A Top-Down Approach to Answering Queries Using Views

Nima Mohajerin

A Thesis

in

The Department

of

Computer Science and Software Engineering

Presented in Partial Fulfillment of the Requirements

for the Degree of Master of Computer Science at

Concordia University

Montréal, Québec, Canada

March 2008

Your file Votre référence
ISBN: 978-0-494-40949-7
Our file Notre référence
ISBN: 978-0-494-40949-7

ABSTRACT

A Top-Down Approach to Answering Queries Using Views

Nima Mohajerin

The problem of answering queries using views is concerned with finding answers to a query using only answers to a set of views. In the context of data integration with LAV approach, this problem translates to finding maximally contained rewriting for a query using a set of views. When both query and views are in conjunctive form, rewritings generated by existing bottom-up algorithms in this context are generally expensive to evaluate. As a result, they often require costly post-processing to improve efficiency of computing the answer tuples.

In this dissertation, we propose a top-down approach to the rewriting problem of conjunctive queries. We first present a graph-based analysis of the problem and identify conditions that must be satisfied to ensure maximal containment of rewriting. We then present TreeWise, a novel algorithm that uses our top-down approach to efficiently generate maximally contained rewritings that are generally less expensive to evaluate. Our experiments confirm that TreeWise generally produces better quality rewritings, with a performance comparable to the most efficient of previously proposed algorithms.

ACKNOWLEDGEMENTS

This thesis would not exist without the valuable guidance and support of my supervisor, Dr. Nematollaah Shiri. I thank him for his generosity with his time, dedication, and his valuable advices at crucial times. I thank Ali Kiani for our countless fruitful discussions and many great ideas he suggested. Indeed, many parts of this work were inspired by our discussions and his suggestions. I also thank Dr. Rachel Pottinger for her instant replies to all of our questions regarding her algorithm and for providing us with the source code of her sample generator. Finally, I thank all of my friends for reminding me to have fun at times when I was too wrapped-up in research and for giving me a better perspective on life.

To my father who taught me the value of optimism, patience, and living life to the fullest. To my mother for believing in me at times when even I did not believe in myself. And to my sister for setting a very hard example to follow.

TABLE OF CONTENTS

LIST OF FIGURES

Chapter 1

Introduction

As the popularity of data integration grows, and as more sources decide to share data and information everyday, the problem of answering queries using views receives more attention from the scientific community. Informally speaking, a mediator-based data integration process involves logically combining multiple independent data sources into an integrated system with a mediated global schema. Once the mediated schema is defined and data sources are linked to this schema, users can pose queries in the integrated system without being concerned with the actual structures of individual sources. The integrated system is now responsible for generating answers to user's queries through its links to the data sources.

Among several approaches proposed for designing links between mediated schema and data sources in a data integration system, one is the Local-As-Views approach, which is the focus of this thesis. In this context, the problem of answering queries becomes synonymous with the problem of rewriting queries using a set of views.

There are several algorithms proposed for rewriting queries using views in the LAV approach to data integration. All the main algorithms introduced in this context employ

a bottom-up approach to the problem, which may result in rewritings that are expensive to evaluate in the sense that generated rewritings include unnecessarily many subgoals. That is thus our motivation in this research to improve the situation by taking a top-down approach and propose a new efficient rewriting algorithm that produces rewritings with fewer subgoals.

1.1 Answering Queries Using Views in Data Integration Frameworks

As opposed to data warehousing, which is mainly concerned with physical merging of data, data integration concentrates on loosely linking data sources by logical means. Simply put, a data integration framework is made up of three parts: a set of autonomous data sources, a logical mediated schema representing the overall structure of the integrated framework, and descriptions of links between data sources and the mediated schema.

The mediated schema in a data integration framework is virtual and sources are physical sources that can have many different forms including relational databases, XML pages, etc. Two main tasks in designing a data integration framework are to first generate the logical mediated schema representing the overall structure of data in the system and then to create links between mediated schema and the actual data sources.

The mediated schema is designed manually for a data integration framework. One approach in creating links between the mediated schema and data sources is to define sources as views over the mediated schema. In the literature, this approach is known as Local-As-Views (LAV). The advantage of this approach is the separation of the logical schema from physical sources, and therefore easy addition of new data sources to the

system, if desired. In this context, the main task in generating answers to user's query is to rewrite the query using only the views representing the data sources. By doing so, the integrated framework will then evaluate the query by obtaining the answers to the views and combining them properly to get the answer to user's query.

The following example illustrates a data integration framework in the LAV approach, and also shows what is involved in query rewriting using views in the context.

***Example* 1:** The city of Montreal decides on a plan to provide its citizens with a new unified library system to access information regarding books available in the city's public libraries. However, each library has its own independent database system under operation and the cost of merging all of them into a single source can be quite expensive and prohibitive. In addition, libraries may not wish to make all the data in their databases available to public, they only want to expose selected information to the outside world.

This scenario, even though naive, shows situations where data integration can help reduce the costs and provide an elegant and affordable solution. By creating a mediated schema that would represent the global structure of this unified information system and by logically linking independent data sources to this schema, data sources maintained by each library can continue their routine operation without any changes, and the integrated framework can be created without any actual merging of data. For example, the following can be part of the global schema of a desired integrated system:

```
Book(id, title, publisher-id, year-published, genre)

Publisher(id, name, address, country)
```

```
Library(id, name, location, municipality)

Is_Located(book-id, library-id, status)
```

However, individual library databases may have schemas that are different from the global schema above. To create links between this mediated schema and schemas of individual data sources, we can follow the LAV approach to represent data sources as views over the global schema. Accordingly, consider the two views expressed below in SQL, which represent two library sources in the city.

```
create view MR_Local_Books as
SELECT Book.title, Book.publisher-id, Library.name
FROM Book, Publisher, Is_Located, Library
WHERE Book.publisher-id = Publisher.id
      AND Publisher.country = 'Canada'
      AND Is_Located.book-id = Book.id
      AND Is_Located.library-id = Library.id
      AND Library.municipality = 'Mont-Royal';


create view Fiction_Books as
SELECT Book.title As title, Library.name As library,
       Publisher.name As publisher, Publisher.address As address,
       Publisher.country As Country
FROM Book, Publisher, Is_Located, Library
WHERE Book.publisher-id = Publisher.id
       AND Is_Located.book-id = Book.id
       AND Is_Located.library-id = Library.id
       AND Book.genre = 'Fiction';
```

The first view V_1 defines all the books published by Canadian companies, available in libraries located in Mont-Royal municipality. The second view V_2 lists all the books classified as fiction, along with the information about their publishers. At this point, users can query the mediated schema without being concerned about the structure of each individual data sources located in libraries across the city. An example of such queries may be to list titles of all books available in the libraries published by Canadian firms, along with the names of the libraries at which we can find such books. This query Q can be expressed in SQL as follows:

```
SELECT Book.title, Library.name
FROM Book,Publishers,Is_Located,Library
WHERE Book.publisher-id =publisher.id
      AND Book.id = Is_Located.book-id
      AND Is_Located.library-id = Library.id
      AND Publisher.country = 'Canada';
```

Since the data sources V_1 and V_2 and the query Q are defined over the mediated schema, it is possible to reformulate the query using only views defined over these data sources. By doing so, we can then use the answer tuples provided by V_1 and V_2 in order to generate the answer to Q. Intuitively, only those reformulations of the query are acceptable that produce results that form a subset of results that we would get by executing the original query itself. Also, the goal is to generate the maximally contained rewriting, that is to get largest possible subset of answers using the defined views. With these requirements in mind, the above query can be reformulated as the following query using the given two view definitions.

```
SELECT title, library
FROM Fiction_Books
WHERE country = 'Canada'
        UNION
SELECT title,library
FROM MR_Local_Books;
```

□

The above example shows the essence of answering queries using views in the LAV approach to data integration system. This reformulation of a query using a given set of view definitions is also known as query rewriting problem.

1.2 Rewriting Problem and Maximal Containment

The problem of rewriting queries using views has been studied extensively in database and artificial intelligence research. This problem is not only related to data integration, but also to query optimization in databases, which aims to find a better rewriting and execution plan for a given query using a set of materialized views. In the query optimization context, the focus is on finding an equivalent rewriting that yields the cheapest execution plan [Hal01].

In a data integration context using the LAV approach, finding an equivalent rewriting in not always possible. In such cases finding a *contained* rewriting is our next best choice. This is due to the fact that data sources may not cover the entire domain [Hal01] and therefore only partial answers to the query would be available. In this case, however, only those rewritings are desired which produce the largest collection of answers, also known

as *maximally contained* rewritings. If both query and views are in conjunctive form (i.e., select-project-join) with no comparison predicates in the query, a maximally contained rewriting will be in the form of union of conjunctive queries.

One classification of the previous efforts related to rewriting problem in the context of data integration is based on the types of query and views allowed. Due to complexity constraints, the previous efforts mainly concentrate on conjunctive queries. Depending on the presence of comparison predicates, conjunctive queries are divided into two classes: standard conjunctive queries and conjunctive queries with built-in predicates. There are several algorithms proposed for rewriting in the standard case. They include the Bucket algorithm [LRO96a], inverse-rules [Dus97], and the Minicon algorithm [PH01] which is the best known algorithm. For conjunctive queries with built-in predicates, there has been some work with less progress due to increased complexity of the problem. In this direction, [ALM02] and [PH01] proposed extensions to the MiniCon algorithm to support a subset of built-in predicates known as Left-Semi-Interval (LSI) queries.

In data integration, there are two possible assumptions often made regarding the contents of the data sources that can affect the rewriting problem in a subtle way. The first is that data sources are assumed to be complete, meaning that the data sources defined include all the tuples in their definitions. The second assumption is that some tuples might be missing from the definitions of data sources. The former assumption is known as Closed-World Assumption (CWA), while the latter is called Open-World Assumption (OWA). It has been shown in [AD98] that the problem of answering queries using views under CWA is computationally harder.

In this work, we consider standard conjunctive queries under OWA and propose

an efficient top-down algorithm, called TreeWise, that has several advantages over the Minicon algorithm, the best known bottom-up rewriting algorithm.

1.3 Thesis Contribution

This thesis investigates the problem of answering queries using views in data integration systems that use the LAV approach to defining views over the mediated schema. We introduce an algorithm for rewriting a Select-Project-Join SQL query also known as conjunctive query, over the mediated schema into union of conjunctive queries over a given set of defined conjunctive views.

Our first contribution is the introduction of a new graph-based model for representing conjunctive queries, called Hyper-node Model for Conjunctive Queries (HMCQ). We use HMCQ and identify conditions that must be satisfied to guarantee maximally contained rewriting in our top-down approach solution. By properly utilizing this model, our second contribution is a new top-down algorithm for rewriting , called TreeWise, that guarantees maximally contained rewritings for standard conjunctive queries and views. With a top-down approach, TreeWise efficiently produces better quality rewritings, compared to bottom-up rewriting techniques mentioned above.

We perform numerous experiments to evaluate the efficiency and quality of the rewritings generated by the TreeWise algorithm. For this, we consider Minicon as a representative of previous algorithms to compare with TreeWise. The results show that in many test cases our top-down, view-based approach produces better quality rewritings in less time than the bottom-up, subgoal-based approach of Minicon.

1.4 Thesis Outline

The next chapter provides a background for our work and reviews concepts and techniques used in this study. In addition to a brief review of graph theory, it includes fundamental concepts and definitions related to query containment, certain answers, and the rewriting problem.

Chapter 3 provides a taxonomy of previous work related to query rewriting. We begin by presenting the results regarding the complexity of the problem for different types of queries and views. The chapter concludes with a description of previous rewriting algorithms and their strengths and weaknesses.

Chapters 4 and 5 contain the main contribution of this thesis. Chapter 4 introduces a new graph-based model we proposed for conjunctive queries. Chaper 5 follows by introducing our top-down approach for rewriting using this model. It concludes by presenting the details of TreeWise algorithm, which implements the proposed approach.

In chapter 6, we describe details of our experiments using TreeWise and performance evaluation results. We compare the results with performance and quality of Minicon. Chapter 7 includes concluding remarks and a list of future work.

Chapter 2

Background

This chapter reviews basic concepts and techniques we need in our work. They include conjunctive queries, query containment, answering queries using views , as well as concepts and definitions from graph theory. Since this dissertation revolves around the language of conjunctive queries, we start this chapter by describing this language.

2.1 Conjunctive Queries and Views

This thesis studies the problem of answering queries using views. It is thus important to define what type of queries are the subject of our study. The main focus of this thesis is on a subset of the language of conjunctive queries, called standard conjunctive queries. This class corresponds to Select-Project-Join queries in SQL.

Definition 2.1. *(Conjunctive Query) A conjunctive query Q is a non-recursive datalog rule of the form:*

$$Q: \quad q(\overline{X}) :- \quad p_1(\overline{X_1}), \ldots, p_n(\overline{X_n})$$

where q, $p_1, \ldots, p_n$ are predicate names. The atomic formula $q(\overline{X})$ is called the head, and $p_1(\overline{X_1}), \ldots, p_n(\overline{X_n})$ are the subgoals in the rule body. Predicates in the body can either

be ordinary subgoals or built-in predicates. The tuples $\overline{X}, \overline{X_1}, \ldots, \overline{X_n}$ *are the arguments of the query, which can contain either attribute variables or constants. The variables in* $\overline{X}$ *are called the* distinguished *variables of the query, denoted as* distVars(Q). *All other variables in Q are* existential *variables, denoted as* existVars(Q). *We assume that Q is a* safe *query meaning that every variable appearing in* $\overline{X}$ *also appears in* $\overline{X_1} \cup \ldots \cup \overline{X_n}$. *All variables will be denoted by uppercase letters. Union of conjunctive queries is expressed as a set of conjunctive queries having the same head predicate. We use* $Q(D)$ *to denote the set of tuples obtained by evaluating Q over the input database instance D.*

Definition 2.2. *(Standard Conjunctive Query) A standard conjunctive query Q is a conjunctive query in the form* $q(\overline{X}) :- \quad p_1(\overline{X_1}), \ldots, p_n(\overline{X_n})$, *where* $p_1(\overline{X_1}), \ldots, p_n(\overline{X_n})$ *are all ordinary subgoals (no built-in predicates).*

The main focus of this thesis is on standard conjunctive queries. There are several classes of conjunctive queries with built-in predicates. Since there are references throughout this thesis to these classes, they are described as follow.

The first subset of conjunctive query language with built-ins uses comparison predicates in the form of $X\theta Y$, where X is a variable, and Y can be either variable or constant and θ is a comparison operator from the set $\{<, \leq, >, \geq, =\}$. The subset so defined is called *Conjunctive Queries with Arithmetic Comparison (CQAC).*

Another subset, introduced in [Klu88] and later extended in [FP04], is called *Left-Semi-Interval (LSI)* queries, which forms a subset of CQAC. This subset of conjunctive queries allows built-in predicates in form of $X\theta C$, where X is a variable, C is a constant, and θ is an operator in $\{\leq, <\}$. If the operator $\leq$ is removed from this set, the subset is called *Open-LSI* queries. Corresponding to these subsets, there are *Right-Semi-Interval (RSI)* and *Open-RSI* queries.

Another class of conjunctive queries, introduced in [Ali05], allows linear equality arithmetic expressions. This subset is called *CQEL* for short.

In the context of this thesis, a view is a named conjunctive query. A view instance is a set of tuples obtained by evaluating the view over the database instance.

2.2 Query Containment and Equivalence

Query containment and equivalence provide means of comparison between answers of two conjunctive queries for any database instance. In the context of this thesis, we use these concepts to verify proper reformulation of a query using the views. Simply put, a query Q_1 is contained in query Q_2, if for every instance of database D, $Q_1(D)$ is a subset of $Q_2(D)$. If for every instance of D, $Q_1(D)$ and $Q_2(D)$ are the same, we say that Q_1 and Q_2 are equivalent.

Extensive research has been devoted to the problem of query containment and equivalence. Chandra and Merlin [CM77] studied the problem for standard conjunctive queries and showed that containment mapping is the necessary and sufficient condition for containment. In [GSUW94], the containment problem was studied for CQACs and later on Klug examined *homomorphism property* of LSI-queries and its role in the problem [Klu88]. More recently, Afrati et al. [FP04] studied the containment problem in the context of LSI and open-LSI queries.

Next we recall definition of containment and its necessary and sufficient condition in the standard case, namely the containment mapping.

Definition 2.3. *(Containment Mapping) Given two standard conjunctive queries Q_1 and Q_2, a containment mapping ρ from Q_2 to Q_1, denoted by $\rho : Q_2 \rightarrow Q_1$, is a symbol*

mapping which is identity on the constants and predicate names such that 1) $head(Q_1) = \rho(head(Q_2))$*, and 2)* $\rho(body(Q_2)) \subseteq body(Q_1)$*.* $\rho(head(Q))$ *and* $\rho(body(Q))$ *respectively represent the head and body of* Q *after* ρ *is applied to* Q*.*

Definition 2.4. *(Partial Mapping) Let* Q_1 *and* Q_2 *be conjunctive queries. A partial mapping from* Q_1 *to* Q_2 *is a mapping restricted to some subset of the variables in* Q_1*.*

Definition 2.5. *(Standard Containment) Let* Q_1 *and* Q_2 *be standard conjunctive queries. We say query* Q_1 *is contained in* Q_2*, denoted by* $Q_1 \sqsubseteq Q_2$*, iff there exists a containment mapping from* Q_2 *to* Q_1*.*

Definition 2.6. *(Query Equivalence) Two conjunctive queries* Q_1 *and* Q_2 *are equivalent, denoted by* $Q_1 \equiv Q_2$*, iff* $Q_1 \sqsubseteq Q_2$ *and* $Q_2 \sqsubseteq Q_1$*.*

In the case of conjunctive queries with built-in predicates, the notion of containment differs from definition 2.5. In case of CQAC, it was shown that for containment to hold, *constraint implication* must also hold [GSUW94], defined as follows.

Definition 2.7. *(Constraint Implication) Let* Q_1 *and* Q_2 *be conjunctive queries with arithmetic comparison. We say* Q_1 *is contained in* Q_2*, if the following implication holds:*

$D \Rightarrow (\rho_1(C) \vee \ldots \vee \rho_k(C))$*, where* C *and* D *are the constraints in* Q_2 *and* Q_1 *respectively, and* $\rho_1, \ldots, \rho_k$ *are all the containment mappings from* Q_2 *to* Q_1*.*

In general, the implication above can be constructed from multiple containment mappings. As shown in [Klu88], when the homomorphism property holds, the disjunction in the implication above reduces to testing for only one containment mapping.

2.3 Answering Queries in Data Integration Systems

The problem of answering queries using views is concerned with finding answers to a query Q over a database schema using only the answers to a set of view definitions $V_1, \ldots, V_j$ over the same schema. This problem is related to a number of data management problems including query optimization, data warehouse, semantic data catching in client-server systems [KB94, DFJ+96]. In this thesis, we focus on the application of this problem in the context of data integration.

A data integration system provides a uniform interface to a multitude of autonomous (possibly heterogeneous) data sources [Hal01]. Querying multiple data sources on the World-Wide Web or integration of data from distributed systems are examples of data integration applications. With data integration systems, users are not bothered with locating sources relevant to their queries and communicating with each source, nor they are concerned about combining the results of different sources to find answers to their original queries. Instead, they pose queries over a *mediated schema* provided by the integrated framework. Using a set of *source descriptions*, the task of data integration system is to translate user query into one that refers directly to the schemas of data sources. There are two important issues involved in this process: design of the mediated schema and the approach used to describe data sources.

Typically mediated schema is designed manually for a particular data integration system. However as for source descriptions, several approaches have been proposed in the literature. Two main approaches for this are Global-As-Views (GAV) and Local-As-Views (LAV). In the GAV approach, the mediated schema is defined in terms of the data

sources. The LAV approach on the other hand, describes the data sources as views over the mediated schema. The main issue with GAV approach is modeling of the data and hence the design and maintenance of the mediated schema. The main challenge in LAV is query processing; the modeling of data is an easier task.

2.3.1 Query Rewriting

In the context of data integration using the LAV approach, the query translation problem amounts to finding ways to find the answer to the query over the mediated schema using a set of view definitions over the same schema. Next we formally define this reformulation of the query also known as rewriting.

Definition 2.8. *(Rewriting) Given a query Q and a set of view definitions $V = \{V_1, \ldots, V_n\}$, a rewriting of Q using the views in V is a query R whose ordinary subgoals are all from V.*

Intuitive, only rewritings are useful that are contained in the original query. It is noteworthy that the above definition is intended for *complete* rewritings [LMS95], as opposed to a rewriting that may also include predicates from the mediated schema in the query body. Since, in the context of data integration, user posts queries over mediated schema that is virtual, only a complete rewriting will be desired. Furthermore, since views in this context may not entirely cover the domain of mediated schema, rewriting can be either equivalent to or contained in the original query. To determine containment or equivalence of a rewriting to the original query, we use the unfolding technique, defined below.

Definition 2.9. *(Query Unfolding) Given a query Q and a set $V = \{V_1, \ldots, V_n\}$ of view definitions, unfolding of a rewriting R of Q is a query R' equivalent to R that is obtained by replacing each view V_i in the body of R with the body of V_i. Existential variables in view V_i are replaced by fresh variables in R'.*

Since finding all answers to a query in a data integration system is not always possible, contained rewritings are the best we can hope for. In this case, we need to determine a rewriting that returns best possible set of answers to the query. This rewriting is called *maximally contained rewriting*, and is a language dependent notion.

Definition 2.10. *(Maximally Contained Rewriting) Given a language L, query Q, and a set of view definitions $V = \{V_1, \ldots, V_n\}$, a query R is a Maximally contained (MC) rewriting of Q using V with respect to L if:*

1. *R is expressed in L and $R \sqsubseteq Q$.*
2. *$R' \sqsubseteq R$, for every contained rewriting R' of Q using V, where R' is expressed in L.*

As stated above, a MC rewriting R is defined with respect to a specific language, and depending on the language we choose, R may or may not exist. This fact contributes and is related to the expressive power of the language and the type of query and views being considered. If query and views are standard conjunctive queries, MC rewriting exists in the language of union of standard conjunctive queries (from now on, we call each conjunctive query in the union, a *rewriting query*). Furthermore, it was shown that the number of subgoals in each rewriting query in this union will be no more than the number of ordinary subgoals in the query. [LMS95]

After describing what type of rewriting returns the best answer to a query, the following question arises: what is the set of answers to the query that returns a maximally

contained rewriting? This brings us to the notion of *certain answers* [AD98], which we briefly describe in the following section.

2.3.2 Certain Answers and Maximally Contained Rewriting

Informally speaking, given a query Q, a view definition V and an instance I of V, certain answers are those tuples that are in the answer to Q, for each possible database instance that yields the instance I for the view V. However determining whether or not a tuple is a certain answer depends on our assumption on view instance I. In this regard there are two possibilities, as described in [AD98].

With *Closed World Assumption (CWA)*, we assume that view instance I is complete, that is, I includes all the tuples that satisfy view definition V. Alternatively, in *Open World Assumption (OWA)*, we assume that I is not complete in that it may be missing some tuples that satisfy the definition of V. A formal definition of these two assumptions can be found in [Rac04, GM99], where the notions of sound and complete view definitions were introduced. A sound view definition is one for which results of the view are tuples defined by the body of the view. A complete view definition provides all tuples that match it. Naturally all view definitions are assumed to be sound. Under OWA, view definitions are assumed to be sound, but not complete. A good example of such views is presented in [GM99].

In [AD98], it was shown that due to the reasoning involved, computing certain answers under CWA is harder than OWA. Next we provide a formal definition of certain answer under both assumptions.

Definition 2.11. *(Certain Answer) Given a query Q, view definition V_i and its instance*

I, a tuple t is a certain answer under OWA, if t is in $Q(D)$ for each database instance D with $I \subseteq V_i(D)$. Under CWA, t is a certain answer if t is in $Q(D)$ for each database instance D with $I = V_i(D)$.

In the context of data integration using the LAV approach, views are not assumed to be complete and hence OWA is considered. On the basis of this assumption, the set of all certain answers is the answer to our earlier question on the maximum number of answers we can find for a standard conjunctive query using a set of standard conjunctive views under OWA. In [AD98], it was shown that in this context, maximally contained rewriting returns exactly this set of tuples.

2.4 Graph Theory: Notations and Concepts

In this section, we briefly review notations and concepts related to graph theory used throughout this thesis.

Definition 2.12. *(Undirected graph) An undirected graph G is defined as a pair (V, E) where V is a finite nonempty set of elements, called nodes or vertices, and $E \subseteq V \times V$ is a set of unordered pairs in V, called edges.*

We say that an edge $< x, y >$ in G is *incident* with vertices x and y. Furthermore, x and y are said to be *adjacent* to each other. The edge $< x, x >$ is called a *loop* and the vertex n that has no incident edges is called an *isolated* vertex. If set E is empty, we call G an *edgeless* graph.

In graph G, two edges e_1 and e_2 that share a vertex n are called adjacent. A *walk* is a sequence of adjacent edges.

Definition 2.13. *(Path) Let x, y be vertices in an undirected graph $G = (V, E)$. A path from x to y in G, denoted as $x - y$, is a (loop-free) finite alternating sequence*

$$x = x_0, e_1, x_1, e_2, x_2, e_3, \ldots, e_{n-1}, x_{n-1}, e_n, x_n = y$$

of distinct vertices and edges from G, starting at vertex x and ending at vertex y and involving the n edges $e_i =< x_{i-1}, x_i >$, where $1 \leq i \leq n$. The length of this path is n. In other words, a path *is a walk where no edge is repeated. A* simple path *is a path where no node is repeated. When $x = y$, this path is called a cycle.*

***Example* 2:** Figure 2.1 illustrates an undirected graph G on $V = \{a, b, c, d, e\}$ with edges $E = \{< a, b >, < a, c >, < b, c >, < b, d >, < d, d >\}$.

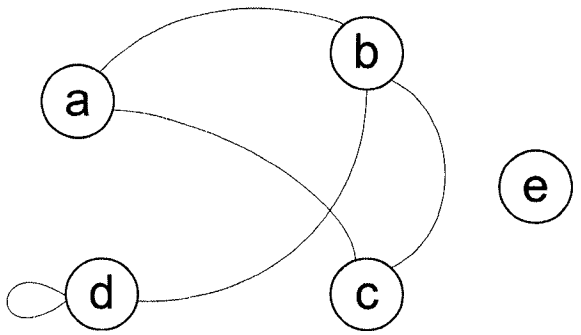

Figure 2.1: Graph of Example 2

Here, e is an isolated vertex. The edge $< d, d >$ is a loop in G, and there are two paths from a to d of lengths 2 and 3. The edges $< a, b >, < b, c >, < c, a >$ form an $a - a$ cycle. □

The edges in an undirected graph represent a symmetric relation on the vertices. If every vertex in the graph has a loop, that is $< x, x > \in E$, for all $x \in V$, then the relation represented by the edges E is reflexive.

Definition 2.14. *(Connected graph) A graph $G = (V, E)$ is connected if there is a path between every two distinct vertices in G. A graph that is not connected is said to be disconnected.*

For a disconnected graph $G = (V, E)$, the set V can be partitioned into subsets $V_1, \ldots, V_n$, where $n \geq 2$, such that there is no edge $< x, y >$ in E, where $x \in V_i$, and $y \in V_j$, for $i \neq j$. Each of the partitions $V_1, \ldots, V_n$ along with its respective edges in E that form a connected graph, is called a *connected component* of G. Hence, an undirected graph is connected if and only if it has only one connected component, that is, there is only one partition defined by G.

Definition 2.15. *(Transitive closure graph) Let $G = (V, E)$ be an undirected graph. The transitive closure of G is a graph $G^+ = (V, E^+)$, where the edge $< x_i, x_j > \in E^+$ if and only if there is a path in G between x_i and x_j.*

Definition 2.16. *(Hypergraph) A generalized graph $G = (V, E)$ is called a hypergraph where E is a set of generalized edges with some edges connecting more than two nodes in V.*

Chapter 3

Related Work

This chapter reviews previous work related to the rewriting problem. We also discuss the complexity of the problem for different classes of queries and views. Since this problem is related to the containment problem, there is also a discussion of complexity of containment problem in the opening section. In the remainder of the chapter, we review details of main existing rewriting algorithms and their strengths and limitations.

3.1 Complexities of Containment and Rewriting Problems

In this section we briefly review results on complexity of containment and rewriting problems for different types of queries and views. Since the rewriting problem is closely related to the containment problem, it is instructive to study the complexity of the latter problem to better understand the complexity of the rewriting problem. In [CM77] the problem of containment for standard conjunctive query is shown to be NP-complete. In [Klu88], this problem is shown to be in $\prod_2^P$ when queries are in the form CQAC. However, for LSI queries the complexity is in NP, due to existence of homomorphism property [Klu88]. Finding other classes of conjunctive queries which also enjoy homomorphism property is

an open problem [Klu88]. In [Ali05] they consider the problem for CQEL, a class of conjunctive queries with linear equality constraints. There are also studies on containment for other classes of queries. Interested readers are referred to [CK86, CLM81, CV92, Shm93] for details.

The complexity of rewriting problem has been the subject of numerous studies. However for this problem, as well as the complexity, the language we consider for maximally contained (MC) rewriting has also played a major role. In [LMS95], this problem was shown to be NP-complete, for standard conjunctive query and views, even when neither the query nor the views have repeated predicates in the rule bodies. The main complexity comes from the fact that an exponential number of rewriting candidates must be considered. In this context, MC rewriting is in the language of union of standard conjunctive queries.

Interestingly, for standard conjunctive query, even if the views are LSI queries, the complexity and the language of rewriting will not change [PH01]. However, [PH01] shows that if the query is also in the form of LSI (or RSI), then MC rewriting can be found in the language of union of LSI (or RSI) queries. [ALM02] shows that when the query is in the form of LSI (or RSI) and even if the views are conjunctive queries with arithmetic comparison constraints, MC rewriting can still be found in the same language.

If we allow inequality ($\neq$) built-in predicates, then the problem becomes NP-hard and a MC rewriting may not exist even if we consider datalog [AD98]. This is due to possibility of using reasoning to determine certain answers (answers that maximally contained rewriting would return) [AD98].

Finally [ALM02] shows that if query is a CQAC and views are all CQACs with all

variables being distinguished, then the complexity of the rewriting problem is exponential and a MC rewriting can be found in the language of union of CQACs. As with containment, the rewriting problem has also been studied in the context of other languages such as datalog and description logics. However these topics fall out of the scope of this thesis.

3.2 Previous Techniques

3.2.1 Bucket Algorithm

The bucket algorithm was introduced as part of a data integration system called Information Manifold [LRO96b]. This system followed the LAV approach for describing information sources and considered OWA regarding the contents of information sources. Details of this algorithm for answering queries over virtual mediated schema was presented in [LRO96a] using source descriptions in the form of conjunctive queries. Bucket algorithm only claims to return MC rewriting for standard conjunctive query and views, which is also the context of this thesis.

The algorithm also considers the case of CQAC queries using source descriptions of the same form, but only guarantees to return contained rewriting for this more general case of conjunctive queries.

Description of the Algorithm

The bucket algorithm is essentially comprised of two steps. For each ordinary subgoal in the body of the query, the algorithm in the first step determines which views are *relevant.* Relevancy of a view for standard conjunctive query reduces to determine whether the variables of the view to which the distinguished variables of the query are mapped are

distinguished or not. For instance, if $q(\overline{X})$ is a subgoal in query Q, then a view V is relevant for Q in regards to subgoal $q(\overline{X})$, if V includes $q(\overline{Y})$ in its body and that if the i^{th} argument X_i in Q is distinguished then so is Y_i in V. In the context of CQAC, this also means consistency of the built-in predicates of the view and query as well. The outcome of this step of the algorithm is a set of relevant views for each subgoal of the query, hence the name bucket.

After creating the buckets for the subgoals in Q, in the second step the algorithm creates a set of candidate rewriting queries by examining combinations of views in the buckets. Each view in a combination comes from one particular bucket, covering one subgoal in the query. Every element of the set of combinations (i.e., each being a rewriting query) is tested for *soundness*, meaning containment in the query. During this process, the algorithm also tries to establish containment by adding extra constraints to the query, which in the standard case amounts to equating distinguished variables.

In [LRO96a], there is also of a post-processing step to remove redundant predicates in the queries after generation of the rewriting queries. The following example illustrates the bucket algorithm.

***Example* 3:** Consider the following query and views:

$$Q : q(X, W) :- \quad p(X, Y), r(Y, Z), s(Z, W).$$

$$V_1 : v_1(A, B) :- \quad p(A, B).$$

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

$$V_3 : v_3(A, B, C) :- \quad r(A, B), s(B, C).$$

$$V_4 : v_4(A) :- \quad s(A, B).$$

In the first step, the algorithm creates the buckets for the subgoals of the query Q,

and for each bucket it searches to find the relevant views. Result of this step is shown in Table 3.

$p(X,Y)$	$r(Y,Z)$	$s(Z,W)$
$v_1(X,Y)$	$v_2(C_1)$	$v_2(W)$
	$v_3(Y,Z,C_2)$	$v_3(A_1,Z,W)$

Table 3.1: Contents of buckets in our example.

Since variable B is not distinguished, V_4 is not relevant for covering predicate p. Note that the corresponding bucket includes the head of the view V that covers the predicate and variables of V are those variables of the query that are in the domain of mapping. The indexed variables replace the remaining ones in the view definitions.

After creating the buckets, in the second step the algorithm considers combining buckets to generate candidates for rewriting. Possible combinations in this case are as follows:

1. $q'(X,W) :- \quad v_1(X,Y), v_2(C_1), v_2(W).$

2. $q'(X,W) :- \quad v_1(X,Y), v_2(C_1), v_3(A_1,Z,W).$

3. $q'(X,W) :- \quad v_1(X,Y), v_3(Y,Z,C_2), v_2(W).$

4. $q'(X,W) :- \quad v_1(X,Y), v_3(Y,Z,C_2), v_3(A_1,Z,W).$

Candidate 1 is not a contained rewriting nor it can be turned into one. The problem here is the join between the predicates p and r in the query with respect to variable Y. Since A is not a distinguished variable in V_2, we can not enforce the join by equating the two. The bucket algorithm does not detect this before the second step.

Candidate 2 is not a contained rewriting, but by equating A_1 and Y, it can be made

into one. Note that by doing so, V_2 becomes redundant in the rewriting. Similarly, by equating C_2 and W, candidate 3 becomes a contained rewriting. However this rewriting is equivalent to the second one and therefore redundant. Finally, candidate 4 is also a contained rewriting and is equivalent to candidates 2 and 3. □

Strengths and Limitations

The algorithm uses buckets to record relevancy of the views in answering a given query. Advantage of such an approach is three fold; Using the information in the query, the algorithm focuses on views that may prove useful in producing rewriting for the query. As a result, the algorithm limits its search to a subset of views. Secondly, by dividing views into appropriate buckets, the algorithm captures information on subgoal coverage and the partial mappings for each view. This further improves efficiency during the second step of the algorithm. Finally, by checking relevancy of the views, the bucket algorithm verifies usefulness of views and therefore reduces the size of each bucket.

There are mainly two sources of inefficiency in the bucket algorithm. First is the search space of the second (combining) step, which is quite large. A naive checking of relevancy in the first step results in large size buckets. Additionally, in the second step the algorithm performs the cartesian product of the buckets to generate rewriting candidates, which can become quite large. The second source of inefficiency is in the second step, in which each rewriting candidate is tested for containment in the query. It is well-known that testing containment is NP-complete. As shown in the above example, if a candidate is not contained in the query, the algorithm examines possibilities of adding constraints to make a candidate into a contained rewriting.

In addition to inefficiencies of the bucket algorithm in generating the rewritings, it is also expensive to evaluate the rewriting queries in general since they are large in the number of subgoals. To improve this, there is a post-processing step to reduce the size of the queries in terms of the number of subgoals, if possible, which comes at an extra cost for time. As discussed in [LRO96a], this optimization is exponential in the size of the query. In addition to large size, the algorithm also generates redundant rewriting queries in some cases, which results in increased query execution time.

3.2.2 Inverse-Rules Algorithm

The inverse-rules algorithm [Dus97] is another rewriting technique proposed in the context of data integration. This algorithm was intended to produce maximally contained rewriting for query and views in datalog.

Description of the Algorithm

The main step of this algorithm is concerned with creating a set of inverse-rules. More specifically, for every predicate in each view, an inverse-rule is created. This rule defines how to calculate tuples of the predicate using instances of the view. That is, given a view definition:

V: $v(\overline{X}) :- \quad p_1(\overline{X_1}), \ldots, p_n(\overline{X_n})$.

for $j = 1, \ldots, n$, a set of inverse rules are created which are of the form:

$p_j(\overline{X'_j}) :- \quad v(\overline{X})$

where $\overline{X'_j}$ is obtained from $\overline{X_j}$ in the following manner: if Y is a constant or distinguished variable in V, then Y appears unchanged in $\overline{X'_j}$. Otherwise Y is replaced by functional

Skolem term $f_i(\overline{X})$ in $\overline{X'_j}$, where i is the ith variable for which we created a function.

After constructing inverse rules for each view, generation of rewritings for a query Q is straightforward. MC rewriting is simply composition of Q and the inverse-rules defined.

***Example* 4: Consider** the query and views of Example 3. The inverse-rules algorithm generates the following rules:

1. $p(A,B) :- \quad v_1(A,B).$
2. $r(f_1(C), f_2(C)) :- \quad v_2(C).$
3. $s(f_2(C), C) :- \quad v_2(C).$
4. $r(A,B) :- \quad v_3(A,B,C).$
5. $s(B,C) :- \quad v_3(A,B,C).$
6. $s(A, f_3(A)) :- \quad v_4(A).$

The above rules show how to extract tuples for the predicates using extensions of the views. For example, rule 3 declares that a tuple (c) in the extension of view v_2 indicates that predicate s contains tuple (U,c), for some value of U.

The rewriting of Q in Example 3 is simply the composition of Q and the above rules. However some rules in this composition are irrelevant to the query and they yield useless tuples. For example, if the extension of view v_4 contained tuple (a), then rule 6 would produce tuple $(a, f_3(a))$ for predicate s. But this tuple would be useless to the query since the value of $f_3(a)$, which is distinguished in Q, is not known. □

Strengths and Limitations

Strength of the inverse-rules algorithm is in its simplicity. First and for most, since inverse-rules are independent of the query and can be created using only the views, the creation of inverse rules is polynomial, which can be performed once, and the rules can be reused for any given query. Secondly, the generation of rewriting from inverse-rules is just a simple composition.

At first glance it seems that this algorithm can generate MC rewriting in polynomial time. However, looking more closely at the output of this algorithm, we can note that evaluation of the generated rewriting is expensive as much work is needed to be done in that stage.

The rewriting generated by the inverse-rules algorithm has two main points of inefficiencies. First, to generate instances for inverse-rules, we have to evaluate view instances multiple times (depending on the number of rules and the size of each rule). During this evaluation process many functional terms are introduced that must be kept track of and evaluation of query is dependant on efficiency of this book-keeping. Secondly, since the algorithm uses all the inverse-rules for every query, many views that are irrelevant to the query are accessed during evaluation process, which results in producing useless tuples.

As pointed out in [PH01], for a fair comparison of this algorithm with others, we need to further process the generated rewriting to make it more evaluation friendly. For this, [PH01] suggests a process during which we expand the rewriting using every possible combination of inverse-rules. Unfortunately, the work involved here is similar to (although more efficient than) the process involved in second phase of the bucket algorithm. The

experiment results in [PH01] show that the extended version of inverse-rules algorithm performs much better than the bucket algorithm. This algorithm, however, is inferior to the Minicon algorithm explained next.

3.2.3 MiniCon Algorithm

The Minicon algorithm proposed in [PH01] is an efficient rewriting algorithm designed mainly for answering standard conjunctive queries using standard conjunctive views. This algorithm was later extended in [PH01] and [ALM02] to handle query and views with attribute constraints of the form LSI. The Minicon algorithm was proposed to address the limitations of bucket and inverse-rules algorithms. As in the bucket algorithm, Minicon starts by considering for each subgoal in the query, which views contain the corresponding subgoal (hence subgoal-based approach). However, the main contributing factor to efficiency of Minicon comes from considering the role of *shared-existential-variables* in the query, which helps eliminate the need for repeated containment testing of candidate rewriting queries. In what follows, we explain this point in more detail.

Description

The Minicon algorithm consists of two phases. In the first phase, for each view, the algorithm forms a set of *MiniCon Description (MCD)* tuples, each representing a partial mapping (Definition 2.4) from query to the view. In the second phase, MCDs are combined to generate the rewriting. Given a query Q and a set V of views, the Minicon algorithm proceeds as follows.

Forming MCDs: The objective of this phase is to produce, for each view, a set of MCD

tuples. Each MCD C for a view V_i is a tuple of the form $(h_C, V_i(\overline{Y})_C, \varphi_C, G_C)$, where:

- h_C, called *head homomorphism* on V_i, is basically a mapping applied to the head atom $V_i(\overline{Y})$, which may equate some distinguished variables of view in order to enforce necessary joins and equality constraints in the query. h_C is identity on the existential variables in V_i.
- $V_i(\overline{Y})_C$ is the view definition after applying h_C to it.
- φ_C is a partial mapping from the variables in Q to variables in the view after applying h_C.
- G_c indicates the subgoal coverage of φ_C.

Minicon start by creating MCDs in a subgoal-based fashion, as the bucket algorithm. However, after finding a partial mapping for a view covering the subgoal in question, it changes perspective and begins examining the variables in the query and view. For this examination, the algorithm uses the following property of rewriting queries [PH01].

***Property* 3.1.** *Let C be a MCD for Q with respect to views in V. The following conditions must hold for C to be useful in generating non-redundant rewriting of Q:*

1. *For each distinguished variable X in Q in the domain of φ_C, $\varphi_C(X)$ is a distinguished variable in $h_C(V)$.*
2. *For each existential variable Y in Q in the domain of φ_C, if $\varphi_C(Y)$ is an existential variable in $h_C(V)$, then for every subgoal g of Q that includes Y, it should hold that (1) every variable in g is in the domain of φ_C, and (2) $\varphi_C(g) \in h_C(V)$.*

Condition 1 above is the same as checking view relevancy in the bucket algorithm. Condition 2 states that if a variable X is part of a join that is not enforced in the view, in order for MCD to be useful, X must be mapped to a distinguished variable in the view so that the join can be enforced in the rewriting. It is important to note that for each MCD, Minicon algorithm tries to find the minimal subset of subgoals that satisfy property 3.1. In this sense, we can say that the algorithm proceeds in a *bottom-up* fashion.

Combining MCDs: Second phase of the algorithm focuses on finding combination of MCDs that would generate non-redundant rewriting queries. Taking the bottom-up approach in the first phase pays off for having an efficient second phase. By creating MCDs covering minimal subgoals, Minicon uses the following property to reduce candidates for rewriting queries.

> ***Property* 3.2.** *Let C be a set of MCDs for Q with respect to views in V. Combinations of C that result in non-redundant rewriting queries of Q are the MCDs $C_1, \ldots, C_l$, for which the following two conditions hold:*
>
> 1. $G_{C_1} \cup \ldots \cup G_{C_l}$ *forms exactly the subgoals of* Q.
> 2. $G_{C_i} \cap G_{C_j} = \emptyset$, *for* $i \neq j$.

Condition 1 above captures the requirement that the rewriting generated is contained in the query. However, the *disjoint property* of MCDs stated in condition 2 is what makes Minicon efficient in the second phase. Based on this condition, we only need to consider combinations of MCDs that have disjoint subgoal coverage. Generating rewriting queries from this combination is straightforward, since all required information is captured conveniently in the MCDs.

$V(\overline{Y})$	φ	h_C	G
$v_1(A,B)$	$X \to A, Y \to B$	$A \to A, B \to B$	p
$v_3(A,B,C)$	$Y \to A, Z \to B$	$A \to A, B \to B, C \to C$	r
$v_3(A,B,C)$	$Z \to B, W \to C$	$A \to A, B \to B, C \to C$	s

Table 3.2: MCDs generated by the Minicon algorithm.

***Example* 5:** Consider the query and views in Example 3. Table 5 shows the MCDs created by the Minicon algorithm in the first phase of operation.

Note that view V_2 is not shown in the table. This is because while creating the MCDs, the algorithm detects that v_2 will not be useful in generating rewriting, since the variable A is not distinguished and a shared variable Y is mapped to it. Therefore no MCD is created for this view.

During the second phase, there is only one combination that the algorithm needs to consider and that is to combine all the three. Since the subgoal coverage of MCDs do not intersect and they cover exactly the body of the query, the result is as follows.

$$q'(X,W) :- \quad v_1(X,Y), v_3(Y,Z,C_1), v_3(A_1,Z,W). \qquad \square$$

Strengths and limitations

The strength of the Minicon algorithm is in its efficiency in generating MC rewriting for standard conjunctive queries. Although having a subgoal-based approach, through property 3.1, it also examines, to some extent, the relationship between the variables in the query and views. This examination ensures that combination of MCDs that satisfy property 3.2 will ultimately produce contained rewriting queries without requiring containment testing. Furthermore, by taking a bottom-up approach in forming MCDs and providing a

basis for disjoint condition in property 3.2, the algorithm prunes the search space significantly and avoids useless combinations that result in redundant rewriting queries being generated or those that will not be contained in the query.

The main limitation of Minicon algorithm also stems from its subgoal-based, bottom-up approach in forming MCDs, which results in generating MC rewritings that are expensive to evaluate for having large number of subgoals in the body of the rewriting queries, in general.

Subgoal-based approach to MCD construction in some cases results in generation of redundant MCDs, which may ultimately result in redundant rewriting queries. To avoid this, each newly generated MCD must be compared with other MCDs. This becomes quite expensive as the number of MCDs grows. The following example illustrates this point.

***Example* 6:** Consider the following query Q and view V:

$$Q : q(X, Z) :- \quad p(X, Y), r(Y, Z).$$

$$V : v(A, C) :- \quad p(A, B), r(B, C).$$

Using the procedure *formMCDs(Q, V)* described in [PH01], the algorithm will generate the MCD $C_1 = (\{A \rightarrow A, C \rightarrow C\}, V(A, C), \{X \rightarrow A, Y \rightarrow B, Z \rightarrow C\}, \{p, r\})$ twice, once when examining the predicate p and again during examination of r. To avoid this redundancy, the procedure states that new MCDs are only added. For this, the algorithm must perform comparisons and discard the second copy of C_1. □

If Minicon avoids this expensive MCDs comparisons, then it will produce redundant rewriting queries.

Although, the bottom-up approach of Minicon increases the efficiency of the second phase, it also contributes to generation of rewritings that are expensive to evaluate in

general. Since each MCD is created to only cover minimal subset of subgoals in the query, combining them will result in rewriting queries with large number of subgoals. To improve this, a post-processing for *tightening* (i.e. reducing the size of) rewriting queries is introduced in [PH01]. This procedure is very similar to the post-processing suggested for the bucket-algorithm. Detail of this optimization is as follows.

Postprocessing for Tightening Rewritings: For every query R_i in the rewriting, Minicon tightens the query by removing the redundant subgoals as follows. Suppose R_i includes two atoms A_1 and A_2 of the same view V_j, whose MCDs are C_1 and C_2 and the following conditions hold.

1. Whenever A_1 (respectively A_2) has a variable from Q in position m, then A_2 (respectively A_1) either has the same variable or a variable that does not appear in Q in position m.

2. Ranges of ρ_{C_1} and ρ_{C_2} do not overlap on existential variables of V_j.

In this case, the algorithm removes one of the two atoms by applying to R_i the homomorphism τ that is (1) the identity on the variables of Q and (2) is the most general unfier of A_1 and A_2.

We may conclude that although Minicon performs more efficiently in generating MC rewriting than previous algorithms, this efficiency comes at the expense of the *quality* of rewriting. Furthermore, measures suggested for improving quality of rewritings are expensive and will result in reduced efficiency of the algorithm.

Table 3.2.3 summarizes the strengths and limitations of the three rewriting algorithms described in this chapter. From our analysis and experiments, we believe that rewritings generated by these algorithms, including Minicon, could be further improved in terms of efficiency and quality. This was our goal in this work to investigate and develop a top-down algorithm to the rewriting problem, introduced in chapter 5.

Algorithm	Strengths	Limitations
Bucket	• Uses buckets to identify and combine relevant data	• Ignores the role of variables • Must consider cartesian product of the buckets • Needs to test containment for each candidate • Low quality of rewriting (redundancy and large size queries)
Inverse-rules	• Generates inverse-rules in polynomial time • Rules can be generated independent of the query • Rewriting is a simple composition of the query and inverse-rules	• Instances of rules must be calculated • Low quality of rewritings • Some rules generate irrelevant tuples to the query • Functional terms are introduced that must be kept track of • Expensive query evaluation
Minicon	• Considers the role of variables • Disjoint condition of property 3.2 prunes the search space in the second phase • No containment testing is required	• Due to its bottom-up approach, the algorithm generates rewriting queries with large sizes • Due to subgoal-based approach, it produces redundant rewriting queries • Expensive post-processing needed to reduce size of the rewriting queries

Table 3.3: Summary of strengths and limitations of bottom-up rewriting algorithms.

Chapter 4

A Graph-based Model for Conjunctive Queries

This chapter presents a new graph-based model to represent conjunctive queries [NN08] and also as an aid to study and analyze the problem of finding maximally contained rewriting of a conjunctive query using views. In the next chapter, we use this model to identify conditions that must be satisfied in order to guarantee maximal containment of rewritings.

4.1 Hyper-node Model for Conjunctive Queries (HMCQ)

Qian [Qia96] proposed to use hyper-graphs to represent conjunctive queries. Nodes in this graph represent the predicates in the query and edges indicate joins between the predicates. Qian used this model to develop a rewriting algorithm called *query folding*, but we believe this technique is insufficient to apply to the rewriting problem in our case, since it lacks the expressiveness needed to capture the relationship between the variables in the queries. Another graph-based approach proposed in [LP90] is the *hyper-node model*

that was intended for representing data models, in general. The nodes in this model can themselves be hyper-nodes. We adopt this model in our work and extend it to *Hyper-node Model for Conjunctive Queries (HMCQ).*

A standard conjunctive query in our HMCQ is a super-graph which consists of four graphs, each representing a level of abstraction. The lowest level in this representation captures relationships among the attributes in the query and possibly equality constraints (i.e., joins) among the attributes. Predicates in the query are represented in the second level in the HMCQ model. The two next levels in HMCQ are used to represent the head of the query.

At the lowest level of abstraction, a conventional undirected graph is used, in which each node represent an attribute in the query and multiple occurrences of the same attribute contributes to multiple distinct nodes. The edges represent equality relationships among the attributes, which could be either joins between predicates of the query or equality constraints inside each predicate, shown by repeated attributes in a subgoal. We refer to this graph as the *attributes-graph* of the query.

At the second level, the hyper-node concept is used to represent the predicates in the query. We refer to this graph with hyper-nodes as the *predicates-graph* of the query. Each hyper-node in this graph is a set of nodes from the attributes-graph. The third level is yet another set of hyper-nodes representing the attributes of the head predicate. Each of these hyper-nodes includes a connected component from a subset of nodes in the attributes-graph, representing a single attribute in the head predicate of the query. Finally, as the fourth level of abstraction, we use one hyper-node to represent the head of the query. Naturally, this hyper-node contains all the elements from the third level as its

nodes. We now formally define the above concepts and notions.

Let Q be a standard conjunctive query. In HMCQ, Q can be represented using a super-graph called G_Q, which contains the following four graphs. We use the following conjunctive query Q to illustrate the concepts as we go through the definitions below.

$Q : q(B, E, B) : - \quad p(A, B, A), p(C, B, D), s(E, C).$

Definition 4.1. *(Attributes-graph). The attributes-graph of Q denoted as $G_A = (V_A, E_A)$ is an undirected, node-labeled graph that has the following properties:*

1. *V_A is the set of constants and variables in the body of Q. There is a separate node for each occurrence of an attribute in each predicate regardless of attribute distinctness. This set can be divided into three subsets V_{hc}, V_d, and V_e. Included in V_{hc} are all the nodes representing constants in the head of Q. V_d contains nodes for variables in Q that are distinguished. V_e represents the existential attributes in Q. V_e itself is further divided into two disjoint subsets: V_{ev} includes existential variables and V_{ec} includes the constants.*

2. *Set E_A contains the edges between nodes in V_A representing equality relationships among variables in Q. This set is further divided into two subsets of E_d and E_e, where E_d includes the edges between nodes of V_d, and E_e represents the edges between the nodes in V_{ev}. Nodes in V_{ec} and V_{hc} in the graph will not be part of any edges.*

3. *Node labels are used to represent orders at different levels of super-graph G_Q. They are also used to differentiate between types of nodes. Labeling is performed in such a manner that the node representing attribute 'A' in the j^{th} position in predicate p_i would be labeled (i, j, k, val), where $i \geq 0$ is called the* predicate-index, *which is the unique identifier assigned to predicate p_i in predicates-graph of Q, k is a set called the* head-index *representing the positions of 'A' in the head of Q. The value of k is 0 for elements in V_e. Finally, val is the* value-index, *representing the value of the constant nodes in V_{ec}. Naturally val is empty for the variable nodes.*

4. *Since equality is an equivalence relation, for each node v in V, the edge (v, v) is in E_A. For ease of representation, we omit these edges from our representations. Additionally, G_A is transitive.*

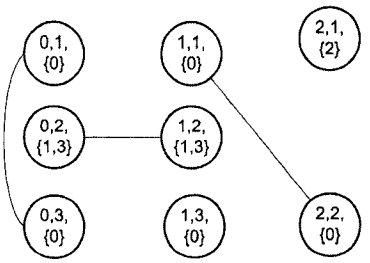

Figure 4.1: Attributes-graph of Q

Figure 4.1 shows the attributes-graph for our query example Q. Note that since all nodes are variables, the value-index (*val*) is omitted from the labels for ease of representation.

Definition 4.2. *(Predicates-graph). The predicates-graph G_P of Q denoted as $G_P = (V_P, E_P)$ is an undirected, node-labeled, edgeless graph with the following properties:*

1. *V_P is a set of hyper-nodes, each of which represents a predicate p_i in the body of Q and each containing a set of nodes from the graph G_A. The nodes in the hyper-node representing p_i are arguments of p_i in Q. As mentioned earlier, positions of attributes in p_i are captured through the labels of the nodes.*

2. *For the hyper-node representing predicate p_i in Q, we use $(i, name)$ as the label, where $i \geq 0$ is a unique identifier assigned to predicate p_i indicating the i^{th} subgoal in the body, and name is the predicate name.*

3. *E_P is empty.*

Figure 4.2 shows the predicates-graph of Q presented earlier.

Definition 4.3. *(Head-variables-graph). This graph of Q, denoted as $G_{HV} = (V_{HV}, E_{HV})$, is an undirected, node-labeled, edgeless graph with the following properties:*

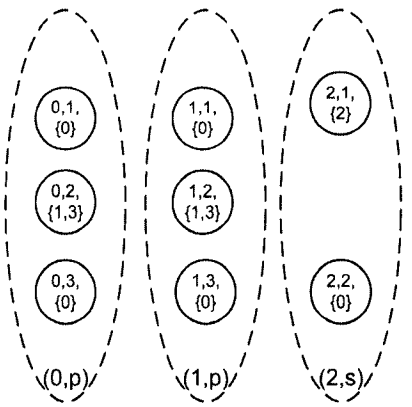

Figure 4.2: Predicates-graph of Q

1. *V_{HV} is a set of hyper-nodes, each representing an attribute of the head predicate, which is either a distinguished variable or a constant. Therefore V_{HV} contains either a connected component from V_d or an element in set V_{hc} of graph G_A.*

2. *As mentioned earlier, head-index in the label of the V_A nodes determines the position of a component or a constant, and hence the V_{HV} nodes in the head. Depending on the size of this index, a connected component from (V_d) can represent several nodes in V_{HV}. Each hyper-node in head-variables-graph is labeled with this index to present the position of variable (which is represented by the hyper-node) in the head.*

3. *E_{HV} is empty.*

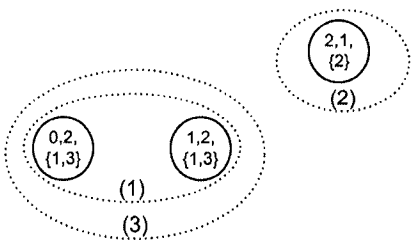

Figure 4.3: Head-variables-graph of Q

Figure 4.3 illustrates the hyper-nodes and their components in the head-variables-graph of query Q.

Definition 4.4. *(Head-graph). The head-graph of Q, denoted as $G_H = (V_H, E_H)$, is an undirected, edgeless graph that has the following properties:*

1. *V_H contains a hyper-node representing the head predicate of the conjunctive query. This hyper-node itself contains all the hyper-nodes of graph G_{HV}.*
2. *E_H is empty.*

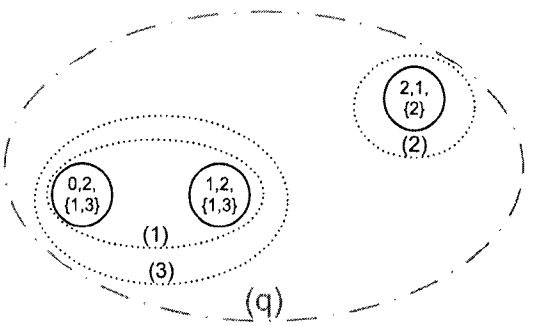

Figure 4.4: Head-graph of Q

Figure 4.4 illustrates the head hyper-node of Q and Figure 4.5 is the complete representation of query Q in the HMCQ model. From the above definitions, we observe the following:

- In V_A, there is no edge between the elements in V_d and V_e.
- The sets V_{hc}, V_d, and V_e are pair-wise disjoint and $V_{hc} \cup V_d \cup V_e = V_A$.
- The sets E_d and E_e are disjoint and $E_d \cup E_e = E_A$.
- Since the essential information of G_H and G_{HV} are captured in labeling of the attribute-nodes, for the sake of clarity, we sometimes omit them from our representations in HMCQ.

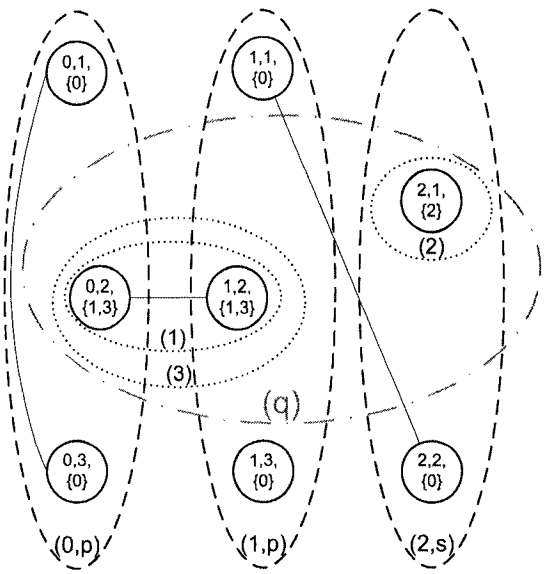

Figure 4.5: HMCQ representation of Q

Table 4.1 summarizes the notation used to represent a conjunctive query Q in the HMCQ. The following example illustrates the above concepts and definitions and also shows the differences between our HMCQ representation and the hyper-graph model presented in [Qia96].

Level	Notation	Description
One	$G_A = (V_A, E_A)$	G_A : the attributes graph of the query. V_A : a set of *nodes* representing the arguments in the body of the query. E_A : a set of *edges* representing the equality relationship between attributes.
Two	$G_P = (V_P, E_P)$	G_P : the predicates graph of the query. V_P : a set of *hyper-nodes* representing the subgoals in the body of the query. $E_A = \emptyset$
Three	$G_{HV} = (V_{HV}, E_{HV})$	G_{HV} : the head-variables graph of the query. V_{HV} : a set of *hyper-nodes* representing the arguments in the head of the query. $E_{HV} = \emptyset$
Four	$G_H = (V_H, E_H)$	G_H : the head graph of the query. V_H : a set containing one *hyper-node* representing the head of the query. $E_H = \emptyset$

Table 4.1: Summary of notations in HMCQ for a query Q.

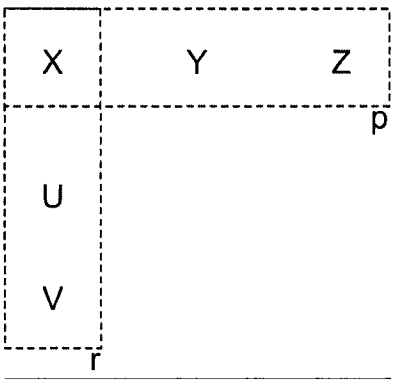

Figure 4.6: Hyper-graph of query Q1 as defined in [Qia96]

***Example* 7:** Consider the following conjunctive queries:

$$Q1(X,Y,Z) :- \quad p(X,Y,Z), r(X,U,V).$$

$$Q2(X,Y,Z) :- \quad p(X,Y,X), r(X,Y,Z).$$

Figure 4.6 shows the hyper-graph representation of $Q1$ as described in [Qia96]. A limitation of this model is that it is not possible to represent the equality constraint in predicate p in query $Q2$.

In the HMCQ model, $Q1$ and $Q2$ can be represented as shown in Figure 4.7. In the graph of $Q1$, hyper-nodes of the head-variables graph and head-graph are shown, however in the graph of Q2, these are omitted for sake of clarity. If an attribute node does not belong to more than one head variable in the head-variables-graph of the query, we can use an integer index to represent its position in the head in its label instead of a set of integers. This is followed in the graph of $Q2$. □

As can be seen from Example 7, there are several differences between the hyper-graph representation and HMCQ. In the former, the edges can only represent joins in the query, however in E_A of G_A, attribute equality constraints within a predicate are also captured.

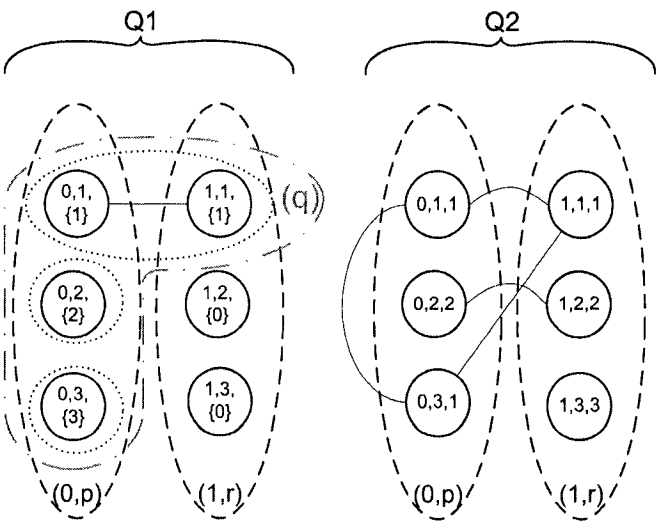

Figure 4.7: Graphs of queries $Q1$ and $Q2$ for Example 7

In addition, the number of nodes in the former representation equals to the number of distinct attributes in the query. However in G_A, nodes include non-distinct attributes as well. Also in HMCQ, the position of attributes in the head are also recorded through the head-index and G_H. This is important when it comes to considering containment mapping and the rewriting problem.

We now need to establish a correspondence between the concepts presented in chapter 2 and the HMCQ model.

4.2 Containment and Rewriting in HMCQ

Through HMCQ, we can provide a graph-based formalization useful in our context of query rewriting. In chapter 2, it was mentioned that the rewriting problem in the standard case and in the context of data integration consists of steps to generate maximally contained rewriting in the form of union of conjunctive queries, where each query in the union uses

in the body the heads of some views as subgoals. Also, using the unfolding technique, we can generate a query that is equivalent to the rewriting by replacing the view heads in the bodies of the rewriting with their respective rule bodies. To use our graph-based model to study the rewriting problem, we first need to establish correspondence between these concepts and HMCQ.

Definition 4.5. *(Containment Mapping) Given conjunctive queries Q_1 and Q_2, a containment mapping from G_{Q1} to G_{Q2} in HMCQ is a mapping ρ from nodes in $G_A(Q1)$ to nodes in $G_A(Q2)$ such that the following conditions holds:*

1. *Every node n_i in $(V_A)_{Q1}$ is mapped to a node $n_{i'}$ in $(V_A)_{Q2}$, where $PosIndex(n_i) = PosIndex(n_{i'})$. Function $PosIndex(n)$ returns the position-index in the label of node n.*

2. *Every node n_i in $(V_d)_{Q1}$ is mapped to a node $n_{i'}$ in $(V_d)_{Q2}$, where $HIndex(n_i) \subseteq HIndex(n_{i'})$. $HIndex(n)$ returns the head-index in the label of the node n, which is a set of integers. We refer to this as head unification condition.*

3. *Every hyper-node $n_i \in (V_P)_{Q1}$ is mapped to a hyper-node $n_{i'} \in V_P(Q2)$, where $NIndex(n_i) = NIndex(n_{i'})$. Function $NIndex(n)$ returns the* name *element in the label of the node n. We refer to this condition as subgoal name consistency.*

4. *Every mode $n_i \in (V_{ec})_{Q1}$ is mapped to some node $n_{i'} \in (V_{ec})_{Q2}$, where $VIndex(n_i) = VIndex(n_{i'})$. Function $VIndex(n)$ returns the* value *element in the label of the node n. We refer to this condition as constant matching.*

5. *The edges in set $\rho((E_A)_{Q1})$ form a subset of the edges in $(E_A)_{Q2}$. We refer to this condition as containment mapping consistency. Note the abuse of notation: we use $\rho(E_A)$ to refer to $< \rho(n_i), \rho(n_j) >$ for every edge $< n_i, n_j > \in E_A$.*

Definition 4.6. *(Graph Containment) Conforming to the definition of containment, the graph G_{Q1} is* contained *in G_{Q2}, denoted as $G_{Q_1} \sqsubseteq G_{Q_2}$, iff there is a containment mapping from G_{Q2} to G_{Q1}.*

The notion of graph containment in HMCQ is explained using the following example.

***Example* 8:** Consider the following queries:

$$Q1(X, Z) :- \quad p(X, Y), r(Y, Z).$$

$$Q2(A, A) :- \quad p(A, B), r(B, A).$$

Figure 4.8 represents the super-graphs of $Q1$ and $Q2$. Consider a mapping ρ from G_{Q1} to G_{Q2}:

$\rho = \{(0, 1, \{1\}) \rightarrow (0, 1, \{1, 2\}), (0, 2, \{0\}) \rightarrow (0, 2, \{0\}), (1, 1, \{0\}) \rightarrow (1, 1, \{0\}), (1, 2, \{2\}) \rightarrow (1, 2, \{1, 2\})\}$.

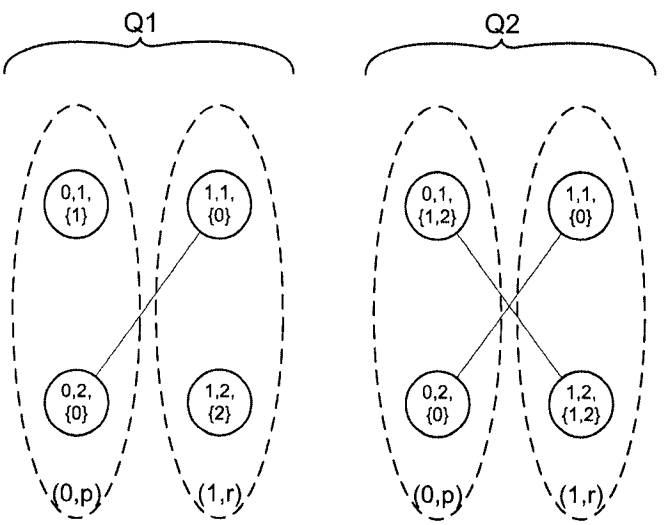

Figure 4.8: Super-graphs of Q1 and Q2

Here, ρ is a containment mapping, explained as follows:

- every distinguished attribute in $Q1$, i.e., each node in $(V_A)_{Q1}$ is mapped to a node in $(V_A)_{Q2}$ with the same position-index.

- since $(0, 1, \{1\}) \rightarrow (0, 1, \{1, 2\})$ and $\{1\} \subseteq \{1, 2\}$, and since $(1, 2, \{2\}) \rightarrow (1, 2, \{1, 2\})$

and $\{2\} \subseteq \{1,2\}$, the head unification condition is satisfied.

- since every hyper-node in $(V_P)_{Q1} = \{(0,p),(1,r)\}$ is mapped to a hyper-node in $(V_P)_{Q2} = \{(0,p),(1,r)\}$ with the same name-index, the subgoal name consistency condition is satisfied.

- The set of edges $\rho((E_A)_{Q1}) = \{< (0,2,0),(1,1,0) >\}$ is a subset of $(E_A)_{Q2} = \{< (0,2,0),(1,1,0) >, < (0,1,1,2),(1,2,1,2) >\}$, and hence the containment mapping consistency is satisfied.

Therefore $Q2 \sqsubseteq Q1$. □

Definition 4.7. *(Rewriting Graph) In HMCQ, a rewriting R of a query Q using view definitions $V = \{v_1, v_2, \ldots, v_n\}$ is a super-graph G_R contained in G_Q, and G_R includes the graphs of V as subgraphs such that:*

1. *The set $(V_P)_R$ of hyper-nodes consist of only the head hyper-nodes from the graphs G_H of view super-graphs in set V.*
2. *Nodes in $(V_A)_R$ are hyper-nodes from V_{HV} in view graphs in set V. Edges in $(E_A)_R$ are equality relationships between these hyper-nodes.*
3. *Labeling of the nodes correspond to the hyper-node indexes of R.*

The above correspondence indicates that a rewriting of query using views is a super-graph whose level-two graph is made of level-four graphs of the views. Also nodes in level-one of rewriting super-graph corresponds to level-three graphs of views. As mentioned in chapter 2, to test containment of rewriting in the standard case, we can use unfolding.

Definition 4.8. *(Unfolding) In HMCQ, a rewriting R with super-graph G_R is unfolded into an equivalent query R' with super-graph $G_{R'}$ by:*

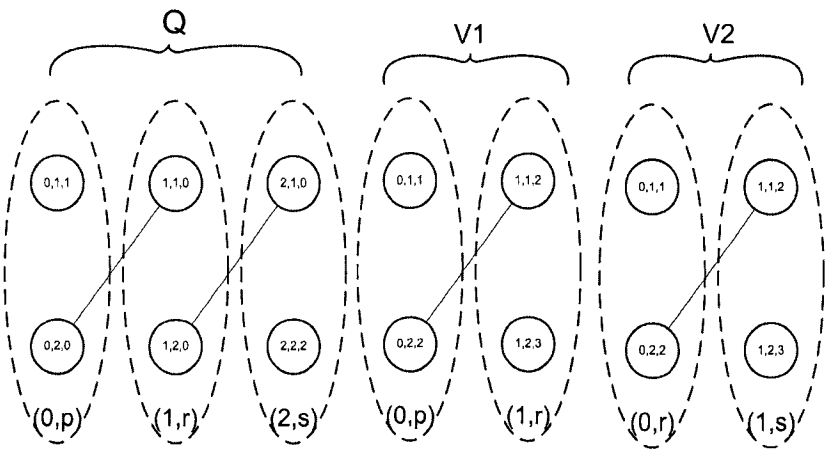

Figure 4.9: The super-graphs of example 9

- *Replacing each v in $(V_A)_R$ with their corresponding components from graphs G_A in view graphs in set V. Additionally, we transfer each edge in $(E_A)_R$ to every node in their respective components. Also we add to $(G_A)_R$, all the nodes from V_e in graphs G_A of set V.*

- *Replacing each hyper-node in $(V_P)_R$ representing head graph of $v_i \in V$, with all the hyper-nodes in $(E_P)_{v_i}$.*

- *New labeling of nodes are performed to reflect the hyper-node indexes of R'.*

To illustrate the above concepts, we consider the following example.

***Example* 9:** Consider the following query Q and views $V1$ and $V2$:

$$Q(X,T) :- \quad p(X,Y), r(Y,Z), s(Z,T).$$

$$V1(A,B,C) :- \quad p(A,B), r(B,C).$$

$$V2(A,B,C) :- \quad r(A,B), s(B,C).$$

Figure 4.9 presents the super-graphs G_Q, G_{V1}, and G_{V2}. Figure 4.10 shows the graph G_R of a rewriting set that includes the following conjunctive queries$R1$ and $R2$:

$$R1(X,T) :- \quad V1(X,Y,Z), V2(A_1,Z,T).$$

$$R2(X,T) :- \quad V1(X,Y,C_1), V2(Y,Z,T).$$

Finally, Figure 4.11 shows the unfolded graphs of these two rewritings using HMCQ.

□

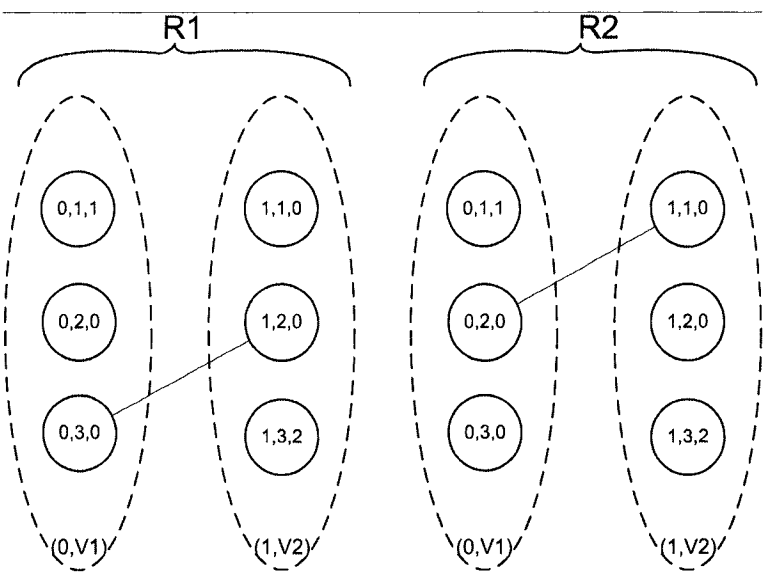

Figure 4.10: The super-graphs of the rewriting in Example 9

For every standard conjunctive query Q, there exists a unique super-graph G_Q in HMCQ representing Q. We use **Graph(Q)** to denote a function which returns this unique graph G_Q of Q. Interestingly, this function is one-to-one (trivial variable renaming may

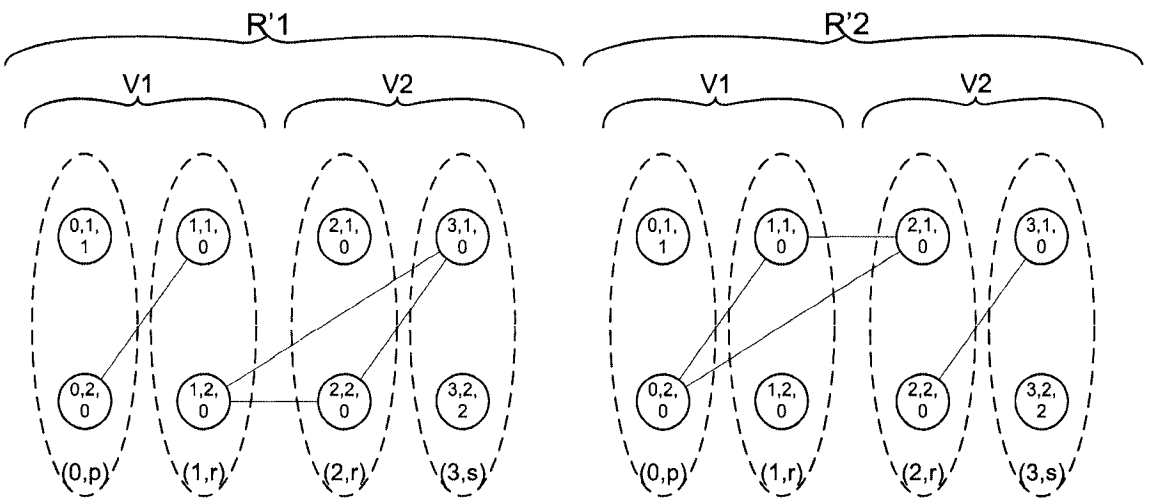

Figure 4.11: Super-graphs of unfolded queries of $R1$ and $R2$ in example 9

be needed) and therefore $\mathbf{Graph^{-1}(G_Q)}$ is a function that returns Q. We use these two functions to alternate between the two forms of the query.

So far we have established correspondence between rewriting concepts and HMCQ. We use this model in the next chapter as an abstraction in our top-down approach to generate maximally contained rewriting for standard conjunctive queries.

Chapter 5

A Top-Down Approach to Rewriting

5.1 Advantages and Challenges of Top-Down Approach

In our study of related work in chapter 3, we mentioned that previous rewriting algorithms took a bottom-up approach to the problem. That is, when rewriting a query using a set of views, those algorithms focus on minimal set of subgoals (optimally a single subgoal) of the query that can be covered using a view. Therefore, in the bucket and Minicon algorithms, to represent each minimal coverage of query subgoals, a copy of the head of a view is added to the body of the rewriting query. In the inverse-rule algorithm, for each subgoal covered by a view, an inverse-rule is added to the rewriting. Alternatively, we propose a top-down approach to rewriting and focus on partial mappings that have maximal coverage of query subgoals using a copy of the head of a view. Our goal in this approach is to improve the quality of rewriting, compared to the bottom-up approach, and do this efficiently. We use Minicon as the best representative of the bottom-up approach

to which we compare our proposed solution. Before we proceed, we define a metric for measuring the quality of rewriting. Referring to [LMS95], we consider the quality of a query to be determined by the cost of its evaluation. Since evaluation of a query depends on the number of predicates in its body, we use the following metric to measure the quality of a rewriting that is in the form of union of conjunctive queries.

Definition 5.1. *(Quality of Rewriting): Let $R = \{Q_1, \ldots, Q_k\}$ be a rewriting for a given query and views, where each Q_i is a standard conjunctive query. Let $|Q_i|$ denote the number of subgoals in the body of Q_i. We define the* area *of R as $A = \sum_{i=1}^{k} |Q_i|$, i.e., the number of predicates in the body of queries in R.*

Quality of rewriting is inversely proportional to the area of rewriting. As can be seen from this definition, for a rewriting R in the form of union of standard conjunctive queries, the area is directly proportional to two factors, width and length defined as follows:

1. the number $|Q_i|$ of predicates in the body of individual rewriting queries, to which we refer as the *width* of Q_i.

2. the number k of queries in the resulting rewriting R, to which we refer to as the *length* of the rewriting.

Using the metric defined above, the subgoal-based, bottom-up techniques such as Minicon generally generate rewriting queries with large widths, which in turn results in larger areas and hence not high quality of rewriting. As mentioned earlier, a post-processing is described in [PH01] to decrease the width of generated rewriting queries, which is costly since this post-processing should be performed on every rewriting and since the number of rewriting queries are usually much larger than the generated MCD

tuples (usually close to cartesian product of MCDs). In addition, as shown in Example 6, due to subgoal-based approach, the Minicon algorithm does not produce the optimal length k of rewriting either, at least not efficiently as costly comparisons are needed to identify redundant MCDs. In our work, we consider a top-down approach to generate high-quality rewritings efficiently, by simultaneously decreasing the width and length of the rewriting. For example, for the query Q and views V in Example 3, our top-down technique generates the following rewriting, which is better than the results produced by Minicon.

$$q'(X, W) :- \quad v_1(X, Y), v_3(Y, Z, W).$$

Now the question that arises is how far the quality can be improved? More specifically, is there an ideal rewriting which has minimal length and width and if yes, what is the complexity of finding it?

It can be shown that in the standard case of conjunctive queries, under Open-World-Assumption (OWA) there exists a unique minimal length for the rewriting. We claim that the view-based approach of top-down always produces the minimal length rewriting. However, as for the width, while a minimal rewriting exists in general [LMS95], it is not unique and in some cases it is very hard to find. That is why the post-processing described in [PH01] does not claim to always find minimal width. Our top-down approach does not guarantee to find the minimal width either, however our goal is to decrease the width as much as efficiently possible. We next describe our top-down approach.

Simply put, for each rewriting in our top-down approach, we try to create partial mappings from largest possible subset of query subgoals to each view. For each of these mappings, a copy of head predicate of the view is added to the body of the rewriting query

generated so that the entire body of the query is covered. This is what we mean by our top-down approach to the rewriting problem. Since we aim for maximal coverage in the top-down approach, these mappings are expected to produce rewriting queries with fewer subgoals and therefore higher quality without post-processing.

Although the top-down approach seems to be more suitable for producing higher-quality rewritings, this advantage comes at the cost of added complexity. Also, due to underlying assumptions of the rewriting problem in the context of data integration (namely OWA), there are additional complexities and issues associated with this approach, and any algorithm that guarantees maximality of rewriting should take measures to deal with such issues. We next briefly outline these issues, and later we describe details of measures taken to deal with these issues.

One of the pitfalls in a top-down approach is at the stage of finding consistent partial mappings for each view. At that point, we have to ensure that maximal coverage of subgoals with a single copy of view will not affect maximality of rewriting. More specifically, there are cases where a consistent partial mapping for maximal coverage can add constraints to the rewriting, as apposed to using several partial mappings of smaller sizes. In such cases, the constraints added may result in loss of maximality of rewriting. The following example illustrates this point.

***Example* 10:** Consider the following query Q and view V:

$$Q(X, Y, U) :- \quad p(X, Y), r(Y, Z, U).$$

$$V(A, B, C) :- \quad p(A, B), r(B, A, C).$$

For convenience, throughout this chapter we abuse the notation and use $Q(X, Y, U)$

to refer to the head of Q. Here, V can cover both subgoals p and r in Q and we can generate a mapping for covering both subgoals with a single occurrence of the head of the view in the rewriting. The resulting rewriting R_1 is as follows:

$$R_1(X,Y,U) :- \quad V(X,Y,U).$$

However R_1 is not maximal. The following rewriting R_2 is a maximally contained rewriting of Q.

$$R_2(X,Y,U) :- \quad V(X,Y,C), V(Z,Y,U).$$

To show this, consider the following instances of predicates p and r, and the tuples defined by R_1 and R_2. Here $R_1 \sqsubseteq R_2$. The tuples defined only by R_2 are shown in bold.

p	r	R_1	R_2
(a,b)	(b,a,c)	(a,b,c)	(a,b,c)
(d,b)	(b,d,e)	(d,b,e)	(d,b,e)
			$\mathbf{(a,b,e)}$
			$\mathbf{(d,b,c)}$

□

Example 10 shows that if there are joins in the view that do not exist in the query, the maximal coverage (R_1) may result in loss of maximality of the rewriting. To capture this, we use HMCQ model described in chapter 4 to identify the necessary conditions that must be satisfied to avoid this problem. Using this formalization, our algorithm correctly determines whether a mapping is required to be broken into smaller mappings.

Another problem that we may encounter in a top-down approach is inherited from the OWA described in chapter 2. In OWA, we assume that data sources are not complete.

This means that views with similar definitions, covering the same subgoals, must also be used in the rewriting to ensure maximality of rewriting. In a top-down approach, this implies that while maximal coverage of each view can produce a rewriting, this rewriting may not be maximal. Therefore, depending on the coverage of other views, it is necessary to sometimes break this coverage into smaller pieces and combine them as needed. The following example illustrates this point.

***Example* 11:** Consider the query Q and views V_1 and V_2:

$$Q(X,Y) :- \quad p(X,Y), r(Y,Z).$$

$$V_1(A,B,C) :- \quad p(A,B), r(B,C).$$

$$V_2(A,B) :- \quad r(A,B).$$

Two possible rewritings of Q are as follows:

$$R_1(X,Y) :- \quad V_1(X,Y,Z).$$

$$R_2(X,Y) :- \quad V_1(X,Y,C_1), V_2(Y,Z).$$

Under OWA, R_1 by itself is not maximal but $R_1 \cup R_2$ is a maximally contained rewriting. □

From this example, it is evident that in a top-down approach, we have to consider the coverage of other views as well, when deciding on subgoal coverage of a view. Later in this chapter, we will describe how our algorithm deals with this issue by using a two-phase approach in determining subgoal coverage of views.

We will next use HMCQ model presented in the previous chapter to study and analyze the rewriting problem. The goal is to determine general conditions that must be satisfied in our top-down approach in order to ensure maximality of rewritings.

5.2 A Top-down Approach to Rewriting using HMCQ

Using the HMCQ model, in this section we present a top-down approach to generate rewriting for standard case. We study the conditions that must be satisfied by HMCQ to ensure maximality of rewriting. These conditions reflect issues inherent to a top-down approach that were briefly pointed out at the beginning of this chapter.

Our top-down approach to rewriting problem includes the following three phases: establishing consistent partial mappings from the query to each view, examining partial mappings to ensure maximality of rewriting, and finally combining the partial mappings properly to generate maximally contained rewriting in the form of union of standard conjunctive queries. For ease of exposition, we assume the query and views do not include constant arguments. These phases are described in details, as follows.

5.2.1 Generating partial mappings

In our top-down approach to generate rewriting for a query and a set of views, in the first phase we examine each view V_i, in isolation, and construct a set of consistent partial mappings from the query to V_i, each of which covering maximal number of subgoals of the query. Intuitively, each partial mapping is a variable mapping from some variables in the query to a subset of variables in some view. Each partial mapping indicates which subgoals in the query are covered by a copy of the view in the rewriting. Later on, we combine these partial mappings to generate a rewriting for the query. This phase may look somewhat similar to the phase in Minicon algorithm, which forms MCDs, each of which is a tuple created to represent each partial mapping. While our goal in this phase is to build consistent partial mappings, each covering as many query subgoals as possible without

loss of maximality of the rewriting. In contrast, Minicon uses each partial mapping to cover as few subgoals as possible. Also, it is not required in Minicon to test for loss of maximality of rewriting, since partial mappings are minimal. Next, we discuss the details of this phase using the graph-based model HMCQ.

Let G_Q be the super-graph representing the query Q, and G_{V_i} be the one representing a view in the set V of views. From Definition 4.5 of containment mapping in HMCQ presented earlier, we know that this mapping is identity on name-index in the labels of hyper-nodes in predicates graphs G_Q and G_{V_i}. In general, a desired partial mapping must satisfy the following four conditions to ensure containment and maximality of a rewriting:

Condition 1: [*Head-Unification*] Since for a rewriting R to be contained in the query, the head of R must be unifiable to the head of Q, all the distinguished variables of the query must be mapped to distinguished variables in the view. We refer to this condition as *head-unification.*

Condition 2: [*Join-Recoverability*] Since a rewriting $R = r_1, \ldots, r_k$ of a query Q is a union of queries defined by the partial mappings of the views, and since each rewriting query r_j in the union must be contained in Q, therefore joins between the variables in the subgoals across domains of individual partial mappings must be enforceable to ensure consistency among the partial mappings. We refer to this condition as *join-recoverability.*

Condition 3: [*Partial-Mapping-Consistency*] Since each partial mapping by itself must be consistent, the joins in the query between the nodes in the domain of partial mapping must either exist in the view or must be enforceable.

Condition 4: [*Partial-Mapping-Maximality*] Since additional edges may exist in a view, consistent partial mappings that are not minimal (in coverage of the subgoals of Q), may not produce maximally contained rewriting (ref. Example 10). To ensure maximality, unless inevitable, equality constraints in the view should not enforce constraints on the rewriting that do not already exist in the original query.

The first three conditions above govern consistency of the mappings and containment of each rewriting query r_j induced by them. That is, by ensuring that these conditions are satisfied, we are assured the rewritings generated will be contained in the query. The last condition ensures maximality of a rewriting induced by these partial mappings. We next describe the correspondence of these conditions for a partial mapping μ_j in the HMCQ model. In the following conditions, D_{μ_j} is a set representing the domain of μ_j.

Head-unification condition in HMCQ

Testing this condition is straightforward. To satisfy this condition in HMCQ for a partial mapping μ_j from G_Q to G_{V_i}, the following must hold:

$$\forall n_k \in (V_d)_Q : \qquad n_k \in D_{\mu_j} \Rightarrow \mu_j(n_k) \in (V_d)_{V_i} \tag{5.1}$$

The above indicates that if a distinguished node in G_Q is mapped to an existential node in G_{V_i}, this condition is violated. This is also consistent with the definition of containment mapping in HMCQ, where all the elements in V_d of containing query (G_Q)

must be mapped only to the elements in set V_d of the containee (e.g. a rewriting query G_r).

Join-recoverability condition in HMCQ

The following must hold for all existential edges $e =< n_k, n_{k'} >$ in $(E_e)_Q$ in order to satisfy this condition:

$$\forall e =< n_k, n_{k'} > \in (E_e)_Q : \quad (n_k \in D_{\mu_j} \wedge n_{k'} \notin D_{\mu_j}) \Rightarrow \mu_j(n_k) \in (V_d)_{V_i} \tag{5.2}$$

This condition focuses on the existential edges in the attributes-graph of G_Q, where one of the nodes, but not both, is in the domain of μ_j. In order to satisfy this condition, the node in the domain of μ_j must be mapped to a distinguished node in G_{V_i}. It is noteworthy that since condition 1 ensures that condition 2 is satisfied for distinguished edges $(E_d)_Q$ of G_Q, there is no need to check this set again.

Partial-mapping-consistency condition in HMCQ

Let H be a subset of all possible edges that can be created between distinguished nodes in attributes-graph of the view V_i, that is $H \subseteq (V_d)_{V_i} \times (V_d)_{V_i}$. To satisfy partial-mapping-consistency condition, the following must hold for μ_j:

$$\exists H : \forall < n_k, n_{k'} > \in (E_A)_Q : \quad n_k, n_{k'} \in D_{\mu_j} \Rightarrow < \mu_j(n_k), \mu_j(n_{k'}) > \in (H \cup (E_A)_{V_i}) \tag{5.3}$$

Here, we are concerned with the edges in G_Q connecting the nodes that both are in the domain of μ_j. To satisfy this condition, all such edges must either exist in G_{V_i} or can be created in the graph of the view. In the conjunctive form, set H corresponds to the head-homomorphism on V_i that equates some variables in the head of the view to make the mapping consistent. In HMCQ, we represent this homomorphism as a set of edges that can be added between distinguished nodes in the attributes-graph of the view. For each edge $e =< n_k, n_{k'} >$ in $(E_A)_Q$ where both nodes n_k and $n_{k'}$ are in the domain of μ_j, there exist three possibilities:

1. Both $\mu_j(n_k)$ and $\mu_(n_{k'})$ are in $(V_e)_{V_i}$, that is they are both mapped to existential nodes in G_{V_i}. Then the edge $< \mu_j(n_k), \mu_j(n_{k'}) >$ must be in $(E_e)_{V_i}$ to satisfy this condition. Otherwise, the condition is violated.

2. Only one of $\mu_j(n_k)$ or $\mu_j(n_{k'})$ is in $(V_e)_{V_i}$, say $\mu_j(n_k)$, and the other one $\mu_j(n_{k'})$ is in $(V_d)_{V_i}$. That is one node is mapped to an existential node and the other to a distinguished node. In this case, we know that edge $< \mu_j(n_k), \mu_j(n_{k'}) >$ will not be in $(E_e)_{V_i}$ nor in $(E_d)_{V_i}$, and therefore the condition is violated.

3. Both $\mu_j(n_k)$ and $\mu_(n_{k'})$ are in $(V_d)_{V_i}$. In this case, if the edge $< \mu_j(n_k), \mu_j(n_{k'}) >$ is not in $(E_d)_{V_i}$, this condition can be satisfied by adding this edge to set H.

In case 1, presence of edge $< \mu_j(n_k), \mu_j(n_{k'}) >$ in $(E_e)_{V_i}$ indicates that there is a dependency between the hyper-nodes p_n and $p_{n'}$ with respect to μ_j, where $n_k \in p_n$ and $n_{k'} \in p_{n'}$. That is, removal of the attributes of any one (but not both) of these predicates from domain of μ_j will cause violation of join-recoverability condition by the attribute nodes of the other predicate.

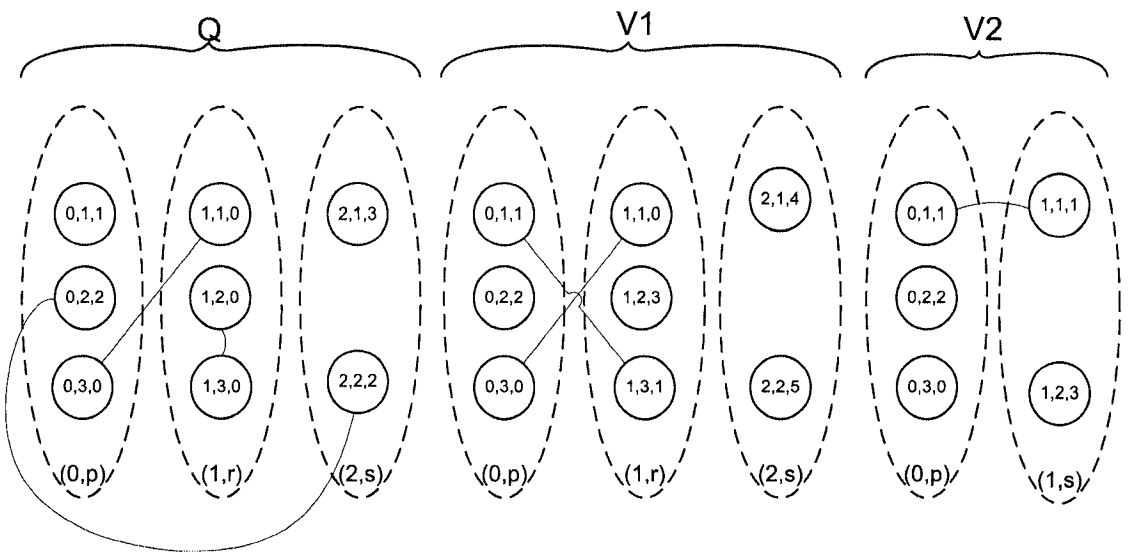

Figure 5.1: Super-graphs of Q, V1, and V2 in Example 12

***Example* 12:** Consider the following query and views:

$$Q(X,Y,W) :- \quad p(X,Y,Z), r(Z,U,U), s(W,Y).$$

$$V1(A,B,D,E,F) :- \quad p(A,B,C), r(C,D,A), s(E,F).$$

$$V2(A,B,D) :- \quad p(A,B,C), s(A,D).$$

Figure 5.1 shows the super-graphs of Q, $V1$, and $V2$. The only maximal mapping from G_Q to G_{V1} is the following:

$\mu_1 = \{(0,1,1) \rightarrow (0,1,1), (0,2,2) \rightarrow (0,2,2), (0,3,0) \rightarrow (0,3,0), (1,1,0) \rightarrow (1,1,0), (1,2,0) \rightarrow (1,2,3), (1,3,0) \rightarrow (1,3,1), (2,1,3) \rightarrow (2,1,4), (2,2,2) \rightarrow (2,2,5)\}$

This mapping covers the hyper-nodes $\{(0,p), (1,r), (2,s)\}$ in G_Q. We now check μ_1 for the above three conditions.

- *head-unification:* The subset of $(V_d)_Q$ in the domain of μ_1 is $\{(0,1,1), (0,2,2), (2,1,3), (2,1,3), (2,2,2)\}$. Since every node in this set is mapped to an element in $(V_d)_{V1}$, none of the mappings violate head-unification condition.

- *join-recoverability:* Since all the nodes in set, $(E_e)_Q = \{< (0,3,0),(1,1,0) >,$ $< (1,2,0),(1,3,0) >\}$ are in the domain of μ_1, this condition is satisfied.

- *partial-mapping-consistency:* For the set $(E_A)_Q = \{< (0,3,0),(1,1,0) >,$ $< (1,2,0),(1,3,0) >, < (0,2,2),(2,2,2) >\}$, we have the following:

  - according to μ_1, the source and destination nodes of edge $< (0,3,0),(1,1,0) >$ are mapped to $(0,3,0)$ and $(1,1,0)$, respectively. Since both of these nodes belong to set $(V_e)_{V1}$, the edge $< (0,3,0),(1,1,0) >$ must be in $(E_e)_{V1}$, which is indeed true. Additionally, this edge creates a dependency, in the context of μ_1, between the hyper-nodes $(0,p)$ and $(1,r)$ of G_Q for μ_1.

  - according to μ_1, the nodes of edge $e =< (1,2,0),(1,3,0) >$ are mapped to $(1,2,3)$ and $(1,3,1)$ respectively, both of which belong to set $(V_d)_{V1}$. The edge e can thus be added to set H, and therefore condition 3 is not violated.

  - Since edge $< (0,2,2),(2,2,2) >$ belongs to $(E_d)_Q$, according to condition 1, both nodes of this edge will be mapped to distinguished nodes in $(V_d)_{V1}$. These nodes are mapped to $(0,2,2)$ and $(2,2,5)$, respectively. The edge does not exist in $(E_d)_{V1}$, but it can be added and therefore condition 3 is not violated.

At this point, the first three conditions are satisfied in the mapping μ_1. Similarly, for $V2$ we have the partial mapping $\mu_1' = \{0,1,1) \rightarrow (0,1,1),(0,2,2) \rightarrow (0,2,2),$ $(0,3,0) \rightarrow (0,3,0),(2,1,3) \rightarrow (1,1,1),(2,2,2) \rightarrow (1,2,3)\}$, which also satisfies all three conditions mentioned above and covers the set $\{(0,p),(2,s)\}$ from G_Q. □

***Lemma* 5.1.** *Let Q be a query and V be a set of views all in the standard conjunctive form. Let U be the set of all possible partial mappings from the super-graph G_Q to those for views, and G_R be the rewriting induced by subset T of U. If the query $Graph^{-1}(G_R)$ is contained in Q, then every element in T satisfies head-unification, join-recoverability, and partial-mapping-consistency conditions.* □

Proof. (Sketch) Since $Graph^{-1}(G_R)$ is contained in Q, it indicates that there exists a containment mapping ρ from Q to the unfolding of $Graph^{-1}(G_R)$, called R', where ρ maps each variable in Q to a variable in R' such that (1) the subgoals in the body of Q become a subset of subgoals in R', and (2) the heads of Q and R' become identical. If there exists a partial mapping μ_i from G_Q to G_{V_i} in T such that:

- μ_i violates head-unification condition, it implies that there exists a distinguished variable X in Q such that X is mapped to an existential variable in V_i. If X is not a shared-variable in Q, then the above implies that X will not be present in the body of $Graph^{-1}(G_R)$ and therefore query $Graph^{-1}(G_R)$ will not be safe. Our requirement is that every conjunctive query should be safe. If X is a shared-variable in Q, the above implies that ρ is not consistent. Hence, no such μ_i can exist in T.

- μ_i violates join-recoverability condition, it implies that there must exist a variable X in Q that appears in both subgoals p_1 and p_2 and X in p_1 is mapped to an existential variable in V_i. Since in unfolding $Graph^{-1}(G_R)$, existential variables of V_i are replaced with fresh variables, ρ will not be consistent. Hence, μ_i cannot exist in T.

- μ_i violates partial-mapping-consistency condition, then this implies that ρ is not consistent. Therefore, no such μ_i can exist in T. □

So far we described representations of first three conditions that, if satisfied, will ensure containment of a rewriting. Next, we describe details of the forth condition.

Partial-mapping-maximality condition in HMCQ

In order to ensure that a partial mapping μ_j would produce maximally contained

rewriting, we need to closely examine the edges between the attributes of the view in the range of μ_j. This is to ensure that no extra constraints are added to the rewriting unless it is necessary. It was mentioned earlier that the partial-mapping-consistency condition may sometimes create dependencies between predicates of the query that are in the domain of μ_j. As a result, the hyper-nodes in the predicates-graph of query in domain of μ_j are partitioned into a set of connected components, where the predicates in each component can not be removed from the mapping without violating the join-recoverability condition by the remaining predicates of that component. This dependency information is important during checking for partial-mapping-maximality condition. Therefore, we refer to this set of connected components in the domain of μ_j as C_{μ_j}. Before describing this condition, we introduce a few concepts in HMCQ, used in our formalization.

- $Pred(G_Q, n)$ is a function from $(V_A)_Q$ to $(V_P)_Q$, which returns for each node $n \in (V_A)_Q$ the hyper-node $p \in (V_P)_Q$, where $n \in p$.

- For partial mapping μ_j, we use μ_j^{-1} to denote the inverse of μ_j, defined in the usual way. Note that the inverse of a partial mapping need not be a function.

- $Comp(C_{\mu_j}, p)$ is a function which returns the component c in C_{μ_j} to which the predicate hyper-node p belongs.

Now we are ready to describe the partial-mapping-maximality condition. As mentioned above, we focus on the set of edges that a view G_{V_i} enforces on to the attribute nodes of G_Q induced by a mapping μ_j. This set has two sources for its elements. One obvious source is the inverse mapping of a subset of edges in $(E_A)_{V_i}$, which includes all the edges $e =< n, n' >$ in G_{V_i} where both n and n' are in the range of μ_j. The other source

that is less obvious is the set of edges generated by the mapping of repeated predicates, i.e., hyper-nodes with identical name-index value, in G_Q to the same predicate hyper-node in G_{V_i}. As the result, edges are enforced between those attributes of these predicates that are mapped to the same predicate in the view. This set of edges is formalized as follows:

Definition 5.2. *Let S be the set of all edges that are enforced by G_{V_i} through the partial mapping μ_j. This set includes the following elements:*

- *The collection of sets $\mu_j^{-1}(e_k)$, for every edge $e_k =< n, n' >\in (E_A)_{V_i}$ where both n and n' are in the range of μ_j. Note the abuse of notation: we use $\mu_j^{-1}(e_k)$ to refer to all possible edges between sets $\mu_j^{-1}(n)$ and $\mu_j^{-1}(n')$ for $e_k =< n, n' >$.*
- *For set P of predicate hyper-nodes in G_Q that are mapped to the same predicate hyper-node $p_{k'}$ of G_{V_i}, edges created between every pair of nodes from set P that have the same position-index in their labels.*

Using S and also considering the set of components C_{μ_j}, partial-mapping-maximality is satisfied if the following condition holds:

$$\forall < n_k, n_{k'} >\in S :$$

$$Comp(C_{\mu_j}, Pred(G_Q, n_k)) \neq Comp(C_{\mu_j}, Pred(G_Q, n_{k'})) \quad \Rightarrow \quad < n_k, n_{k'} >\in (E_A)_Q \tag{5.4}$$

The above indicates that for each edge $< n_k, n_{k'} >$ in S, where predicates for the

nodes of the edges do not belong to the same components in C_{μ_j}, if $< n_k, n_{k'} >$ is not in $(E_A)_Q$, then this condition is violated.

***Example* 13:** Consider the partial mappings μ_1 and μ_1' for the query and views in Example 12. We now can test these two mapping for partial-mapping-maximality.

For $\mu_1 = \{(0,1,1) \rightarrow (0,1,1), (0,2,2) \rightarrow (0,2,2), (0,3,0) \rightarrow (0,3,0), (1,1,0) \rightarrow (1,1,0), (1,2,0) \rightarrow (1,2,3), (1,3,0) \rightarrow (1,3,1), (2,1,3) \rightarrow (2,1,4), (2,2,2) \rightarrow (2,2,5)\}$, we have $S_{\mu_1} = \{< (0,1,1), (1,3,0) >, < (0,3,0), (1,1,0) >\}$ and $C_{\mu_1} = \{\{(0,p), (1,r)\}, \{(2,s)\}\}$. It can be seen, $< (0,1,1), (1,3,0) >$ is the only edge in S that is not present in $(E_A)_Q$. However, since $Comp(C_{\mu_1}, (0,p)) = Comp(C_{\mu_1}, (1,r))$, condition 4 is not violated by μ_1.

Now let us consider the mapping $\mu_1' = \{(0,1,1) \rightarrow (0,1,1), (0,2,2) \rightarrow (0,2,2), (0,3,0) \rightarrow (0,3,0), (2,1,3) \rightarrow (1,1,1), (2,2,2) \rightarrow (1,2,3)\}$ for $V2$. We have: $S_{\mu_1'} = \{< (0,1,1), (2,1,3) >\}$ and $C_{\mu_2} = \{\{(0,p)\}, \{(2,s)\}\}$. Here, the edge $< (0,1,1), (2,1,3) >$ is not in $(E_A)_Q$. Since $Comp(C_{\mu_2}, (0,p)) \neq Comp(C_{\mu_2}, (2,s))$, condition 4 is violated. □

***Lemma* 5.2.** *Let Q and v be a query and a view in standard conjunctive form. If G_R is a contained rewriting of G_Q generated from set T of partial mappings with disjoint subgoal coverage from G_Q to G_v, where each element in T satisfies partial-mapping-maximality condition, then $Graph^{-1}(G_R)$ is maximally contained in Q.* □

Proof. Since the rewriting G_R is contained in G_Q and there is only one view v, if $Graph^{-1}(G_R)$ is not maximally contained in Q, it indicates that partial mappings in T enforce new equality constraints on unfolding of $Graph^{-1}(G_R)$, called R', which do not already exist in Q. Since elements in T have disjoint subgoal coverage, equality constraints added to subgoals outside the domain of each partial mapping will not affect the

maximality of containment of R' in Q. Only equality constraints within the domain of each partial mapping can result in loss of maximality. However, partial-mapping-maximality ensures that these equality constraints either already exist in Q or they are between subgoals that are dependent to each other due to condition 3. □

It is important to note that violation of this condition is necessary but not sufficient condition for loss of maximality. However finding condition that is both necessary and sufficient is harder in general.

This concludes our analysis of the phase of generating partial mappings. Due to other influential factors such as OWA, in the next phase of our top-down approach, we examine partial mappings from a different perspective to ensure maximality of rewritings.

5.2.2 Partial Mappings and The Impact of OWA

So far we used HMCQ to study generation of consistent partial mappings for each view in set V of views and verified the conditions described earlier. Also Lemmas 5.1 and 5.2 indicate that after verification of each partial mapping, combination of these partial mappings will produce maximally contained rewriting, as far as each individual view is concerned. Our next goal is to show how to combine these mappings to generate a desired rewriting. Compared to a bottom-up approach, there are two issues in our top-down approach which contribute to additional complexity in this combining phase of partial mappings of different views. To illustrate these issues, we consider Minicon again as a representative of the bottom-up algorithms.

First noticeable difference in complexity between the Minicon approach and our top-down approach is related to disjoint condition in property 3.2, described earlier. This

property significantly reduces the search space of Minicon during the second phase (for combining MCDs). According to this property, only disjoint MCDs (based on subgoal coverage) can be combined together to generate non-redundant rewriting queries. This implies that many combinations of MCDs will be avoided due to having intersections with each other. This property stems from the fact that partial mappings in MCDs of Minicon have minimal subgoal coverage. Since in our top-down approach the partial mappings are maximal, this property no longer holds, since it is possible to break partial mappings with maximal subgoals into smaller ones in order to create disjoint mappings.

Secondly, as mentioned at the beginning of this chapter, one of the challenges of top-down approach results from Open-World-Assumption, which implies that partial mappings from different views which cover the same subgoals must also be broken down before combining them together to guarantee maximally contained rewriting.

To address the above two problems, we consider an additional phase in our top-down approach, in which we analyze and compare the partial mappings. In this phase, we do not need to compare partial mappings of the same view. This is because during the mapping generation phase, partial mappings of each individual view are generated in such a way that combining them together would ensure maximality of rewriting and only disjoint partial mappings can produce non-redundant rewriting queries. However, if the mappings are from different views, when their subgoal coverage are not disjoint, we must separate disjoint portions of the mappings from which we create new mappings. This condition is formalized next, for which we need to introduce some terms as follows.

- $View(\mu_j)$ is a function that returns the view for which the mapping μ_j is defined.

- $Subgoals(\mu_j)$ denotes the maximal set of predicate hyper-nodes of the query in the domain of μ_j. Similarly, for a set M of partial mappings, $Subgoals(M)$ returns the union of maximal predicate hyper-nodes in the domain of each partial mapping μ_k in M.

- $Comps(\mu_j)$ is the set of components for subgoals of the query in the domain of μ_j. As mentioned earlier, these components represent dependencies between the subgoals with respect to μ_j.

- Subset relationship ($\mu_1 \subseteq \mu_2$) between two partial mappings μ_1 and μ_2 (from the same view) is defined as μ_1 being an extension of μ_2 and with $Subgoals(\mu_1) \subseteq Subgoals(\mu_2)$.

- Equality relationship ($\mu_1 = \mu_2$) between two partial mappings μ_1 and μ_2 means that $\mu_1 \subseteq \mu_2$ and $\mu_2 \subseteq \mu_1$.

Complete-mapping-maximality condition in HMCQ

We can now present the condition that must be satisfied in this phase, in order to guarantee maximality of rewriting. We refer to this condition as *complete-mapping-maximality.* Let M be the set of partial mappings generated for a query Q and a set of views V. Complete-mapping-maximality condition is satisfied, if the following holds.

$$\forall \mu_i \in M: \quad (\exists \mu_j \in M : View(\mu_i) \neq View(\mu_j) \wedge Subgoals(\mu_i) \cap Subgoals(\mu_j) = S)$$

$$\Rightarrow \exists M' \subseteq M : (\forall \mu'_k \in M' : \mu'_k \subseteq \mu_i \wedge |Comps(\mu'_k)| = 1) \wedge Subgoals(M') = S \tag{5.5}$$

The above condition states that for every partial mapping μ_i in M generated during the first phase, if there exists some other partial mapping μ_j in M from a different view where the two mappings share subgoal coverage, then there should exist a subset M' of the partial mappings in M such that each mapping μ'_k in M' is a subset of μ_i and the subgoal coverage of μ'_k is a component in the intersection of μ_i and μ_j. Simply put, for every two partial mappings from different views, their overlapping subgoal coverage must be created into minimal components.

***Example* 14:** Suppose set M contains the mappings μ_1, μ'_2 and μ'_3 for the query and views in Example 13. Recall that for μ_1, $\{\{(0,p),(1,r)\},\{(2,s)\}\}$ is the coverage of the mapping in form of components. As was shown in the example, μ_2 violates condition 4, and hence we replace it with μ'_2 and μ'_3 having the coverage $\{\{(0,p)\}\}$ and $\{\{(2,s)\}\}$, respectively.

We next compare each mapping with the others to check whether the above condition holds. Hence:

- μ_1: This mapping overlaps with μ'_2 over $(0,p)$. Therefore, an extension of μ_1 must exist in M having coverage equal to component to which $(0,p)$ belongs. However, no such mapping exists in M and therefore this condition is violated. To rectify this,

we can add a new mapping μ_{11} to M with coverage $\{\{(0,p),(1,r)\}\}$. By comparing the mappings μ'_3 and μ_1, we note that the two share the predicate $(2,s)$, and since no such extension of μ_1 exists in M, again this condition is violated. To rectify this, we need to add μ_{12} with coverage $\{(2,s)\}$ to M.

- μ'_2: No comparison is needed between this mapping and μ'_3, since they are both from the same view. Furthermore, since coverage of μ'_2 has only one component, it is not possible to generate a new extension of this mapping and therefore the above condition is not violated.

- μ'_3: It is not necessary to compare this mapping with μ'_2, since they are both from the same view. Furthermore μ'_3 and μ_{11} are disjoint. Again, since coverage of μ'_3 only has one component, the above condition is not violated.

The result of comparing the mappings is the set $M = \{\mu_1, \mu_{11}, \mu_{12}, \mu'_2, \mu'_3\}$ where the subgoal coverage of μ_1 is $\{\{(0,p),(1,r)\},\{(2,s)\}\}$, μ_{11} is $\{\{(0,p),(1,r)\}\}$, μ_{12} is $\{\{(2,s)\}\}$, μ'_2 is $\{\{(0,p)\}\}$, and the subgoal coverage of μ'_3 is $\{\{(2,s)\}\}\}$. □

5.2.3 Rewriting Generation in HMCQ

The final phase in our top-down approach to rewriting focuses on generation of rewriting R using the partial mappings in M. The main task in our rewriting generation phase is to efficiently combine partial mappings in M and generate a set of complete mappings, each of which resulting in a conjunctive query covering the entire body of Q. Union of these conjunctive queries form a maximally contained rewriting of Q using V. In this phase, the emphasis is on efficiency in combining the partial mappings and producing non-redundant

rewriting queries. Benefiting from efforts of previous phases, we can now use the following two properties to reduce the search space and speed up this phase:

***Property* 5.1.** *When combining partial mapping from M to generate rewriting for G_Q using a set graphs of views in V, for every combination C covering all the subgoals of G_Q that results in a non-redundant rewriting, the following two conditions must hold:*

1. $\forall \mu_i \& \mu_j \in C : Subgoals(\mu_i) \cap Subgoals(\mu_j) = \emptyset$
2. $\nexists \mu' \in M : (\exists \mu_1 \ldots \mu_k \in C : (\mu_1 \cup \mu_2 \cup \ldots \cup \mu_k = \mu'))$

Using these two properties, we can eliminate many useless combinations of partial mappings in the rewriting phase. The first condition in Property 5.1 is the same as Property 3.2 in Minicon [PH01]. The second condition in Property 5.1 is unique to our top-down approach and can be used to further reduce the search space in the rewriting generation phase. The following example illustrates these points.

***Example* 15:** Let set M be a set of mappings generated for query Q and views V_1 and V_2 in Example 12. To generate a rewriting R for Q, we examine combinations of those elements in M which satisfy both conditions of the above property as follows. Since $M = \{\mu_1, \mu_{11}, \mu_{12}, \mu'_2, \mu'_3\}$, where $Subgoals(\mu_1) = \{\{(0,p),(1,r)\},\{(2,s)\}\}$, $Subgoals(\mu_{11}) = \{\{(0,p),(1,r)\}\}$, $Subgoals(\mu_{12}) = \{\{(2,s)\}\}$, $Subgoals(\mu'_2) = \{\{(0,p)\}\}$, and $Subgoals(\mu'_3) = \{\{(2,s)\}\}$, the only combinations that satisfy both conditions are $\{\mu_1\}$ and $\{\mu_{11}, \mu'_3\}$. The two graphs of the rewriting queries for these combinations are shown in Figure 5.2. Alternatively, these queries are as follows:

$$R1(X,Y,W) :- \quad V1(X,Y,X,W,Y).$$

$$R2(X,Y,W) :- \quad V1(X,Y,X,E_1,F_1), V2(W,B_1,Y).$$

It is noteworthy that for this example, the Minicon algorithm produces the following rewriting R'.

$$R'1(X,Y,W) :- \quad V1(X,Y,X,E_1,F_1), V1(A_1,B_1,D_1,W,Y).$$

$$R'2(X,Y,W) :- \quad V1(X,Y,X,E_2,F_2), V2(W,B_2,Y).$$

As shown above, the area of rewriting R' generated by the bottom-up approach of Minicon is greater than that of rewriting R produced by our top-down approach. This increase is due to Minicon's focus in generating partial mappings with minimal subgoal coverage. □

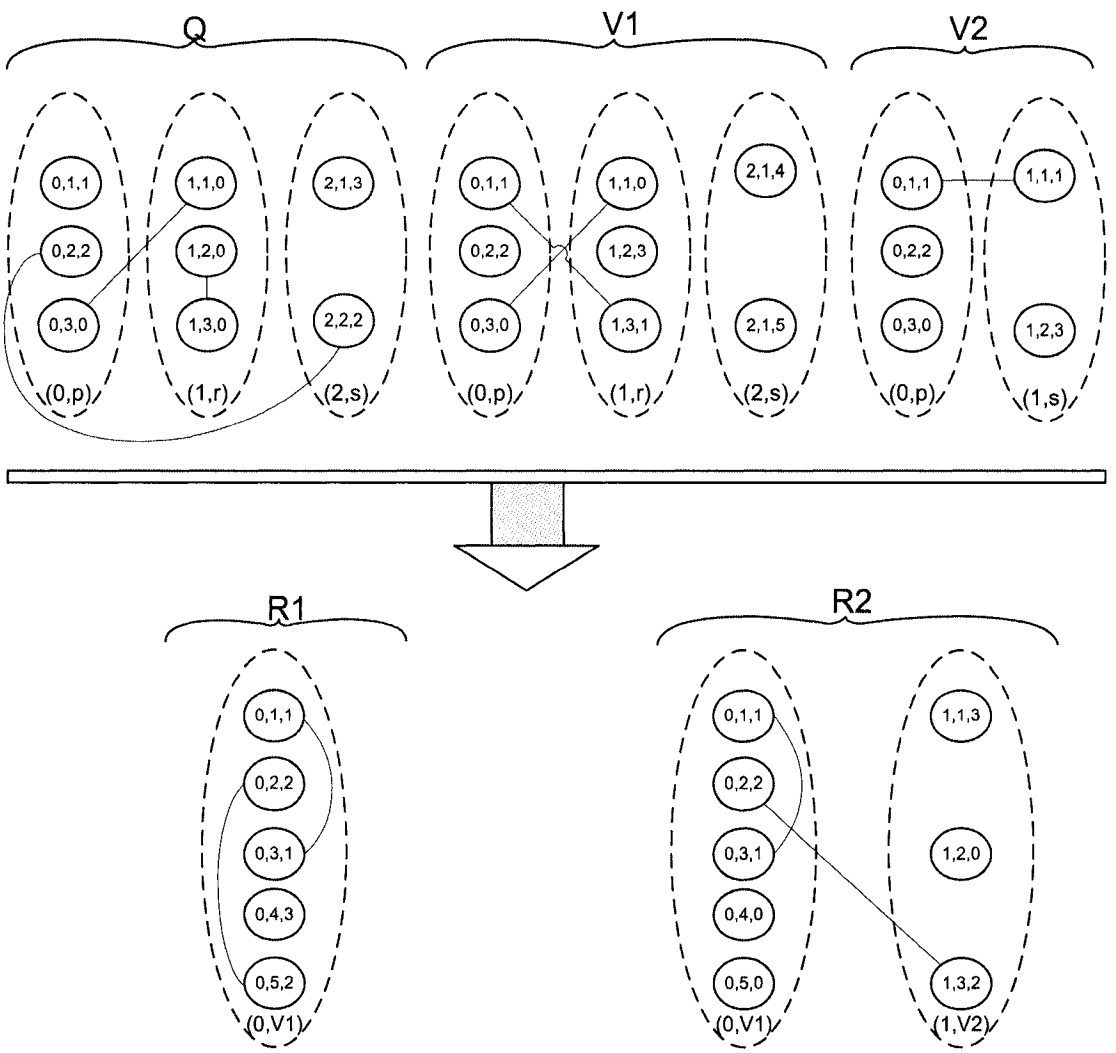

Figure 5.2: Graphs of rewriting $R = \{R1, R2\}$ of G_Q using the G_{V1} and G_{V2}

This concludes our study of the proposed top-down approach using HMCQ in generating maximally contained rewriting for standard conjunctive query using views. In the next section, we introduce a top-down rewriting algorithm, called TreeWise, which implements the proposed ideas and approach.

5.3 TreeWise Algorithm

After using HMCQ to provide an abstraction of our top-down approach to rewriting, we can now present details of our rewriting algorithm, called TreeWise. Naturally, this algorithm implements our top-down approach to generate maximally contained rewriting for a standard conjunctive query using a set of standard conjunctive views. By using the conditions and properties captured in HMCQ and by carefully choosing proper data structures and procedures, TreeWise is set to efficiently generate better quality rewritings without any post-processing.

TreeWise algorithm operates in three phases: mapping tuple construction, binary-tree construction, and rewriting generation phase. The first phase corresponds to the generation of partial mappings presented in the abstraction. In binary tree construction phase, we address the issues involved in partial mapping comparison discussed in our analysis. Finally, the rewriting generation phase includes steps for combining partial mappings and generating rewritings. Technical details of the three phases are provided below.

In order to determine the conditions outlined in the abstraction efficiently, the very first step in TreeWise algorithm includes construction of super-graphs G_Q and G_V of the query Q and set V of views, which will be used in all phases of the algorithm. We do not

present the details of generation of these graphs from the HMCQ description.

5.3.1 Mapping Tuple Construction Phase

In section 5.2.1, we described the four necessary conditions to ensure usability of partial mappings in generating maximally contained rewriting. We now use this knowledge to generate a set of consistent partial mappings each of which satisfying these conditions.

In this phase of the TreeWise Algorithm, for each view V_i in V, a set of mapping tuples T_{V_i} is created. Each element $t =< \mu_t, \rho_t, ((G_H)_{V_i})_t, ((G_P)_Q)_t, C_t, Subs_t >$ in T_{V_i} is defined as follows.

- μ_t is a partial mapping from a subset of attribute-nodes of G_Q to the nodes of G_{V_i}.
- ρ_t is the conjunctive equivalent of the partial mapping μ_t. That is, ρ_t is a partial mapping from Q to V_i.
- $((G_H)_{V_i})_t$ is a copy of the head-graph of view in HMCQ representing μ_t. To this graph, some edges may be added to make the mapping consistent.
- $((G_P)_Q)_t$ is the copy of the predicates-graph of the query that has edges added during the mapping construction phase to reflect subgoal dependencies in the domain of μ_t (refer to partial-mapping-consistency condition).
- C_t is the set of connected components in $((G_P)_Q)_t$.
- $Subs_t$ is the set of subgoal hyper-nodes in predicates-graph of the query covered by μ_t. That is, $Subs_t = Subgoals(\mu_t)$.

Adding edges to $((G_H)_{v_i})_t$ to make μ_t consistent is equivalent to head-homomorphism in MCDs of Minicon (ref. Chapter 3). In presentation of our algorithm, we use the terms μ^{-1}, $Pred(G_Q, n)$, and $Comp(C_t, p)$ introduced in the previous section.

Description of phase one: For creating the set of mapping-tuples for view V_i, we must first find all possible mappings from Q to V_i, each of which covering as many of query subgoals as possible. Each partial mapping maps the variables of subgoals in Q to subgoals in V_i with the same names. Hence, the predicate hyper-nodes in G_{V_i} form a set of equivalence classes, each of which representing targets for a predicate hyper-node in G_Q . Intuitively, the cartesian product of the elements in these classes define all possible mappings from G_Q to G_{V_i}, each of which covering maximal number of predicate hyper-nodes in G_Q. Next, for each partial mapping μ_t, we create a mapping tuple and examine it for the four conditions described in section 5.2.1.

Head-unification condition: Conforming to condition 5.1, we check the set V_d of nodes in attributes-graph of the query in the domain of μ_t. For each node n_k in $(V_d)_Q$ that is mapped to an existential node in $(V_e)_{V_i}$, we remove from the domain of μ_t, attributes of hyper-node $p_m \in (V_p)_Q$, where $n_k \in p_m$.

Join-recoverability condition: To satisfy condition 5.2, for each existential edge $< n_k, n_{k'} >$ in the attributes-graph of the query, where only n_k is in the domain of μ_t, we verify if n_k is mapped to a distinguished node in the view. Otherwise, we have to remove from domain of μ_t, attributes of hyper-node p_m in $(E_P)_Q$, where $n_k \in p_m$.

Partial-mapping-consistency condition: For condition 5.3 to hold, we have to examine each edge $< n_k, n_{k'} >$ in attributes-graph of the query, where n_k and $n_{k'}$ are both in the domain of μ_t, and find a set H of distinguished edges to add to the view in order to make the

mapping consistent. For the three possible cases mentioned in the description of condition 5.3, TreeWise proceeds as follows:

1. If n_k and $n_{k'}$ are both mapped to existential nodes in the view, then edge $< \mu_t(n_k), \mu_t(n_{k'}) >$ must exist in $(E_A)_{V_i}$. If not, then we remove from the domain of μ_t, nodes of hyper-nodes p_m and $p_{m'}$ in $(E_P)_Q$, where n_k and $n_{k'}$ are in $p_{m'}$ and $p_{m'}$, respectively. However, if the mapping exists in the view, then subgoal dependency exists between p_m and $p_{m'}$ and this dependency is captured by TreeWise in the form of an edge $< p_m, p_{m'} >$ in the copy $((G_P)_Q)_t$ of the predicates-graph of the query for μ_t. After addition of a new edge to $(((G_P)_Q)_t)$, this graph has to be replaced by its transitive closure graph, since dependency relation entails transitivity.

2. If n_k is mapped to an existential node and $n_{k'}$ to a distinguished node, then we have to remove the nodes of p_m from domain of μ_t, where $n_k \in p_m$.

3. If n_k and $n_{k'}$ are mapped to distinguished nodes in V_i, even if the mapping of the edge does not exist in the view, we can add it to the rewriting. Hence, in the TreeWise algorithm, if $\mu_t(n_k)$ and $\mu_t(n_{k'})$ belong to different hyper-nodes in head-graph $((G_H)_{V_i})_t$ of the view and the edge does not already exist, we add it.

After checking every edge and taking the appropriate action, tuple t will now have the set of connected components C_t where the subgoals of each component is dependent on one another.

Partial-mapping-maximality condition: As mentioned, violations of the first three conditions result in removal of subgoals from the domain of the mapping, i.e., restricting the

mapping. This is because violation of these conditions indicates that V_i is not suitable for covering some subgoals of the query. However, violation of partial-mapping-maximality in condition 5.4, does not imply that the view is not suitable for covering certain subgoals of the query. Instead, it indicates that the current mapping as a whole may not generate maximally contained rewriting. Therefore, in case of violations of this condition, we should replace the mapping by its smaller constituencies, i.e, breaking the mapping into smaller pieces.

After testing for partial-mapping consistency, $((G_P)_Q)_t$ is divided into set C_t of connected components that will determine options for breaking the partial mapping μ_t when there is a violation of condition 5.4. The focus of our attention is now on set S presented in Definition 5.2. Additionally, to decide the breaking strategy, the algorithm keeps track of violations of condition 5.4 in the form of conflicts between components in predicates-graph of the mapping tuples in the following manner. For every edge e_i in S with nodes belonging to different components of C_t, we test whether e_i is present in the graph of Q. If not, a conflict tuple between predicate hyper-nodes containing nodes of e_i is created. After testing this for all elements in S and creating necessary conflict pairs, the algorithm then decides on how to break μ_t into minimal number of conflict-free pieces. Using dependency in $((G_P)_Q)_t$ and the set containing conflict pairs, TreeWise breaks μ_t into smaller mappings in such a way that each new mapping will have maximum number of components without any conflict between its predicate hyper-nodes.

At the end of this phase, TreeWise has generated a set of partial mapping tuples that satisfy the first four conditions required. Figure 5.3 shows the steps of the first phase of the TreeWise algorithm for construction of mapping tuples.

procedure **constructMappingTuples**(Q, V, G_Q, G_V)
Inputs: /* Q and V are conjunctive queries in standard form */
/*G_Q and G_V are the graphs of Q and V in the HMCQ model*/
Output: T is a set of consistent mapping tuples.

$T = \emptyset$.
for each view v in V
$T_v = \emptyset$.
Form set E_v of Equivalent Classes of v for subgoals in Q.
for each element e in the cartesian product of the elements in E_v,
where e covers set g of subgoals in Q

Let h be the least restrictive head-homomorphism
on v such that there exists a mapping ρ_t and
its corresponding μ_t in HMCQ such that $\rho_t(Subs_t) = h(e')$, where
$Subs_t \subseteq g$ and $e' \subseteq e$, and μ_t satisfies conditions 5.1–5.3.
if h and ρ_t exist, then:
form tuple $t = (\mu_t, \rho_t, v, (G_H)_v, ((G_P)_Q)_t, C_t, Subs_t)$, where:
a) $(G_H)_v$ is the head-graph of v with
with minimal set of edges to represent h.
b) $((G_P)_Q)_t$ is the predicates-graph of Q with
minimal set of edges to represent dependency from condition 5.3.
c) C_t are the connected components in $((G_P)_Q)_t$
end if
$Conflicts = \emptyset$.
Form set S_t described in Definition 5.2 for tuple t.
for each edge $< n_1, n_2 >$ in S_t where
$Comp(C_t, Pred(G_Q, n_1)) \neq Comp(C_t, Pred(G_Q, n_2))$:
if $< n_1, n_2 >$ is not in G_Q, then:
add pair $(Pred(n_1), Pred(n_2))$ to $Conflicts$.
end if
end for
Break t into minimal set of mapping tuples T_t such that for each element
t' in T_t, $Subs_{t'}$ does not contain two elements p_1 and
p_2 where (p_1, p_2) is in $Conflicts$.
$T_v := T_v \cup T_t$.
end for
$T := T_v \cup T$.
end for
Return T

Figure 5.3: First phase of the TreeWise algorithm

An Optimization for Mapping Tuple Construction Phase: In order to efficiently avoid generation of redundant mapping tuples in phase two and therefore ensure minimal length of generated rewriting, we add the following optimization for phase one of the TreeWise algorithm.

As mentioned above, each partial mapping maps the variables of subgoals in Q to subgoals in V_i with the same names. Hence, the predicate hyper-nodes in G_{V_i} form a set of equivalence classes, each of which representing targets for a predicate hyper-node in G_Q . In our optimized implementation of phase one, instead of using cartesian product of these classes to generate the partial mappings, we use the following strategy.

First we check every element in each of the equivalency classes for head-unification condition and whenever this condition is violated, we remove the target predicate hyper-node from the class. After pruning the equivalence classes, we next divide the set of equivalent classes into two subsets S_1 and S_2. S_1 will include all those classes that only have one target element (i.e., are of size one) and S_2 will include the rest.

For all elements in set S_1, we create one mapping tuple t_{max} and test it for join-recoverability condition exactly as described above. Whenever a node violates this condition, we remove its hyper-node from the tuple, but we add the equivalence class it belonged to subset S_2. The reason for proceeding in such manner is that join-recoverability condition may be satisfied inside set S_2 using this hyper-node. Next, we check t_{max} for partial-mapping-consistency condition as described above. Whenever two nodes violate this condition, we remove their hyper-node (or hyper-nodes) from t_{max}. This time however, we do not add its equivalence class to S_2. This is due to the fact that partial-mapping-consistency can not be satisfied, even if we add predicate hyper-nodes from S_2. Finally, we

use the same conflict detecting strategy described above for checking for partial-mapping-maximality and, if necessary, breaking t_{max} into minimal number of mapping tuples.

Now that we are finished with equivalence classes of set S_1, we focus on set S_2 and use a bottom-up approach to generate all mapping tuples. That is, for elements in classes of set S_2, we create a set of mapping tuples, each of which having minimal number of targets such that each tuple satisfies both join-recoverability and partial-mapping-consistency conditions. It should be noted that since tuples generated from set S_2 have minimal coverage, partial-mapping-maximality need not to be checked.

By doing the above, we should all the mapping tuples that satisfy head-unification, join-recoverability, partial-mapping-consistency and partial-mapping-maximality conditions.

5.3.2 Binary Tree Construction Phase

After generating the set T of partial mapping tuples that satisfy the first four conditions in the abstraction, the TreeWise algorithm checks for complete-mapping maximality condition. As stated earlier, the goal of this phase is two-fold. One is to address the problem of OWA and the other is to create grounds for property 5.1. Satisfaction of complete-mapping-maximality condition ensures achieving both goals. It is important to note that, similar to partial-mapping-maximality condition, violation of complete-mapping-maximality condition implies that smaller pieces of partial mapping must be added to set T. However, as condition 5.5 implies, unlike condition 5.4, violation of complete-mapping-maximality condition does not mean that the partial mapping in question is not maximal. It merely indicates that combination of this mapping with partial mappings of other views

may not yield maximally contained rewriting. Therefore, in case of violation, the partial mapping does not have to be removed from set T. Instead, new smaller mappings that are all extensions of the original mapping must be created and added to T. We refer to this process as *cloning* a partial mapping tuple into *children* mapping tuples.

To facilitate testing of condition 5.5 and also to keep track of parent-child relationships between the mapping tuples, the TreeWise algorithm uses *binary trees* (hence the name TreeWise) in the following manner.

For each partial mapping tuple generated in the previous phase, TreeWise creates a binary tree with this tuple as its root. Therefore set T of mapping-tuples becomes a set T' of binary trees. Next, the root of each binary tree t'_i in T' is compared with the leaf-tuples of all other trees in T' that are from different view (than the root). For each leaf-tuple m_{j_k} of t'_j that has intersection on subgoal coverage with the root of t'_i, one intersection and one difference tuple are generated and added as children of m_{j_k} in t'_j. Dependencies presented in predicates-graph ($(G_P)_Q$) of m_{j_k} adds, to the intersection child, all the predicates in the components of the intersection.

There are several advantages of this strategy of phase two. First of all, since only leaf-tuples participate in the comparison, no redundant mapping tuples are created. The second benefit is that we can easily keep track of parent-child relationship and therefore we can efficiently take advantage of condition 2 in Property 5.1. Thirdly, since we use binary tree and intersection-difference strategy in creating the leaf-nodes, based on condition 1 of Property 5.1 we can efficiently traverse the trees during the rewriting generation phase. This considerably reduces the search space of this phase.

The main drawback of this implementation is that the outcome of this phase depends

on the order in which the trees are compared with each other. This is due to the facts that tree structures are binary and we only clone the leaf-nodes. Therefore, different orders of comparison can split the trees in different ways and ultimately effect the quality of generated rewriting. Hence, employing a good tuple analysis and determining a good order for comparison may further improve the quality of the rewriting at additional cost.

After comparing roots of every tree with leaves of all other trees that are from different views, condition complete-mapping-maximality may not still be satisfied. We now need to clone the leaf-nodes that have intersection with at least one other tuple into children nodes having only one component based on set C_t of each tuple. With the TreeWise algorithm, this task is straightforward due to using binary trees. Again, the predicates-graphs of the query in the mapping tuples are used to clone nodes into minimal components.

This concludes the binary tree construction phase of our algorithm. At this point, set T' includes all the information needed for generating maximally contained rewriting for the query, which will be the focus of the last phase below.

procedure **binaryTreeConstruction**(T)
Inputs: /* T is the set of partial mapping tuples
created in the first phase of the algorithm */
Output: T' is a set of binary trees with mapping-tuples as nodes.
$T' = \emptyset$.
for each tuple t in T:
Form a binary-tree t' with t as the root and Add it to T'.
end for
for each root r of tree t' in T':
for each tree t'' in T' such that there exists a leaf-node m
in t'' such that $Subs_m$ belongs to more than one component in $((G_P)_Q)_m$:
for each leaf-node m in t'':
if $Subs_m \cap Subs_r \neq \emptyset$ then:
a) form new tuple m_1 such that $Subs_{m_1}$ contains all the
predicates hyper-nodes in the components from C_m that
have a predicate included in $Subs_m \cap Subs_r$.
b) form new tuple m_2 such that $Subs_{m_2} = Subs_m - Subs_{m_1}$.
Set m_1 as the left child and m_2 as the right child of m.
end if
end for
end for
end for
for each tree t' in T' such that there exists a
leaf-node m in t' such that $Subs_m$
belongs to more than one component in C_m:
for each leaf-node m in t' such that $Subs_m$
belongs to more than one component in C_m and $Subs_m$
has intersection with at least one root of a tree in T' :
a) form new child tuple m_1 of m such that $|C_{m_1}| = 1$.
a) form new child tuple m_2 of m such that $Subs_{m_2} = Subs_m - Subs_{m_1}$.
set m_1 as the left child and m_2 as the right child of m.
end for
end for
Return T'

Figure 5.4: Second phase of the TreeWise algorithm

5.3.3 Rewriting Generation Phase

In the last phase of TreeWise algorithm, rewriting queries are generated from partial mapping tuples nodes of the trees in set T'. Using Property 5.1, TreeWise algorithm considerably reduces the search space of this phase.

In this phase, an array of bins is created, each representing a predicate in the query. Copies of all trees from set T' that their root-tuples have coverage for query-predicate of the bin are placed inside the bin. Next, trees in the bins are traversed to find the combinations of tuples covering the entire body of the query. Since each tree has only disjoint subtrees and also because of condition 2 in Property 5.1, traversals are always limited to a small part of each tree. Furthermore, any combination that has tuples that are not disjoint will be ignored. For each valid combination c of tuples satisfying Property 5.1, a rewriting query is generated.

For each tuple t of view v in a combination c, generating rewriting involves creating a subgoal $v(\overline{Y'})$ representing t in the body of the rewriting. For generating subgoal $v(\overline{Y'})$ from view definition $v(\overline{Y})$, we use the partial mapping ρ_n^{-1} to *unmap* distinguished variables $(\overline{Y})$ of the view to variables of the query. There are two issues involved in this task; First of all, for any variable y in $\overline{Y}$ that is not in the range of the partial mapping ρ_n, we must create fresh copy of y and add it to $v(\overline{Y'})$. Secondly, since in ρ_n a set of variables of the query may be mapped to a single variable in $\overline{Y}$ of the view, we define a representative of this set arbitrarily, except we use distinguished variables of query whenever possible. Since this choosing of the representative must be performed uniformly across the tuples of combination c, similar to Minicon algorithm [PH01], we use $EC(\overline{Y})$ to refer to this

uniformity across the body of the rewriting. Figure 5.5 describes details of the rewriting generation phase of the TreeWise algorithm.

procedure **rewritingGeneration**(T')
Inputs: /* T' is the set of binary trees generated
during the second phase of the algorithm */
Output: R is a set of rewriting queries in conjunctive form.

$R = \emptyset$.
for each combination $c = \{t_1, \ldots, t_k\}$ of tuples of tree-nodes in set T' where:
a) $Subs(t_1) \cup \ldots \cup Subs(t_k)$ cover exactly the subgoals of Q.
b) for any $i \neq j$, $Subs(t_i) \cap Subs(t_j) = \emptyset$.
c) for any $i \neq j$, $parent(t_i) \neq parent(t_j)$.
for each tuple t_i in the combination c:
Define mapping ϕ_i on the $\overline{Y}_i = Graph^{-1}(Head_{t_i})$ as follows:
if variable $y \in \overline{Y}_i$ is in the range of ρ_{t_i}
$\phi_i = \rho_{t_i}^{-1}(y)$.
else
$\phi_i(y)$ is a fresh copy of y.
end if
end for
Form conjunctive rewriting r:
$r(EC(head(Q))) :- \quad V_{n_1}(EC(\phi_1(\overline{Y}_1))), \ldots, V_{n_k}(EC(\phi_k(\overline{Y}_k)))$.
add r to R.
end for
Return R

Figure 5.5: Third phase of the TreeWise algorithm

Unlike Minicon [PH01], TreeWise does not have a post-processing phase. The increase in quality of rewriting is achieved by considering tuples covering as larger number of subgoals as possible. The efficiency of this approach is due to keeping track of parent-child relationships and pruning the search space accordingly.

5.3.4 Proof of Correctness of the TreeWise Algorithm

Preliminaries

We show that, given a set V of views, for every query $Q: q(\overline{X}) :- p_1(\overline{X_1}), \ldots, p_n(\overline{X_n})$, the rewriting of Q generated by TreeWise is equivalent to that generated by the Minicon Algorithm.

For this rewriting, suppose Minicon generates the set $M = \{m_1, \ldots, m_l\}$ of MCDs and TreeWise generates the set $T = \{t_1, \ldots, t_k\}$ of partial mapping tuples (PMT). Let MR denote the rewriting generated by Minicon and TR denote the rewriting generated by TreeWise. We know that both MR and TR are in the form of a union of standard conjunctive queries. Next, we will show that for every query r_m in MR, there exists a query r_t in TR such that $r_m \equiv r_t$.

***Property* 5.2.** *Suppose r_m is generated using a combination $C = \{m_i, \ldots, m_j\}$ of MCDs from M. Then, the query r_t is generated using $C' = \{t_{i'}, \ldots, t_{j'}\}$ of PMTs from T, where the following statements $D1$ and $D2$ hold.*

> *D1: $\forall m_i \in C \exists t_{i'} \in C'$: (1) $View(m_i) = View(t_{i'})$ and (2) $Subgoals(m_i) \subseteq Subgoals(t_{i'})$*
>
> *D2: $\forall t_i \in C' \exists H \subseteq C$: (1) $\forall m \in H : View(t_i) = View(m)$ and (2) $Subgoals(c_i) = Subgoals(H)$*

Note that we use Subgoals() and View() for MCDs with the same definitions as presented earlier for PMTs. From the combination C', we know that the query r_t exists in the rewriting TR for the following two reasons:

1. Since Conditions 5.1, 5.2 and 5.3 in TreeWise and Property 3.1 of Minicon are satisfied, we know that the union of subgoal coverage of MCDs in M for a view v

is equal to the union of subgoal coverage of PMTs in T for the same view v. Since MCDs have minimal subgoal coverage, this indicates that if there exists a PMT t in T for v, then there exists a corresponding set M' of MCDs in M for v such that $Subgoals(M') = Subgoals(t)$.

2. Satisfaction of Condition 5.5 and also Property 5.1 in TreeWise ensures that for combination C (representing r_m), there exists a corresponding combination C' (representing r_t) from T, where (i) all the PMTs in C' have disjoint subgoal coverage and they cover the entire body of the given Q, and (ii) tuples in C' come from the same set of views as the MCDs in C. The difference is that the subgoal coverage of each tuple in C' for a view v may correspond to the subgoal coverage of a set of tuples in C for the same view v.

For the purpose of our proof, we assume the following normalization step was taken during generation of rewritings in both Minicon and TreeWise algorithms. Note that in rewriting generation phase, both these algorithms use the function $EC(A)$ that returns an arbitrary variable X_i representing the equivalence class E. Here E is the set of all vaiables in query Q that are mapped to the same variable A in the view. We assume that the rewriting generated will include constraint $X_i = X_j$ for each variable X_j in E.

Proof of Soundness

From Property 5.2 and also the correspondence between Properties 3.2 and 5.1 of Minicon and TreeWise related to the rewriting generation phase, we can conclude that the number of queries in MR always equals to that of TR. Note that we assume the Minicon algorithm includes the length optimization, so it does not generate redundant MCDs. This indicates

that TR does not include any queries other than those contained in some query in MR. Therefore, to prove soundness, we prove r_t is contained in r_m. It suffices to show that there exists a containment mapping from r_m to r_t.

Clause $D1$ in Property 5.2 together with satisfaction of condition 5.3 ensure that for every subgoal $v_i(\phi_n(\overline{Y_n}))$ in r_m there exists subgoal $v_i(\phi'_{n'}(\overline{Y'_{n'}}))$ in r_t such that the following holds. For every variable Y appearing at position k in $\phi_n(\overline{Y_n})$ where Y also appears in the query Q, either Y appears at position k in $\phi'_{n'}(\overline{Y'_{n'}})$ or another variable Y' appears at position k in $\phi'_{n'}(\overline{Y'_{n'}})$, where Y' also appears in Q and the constraint $Y = Y'$ holds in r_t. The above indicates that there is a function ρ that maps every subgoal in the body of r_m to a subgoal in the body of r_t. Since every variable X in the head of r_m also appears in the body of Q, X is also mapped to the same variable in r_t, and hence the heads of r_m and r_t unify. This implies ρ is a containment mapping from r_m to r_t. □

Proof of Completeness

We have to prove that r_m is contained in r_t. For this, it suffices to show that there exists a containment mapping from expansion of r_t, called r'_t, to the expansion of r_m, called r'_m.

It follows from clause D2 in Property 5.2 that for every subgoal $v_i(\phi_n(\overline{Y_n}))$ in r_t, there exists a set $H = \{v_i(\phi'_{n'_1}(\overline{Y'_{n'_1}})), \ldots, v_i(\phi'_{n'_k}(\overline{Y'_{n'_k}}))\}$ of subgoals in r_m where the union of subgoal coverage of MCDs represented by elements in H is equal to subgoal coverage of PMTs represented by $v_i(\phi_n(\overline{Y_n}))$.

The above together with satisfaction of condition 5.4 indicates that for every subgoal $p_m(\overline{Z_{m_j}})$ in r'_t there exists a subgoal $p_m(\overline{Z'_{m'_j}})$ in r'_m where the following holds. For every variable Z appearing at position k in $\overline{Z_{m_j}}$ where Z also appears in the query Q, either Z

appears at position k in $\overline{Z'_{m'_j}}$ or another variable Z' appears at position k in $\overline{Z'_{m'_j}}$, where Z' also appears in the query and the constraint $Z = Z'$ exists in r'_m. This means there exists a mapping ρ' that maps every subgoal in the body of r'_t to a subgoal in the body of r'_m. Since every head variable X in r'_t appears in the query Q, X is also mapped to the same variable in r'_m, and hence the heads of r'_t and r'_m unify. This indicates ρ' is a containment mapping from r'_t to r'_m. □

5.3.5 Complexity of the TreeWise Algorithm

It should be noted that the worst-case asymptotic running time of TreeWise is same as Minicon, Inverse-rules, and the Bucket Algorithm. As stated in [PH01], the running time of the basic Minicon algorithm is $O(nmM)^n$, where n is the number of subgoals in the query, m represents maximal number of subgoal in the views, and M equals to the number of views. It should be noted that optimizations mentioned earlier to check for redundant MCDs and also to tighten the resulting rewritings have performance overheads.

Our observation of the TreeWise algorithm indicate that due to the following reasons the running time of this algorithm is almost identical to Minicon.

1. Since views in data integration systems are static, construction of super-graphs for the views can be performed once and used many times. Moreover, we know that the complexity of constructing each graph is polynomial (n^2) in the number of attributes in the view. Therefore, this overhead is not significant for the TreeWise algorithm.

2. In worst case, the mapping tuple construction phase of TreeWise will search a space identical to Minicon. However since TreeWise takes a top-down approach, in most cases will look at significantly fewer number of mapping tuple candidates. Only

task that TreeWise performs and Minicon does not in the first phase of operation is to check for partial-mapping-maximality condition. The complexity of this task is dependent on the number of components in view graph of the mapping tuple and also the number of joins in the view in the range of the mapping. By taking advantage of the efficiency of graph structures, we observe that the overhead of this check is not significant. We also observe that TreeWise generally produces fewer Mapping Tuples compared MCDs generated by Minicon.

3. During the second phase, for each mapping tuple, a binary-tree is created, which is a task with linear complexity. However, the task of comparing trees is more expensive. Since only leaf-nodes of one tree are compared to the root of the other, each comparison is performed in linear time. Also, number of tree comparisons is polynomial (n^2) to the number of mapping tuples that are not minimal.

4. During rewriting generation phase, by using the tree structures and limiting the search only to one portion of each tree and finally by benefiting from property 5.1, TreeWise in most cases performs more efficiently than Minicon in this phase. Additionally, since MCDs cover minimal number of subgoals, the Minicon algorithm must inspect more buckets while combining them to generate rewriting.

This concludes our description of TreeWise algorithm. The next chapter presents the results of our experiments to evaluate performance and quality of rewritings generated by TreeWise, basic Minicon and optimized Minicon as suggested in [PH01]. Before moving on, we conclude this chapter with a brief discussion on the possibility of extending HMCQ beyond standard conjunctive queries.

5.4 Beyond Standard Conjunctive Queries

The HMCQ model presented can properly capture details of the rewriting problem for standard conjunctive queries. However, introduction of general built-in predicates will add new subtleties and complexities which the current HMCQ as introduced cannot handle. More specifically, the current HMCQ is insufficient to capture relationships between the attributes in the built-in predicates. Extending HMCQ to general case of built-in predicates requires further investigation. Here, we will consider special cases of built-in predicates that HMCQ with minor modifications can handle. We focus on two classes of conjunctive queries with built-in predicates that were introduced earlier in chapter 2: open-LSI queries [FP04] and conjunctive queries with arithmetic equality expressions [Ali05].

5.4.1 Open-LSI Queries

From [PH01], we know that presence of LSI comparison in the views will not effect the rewriting problem. However if the query also contains LSI comparison predicates, then the implication test (Definition 2.7) is required for each candidate rewriting [GSUW94]. Since homomorphism-property holds for LSI queries [Klu88], finding a single containment mapping that makes the implication true suffices. Parallel to this result, it has also been shown in [FP04] that when the LSI comparisons are all open, no coupling is possible in the implication and again only a single mapping is enough to make the implication true.

The above results indicate that steps described earlier for the rewriting problem will not be affected by presence of built-in predicates in the query of type open-LSI. In this case, we can use the HMCQ abstraction with addition of the implication test during the

generation of each partial mapping tuple, and in case the implication does not hold, that tuple should be discarded.

5.4.2 Conjunctive Queries with Arithmetic Equality Expressions

In chapter 2, we presented a formal definition of these conjunctive queries and containment for such queries, which again included an implication test [GSUW94]. When built-in predicates in conjunctive query and views are in the form of linear arithmetic equality expressions, the problem of rewriting query using views is affected in subtle ways, making HMCQ unsuitable for use in such cases.

An issue we face when trying to use HMCQ for the arithmetic equality expression case is how to represent equality expressions in the graph model. We can consider built-in predicates as edges between nodes in the attributes-graph of query. This complicates the model as new nodes representing new constants and attributes which do not exist in the predicates in the query must be added to the graph and new types of edges must be introduced for capturing arithmetic operators.

Since the HMCQ model is not suitable for representing built-in predicates, an alternative is to represent the standard portion of the query using HMCQ and, similar to LSI case, test constraint implication separately. A question that comes to mind is: what impact do arithmetic constraints have on the standard graph of the query in HMCQ?

The first impact is related to distinguished attributes of the graph. We know that distinguishability of attributes is an important issue in the rewriting problem and therefore captured in HMCQ. However, in the context of linear arithmetic equality expression, distinguishability is replaced by computability of attributes [Ali05]. Currently it is not

possible to capture in HMCQ computability of attributes stemming from arithmetic expressions.

A second issued faced is that built-in predicates can create new edges between the nodes of attributes-graph of standard portion of the query (i.e., they can enforce joins). These edges can be identified by solving the system of equations defined by arithmetic equality expressions and adding them to the model.

Chapter 6

Experiments and Results

In this chapter we present details of our experiments to evaluate the performance of TreeWise and three versions of the Minicon algorithm which we have implemented. The results are used to analyze and compare strengths and limitations of these algorithms. For this purpose, we performed about 100 execution batches taking around 123.5 hours. The goal of these experiments are twofold. In the first class of experiments, the TreeWise algorithm is compared with existing rewriting algorithms under different circumstances. In [PH01], authors report results of similar experiments for three existing algorithms (bucket, inverse-rules, and Minicon), and since Minicon generally outperformed the other two, we use Minicon as a representative of this group with which we compare our algorithm. The second class of experiments conducted studies the scalability of the TreeWise algorithm for different types of queries when the number of views grows.

Our experiments includes three classes of queries: chain, star, and complete queries, as used in [PH01]. In chain queries, except for the first and last subgoals, each subgoal is joined on one variable with its adjacent subgoals, creating a chain of joins. In star queries, there is a unique subgoal that is joined with all other subgoals in the body of the

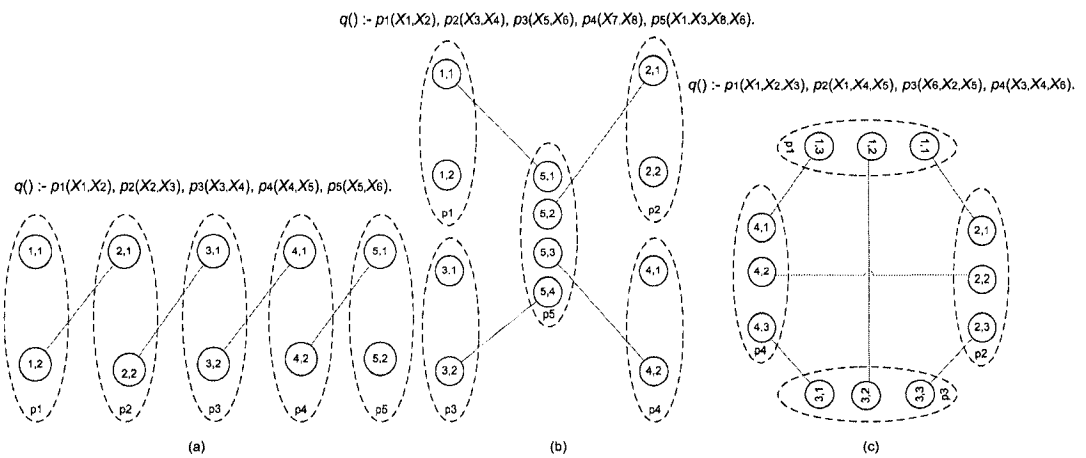

Figure 6.1: Examples of HMCQ representation for: (a) chain query (b) star query (c) complete query.

query, and there is no join between the other subgoals. In complete queries, each subgoal is joined with every other subgoal in the body of the query. Using HMCQ representation, Figure 6.1 illustrates examples of these classes of queries. In addition to these queries, we also tested the algorithms using conjunctive queries, chosen randomly from a pool of such queries.

To make the results of our experiments comparable to Minicon, we used the same random query generator used in [PH01], which was kindly provided to us by Dr. Pottinger. Using this generator, we can control the following parameters: (1) the widths of the queries and views (i.e., number of subgoals in the body), (2) the size of the pool from which query subgoals are chosen, (3) the number of variables in each subgoal, (4) the number of distinguished variables, i.e., the number of arguments in the head, and (5) the number of repeated predicates in the queries and views.

The results are compared using the following parameters: (1)performance, i.e., the elapsed time to generate the rewriting, (2) the length of the rewriting, i.e., number of rules,

(3) the width of the rewriting queries, i.e., number of subgoals in each query, and (4) the area of the rewriting (the product of query width and length introduced in Definition 5.1). For the same set of parameters, results are averaged over multiple runs (between 30 to 100 runs). As for the performance, it is noteworthy that since in the TreeWise algorithm, the graphs of the views (at pre-processing stage) can be generated once and used many times, given that the views are static in data integration systems, we omitted the time to generate these graphs. Also, since these graphs in TreeWise are implemented as regular undirected graphs, this generation time is not long compared to the entire processing time. Finally, we used a Pentium 4, 2793 MHz, 1GB RAM, running Windows XP Professional. Both TreeWise and Minicon algorithms were implemented in Java and run on the Eclipse platform.

6.1 Comparing TreeWise with Minicon

For these experiments, we implemented three versions of the Minicon algorithm. The first version, referred to simply as Minicon, is the implementation of the algorithm without any optimization for reducing lengths (i.e., without checking MCD redundancy) or widths of the rewriting queries (i.e., tightening rewriting queries). For each of the other two versions, we added one of the two optimizations,width or length, just mentioned. We refer to these versions of Minicon with width and length optimizations as *MiniconWM* and *MiniconLM*, respectively. This allows us to estimate the effect and also overhead of each optimization in different circumstances, while comparing TreeWise with Minicon in its optimal performance mode.

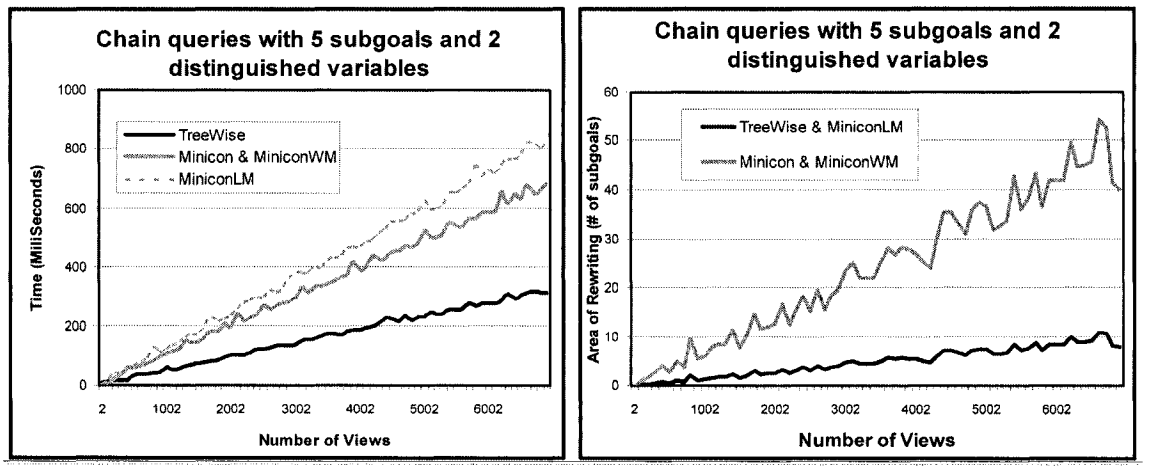

Figure 6.2: Result of experiments for chain queries of size 5 subgoals and 2 distinguished variables.

6.1.1 Chain Queries

To conform to the experiments in [PH01], we consider two cases of chain queries. First, we used queries and views of the same sizes (i.e., widths), where only the first and last variables in the chain are distinguished. In this case, in order to have a rewriting, query and the view must be identical (modulo variable renaming). Hence there are usually very few rewritings. For this case, we found that all three versions of the Minicon algorithm have similar performance. This is due to the fact that since very few MCDs and rewriting queries are generated by the algorithm, the optimization routines of the algorithms in most cases are not reached. But in cases where some views are identical to the query, the overhead of length reduction is noticeable. Also, the other two versions of the algorithm produce redundant queries in the result rewriting. This is due to the subgoal-based approach of Minicon. In contrast, the view-based approach of TreeWise avoids this redundancy.

As shown in Figure 6.2(a), all four algorithms scale linearly in the number of views, but TreeWise generally outperforms all the others by a factor of at least 2. The difference

in the performance is contributed to the subgoal-based nature of Minicon algorithms which generate identical MCD candidates multiple times. By taking the view-based approach, the TreeWise algorithm examines such candidates only once. From Figure 6.3, we notice that as the number of subgoals in the body of the query increases, the performance gap between TreeWise and Minicon algorithm grows. For queries with 5 subgoals, TreeWise outperforms Minicon by a factor of 2, and for queries with 10 subgoals, TreeWise is three time faster. Figure 6.2(b) shows that quality of rewritings generated by TreeWise in this case is identical to MiniconLM but superior to Minicon and MiniconWM.

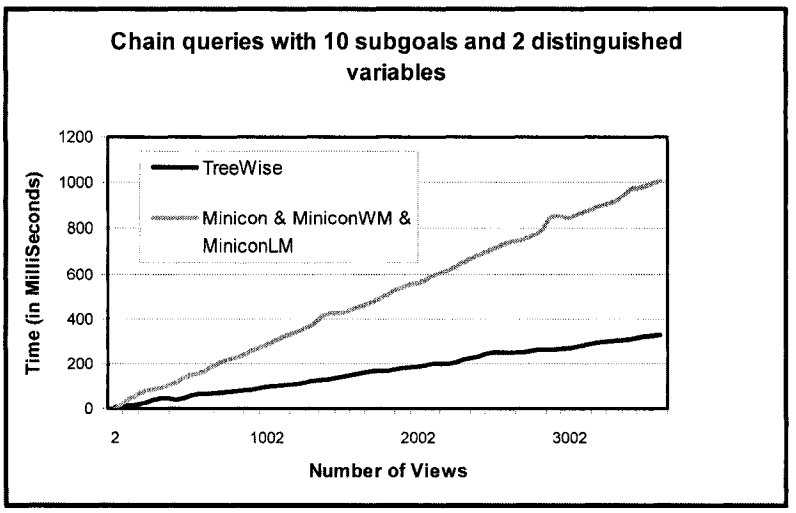

Figure 6.3: Result of experiments for chain queries with 10 subgoals and 2 distinguished variables.

For the second class of chain queries, we consider queries and views of the same widths whose variables are all distinguished. In this case, because almost each view, depending on the size of the subgoals pool, can contribute to generation of a rewriting, the number of queries in each rewriting (i.e., the length of rewriting) is generally quite large. As shown in Figure 6.4(a), while all four algorithms show non-linear behavior in

the number of views, TreeWise and Minicon have similar performance. We also observed that TreeWise performs slightly better than Minicon as the number of views increases. In this case, optimization done by Minicon considerably reduces the widths of the queries, with an overhead that can be seen from the results. Since the number of queries in the rewritings are exponential in the number of subgoals in the views, the performance of MiniconWM grows almost exponentially, as shown in the figure.

On the other hand, Figure 6.4(b) shows that TreeWise produces rewritings of significantly better quality than the basic Minicon. The quality of rewritings of TreeWise is less than MiniconWM, and this depends solely on the order in which the tuples are compared in the second phase of TreeWise and the manner in which the trees are created. Although not shown here, our experiments indicated that as the sizes of query and views increase, the overhead of optimizing the widths of rewritings increases even more rapidly. Also because Minicon uses a subgoal-based approach, as the number of subgoals in the query and views increases, TreeWise with its view-based approach begins to outperform Minicon. It is interesting to note that since all the variables are distinguished, redundancy in MCD generation is generally not much. Also, while the overhead of optimizing length of rewriting is not as high as the previous case, it is still noticeable.

6.1.2 Star Queries

We also consider two cases of star queries in our experiments. In the first case, query and views are of the same size and distinguished variables in the query and views do not participate in the joins. As in the second case of chain queries with two distinguished variables, there are very few rewritings in this case. Since none of the variables in the

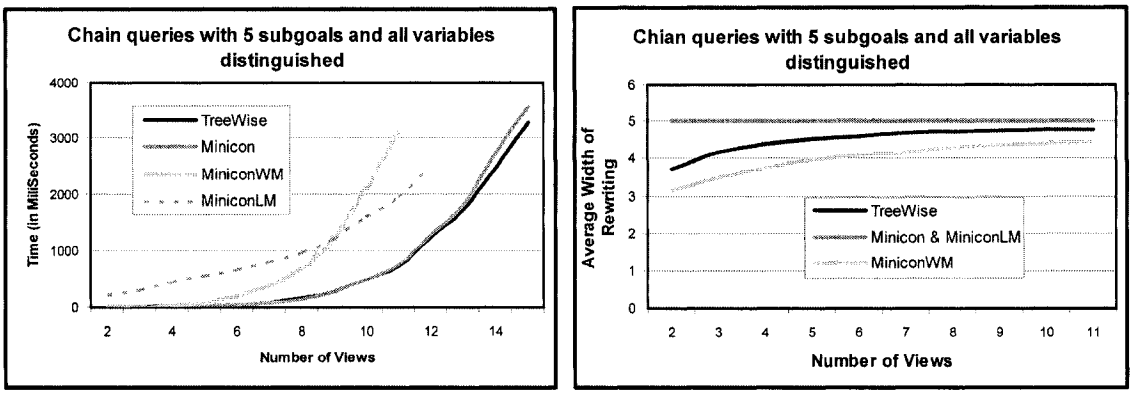

Figure 6.4: Result of experiments for chain queries of size 5 subgoals with all variables distinguished.

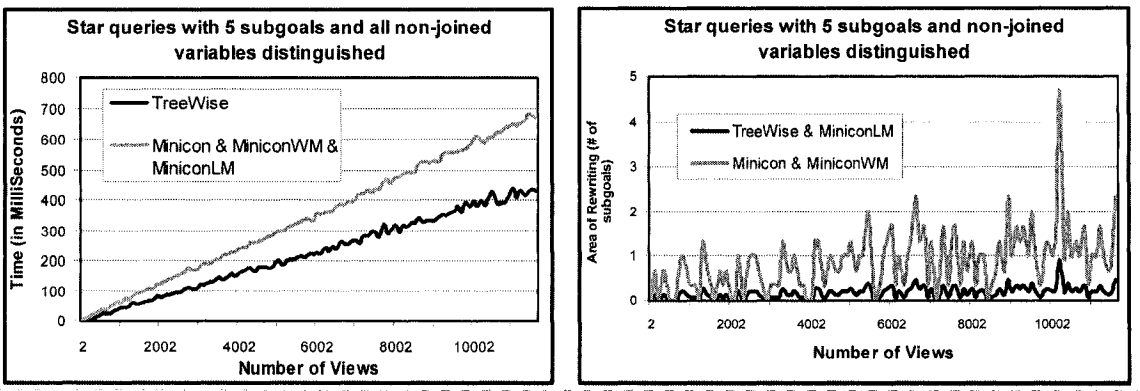

Figure 6.5: Experimental results for star queries of size 5 subgoals and all variables that do not participate in joins as distinguished.

central subgoal (that is joined with all other subgoals) are distinguished, only identical views may contribute to rewriting generation. Figure 6.5(a) shows that in this case, all the four algorithms show linear behavior in the number of views, however TreeWise generally outperforms other versions of Minicon. This is again due to redundancy in generation of MCD candidates by Minicon, which in turn results in more queries in rewritings generated by Minicon and MiniconWM (Figure 6.5(b)). Although not shown here, as we increase the size of queries, different versions of Minicon algorithms require more time to complete than TreeWise.

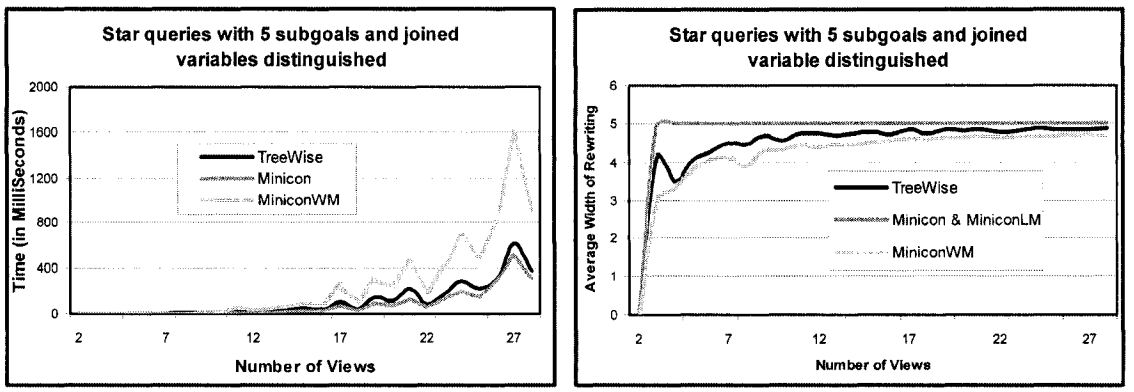

Figure 6.6: Experimental results for star queries of size 5 subgoals with all joined variables distinguished.

Second case of star queries include query and views that have all their *join attributes* distinguished. In this case, since all variables of the central subgoal are distinguished, every central subgoal of each view (depending on the size of the subgoals pool) can produce a mapping tuple for that subgoal, and hence the number of rewriting queries are usually large, although not as large as case 2 of chain queries. For such queries, as Figure 6.6(a) shows, performance of Minicon and TreeWise algorithms are very similar. Due to presence of large number of rewritings, the overhead of width optimization is quite considerable and grows rapidly as the number of views increases. Since MCD tuples generated are usually of minimal size, the overhead of length optimization is fixed in this case and its impact on the quality of rewritings is minimal, and hence figures are omitted.

As Figure 6.6(b) shows, similar to case 2 of chain queries, rewritings generated by TreeWise algorithm are of better quality than those generated by Minicon. Using width optimization of Minicon proves effective but comes at as increase in costs.

6.1.3 Complete Queries

For this class of queries where there is a unique join between every two subgoals, we examine two cases. In the first case, we fix the number of distinguished variables in the query and views to 3. Considering 5 subgoals in the body of query and each view, we need at least 4 attributes in each subgoal in order to make the joins possible. Since we assume at most 50 percent of the variables to be distinguished, which is a low percentage, we expect very few non-empty rewritings for this case of complete queries. As a result, the two optimization of Minicon should not incur a considerable processing overhead. Figure 6.7(a) confirms this expectation and also shows, similar to earlier cases, that view-based approach of TreeWise algorithm performs more efficiently than any of the three versions of Minicon. In this case, TreeWise outperforms Minicon algorithms by a factor of 3. For most of these cases, there is no rewriting and therefore the quality aspect of these algorithms can not be fairly compared. In cases where there is a rewriting, the quality of TreeWise is the same as MiniconLM but better than Minicon and MinconWM (Figure 6.7(b)).

In the second set of experiments for this class of queries, we used similar settings as the previous case with only one difference; Now all the variables of the query and views are distinguished. In this case, similar to case 2 of chain queries, the number of rewritings is exponential and therefore all four algorithms exhibit non-linear behavior in the number of views. Although not shown here, it is evident that TreeWise algorithm with its top-down approach, produces better quality rewritings compared to Minicon, and in less time. We also note that the quality of rewritings generated by MiniconWM and TreeWise differ much less compared to the case of chain queries. It is due to the fact that larger number

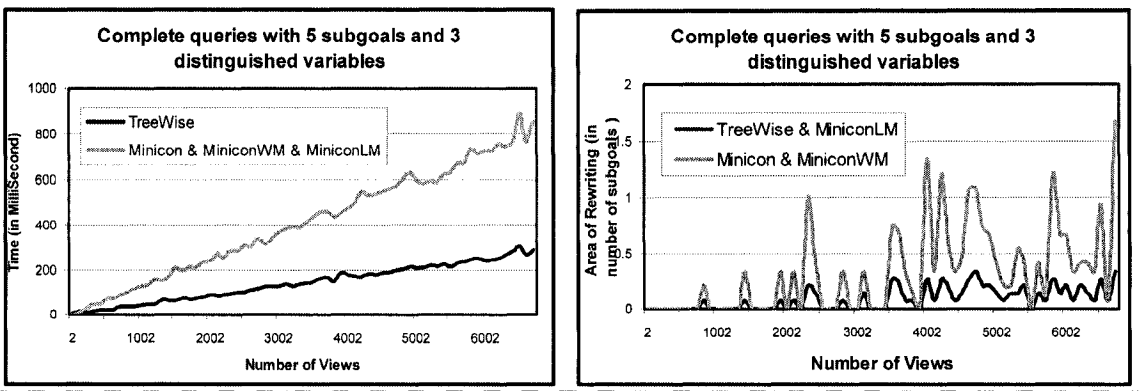

Figure 6.7: Results of experiments with complete queries of size 5 subgoals and 3 distinguished variables.

of joins generally decrease the chance of width optimization, and most opportunities for optimization is taken advantage of by the TreeWise algorithm.

6.1.4 Random Queries

In addition to the above three classes of queries, we also used in our experiments queries chosen randomly. They include random queries and views of the same sizes (5 subgoals) taken from a pool of standard conjunctive queries, produced by also using the sample generator. Our results indicate that when the queries are larger than certain sizes (3 subgoals), TreeWise outperforms Minicon in most cases. The results also show that the overhead of the length and width optimizations of Minicon are generally significant. Overhead of the length minimization grows almost exponentially in the number of generated MCDs, and for the width optimization, it grows rapidly as the number of queries in the result rewriting increases.

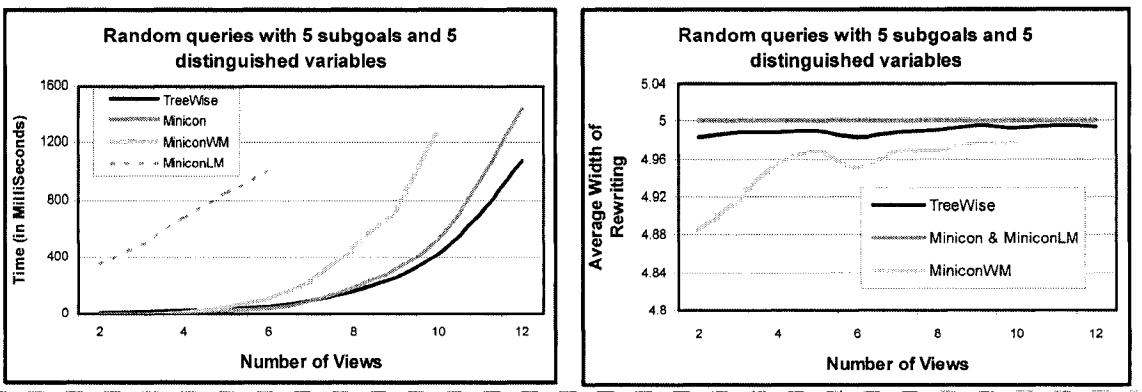

Figure 6.8: Results of experiments with random queries of size 5 subgoals and 5 distinguished variables.

Query type	Distinguished variables	Size of rewriting	Scalabitliy
Chain	Two	Very small (identical views)	Linear
Chain	All	Very large (exponential)	Non-linear
Star	Non joined	Very small	Linear
Star	Joined	large(at least central subgoal of each view covers a sugoal in Q)	Non-linear
Complete	25 percent	Very small	Linear
Complete	100 percent	Very large (exponential)	Non-Linear

Table 6.1: Scalability of TreeWise in number of views for different classes of queries.

6.2 Scalability of TreeWise

We conducted some experiments to study scalability of the TreeWise algorithm under different circumstances when the number of views becomes large. This issue is important in the context of data integration systems, which often deals with large number of data sources (views). Using the results of our previous experiments, we investigate scalability of the four algorithms for different classes of standard conjunctive queries. Summarizing these results in Table 6.2, we note that TreeWise is scalable in the number of views for different classes of queries. In another set of experiments, we used TreeWise in isolation trying to push the algorithm to its limits.

Table 6.2 shows the number of views that the TreeWise algorithm could process under 10 seconds (as done in [PH01]). As the results indicate, in some cases the TreeWise algorithm can process thousands of views within that time. An interesting case to note is the star queries with distinguished joined variables. For this case, it seems that as the size of query and views increases, the number of views that can be processed by our algorithm under 10 seconds also increases. This is due to the fact that when the number of subgoals in the query increases, the possibility of a subgoal (mostly the central subgoal) being left uncovered also increases. However, if every subgoal is covered by some view, then the number of queries in the rewriting result is almost exponential.

Query type	Disntinguished variables	# of subgoals	# of Views
Chain	Two	5	157650
Chain	Two	99	2852
Chain	All	3	87
Chain	All	99	4
Star	Non joined	5	166500
Star	Non joined	99	3250
Star	Joined	10	18
Star	Joined	99	24

Table 6.2: The number of views that the TreeWise algorithm can answer under 10 seconds.

Figure 6.9 displays the user-interface of our rewriting tool, showing the output generated by TreeWise algorithm for query and views in Example 12. The output shows the performance (in milliseconds) and structure of the output for each phase of TreeWise algorithm. This output shows, for the case of Example 12, constructing graphs for the query and views takes less than 1 millisecond.

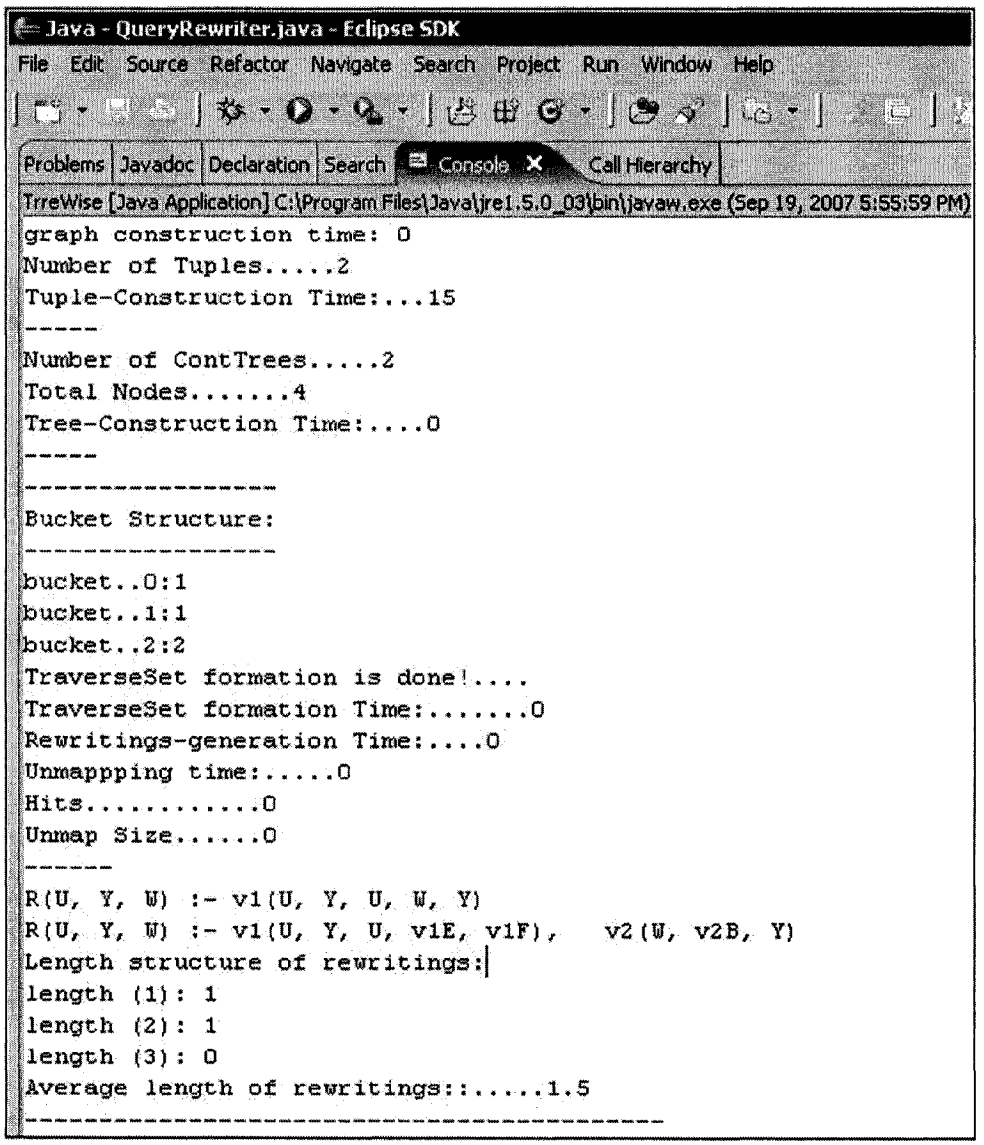

Figure 6.9: Output of TreeWise for query and views of Example 12.

6.3 Summary

Our experimental results indicated that in all cases but one (case 2 of star queries), our top-down, view-based approach of TreeWise algorithm outperforms the bottom-up, subgoal-based approach of Minicon. There are several factors contributing to this advantage:

- Naturally, a top-down approach produces fewer mapping-tuples during the tuple construction phase, compared to the MCDs generated by Minicon during its first phase. In the second phase, TreeWise generates new tuples only if necessary. Furthermore, the tree structure which keeps track of child-parent relationships improves efficiency of the algorithm in the third phase.

- As the number of subgoals in the body of query and (specially) views increases, the overhead of subgoal-based approach of Minicon results in increasing performance gap between the two algorithms.

- When there is a large number of joins between existential variables (case 1 of chain, star, and complete queries), TreeWise with its graph-based approach in recording dependencies between subgoals runs increasingly faster than Minicon algorithm, which discovers these dependencies multiple times.

The overhead of length and width optimization routines for Minicon algorithm, when performed, is quite high. In general, the overhead of length optimization grows rapidly in the number of MCDs. In cases where the number of queries in the rewriting is high (case 2 of chain and complete queries), the overhead of width optimization is almost exponential.

As for the quality of generated rewritings, we may conclude the following from the results of our experiments. In cases where there are numerous joins between existential variables (case 1 of chain, star and complete queries), the Minicon algorithm being subgoal-based produces redundant rewriting queries. In such cases, length optimization in Minicon reduces the length of the rewriting to that of TreeWise. In general, the top-down approach of TreeWise algorithm produces rewritings of lower widths than Minicon. The width optimization of Minicon generally produces rewritings of lower widths than TreeWise algorithm, but at an increased cost. The difference between the width is due to the manner in which we compare tuples in TreeWise in the second phase to create trees. This is the main limitation of our algorithm. In the future, we plan to investigate this issue of the algorithm more closely to find ways to improve the operations of comparison and tree construction processes in phase two. As the number of generated tuples that are not minimal increases, the overhead of this phase increases. This overhead, however, is always polynomial in the size of query and views. Furthermore, in some cases with query and views of small sizes (less than 3 subgoals), the overhead of top-down approach is slightly more than Minicon.

Finally as shown above, similar to Minicon in cases where the number of queries in a rewriting is not large (case 2 of chain and complete queries), the TreeWise algorithm scales linearly in the number of views. In these cases, TreeWise can process thousands of views under 10 seconds. For cases with very large number of queries in the rewriting, both Minicon and TreeWise perform poorly due to exponential number of resulting queries in the rewriting.

Chapter 7

Conclusions and Future Work

7.1 Conclusions

Continuous growth in the volume of data distributed across the globe indicates the growing importance of data integration systems. The problem of answering queries in the context of such systems has been the subject of numerous studies. In this thesis, we investigated the problem of rewriting a query in the LAV approach, where data sources in data integration systems are defined as views in the form of conjunctive queries. Our analysis of existing algorithms shows that rewritings generated by those algorithms are, in general, expensive to evaluate. This is due to the fact that they all take a bottom-up and subgoal-based approach to rewriting. In this thesis, we took an alternative top-down approach to generating rewriting and developed an algorithm that implements that approach. The contributions of this thesis are as follows:

- We presented a graph-based model, called HMCQ, for representing conjunctive queries. This model extends the hyper-node model proposed for data models in

[LP90]. The goal of this model is to provide an abstraction for our top-down approach to rewriting. For this, we first established the related containment and rewriting concepts to HMCQ. Then we used this model to investigate the conditions that must be satisfied during each phase of process to guarantee maximally contained rewriting.

- Utilizing the conditions identified, we introduced a top-down algorithm, called TreeWise, for generating maximally contained rewriting for standard conjunctive query and views. The algorithm is designed to efficiently produce better quality rewritings than Minicon. This was achieved through maximizing the benefit of top-down approach by efficiently utilizing the conditions outlined in the abstraction process. To verify this claim, we presented details of our numerous experiments, in which we studied TreeWise and compared it with the Minicon algorithm for several classes of queries. The results of these experiments indicated that top-down and view-based approach of TreeWise results in increased efficiency and improved quality of rewriting. The experiments also indicated that in majority of cases, TreeWise outperforms Minicon. We also studied the scalability issue in our experiments. Our results showed scalability of TreeWise in similar to that of Minicon.

7.2 Future Work

There are several issues that we are interested to investigate as future work, related to the HMCQ model and the TreeWise algorithm. These are as follows.

- *Extending HMCQ:* We would like to extend the HMCQ model for handling two other

classes of queries : arithmetic equality expressions and also general comparison. As mentioned in chapter 4, our current version of HMCQ is insufficient for capturing the subtleties of the former class. As for the general comparison cases, we think concepts of inequality graphs presented in [ALM02] should be merged with HMCQ to capture the order of the variables in the query. Also as shown in [ALM02], using inequality graphs, it is possible to capture possibility of exporting variables, where an existential variable can be treated as distinguished using some appropriate mappings.

- *Optimizing TreeWise:* Another possible direction to extend our work is to improve TreeWise algorithm to further reduce the widths of rewriting queries to those provided by Minicon with post-processing. For this, we need to first examine the second phase of the algorithm more closely. More specifically, we should investigate the manner in which we compare the partial mapping tuples and build the binary trees to identify an "optimal" order for this process. Naturally our aim here is to create trees of shorter heights. Also, adding some way of bookkeeping in the third phase of the algorithm can help identify the cases where the widths of the rewriting can be further reduced by merging view subgoals in the rewriting queries. These cases are related to view subgoals representing tuples from different levels of the same binary tree. Unlike post-processing of the Minicon algorithm, this optimization does not need to compare the view definitions and their mappings with each other to verify possibility of merging. In TreeWise, this verification is already captured in the binary tree structures. We just need to identify the view definitions in the body of the

rewriting coming from the same tree.

- *Extending TreeWise:* In the future extensions of the TreeWise algorithm, we plan to handle cases of built-in predicates in the forms of LSI and arithmetic equality expressions. For LSI queries, this extension is straightforward and is similar to the extension proposed for Minicon. This only affects the first phase of the TreeWise algorithm. In case of equality expressions, the algorithm can adopt a matrix approach to solve the system of equations to determine computability of the attributes and also constraints satisfaction.

Bibliography

[AD98] Serge Abiteboul and Oliver M. Duschka. Complexity of answering queries using materialized views. In *PODS '98: Proceedings of the seventeenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, pages 254–263, New York, NY, USA, 1998. ACM.

[Ali05] Ali Kiani and Nematollaah Shiri. Containment of conjunctive queries with arithmetic expressions. In *CoopIS '05: Proceedings of 13th Int'l Conf. on Cooperative Information Systems*, pages 439–452, Berlin , Heidelberg, 2005. Springer.

[ALM02] Foto Afrati, Chen Li, and Prasenjit Mitra. Answering queries using views with arithmetic comparisons. In *PODS '02: Proceedings of the twenty-first ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 209–220, New York, NY, USA, 2002. ACM.

[CK86] S Cosmadakis and P Kanellakis. Parallel evaluation of recursive rule queries. In *PODS '86: Proceedings of the fifth ACM SIGACT-SIGMOD symposium on Principles of database systems*, pages 280–293, New York, NY, USA, 1986. ACM.

[CLM81] Ashok K. Chandra, Harry R. Lewis, and Johann A. Makowsky. Embedded implicational dependencies and their inference problem. In *STOC '81: Proceedings of the thirteenth annual ACM symposium on Theory of computing*, pages 342–354, New York, NY, USA, 1981. ACM.

[CM77] Ashok K. Chandra and Philip M. Merlin. Optimal implementation of conjunctive queries in relational data bases. In *STOC '77: Proceedings of the ninth annual ACM symposium on Theory of computing*, pages 77–90, New York, NY, USA, 1977. ACM.

[CV92] Surajit Chaudhuri and Moshe Y. Vardi. On the equivalence of recursive and nonrecursive datalog programs. In *PODS '92: Proceedings of the eleventh ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, pages 55–66, New York, NY, USA, 1992. ACM.

[DFJ+96] Shaul Dar, Michael J. Franklin, Björn Jónsson, Divesh Srivastava, and Michael Tan. Semantic data caching and replacement. In *VLDB '96: Proceedings of the 22th International Conference on Very Large Data Bases*, pages 330–341, San Francisco, CA, USA, 1996. Morgan Kaufmann Publishers Inc.

[Dus97] Oliver M. Duschka. *Query planning and optimization in information integration.* PhD thesis, Stanford University, Stanford, CA, 1997.

[FP04] Foto Afrati, Chen Li, and Prasenjit Mitra. On containment of conjunctive queries with arithmetic comparisons. In *Proc. of Int'l conference on Extending Database Technology (EDBT)*, pages 459–476, 2004.

[GM99] Gösta Grahne and Alberto O. Mendelzon. Tableau techniques for querying information sources through global schemas. In *ICDT '99: Proceedings of the 7th International Conference on Database Theory*, pages 332–347, London, UK, 1999. Springer-Verlag.

[GSUW94] Ashish Gupta, Yehoshua Sagiv, Jeffrey D. Ullman, and Jennifer Widom. Constraint checking with partial information. In *PODS '94: Proceedings of the thirteenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, pages 45–55, New York, NY, USA, 1994. ACM.

[Hal01] Alon Y. Halevy. Answering queries using views: A survey. *The VLDB Journal*, 10(4):270–294, 2001.

[KB94] Arthur M. Keller and Julie Basu. A predicate-based caching scheme for client-server database architectures. In *PDIS '94: Proceedings of the Third International Conference on Parallel and Distributed Information Systems*, pages 229–238, Washington, DC, USA, 1994. IEEE Computer Society.

[Klu88] Anthony Klug. On conjunctive queries containing inequalities. *Journal of the ACM*, 35(1):146–160, 1988.

[LMS95] Alon Y. Levy, Alberto O. Mendelzon, and Yehoshua Sagiv. Answering queries using views (extended abstract). In *PODS '95: Proceedings of the fourteenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, pages 95–104, New York, NY, USA, 1995. ACM.

[LP90] M. Levene and A. Poulovassilis. The hypernode model and its associated query language. In *JCIT: Proceedings of the fifth Jerusalem conference on Information technology*, pages 520–530, Los Alamitos, CA, USA, 1990. IEEE Computer Society Press.

[LRO96a] Alon Y. Levy, Anand Rajaraman, and Joann J. Ordille. Query-answering algorithms for information agents. In *AAAI '96: Proceedings of the Thirteenth National Conference on Artificial Intelligence and the Eighth Innovative Applications of Artificial Intelligence Conference*, pages 40–47, Menlo Park, 1996. AAAI Press / MIT Press.

[LRO96b] Alon Y. Levy, Anand Rajaraman, and Joann J. Ordille. Querying heterogeneous information sources using source descriptions. In *VLDB '96: Proceedings of the 22th International Conference on Very Large Data Bases*, pages 251–262, San Francisco, CA, USA, 1996. Morgan Kaufmann Publishers Inc.

[NN08] Nima Mohajerin and Nematollaah Shiri. A top-down approach to rewriting conjunctive queries using views. In *SKDB '08: Proceedings of EDBT Workshop on Semantics in Data and Knowledge Bases*, Nantes, France, 2008. Springer.

[PH01] Rachel Pottinger and Alon Halevy. Minicon: A scalable algorithm for answering queries using views. *The VLDB Journal*, 10(2-3):182–198, 2001.

[Qia96] Xiaolei Qian. Query folding. In *ICDE '96: Proceedings of the Twelfth International Conference on Data Engineering*, pages 48–55, Washington, DC, USA,

1996. IEEE Computer Society.

[Rac04] Rachel A. Pottinger. *Processing queries and merging schemas in support of data integration.* PhD thesis, University of Washington, Seattle, WA, USA, 2004.

[Shm93] Oded Shmueli. Equivalence of datalog queries is undecidable. *Journal of Logic Programming*, 15(3):231–241, 1993.