. They introduced the conditions $\mathcal{L}_1$, $\mathcal{L}_2$, $\mathcal{L}_3$, $\mathcal{R}_1$, $\mathcal{R}_2$, and $\mathcal{R}_3$ that identify such classes [ALM04]. First, we review the conditions for queries with Left Semi-Interval comparisons.

- Left Semi-Interval Comparisons (LSI) [ALM04]:

  Let Q_1 be an AC query with Left semi-interval arithmetic comparisons only ($X < \alpha$ or $X \leq \alpha$, where X is a variable and α is a constant), and Q_2 is any AC query. If Q_1 and Q_2 satisfy all the following conditions, then homomorphism property holds. First, we define the terms used in these conditions.

  - Close-LSI: A constraint in the form of $X \leq \alpha$.
  - Open-LSI: A constraint in the form of $X < \alpha$.
  - Core(Q_1): the set of ordinary subgoals in the body of Q_1.
  - $AC(Q_1)$: the set of AC constraints in the body of Q_1.

  Note that similar terms are defined for RSI queries.

$\mathcal{L}_1$: *There are not subgoals as follows which all share the same constant* α*: An open-LSI subgoal in* $AC(Q_1)$*, a closed-LSI subgoal in the closure of* $AC(Q_2)$*, and a subgoal in* $core(Q_1)$. This basically prevents forming the following coupling:

$$X \leq \alpha \Rightarrow (X < \alpha \vee X = \alpha)$$

$\mathcal{L}_2$: *Either* $core(Q_1)$ *has no shared variable or there are not subgoals as follows which all share the same constant* α*: An open-LSI subgoal in* $AC(Q_1)$*, a closed-LSI subgoal in the closure of* $AC(Q_2)$ *and, a subgoal in* $core(Q_2)$. This basically prevents forming the following coupling:

$$(X \leq \alpha \wedge Y = \alpha) \Rightarrow (X < \alpha \vee X = Y)$$

$\mathcal{L}_3$: *Either* $core(Q_1)$ *has no shared variables or there are not subgoals as follows which all share the same constant* α*: An open-LSI subgoal in* $AC(Q_1)$ *and two closed-LSI subgoals in the closure of* $AC(Q_2)$. This basically prevents forming the following coupling:

$$(X \leq \alpha \wedge Y \leq \alpha) \Rightarrow (X < \alpha \vee Y < \alpha \vee X = Y)$$

- Right Semi-Interval Comparisons (RSI) [ALM04]:

  Three conditions, $\mathcal{R}_1$, $\mathcal{R}_2$, and $\mathcal{R}_3$ that are similar to those in LSI, $\mathcal{L}_1$, $\mathcal{L}_2$, and $\mathcal{L}_3$ but revised for RSI, identify when homomorphism property holds for a given pair of AC queries, Q_1 and Q_2.

- Afrati et. al also studied queries with both LSI and RSI constraints (Semi-Interval), and queries with Point-Inequality (constraints of the form $A \neq \alpha$

for constant α), and introduced classes with HP [ALM04]. We do not go into details of homomorphism propertyfor AC queries with Semi-Interval or Point-Inequality because we were not able to extend their results to CLAC queries with Semi-Interval or Point-Inequality constraints.

Example 6.7 *Consider the following queries Q_1 and Q_2 where Q_2 is contained in Q_1. Since Q_1 has only LSI constraints, we test the conditions $\mathcal{L}_1$, $\mathcal{L}_2$, and $\mathcal{L}_3$.*

Q_1 : $h(X)$:- $r(X, Y, 4),\ Y < 4,$

Q_2 : $h(X)$:- $r(X, A, 4), r(X, 3, A), A \leq 4$

Note that in this example, conditions $\mathcal{L}_1$ and $\mathcal{L}_2$ are applicable, however, neither one is satisfied. This is because the following three subgoals share the constant 4: (1) $Y < 4$ (open-LSI in Q_1), (2) $A \leq 4$ (closed-LSI in Q_2), and (3) $r(X, Y, 4)$ (a subgoal in $Core(Q_1)$). Accordingly, we cannot conclude whether homomorphism property holds or not, and to test the containment, we need to apply Theorem 6.2. For that, we find all the mappings and test if the constraint implication holds as follows. For Q_1 and Q_2, there are two mappings from Q_1 to Q_2, $\mu_1 = \{X/X, Y/A\}$, $\mu_2 = \{X/X, Y/3, (N' = 4)/A\}$.

$A \leq 4 \Rightarrow \mu_1(Y < 4) \vee \mu_2(Y < 4) \equiv$

$A \leq 4 \Rightarrow (A < 4) \vee (3 < 4 \wedge A = 4) \equiv$

$A \leq 4 \Rightarrow (A < 4) \vee (A = 4).$

That is, implication holds, but as we can see, it is based on a team up of two mappings and not a single mapping.

The following example illustrates a case where homomorphism property holds, however, the conditions in [ALM04] could not help detect this. In section 6.3.3, we extend these conditions which help identifying new classes of AC queries that satisfy homomorphism property.

Example 6.8 *Consider the following queries which are very similar to those in Example 6.7.*

Q_1 : $h() :\text{-} r(X, Y, Z),\ Y < 4,$

Q_2 : $h() :\text{-} r(X, A, 4), r(4, 3, A), A \leq 4$

For the same reason as in Example 6.7, the LSI test fails in this case which means we do not know if the homomorphism property holds. However, if we form the implication test, we can see that actually homomorphism property holds. The containment mappings we consider are $\mu_1 = \{X/X, Y/A, Z/4\}$, $\mu_2 = \{X/4, Y/3, Z/A\}$, *for which we have:*

$A \leq 4 \Rightarrow \mu_1(Y < 4) \vee \mu_2(Y < 4) \equiv$

$A \leq 4 \Rightarrow (A < 4) \vee (3 < 4).$

The implication holds since the constraint $3 < 4$ *in the disjunction evaluates the whole right hand side to* $True$*. That is, the containment test is established using only a single containment mapping.*

In the next section, we define new conditions to identify more AC queries that satisfy homomorphism property.

6.3.3 More AC Queries with Homomorphism Property

In our analysis of conjunctive queries with constraints that satisfy homomorphism property, we identified some other conditions that are necessary for forming a coupling in the implication test. We noted that if these conditions are violated, then the implication holds if and only if at least a single containment mapping satisfies the test, i.e., homomorphism property holds. The important point to note is that testing these conditions is polynomial. Moreover, these conditions are applicable to all subclasses of AC queries defined in section 6.3.2. This extends the classes of AC queries with

homomorphism property identified in [ALM04]. Before introducing these conditions, we first define some terms.

We define the notion of *Join-Closure* based on *connected subgoals* (see Definition 4.5), as follows [KS10].

Definition 6.6 (Variable Join-Closure) *Let Q be a conjunctive query, and A be a variable which appears in a subgoals $r(\bar{X})$ in the body of Q. The join closure of A, denoted $\mathbb{J}_A^*$ is the set of subgoals in the body of Q connected to $r(\bar{X})$.*

We use the term "repeated subgoal" to refer to occurrences of subgoals with the same predicate name in the body.

Let Q_1 be an AC query with Left Semi-Interval (LSI) arithmetic comparisons only ($X < c$ or $X \leq c$, where X is a variable and c is a constant), and Q_2 be any AC query. If Q_1 and Q_2 satisfy at least one of the following conditions, then the homomorphism property holds [KS10].

$\mathcal{L}_4$: *The variable in every closed-LSI in the closure of $AC(Q_2)$ has appeared in less than two repeated subgoals in $Core(Q_2)$.*

This basically prevents formation of the following couplings:

$$X \leq \alpha \Rightarrow (X < \alpha \vee X = \alpha),$$

$$(X \leq \alpha \wedge Y = \alpha) \Rightarrow (X < \alpha \vee X = Y) \text{ or}$$

$$(X \leq \alpha \wedge Y \leq \alpha) \Rightarrow (X < \alpha \vee Y < \alpha \vee X = Y)$$

where α is the shared constant.

It is easy to verify $\mathcal{L}_4$ by looking at the right hand side of these couplings. That is, variable X has appeared in at least two subgoals in $Core(Q_2)$, and in different positions; otherwise we would not have X in different disjunctions.

$\mathcal{L}_5$: *There is no repeated subgoal in $Core(Q_2)$, or the variable in the subject closed-LSI in the closure of $AC(Q_2)$ has not appeared in the repeated subgoals or it has appeared in the same positions in the repeated subgoals.*

Condition $\mathcal{L}_5$ does not allow forming any of the above three types of coupling, explained as follows. If there is no repeated subgoals then we do not have multiple containment mappings. If we have repeated subgoals and variable X in the closed-LSI has not appeared in the repeated subgoals or has appeared in the same position, then it cannot appear in two different constraints on the right hand side of disjunctions in the implication test, hence coupling cannot happen. That is, if the variable in closed-LSI is not connecting the repeated predicates then the homomorphism property exists. The following example illustrates details of $\mathcal{L}_5$.

Example 6.9 *Consider the following queries Q_1 and Q_2 where repeated subgoals in Q_2 do not share a variable. Condition $\mathcal{L}_1$ for homomorphism property of AC queries does not conclude that these queries satisfy homomorphism property, however, since variable A and constant 4 that appeared in the first subgoal do not appear in the second subgoal, the homomorphism property holds.*

Q_1 : $h() \text{:-} r(X, 4),\ X < 4$

Q_2 : $h() \text{:-} r(A, 4), r(3, D), A \leq 4.$

$\mathcal{L}_6$: *The variable A in the open-LSI appears in the head of Q_1.*

If $\mathcal{L}_6$ holds, then A cannot be mapped to different variables of Q_2. The reason is that in such cases the heads do not match and we do not have multiple containment mappings.

Also, we note that to form a coupling, it is vital that variable A in the open-LSI is mapped to different variables; otherwise, it would form the same constraint in the disjunction.

Example 6.10 *Consider the following queries that are similar to those in Example 6.7. Note that variable Y in open-LSI is a head variable and cannot be mapped to two variables.*

Q_1 : $h(Y) :\text{-} r(X, Y, Z),\ Y < 4,$

Q_2 : $h(A) :\text{-} r(X, A, 4), r(3, 4, A), A \leq 4$

The LSI conditions in [ALM04] cannot determine that the homomorphism property holds, however, our condition $\mathcal{L}_6$ implies this.

$\mathcal{L}_7$: Recall the notion of Variable Join-Closure defined in Section 6.3.3. Assume that A is the variable in the open-LSI in Q_1, and that there exists a variable B in the head of Q_1 where $\mathbb{J}^*_A = \mathbb{J}^*_B$. Let S be the set of join variables in the subgoals in $\mathbb{J}^*_B$. If there exists a variable $X \in S$ such that no two partial mappings map X to the same variable, then the homomorphism property holds.

The reason is that to form a coupling we need at least two containment mappings. Since B is a head variable it has to be mapped to the same target in every containment mapping so does every variable X that is a join variable in the subgoal of B and some other subgoal. If there is a variable X for which no two partial mappings map X to the same variable we know that coupling cannot happen.

The following example illustrates a case where $\mathcal{L}_7$ is applicable.

Example 6.11 *Consider the following AC queries.*

$$Q_1: \quad h(M) :\text{-} r(X,Y,Z), s(Z,M),\ Y < 4,$$
$$Q_2: \quad h(N) :\text{-} r(X,A,4), r(4,3,A), s(A,N),\ A \leq 4$$

The LSI conditions defined in [ALM04] cannot determine homomorphism property, however, based on the new condition $\mathcal{L}_7$, we now can conclude that the homomorphism property holds. The reason is that M is a head variable in Q_1 which is chained with Y (open-LSI variable), and there is only a single subgoal with a potential target N for M. To form the coupling, we need some containment mappings that map Y to different variables, and map Z to the same variable which is not possible.

Similar to conditions, $\mathcal{L}_4$, $\mathcal{L}_5$, $\mathcal{L}_6$, and $\mathcal{L}_7$ for LSI, we can define conditions $\mathcal{R}_4$, $\mathcal{R}_5$, $\mathcal{R}_6$, and $\mathcal{R}_7$, for RSI queries. Using these conditions we can identify more AC queries that enjoy homomorphism property.

Next, we introduce another subclass of CLAC queries for which we identify some necessary conditions of homomorphism property.

6.3.4 Homomorphism Property and Conjunctive Queries with Equality Constraints and Arithmetic Comparisons

In this section, we introduce Conjunctive Queries With Equality Expression and Arithmetic Comparison (CQEC+AC) which contains both equality constraints (defined in section 6.3.1) and LSI, RSI or SI constraints (defined in section 6.3.2), and identify some of the conditions under which CQEC+AC queries have homomorphism property.

Definition 6.7 (Queries with Equality Constraint and Arithmetic Comparison) *A CQEC+AC query Q is a CLAC query in which every constraint α_i is either of the*

form $E_l = E_r$ or $A\theta B$, where E_l and E_r are linear arithmetic expressions over variables and constants in Q, A is a variable, B is either a variable or a constant, and θ is a comparison operator in $\{=, \leq, <, >, \geq, \neq\}$.

In our analysis, we noted that adding equality expression to AC queries changes the way in which subgoals and variables are related. Intuitively, in CQEC+AC queries, a variable might not be used in a subgoal r but could be related to r indirectly through other variables. To formalize this, we define the notion of Variable-Subgoal relationship as follows [KS10].

Definition 6.8 (Variable-Subgoal Relationship) *Let Q be a CLAC query, and $s(\bar{X})$ be a subgoal in Q. If variable A can be defined as a function of the variables $\bar{X}$ in a $s(\bar{X})$, we say that A has appeared in $s(\bar{X})$, directly or indirectly. If the function is the identity function, then A has appeared directly; otherwise, A has appeared indirectly.*

Based on this notion, we extend conditions $\mathcal{L}_4$, $\mathcal{L}_5$, $\mathcal{L}_6$ and $\mathcal{L}_7$ for testing homomorphism property for CQEC+AC queries considering the cases where a variable appears in some subgoals indirectly.

Let Q_1 be a CQEC+AC query with Left semi-interval arithmetic comparisons and Q_2 be any CQEC+AC query. If Q_1 and Q_2 satisfy at least one of the following conditions, then the homomorphism property holds [KS10].

$\mathcal{L}'_4$: *The variable A in every closed-LSI in the closure of $AC(Q_2)$ has appeared in less than two repeated subgoals in $Core(Q_2)$, directly or indirectly.*

$\mathcal{L}'_5$: *There is no repeated subgoal in $Core(Q_2)$, or the variable A in the subject closed-LSI in the closure of $AC(Q_2)$ has not appeared in the repeated subgoals (directly or indirectly) or A has appeared in the same positions in the repeated subgoals (directly or indirectly).*

$\mathcal{L}'_6$: *The variable A in the open-LSI appears in the head of Q_1, directly or indirectly.*

Example 6.12 *Consider the following queries that are similar to those in Example 6.7. Note that variable W in the head can be defined based on the variable Y which is in the open-LSI hence Y cannot be mapped to two variables.*

Q_1 $\quad h(W) :\text{-} r(X,Y,Z),\ Y < 4, W = 2 * Y$

Q_2 $\quad h(A) :\text{-} r(X,A,4), r(3,4,A), A \leq 4$

The LSI conditions cannot determine homomorphism property, however, based on condition $\mathcal{L}'_6$, we can conclude that the homomorphism property exists.

$\mathcal{L}'_7$: Assume that A is the variable in the open-LSI in Q_1. Moreover, assume that there exists a variable B appeared in the head of Q_1, directly or indirectly, where $\mathbb{J}^*_A = \mathbb{J}^*_B$. Let S be the set of join variables in the subgoals in $\mathbb{J}^*_B$. If there exists a variable $X \in S$ such that no two partial mappings map X to the same variable, then the homomorphism property holds.

The following example shows that the conditions for identifying homomorphism property in AC queries are not applicable for CQAC+EC queries.

Example 6.13 *Consider the following queries Q_1 and Q_2.*

Q_1 $\quad h() :\text{-} r(X,Y),\ Y < 5, X + 1 = Y$

Q_2 $\quad h() :\text{-} r(A,B), r(3,A), A \leq 4, A + 1 = B.$

Here, since OLSI ($Y < 5$) and core(Q_2) $\{r(A,B), r(3,A)\}$ in this example do not share a constant, the conditions defined for AC queries in [ALM04] confirm that homomorphism property holds, however, this is not the case. That is, unlike AC queries,

in order to show that homomorphism property does not hold in CQAC+EC queries, the terms OLSI, CLSI, and a subgoal in core(Q_2) do not need to share the same constant. The implication test based on the containment mappings $\mu_1 = \{X/A, Y/B\}$ and $\mu_2 = \{X/3, Y/A\}$ is as follows.

$(A \leq 4 \wedge A + 1 = B) \Rightarrow (B < 5 \wedge A + 1 = B) \vee (A < 5 \wedge 3 + 1 = A) \equiv$

$(A \leq 4 \wedge A + 1 = B) \Rightarrow (A < 4 \wedge A + 1 = B) \vee (A = 4) \equiv$

$(A \leq 4 \wedge A + 1 = B) \Rightarrow (A < 4) \vee (A = 4) \equiv True.$

That means, $Q_2 \sqsubseteq Q_1$.

Similar to conditions, $\mathcal{L}'_4$, $\mathcal{L}'_5$, $\mathcal{L}'_6$, and $\mathcal{L}'_7$ for CQEC+LSI queries, we can define conditions $\mathcal{R}'_4$, $\mathcal{R}'_5$, $\mathcal{R}'_6$, and $\mathcal{R}'_7$ for CQEC+RSI queries.

6.4 Query Rewriting for CLAC Queries with Homomorphism Property

In this section, we extend our proposed pattern-based query rewriting algorithm ([KS09]) to support CLAC queries with Homomorphism Property.

Recall that a rewriting R of a query Q is a set of rules (conjunctive queries), each of which is contained in Q. We get the union of contained rules to generate the *Maximally Contained Rewriting.*

The following example illustrates query rewriting for CLAC queries with homomorphism property.

Example 6.14 *Let $r(A, B)$ and $s(C, D, E)$ be relations. Consider the following query and views:*

Q : $h(X, Y)$:- $r(X, Y)$, $s(Y, Z, W), X \leq 3, Z = 2$

V_1 : $v_1(A, B)$:- $r(A, B)$

V_2 : $v_2(B, D)$:- $s(B, D, D)$

V_3 : $v_3(A, C)$:- $r(A, D), s(D, C, D)$

Rule R below is a contained rewriting for Q. The reason is that if we unfold views V_1, V_2 *and* V_3 *[Ull00], we get a query which satisfies the containment* $R \sqsubseteq Q$.

R : $h(X, Y)$:- $v_1(X, Y)$, $v_2(Y, Z), X \leq 3, Z = 2$

$h(X, Y)$:- $v_3(X, Z), X \leq 3, Z = 2$

In example 6.14, there are coverages C_1=<$\{r(X, Y)\}, \{X/A, Y/B\}, v_1(A, B), \{\}$>, C_2=<$\{s(Y, Z, W)\}, \{Y/B, Z/D, W/D\}, v_2(B, D), \{W/Z\}$>, and C_3=<$\{r(X, Y), s(Y, Z, W)\}, \{X/A, Y/D, Z/C, W/D\}, v_3(A, C), \{\}$>, where C_1, C_2 and C_3 generate specializations $v_1(X, Y)$, $v_2(Y, Z)$, and $v_3(X, Z)$, respectively.

Next, we explain the steps of query rewriting for CLAC queries with homomorphism property.

6.5 Phases of Rewriting

In order to extend our query rewriting technique to support CLAC queries, we add a new step to the algorithm and assume that the input has homomorphism property. Rewriting of CLAC queries with homomorphism property consists of the following phases: (1) finding coverages, (2) combining coverages to generate rules, and (3) checking every generated rule to see if it satisfies the constraints in the query. Here, the first two phases are the same as in standard query rewriting, and the third step is added to handle the constraints in the input query and/or views.

6.5.1 Finding Coverages

Recall the notion of coverage defined in Chapter 3. There are two approaches to find coverage, Subgoal-based and View-based. Here, we repeat the subgoal-based

approach from Section 3.3.1. To find coverages, we consider a set S with a single subgoal sg_i in the query Q and its joint variables J_S. For every view V_j that includes sg_i, we check whether all the variables A in J_S are accessible through V_j, that is, A is distinguished in V_j. If this is the case, we create a new coverage C_{ji} based on V_j and assign S to C_{ji}. Otherwise, we add more subgoals to S, update J_S accordingly, and inquire if V_j can be useful in forming a coverage.

Adding subgoals to S is based on the join variables that are not accessible. If $A \in J_S$ is not accessible, we include all the subgoals in Q in which A has appeared. After adding the new subgoals to S, we recalculate joint variables J_S of S and repeat the test. This process terminates if a basic coverage for sg_i (and possibly some other subgoals) is found, or all possible subgoals are added to S, and there is at least one joint variable that is not accessible.

6.5.2 Combining Coverages

As explained in Section 3.3.2, there are three approaches to combine coverages: (1) Detection and Recovery using simple Cartesian Product, (2) Avoidance using optimized Cartesian Product, and (3) Prevention using Pattern-based Cartesian Product. Here, we briefly review the third approach, the Pattern-based Cartesian Product [KS09]. This approach is based on the idea of identifying non-overlapping patterns to break an original Cartesian product into a set of smaller Cartesian Products. It assigns to a query Q with n subgoals, the number $2^n - 1$, and assigns an identifier to every coverage C_i that is based on the subgoals it contains and their positions in the query body. For instance, in Example 6.14, the identifiers assigned to C_1, C_2 and C_3 would be 2 ($= 10_{binary}$), 1 ($= 01_{binary}$), and 3 ($= 11_{binary}$) , respectively. The identifiers that do not have overlap define the patterns. For example, here, we have two patters $\{3\}$ and $\{1, 2\}$.

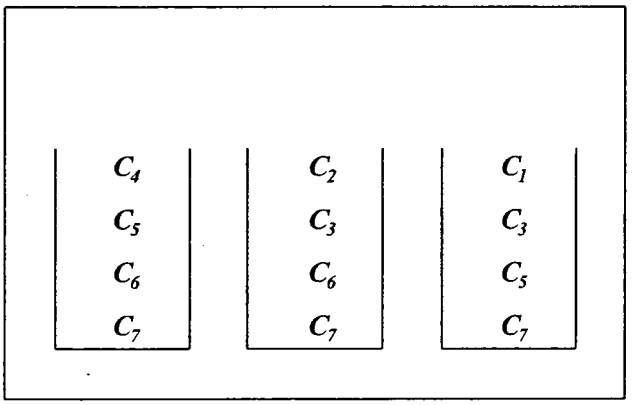

Figure 6.2: All possible occurrences of coverages for a query with 3 subgoals.

For each pattern $\mathbb{P}$, we create the buckets and perform a simple Cartesian product. For the coverages listed in Figure 6.2, we break down the original large bucket into a few smaller buckets and perform 5 different Cartesian products (since there are 5 different pattern combinations) listed as follows.

1. $\mathbb{P}_1 = \{\{7\}\}$: No need for Cartesian product
2. $\mathbb{P}_2 = \{\{3\}, \{4\}\}$: Cartesian product of two buckets: $[C_3] \times [C_4]$
3. $\mathbb{P}_3 = \{\{6\}, \{1\}\}$:Cartesian product of two buckets: $[C_6] \times [C_1]$
4. $\mathbb{P}_4 = \{\{5\}, \{2\}\}$: Cartesian product of two buckets:$[C_5] \times [C_2]$
5. $\mathbb{P}_5 = \{\{4\}, \{2\}, \{1\}\}$: Cartesian product of three buckets: $[C_4] \times [C_2] \times [C_1]$

Note that the bitwise "and" of the numbers in each pattern $\mathbb{P}_i$ is 0 and their sum is $7=2^3 - 1$, e.g., for $\mathbb{P}_5 = \{\{4\}, \{2\}, \{1\}\}$, 7=1+2+4, and the values 1, 2, and 4 are bitwise disjoint. This makes sure that the coverages in the same pattern are non-overlapping and they cover all the subgoals in the query.

6.5.3 Handling the Constraints

For every combination generated in the previous step, we create a rule R_i, and check R_i to see if it is contained in Q. For this, we need to see if the constraints in Q can be satisfied by the constraints in the views used in R_i. In this case, we add R_i to the rewriting R. Recall that the rewriting R is the union of contained conjunctive queries each of which is contained on Q. For every rule R_i, we may have three possibilities, as follows [KS10].

1. The constraints C in Q are satisfied by the constraints in the views used in R_i. In this case, we simply add R_i to the result set R.

2. A set of constraints C' can be added to R_i so that C' and the constraints in the views of R_i satisfy C. In this case, we add C' to R_i, and add R_i to R.

3. None of the above. We discard R_i.

Finally, we return R, the union of R_is, as a maximally contained rewriting of Q.

6.6 Discussion

So far, we have investigated classes of conjunctive queries with homomorphism property. We believe identifying classes of queries for which homomorphism property does not hold but containment test can be done efficiently is also important. The following example illustrates this point.

Example 6.15 *Consider the following queries Q_1 and Q_2, for which we test containment of Q_2 in Q_1.*

Q_1: $\quad h(X) :\text{-}\ p(X), r(Y, Z), Y < Z.$

Q_2: $h(X)$:- $p(X), r(Y,Z), r(Z,Y)$.

Intuitively, the constraint $Y < Z$ in Q_1 cannot be covered by Q_2, hence Q_2 is not contained in Q_1. The reason is that if $Y = Z$ then Q_1 would return no answer while Q_2 would have the same tuples X as in relation $P(X)$.

Example 6.16 *Consider the following queries Q_1 and Q_2, for which we test containment of Q_2 in Q_1.*

Q_1: $h(Y)$:- $p(Y), r(Y,Z), Y \leq Z$.

Q_2: $h(Y)$:- $p(Y), r(Y,Z), r(Z,Y)$.

In this example, variable Y appears in the constraint $Y \leq Z$ and also in the query head. Here, $Q_2 \not\sqsubseteq Q_1$. The reason is that Q_1 includes only values Y in P for which there is a tuple (Y,Z) in r such that $Y \leq Z$ in r. However, Q_2 contains more tuples than Q_1 because tuples (Y,Z) in r in Q_2 do not need to satisfy the constraint $Y \leq Z$.

Example 6.17 *Consider the following queries Q_1 and Q_2, for which we test containment of Q_2 in Q_1.*

Q_1: $h(X)$:-$p(X), r(Y,Z), Y \leq Z$.

Q_2: $h(X)$:-$p(X), r(Y,Z), s(Y,Z), s(Z,Y)$.

Here, subgoal s in Q_2 is repeated, however, this does not form several containment mappings which required for having implicit constraint. Moreover, Q_2 does not imply constraint $Y \leq Z$ explicitly, hence $Q_2 \not\sqsubseteq Q_1$.

The following examples illustrate queries Q_1 and Q_2 with implicit constraints, where $Q_2 \sqsubseteq Q_1$.

Example 6.18 (Queries without homomorphism property) *Consider the following conjunctive queries Q_1 and Q_2.*

Q_1: $h(X) :- p(X), r(U, Y, Z), Y \leq Z$

Q_2: $h(X) :- p(X), r(M, Y, Z), r(N, Z, W), r(K, W, Y)$

To test the containment of Q_2 in Q_1, the constraint $Y \leq Z$ in Q_1 together with the containment mappings from Q_1 to Q_2 (formed by the special pattern of the repetition of variables in the predicates of Q_2) establish the following implication, and hence confirm the containment of Q_2 in Q_1.

$True \Rightarrow (Y \leq Z) \vee (Z \leq W) \vee (W \leq Y)$.

We show that these kind of implications can be verified in an efficient way.

To distinguish such queries, we check whether the following conditions hold. Consider the containment of an AC query Q_2 in an AC query Q_1. If the following conditions hold, then Q_2 would imply the constraints in Q_1.

1. Constraint in Q_1 is of the form $A\theta B$, where θ is $\leq$ or $\geq$.

2. Variables A and B in the constraint in Q_1 appear in the same subgoal r and do not appear in any other subgoals including the head.

3. Q_2 includes several occurrences of the same subgoal r that share a variable with the constraint in Q_1.

4. The variables in Q_2 that correspond to the variable in the constraint of Q_1 form a cycle in n occurrence of the corresponding subgoals r in Q_2, for some integer $n \geq 1$ (see Theorem 6.6). The following theorem captures these conditions.

Based on the conditions discussed above, the following theorem identifies a class of conjunctive queries that do not enjoy homomorphism property however their containment can be tested efficiently.

Theorem 6.6 (Queries without Homomorphism property) *Let Q_1, Q_2, and Q_1' be conjunctive queries, defined as follows. Then $Q_2 \sqsubseteq Q_1$ if $Q_2 \sqsubseteq Q_1'$.*

$$Q_1: \quad h(\bar{X}) :\text{-} p_1(\bar{X}_1), p_2(\bar{X}_2), \cdots, r(\underbrace{\cdots}_{i}, Y, \underbrace{\cdots}_{j}, Z, \underbrace{\cdots}_{k}), Y\theta Z, \alpha.$$

$$Q_1': \quad h(\bar{X}) :\text{-} p_1(\bar{X}_1), p_2(\bar{X}_2), \cdots, r(\underbrace{\cdots}_{i}, Y, \underbrace{\cdots}_{j}, Z, \underbrace{\cdots}_{k}), \alpha.$$

That is, Q_1' is Q_1 without the constraint $Y\theta Z$.

$$\begin{aligned} Q_2: \quad & h(\bar{X}) :\text{-} p_1(\bar{X}_1), p_2(\bar{X}_2), \cdots, q_1(\bar{Y}_1), q_2(\bar{Y}_2), \cdots, \\ & r(\underbrace{\cdots}_{i}, Y, \underbrace{\cdots}_{j}, W_1, \underbrace{\cdots}_{k}), \\ & r(\underbrace{\cdots}_{i}, W_1, \underbrace{\cdots}_{j}, W_2, \underbrace{\cdots}_{k}), \cdots, \\ & r(\underbrace{\cdots}_{i}, W_n, \underbrace{\cdots}_{j}, Y, \underbrace{\cdots}_{k}), \beta. \end{aligned}$$

where θ is either $\leq$ or $\geq$, and the variables in $\bar{X}$ and $\{Y, Z, W_1, \cdots, W_n\}$ are disjoint.

Proof *Since $Q_2 \sqsubseteq Q_1'$, there exists some containment mappings from Q_1' to Q_2. Since Q_1 and Q_1' have the same structure in the regular predicates, we have the same set of containment mappings from Q_1 to Q_2. The only difference is in the constraint $Y\theta Z$ in Q_1 for which we apply Theorem 6.2 or 6.4. That is, we have to test one of the following implications:*

(1) $\beta \Rightarrow (Y \leq W_1) \vee (W_1 \leq W_2) \vee \cdots \vee (W_1 \leq Y)$, for θ being $\leq$, and

(2) $\beta \Rightarrow (Y \geq W_1) \vee (W_1 \geq W_2) \vee \cdots \vee (W_1 \geq Y)$, for θ being $\geq$.

Let us assume the implication in case (1) is false. This is possible only if β is true and $(Y \leq W_1) \vee (W_1 \leq W_2) \vee \cdots \vee (W_1 \leq Y)$ is false, which means its negation is true, as shown below.

$$(Y \leq W_1) \vee (W_1 \leq W_2) \vee \cdots \vee (W_1 \leq Y) \equiv F \Longrightarrow$$

$$not \Big((Y \leq W_1) \vee (W_1 \leq W_2) \vee \cdots \vee (W_1 \leq Y) \Big) \equiv T \Longrightarrow$$

$$(Y > W_1) \wedge (W_1 > W_2) \wedge \cdots \wedge (W_1 > Y)) \equiv T \Longrightarrow$$

$(Y > W_1) \wedge (W_1 > Y)) \equiv T$.

But this is a contradiction. Therefore, our original assumption was wrong and hence the implication case (1) is true. Case 2 can be shown in a similar way. ■

Note that since identifying such queries is based on cycle detection which is polynomial in complexity, it is reasonable to consider this test for a given query before doing the actual containment test.

6.7 Summary

In this chapter, we studied the problem of containment and rewriting for conjunctive queries with linear arithmetic constraints, and identified new classes of queries that enjoy the homomorphism property. Based on this, we extended our query rewriting technique to support conjunctive queries with linear arithmetic constraints.

Chapter 7

Conclusion and Future Work

7.1 Conclusion

In this thesis, we investigated the problems of minimization and rewriting of conjunctive queries. Even though these problems are not new and a number of good solutions have been proposed, there has been increasing interest to improve scalability and efficiency issues. We considered conjunctive queries and investigated the problems of rewriting and minimization of such queries under open world assumption and set semantics. The main contributions in this thesis are as follows.

- We investigated the two phases in rewriting, i.e., finding coverages and combining coverages, and identified that combining coverages is a partitioning problem. Based on this, we proposed a formula that uses Stirling numbers to determine the number of rules in the rewriting and developed a algorithm that exploits some numeral patterns to break the large combinatorial problem into several smaller ones.

  Since the patterns are independent of queries, one can pre-generate patterns

for queries with different sizes and use them during rewriting. However, experiments results showed that the overhead for pattern generation compared to rewriting time is not considerable.

- We summarized the approaches for each phases in rewriting of conjunctive queries as follows.
  - Finding Coverages
    1. Subgoal-based
    2. View-based
  - Combining Coverages
    1. Detection and Recovery
    2. Avoidance
    3. Prevention
- Based on this, we proposed our pattern-based rewriting algorithm that uses view-based approach for finding coverages, and prevention approach for combining them.
- We performed a comprehensive set of experiments for different types and sizes of queries to evaluate efficiency, memory requirement and scalability of our pattern-based rewriting technique compared to the existing ones, results of which indicate superiority of our solution, for instance, two orders of magnitude in some cases.
- We considered quality of rewriting as another important aspect in this problem, and defined the number of subgoals in the rewriting as a measure for quality. We classified the rewriting algorithms into two classes: bottom-up query rewriting and top-down query rewriting. We showed that a top-down query rewriter

generates rewriting with a better quality compare to bottom-up query rewriter, i.e., there are fewer number of subgoals in the rewriting in a top-down approach.

- We considered conjunctive queries, and proposed an efficient minimization algorithm that uses special endomorphism together with some heuristics to iteratively identify and remove redundant subgoals from a given query.

- We showed that by exploiting our query minimization technique as a post-processing phase in the rewriting, a bottom-up rewriting technique can also generate a rewriting with the same quality as in a top-down technique.

- We considered the case where input query and views contain redundant subgoal and showed that in order to obtain the optimal quality in the rewriting, input query and views should be minimized. That is, we exploit our query minimization algorithm as pre-processing and post-processing phases of our pattern-based rewriting technique. To evaluate the performance and quality of our solution technique, we performed extensive experiments on differen types of inputs, result of which shows that pattern based technique coupled with our minimization algorithm not only reaches the optimal rewriting quality but also outperforms existing techniques, namely, Minicon and Treewise.

- We defined CLAC queries as a practical form of conjunctive queries with constraints, and identified several classes of CLAC queries that satisfy homomorphism property, i.e., classes of queries for which, despite the presence of constraints, the complexity of containment remains NP-complete. Accordingly, we extended our query rewriting algorithm to support CLAC queries that satisfy homomorphism property.

7.2 Future Work

There are a number of interesting problems related to both query rewriting and query minimization that we intend to work on. These problems are mostly related to queries with constraints. In general, we have two types of constraints: constraints inside the query and constraints outside the queries (e.g., functional dependencies). Some of these problems are discussed as follows.

7.2.1 Query Rewriting and Functional Dependencies

In the context of query rewriting, considering functional dependencies raises new research issues and adds to the difficulty of the problem. Looking at a closely related problem, query containment, adding functional dependencies as extra information adds to the complexity of the problem [JK82]. We expect to see the same in the context of query rewriting. The following is a motivating example that shows the impact of functional dependencies on the result of rewriting.

Example 7.1 *Consider the following query Q and views V_1 and V_2.*

Q : $h(X,Y,Z) :- r(X,Y,Z).$

V_1 : $v_1(A,B) :- r(A,B,C).$

V_2 : $v_2(A,C) :- r(A,B,C).$

View V_1 and V_2 cannot contribute to a contained rewriting for Q. To see this, we consider view V_1, using which we try to build a coverage for subgoal r in Q. We assign $S = \{r(X,Y,Z)\}$, and find the joint variables for S: $J_S = \{X,Y,Z\}$. Here, variable Z is a joint attribute but it is not accessible based on V_1. Therefore, there is no coverage for subgoal r based on V_1. For the same reason, there is no coverage for r based on V_2, which means there is no rewriting for Q based on V_1 and V_2.

Next, assume that the following set of functional dependencies (that holds on relation r) is provided as extra information.

$F = \{X \rightarrow Y, X \rightarrow Z\}$

In the presence of functional dependencies F, the following is the maximally contained rewriting for Q.

Q' : $\quad h(X,Y,Z) :- v_1(X,Y), v_2(X,Z).$

The reason is that V_1 and V_2 could be considered as relations generated in a lossless join decomposition of r. Therefore, joining V_1 and V_2 doesn't create any extra tuple [GUW08]. As a result, the rewriting, Q', is a contained rewriting for Q in the context of conjunctive queries.

7.2.2 Query Minimization and Linear Arithmetic Constraints

The problem of query minimization is closely related to the problem of query equivalence and hence query containment. Since constraints, in general, and linear arithmetic constraints in particular, are essential part of queries in practice, studying query minimization in the context of queries with such constraints is important. It becomes even more important when we consider query minimization in the context of applications such as query rewriting that generate queries automatically and hence there is more possibility of having redundant subgoals.

7.2.3 Query Minimization and Functional Dependencies

There have been studies on the complexity of query containment in the presence of functional dependencies [JK82, CDL98]. In general, considering functional dependencies as extra information during query minimization adds to the complexity of the problem. Therefore, finding efficient minimization algorithms in the presence of functional dependencies is another interesting problem to investigate.

Bibliography

[AD98] Abiteboul, Serge and Duschka, Oliver. Complexity of answering queries using materialized views. In *PODS '98: Proceedings of the 17th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 254–263. ACM, 1998.

[AGK99] Afrati, Foto, Gergatsoulis, Manolis, and Kavalieros, Theodoros. Answering queries using materialized views with disjunctions. In *ICDT '99: Proceedings of the 7th International Conference on Database Theory*, pages 435–452. Springer-Verlag, 1999.

[ALM02] Afrati, Foto, Li, Chen, and Mitra, Prasenjit. Answering queries using views with arithmetic comparisons. In *PODS'02: Proceedings of the 21st ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, pages 209–220, 2002.

[ALM04] Afrati, Foto, Li, Chen, and Mitra, Prasenjit. On containment of conjunctive queries with arithmetic comparisons. In *EDBT'04: Proceedings of the 9th International conference on Extending Database Technology*, pages 459–476, 2004.

[BDD+98] Bello, Randall G., Dias, Karl, Downing, Alan, Feenan, Jr., James,

Finnerty, James L., Norcott, William D., Sun, Harry, Witkowski, Andrew, and Ziauddin, Mohamed. Materialized views in oracle. In *VLDB'98: Proceedings of the 24rd International Conference on Very Large Data Bases*, pages 659–664. Morgan Kaufmann, 1998.

[Ber01] Bernstein, Philip A. Generic model management: A database infrastructure for schema manipulation. In *CoopIS'01: Proceedings of the 9th International Conference on Cooperatice Information Systems*, pages 1–6. Springer-Verlag, LNCS-2172, 2001.

[BLRR97] Beeri, Catriel, Levy, Alon Y., Rajaraman, Anand, and Rousset, Marie-Christine. Rewriting queries using views in description logics. In *PODS'97: Proceedings of the 16th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 99–108. ACM Press, 1997.

[BR00] Bernstein, Philip A. and Rahm, Erhard. Data warehouse scenarios for model management. In *ER'00: Proceedings of the International Conference on Conceptual Modeling / the Entity Relationship Approach*, pages 1–15. Springer-Verlag, LNCS-1920, 2000.

[CDL98] Calvanese, Diego, De Giacomo, Giuseppe, and Lenzerini, Maurizio. On the decidability of query containment under constraints. In *PODS'98: Proceedings of the 17th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 149–158. ACM Press, 1998.

[CDL+01] Calvanese, Diego, De Giacomo, Giuseppe, Lenzerini, Maurizio, Nardi, Daniele, and Rosati, Riccardo. Data integration in data warehousing. *International Journal of Cooperative Information Systems*, 10(3):237–271, 2001.

[CDLV99] Calvanese, Diego, De Giacomo, Giuseppe, Lenzerini, Maurizio, and Vardi, Moshe Y. Rewriting of regular expressions and regular path queries. In *PODS'99: Proceedings of the 18th ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, pages 194–204. ACM Press, 1999.

[CKPS95] Chaudhuri, Surajit, Krishnamurthy, Ravi, Potamianos, Spyros, and Shim, Kyuseok. Optimizing queries with materialized views. In *ICDE'95: Proceeding of IEEE International Conference on Data Engineering*, pages 190–200. IEEE Computer Society, 1995.

[CM77] Chandra, A.K. and Merlin, P.M. Optimal implementation of conjunctive queries in relational databases. In *Proceeding of the 9th Annual ACM Symposium on the Theory of Computing*, pages 77–90. ACM Press, 1977.

[CNS99] Cohen, Sara, Nutt, Werner, and Serebrenik, Alexander. Rewriting aggregate queries using views. In *PODS'99: Proceedings of the 18th ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, pages 155–166. ACM Press, 1999.

[Cod70] Codd, Edgar. A relational model for large shared data banks. *Communications of the ACM*, 13:337–387, 1970.

[CR00] Chekuri, Chandra and Rajaraman, Anand. Conjunctive query containment revisited. *Theoretical Computer Science*, 239(2):211–229, 2000.

[DG97a] Duschka, Oliver M. and Genesereth, Michael R. 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. ACM Press, 1997.

[DG97b] Duschka, Oliver M. and Genesereth, Michael R. Query planning in infomaster. In *Proceedings of the 1997 ACM Symposium on Applied Computing*, pages 109–111. ACM, 1997.

[DL00] Genesereth, Michael R. Duschka, Oliver M. and Levy, Alon Y. Recursive query plans for data integration. *Journal of Logic Programming*, 43(1):49–73, 2000.

[Dus97] Duschka, Oliver M. Recursive plans for information gathering. In *IJCAI'97: Proceedings of the 15th International Joint Conference on Artificial Intelligence*, pages 778–784. Morgan Kaufmann, 1997.

[FCS00] Fernandez, Mary F., Chiew Tan, Wang, and Suciu, Dan. Silkroute: trading between relations and xml. *Journal of Computer Networks*, 33(1-6):723–745, 2000.

[FRV95] Florescu, Daniela, Rashid, Louiqa, and Valduriez, Patrick. Using heterogeneous equivalences for query rewriting in multidatabase systems. In *CooipIS'95: Proceedings of the International Conference on Cooperative Information Systems*, pages 158–169. CoopIS-95, 1995.

[FRV96] Florescu, Daniela, Rashid, Louiqa, and Valduriez, Patrick. Answering queries using OQL view expressions. In *Workshop on Materialized Views, in conjunction with ACM SIGMOD*, pages 84–90. ACM, 1996.

[GHQ95] Gupta, Ashish, Harinarayan, Venky, and Quass, Dallan. Aggregate-query processing in data warehousing environments. In *VLDB'95: Proceedings of the 21th International Conference on Very Large Data Bases*, pages 358–369. Morgan Kaufmann, 1995.

[GM99] Grahne, Gösta and Mendelzon, Alberto O. Tableau techniques for querying information sources through global schemas. In *ICDT'99: Proceedings of the 7th International Conference on Database Theory*, pages 332–347. Springer-Verlag, 1999.

[GM05] Gupta, Himanshu and Mumick, Inderpal Singh. Selection of views to materialize in a data warehouse. *IEEE Transactions on Knowledge and Data Engineering*, 17(1):24–43, 2005.

[GSUW94] Gupta, Ashish, Sagiv, Yehoshua, Ullman, Jeffrey D., and Widom, Jennifer. Constraint checking with partial information. In *PODS'94: Proceedings of the 13th ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, pages 45–55. ACM Press, 1994.

[GUW08] Garcia-Molina, Hector, Ullman, Jeffrey, and Widom, Jennifer. *Database Systems: The Complete Book.* Prentice Hall, 2nd edition, 2008.

[Hal00] Alon Y. Halevy. Theory of answering queries using views. *SIGMOD Record*, 29:40–47, 2000.

[Inm96] Inmon, W.H. *Building the Data Warehouse, 2nd Edition.* John Wiley & Sons, New York, NY, 1996.

[IS99] Ibarra, Oscar H. and Su, Jianwen. A technique for proving decidability of containment and equivalence of linear constraint queries,. *Journal of Computer and System Sciences*, 59(1):1 – 28, 1999.

[JK82] Johnson, D. S. and Klug, A. Testing containment of conjunctive queries under functional and inclusion dependencies. In *PODS'82: Proceedings of the 1st ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*, pages 164–169. ACM, 1982.

[JLVV03] Jarke, M., Lenzerini, M., Vassiliou, Y., and Vassiliadis, P. *Fundamentals of Data Warehouses.* Springer-Verlag, LNCS-760, 2nd, revised and extended edition, 2003.

[Klu88] Klug, A. On conjunctive queries containing inequalities. *Journal of the ACM*, pages 35(1): 146–160, 1988.

[KS02] Kunen, I. and Suciu, D. A scalable algorithm for query minimization. Technical report, University of Washington, 2002.

[KS05a] Kiani, Ali and Shiri, Nematollaah. Answering queries in heterogenuous information systems. In *IHIS'05: Proceedings of ACM Workshop on Interoperability of Heterogeneous Information Systems*, pages 17–24. ACM, 2005.

[KS05b] Kiani, Ali and Shiri, Nematollaah. Containment of conjunctive queries with arithmetic expressions. In *CoopIS'05: Proceedings of 13th International Conference on Cooperative Information Systems*, pages 439–452. Springer-Verlag, LNCS-3760, 2005.

[KS09] Kiani, Ali and Shiri, Nematollaah. Using patterns for faster and scalable rewriting of conjunctive queries. In *Proceedings of the 3rd Alberto Mendelzon International Workshop on Foundations of Data Management.* CEUR-WS.org, 2009.

[KS10] Kiani, Ali and Shiri, Nematollaah. Conjunctive queries with constraints: Homomorphism, containment and rewriting. In *FoIKS'10: Proceedings of the 6th International Symposium on Foundations of Information and Knowledge Systems*, pages 40–57. Springer, LNCS-5956, 2010.

[Len02] Lenzerini, Maurizio. Data integration: a theoretical perspective. In *Proceedings of the 21st ACM SIGMOD-SIGACT-SIGART Symposium on Principles of database systems*, pages 233–246. ACM, 2002.

[Lev01] Levy, Alon Y. Answering queries using views: A survey. *The VLDB Journal*, 10(4):270–294, 2001.

[LMSS95] Levy, Alon Y., Mendelzon, Alberto O., Sagiv, Yehoshua, and Srivastava, Divesh. Answering queries using views. In *PODS '95: Proceedings of the 14th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 95–104. ACM, 1995.

[LRO96] Levy, Alon Y., Rajaraman, Anand, and Ordille, Joann J. Querying heterogeneous information sources using source descriptions. In *VLDB'96: Proceedings of the 22nd International Conference on Very Large Databases*, pages 251–262. Morgan Kaufmann, 1996.

[LRU96] Levy, Alon Y., Rajaraman, Anand, and Ullman, Jeffrey D. Answering queries using limited external processors. In *PODS'96: Proceedings of the 15th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 227–237. ACM, 1996.

[LS92] Levy, Alon and Sagiv, Yehoshua. Constraints and redundancy in datalog. In *PODS'92: Proceedings of the 11th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 67–80, New York, NY, USA, 1992. ACM Press.

[Mai83] Maier, D. *The Theory of Relational Databases.* Computer Science Press, Rockville, Maryland, 1983.

[Mey92] Meyden, Ron van der. The complexity of querying indefinite data about linearly ordered domains. In *PODS'92: Proceedings of the 11th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 331–345. ACM, 1992.

[Mit01] Mitra, Prasenjit. An algorithm for answering queries efficiently using views. In *Proceedings of the 12th Australasian Database Conference*, pages 99–106. IEEE Computer Society, 2001.

[MRB03] Melnik, Sergey, Rahm, Erhard, and Bernstein, Philip A. Rondo: a programming platform for generic model management. In *Proceedings of the 22nd ACM SIGMOD International Conference on Management of Data*, pages 193–204. ACM Press, 2003.

[MS08] Mohajerin, Nima and Shiri, Nematollaah. A top-down approach to rewriting conjunctive queries using views. In *Proceedings of the 3rd International Workshop on Semantics in Data and Knowledge Bases*, pages 180–198. Springer-Verlag, 2008.

[PL00] Pottinger, Rachel and Levy, Alon Y. A scalable algorithm for answering queries using views. *The VLDB Journal*, pages 484–495, 2000.

[PTU00] Palopoli, L., Terracina, G., and Ursino, D. The system dike: Towards the semi-automatic synthesis of cooperative information systems and data warehouses. In *Proceedings of ADBIS-DASFAA Symposium*, pages 108–117. Matfyz Press, 2000.

[Qia96] Qian, Xiaolei. Query folding. In *ICDE '96: Proceedings of the 12th International Conference on Data Engineering*, pages 48–55. IEEE Computer Society, 1996.

[Rio58] Riordan, John. *Introduction to Combinatorial Analysis.* Dover Publication, Mineola, NY, 1958.

[SRV95] Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of Databases.* Addison Wesley, Reading, MA, 1995.

[TS97] Theodoratos, Dimitri and Sellis, Timos. Data warehouse configuration. In *VLDB'97, Proceedings of 23rd International Conference on Very Large Data Bases*, pages 126–135. Morgan Kaufmann, 1997.

[Ull89] Ullman, Jeffrey D. *Principles of Database and Knowledge-Base Systems*, volume II. Computer Science Press, Maryland, 1989.

[Ull00] Ullman, Jeffrey D. Information integration using logical views. *Theoretical Computer Science*, 239(2):189–210, 2000.