QUERY PROCESSING AND OPTIMIZATION IN INFORMATION-INTEGRATION SYSTEMS

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF COMPUTER SCIENCE
AND THE COMMITTEE ON GRADUATE STUDIES
OF STANFORD UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

Chen Li
August 2001

I certify that I have read this dissertation and that in my opinion it is fully adequate, in scope and quality, as a dissertation for the degree of Doctor of Philosophy.

Jeffrey D. Ullman
(Principal Adviser)

I certify that I have read this dissertation and that in my opinion it is fully adequate, in scope and quality, as a dissertation for the degree of Doctor of Philosophy.

Foto N. Afrati

I certify that I have read this dissertation and that in my opinion it is fully adequate, in scope and quality, as a dissertation for the degree of Doctor of Philosophy.

Gio Wiederhold

Approved for the University Committee on Graduate Studies:

Abstract

The advent of the Internet provides us the access to many autonomous and heterogeneous information sources. The purpose of information integration is to support seamless access to these data sources. To deal with source heterogeneity, many systems use a mediation architecture, in which a mediator processes user queries by accessing source data. There are two approaches to information integration: the source-centric approach (taken by the TSIMMIS system at Stanford), and the query-centric approach (taken by many systems, such as the Information Manifold at AT&T). My thesis focuses on efficient query processing in both approaches. In the first approach, I work on how to process queries efficiently when sources have limited capabilities of answering queries. In the second approach, I develop query-optimization techniques, such as how to use mediator caching to improve query performance, and how to generate efficient rewritings of queries using views. Most of my thesis work is developed in the TSIMMIS project at Stanford University.

Acknowledgments

I would like to thank my advisor, Jeff Ullman. He gave me a lot of freedom to choose the research problems that interested me, and he was also always willing to discuss my ideas. Not only did Jeff advise me how to do research, he also taught me how to solve problems in my life. Hector Garcia-Molina advised me on many research topics throughout my Stanford years. Jennifer Widom provided me an excellent model for organization, planning, and balancing work with the rest of life. Gio Wiederhold provided much help not only on my research on information integration, but also on my other interests in image databases. Foto Afrati kindly served on my oral committee and read my dissertation. Our collaborations have always been productive and enjoyable.

I dedicate this work to my wife, Yu Zhao. I thank her for her love, courage, and support. She always stands by me, keeping me balanced and focused. I am also indebted to my parents, Songtao Li and Jinfeng Wang, for their constant love. Thanks are also due to my brother, Nianqing Li, for sharing thoughts about life.

I thank my co-authors: Foto Afrati, Mayank Bawa, Edward Chang, Hector Garcia-Molina, Yannis Papakonstantinou, Jeff Ullman, Murty Valiveti, Vasilis Vassalos, and Ramana Yerneni, for their ideas, encouragement, collaborations, and for teaching me how to do good research.

The Stanford Database Group provided a great environment for research and play. The Social Committee organized many wonderful events. I will miss the delicious DB-lunch food every Friday, and the fruitful discussions in many project meetings. Marianne Siroker and Sarah Weden were always available in making sure all business was done without much effort on my part. I enjoyed the time with my officemates, Arvind Arasu, Mayank Bawa, Kevin Chang, Neil Daswani, Prasenjit Mitra, and Jun Yang. We had a lot of fun.

I thank all the TSIMMIS researchers: Sudarshan Chawathe, Hector Garcia-Molina, Joachim Hammer, Svetlozar Nestorov, Yannis Papakonstantinou, Dallan Quass, Anand

Rajaraman, Michael Rys, Murty Valiveti, Vasilis Vassalos, Jeff Ullman, Jennifer Widom, Ramana Yerneni, for all their time and effort on the TSIMMIS base. TSIMMIS is a product of everyone's hard work.

This research was partially supported by the Stanford Graduate Fellowship and the National Science Foundation under grant #IRI-9631952, and by equipment grants from IBM Corporation.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Information integration

The purpose of information integration (a.k.a. data integration, information mediation) is to support seamless access to autonomous, heterogeneous information sources. These sources can be legacy databases, corporate databases connected by intranets, and data sources on the World Wide Web. Users can access these data sources as if they were accessing one large database.

Information integration has recently received considerable attention because of its relevance to many data-management applications. In particular, the advent of the Internet provides us with a huge amount of information stored in many sources on the Web. Information integration provides uniform interfaces for users to access these data sources. For instance, comparison-shopping companies collect merchandise information from online stores, such as prices, manufacturers, and availabilities. They provide Web interfaces for users to submit queries to find merchandise information, thus users can find best deals without going to individual underlying sources. For another example, in supply-chain management, many suppliers have products that buyers are interested in. By building an intermediate layer among them, suppliers and buyers can share the product information easily. Furthermore, many biological discovery groups are challenged with a plethora of public, private, intercompany, intra-company, current, and legacy data sources. The Human Genome Project (http://www.ornl.gov/hgmis/) is an example. Biological data is often stored in multiple sites and databases. Biological discovery is being slowed by an inability to access the data, and a rigidity in the handling and manipulation of the data. Information integration provides these groups the ability to access, manipulate, and understand the increasingly vast amounts of data.

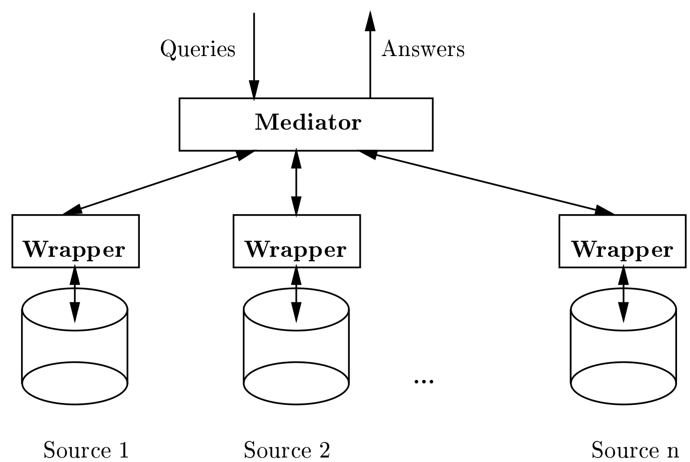


Figure 1.1: Mediation architecture.

Data sources in information integration can be quite heterogeneous, i.e., different sources can use different data models, schemas, and query interfaces. To deal with this heterogeneity, many systems use a mediation architecture [Wie92], as shown in Figure 1.1. In the architecture, a wrapper is built on the top of each source. The purpose of the wrapper is to translate the source data model to a universal data model shared by the wrappers, so that the mediator can exchange data with the sources. Another functionality of the wrapper is to do local query processing to retrieve data from the source. For example, a wrapper on a Web source can fill in a Web search form, send an HTTP request to the source, and parse data in the returned HTML page to extract useful values. On the top of all the wrappers, a mediator can accept user queries and answer the queries by accessing the relevant sources to retrieve the necessary data.

1.2 Two Approaches to Information Integration

There are two approaches to information integration: the *query-centric approach* and the *source-centric approach* [Dus97, Ull97]. Both approaches have been widely used by many systems.

1.2.1 The Query-Centric Approach

In the query-centric approach, the mediator exports *views* defined on the source data. After a user poses a query on the synthesized views, the mediator *expands* the query to a plan that involves source data. The TSIMMIS project [C+94] takes this approach. The following is an example.

EXAMPLE 1.2.1 Consider a mediator on the top of the following three movie sources.

- `R(Star, Title)`. A tuple `R(s,t)` means that star `s` starred in movie `t`.
- `S(Title, Studio)`. A tuple `S(t,d)` means that movie `t` was produced by studio `d`.
- `T(Title, Year)`. A tuple `T(t,y)` means that movie `t` was made in year `y`.

The mediator synthesizes the following view:

```
CREATE VIEW Movie
   SELECT Star, Title, Studio, Year
   FROM R, S, T
   WHERE R.Title = S.Title AND S.Title = T.Title;
```

Suppose a user wants to find all the movies produced by Warner Brothers in 1999, and starred by Keanu Reeves. She can pose the corresponding query Q_0 on the view:

```
SELECT Title
FROM Movie
WHERE Star = 'Reeves' AND Studio = 'Warner' AND Year = '1999';
```

For simplicity, we use abbreviations for the constants. Using the view definition, the mediator expands query Q_0 to the corresponding query Q on the source relations:

```
SELECT Title
FROM R, S, T
WHERE R.Title = S.Title AND S.Title = T.Title
  AND R.Star = 'Reeves' AND S.Studio = 'Warner' AND T.Year = '1999';
```

□

1.2.2 The Source-Centric Approach

In the source-centric approach, there is a collection of *global predicates* on which user queries are formulated. Each data source is associated with one or more views, which are also defined in terms of these global predicates. The following is an example.

Assume we have a source that has information about car dealers. It has the following view that includes car dealers in the city Palo Alto:

```
CREATE VIEW PA_Dealer
   SELECT Name, Make
   FROM Dealer
   WHERE City = 'Palo Alto';
```

in which `Dealer(name,city,make)` is a global predicate (a.k.a. world relation) that can be used to formulate queries and source views. If a query asks for dealers in Palo Alto that sell Toyota cars, the following is the query:

```
SELECT Name
FROM Dealer
WHERE City = 'Palo Alto' AND Make = 'Toyota';
```

We can answer the query by using the source view as follows:

```
SELECT Name
FROM PA_Dealer
WHERE Make = 'Toyota';
```

In general, after a user poses a query, the mediator decides how to use the source views to answer the query. This process is also known as *answering queries using views* [LMSS95]. Systems that take this source-centric approach include the Information Manifold [LRO96], and Infomaster [GKD97].

1.3 Limited Source Capabilities

Source relations in information integration often have limited query interfaces. These sources do not allow us to retrieve all their data "for free." Instead, they have limitations on access patterns to their data; that is, one must provide values for certain attributes of a relation in order to retrieve its tuples. For example, if we view the source IMDB.COM as a relation that provides movie information. This source publishes several Web search forms, using which users can submit movie queries. Figure 1.2 is an example. Using this form, a user must provide either a movie title, a cast name, or a character name, then the source returns movies that satisfy the conditions. Without providing a value, we cannot retrieve information from the source. There are many reasons for this kind of restrictions, such as convenience for users to retrieve information, and concerns of security, privacy, and performance. In some cases, legacy databases or structured files may also have limited interfaces, leading to similar restrictions.

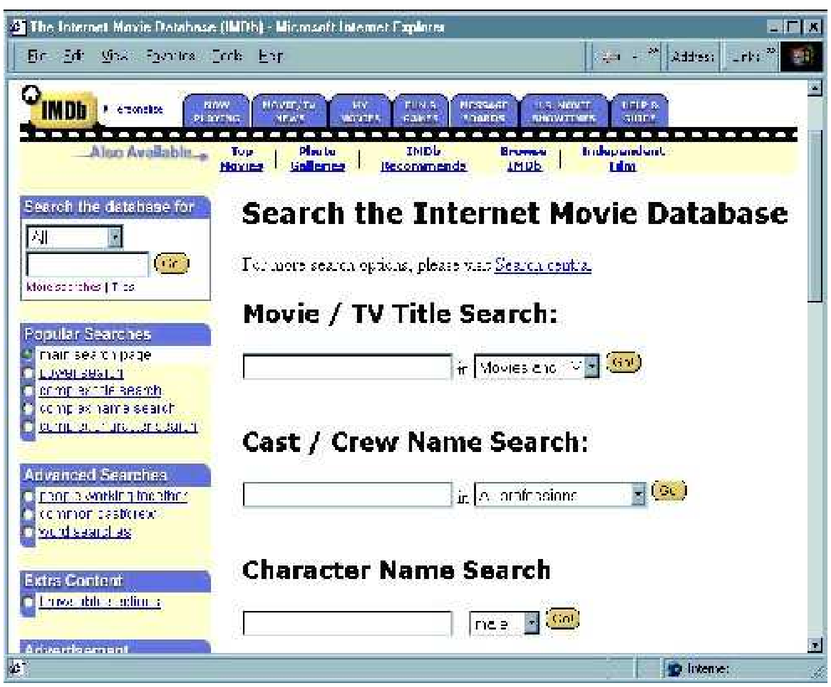


Figure 1.2: A Web search form of IMDB.COM.

Source restrictions make it challenging for the mediator to process and optimize queries.

In particular, the mediator should consider the source capabilities while generating a *feasible* and *efficient* plan to answer a query, which accesses the source relations using legal patterns to compute answers efficiently.

1.4 Research Issues

This thesis focuses on efficient query processing in information integration. We consider problems that arise in systems that take the query-centric approach and the source-centric approach.

1.4.1 Query Processing in the Presence of Source Limited Capabilities

The first part of this thesis focuses on query optimization in the query-centric approach to information integration, especially when sources have limited capabilities. The following are areas of investigation.

Computing Answers to Queries Efficiently

Given a query on source relations with restrictions, we want to know whether there exists a plan that computes the answers to the query by accessing the relations using legal patterns [LYV+98]. In particular, we should generate a *feasible* plan that respects the limitations of the relation interfaces, and the existence of such a plan closely depends on the capabilities of the relations. In addition, there might be different plans to compute the same answers to a query, while these plans have quite different efficiency. Thus we want to generate not only a feasible plan, but also a plan that is *efficient*. We study this problem in Chapter 3.

Computing Maximal Partial Answers to Queries

In some cases, we might not be able to compute all the answers to a query due to the source restrictions. If the user is interested in a partial answer, we can compute a maximal partial answer by retrieving as much information as possible from the relations. In this case, we can answer the query by borrowing bindings for certain domains from other relations that are not used in the query.

EXAMPLE 1.4.1 Assume a source R requires a movie title to return its movie information, such as studio names and star names. Suppose a user wants to find movies made by

Universal Studios. We cannot retrieve movie information from R directly due to its restriction. However, if there is a Web source that provides movie titles for free, then we can use these movies to access the relation R to retrieve some movies, and return those made by Universal Studios. Essentially we can compute a partial answer to the query by borrowing movie titles from the second relation. □

In Chapter 4 we study how to compute a maximal answer to a query by using information from other relations. We solve several optimization problems to trim useless sources. The first problem is how to decide which relations should be accessed to compute the maximal answer. As the number of relations increases, and their restrictions become complicated, it is unclear how to decide which relations can really help us answer a query. We develop an efficient algorithm for finding all the useful relations that need to be accessed to compute the maximal answer to a query. Another optimization problem is to test query containment in the presence of source restrictions, i.e., whether the maximal answer to a query is contained in that to another query. We show that this problem is decidable by using some existing results in [CGKV88], and develop a polynomial-time algorithm for testing the containment in certain cases.

Computing Complete Answers to Queries

Often we cannot retrieve all tuples from relations due to their limited query capabilities. In particular, after some queries are sent to sources, there can always be some tuples in the relations that are not retrieved, since we cannot provide the necessary values to retrieve them. In Chapter 5, we answer the following question: given a query on relations with restrictions, is there a plan that accesses the relations with legal patterns, such that this plan computes *all* the answers that satisfy the query conditions? Since we cannot retrieve all tuples from the relations, we need to do reasoning about whether the answers computed by a plan are all the answers or not. We study this problem for several classes of queries, and give the corresponding algorithms and decidability results.

1.4.2 Using Mediator Caching to Improve Performance

In information integration, each source access is expensive due to network traffic and delay, dynamic source availability, and possible source charges. To reduce the number of source accesses, we can cache the results of previous queries at the mediator, and use the cached data to answer future queries without accessing the sources. In addition, since the mediator has only limited resources to store data, it might not be possible to cache the results of all queries. Thus we need to decide what query results should be kept in the cache. In Chapter 6 we study how to decide what query results should be cached at the mediator in order to answer as many future queries as possible.

1.4.3 Generating Efficient Plans for Queries Using Views

In the source-centric approach to information integration, for each user query, the mediator needs to decide how to compute the answers to the query using the source views [LMSS95]. Several algorithms have been developed on how to answer queries using views, such as the Bucket algorithm [GM99a, LRO96], the Inverse-Rule algorithm [DG97, Qia96, AGK99], the MiniCon algorithm [PL00], and the Shared-Variable-Bucket algorithm [Mit01]. However, most of these algorithms focus on *how* to generate a plan for a query, while their generated plans might not be efficient. The following is an example.

EXAMPLE 1.4.2 Consider the following two global predicates:

- `Car(Make,Dealer)`;
- `Loc(Dealer,City)`.

A source view is defined on the predicates:

```
CREATE VIEW V
   SELECT Make, Dealer, City
   FROM Car, Loc
   WHERE Car.Dealer = Loc.Dealer;
```

The following query Q:

```
   SELECT Make, City
   FROM Car, Loc
   WHERE Car.Dealer = 'Anderson' AND Loc.Dealer = 'Anderson';
```

asks for car makes and cities of dealer `anderson`. We can use the following plan P_1 to answer Q using V:

```
SELECT Make, City
FROM V
WHERE Dealer = 'Anderson';
```

However, existing algorithms (such as the Bucket algorithm and the MiniCon algorithm) generate the following plan P_2 instead:

```
SELECT V1.Make, V2.City
FROM V AS V1, V AS V2
WHERE V1.Dealer = 'Anderson' AND V2.Dealer = 'Anderson';
```

Clearly P_2 is more efficient than P_1, since P_2 needs one access to view v, while P_1 needs two accesses and a join operation. $\square$

These algorithms generate plan P_1 instead of P_2 because they take the open-world assumption about the views. (See Section 2.4 for the definitions of the open-world assumption and the closed-world assumption.) In Chapter 7 we study the problem of generating efficient, equivalent rewritings using views to compute the answers to a query by taking the closed-world assumption.

1.5 Thesis Organization

The rest of the thesis is organized in seven chapters. Chapter 2 introduces the notation used throughout the thesis. Chapter 3 to Chapter 6 cover query-optimization problems in systems that take the source-centric approach to information integration. Chapter 3 discusses how to optimize large-join queries when sources have limited access patterns. As we have seen in Example 1.2.1, we want to generate feasible plans that can answer a query efficiently. We study the complexity of this problem under several cost models, and give efficient algorithms for finding good plans.

In Chapter 4 we study how to compute a maximal answer to a query by borrowing information from other relations. We solve two optimization problems. The first one is how to decide which relations can be used to answer a query. The second one is to test query containment. Using these optimization techniques, we can trim useless source accesses to compute the maximal answer to a query.

In Chapter 5, we answer the following question: given a query on relations with restrictions, is there a plan that computes the complete answers to the query by accessing the relations with legal patterns? The *complete* answers to a query are all the answers that satisfy the query conditions. If the relations did not have restrictions, we can easily compute the complete answers by retrieving all the tuples from relations. However, since we can only retrieve *some* tuples, we need to decide whether the answers computed by a plan are the complete answers or not. We study this problem for several classes of queries, such as conjunctive queries, conjunctive queries with arithmetic comparisons, unions of conjunctive queries, and datalog queries. We give algorithms and decidability results for these classes.

Chapter 6 discusses how to use mediator caching to improve query performance. In particular, we study what query results should be cached at the mediator in order to answer as many future queries as possible. We show that traditional query containment [CM77] is *not* a good basis for deciding whether or not a query result should be kept. Instead, we introduce a concept called *equipotence* that describes the fact that two sets of views have the same power to answer queries. We use this concept to discuss how to minimize a set of query results without losing its *query-answering power*.

Chapter 7 studies the following problem that exists in systems that take the query-centric approach to information integration: how to generate *efficient* plans for a query using source data? In particular, in what space should we search for optimal rewritings? How do we find optimal rewritings efficiently? How does an optimizer generate an efficient physical plan from a logical plan by considering the view definitions? In this chapter we solve these problems by considering several cost models. We define search spaces for finding optimal rewritings, and develop efficient algorithms for finding optimal rewritings in each search space. In Chapter 8 we summarize the results in the thesis and discuss future work on data integration.

Some of the material in this thesis appears in conference and journal papers [YLUGM99, LC00, LC01b, LBU01, ALU01, LC01a]. At the end of each chapter, we conclude the chapter and discuss related work.

Chapter 2

Preliminaries

In this chapter we introduce the notation used throughout the thesis.

2.1 Queries

2.1.1 Conjunctive Queries

We consider a variety of queries. In particular, we focus on *conjunctive queries*, a.k.a. select-project-join (SPJ) queries. A conjunctive query ("CQ" for short) is denoted by a rule:

$$h(\bar{X}) \text{ :- } g_1(\bar{X}_1), \ldots, g_n(\bar{X}_n)$$

In the query, $h(\bar{X})$ is called the *head*, and it represents the results of the query. The *body* is a set of *subgoals* $g_1(\bar{X}_1), \ldots, g_n(\bar{X}_n)$. Each subgoal $g_i(\bar{X}_i)$ includes a relation g_i, and a tuple of arguments $\bar{X}_i$ corresponding to the relation schema. Each argument can be either a variable or a constant. The variables $\bar{X}$ are called *distinguished* variables. The query is *safe* if every distinguished variable appears in the body. For instance, given two relations $r(A_1, A_2)$ and $s(B_1, B_2, B_3)$, the following is a safe conjunctive query:

$$ans(X, Z) \text{ :- } r(X, Y), s(Y, Z, 1)$$

in which X, Y, and Z are variables, and 1 is a constant. The query corresponds to the following SPJ query:

$$\pi_{A_1, B_2}(\sigma_{A_2=B_1, B_3=1}(r(A_1, A_2) \bowtie s(B_1, B_2, B_3)))$$

Given a conjunctive query Q on a database D, the answers to Q, denoted $ANS(Q,D)$,[1] are the set of heads of Q that are obtained when we:

- Substitute constants for variables in the body of Q in all possible ways;
- Require all subgoals to become true.

For instance, given a database $D = \{r(2,3), s(3,4,1), s(3,5,1)\}$, the answers to the query above include two tuples, $ans(2,4)$ and $ans(2,5)$, corresponding to the following substitutions:

1. $X \to 2, Y \to 3, Z \to 4$;
2. $X \to 2, Y \to 3, Z \to 5$.

Definition 2.1.1 (**query containment and equivalence**) A query Q_1 is *contained* in a query Q_2, denoted $Q_1 \sqsubseteq Q_2$, if for any database D, the set of answers to Q_1 is a subset of the answers to Q_2, i.e., $ANS(Q_1, D) \subseteq ANS(Q_2, D)$. The two queries are *equivalent*, denoted $Q_1 \equiv Q_2$, if $Q_1 \sqsubseteq Q_2$ and $Q_2 \sqsubseteq Q_1$. □

Chandra and Merlin [CM77] showed that for two conjunctive queries Q_1 and Q_2, $Q_1 \sqsubseteq Q_2$ if and only if there is a *containment mapping* from Q_2 to Q_1, such that the mapping maps a constant to the same constant, and maps a variable to either a variable or a constant. Under this mapping, the head of Q_2 becomes the head of Q_1, and each subgoal of Q_2 becomes *some* subgoal in Q_1. For instance, consider the following two queries:

$$\begin{array}{lll} Q_1: & ans(X,Z) & \text{:- } r(X,Y), s(Y,Z), t(Z,a) \\ Q_2: & ans(X,Z) & \text{:- } r(X,Y), s(W,Z), t(Z,U) \end{array}$$

in which a is a constant. $Q_1 \sqsubseteq Q_2$, since the following is a containment mapping from Q_2 to Q_1:

$$X \to X, Y \to Y, W \to Y, Z \to Z, U \to a$$

In addition, $Q_2 \not\sqsubseteq Q_1$, since there is no containment mapping from Q_1 to Q_2. In particular, we cannot map variable Y to Y and W at the same time.

Besides conjunctive queries, in this thesis we also study unions of conjunctive queries, conjunctive queries with arithmetic comparisons, and datalog queries.

[1] We also use $ANS(Q,D)$ to denote the answers to a general query Q on a database D.

2.1.2 Datalog Queries

A datalog query is a Horn-clause program without function symbols. A datalog query on a database D is composed of a set of rules, and each rule has the form:

$$h(\bar{X}) \text{ :- } g_1(\bar{X}_1), \ldots, g_n(\bar{X}_n)$$

Each g_i in the body is a predicate on a relation. A predicate whose relation is stored in the database D is called an *extensional database* (EDB) relation, while one defined by the rules is called an *intensional database* (IDB) relation. A particular IDB predicate, ans, is designated as a *goal predicate*, and the answers to the query are all facts about the goal predicate that can be deduced from the EDB relations.

For instance, suppose we have a database with one relation $flight$. A tuple $flight(F, T)$ means that there is a nonstop flight from airport F to airport T. The following is a datalog query:

$$\begin{array}{lll} r_1: & reachable(X, Y) & \text{:- } flight(X, Y) \\ r_2: & reachable(X, Y) & \text{:- } flight(X, Z), reachable(Z, Y) \end{array}$$

The query has two rules, r_1 and r_2. It has an EDB predicate $flight$, and an IDB predicate $reachable$. Each tuple $\langle x_0, y_0 \rangle$ in $reachable$ means that airport y_0 is reachable from airport x_0, possibly via several intermediate flights. As shown by the query above, a datalog query can be recursive. A conjunctive query can be viewed as a single-rule datalog query.

2.2 Describing Relation Restrictions

We consider relation capabilities described as *bf* adornment patterns that distinguish bound (*b*) and free (*f*) argument positions [Ull89]. We use *binding patterns* to describe limited access patterns of relations. Each relation is associated with a set of *bf* binding patterns. Each binding pattern represents a form of possible queries that are acceptable by the relation. An attribute adorned *b* requires that a query to this relation using this binding pattern must provide a constant for this attribute. An attribute adorned *f* does not require a constant. For instance, a relation $R(A, B, C)$ with the binding patterns $\{R^{bff}(A, B, C), R^{ffb}(A, B, C)\}$ requires that every query to the relation must either supply a value for the first argument, or supply a value for the third argument.

Note that if an attribute is adorned *f* in a binding pattern, it may still be bound to a value in a query. Consequently some binding patterns are more general than others, in the sense that every query that is acceptable by one pattern is also acceptable by another. For example, binding pattern *bff* is more general than *bbf*. A source query on a relation must satisfy one of the binding patterns supported by the source. Satisfying a pattern means that the query should provide values for a superset of the attributes that are required to be bound.

2.3 Supplementary Relations

Given a sequence of n subgoals $G_1, G_2, \ldots, G_n$ in a conjunctive query, we consider the corresponding sequence of n *supplementary relations* $S_1, \ldots, S_n$ [BR87].[2] The arguments of each supplementary relation G_j correspond to those variables that appear in the first j subgoals, and they are relevant after these j subgoals have been solved. A variable is *relevant* if it appears either in the head or in the $(j+1)$st or a subsequent subgoal. S_1 includes the tuples that satisfy the first subgoal G_1. For $j = 2, \ldots, n$, relation S_j is computed by taking $G_{j-1} \bowtie G_j$, then dropping the irrelevant arguments. The answer to the query is the supplementary relation S_n.

For instance, suppose we have three relations, $r(A_1, A_2, A_3)$, $s(B_1, B_2)$, and $t(C_1, C_2)$. Consider the following query:

$$ans(X, W) \text{ :- } r(1, X, Y), s(Y, Z), t(Z, W)$$

For the sequence of the subgoals as written in the query, the supplementary relations are:

$$\begin{aligned} S_1(X, Y) &\text{ :- } \pi_{A_2, A_3}(\sigma_{A_1=1} r(A_1, A_2, A_3)) \\ S_2(X, Z) &\text{ :- } \pi_{X,Z}(S_1(X, Y) \bowtie s(Y, Z)) \\ S_3(X, W) &\text{ :- } \pi_{X,W}(S_2(X, Z) \bowtie t(Z, W)) \end{aligned}$$

For example, S_2 does not include variable Y, since Y is not used in the later subgoal $t(Z, W)$ or the head. As we will see in Section 3.2, when relations have binding restrictions, there could be different ways to compute supplementary relations.

[2] Here we do not consider the supplementary relation S_0 as defined in [BR87].

2.4 Open-World Assumption and Closed-World Assumption

Given a set of relations, we can use *views* to define more relations. For instance, assume we have two relations $r(Star, Movie)$ and $s(Movie, Studio)$. The following view:

$$v(Star, Movie, Studio) \text{ :- } r(Star, Movie), s(Movie, Studio)$$

defines the relation v that includes all the tuples in the natural join of the two relations. Using views we can hide some data from some users, make certain queries easier or more natural to express, and write queries modularly.

We often use $\mathcal{V} = \{V_1, \ldots, V_n\}$ to denote a set of views. Given a view set $\mathcal{V}$ on predicates $p_1, \ldots, p_m$ and an instance I of $\mathcal{V}$, there are two assumptions about the database D of predicates $p_1, \ldots, p_m$. Under the *closed-world assumption* ("CWA" for short), instance I stores all the tuples that satisfy the view definitions in $\mathcal{V}$, i.e., $I = ANS(\mathcal{V}, D)$. However, under the *open-world assumption* ("OWA" for short), instance I might only store *some* of the tuples that satisfy the view definitions $\mathcal{V}$, i.e., $I \subseteq ANS(\mathcal{V}, D)$. The following example illustrates the difference between the two assumptions.

EXAMPLE 2.4.1 Consider the following two views:

$$\begin{aligned} v_1(M, D) &\text{ :- } car(M, D) \\ v_2(M, D) &\text{ :- } car(M, D) \end{aligned}$$

The two views have the same definition. As illustrated in Figure 2.1(a), under CWA, we are sure that v_1 and v_2 both have *all* the tuples in the *car* relation. Thus these two views must have the same set of tuples. This assumption is true in many cases, e.g., these views and the *car* table exist in the same database.

Under OWA, however, *car* is a global predicate used to formulate the two views. Each of the two views only has *some* car tuples, so it is possible for the two views to have *different* sets of tuples, as shown in Figure 2.1(b). This case could happen when these two views are from two different Web sources that have car information, thus they could have different tuples. □

As we will see later in the thesis, under different assumptions about views, we might have different solutions to the same problem.

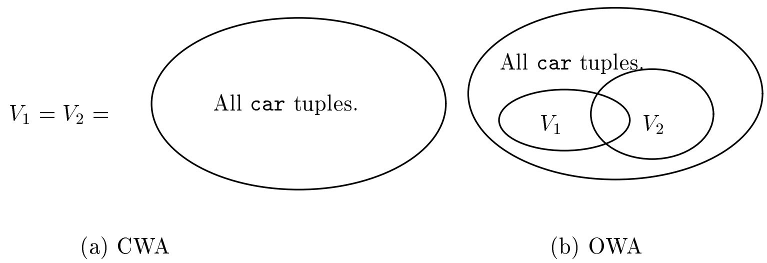


Figure 2.1: Closed-world assumption (CWA) and open-world assumption (OWA).

2.5 Answering Queries Using Views

In many applications, we often have a set of views, and we want to answer a user query using the results of these views. This problem is called "answering queries using views" [LMSS95].

EXAMPLE 2.5.1 Suppose we have two relations $car(Make, Dealer)$ and $loc(Dealer, City)$ that store information about cars, their dealers, and located cities. Assume we have a view:

$$V : v(M, D, C) \text{ :- } car(M, D), loc(D, C)$$

which is a natural join of the two relations. For a following query

$$Q : q(D, C) \text{ :- } car(toyota, D), loc(D, C)$$

that asks for the Toyota dealers and their cities, we can use the following rewriting

$$P : q(D, C) \text{ :- } v(toyota, D, C)$$

to answer the query. □

Now let us define some concepts about answering queries using views.

Definition 2.5.1 (expansion of a query using views) The *expansion* of a query P on a set of views $\mathcal{V}$, denoted by P^{exp}, is obtained from P by replacing all the views in P with their corresponding base relations. Existentially quantified variables in a view are replaced by fresh variables in P^{exp}. □

Definition 2.5.2 (**rewritings and equivalent rewritings**) Given a query Q and a view set $\mathcal{V}$, a query P is a *rewriting* of query Q using $\mathcal{V}$ if P uses only the views in $\mathcal{V}$, and $P^{exp} \sqsubseteq Q$. P is an *equivalent rewriting* of Q using $\mathcal{V}$ if P^{exp} and Q are equivalent, i.e., $P^{exp} \equiv Q$. We say a query Q is *answerable* by $\mathcal{V}$ if there exists an equivalent rewriting of Q using $\mathcal{V}$. □

In Example 2.5.1, P is an equivalent rewriting of the query Q using view V, because the expansion of P:

$$P^{exp} : q(D, C) \text{ :- } car(toyota, D), loc(D, C)$$

is equivalent to Q, i.e., $P^{exp} \equiv Q$.

Several algorithms have been developed for answering queries using views, such as the bucket algorithm [LRO96, GM99a], the inverse-rule algorithm [Qia96, DG97], and the algorithms in [Mit01, PL00]. Most of these algorithms take the open-world assumption. [LMSS95, AD98] study the complexity of answering queries using views. In particular, it has been shown that the problem of rewriting a query using views is $\mathcal{NP}$-hard. In Chapter 7 we will give an efficient algorithm for finding equivalent rewritings of a query using views under the closed-world assumption.

Definition 2.5.3 (**maximally-contained rewritings**) P is a *maximally-contained rewriting* of a query Q using a set of views $\mathcal{V}$ if the following hold: (1) P is a union of conjunctive queries using only the views in $\mathcal{V}$; (2) For any database, the answer computed by P is a subset of the answer to Q; and (3) No other unions of conjunctive queries that satisfy the two conditions above can properly contain P. □

Intuitively, a maximally-contained rewriting ("MCR" for short) is a plan that uses only the views in $\mathcal{V}$ and computes the maximal answer to the query Q. If Q has two MCR's, by definition, they must be equivalent. Clearly if there is an equivalent rewriting P of Q using $\mathcal{V}$, then P is also an MCR of Q.

EXAMPLE 2.5.2 Given two relations $car(Make, Dealer)$ and $loc(Dealer, City)$, consider query Q and view V:

$$\begin{array}{lll} Q: & q(M, D, C) & \text{:- } car(M, D), loc(D, C) \\ V: & v(M, D) & \text{:- } car(M, D), loc(D, sf) \end{array}$$

The following rewriting

$$q(M, D, sf) \text{ :- } v(M, D)$$

is an MCR of the query Q using the view V. That is, we can give the user the information about car dealers in San Francisco (*sf*) as an answer to the query, but not anything more. $\square$

2.6 Technical Problems

After introducing the notation, now we summarize the technical problems discussed in the following chapters.

1. Chapter 3 discusses how to optimize conjunctive queries on relations that have binding patterns. We consider different cost models, and propose efficient algorithms for finding good plans.

2. In Chapter 4 we study how to compute a maximal answer to a conjunctive query on relations with binding patterns. The main idea of the approach is to borrow bindings from other relations not mentioned in a query. We solve optimization problems to minimize the number of relation accesses.

3. In Chapter 5, we study the following problem: given a query on relations with binding patterns, is there a plan that computes the answers to the query by accessing the relations using legal patterns? We study this problem for several classes of queries, including conjunctive queries, conjunctive queries with arithmetic comparisons, unions of conjunctive queries, and datalog queries. We give algorithms and decidability results for these classes.

4. Chapter 6 discusses how to use mediator caching to improve query performance. In particular, we study what query results should be cached at the mediator in order to answer as many future queries as possible. We take the closed-world assumption in this chapter.

5. Chapter 7 studies the following problem: how to use views to generate efficient plans for a conjunctive query? We take the closed-world assumption in this chapter, while previous algorithms on this problem take the open-world assumption.

Chapter 3 to Chapter 6 focus on query-optimization problems in systems that take the source-centric approach to information integration. Chapter 7 studies problems in systems that take the query-centric approach.

Chapter 3

Optimizing Queries with Source Restrictions

In the query-centric approach to information integration, mediators define integrated views based on the data provided by sources. A user query posed on synthesized views at the mediator is translated into source queries and postprocessing operations on the source query results. Often these translated queries need to involve many different source relations. In addition, these sources can have limited capabilities to answer queries.

In this chapter we study the problem of finding feasible and efficient query plans for conjunctive queries when sources have binding patterns. We use a simple cost model that focuses on the major costs in mediation systems, those involved with sending queries to sources and getting answers back. Under this metric, we develop two algorithms for source query sequencing – one based on a simple greedy strategy and another based on a partitioning scheme. The first algorithm produces optimal plans in many scenarios, and we show a linear bound on its worst-case performance when it misses optimal plans. The second algorithm generates optimal plans in more scenarios, while having no bound on the margin by which it misses the optimal plans. We also report on the results of the experiments that study the performance of the two algorithms.[1]

[1]The material in this chapter was developed jointly with Ramana Yerneni.

Chapter Organization

Section 3.1 gives an example to motivate the problem. In Section 3.2, we introduce the cost model and our notation. We present an efficient algorithm named CHAIN in Section 3.3, and prove its n-competitiveness (i.e., bounded optimality). In Section 3.4, we describe our second algorithm, called PARTITION, and discuss its ability to find optimal feasible plans. The results of our performance analysis of the two algorithms are reported in Section 3.5. Finally, we discuss in Section 3.6 how the main results under this cost model can be extended to other cost models. In Section 3.7, we conclude the chapter and discuss related work.

3.1 Introduction

The following example shows that when sources have limited access patterns, the mediator should consider these restrictions to generate feasible and efficient plans.

EXAMPLE 3.1.1 Assume the three movie sources in Example 1.2.1 have the following limited access patterns:

Source Relation	Bind Patterns
`r(Star, Title)`	*bf, fb*
`s(Title, Studio)`	*bf*
`t(Title, Year)`	*bf*

For instance, the binding patterns of relation r say that queries sent to the relation must either provide a star or a movie title. The query Q on the three relations can be rewritten to the following conjunctive form:

$$Q : q(T) \text{ :- } r(reeves, T), s(T, warner), t(T, 1999)$$

The mediator needs to join the results of the three source queries on the `Title` attribute to answer the query. There are many physical plans for this query with various join orders and join methods, while some of these plans are feasible and others are not. For example, Figure 3.1 shows three possible physical plans:

- Plan P_1: Send query `r(reeves, T)` to r; send query `s(T, warner)` to s; and send query `t(T, 1999)` to t. Join the movie results of the three source queries on the `Title` attribute, and return the titles to the user.

- Plan P_2: First get the titles of movies starred by Reeves from source r. For each returned title t_i, send a query to s to get its studio and check if it is "warner." If so, send a query to t to get the year of movie t_i. If the year is 1999, return t_i to the user.

- Plan P_3: This plan is similar to P_2, except that we reverse the s and t queries. That is, after getting Reeves movie titles from relation r, for each title we get its year from t. If the year is 1999, we send a query to s to get its studio. If the studio is "warner," we return this title to the user.

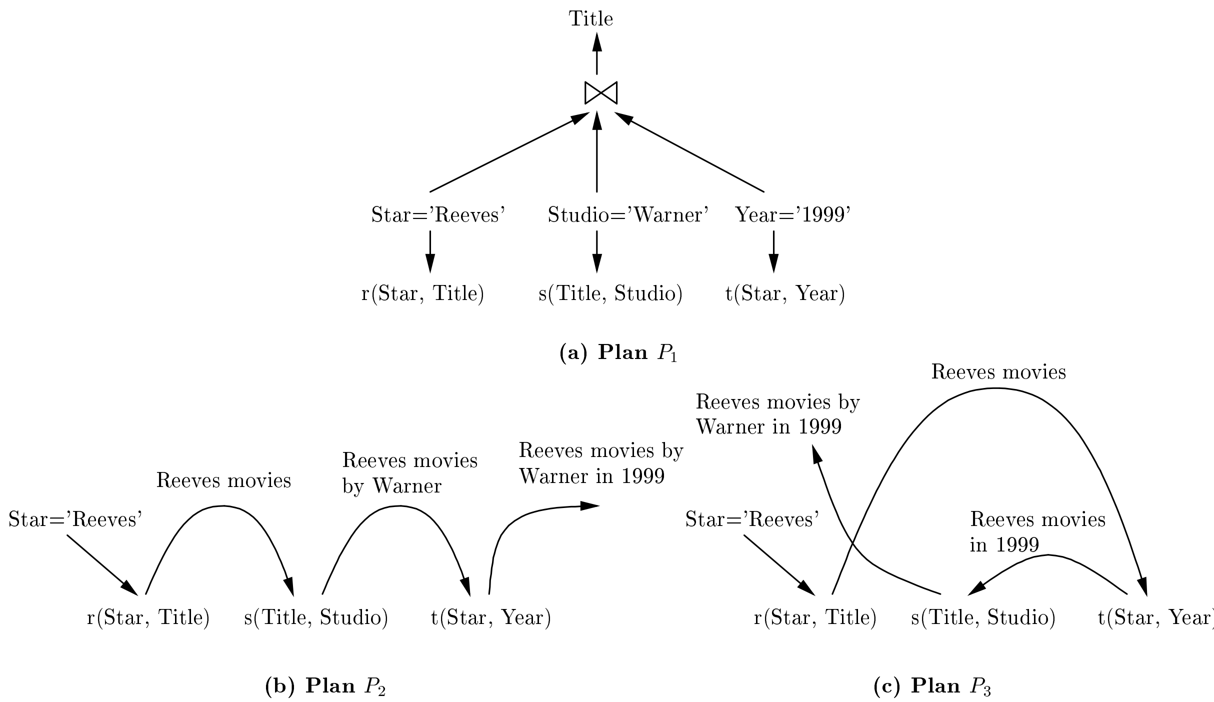


Figure 3.1: Three physical plans to answer a query.

Given the binding patterns of the relations, plan P_1 is not feasible, because the queries to sources s and t do not provide a necessary binding for the `Title` attribute. The other two plans are feasible, since all their queries to the sources provide the necessary bindings. If the studio value "warner" is more selective than the year value "1999," then plan P_2 may need fewer source queries than P_3. If we want to minimize the execution time, we could even send queries to relations s and t in parallel after getting Reeves movies from relation r. □

In general, after a user poses a query over integrated views provided by the mediator, the query is translated into queries over the source views to arrive at *logical query plans*. The logical plans deal only with the content descriptions of the sources. That is, they tell the mediator which sources provide the relevant data and what postprocessing operations need to be performed on the data. The logical plans are later translated into *physical plans* that specify details such as the order in which the sources are contacted and the exact queries to be sent. Our goal in this chapter is to develop algorithms that will translate a mediator logical plan into an efficient and feasible (does not exceed the source capabilities) physical plan. As illustrated by Example 3.1.1, we need to solve the following problems:

1. Given a logical plan and the description of the source capabilities, find feasible physical plans for the logical plan. The central problem is to determine the evaluation order for the subgoals in the logical plan, so that attributes are appropriately bound.

2. Among all the feasible physical plans, pick a most efficient one.

In this chapter, we focus on logical plans that are conjunctive queries. We first consider a simple cost model that focuses on the major costs in mediator query processing: the number of source accesses. Under this cost model, we develop two algorithms that can find good feasible plans rapidly. The first algorithm, called "CHAIN," runs in $O(n^2)$ time, where n is the number of subgoals in the logical plan. We also provide a linear bound on the margin by which this algorithm can miss the optimal plans. Our second algorithm, called "PARTITION," can guarantee optimal plans in more scenarios than CHAIN, although there is no bounded optimality for its plans. Both our algorithms are guaranteed to find a feasible plan, if the query has a feasible plan. Furthermore, we show through experiments that our algorithms have excellent running-time profiles in a variety of scenarios, and very often find optimal or near-optimal plans. We also discuss how to extend the results under this simple cost model to more general cost models, e.g., those that consider different costs of different sources and the size of data transferred on the network.

3.2 Preliminaries

In this section we formalize the problem of optimizing conjunctive queries when sources have limited access patterns. We also discuss the simple cost model used in our optimization algorithms.

User queries to a mediator are conjunctive queries on integrated views provided by the mediator. Each integrated view is defined as a conjunctive query over the source relations. The user query is translated into a logical plan, which is a conjunctive query on the source relations. For instance, query Q_0 in Example 1.2.1 over the `Movie` view was translated into the conjunctive query Q over the source relations.

Given a sequence of n subgoals $G_1, G_2, \ldots, G_n$ in a conjunctive query Q, we consider the corresponding sequence of n supplementary relations $S_1, \ldots, S_n$. Because the relations have restrictions, there are two ways to compute S_j by evaluating $S_{j-1} \bowtie G_j$:

1. Use the bound arguments in G_j to send a query to the source of G_j by binding a subset of its attributes; perform the join of the result of this source query with S_{j-1} at the mediator, and project out irrelevant arguments to obtain S_j.

2. For $j \geq 2$, use the tuples in S_{j-1} to send a set of queries to the source relation of G_j by binding a subset of its attributes; take the union of the results of these source queries; perform the join of this union relation with S_{j-1} and project out irrelevant arguments to obtain S_j.

We call the first kind of source query a *block query* and the second kind a *parameterized query*. Obviously, answering G_j through the first method takes a single source query, while the second method can take many source queries. The main reason why we need to consider parameterized queries is that it may not be possible to answer some subgoals in the logical plan through block queries. That is, the access templates for the corresponding source relations may require bindings of variables that are not available in the logical plan. In order to answer these subgoals, we must use parameterized queries by executing other subgoals first, and collecting bindings for the required parameters of G_j.

3.2.1 The Plan Space

A plan for a query (logical plan) is *feasible* if all its source queries are answerable by the sources. We consider the space of *linear plans*, each of which follows a *feasible sequence* of all the subgoals in the query.

Definition 3.2.1 (feasible sequence of subgoals) Some subgoals $g_1(\bar{X}_1), \ldots, g_k(\bar{X}_k)$ in a query form a *feasible sequence* if for each subgoal $g_i(\bar{X}_i)$ in the sequence, given the variables that are bound by the previous subgoals, subgoal $g_i(\bar{X}_i)$ is *answerable*; that is, there is a binding pattern p_{ij} of the relation g_i, such that for each argument X in subgoal $g_i(\bar{X}_i)$ that is adorned as b in p_{ij}, either X is a constant, or X appears in a previous subgoal. A query is *feasible* if it has a feasible sequence of *all* its subgoals. □

The feasibility of a query can be checked by a greedy algorithm, called *Inflationary*. That is, initialize a set Φ_a of answerable subgoals to be empty. With the variables bound by the subgoals in Φ_a, whenever a subgoal becomes answerable by its relation, add this subgoal to Φ_a. Iterate by adding more subgoals to Φ_a. The query is feasible if and only if all the subgoals in the query become answerable.

For a feasible sequence of all the subgoals, we decide on the choice of queries for each subgoal (among the set of block queries and parameterized queries available for the subgoal). Note that the number of feasible physical plans for a given logical plan can be exponential in the number of subgoals in the logical plan.

The space of plans we consider is similar to the space of left-deep-tree executions of a join query [SAC+79]. We assume that a bushy-tree execution only provides bindings from the left. As stated in the following lemma, we do not miss feasible plans by not considering bushy-tree executions.

Lemma 3.2.1 *We do not miss feasible plans by considering only left-deep-tree executions of the subgoals.* □

Proof: Given any feasible execution of the logical plan based on a bushy tree of subgoals, we construct another feasible execution based on a left-deep tree of subgoals. The constructed left-deep tree will have the same leaf order as the given bushy tree.

Consider a feasible plan P based on the bushy tree of subgoals. We will derive a feasible plan P' from the left-deep tree constructed above. If a subgoal is answered by a block query in P, it is also answered by a block query in P'. If it is answered by parameterized queries in P, it can also be answered by parameterized queries in P', because the left-deep-tree execution keeps the cumulative bindings of all the variables of all the subgoals to the left of the subgoal under consideration. This observation is similar to the *bound-is-easier* assumption in [UV88]. Thus, we conclude that if a bushy tree of subgoals has a feasible plan, we are guaranteed to find a feasible plan by considering only left-deep-tree executions.

■

3.2.2 The Formal Cost Model

In many applications, the cost of query processing in mediation systems is dominated by the cost of interacting with the sources. Hence, we focus on the costs associated with issuing queries to sources. Our results are first stated using a very simple cost model where we count the total number of source queries in a plan. Our cost model is defined as follows:

1. The cost of a subgoal in a feasible plan is the number of source queries needed to answer the subgoal.

2. The cost of a feasible plan is the sum of the costs of all the subgoals in the plan.

We illustrate our cost model by the following example. Consider plan P_1 in Example 3.1.1. Since we need to send only one query to r to answer the subgoal `r(reeves, Title)`, the cost of this subgoal is 1. Suppose there are 5 movies in which `reeves` starred. For each movie title $t0$, we send relation s a parameterized query `s(t0, Year)` to get its year information. Thus, the cost of the second subgoal is 5. If 2 of these 5 movies were produced by Warner Brothers, we need to send 2 parameterized queries to relation t. Thus the cost of the third subgoal is 2, and the total cost of the plan $Cost(P_1) = 1 + 5 + 2 = 8$.

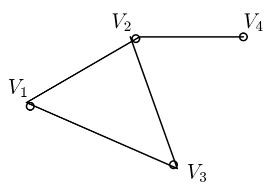
In this chapter we first develop the main results under this simple cost model. In Section 3.6, we will show how to extend these results to more complex cost models. In spite of the simplicity of the cost model, the optimization problem we are dealing with is quite hard.

Theorem 3.2.1 *The problem of finding a feasible plan with the minimum number of source queries is $\mathcal{NP}$-hard.* □

Proof: We reduce the vertex-cover problem [GJ79] to our problem. The vertex-cover problem is to find a vertex cover of minimum size in a given graph. Since this problem is $\mathcal{NP}$-complete, our problem is $\mathcal{NP}$-hard.

Given a graph G with n vertices $V_1, \ldots, V_n$, we construct a database and a logical plan as follows. Corresponding to each vertex V_i we define a relation R_i. For all $1 \leq i \leq j \leq n$, if V_i and V_j are connected by an edge in G, then R_i and R_j include the attribute A_{ij}. In addition, we define a special attribute X and two special relations R_0 and R_{n+1}. Thus we have a total of $m + 1$ attributes, where m is the number of edges in G. The special attribute X is in the schema of all the relations. The special relation R_{n+1} also has all the A_{ij} attributes. That is, R_0 has only one attribute, and R_{n+1} has $m+1$ attributes. Each relation has a tuple with a value of 1 for each attribute. In addition, all relations except R_{n+1} include a second tuple with a value of 2 for all their attributes. Each relation has a single access template: R_0 has no binding requirements, R_1 through R_n require the attribute X to be bound, and R_{n+1} requires all of the attributes to be bound. Finally, the logical plan consists of all the $n+2$ relations, with no variables bound.

For instance, Figure 3.2(a) shows a graph, and Figure 3.2(b) shows the corresponding query, relation restrictions, and the database.



(a)

$$ans(X, A_{12}, A_{23}, A_{13}, A_{24}) :- R_0(X), R_1(X, A_{12}, A_{13}), R_2(X, A_{12}, A_{23}, A_{24}),$$
$$R_3(X, A_{13}, A_{23}), R_4(X, A_{24}), R_5(X, A_{12}, A_{13}, A_{23}, A_{24})$$

Restrictions and database:

R_0^{b}	R_1^{bff}	R_2^{bfff}	R_3^{bff}	R_4^{bf}	R_5^{bbbbb}
1	1 1 1	1 1 1 1	1 1 1	1 1	1 1 1 1 1
2	2 2 2	2 2 2 2	2 2 2	2 2	

(b)

Figure 3.2: Constructing a query and database from a graph.

It is obvious that the above construction of the database and the logical plan takes time that is polynomial in the size of G. Now, we show that G has a vertex cover of size k if and only if the logical plan has a feasible physical plan that requires $(n+k+3)$ source queries.

Suppose G has a vertex cover of size k. Without loss of generality, let it be $\{V_1, \ldots, V_k\}$. Consider the physical plan P that first answers the subgoal R_0 with a block query, then answers $R_1, \ldots, R_k, R_{n+1}, R_{k+1}, \ldots, R_n$ using parameterized queries. P is a feasible plan because R_0 has no binding requirements, $R_1, \ldots, R_k$ need X to be bound and X is available from R_0, and $R_1, \ldots, R_k$ will bind all the variables (since $\{V_1, \ldots, V_k\}$ is a vertex cover). Subgoal R_0 in P is answered by a single source query, $R_1, \ldots, R_k$ and R_{n+1} are answered by two source queries each, and $R_{k+1}, \ldots, R_n$ are answered by one source query each. Thus the number of source queries for this plan is $n+k+3$. We see that if G has a vertex cover of size k, we have a feasible plan with $(n+k+3)$ source queries.

Suppose there is a feasible plan P' with f source queries. The first subgoal in P' must be R_0, and it must be answered by a block query (because the logical plan does not bind any variables). All the other subgoals must be answered by parameterized queries. Consider the set of subgoals in P' that are answered before R_{n+1} is answered. Let j be the size of this set of subgoals (excluding R_0). Since R_{n+1} needs all attributes to be bound, the union of the schemas of these j subgoals must be the entire attribute set. That is, the vertices corresponding to these j subgoals form a vertex cover in G. In P', each of these j subgoals takes two source queries, along with R_{n+1}, while the rest of $(n-j)$ subgoals in $R_1, \ldots, R_n$ take one source query each. That is, $f = 1 + 2 * j + 2 + (n - j)$. Thus we can find a vertex cover for G of size $f - n - 3$.

Hence, G has a vertex cover of size k if and only if there is a feasible plan with $(n+k+3)$ source queries. That is, we have reduced the problem of finding the minimum vertex cover in a graph to our problem of finding a feasible plan with minimum source queries. ∎

Recall that the space of plans we consider does not include bushy-tree executions of the subgoals. It turns out that under this cost model, we can safely restrict our attention to plans based on left-deep-tree executions of the subgoals without missing an optimal plan.

Theorem 3.2.2 *We do not miss an optimal plan by not considering the executions of the logical plan based on bushy trees of subgoals.* □

Proof: The proof is similar to that of Lemma 3.2.1. Once again, given any physical plan based on a bushy tree of subgoals, we can construct an physical plan based on a left-deep tree of subgoals (with the same leaf order) that is at least as good.

If a subgoal is answered by a block query in the bushy-tree plan, it will also be answered by a block query in the left-deep-tree plan. This subgoal will have the same cost (of 1) in both plans. If a subgoal is answered by parameterized queries in the bushy-tree plan, it is also answered by parameterized queries in the left-deep-tree plan. Note that by using cumulative supplementary relations in the left-deep-tree plan, we can only reduce the number of distinct values for the parameters (to the queries) of the subgoal. Thus, the cost of the subgoal in the left-deep-tree plan can be at most equal to that in the bushy-tree plan. The plan based on the constructed left-deep tree is at least as cheap as the plan based on the bushy tree. Hence, by considering only left-deep trees of subgoals, we do not miss an optimal plan. ∎

3.2.3 Subgoal Sequences and Physical Plans

We extend the notion of cost and feasibility of physical plans to sequences of subgoals in the logical plan. For each sequence of subgoals, we associate a set of physical plans with it by making the choices of source queries (block queries and parameterized queries) for the subgoals. The cost of a given sequence of subgoals is the cost of the cheapest physical plan associated with the sequence. The cost of a subgoal in a sequence is the cost of the subgoal in the best physical plan for the sequence. Sequences of subgoals satisfy some interesting properties stated by the following lemmas.

Lemma 3.2.2 *Given a sequence of subgoals, one can ascertain its feasibility in $O(n)$ time, where n is the number of subgoals.* □

Lemma 3.2.3 *Given a sequence of subgoals, one can find its cost in $O(n)$ time, where n is the number of subgoals.*[2] □

Lemma 3.2.4 *Given a sequence of subgoals, one can find a best physical plan for the sequence in $O(n)$ time, where n is the number of subgoals.* □

Lemma 3.2.5 *Postponing the processing of an answerable subgoal in a sequence can not make it unanswerable.* □

Lemma 3.2.6 *Postponing the processing of an answerable subgoal in a sequence can not increase its cost.* □

3.2.4 Problem Statement

The problem we are addressing in this chapter is how to find efficient, feasible physical plans for given logical plans. As noted above, given a sequence of subgoals, one can easily compute a best physical plan for that sequence. Because it is easy to go from a sequence of subgoals to its best plan, we sometimes refer to our problem as finding the best sequence of subgoals. In particular, the algorithms of Section 3.3 and Section 3.4 actually find the best sequence of subgoals and then translate it into a best physical plan.

[2] *We assume that finding the cost of a subgoal following a partial sequence takes $O(1)$ time.*

3.3 The CHAIN Algorithm

In this section, we present an algorithm, called "CHAIN," for finding an optimal feasible query plan. This algorithm uses a greedy strategy to build a single sequence of subgoals that is feasible and efficient. We first describe the algorithm formally, then analyze its complexity and its ability to generate efficient, feasible plans.

As shown in Figure 3.3, CHAIN finds a physical plan as follows. Initially it finds all subgoals that are answerable with the initial bindings in the logical plan, and picks the one with the least cost. It computes the additional variables that are now bound due to the chosen subgoal. It repeats the process of finding answerable subgoals, picking the cheapest among them, and updating the set of bound variables, until no more subgoals are left or some subgoals are left, but none of them is answerable. If there are subgoals left over, CHAIN declares that there is no feasible plan. Otherwise it outputs its constructed plan.

3.3.1 Feasible-Plan Generation

Lemma 3.3.1 *If a logical plan has feasible physical plans, CHAIN will not fail to generate a feasible plan.* □

Proof: If CHAIN fails to generate a feasible plan, there are some left-over subgoals in the logical plan for which the initial bindings along with the variables of the other subgoals are not sufficient. This case can only be possible if there is no feasible physical plan for the logical plan. Otherwise, consider the first subgoal in a feasible physical plan that is one of the left-over subgoals in CHAIN. Since all the subgoals preceding this subgoal in the feasible plan are accumulated in the sequence built by CHAIN (before it gave up), this subgoal will also be deemed answerable by CHAIN. Thus it cannot be one of the left-over subgoals. Hence, it is not possible for the feasible physical plan to exist. ∎

Lemma 3.3.2 *CHAIN runs in $O(n^2)$ time where n is the number of subgoals in the logical plan.* □

Proof: There can be at most n iterations in CHAIN, and in each iteration it adds a subgoal to the constructed plan. Each iteration takes $O(n)$ time as it examines at most n subgoals to find the next cheapest answerable subgoal. Thus, CHAIN takes a total of $O(n^2)$ time. ∎

The CHAIN Algorithm
Input: A logical plan (conjunctive query) on relations with binding patterns.
Output: A feasible physical plan.

- Initialize:
  - $S \leftarrow \{C_1, C_2, \ldots, C_n\}$ /*set of subgoals in the logical plan*/
  - $B \leftarrow$ set of bound arguments in the logical plan
  - $L \leftarrow \phi$ /* start with an empty sequence */
- Construct the sequence of subgoals:

```
while (S ≠ φ ) do
  M ← infinity;
  N ← null;
  for each subgoal C_i in S do    /* find the cheapest subgoal */
    if (C_i is answerable with B) then
      c ← Cost_L(C_i);        /* get the cost of this subgoal in sequence L */
      if ( c < M ) then
        M ← c;
        N ← C_i;
  if (N = null) /* If no next answerable subgoal, declare no feasible plan*/
    return(φ);
  L ← L + N; /* Add next subgoal to plan */
  S ← S − {N};
  B ← B ∪ { arguments of  N};
```

- Return the feasible physical plan:
  - construct $Plan(L)$ from sequence L;
  - return $Plan(L)$;

Figure 3.3: Algorithm CHAIN.

3.3.2 Optimality of Plans Generated by CHAIN

Lemma 3.3.3 *If the result of the query is nonempty, and the number of subgoals in the logical plan is less than 3, CHAIN is guaranteed to find the optimal plan.* □

Proof: If there is only one subgoal, CHAIN will obviously find the optimal plan consisting of the only subgoal. If there are two subgoals, we have two cases: (i) both can be executed using block queries, (ii) only one of them can be executed using a block query, and the other one needs parameterized queries. In the first case, the cost of the plan chosen by CHAIN is 2, which is the cost of the optimal plan. In the second case, there is only one feasible sequence of subgoals. Thus CHAIN will end up with the optimal plan. ■

Lemma 3.3.4 *CHAIN can miss the optimal plan if the number of subgoals in the logical plan is greater than 2.* □

$r^{bff}(A, B, D)$	$s^{bf}(B, E)$	$t^{bf}(D, F)$
(1, 1, 1)	(1, 1)	(4, 1)
(1, 2, 2)	(2, 1)	(5, 1)
(1, 3, 3)	(3, 1)	(6, 1)
(1, 1, 4)	(4, 1)	(7, 1)

Table 3.1: Proof of Lemma 3.3.4: three data sources.

Proof: We construct a logical plan with three subgoals and a database instance that result in CHAIN generating a suboptimal plan. Consider a logical plan

$$ans(F) \text{ :- } r(1, B, D), s(B, E), t(D, F)$$

and the database instance shown in Table 3.1. For this logical plan and database, CHAIN will generate the plan: $r \rightarrow s \rightarrow t$, with a total cost of $1 + 3 + 4 = 8$. We observe that a cheaper feasible plan is: $r \rightarrow t \rightarrow s$, with a total cost of $1 + 4 + 1 = 6$. ■

It is not difficult to find situations in which CHAIN misses the optimal plans. However, surprisingly, there is a linear upper bound on how far its plan can be from the optimal plans. In fact, we prove a stronger result in Lemma 3.3.5.

Lemma 3.3.5 *Suppose P^c is the plan generated by CHAIN for a logical plan with n subgoals; P^o is an optimal plan, and E_{max} is the cost of the most expensive subgoal in P^o. Then,*

$$Cost(P^c) \leq n \times E_{max}$$

□

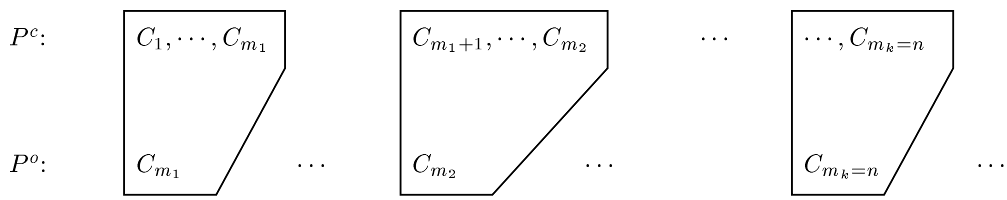


Figure 3.4: Proof for Lemma 3.3.5.

Proof: Suppose the sequence of subgoals in P^c is $C_1, C_2, \ldots, C_n$. As shown in Figure 3.4, let the first subgoal in P^o be C_{m_1}. Let G_1 be the prefix of P^c, such that $G_1 = C_1, \ldots, C_{m_1}$. When CHAIN chooses C_1, the subgoal C_{m_1} is also answerable. Thus the cost of C_1 in P^c is less than or equal to the cost of C_{m_1} in P^o. After processing C_1 in P^c, the subgoal C_{m_1} remains answerable (see Lemma 3.2.5) and its cost cannot increase (see Lemma 3.2.6). Thus if CHAIN has chosen another subgoal C_2 instead of C_{m_1}, once again we can conclude that the cost of C_2 in P^c is not greater than the cost of C_{m_1} in P^o. Finally, at the end of G_1, when C_{m_1} is processed in P^c, we note that the cost of C_{m_1} in P^c is no more than the cost of C_{m_1} in P^o. Therefore, the cost of each subgoal of G_1 is less than or equal to the cost of C_{m_1} in P^o.

We call C_{m_1} the first *pivot* in P^o. We define the next pivot C_{m_2} in P^o as follows. C_{m_2} is the first subgoal after C_{m_1} in P^o such that C_{m_2} is not in G_1. Now, we can define the next subsequence G_2 of P^c such that the last subgoal of G_2 is C_{m_2}. The cost of each subgoal in G_2 is less than or equal to the cost of C_{m_2}.

We continue finding the rest of the pivots $C_{m_3}, \ldots, C_{m_k}$ in P^o and the corresponding subsequences $G_3, \ldots, G_k$ in P^c. Based on the above argument, we have:

$$\forall C_i \in G_j : (\text{cost of } C_i \text{ in } P^c) \leq (\text{cost of } C_{m_j} \text{ in } P^o)$$

Therefore

$$Cost(P^c) = \sum_{j=1}^{k} \sum_{C_i \in G_j} (\text{cost of } C_i \text{ in } P^c) \leq \sum_{j=1}^{k} |G_j| \ \times \ (\text{cost of } C_{m_j} \text{ in } P^o) \leq n \times E_{max}$$

where E_{max} is the cost of the most expensive subgoal in P^o. ∎

From Lemma 3.3.5, by observing that $E_{max} \leq$ cost of P^o, we have the following theorem.

Theorem 3.3.1 *CHAIN is n-competitive. That is, the plan generated by CHAIN can be at most n times as expensive as the optimal plans, where n is the number of subgoals.* □

The cost of the plan generated by CHAIN can be arbitrarily close to the cost of the optimal plan multiplied by the number of subgoals. In this sense, the linear bound on optimality for CHAIN is tight. However, in many situations CHAIN yields optimal plans or plans whose cost is very close to that of the optimal plan. In Section 3.5, we study the quality of plans generated by CHAIN in a wide range of scenarios.

3.4 The PARTITION Algorithm

We present another algorithm called PARTITION for finding efficient, feasible plans. PARTITION takes a very different approach to the plan-generation problem. It is guaranteed to generate optimal plans in more scenarios than CHAIN, but has a worse running time. We first formally present the PARTITION algorithm and discuss its ability to generate plans. We then describe two variations of PARTITION that specifically target the generation of optimal plans and the efficiency of the plan-generation process, respectively.

3.4.1 PARTITION

PARTITION organizes the subgoals into *clusters* based on the capabilities of the sources. Then it performs local optimization within each cluster, and builds the feasible plan by merging the subplans from all the clusters.

As shown in Figure 3.5, PARTITION has two phases. The first phase organizes the set of subgoals in the logical plan into a list of clusters. The property satisfied by the clusters is as follows. All the subgoals in the first cluster are answerable by block queries; all the subgoals in each subsequent cluster are answerable by parameterized queries that use attribute bindings from the subgoals of the earlier clusters. In the second phase, PARTITION finds

The PARTITION Algorithm
Input: A logical plan (conjunctive query) on relations with binding patterns.
Output: A feasible physical plan.

- Initialize:
  $S \leftarrow \{C_1, C_2, \ldots, C_n\}$ /* set of subgoals in the logical plan */
  $B \leftarrow$ set of bound arguments in the logical plan
  $\Psi \leftarrow \phi$ /* start with the empty list of clusters */
- Phase 1: Construct the list of clusters.
  while ($S \neq \phi$) do
   $\Gamma \leftarrow null$;
   for each subgoal C_i in S do
    if (C_i is answerable with B) then
     $\Gamma \leftarrow \Gamma \cup \{C_i\}$;
     $S \leftarrow S - \{C_i\}$
   if ($\Gamma = null$) then /* If no next cluster, declare no feasible plan */
    return(ϕ);
   $\Psi \leftarrow \Psi + \Gamma$; /* Add new cluster to the list of clusters */
   $B \leftarrow B \cup \{$ arguments in subgoals of $\Gamma\}$;
- Phase 2: Construct the sequence of subgoals.
  $L \leftarrow \phi$ /* start with an empty sequence */
  for each cluster Γ in Ψ do
   $L' \leftarrow$ the best subsequence of subgoals in Γ;
   $L \leftarrow L \mid\mid L'$;
- Return the feasible plan:
  return($Plan(L)$); /* construct plan from sequence L */

Figure 3.5: Algorithm PARTITION.

the best subplan for each cluster of subgoals. It then combines all these subplans to arrive at the best feasible plan for the user query.

Let us describe the algorithm in more detail. Let Ψ and S denote the list of clusters and the set of subgoals respectively. Initially, Ψ is empty and S contains all the subgoals in the logical plan. S will become smaller as more subgoals are removed from it. Let Γ denote the new cluster that would be generated in each round by adding more feasible subgoals. Let B denote the cumulative set of bound variables in the process of finding clusters. Initially, B contains the set of variables that are bound in the logical plan. Using B, PARTITION finds the set of subgoals that are answerable and collects them into Γ. When all the subgoals that are answerable at this stage have been added to Γ, this cluster is added to Ψ, and the bound variables of these subgoals are added to B. Given the new B, some new subgoals will become feasible in the second round, and they are put into the second cluster. This process is repeated until one of two cases happens:

1. No subgoals are left (S is empty), and we end up with a complete list of clusters.

2. Some subgoals are left in S, and no more new clusters can be formed.

In case 2, PARTITION declares that there is no feasible sequence for this logical plan. In case 1, PARTITION will perform its second phase to generate the best feasible plan. For each cluster of subgoals, the algorithm performs local optimization by trying all the possible permutations of these subgoals. Then it will combine all the local subplans and create the global plan for the query.

EXAMPLE 3.4.1 Consider a logical plan with four subgoals:

$$ans(B) \text{ :- } r(A,B),\ s(A,D),\ t(B,E),\ u(D,B)$$

No argument in the query has been bound. Suppose we have the following relation restrictions:

$$r^{ff}(A,B), s^{bf}(A,D), t^{bf}(B,E), u^{bf}(D,B)$$

Since each subgoal has a different predicate, we refer to the subgoals by the predicate names. Let us consider the first phase of PARTITION. Initially, only the subgoal r is answerable.[3] Therefore, the first cluster contains only this subgoal r. After r is answered, variables A and

[3] For simplicity, we use the corresponding relation name to represent a subgoal.

B are bound. Then subgoals s and t will both become feasible, and they are put into the second cluster. After s and t are answered, variables D and E are also bound, and subgoal u becomes feasible. It is the only subgoal in the third cluster. So the clusters generated by PARTITION are: $\{r\}$, $\{s, t\}$, and $\{u\}$.

In the second phase, PARTITION considers the two feasible subsequences in Γ_2, and picks the one with the lower cost, say $s \rightarrow t$. The plan output by PARTITION would be $r \rightarrow s \rightarrow t \rightarrow u$. Essentially, PARTITION considers two possible sequences: $r \rightarrow s \rightarrow t \rightarrow u$ and $r \rightarrow t \rightarrow s \rightarrow u$. □

3.4.2 Feasible-Plan Generation

Lemma 3.4.1 *If feasible physical plans exist for a given logical plan, PARTITION is guaranteed to find a feasible plan.* □

Proof: If PARTITION fails to find a feasible plan, there must be some subgoals in the logical plan that are not answerable with respect to the set of variables that are initially bound or that occur in the other subgoals. If there is a feasible physical plan, it is not possible to have such unanswerable subgoals in the logical plan. Hence, PARTITION will not fail to find a feasible plan when feasible plans exist. ■

Lemma 3.4.2 *If the number of clusters generated is less than 3, and the result of the query is not empty, then PARTITION will find the optimal plan.* □

Proof: We proceed by a simple case analysis. There are two cases to consider. The first case is when there is only one cluster Γ_1. PARTITION finds the best sequence among all the permutations of the subgoals in Γ_1. The second case is when there are two clusters Γ_1 and Γ_2. Let P be an optimal feasible plan. We will show how we can transform P into a plan in the plan space of PARTITION that is at least as good as P. Let C_i be a subgoal in Γ_1. There are two possibilities:

1. C_i is answered in P using a block query;

2. C_i is answered in P using parameterized queries.

If C_i is answered by a block query, we make no change to P. Otherwise, we modify P as follows. As the result of the query is not empty, the cost of subgoal C_i (using parameterized queries) in P must be at least 1. Since C_i is in the first cluster, it can be answered by using a block query. So we can modify P by replacing the parameterized queries for C_i with the block query for C_i. Since the cost of a block query can be at most 1, this modification cannot increase the cost of the plan. For all subgoals in Γ_1, we repeat the above transformation until we get a plan P', in which all the subgoals in Γ_1 are answered by using block queries.

We apply a second transformation to P' with respect to the subgoals in Γ_1. Since all these subgoals are answered by block queries in P', we can move them to the beginning of P' to arrive at a new plan P''. Moving these subgoals ahead of the other subgoals will preserve the feasibility of the plan (see Lemma 3.2.5). It is also true that this transformation cannot increase the cost of the plan, because it does not change the cost of these subgoals, and it cannot increase the cost of the other subgoals in the sequence (see Lemma 3.2.6). Hence, P'' cannot be more expensive than P'.

After the two-step transformation, we get a plan P'' that is as good as the optimal plan. Note that P'' is in the plan space of PARTITION. So the plan generated by PARTITION cannot be worse than P''. Thus, the plan found by PARTITION must be as good as the optimal plan. ∎

Lemma 3.4.3 *If the number of subgoals in the logical plan does not exceed 3, and the result of the query is not empty, then PARTITION will always find the optimal plan.* □

Proof: If the number of subgoals in the logical plan does not exceed 3, the number of clusters generated is at most 3. In Lemma 3.4.2, we proved that if the number of clusters is 1 or 2, PARTITION finds an optimal plan. Now, we show that in the case where there are three clusters with 1 subgoal each, PARTITION will find the optimal plan. Let the clusters generated be $\Gamma_1 = \{C_1\}$, $\Gamma_2 = \{C_2\}$, $\Gamma_3 = \{C_3\}$. Then the only feasible sequence of subgoals is (C_1, C_2, C_3), which is what PARTITION will output. ∎

It is not true that PARTITION can generate an optimal plan in all cases. One can construct logical plans with as few as 4 subgoals that lead the algorithm to generate sub-optimal plans. We also note that PARTITION can miss the optimal plan by a margin that is unbounded by the query parameters.

Lemma 3.4.4 *For any $k > 0$, there exists a logical plan and a database for which PARTITION generates a plan that is at least k times as expensive as the optimal plan.* □

Proof: Consider Example 3.4.1. Suppose the tables of the four sources are as shown in Table 3.2. PARTITION essentially considers two plans: $r \rightarrow s \rightarrow t \rightarrow u$, $r \rightarrow t \rightarrow s \rightarrow u$. In the first sequence: $cost(r) = 1$, $cost(s) = 1$, $cost(t) = 10000$, and $cost(u) = 1$. So the cost of the first plan is 10003. For the second sequence: $cost(r) = 1$, $cost(t) = 10000$, $cost(s) = 1$, and $cost(u) = 1$. So the cost of the second plan is 10003. PARTITION picks one of these two plans as the final physical plan with a cost of 10003.

Notice that after subgoal s has been answered, subgoal u becomes feasible. By answering u before t, all the tuples whose B value is not 2 will be filtered, so they do not need to be parameterized to answer subgoal T. Then we can define the following plan: $r \rightarrow s \rightarrow u \rightarrow t$, in which $cost(r) = 1$, $cost(s) = 1$, $cost(u) = 1$, and $cost(t) = 1$, for a total cost of 4. But PARTITION misses this plan. Thus, the ratio of the cost of the PARTITION plan to the optimal cost is at least 10003/4. We can make this ratio arbitrarily large by having the appropriate number of tuples in r. Thus, PARTITION can generate plans that are k times as expensive as optimal plans, for any $k > 0$. ∎

$r^{ff}(A,B)$	$s^{bf}(A,D)$	$t^{bf}(B,E)$	$u^{bf}(D,B)$
(1, 1)	(1, 1)	(1, 1)	(1, 2)
(1, 2)			
(1, 3)			
...			
(1, 10000)			

Table 3.2: Proof of Lemma 3.4.4: four data sources.

Lemma 3.4.5 *The PARTITION algorithm runs in $O(n^2+(k_1!+k_2!+\ldots+k_p!))$, where n is the number of subgoals in the logical plan, p is the number of clusters found by PARTITION, and k_i is the number of subgoals in the i^{th} cluster.*[4] □

3.4.3 Variations of PARTITION

We have seen that the PARTITION algorithm can miss the optimal plan in many scenarios, and in the worst case it has a running time that is exponential in the number of subgoals in the logical plan. In a way, it attempts to strike a balance between running time and the ability to find optimal plans. A naive algorithm that enumerates all sequences of

[4] *If the query result in nonempty, PARTITION can consider just one sequence (instead of $k_1!$) for the first cluster.*

subgoals will always find the optimal plan, but it may take much longer than PARTITION. PARTITION tries to cut down on the running time, and gives up the ability to find optimal plans to a certain extent. Here, we consider two variations of PARTITION that highlight this trade-off.

We call the first variation FILTER, which is based on the observation of Lemma 3.4.2. FILTER is guaranteed to find the optimal plan (as long as the query result is nonempty), but its running time is much worse than PARTITION. Yet, it is more efficient than the naive algorithm that enumerates all plans.

FILTER also has two phases like PARTITION. In its first phase, it mimics PARTITION to arrive at the clusters $\Gamma_1, \Gamma_2, \ldots, \Gamma_p$. At the end of the first phase, it keeps the first cluster as is, and collapses all the other clusters into a new second cluster Γ'. That is, it ends up with Γ_1 and Γ'. The second phase of FILTER is identical to that of PARTITION.

Lemma 3.4.6 *If the user query has nonempty result, FILTER will generate the optimal plan.* □

Proof: We can prove this lemma in the same way we proved Lemma 3.4.2. ■

Lemma 3.4.7 *The running time of FILTER is* $O(n^2 + (k_1! + (n - k_1)!))$. □

The second variation of PARTITION is called SCAN. This variation focuses on efficient plan generation. The main idea here is to simplify the second phase of PARTITION so that it can run efficiently. The penalty is that SCAN may not generate optimal plans in many cases where PARTITION does.

SCAN also has two phases of processing. The first phase is identical to that of PARTITION. In the second phase, SCAN picks an order for each cluster without searching over all the possible orders. So the second phase runs in $O(n)$ time. Note that since it does not search over the space of subsequences for each cluster, SCAN tends to generate plans that are inferior to those of PARTITION.

Lemma 3.4.8 *SCAN runs in* $O(n^2)$ *time, where* n *is the number of subgoals in the logical plan.* □

3.5 Performance Analysis

In this section, we address the following questions regarding the performance of CHAIN and PARTITION: How often do they find optimal plans? When they miss the optimal plans, what is the expected margin by which they miss? We attempt to answer these questions by way of performance analysis of the algorithms in a simulated environment.

3.5.1 Simulation Parameters

In our experiments, we had a test bed of 15 sources, which participated in integrated views on which queries could be posed. The source size distribution was 30% small, 60% medium, and 10% large. Each source in our test bed had two access templates, each requiring that a different attribute be bound.

We employed randomly generated queries that created logical plans over a subset of the 15 sources in the test bed. We varied the subset size from 1 to 10. For each query, we computed the plans generated by CHAIN and PARTITION. We also exhaustively searched for the optimal plan for the query. The cost of the three plans was computed based on the model in Section 3.2.

3.5.2 Experimental Results

For each number of subgoals in a logical plan n, we generated 1000 user queries, and studied the performance of the algorithms on these queries. Figure 3.6 plots the number of query subgoals (on the horizontal axis) versus the average margin by which generated plans miss the optimal plan (on the vertical axis). Both CHAIN and PARTITION found near-optimal plans in the entire range of inputs (a total of 1000 randomly generated queries) and, on the average, missed the optimal plan by less than 10%.

Figure 3.6 also shows that as n increased, the average cost of the plan did not necessarily increase. Even though this result is a bit surprising, we realize that it is because more subgoals in a logical plan means more chances to choose some subgoals that have low cost, and thereby decrease the cost of other expensive subgoals.

We also conducted experiments that measured the fraction of the queries for which the algorithms found optimal plans, and the maximum margin by which the algorithms missed the optimal plans. The results of these experiments are shown in Figure 3.7. Figure 3.7(a) shows how often CHAIN and PARTITION found optimal plans. Over the entire set of 1000

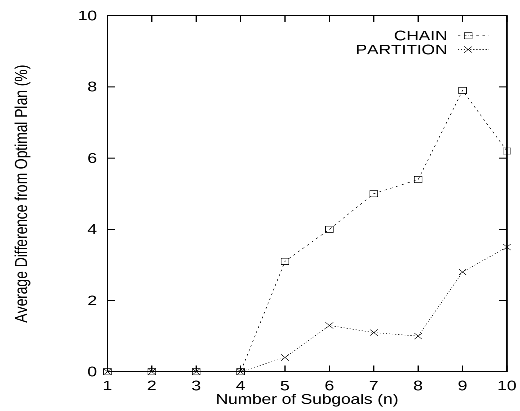


Figure 3.6: Average cost.

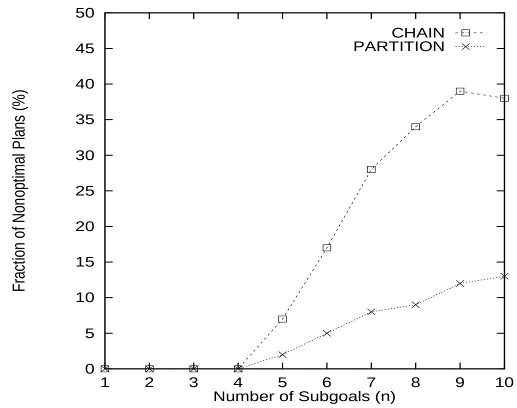
queries with n ranging from 1 to 10, CHAIN found the optimal plans more than 80% of the time, while PARTITION found the optimal plans more than 95% of the time. This result is surprising because we had proved in Section 3.3 that logical plans with as few as 3 subgoals can lead CHAIN to miss the optimal plans, and logical plans with as few as 4 subgoals can lead PARTITION to suboptimal plans.

Figure 3.7(b) shows the largest margin by which CHAIN and PARTITION miss the optimal plans. In the worst case, over the 1000 queries, CHAIN generated a plan that was 1.95 times as expensive as the optimal plan; the worst-case miss for PARTITION was a plan that was 1.5 times as expensive as the optimal plan. Once again, these results are surprising in that our theoretical results predicted that CHAIN can generate plans that cost as much as 10 times the optimal plan. We also proved that PARTITION can miss the optimal plan by an unbounded margin.

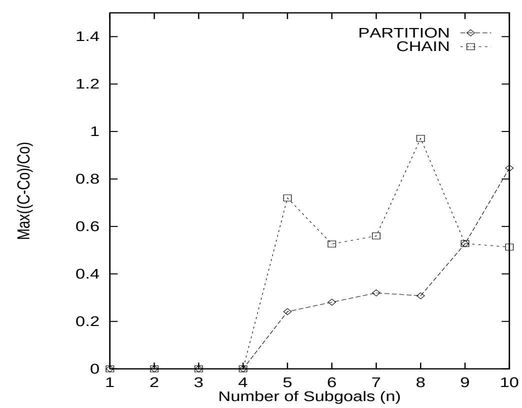
In summary, our experiments show that the PARTITION algorithm has excellent practical performance, even though it gives very few theoretical guarantees. The CHAIN algorithm also has very good performance, well beyond the theoretical guarantees we proved in Section 3.3. Finally, comparing the two algorithms, we observe that PARTITION consistently outperforms CHAIN in finding near-optimal plans.

3.6 Other Cost Models

So far, we discussed algorithms that minimize the number of source queries. Now, we consider more complex cost models where different source queries can have different costs.



(a) Fraction of non-optimal plan.



(b) Worst-case miss.

Figure 3.7: Optimality of each algorithm.

First, we consider a simple extension (called M_1) where the cost of a query to source S_i is e_i. That is, queries to different sources cost different amounts. Note that in M_1, we still do not charge for the amount of data transferred. All of our results presented so far hold in this new model.

Theorem 3.6.1 *In the cost model M_1, Theorem 3.3.1 holds. That is, the CHAIN algorithm is n-competitive, where n is the number of subgoals.* □

Theorem 3.6.2 *In the cost model M_1, Lemma 3.4.2 holds. That is, the PARTITION algorithm will find an optimal plan, if there are at most two clusters and the user query has a nonempty result.* □

Next, we consider a more complex cost model (called M_2) where the data-transfer costs are factored in. That is, the cost of a query to source S_i is $e_i + f_i \times$ (size of query result). Note that this cost model is strictly more general than M_1.

Theorem 3.6.3 *In the cost model M_2, Theorem 3.3.1 holds. That is, the CHAIN algorithm is n-competitive, where n is the number of subgoals.* □

Theorem 3.6.4 *In the cost model M_2, Lemma 3.4.2 does not hold. That is, the PARTITION algorithm cannot guarantee an optimal plan, even when there are at most two clusters.* □

When considering more complex cost models, we note that the problem of finding an optimal feasible plan remains $\mathcal{NP}$-hard. It is also clear that both CHAIN and PARTITION guarantee the generation of feasible plans (if they exist), irrespective of the cost model being considered. The only issue at hand is the ability of these algorithms to generate near-optimal plans.

We observe that the n-competitiveness of CHAIN holds in any cost model with the following property: the cost of a subgoal in a plan does not increase by postponing its processing to a later time in the plan. Both M_1 and M_2 have this property, so CHAIN is n-competitive in those models. We also note that the PARTITION algorithm with two clusters will always find the optimal plan (assuming the query has nonempty result) if block queries cannot cost more than the corresponding parameterized queries. This property holds, for instance, in model M_1, and not in model M_2.

When one considers cost models other than those discussed here, the properties noted above may hold in those models. Consequently CHAIN and PARTITION may yield very good results. Even when the properties do not hold, the strategies employed by the two algorithms may act as good heuristics and help them generate efficient plans. For instance, in the cost model M_2, PARTITION cannot guarantee the generation of optimal plans even when there are only two clusters. In our simulation experiments we studied the performance of PARTITION in this cost model, and found that it continues to find plans that are very close to optimal plans.

3.7 Conclusions and Related Work

In this chapter, we considered the problem of optimizing large-join queries in information integration when sources have limited access patterns. We employed a cost model that counts the number of source accesses in a plan. Under this cost model, we showed that the problem of finding an optimal feasible plan is $\mathcal{NP}$-hard. We developed two algorithms that guarantee the generation of feasible plans (when they exist). The first one, named CHAIN, uses a greedy approach to generate plans. It runs in polynomial time, and has a linear bound on the worst case margin by which it misses optimal plans. Another algorithm, called PARTITION, groups the subgoals into clusters, and constructs a plan by combining local plans in these clusters. Our experimental results showed that these algorithms generate optimal plans in many cases, and in other cases their generated plans have small distance from the optimal plans. We also discussed how to extend the results under this cost model to other more complex models.

Related Work

The problem of ordering subgoals to find the best feasible sequence can be viewed as the well known *join-order* problem. More precisely, we can assign *infinite* cost to infeasible sequences and then find the best join order. The join-order problem has been extensively studied in the literature, and many solutions have been proposed. Some solutions perform a rather exhaustive enumeration of plans, and hence do not scale well [AHY83, BGW+81, CM95, ES80, HKWY97, LYV+98, OL90, PGH96, PGLK97, SAC+79, VM96]. In particular, we are interested in Web scenarios with many sources and subgoals, so these schemes are too expensive. Some other solutions reduce the search space through techniques such as simulated annealing, random probes, or other heuristics [GLPK94, IK90, IW87, Mor88, PS82, SMK97, Swa89, SG88]. While these approaches may generate efficient plans in some cases, they do not have any performance guarantees in terms of the quality of generated plans, i.e., their generated plans can be arbitrarily far from the optimal ones. Many of these techniques may even fail to generate a feasible plan, while the user query does have a feasible plan.

Other solutions [IK84, KBZ86, SM97] use specific cost models and clever techniques that exploit them to produce optimal join orders efficiently. While these solutions are very good for the join-order problem where those cost models are appropriate, they are hard to adopt in our context because of two difficulties. The first is that it is not clear how to model the feasibility of mediator query plans in their frameworks. A direct application of their algorithms to the problem we are studying may end up generating infeasible plans, even though a feasible plan might exist. The second difficulty is that when we use cost models that emphasize the main costs in mediation systems, the optimality guarantees of their algorithms may not hold.

Another related work is [FLMS99], which also considers query optimization when relations have limited access patterns. There are two main differences between our work and theirs. First, we consider specific cost models that are realistic in Web scenarios. Thus we can develop several algorithms with good properties. For instance, CHAIN runs in polynomial time and has a linear bound on the worst-case margin by which it misses optimal plans. [FLMS99] considers general cost models, and its algorithms are exponential. Second, [FLMS99] shows that busy-tree plans are necessary to generate optimal plans in some scenarios, e.g., when some query results are cached. In our work, since we do not consider these scenarios, we proved that left-deep-tree plans can guarantee to include a feasible optimal plan.

Chapter 4

Answering Queries with Useful Bindings

Since sources in information integration can have diverse and limited query capabilities, in order to obtain maximum information from these restrictive sources to answer a query, one can access sources that are not specified in the query (i.e., off-query sources). For instance, suppose a relation R requires a movie title to return movie information. If there are other relations that provide movie titles for free, we can access these relations to obtain movie titles, and then use them to retrieve movie information from relation R. In this chapter, we propose a query-planning framework to compute maximal answers to queries in the presence of limited access patterns. We solve optimization problems in this framework, including how to decide whether accessing off-query sources is necessary, how to choose useful sources for a query, and how to test query containment. We develop algorithms to solve these problems, and thus compute the maximal answer to a query efficiently.

Chapter Organization

Section 4.1 shows that sources not in a query can contribute to the query result by providing useful binding information. Section 4.2 uses an example to introduce the notation in the chapter. In Section 4.3, we propose a query-planning framework in integration systems with source restrictions. In the framework, source descriptions and a query are translated into a datalog program, which can be evaluated on the sources to compute the maximal answer to the query. In Sections 4.4 and 4.5, we solve the problem of trimming useless source relations for a query. In Section 4.4, we discuss in what cases accessing off-query relations is necessary to answer a query. In Section 4.5, we develop a polynomial-time algorithm for finding all the relevant sources for a query. In Section 4.6, we show how to test whether the answer to one query is contained in that to another query. In such a case, the source accesses in the contained query can be saved. In Section 4.7 we discuss variations of the problem of computing maximal answers to queries. We conclude in Section 4.8 and discuss related work.

4.1 Introduction

The following example shows that because sources have restrictions on retrieving their information, sources not directly mentioned in a query can contribute to the query result.

EXAMPLE 4.1.1 Suppose we want to compare the average prices of the books sold by AMAZON.COM and BARNESANDNOBLE.COM. Since both sources require each source query to specify at least an ISBN, an author, or a title, we cannot retrieve all their book records. Suppose we can access PRENHALL.COM to retrieve all authors who have published books through the publisher. We can use this author list to query AMAZON.COM and BARNESANDNOBLE.COM to get books and their prices and to compute the average prices.

Note that the authors who publish books through Prentice Hall may also publish books through other publishers. We can use the author list of Prentice Hall as a seed list to retrieve book titles from the two bookstore sources, then use these titles to retrieve more authors, and so on so forth. We can repeat the iteration until the author list remains stable. Then we average the prices of the books from the two sources. □

Example 4.1.1 suggests that we can use the source PRENHALL.COM to retrieve bindings for the author domain, and use the bindings to answer the query, although this source is not mentioned directly in the query. In some cases, we can even access sources repeatedly to obtain bindings to compute more results to a query. In Section 4.3 we propose a framework of query planning in integration systems with source restrictions. In the framework, source descriptions and a query are translated into a datalog program, and we compute the maximal answer to the query by evaluating the program on the source relations. Datalog is used in the query planning since the planning process for a query can be recursive, although the query itself is not.

Being able to obtain the maximal results is desirable. However, the challenge is to return the results with the minimum cost. In other words, we do not want to involve all sources blindly during the plan-generation process. In particular, this framework exhibits two challenging problems. First, we need to decide when accessing off-query sources is necessary. Some of these accesses are essential, as they provide bindings that let us query sources, which we could not do otherwise. However, some accesses can be proven not to add anything to the query's answer. We show in what cases off-query accesses are necessary, and develop an algorithm for finding all the relevant sources for a query.

Second, we need to determine whether the maximal answer to a query is contained in that to another query. Since our framework often produces a recursive datalog program to answer a query optimally, and containment of datalog programs is undecidable [Shm93], our containment problem seems to be undecidable. (Our containment problem uses set semantics, not bag semantics.) We show that this containment problem is decidable since it can be reduced to containment of monadic programs, which is known to be decidable [CGKV88]. In addition, we introduce the question of *boundedness* for the programs in our framework. When one of the two programs in the containment test is bounded (i.e., it is equivalent to a finite union of conjunctive queries), the containment test can be performed efficiently [CM77, CV92, SY80]. We develop a polynomial-time algorithm for testing query boundedness.

In this chapter we focus on a class of conjunctive queries, called *connection queries*. A connection query is a natural join of distinct source views with the necessary selection and projection. (The details are described in Section 4.2.) Here we are taking the following universal-relation-like assumption [Ull89]: different attributes that share the same name in different views have the same meaning. However, universal-relation study did not consider restrictions of retrieving information from relations. As we will see in Section 4.2.2, a connection query can be generated in various cases, where our techniques are applicable. In Section 4.7 we discuss how to extend our results on connection queries to conjunctive queries.

4.2 Preliminaries

This section uses an example to introduce the notation used throughout the chapter.

EXAMPLE 4.2.1 Assume we are building a system to integrate the information from four sources of musical CDs, as shown in Table 4.1. Sources S_1 and S_2 have information about CDs and their songs; sources S_3 and S_4 have information about CDs, their artists, and their prices. To simplify the notation, we use attribute *Song* for song title and attribute *Cd* for CD title. The table also includes the binding patterns of the relations. For instance, every query sent to S_2 must provide a CD title. In other words, without the information about CD titles, source S_2 cannot be queried to produce answers.

Table 4.1: Four sources of musical CDs.

Source	Contents	Binding Pattern
S_1	$v_1(Song, Cd)$	*bf*
S_2	$v_2(Song, Cd)$	*fb*
S_3	$v_3(Cd, Artist, Price)$	*bff*
S_4	$v_4(Cd, Artist, Price)$	*fbf*

Source-view schemas can be represented by a hypergraph [Ull89], in which each node is an attribute and each hyperedge is a source relation (e.g., source view). The hypergraph of the four views is shown in Figure 4.1, which also shows the tuples at each source. To simplify the presentation, we use symbols t_i, c_j, and a_k to represent a song title, CD, and artist, respectively. For instance, the source view $v_1(Song, Cd)$ contains two tuples: $\langle t_1, c_1 \rangle$ and $\langle t_2, c_3 \rangle$. The figure shows the adornments of the attributes in each view.

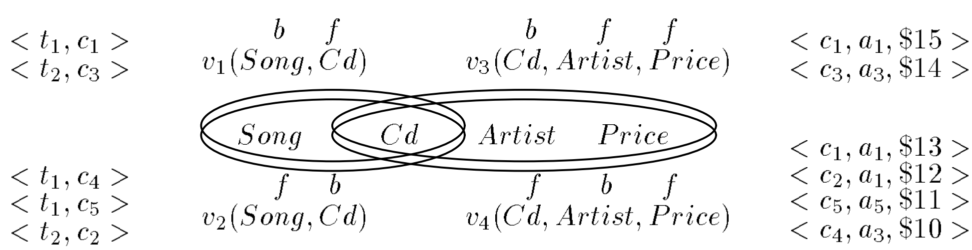


Figure 4.1: The hypergraph representation.

Suppose a user wants to find the prices of the CDs that contain a song titled t_1. The answer can be obtained by taking the union of the following four joins: $v_1 \bowtie v_3$, $v_1 \bowtie$

v_4, $v_2 \bowtie v_3$, and $v_2 \bowtie v_4$, and performing a selection $Song = t_1$ and then a projection onto the attribute *Price*. Figure 4.1 shows that there are four CDs containing the song: $\langle c_1, a_1, \$15\rangle$, $\langle c_1, a_1, \$13\rangle$, $\langle c_5, a_5, \$11\rangle$, and $\langle c_4, a_3, \$10\rangle$. Therefore, without considering the source restrictions, the answer is $\{\$15, \$13, \$11, \$10\}$. However, due to the limited source capabilities, only the \$15 can be computed if we process each join in the query at one time (as in [HKWY97, LRO96, LYV+98]). The reason is that, $v_1 \bowtie v_3$ yields the \$15 in the answer; $v_1 \bowtie v_4$ cannot be executed by using only v_1 and v_4, since v_4 requires that attribute *Artist* be specified, but we cannot bind this attribute using only these two views. Similarly, neither of the other two joins can be executed. As a consequence, the user misses the cheaper source for CD c_1 and entirely misses CDs c_4 and c_5. □

In this chapter, we propose a framework that can retrieve more results from sources with restrictions. Instead of considering each join individually, the framework involves other sources not in a join to produce bindings to answer the join. For instance, when joining v_1 and v_4, we also consider the information provided by v_2 and v_3. As we will see in Section 4.3.3, the framework can find two additional CDs containing the song titled t_1: $\langle c_1, a_1, \$13\rangle$ and $\langle c_4, a_3, \$10\rangle$. If the user wants to find the cheapest CD, this approach can save \$5 for the user!

4.2.1 Source Views

Now we give the notation used throughout the chapter. Let an information-integration system have n sources, say, $S_1, \ldots, S_n$. Assume that each source S_i provides its data in the form of a relational view v_i. If sources have other data models, we can use *wrappers* [HGMN+97] to create a simple relational view of data. In the case where one source has several relations, we can represent this source with several logical sources, each of which exports only one relational view.

We assume that differences in ontologies, vocabularies, and formats used by sources have been resolved. In particular, if two sources share an attribute name, we assume that the attributes are equivalent, i.e., wrappers take care of any differences. Related research [MW97, PGMW95, MKW00] suggests ways to deal with ontology and format differences. We assume that the schemas of the source views are defined on a global set of attributes. Each view schema is a list of global attributes, and different views may share the same schema. For instance, in Example 4.2.1, we have four global attributes: *Song*, *Title*, *Artist*, and *Price*; views v_1 and v_2 share the same schema $(Song, Cd)$.

For simplicity of exposition, we assume that each view has one template. We use v_i to stand for both the source view and its adorned template, and we believe the distinction should be clear in context. Let $\mathcal{A}(v_i)$ denote the attributes in a source view v_i, and let $\mathcal{B}(v_i)$ and $\mathcal{F}(v_i)$ be the sets of bound and free attributes in the adorned template of v_i, respectively. For instance, in Example 4.2.1, $\mathcal{B}(v_1) = \{Song\}$, $\mathcal{F}(v_1) = \{Cd\}$, and $\mathcal{A}(v_1) = \{Song, Cd\}$. Let $\mathcal{V}$ denote the source views with their adornments, $\mathcal{A}(\mathcal{V})$ be the attributes in $\mathcal{V}$.

4.2.2 Connection Queries

We consider a class of conjunctive queries, called *connection queries*. Each connection query is represented in the form

$$\mathcal{Q} = \langle \mathcal{I}, O, \mathcal{C} \rangle$$

where $\mathcal{I}$ is a list of input assignments of the form `attribute = constant`, O is a list of output attributes whose values the user is interested in, and $\mathcal{C}$ is a list of *connections*. Each connection is a set of source views that connect the input attributes and the output attributes. As we will see shortly, we interpret a connection as the natural join of the views in the connection. The following are some possible ways in which $\mathcal{C}$ could be generated:

1. It is generated by query expansion at a mediator, as in TSIMMIS [LYV+98].
2. It is generated by a minimal-connection algorithm, as in universal-relation systems [Ull89].
3. It is explicitly specified by the user.

For instance, the query in Example 4.2.1 can be represented as

$$\mathcal{Q} = \langle \{Song = t_1\}, \{Price\}, \{T_1, T_2, T_3, T_4\} \rangle$$

in which the four connections are: $T_1 = \{v_1, v_3\}$, $T_2 = \{v_1, v_4\}$, $T_3 = \{v_2, v_3\}$, and $T_4 = \{v_2, v_4\}$. Note that there can be several input attributes and several output attributes in a query. Let $I(\mathcal{Q})$ and $O(\mathcal{Q})$ respectively denote the input attributes and the output attributes of query $\mathcal{Q}$. $I(\mathcal{Q})$ and $O(\mathcal{Q})$ do not overlap. Let $\mathcal{A}(T)$ be all the attributes in a connection T.

4.2.3 The Answer to a Query

Suppose T is a connection in query $\mathcal{Q}$. For those tuples in the *natural join* of the relations in T that satisfy the input constraints in $\mathcal{Q}$, their projections onto the output attributes are the *complete answer to connection* T. The union of the answers to all the connections in $\mathcal{Q}$ is the *complete answer to query* $\mathcal{Q}$. Due to the limited source capabilities, the *obtainable answer to a connection* is the *maximal* answer to the connection that can be retrieved from the sources, using only the initial bindings in the query and the source relations. The union of the obtainable answers to all the connections in $\mathcal{Q}$ is the *obtainable answer to query* $\mathcal{Q}$.

The complete answer to a user query could be retrieved if the sources did not have limited capabilities. However, we may get only a partial answer to the query due to the source restrictions. For instance, in Example 4.2.1, the complete answer to the query is $\{\$15, \$13, \$11, \$10\}$, while as we will see in Section 4.3.3, the obtainable answer to the query is $\{\$15, \$13, \$10\}$. Given source descriptions and a query, if the complete answer to the query cannot be computed, our framework collects as much information as possible to answer the query. In the rest of this chapter, unless otherwise specified, the *answer to a connection* means the obtainable answer to the connection, and the *answer to a query* is the union of the obtainable answers to all the connections in the query.

4.3 A Query-Planning Framework

In this section, we propose a query-planning framework in the presence of source restrictions. In the framework, source descriptions and a query are translated into a datalog program, which can be evaluated to answer the query.

4.3.1 Constructing the Program $\Pi(\mathcal{Q}, \mathcal{V})$

Given source descriptions $\mathcal{V}$ and a query $\mathcal{Q}$, we translate them into a datalog program, denoted $\Pi(\mathcal{Q}, \mathcal{V})$. For instance, Figure 4.2 shows the datalog program $\Pi(\mathcal{Q}, \mathcal{V})$ for the query and the source views in Example 4.2.1. We use names beginning with lower-case letters for constants and predicate names, and names beginning with upper-case letters for variables. Note that this program is recursive, although query $\mathcal{Q}$ is not.

Let us look at the details of how the program $\Pi(\mathcal{Q}, \mathcal{V})$ is constructed. For each source view v_i, we introduce an EDB predicate ([Ull89]) v_i and an IDB predicate $\widehat{v_i}$ (called the *α-predicate* of v_i). Predicate v_i represents all the tuples at source S_i, and $\widehat{v_i}$ represents the

r_1:	$ans(P)$:- $\widehat{v_1}(t_1, C), \widehat{v_3}(C, A, P)$	r_9:	$\widehat{v_3}(C, A, P)$:- $cd(C), v_3(C, A, P)$
r_2:	$ans(P)$:- $\widehat{v_1}(t_1, C), \widehat{v_4}(C, A, P)$	r_{10}:	$price(P)$:- $cd(C), v_3(C, A, P)$
r_3:	$ans(P)$:- $\widehat{v_2}(t_1, C), \widehat{v_3}(C, A, P)$	r_{11}:	$artist(A)$:- $cd(C), v_3(C, A, P)$
r_4:	$ans(P)$:- $\widehat{v_2}(t_1, C), \widehat{v_4}(C, A, P)$	r_{12}:	$\widehat{v_4}(C, A, P)$:- $artist(A), v_4(C, A, P)$
r_5:	$\widehat{v_1}(S, C)$:- $song(S), v_1(S, C)$	r_{13}:	$cd(C)$:- $artist(A), v_4(C, A, P)$
r_6:	$cd(C)$:- $song(S), v_1(S, C)$	r_{14}:	$price(P)$:- $artist(A), v_4(C, A, P)$
r_7:	$\widehat{v_2}(S, C)$:- $cd(C), v_2(S, C)$	r_{15}:	$song(t_1)$:-
r_8:	$song(S)$:- $cd(C), v_2(S, C)$		

Figure 4.2: The datalog program $\Pi(\mathcal{Q}, \mathcal{V})$ in Example 4.2.1.

obtainable tuples at S_i. Introduce a goal predicate *ans* to store the answer to the query; the arguments of *ans* correspond to the output attributes $O(\mathcal{Q})$ in $\mathcal{Q}$.

Let $T = \{v_1, \ldots, v_k\}$ be a connection in $\mathcal{Q}$. The following rule is the *connection rule* of T:

$$ans(O(\mathcal{Q})) \text{ :- } \widehat{v_1}(\mathcal{A}(v_1)), \ldots, \widehat{v_k}(\mathcal{A}(v_k))$$

where the arguments in predicate *ans* are the corresponding attributes in $O(\mathcal{Q})$. For each argument in $\widehat{v_i}$, if the corresponding attribute in view v_i is an input attribute of $\mathcal{Q}$, this argument is replaced by the initial value of the attribute in $\mathcal{Q}$. Otherwise, a variable corresponding to the attribute name is used as an argument in predicate $\widehat{v_i}$. For instance, in Figure 4.2, rules r_1, r_2, r_3, and r_4 are the connection rules of the connections T_1, T_2, T_3, and T_4, respectively.

Decide the domains of all the attributes in the views, and group the attributes into sets while the attributes in each set share the same domain. Introduce a unary *domain predicate* for each domain to represent all its possible values that can be deduced.[1] In Figure 4.2, the predicates *song*, *cd*, *artist*, and *price* represent the domains of song titles, CD titles, artists, and prices, respectively.

Suppose that source view v_i has m attributes, say $A_1, \ldots, A_m$. Assume the adornment of v_i says that the arguments in positions $1, \ldots, p$ need to be bound, and the arguments in positions $p+1, \ldots, m$ can be free. The following rule is the α-*rule* of v_i:

$$\widehat{v_i}(A_1, \ldots, A_m) \text{ :- } domA_1(A_1), \ldots, domA_p(A_p), v_i(A_1, \ldots, A_m)$$

[1]The idea of using domain predicates in the framework is borrowed from [DL97]. However, in our framework, different domains have different domain predicates, while in [DL97] only one domain predicate is used for all attributes. In addition, we use the query-centric approach to information integration, while [DL97] uses the source-centric approach to information integration [Dus97].

in which each $domA_j$ ($j = 1, \ldots, p$) is the domain predicate for attribute A_j. For $k = p+1, \ldots, m$, the following rule is a *domain rule* of v_i:

$$domA_k(A_k) \text{ :- } domA_1(A_1), \ldots, domA_p(A_p), v_i(A_1, \ldots, A_m)$$

For instance, rule r_9 in Figure 4.2 is the α-rule of v_3; rules r_{10} and r_{11} are its domain rules. Assume that $A_i = a_i$ is in the assignment list $\mathcal{I}$ of $\mathcal{Q}$, the following rule is a *fact rule* of attribute A_i:

$$domA_i(a_i) \text{ :- }$$

For instance, rule r_{15} in Figure 4.2 is a fact rule of attribute $Song$, since we know from the query that t_1 is a song title.

The program $\Pi(\mathcal{Q}, \mathcal{V})$ is constructed in three steps:

1. Write the connection rule for each connection in $\mathcal{Q}$.
2. Write the α-rule and the domain rules for each source view in $\mathcal{V}$.
3. Write the fact rule for each input attribute in $\mathcal{Q}$.

In Figure 4.2, rules r_1, r_2, r_3, and r_4 are the connection rules of T_1, T_2, T_3, and T_4, respectively. Rule r_5 is the α-rule of v_1, and r_6 is the domain rule of v_1; rules r_7 to r_{14} are the α-rules and the domain rules of the other three source views. Finally, r_{15} is the fact rule of the attribute $Song$. Recall that the views in each connection link the input attributes and the output attributes in the query. Based on how program $\Pi(\mathcal{Q}, \mathcal{V})$ is constructed, we have the following proposition:

Proposition 4.3.1 *Given source descriptions $\mathcal{V}$ and a query $\mathcal{Q}$, the datalog program $\Pi(\mathcal{Q}, \mathcal{V})$ is safe.* □

4.3.2 Binding Assumptions

During the construction of the program $\Pi(\mathcal{Q}, \mathcal{V})$, we make the following important assumptions about the way we use bindings:

1. Each binding for an attribute must be from the domain of this attribute.
2. If a source view requires a value, say, a string, as a particular argument, we will not allow the strategy of trying all the possible strings to "test" the source.

3. Rather we assume that any binding is either obtained from the user query, or from a tuple returned by another source query.

We use Example 4.2.1 to explain these assumptions. The first assumption says that we would not use an artist name as a binding for attribute *Song*. View $v_3(Cd, Artist, Price)$ requires each query to source S_3 to give a CD title. The second assumption says that we would *not* allow the following naive "strategy": generate all possible strings to test whether S_3 has CDs with these strings as titles. This approach would not terminate, since there will be an infinite number of strings that need to be tested. The third assumption says that each binding of an attribute A must either be derived from the user query, or be a value of A in a tuple returned by another source query. For instance, if c_1 is a CD title returned from source S_1, and $Cd = c_2$ is an initial binding in a query, then we know that c_1 and c_2 are two CD titles, and we can use them to query source S_3. In Section 4.7 we will discuss other possibilities for obtaining bindings.

4.3.3 Evaluating the Program $\Pi(\mathcal{Q}, \mathcal{V})$

We evaluate the datalog program $\Pi(\mathcal{Q}, \mathcal{V})$ on the source relations to compute the facts for the *ans* predicate. Note that the v_i's are the only EDB predicates in $\Pi(\mathcal{Q}, \mathcal{V})$. However, because of the source restrictions, we do not know the tuples at each source before sending source queries. Now we show how to evaluate $\Pi(\mathcal{Q}, \mathcal{V})$ to answer the query.

To evaluate the domain rules and the α-rule of a source view v_i, predicate v_i is "populated" by source queries to S_i. Suppose that the right-hand side of its domain rules and its α-rule is:

$$domA_1(A_1), \ldots, domA_p(A_p), v_i(A_1, \ldots, A_m)$$

Once we know that $(a_1, \ldots, a_p)$ are the values of the $domA_j$'s ($j = 1, \ldots, p$), respectively, we can send a query $v_i(a_1, \ldots, a_p, A_{p+1}, \ldots, A_m)$ to source S_i. This source query is guaranteed to be executable, since it satisfies the binding requirements of v_i. The results of this source query add more tuples to the predicate $\widehat{v_i}$ (for the α-rule) and to the predicates $domA_j$'s (for the domain rules).

After the evaluation of the program terminates, the facts for the domain predicates include *all* the obtainable values of these domains. Similarly, the α-predicate facts are *all* the obtainable tuples at the sources. Thus, we have the following proposition:

Proposition 4.3.2 *Given source descriptions $\mathcal{V}$ and a query $\mathcal{Q}$, for any database $\mathcal{D}$ of $\mathcal{V}$, if we evaluate $\Pi(\mathcal{Q},\mathcal{V})$ on $\mathcal{D}$, the facts for the ans predicate are the obtainable answer to $\mathcal{Q}$.* □

Table 4.2: Evaluating the program in Figure 4.2.

Order	Source Query	Returned Tuple(s)	New Bindings(s)
1	$v_1(t_1, C)$	$\langle t_1, c_1\rangle$	$Cd = c_1$
2	$v_3(c_1, A, P)$	$\langle c_1, a_1, \$15\rangle$	$Artist = a_1$
3	$v_4(C, a_1, P)$	$\langle c_1, a_1, \$13\rangle$,$\langle c_2, a_1, \$12\rangle$	$Cd = c_2$
4	$v_2(S, c_2)$	$\langle t_2, c_2\rangle$	$Song = t_2$
5	$v_1(t_2, C)$	$\langle t_2, c_3\rangle$	$Cd = c_3$
6	$v_3(c_3, A, P)$	$\langle c_3, a_3, \$14\rangle$	$Artist = a_3$
7	$v_4(C, a_3, P)$	$\langle c_4, a_3, \$10\rangle$	$Cd = c_4$
8	$v_2(S, c_4)$	$\langle t_1, c_4\rangle$	

Table 4.3: Evaluation results.

IDBs	**Results**	**IDBs**	**Results**
$\widehat{v_1}$	$\langle t_1, c_1\rangle\langle t_2, c_3\rangle$	*song*	t_1, t_2
$\widehat{v_2}$	$\langle t_1, c_4\rangle\langle t_2, c_2\rangle$	*cd*	c_1, c_2, c_3, c_4
$\widehat{v_3}$	$\langle c_1, a_1, \$15\rangle\langle c_3, a_3, \$14\rangle$	*artist*	a_1, a_3
$\widehat{v_4}$	$\langle c_1, a_1, \$13\rangle, \langle c_2, a_1, \$12\rangle, \langle c_4, a_3, \$10\rangle$	*price*	$\$15, \$14, \$13, \$12, \$10$
ans	$\$15, \$13, \$10$		

Table 4.2 shows how to evaluate the program in Figure 4.2 to compute the answer to the query in Example 4.2.1, and Table 4.3 shows the results. Clearly the program computes all the obtainable values of song titles, CD titles, artists, and prices from the four sources and the query, as well as all the obtainable tuples at the sources. The set of *ans* facts is the answer to the query. Therefore, this approach returns two more tuples, \$13 and \$10, than the approach in Example 4.2.1. Note that we cannot retrieve the tuple $\langle t_1, c_5\rangle$ of v_2 or the tuple $\langle c_5, a_5, \$11\rangle$ of v_4, since we cannot obtain the binding a_5 for attribute *Artist*, no matter what legal source queries we execute.

The program $\Pi(\mathcal{Q},\mathcal{V})$ is constructed in a brute-force way, and it needs to be optimized. In particular, for each connection T in the query, the program may access views that are not in T. Some of these off-connection accesses do not add anything to the query's answer. We thus want to include judiciously only those sources that provide some values at a place where they are needed. In the rest of the chapter, we solve some optimization problems in this framework, including how to decide whether accessing off-query sources is necessary, how to choose the relevant sources to obtain useful bindings, and how to test query containment.

4.4 Accessing Off-connection Views

In this section, we discuss in what cases accessing off-connection views is necessary to answer a connection. The following example shows that accessing off-connection views is not always necessary.

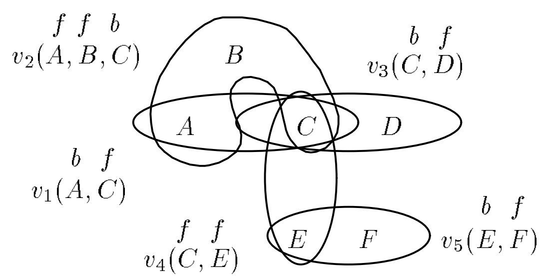


Figure 4.3: Source views in Example 4.4.1.

EXAMPLE 4.4.1 Consider the five views in Figure 4.3. Suppose that a user submits a query

$$\mathcal{Q} = \langle \{A = a_0\}, \{D\}, \{T_1, T_2\} \rangle,$$

which has two connections $T_1 = \{v_1, v_3\}$, $T_2 = \{v_2, v_3\}$. That is, the user knows the value of A is a_0, and wants to get the associated D values using $v_1 \bowtie v_3$ and $v_2 \bowtie v_3$. Assume that different attributes have different domains. The corresponding datalog program $\Pi(\mathcal{Q}, \mathcal{V})$ is shown in Figure 4.4.

Consider connection T_1. The program $\Pi(\mathcal{Q}, \mathcal{V})$ accesses the three views that are not in T_1 during the evaluation of the program. However, these off-connection accesses do not contribute to T_1's results. Indeed, suppose $t = \langle d \rangle$ is a tuple in the complete answer to T_1, and t comes from tuple $t_1 = \langle a_0, c \rangle$ of v_1 and tuple $t_3 = \langle c, d \rangle$ of v_3. By sending a query $v_1(a_0, C)$ to S_1 we can retrieve tuple t_1. With the new binding $C = c$, we can send a query $v_3(c, D)$ to S_3, and retrieve tuple t_3. Therefore, by using only the views in T_1 we can compute its complete answer.

$$
\begin{array}{llll}
r_1: & ans(D) & \text{:-} & \widehat{v_1}(a_0, C), \widehat{v_3}(C, D) \\
r_2: & ans(D) & \text{:-} & \widehat{v_2}(a_0, B, C), \widehat{v_3}(C, D) \\
r_3: & \widehat{v_1}(A, C) & \text{:-} & domA(A), v_1(A, C) \\
r_4: & domC(C) & \text{:-} & domA(A), v_1(A, C) \\
r_5: & \widehat{v_2}(A, B, C) & \text{:-} & domC(C), v_2(A, B, C) \\
r_6: & domA(A) & \text{:-} & domC(C), v_2(A, B, C) \\
r_7: & domB(B) & \text{:-} & domC(C), v_2(A, B, C) \\
r_8: & \widehat{v_3}(C, D) & \text{:-} & domC(C), v_3(C, D)
\end{array}
\qquad
\begin{array}{llll}
r_9: & domD(D) & \text{:-} & domC(C), v_3(C, D) \\
r_{10}: & \widehat{v_4}(C, E) & \text{:-} & v_4(C, E) \\
r_{11}: & domC(C) & \text{:-} & v_4(C, E) \\
r_{12}: & domE(E) & \text{:-} & v_4(C, E) \\
r_{13}: & \widehat{v_5}(E, F) & \text{:-} & domE(E), v_5(E, F) \\
r_{14}: & domF(F) & \text{:-} & domE(E), v_5(E, F) \\
r_{15}: & domA(a_0) & \text{:-} &
\end{array}
$$

Figure 4.4: The datalog program $\Pi(\mathcal{Q}, \mathcal{V})$ in Example 4.4.1.

Consider connection T_2. Since we cannot get any binding for attribute C by using only the two views in T_2, we need v_2 and v_4 to contribute C bindings. Thus these two off-connection views are useful to T_2. On the other hand, $v_5(E, F)$ does not contribute to T_2's results, because the F bindings from S_5 do not help obtain more answers to T_2. $\Box$

In general, given a connection T in a query $\mathcal{Q}$, we need to decide whether accessing the views outside T is necessary. Before giving the solution, we first introduce some definitions.

4.4.1 Forward-closure

Definition 4.4.1 (**forward-closure**) Given a set of source views $\mathcal{W} \subseteq \mathcal{V}$ and a set of attributes $X \subseteq \mathcal{A}(\mathcal{V})$, the *forward-closure of X given $\mathcal{W}$*, denoted *f-closure*$(X, \mathcal{W})$, is a set of the source views in $\mathcal{W}$ such that, starting from the attributes in X as the initial bindings, the binding requirements of these source views are satisfied by using only the source views in $\mathcal{W}$. $\Box$

We can compute *f-closure*$(X, \mathcal{W})$ as follows. At the beginning, only the attributes in X are bound, and *f-closure*$(X, \mathcal{W})$ is empty. At each step, for each source view $v \in \mathcal{W}$ − *f-closure*$(X, \mathcal{W})$, check whether $\mathcal{B}(v)$, the bound attributes of v, is a subset of the bound attributes so far. If so, add v to *f-closure*$(X, \mathcal{W})$, and each attribute in $\mathcal{F}(v)$, the free attributes of v, becomes bound. Repeat this process until no more source views can be added to *f-closure*$(X, \mathcal{W})$. Let $\mathcal{A}($*f-closure*$(X, \mathcal{W}))$ denote all the attributes of the source views in *f-closure*$(X, \mathcal{W})$. Therefore, $\mathcal{A}($*f-closure*$(X, \mathcal{W}))$ includes all the attributes that can be bound eventually by using the source views in $\mathcal{W}$ starting from the initial bindings in X.

EXAMPLE 4.4.2 In Example 4.4.1, *f-closure*$(\{A\}, \{v_1, v_2, v_3\}) = \{v_1, v_2, v_3\}$, since we can use the bound attribute A to get tuples of v_1 and bind C, which is the only bound attribute of v_2 and v_3. Similarly, in Example 4.2.1, *f-closure*$(\{Song\}, \{v_1, v_4\}) = \{v_1\}$, and *f-closure*$(\{Song\}, \{v_1, v_3\}) = \{v_1, v_3\}$. □

4.4.2 Independent Connections

Definition 4.4.2 (**independent connections**) A connection T in a query $\mathcal{Q}$ is *independent* if

$$f\text{-}closure(I(\mathcal{Q}), T) = T.$$

That is, the binding requirements of the source views in the connection can be satisfied by using only these source views and starting from the initial bindings in $I(\mathcal{Q})$. □

In other words, if connection $T = \{w_1, \ldots, w_k\}$ is independent, then there exists a feasible sequence of all the source views in connection T: $w_{i_1}, \ldots, w_{i_k}$, such that $\mathcal{B}(w_{i_1}) \subseteq I(\mathcal{Q})$, and for $j = 2, \ldots, k$, $\mathcal{B}(w_{i_j}) \subseteq I(\mathcal{Q}) \cup \mathcal{A}(w_{i_1}) \cup \cdots \cup \mathcal{A}(w_{i_{j-1}})$. (See Section 3.2.1.) For instance, the connection $T_1 = \{v_1, v_3\}$ in Example 4.4.1 is independent, since it has a feasible sequence: v_1, v_3. The following theorem shows that an independent connection does not require bindings from views outside the connection.

Theorem 4.4.1 *If connection T is independent, then for any database of the sources, we can compute the complete answer to T by using only the source views in T.* □

Proof: Suppose connection T has k source views, and it has a feasible sequence $v_1, \ldots, v_k$. Consider each tuple t in the complete answer to T. Assume that tuple t comes from tuples $t_1, \ldots, t_k$ of source views $v_1, \ldots, v_k$, respectively. Since $v_1, \ldots, v_k$ is a feasible sequence, the binding requirements of v_1 are satisfied by $I(\mathcal{Q})$, i.e., $\mathcal{B}(v_1) \subseteq I(\mathcal{Q})$. Thus, we can send a source query to S_1 by binding the attributes in $\mathcal{B}(v_1)$ to their initial values in $\mathcal{Q}$, and retrieve the tuple t_1 from v_1. We then use the bound values of $I(\mathcal{Q}) \cup \mathcal{F}(v_1)$ to send S_2 a source query to get tuple t_2. Repeat this process following the feasible sequence, until we retrieve all the t_i's. Therefore, by using only the views in T, we can retrieve the tuple t in the complete answer to the connection. ■

Theorem 4.4.2 *For a nonindependent connection T, there exists a database of the sources, such that some tuples in the complete answer to T cannot be obtained.* □

Proof: If connection T is not independent, i.e., *f-closure*$(I(\mathcal{Q}), T) \neq T$, we construct an instance of source relations, such that a tuple in the complete answer to the connection cannot be obtained. Let $\mathcal{A}(T) = \{A_1, \ldots, A_n\}$ be the set of attributes in T. Let tuple $t = (a_1, \ldots, a_n)$, where a_i is a distinct value for attribute A_i. If A_i is in $I(\mathcal{Q})$, then its value in t, a_i, is its initial value in $\mathcal{Q}$. Each view v_i in T has only one tuple t_i, which is the projection of t onto the attributes $\mathcal{A}(v_i)$. Other sources are empty. Then the projection of t onto $O(\mathcal{Q})$ is in the complete answer to T. However, since *f-closure*$(I(\mathcal{Q}), T) \neq T$, and all other sources are empty, we can only retrieve the tuples in *f-closure*$(I(\mathcal{Q}), T)$. We do not have the necessary bindings to retrieve the tuples in $T -$ *f-closure*$(I(\mathcal{Q}), T)$, thus we cannot compute any answer to connection T. ∎

Many related studies (e.g., [FLMS99, LYV+98]) consider the case where a connection in a query is independent. If the connection is not independent, their algorithms give up attempting to answer the connection. However, our framework can still compute a partial answer to the connection by accessing off-connection views.

4.5 Finding Relevant Source Views of a Connection

For a nonindependent connection, not all its off-connection accesses can contribute to the connection's results. In this section, we discuss what sources should be accessed to answer a connection. To simplify the presentation, in the rest of the chapter we assume that different attributes are from different domains.

Definition 4.5.1 (**relevant source view of a connection**) Given source descriptions $\mathcal{V}$, a query $\mathcal{Q}$, and a connection T in $\mathcal{Q}$, a source view $v \in \mathcal{V}$ is *relevant* to connection T if for some source relations, removing v from $\mathcal{V}$ can change the obtainable answer to connection T; otherwise, v is *irrelevant* to connection T. □

In other words, a source view is relevant to a connection T if we can miss some answers to T if we do not use this view. Note that whether a source view is relevant to a connection does not depend on other connections in the query.

EXAMPLE 4.5.1 Consider the five views in Figure 4.5. Suppose that a user submits a query

$$\mathcal{Q} = \langle \{A = a\}, \{F, G\}, \{T\} \rangle,$$

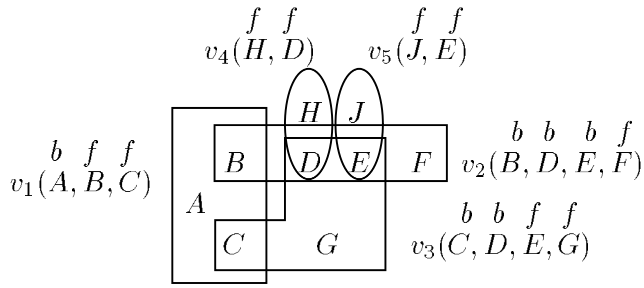


Figure 4.5: The source views in Example 4.5.1.

which has one connection $T = \{v_1, v_2, v_3\}$. That is, the user knows the value of A is a, and wants to get the associated F and G values using $v_1 \bowtie v_2 \bowtie v_3$. Connection T is not independent, since we cannot bind attributes D and E by using only the views in T starting from the initial binding in $\mathcal{Q}$. We need other views to bind D and E, so that we can query S_2 and S_3 to retrieve tuples. Thus, v_4 and v_5 may be useful.

However, although view v_5 can bind attribute E, it is *not relevant* to connection T. To illustrate the reason, we prove that the obtainable answers to T can be computed by using only v_1, v_2, v_3, and v_4. Suppose tuple $t = \langle f, g \rangle$ is in the obtainable answers, and t comes from tuple $t_1 = \langle a, b, c \rangle$ of v_1, tuple $t_2 = \langle b, d, e, f \rangle$ of v_2, and tuple $t_3 = \langle c, d, e, g \rangle$ of v_3. Since the initial value of A in the query is a, we can send a source query $v_1(a, B, C)$ to retrieve tuple t_1 from v_1. Because attribute D is not in $I(\mathcal{Q})$, and only v_4 (with binding pattern *ff*) takes D as a free attribute, the value d of D must be derived from the result of a source query to S_4, which includes a tuple whose D value is d. With $C = c$ and $D = d$, we can retrieve tuple t_3 from v_3 by sending a source query $v_3(c, d, E, G)$, and then retrieve tuple t_2 from v_2 by sending a source query $v_2(b, d, e, F)$. Thus, without using v_5, we can get tuple t in the obtainable answers to connection T. The proof also shows that without using v_4, we cannot get any answer to T. $\square$

As there may be many views with different schemas and binding patterns, it becomes challenging to decide which views can really contribute to the results of a connection. Before giving the algorithm for finding all the relevant views of a connection, we require a series of definitions.

4.5.1 Queryable Source Views

A source view is *queryable* if it is in *f-closure*$(I(\mathcal{Q}), \mathcal{V})$. All the queryable source views are those that we may eventually query, starting from the initial bindings in $I(\mathcal{Q})$, and perhaps using several preliminary queries to other sources in order to get the bindings we need for these source views. Let $\mathcal{V}_q$ denote all the queryable source views in $\mathcal{V}$, and $\mathcal{A}(\mathcal{V}_q)$ be all the attributes in $\mathcal{V}_q$.

Lemma 4.5.1 *If attribute A is in $\mathcal{A}(\mathcal{V}_q)$, then either A is in $I(\mathcal{Q})$, or there exists a feasible sequence of source views, such that A is a free attribute of the tail in the sequence.* □

Proof: If attribute A is not in $I(\mathcal{Q})$, since A is in $\mathcal{A}(\mathcal{V}_q)$, there must exist a source view v in $\mathcal{V}_q$, such that A is in $\mathcal{F}(v)$. Based on how $\mathcal{V}_q =$ *f-closure*$(I(\mathcal{Q}), \mathcal{V})$ is computed, there must exist a feasible sequence of source views in which v is the tail. This sequence can be obtained by backtracking from v following the reverse order in which these source views are added into *f-closure*$(I(\mathcal{Q}), \mathcal{V})$. ■

We cannot get any tuples from a nonqueryable source view, no matter what the source relations are. If a connection contains a nonqueryable source view, we cannot get any answer to this connection. Thus we need to consider only the *queryable connections* in $\mathcal{Q}$, i.e., the connections that do not have any nonqueryable source view. Clearly an independent connection is also a queryable connection, but not vice versa. For instance, in Example 4.4.1, connection T_2 is queryable, since both v_2 and v_3 are queryable source views, but T_2 is not independent.

4.5.2 Kernel, BF-chain, and Backward-closure

Definition 4.5.2 (kernel) Assume T is a queryable connection in query $\mathcal{Q}$. A set of attributes $\mathcal{K} \subseteq \mathcal{A}(T)$ is a *kernel* of T if

$$\textit{f-closure}(\mathcal{K} \cup I(\mathcal{Q}), T) = T$$

and

$$\textit{f-closure}((\mathcal{K} - \{A\}) \cup I(\mathcal{Q}), T) \neq T$$

for any attribute $A \in \mathcal{K}$. □

Intuitively, a kernel $\mathcal{K}$ of connection T is a minimal set of attributes in $\mathcal{A}(T)$ such that, if the attributes in $\mathcal{K}$ have been bound, together with the initial bindings in $I(\mathcal{Q})$, we can bind all the attributes $\mathcal{A}(T)$ by using only the source views in T. In Example 4.4.1, $\{C\}$ is a kernel of connection T_2, because $\textit{f-closure}(\{C\} \cup I(\mathcal{Q}), T_2) = \textit{f-closure}(\{C, A\}, T_2) = T_2$. Also since $\textit{f-closure}(\{A\} \cup I(\mathcal{Q}), T_2) = \textit{f-closure}(\{A\}, T_2) \neq T_2$, then $\{A\}$ is not a kernel. In Example 4.5.1, $\{D\}$ is a kernel of the connection T, while $\{D, E\}$ is not. Since a kernel of a connection must be minimal, it cannot share any attribute with $I(\mathcal{Q})$.

We compute a kernel of a connection T by shrinking the set of attributes $X = \mathcal{A}(T) - I(\mathcal{Q})$ as much as possible, while X satisfies: $\textit{f-closure}(X \cup I(\mathcal{Q}), T) = T$. When X cannot be smaller, it will be a kernel of T. An independent connection has only one kernel: the empty set. A nonindependent connection has only nonempty kernels. It may have multiple kernels, as shown by the following example.

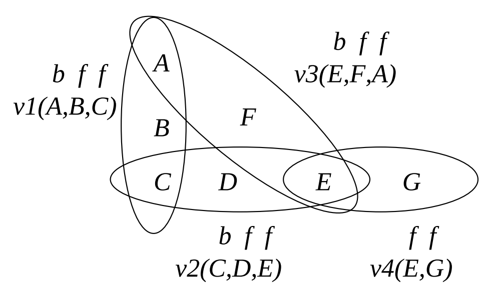


Figure 4.6: Multiple kernels of a connection.

EXAMPLE 4.5.2 Figure 4.6 shows a hypergraph of four source views. The binding patterns for $v_1(A, B, C)$, $v_2(C, D, E)$, and $v_3(E, F, A)$ are all *bff*, and the binding pattern for $v_4(E, G)$ is *ff*. Assume a user query is $\mathcal{Q} = \langle \{B = b_0\}, \{A, C, E\}, \{T\} \rangle$, in which the only connection is $T = \{v_1, v_2, v_3\}$. T has three kernels: $\{A\}$, $\{C\}$, and $\{E\}$. For instance, $\{A\}$ is a kernel because $\textit{f-closure}(\{A\} \cup I(\mathcal{Q}), T) = \textit{f-closure}(\{A, B\}, \{v_1, v_2, v_3\}) = \{v_1, v_2, v_3\} = T$. $\Box$

Definition 4.5.3 (**BF-chain**) A sequence of queryable source views $w_1, \ldots, w_k$ (i.e., each $w_i \in \mathcal{V}_q$) forms a *BF-chain* (bound-free chain) if for $i = 1, \ldots, k-1$, $\mathcal{F}(w_i) \cap \mathcal{B}(w_{i+1})$ is not empty. The source views w_1 and w_k are the *head* and the *tail* of the BF-chain, respectively. $\Box$

In other words, for every two adjacent views in a BF-chain, the free attributes of the first one overlap with the bound attributes of the second, and thus the first view contributes bindings to the second one. In Example 4.4.1, (v_4, v_2, v_1, v_3) is a BF-chain, in which v_4 is the head and v_3 is the tail.

Definition 4.5.4 (**backward-closure**) Suppose A is an attribute in $\mathcal{A}(\mathcal{V}_q)$. The *backward-closure of* A, denoted *b-closure*(A), is the set of queryable views that can be backtracked from A by following some BF-chain in a reverse order, in which A is a free attribute of the tail in the BF-chain. □

We can compute *b-closure*(A) as follows. Start by setting *b-closure*(A) to those source views in $\mathcal{V}_q$ that take A as a free attribute. For each view $v \in \mathcal{V}_q -$ *b-closure*(A), if there is a view $w \in$ *b-closure*(A) such that $\mathcal{F}(v)$ and $\mathcal{B}(w)$ overlap, then add v to *b-closure*(A). Repeat this process until no more queryable source views can be added to *b-closure*(A). In Example 4.4.1, the backward-closure of attribute C is $\{v_1, v_2, v_4\}$. In Figure 4.7, the backward-closure of $\{B\}$ is $\{v_2, v_3, v_4\}$. The backward-closure of a set of attributes $X \subseteq \mathcal{A}(\mathcal{V}_q)$, denoted *b-closure*$(X)$, is the union of all the backward-closures of the attributes in X, i.e., *b-closure*$(X) = \bigcup_{A \in X}$ *b-closure*(A).

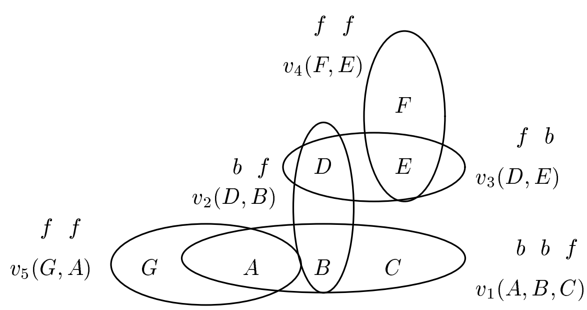


Figure 4.7: A backward-closure.

By the definitions of kernel, BF-chain, and backward-closure, we have the following lemmas.

Lemma 4.5.2 *If* $\mathcal{K}$ *is a kernel of a queryable connection* T *and* A *is an attribute in* $\mathcal{K}$*, then* A *is not in* $\mathcal{A}\Big(f\text{-}closure((\mathcal{K} - \{A\}) \cup I(\mathcal{Q}), T)\Big)$*. That is, starting from the attributes of* $(\mathcal{K} - \{A\}) \cup I(\mathcal{Q})$ *as the initial bindings, we cannot bind attribute* A *by using only the source views in* T*.* □

Proof: Suppose that attribute A is in $\mathcal{K}$, and A is also in $\mathcal{A}\Big(\textit{f-closure}((\mathcal{K}-\{A\})\cup I(\mathcal{Q}),T)\Big)$. Then starting from the attributes of $(\mathcal{K}-\{A\})\cup I(\mathcal{Q})$ as the initial bindings, we can bind attribute A by using only the source views in T. Therefore, we can bind all the attributes in $\mathcal{K}$, and then bind all the attributes in $\mathcal{A}(T)$ (since $\mathcal{K}$ is a kernel of T). Thus, $\mathcal{K}-\{A\}$ would be a kernel of connection T. Then $\mathcal{K}$ could not be a kernel since it is not minimal. ∎

Lemma 4.5.3 *If A_1 and A_2 are two attributes, and there is a BF-chain such that A_1 is a bound attribute of the head and A_2 is a free attribute of the tail, then b-closure$(A_1) \subseteq$ b-closure(A_2).* □

Proof: Since A_2 is a free attribute of the BF-chain tail, we can backtrack from A_2 along the BF-chain until we reach A_1 in the head. Thus all the views on the BF-chain are in *b-closure*(A_2). By the definition of *b-closure*(A_2), all the views in *b-closure*(A_1) are also in *b-closure*(A_2). Therefore, *b-closure*$(A_1) \subseteq$ *b-closure*(A_2). ∎

Lemma 4.5.4 *If connection T has two different kernels $\mathcal{K}_1$, $\mathcal{K}_2$, then b-closure$(\mathcal{K}_1) =$ b-closure$(\mathcal{K}_2)$.* □

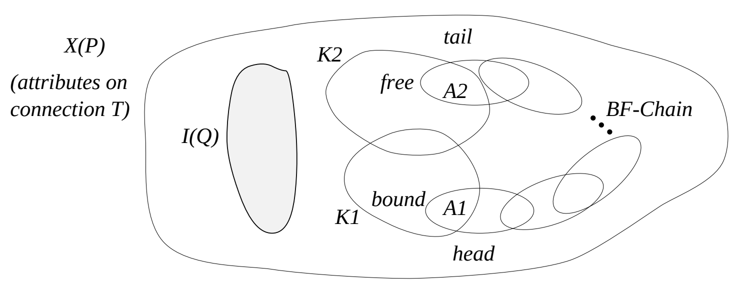


Figure 4.8: Proof of Lemma 4.5.4.

Proof: The main idea of the proof is shown in Figure 4.8. Since connection T has two different kernels, T cannot be independent, and both $\mathcal{K}_1$ and $\mathcal{K}_2$ are not empty. Since $\mathcal{K}_1$ is a kernel of T, for each attribute in $\mathcal{K}_1$, say A_1, by Lemma 4.5.2, we have $A_1 \notin \mathcal{A}\Big(\textit{f-closure}((\mathcal{K}_1-\{A\})\cup I(\mathcal{Q}),T)\Big)$. We also have *f-closure*$(\mathcal{K}_1\cup I(\mathcal{Q}),T) = T$, while *f-closure*$((\mathcal{K}_1-\{A_1\})\cup I(\mathcal{Q}),T) \neq T$. In addition, $\mathcal{A}(\textit{f-closure}((\mathcal{K}_1-\{A_1\})\cup I(\mathcal{Q}),T))$ cannot be a superset of $\mathcal{K}_2$, otherwise starting from the attributes of $(\mathcal{K}_1-\{A_1\})\cup I(\mathcal{Q})$

as initial bindings and using only the source views in T, we could bind all the attributes in $\mathcal{K}_2$, and then bind all the attributes in T (since $\mathcal{K}_2$ is a kernel of T), and $\mathcal{K}_1 - \{A_1\}$ would be a kernel.

Let A_2 be an attribute in $\mathcal{K}_2$ that is not in $\mathcal{A}(\textit{f-closure}(\mathcal{K}_1 - \{A_1\}) \cup I(\mathcal{Q}), T))$. As shown in Figure 4.8, since A_2 must be in $\mathcal{A}(\textit{f-closure}(\mathcal{K}_1) \cup I(\mathcal{Q}), T))$, then either $A_2 = A_1$, or there must exist a BF-chain, such that all the source views on the BF-chain are in T, and the head of the BF-chain takes A_1 as a bound attribute, and the tail takes A_2 as a free attribute. Note that in Figure 4.8, the attributes in $I(\mathcal{Q})$ may overlap with the attributes on the BF-chain.

If $A_2 = A_1$, then $\textit{b-closure}(A_1) = \textit{b-closure}(A_2) \subseteq \textit{b-closure}(\mathcal{K}_2)$. If $A_2 \neq A_1$, then the above BF-chain exists. By Lemma 4.5.3, $\textit{b-closure}(A_1) \subseteq \textit{b-closure}(A_2) \subseteq \textit{b-closure}(\mathcal{K}_2)$. Then we have $\textit{b-closure}(\mathcal{K}_1) \subseteq \textit{b-closure}(\mathcal{K}_2)$, since $\textit{b-closure}(\mathcal{K}_1) = \bigcup_{A \in \mathcal{K}_1}(\textit{b-closure}(A))$. Similarly, we can prove $\textit{b-closure}(\mathcal{K}_2) \subseteq \textit{b-closure}(\mathcal{K}_1)$. Therefore, we have $\textit{b-closure}(\mathcal{K}_1) = \textit{b-closure}(\mathcal{K}_2)$. ∎

For instance, in Example 4.5.2, the connection $T = \{v_1, v_2, v_3\}$ has three kernels: $\{A\}$, $\{C\}$, and $\{E\}$, and they have the same backward-closure: $\{v_1, v_2, v_3, v_4\}$.

4.5.3 Finding Relevant Source Views of a Connection

Now we show how to find all the relevant views of a connection by giving the following theorem:

Theorem 4.5.1 *If $\mathcal{K}$ is a kernel of a queryable connection T, then $\textit{b-closure}(\mathcal{K}) \cup T$ are all the relevant source views of connection T.* □

Proof: We need to prove that, for a kernel $\mathcal{K}$ of a queryable connection T:

1. All the source views that are not in $\textit{b-closure}(\mathcal{K}) \cup T$ are irrelevant to T.

2. Every source view on T is relevant to T.

3. Every source view in $\textit{b-closure}(\mathcal{K})$ is relevant to T.

Proof of (1) We need to show that, by considering only the source views in $\textit{b-closure}(\mathcal{K}) \cup T$, we can get the obtainable answer for connection T. Suppose $T = \{v_1, \ldots, v_k\}$. For each tuple t in the obtainable answer for T, assume that t comes from the tuples $t_1, \ldots, t_k$ of

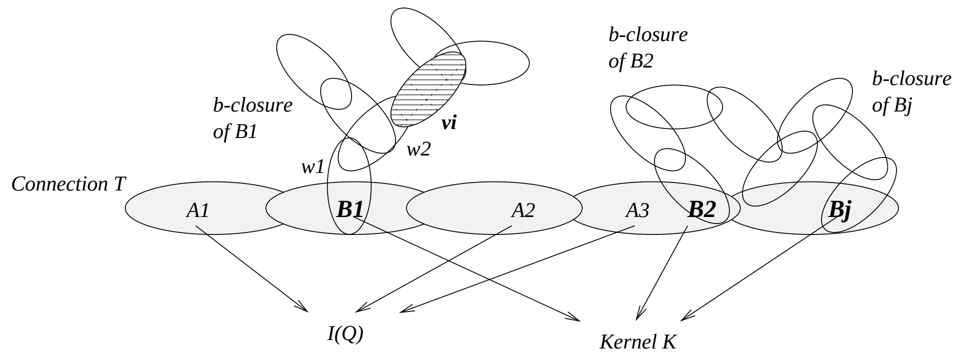


Figure 4.9: The backward-closure of a kernel of a queryable connection.

source views $v_1, \ldots, v_k$, respectively. The values of the attributes $I(\mathcal{Q})$ in these t_i's must be their initial values in $\mathcal{Q}$. Now consider the attributes in $\mathcal{K}$ and how their values in the t_i's are derived. These values must be obtained by some source queries using a set of source views, say $\mathcal{V}_1$, whose free attributes include $\mathcal{K}$. By the definition of *b-closure*$(\mathcal{K})$, the source views in $\mathcal{V}_1$ must be in *b-closure*$(\mathcal{K})$. In order to send source queries to the sources of $\mathcal{V}_1$, the binding requirements of the views in $\mathcal{V}_1$ must be satisfied. Let $\mathcal{A}$ be the attributes of $\mathcal{V}_1$ such that the values of $\mathcal{A}$ are not the initial values in $\mathcal{Q}$. Then the values of $\mathcal{A}$ must be obtained using a set of source views, say $\mathcal{V}_2$, whose free attributes include $\mathcal{A}$. Clearly the source views in $\mathcal{V}_2$ are also in *b-closure*$(\mathcal{K})$. We can see that all the source views used to derive the values of the attributes $\mathcal{K}$ in the t_i's are in *b-closure*$(\mathcal{K})$. After we get these values, following the same idea as in the proof of Theorem 4.4.1, we can get tuple t in the obtainable answer for T by using only the source views in T. Therefore, by using the source views in *b-closure*$(\mathcal{K}) \cup T$, we can get all the tuples in the obtainable answer for T.

Proof of (2) By Lemma 4.5.1, we can construct an instance of source relations $\mathcal{R}$ such that the obtainable answer for T is not empty. But if we remove any source view in T, the obtainable answer for T becomes empty.

Proof of (3) For every source view v_i in *b-closure*$(\mathcal{K})$, we prove that v_i is relevant to connection T by constructing an instance of source relations $\mathcal{R}$, such that without using v_i, we cannot get a tuple in the obtainable answer for T. By the definition of *b-closure*$(\mathcal{K})$, there must be a BF-chain, as shown in Figure 4.9, such that the head of the BF-chain is v_i, and the tail (denoted w_1 in the figure) takes an attribute in $\mathcal{K}$ (denoted B_1) as a free attribute.

$\mathcal{R}$ is constructed in four steps. Start by setting all the source relations to be empty. In step 1, populate the relations in connection T following the same idea as in the proof of Theorem 4.4.2. Let b_1, b_2, ...be the values of attributes B_1, B_2, ...in Figure 4.9, and t be the projection of the join of these tuples onto the attributes $O(\mathcal{Q})$. After we finish the construction of $\mathcal{R}$, tuple t will be in the obtainable answer for T.

In step 2, as shown in Figure 4.10, populate the source relations on the BF-chain from v_i to w_1 in such a way that only following this BF-chain we can get the value b_1 of attribute B_1. Follow the same idea as in the proof of Theorem 4.4.2 to populate the source relations on the BF-chain, except that the B_1-value in the tuple of w_1 is b_1. For each source view w_j on the BF-chain, w_j may have a set of bound attributes $Z(w_j) \subseteq \mathcal{B}(w_j)$ that do not overlap with the free attributes of the previous source view on the BF-chain. For view v_i, $Z(v_i) = \mathcal{B}(v_i)$. For instance, in Figure 4.10, region I includes the attributes of $\mathcal{F}(w_1)$, region II includes the attributes of $\mathcal{F}(w_2)$, and region III includes the attributes $Z(w_1) = \mathcal{B}(w_1) - \mathcal{F}(w_2)$. Regarding the attributes in $Z(w_j) \cap I(\mathcal{Q})$, their values in the tuple of w_j in step 2 are their initial values in $\mathcal{Q}$. The values of all other attributes in $Z(w_j) - I(\mathcal{Q})$ in the tuples are some new distinct values.

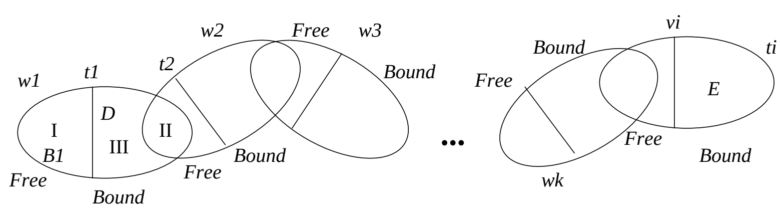


Figure 4.10: Populating the relations on the BF-chain in Figure 4.9.

In step 3, for each source view w_j on the BF-chain, if $Z(w_j) - I(\mathcal{Q})$ is not empty, consider each attribute A in it. By Lemma 4.5.1, there exists a feasible sequence of source views such that A is a free attribute of the tail. Populate the source relations of this sequence following the same idea as in the proof of Theorem 4.4.2, except that the value of A in the tuple of the tail is the value of A in the tuple of w_j that we chose in step 2.

In step 4, consider each attribute in $\mathcal{K} - \{B_1\}$, say B_j. By Lemma 4.5.1, there exists a feasible sequence of source views such that B_j is a free attribute of the tail. Populate the source relations of this sequence following the same idea as in the proof of Theorem 4.4.2, except that the value of B_j in the tuple of the tail is b_j, which is the value that we chose for B_j in step 1.

After these four steps, it is possible that more than one tuple has been added to a source relation. Based on how the relations are populated, all the source views that are populated are in $b\text{-}closure(\mathcal{K}) \cup T$, and we can get the b_i's for the B_i's in the connection by using all these populated source views. Following the same idea of the proof of Theorem 4.4.1, we can prove that by using all these source views, we can get tuple t in the obtainable answer for T.

Note that the only shared values among these source relations are those initial values in $\mathcal{Q}$, those b_i's in connection T, and the values of the attributes in some $Z(w_j) - I(\mathcal{Q})$ during step 2, in which w_j is a source view on the BF-chain from v_i to w_1. Since $\mathcal{K}$ is a kernel of T, B_1 cannot be bound by using only the source views in T starting from the initial bindings in $I(\mathcal{Q})$. On the other hand, without using value b_1 of B_1, we cannot get tuple t in the answer for T. Based on how $\mathcal{R}$ is constructed, the only way to get b_1 of B_1 is to use the source relations on the BF-chain from v_i to w_1, plus those source views that are populated in step 3. Without using view v_i, we cannot get the value b_1 of B_1, so we cannot get tuple t in the answer for T. Therefore, v_i is a relevant source view of connection T. ∎

4.5.4 The Algorithm FIND_REL

Using Theorem 4.5.1, we give the algorithm FIND_REL that finds all the relevant source views of a queryable connection in a query. The algorithm is shown in Figure 4.11.

Algorithm FIND_REL: Find the relevant views of a connection.
Input: • $\mathcal{V}$: Source views with binding restrictions.
• $\mathcal{Q}$: A query.
• T: A queryable connection in $\mathcal{Q}$.
Output: All the relevant views of T.
Method:
(1) Compute all the queryable source views $\mathcal{V}_q = f\text{-}closure(I(\mathcal{Q}), \mathcal{V})$.
(2) Compute a kernel $\mathcal{K}$ of connection T.
(3) Compute the backward-closure $b\text{-}closure(\mathcal{K})$.
(4) Return $b\text{-}closure(\mathcal{K}) \cup T$.

Figure 4.11: The algorithm FIND_REL.

EXAMPLE 4.5.3 In Example 4.4.1, all the five source views are queryable. Connection $T_1 = \{v_1, v_3\}$ is independent, so the only relevant source views of T_1 are v_1 and v_3. Connection $T_2 = \{v_2, v_3\}$ is not independent, and it has only one kernel: $\{C\}$. The backward-closure of the kernel is $\{v_1, v_2, v_4\}$, so only v_1, v_2, v_3, and v_4 are relevant to T_2.

In Example 4.5.1, connection $T = \{v_1, v_2, v_3\}$ has one kernel $\{D\}$, whose backward-closure is $\{v_4\}$. Thus the relevant source views of the connection are v_1, v_2, v_3, and v_4. The connection in Example 4.5.2 has three kernels: $\{A\}$, $\{C\}$, and $\{E\}$. We choose one of them, say $\{A\}$, and compute its backward-closure, which is $\{v_1, v_2, v_3, v_4\}$. Thus all the four views are relevant to the connection. $\Box$

Let us analyze the complexity of the algorithm FIND_REL. Suppose that there are n source views in $\mathcal{V}$. Consider a queryable connection T with m source views and k attributes. Assume it takes $O(1)$ time to check whether a set of attributes is a subset of another set of attributes. As described in Section 4.5.1, we can get all the queryable source views by computing *f-closure*$(I(\mathcal{Q}), \mathcal{V})$. Step 1 thus can be done in $O(n^2)$ time. Step 2 can be done by following the approach described in Section 4.5.2, which shrinks the attributes in $\mathcal{A}(T) - I(\mathcal{Q})$ as much as possible. Since for each set of attributes $X \subseteq \mathcal{A}(T) - I(\mathcal{Q})$, it takes $O(m^2)$ time to compute *f-closure*$(X \cup I(\mathcal{Q}), T)$, step 2 can be done in $O(km^2)$ time.

In step 3, for each attribute A in a kernel $\mathcal{K}$ of T, *b-closure*(A) can be computed in $O(n^2)$ time, where n is the number of views in $\mathcal{V}$. The reason is that during the computation, we can keep a set of attributes $\mathcal{A}_b$ as the union of the $\mathcal{B}(w_i)$'s for each w_i in *b-closure*(A) that has been computed so far. At each step, for each queryable source view v that is not in the current *b-closure*(A), we check whether $\mathcal{F}(v) \cap \mathcal{A}_b$ is not empty. If so, v is added to *b-closure*(A). Thus step 3 can be done in $O(kn^2)$ time. Therefore, the total time complexity of finding the relevant source views of the connection is $O(n^2) + O(km^2) + O(kn^2) = O(k(m^2 + n^2)) = O(kn^2)$.

4.5.5 Constructing an Efficient Program

Given source descriptions $\mathcal{V}$ and a query $\mathcal{Q}$, we can construct an efficient program using Theorem 4.5.1. We first find the relevant views of all the connections in $\mathcal{Q}$ as follows:

1. Compute all the queryable source views $\mathcal{V}_q =$ *f-closure*$(I(\mathcal{Q}), \mathcal{V})$.

2. Remove the nonqueryable connections, i.e., the connections that have a nonqueryable view.

3. Compute the relevant views for each queryable connection by calling the algorithm FIND_REL.

4. Take the union of all these relevant source views.

We then use only these relevant source views (denoted $\mathcal{V}_r$) of query $\mathcal{Q}$ to construct a datalog program $\Pi(\mathcal{Q}, \mathcal{V}_r)$ in the same way as $\Pi(\mathcal{Q}, \mathcal{V})$ is constructed. For instance, in Example 4.4.1, all five source views are queryable. By calling the algorithm FIND_REL we find that views v_1 and v_3 are relevant to connection T_1; views v_1, v_2, v_3, and v_4 are relevant to connection T_2. Therefore, the relevant views for both connections are v_1, v_2, v_3, and v_4. We use these four views to construct a more efficient program, which can be obtained by dropping the rules r_{13} and r_{14} in Figure 4.4.

In addition, some useless rules in the program $\Pi(\mathcal{Q}, \mathcal{V}_r)$ can be removed, since they do not contribute to the answer. For instance, in Example 4.4.1, the user is not interested in the B and E values, so rules r_7 and r_{12} in Figure 4.4 can be dropped. Rules r_9 and r_{10} can also be removed since the predicates in their heads are not used by other rules. Figure 4.12 shows the optimized program that can compute the same answer as before.

$$
\begin{array}{llll}
r_1: & ans(D) & \text{:-} & \widehat{v_1}(a_0, C), \widehat{v_3}(C, D) \\
r_2: & ans(D) & \text{:-} & \widehat{v_2}(a_0, B, C), \widehat{v_3}(C, D) \\
r_3: & \widehat{v_1}(A, C) & \text{:-} & domA(A), v_1(A, C) \\
r_4: & domC(C) & \text{:-} & domA(A), v_1(A, C) \\
r_5: & \widehat{v_2}(A, B, C) & \text{:-} & domC(C), v_2(A, B, C) \\
r_6: & domA(A) & \text{:-} & domC(C), v_2(A, B, C) \\
r_8: & \widehat{v_3}(C, D) & \text{:-} & domC(C), v_3(C, D) \\
r_{11}: & domC(C) & \text{:-} & v_4(C, E) \\
r_{15}: & domA(a_0) & \text{:-} &
\end{array}
$$

Figure 4.12: The optimized datalog program in Example 4.4.1.

In general, we can simplify the program $\Pi(\mathcal{Q}, \mathcal{V}_r)$ as follows. Scan through all the rules in the program $\Pi(\mathcal{Q}, \mathcal{V}_r)$, except for the connection rules. For each rule r, check whether the IDB predicate in its head is used by other rules in the program. If not, rule r is useless and can be removed from the program.

4.6 Testing Containment Between Connections

We have shown so far how to trim useless source accesses for individual connections in a query. In this section, we study the containment problem between two connections in five steps:

1. We formally define connection containment (Section 4.6.1).

2. We prove that connection containment is decidable (Section 4.6.2).

3. We introduce the question of *boundedness* for the program of a connection (Section 4.6.3). When one of the two programs in the containment test is bounded, we can perform the test efficiently.

4. We develop a polynomial-time algorithm for testing connection boundedness (Section 4.6.4).

5. Finally, we compare our decidability proof to the approach of [MLF00] (Section 4.6.5).

4.6.1 Connection Containment

Definition 4.6.1 (**program for a connection**) Let T be a connection in a query $\mathcal{Q}$ on source descriptions $\mathcal{V}$. The *program for connection* T is the program $\Pi(\mathcal{Q}_T, \mathcal{V})$, where $\mathcal{Q}_T$ is the query that has only one connection T, with the input attributes $I(\mathcal{Q})$ and the output attributes $O(\mathcal{Q})$. For any database $\mathcal{D}$ of the source relations, the *ans* facts computed by the program $\Pi(\mathcal{Q}_T, \mathcal{V})$ is the maximal answer to the connection T. □

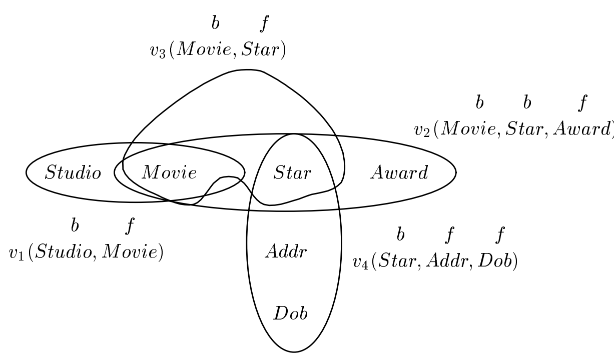


Figure 4.13: The hypergraph representation of four movie sources.

EXAMPLE 4.6.1 Assume we have four sources of movie information as shown in Figure 4.13. Suppose that a user wants to find the addresses and dates of birth of the stars in movies produced by Disney. The query can be represented as

$$\mathcal{Q} = \langle \{Studio\}, \{Addr, Dob\}, \{T_1, T_2\} \rangle$$

in which the two connections are $T_1 = \{v_1, v_2, v_4\}$ and $T_2 = \{v_1, v_3, v_4\}$. Clearly connection T_2 is independent, while T_1 is not. Figure 4.14 shows the corresponding datalog program $\Pi(\mathcal{Q}, \mathcal{V})$.

$$
\begin{array}{lll}
r_1: & ans(A, D) & \text{:- } \widehat{v_1}(disney, M), \widehat{v_2}(M, S, W), \widehat{v_4}(S, A, D) \\
r_2: & ans(A, D) & \text{:- } \widehat{v_1}(disney, M), \widehat{v_3}(M, S), \widehat{v_4}(S, A, D) \\
r_3: & \widehat{v_1}(T, M) & \text{:- } studio(T), v_1(T, M) \\
r_4: & movie(M) & \text{:- } studio(T), v_1(T, M) \\
r_5: & \widehat{v_2}(M, S, W) & \text{:- } movie(M), star(S), v_2(M, S, W) \\
r_6: & award(W) & \text{:- } movie(M), star(S), v_2(M, S, W) \\
r_7: & \widehat{v_3}(M, S) & \text{:- } movie(M), v_3(M, S) \\
r_8: & star(S) & \text{:- } movie(M), v_3(M, S) \\
r_9: & \widehat{v_4}(S, A, D) & \text{:- } star(S), v_4(S, A, D) \\
r_{10}: & addr(A) & \text{:- } star(S), v_4(S, A, D) \\
r_{11}: & dob(D) & \text{:- } star(S), v_4(S, A, D) \\
r_{12}: & studio(disney) & \text{:- }
\end{array}
$$

Figure 4.14: The datalog program $\Pi(\mathcal{Q}, \mathcal{V})$ for the query in Example 4.6.1.

Figure 4.15 shows the program $\Pi(\mathcal{Q}_{T_1}, \mathcal{V})$ for connection T_1. It can be constructed from the program $\Pi(\mathcal{Q}, \mathcal{V})$ by removing the connection rule r_2. That is, it includes the connection rule for T_1, and the α-rules, and the fact rule in $\Pi(\mathcal{Q}, \mathcal{V})$. Similarly, $\Pi(\mathcal{Q}_{T_2}, \mathcal{V})$ can be constructed by removing the connection rule r_1 from the program $\Pi(\mathcal{Q}, \mathcal{V})$ in Figure 4.14.

Although T_1 and T_2 have different views, surprisingly, as we will see in Section 4.6.2, the answer to connection T_1 is *contained* in the answer to connection T_2, i.e., $\Pi(\mathcal{Q}_{T_1}, \mathcal{V}) \sqsubseteq \Pi(\mathcal{Q}_{T_2}, \mathcal{V})$. Therefore, we can compute the answer to the query by considering only connection T_2, and thus save the queries to source S_2. □

$$
\begin{array}{lll}
r_1: & ans(A, D) & \text{:- } \widehat{v_1}(disney, M), \widehat{v_2}(M, S, W), \widehat{v_4}(S, A, D) \\
r_3: & \widehat{v_1}(T, M) & \text{:- } studio(T), v_1(T, M) \\
r_4: & movie(M) & \text{:- } studio(T), v_1(T, M) \\
r_5: & \widehat{v_2}(M, S, W) & \text{:- } movie(M), star(S), v_2(M, S, W) \\
r_6: & award(W) & \text{:- } movie(M), star(S), v_2(M, S, W) \\
r_7: & \widehat{v_3}(M, S) & \text{:- } movie(M), v_3(M, S) \\
r_8: & star(S) & \text{:- } movie(M), v_3(M, S) \\
r_9: & \widehat{v_4}(S, A, D) & \text{:- } star(S), v_4(S, A, D) \\
r_{10}: & addr(A) & \text{:- } star(S), v_4(S, A, D) \\
r_{11}: & dob(D) & \text{:- } star(S), v_4(S, A, D) \\
r_{12}: & studio(disney) & \text{:- }
\end{array}
$$

Figure 4.15: The program $\Pi(\mathcal{Q}_{T_1}, \mathcal{V})$ for the connection T_1 in Example 4.6.1.

Definition 4.6.2 (**connection containment**) Suppose T_1 and T_2 are two connections in a query $\mathcal{Q}$ on source descriptions $\mathcal{V}$. T_1 is *contained* in T_2, denoted $T_1 \subseteq^{ans} T_2$, if the program $\Pi(\mathcal{Q}_{T_1}, \mathcal{V})$ is contained in $\Pi(\mathcal{Q}_{T_2}, \mathcal{V})$ with respect to the goal predicate *ans*, i.e., $\Pi(\mathcal{Q}_{T_1}, \mathcal{V}) \sqsubseteq \Pi(\mathcal{Q}_{T_2}, \mathcal{V})$. □

This connection-containment problem is also called *relative containment* in [MLF00]. For brevity, further on, we use "connection containment" to mean "relative containment." In general, given two connections T_1 and T_2 in a query $\mathcal{Q}$ on source descriptions $\mathcal{V}$, we want to test whether $T_1 \subseteq^{ans} T_2$.

4.6.2 Connection Containment Is Decidable

The program $\Pi(\mathcal{Q}_T, \mathcal{V})$ for a connection T could be recursive (as shown in Example 4.2.1), connection containment appears undecidable, since containment of datalog programs is undecidable [Shm93]. However, we prove connection containment *is* decidable, since it can be reduced to containment of monadic programs. A datalog program is *monadic* if its recursive IDB predicates are monadic (the nonrecursive predicates can have arbitrary arity). An IDB predicate is nonrecursive if it either does not depend on another IDB predicate, or it depends *only* on nonrecursive predicates. Under this definition of nonrecursive IDB predicates, it is shown in [CGKV88] that containment of monadic programs is decidable [Var99].

Theorem 4.6.1 *Connection containment is decidable.* □

The main idea of the proof is as follows. Let $\Pi(\mathcal{Q}_{T_1}, \mathcal{V})$ and $\Pi(\mathcal{Q}_{T_2}, \mathcal{V})$ be the programs for connection T_1 and T_2, respectively. We construct two programs Φ_1 and Φ_2, such that:

1. Both Φ_1 and Φ_2 are monadic programs;

2. $\Pi(\mathcal{Q}_{T_1}, \mathcal{V}) \sqsubseteq \Pi(\mathcal{Q}_{T_2}, \mathcal{V})$ if and only if $\Phi_1 \sqsubseteq \Phi_2$.

Since containment of monadic programs is decidable, the problem of testing $\Pi(\mathcal{Q}_{T_1}, \mathcal{V}) \sqsubseteq \Pi(\mathcal{Q}_{T_2}, \mathcal{V})$ is decidable. The construction of Φ_1 from $\Pi(\mathcal{Q}_{T_1}, \mathcal{V})$ has two steps. (Φ_2 can be constructed from $\Pi(\mathcal{Q}_{T_2}, \mathcal{V})$ similarly.) In step (1), each α-predicate $\widehat{v_i}$ in the connection rule is substituted by the body of the corresponding α-rule of v_i, with the necessary variable unification. After the substitutions, remove the α-rules from $\Pi(\mathcal{Q}_{T_1}, \mathcal{V})$, and the new program, denoted $\Pi(\mathcal{Q}_{T_1}, \mathcal{V})'$, is equivalent to $\Pi(\mathcal{Q}_{T_1}, \mathcal{V})$. Let r_0 be the modified connection rule. For instance, the program $\Pi(\mathcal{Q}_{T_1}, \mathcal{V})$ in Figure 4.15 can be rewritten to the following equivalent program:[2]

$$
\begin{array}{ll}
r_0\colon ans(A, D) & \text{:- } studio(disney), v_1(disney, M), movie(M), star(S), v_2(M, S, W), v_4(S, A, D) \\
r_4\colon movie(M) & \text{:- } studio(T), v_1(T, M) \\
r_6\colon award(W) & \text{:- } movie(M), star(S), v_2(M, S, W) \\
r_8\colon star(S) & \text{:- } movie(M), v_3(M, S) \\
r_{10}\colon addr(A) & \text{:- } star(S), v_4(S, A, D) \\
r_{11}\colon dob(D) & \text{:- } star(S), v_4(S, A, D) \\
r_{12}\colon studio(disney) & \text{:- }
\end{array}
$$

In the new program, ans is the only IDB predicate that might not be unary. It could be recursive, since it might depend on recursive domain predicates. Thus this program is not monadic, and we cannot use the decidability result of monadic programs directly. Suppose the modified connection rule r_0 is:

$$ans(X_1, \ldots, X_m) \text{ :- } dom_1(Y_1), \ldots, dom_k(Y_k), F$$

where $dom_1, \ldots, dom_k$ are unary domain predicates, and F is a set of EDB formulas. In step (2), we construct program Φ_1 by replacing r_0 with the following rule:

$$final(Z) \text{ :- } E_1(X_1, Z), \ldots, E_m(X_m, Z), dom_1(Y_1), \ldots, dom_k(Y_k), F$$

where $final$ is a new IDB predicate representing the answer to the new program, Z is a fresh variable, and $E_1, \ldots, E_m$ are new EDB predicates. For instance, the rule r_0 in the program above is replaced with:

$$
\begin{array}{ll}
r_0 : final(Z) \text{ :- } & E_1(A, Z), E_2(D, Z), studio(disney), v_1(disney, M), movie(M), star(S), \\
& v_2(M, S, W), v_4(S, A, D)
\end{array}
$$

Clearly the program Φ_1 is monadic, since all its IDB predicates are unary. Now we prove that $\Pi(\mathcal{Q}_{T_1}, \mathcal{V}) \sqsubseteq \Pi(\mathcal{Q}_{T_2}, \mathcal{V})$ if and only if $\Phi_1 \sqsubseteq \Phi_2$. The "only if" part is obvious by the construction of the two programs. For the "if" part, suppose $\Phi_1 \sqsubseteq \Phi_2$, but $\Pi(\mathcal{Q}_{T_1}, \mathcal{V}) \not\sqsubseteq \Pi(\mathcal{Q}_{T_2}, \mathcal{V})$. Then there exists a database D, such that there is a tuple $ans(x_1, \ldots, x_m)$

[2]We use this example to illustrate how to construct monadic program Φ_1 for program $\Pi(\mathcal{Q}_{T_1}, \mathcal{V})$, even though the program $\Pi(\mathcal{Q}_{T_1}, \mathcal{V})$ in this example is not recursive. In general, this program can be recursive, as shown in Example 4.2.1.

in $\Pi(\mathcal{Q}_{T_1}, \mathcal{V})(D)$, but not in $\Pi(\mathcal{Q}_{T_2}, \mathcal{V})(D)$. Let z be a fresh constant. We add tuples $E_1(x_1, z), \ldots, E_m(x_m, z)$ to D, and get a new database D'. By the construction of the two programs and D', tuple z is in $\Phi_1(D')$, but not in $\Phi_2(D')$, contradicting the fact that $\Phi_1 \sqsubseteq \Phi_2$.

In conclusion, we can test $\Pi(\mathcal{Q}_{T_1}, \mathcal{V}) \sqsubseteq \Pi(\mathcal{Q}_{T_2}, \mathcal{V})$ by testing $\Phi_1 \sqsubseteq \Phi_2$, which is decidable since both Φ_1 and Φ_2 are monadic programs.

4.6.3 Connection Boundedness

If one of the two programs in testing $\Pi(\mathcal{Q}_{T_1}, \mathcal{V}) \sqsubseteq \Pi(\mathcal{Q}_{T_2}, \mathcal{V})$ is *bounded*, the containment can be tested efficiently using the algorithms in [CM77, CV92, SY80]. A datalog program is *bounded* if it is equivalent to a finite union of conjunctive queries. For instance, the programs for the two queries in Example 4.6.1 are both bounded, because each of them can be rewritten to an equivalent conjunctive query. In this section, we study the following problem: given a connection T on source views $\mathcal{V}$ with binding restrictions, how do we test the boundedness of $\Pi(\mathcal{Q}_T, \mathcal{V})$? We develop a polynomial-time algorithm for testing boundedness of connections. The following example shows an unbounded connection.

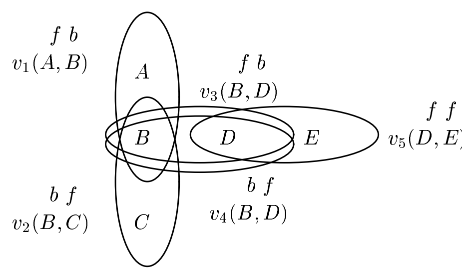


Figure 4.16: The source descriptions in Example 4.6.2.

EXAMPLE 4.6.2 Consider the five source views in Figure 4.16. Assume a user knows the value of A is a, and wants to get the C values by joining the views v_1 and v_2. The following is the query:

$$\mathcal{Q} = \langle \{A\}, \{C\}, \{T\} \rangle$$

in which the only connection is $T = \{v_1, v_2\}$. Assume the five attributes have five different domains. The program $\Pi(\mathcal{Q}_T, \mathcal{V})$ is shown in Figure 4.17. This program is unbounded. Intuitively, since the binding pattern of $v_3(B, D)$ is fb, and the binding pattern of $v_4(B, D)$ is *bf*, we can visit these two source views repeatedly to retrieve more B values. Each new

B value can participate in $v_1 \bowtie v_2$, and generate more answers to the query. (We will give a proof of the unboundedness in Section 4.6.4.) □

r_1:	$ans(C)$	:- $\widehat{v}_1(a, B), \widehat{v}_2(B, C)$		r_8:	$\widehat{v}_4(B, D)$	:- $domB(B), v_4(B, D)$
r_2:	$\widehat{v}_1(A, B)$	:- $domB(B), v_1(A, B)$		r_9:	$domD(D)$	:- $domB(B), v_4(B, D)$
r_3:	$domA(A)$	:- $domB(B), v_1(A, B)$		r_{10}:	$\widehat{v}_5(D, E)$	:- $v_5(D, E)$
r_4:	$\widehat{v}_2(B, C)$	:- $domB(B), v_2(B, C)$		r_{11}:	$domD(D)$	:- $v_5(D, E)$
r_5:	$domC(C)$	:- $domB(B), v_2(B, C)$		r_{12}:	$domE(E)$	:- $v_5(D, E)$
r_6:	$\widehat{v}_3(B, D)$	:- $domD(D), v_3(B, D)$		r_{13}:	$domA(a)$	:-
r_7:	$domB(B)$	:- $domD(D), v_3(B, D)$				

Figure 4.17: The program $\Pi(\mathcal{Q}_T, \mathcal{V})$ in Example 4.6.2.

4.6.4 Testing Connection Boundedness

In this section, we develop a polynomial-time algorithm for testing connection boundedness, even though boundedness of datalog programs in general is undecidable [GMSV93].

Independent Connections

Recall that a connection T in a query $\mathcal{Q}$ is *independent* if *f-closure*$(I(\mathcal{Q}), T) = T$. For instance, the connection T_2 in Example 4.6.1 is independent, while connection T_1 is not. Similarly, the connection in Example 4.6.2 is not independent.

Lemma 4.6.1 *If a connection T is independent, then T is bounded.* □

Proof: Assume $T = \{w_1, \ldots, w_k\}$. Since *f-closure*$(I(\mathcal{Q}), T) = T$, there exists a feasible sequence of the views in connection T, say, $w_{i_1}, \cdots, w_{i_k}$, that satisfies: (i) $\mathcal{B}(w_{i_1}) \subseteq I(\mathcal{Q})$; (ii) for $j = 2, \ldots, k$, $\mathcal{B}(w_{i_j}) \subseteq I(\mathcal{Q}) \cup \mathcal{A}(w_{i_1}) \cup \cdots \cup \mathcal{A}(w_{i_{j-1}})$. For any database of $\mathcal{V}$, we can compute the maximal answer to T as follows. Compute the corresponding sequence of n supplementary relations $S_1, \ldots, S_n$, where S_i is the supplementary relation after the first i subgoals have been processed. The supplementary relation S_n is the answer to query Q. Therefore, we can compute the answer to T after $n+1$ applications of the rules in $\Pi(\mathcal{Q}_T, \mathcal{V})$ (the last application is to evaluate the connection rule). ■

If a connection T is not independent, the predicate *ans* in $\Pi(\mathcal{Q}_T, \mathcal{V})$ may not be bounded, as shown in Example 4.6.2.

BF-loop and BF-graph

Definition 4.6.3 (**BF-loop**) A sequence of views forms a *BF-loop* if it forms a BF-chain, and the bound attributes of the head overlap with the free attributes of the tail. □

For instance, in Figure 4.16, (v_3, v_4) forms a BF-loop, because $\mathcal{F}(v_3) \cap \mathcal{B}(v_4) = \{B\}$ and $\mathcal{F}(v_4) \cap \mathcal{B}(v_3) = \{D\}$.

Definition 4.6.4 (**BF-graph**) The *BF-graph* of a set of source views $\mathcal{W}$ is a directed graph in which each vertex corresponds to a view in $\mathcal{W}$, and there is an edge from vertex v_i to vertex v_j if and only if $\mathcal{F}(v_i) \cap \mathcal{B}(v_j) \neq \phi$. □

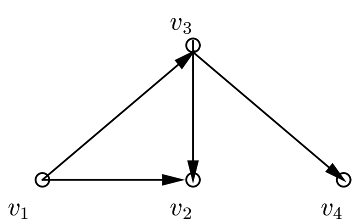


Figure 4.18: BF-graph for Example 4.6.1.

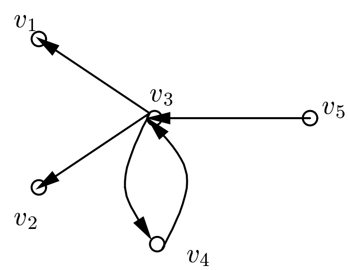


Figure 4.19: BF-graph for Example 4.6.2.

Figures 4.18 and 4.19 show the BF-graphs of the source views in Example 4.6.1 and Example 4.6.2, respectively. For instance, in Figure 4.18, there is an edge from vertex v_1 to vertex v_2 because $\mathcal{F}(v_1) \cap \mathcal{B}(v_2) = \{Movie\}$. Clearly there is a BF-loop in a set of source views if and only if the BF-graph of these views is cyclic.

Algorithm TestBoundedness

Theorem 4.6.2 *If T is a connection in a query $\mathcal{Q}$ on source descriptions $\mathcal{V}$, and all the source views on T are queryable, then T is bounded if and only if there is no BF-loop among the views in b-closure($\mathcal{K}$), in which $\mathcal{K}$ is a kernel of T.* □

Proof: *If*: Assume $T = \{w_1, \ldots, w_n\}$, and *b-closure*$(\mathcal{K}) = \{v_1, \ldots, v_k\}$. Since there is no BF-loop among the views in *b-closure*$(\mathcal{K})$, there exists a BF-chain $v_{i_1}, \ldots, v_{i_k}$ in *b-closure*$(\mathcal{K})$ with distinct views, such that the free attributes of each view v_{i_j} do not overlap with the bound attributes of any previous source view. Starting with the initial bindings in $\mathcal{Q}$ and following the sequence $v_{i_1}, \ldots, v_{i_k}$, we use the views in this sequence to send source queries and retrieve all the possible bindings X of the attributes in $\mathcal{K}$. With these bindings X and the initial bindings in $I(\mathcal{Q})$, there exists a sequence of the views in T, say $w_{l_1}, \ldots, w_{l_n}$, such that the binding requirements of each view in the sequence can be satisfied. We follow this sequence to send source queries, collect tuples from the sources in the connection, and evaluate the connection rule in $\Pi(\mathcal{Q}_T, \mathcal{V})$ to compute the answer to T. Therefore, we can evaluate the rules in a finite number of steps to compute the answer to the connection, and the number is independent of the source relations. Thus T is bounded.

Only If: If there is a BF-loop among the views *b-closure*$(\mathcal{K})$, we prove T is unbounded by showing that for any integer $k > 0$, there exists some database $\mathcal{D}$, such that only after k applications of the rules in $\Pi(\mathcal{Q}_T, \mathcal{V})$ we can compute a tuple in the answer to T. Since there is a BF-loop among *b-closure*$(\mathcal{K})$, there exists an attribute A in $\mathcal{K}$, such that there is a BF-loop among *b-closure*(A). For any integer $k > 0$, there is a BF-chain $v_1, \ldots, v_k$ with length k, and $A \in \mathcal{F}(v_k)$. We can add tuples to the relations on the BF-chain, such that in following the BF-chain only, we can retrieve a tuple in the answer to T. In other words, we "populate" the relations in a BF-loop of the views in *b-closure*$(\mathcal{K})$ along the loop as many times as we want. By the construction of database $\mathcal{D}$, we can only compute a tuple in the answer to T after k applications of the rules in $\Pi(\mathcal{Q}_T, \mathcal{V})$. ∎

Consider the two connections in Example 4.6.1. Connection T_1 has one kernel $\{Star\}$, whose backward-closure is $\{v_1, v_3\}$. Clearly there is no BF-loop in $\{v_1, v_3\}$, thus T_1 is bounded. Similarly, connection T_2 has one kernel ϕ, and there is no BF-loop in its backward-closure, thus T_2 is also bounded. Thus the two programs $\Pi(\mathcal{Q}_{T_1}, \mathcal{V})$ and $\Pi(\mathcal{Q}_{T_2}, \mathcal{V})$ can be rewritten to unbounded queries. In Example 4.6.2, $\{B\}$ is the only kernel of the connection $\{v_1, v_2\}$, and the backward-closure of $\{B\}$ is $\{v_1, v_3, v_4, v_5\}$. Since there is a BF-loop, (v_3, v_4), among these four views, by Theorem 4.6.2, the connection is unbounded. If a connection T is independent, it has only one kernel, the empty set ϕ. Thus the backward-closure of this kernel is empty, and there is no BF-loop among the views in the backward-closure. By Theorem 4.6.2, connection T is bounded, which is consistent with Lemma 4.6.1.

By Theorem 4.6.2, we give an algorithm called *TestBoundedness* for testing connection boundedness, as shown in Figure 4.20.

Let us see the complexity of this algorithm. Assume $\mathcal{V}$ has n views, T has m views and k attributes, and *b-closure*$(\mathcal{K})$ has p views. Section 4.5.4 gives the details how steps 1 to 4 are executed in $O(kn^2)$ time. Steps 5 and 6 can be done in $O(p^2)$ time, since we can test the cyclicity of the BF-graph in $O(p^2)$ time [AHU83]. Therefore, the complexity of the

Algorithm TestBoundedness: Test connection boundedness
Input: • $\mathcal{V}$: Source views with binding restrictions.
• $\mathcal{Q}$: A query on $\mathcal{V}$.
• T: A connection in $\mathcal{Q}$.
Output: Decision about the boundedness of T.
Method:
(1) Compute the queryable views $\mathcal{V}_q = \textit{f-closure}(I(\mathcal{Q}), \mathcal{V})$.
(2) If there is one view $v \in T$ that is not in $\mathcal{V}_q$, T is bounded and return.
(3) Compute a kernel $\mathcal{K}$ of T.
(4) Compute $\textit{b-closure}(\mathcal{K})$.
(5) Build the BF-graph G of $\textit{b-closure}(\mathcal{K})$.
(6) Test the acyclicity of G. If G is acyclic, then T is bounded; otherwise, T is unbounded.

Figure 4.20: Testing the boundedness of a connection.

algorithm is:

$$O(kn^2) + O(p^2) = O(kn^2)$$

4.6.5 Comparison with the Approach in [MLF00]

[MLF00] proved the same decidability result as Theorem 4.6.1. Assume Q_1 and Q_2 are two conjunctive queries on relations with binding restrictions. Let P_1 and P_2 be the datalog programs that compute the maximal answers to P_1 and P_2, respectively. The authors showed that, surprisingly, $P_1^{exp} \sqsubseteq P_2^{exp}$ if and only if $P_1^{exp} \sqsubseteq Q_2$, where P_1^{exp} is the expansion of P_1 (see Section 2.5 for the definition). It is decidable to test $P_1^{exp} \sqsubseteq Q_2$ [CV92].

Readers should compare the two approaches to the decidability result. Besides the fact that we studied this problem in 1999, the following are the differences:

1. [MLF00] uses the source-centric approach to information integration, while we use the query-centric approach. However, our proof can be generalized to the source-centric approach [LC99].

2. The decidability proof in [MLF00] is based on the assumption that the set of bindings for the contained query is a subset of the bindings for the containing query. Theorem 4.6.1 is true even if the two queries have different initial bindings. However, we assume both queries are connection queries, while in [MLF00] the contained query can be a recursive datalog program.

3. We also discuss how to test the boundedness of the program for a query.

4.7 Discussion

In this section we discuss variations of the problem of computing maximal answers to queries.

4.7.1 Other Possibilities for Obtaining bindings

Theorem 4.4.1 suggests that accessing off-connection views is only necessary for nonindependent connections. So far, we have assumed that the bindings of a domain are either from a user query or from other source queries. If cached data is available, it can be incorporated into the program $\Pi(\mathcal{Q}, \mathcal{V})$ for a query $\mathcal{Q}$ and source descriptions $\mathcal{V}$. Suppose that we have a cached tuple $t_i(a_1, \ldots, a_n)$ for source view $v_i(A_1, \ldots, A_n)$. The following rules are added to the program $\Pi(\mathcal{Q}, \mathcal{V})$:

$$\begin{array}{lll} \widehat{v_1}(a_1, \ldots, a_n) & \text{:-} & \\ domA_i(a_i) & \text{:-} & (i = 1, \ldots, n) \end{array}$$

Predicates $domA_1, \ldots, domA_n$ are the domain predicates for the attributes $A_1, \ldots, A_n$, respectively. The first rule says that tuple $t_i(a_1, \ldots, a_n)$ is an obtained tuple of source view v_i. The other fact rules represent the bindings for the corresponding domains. The new rules can contribute more answers to the query. Some views that were nonqueryable when we considered only the initial bindings in $\mathcal{Q}$ may now become queryable with the new bindings from the cached data. The definition of kernel does not change. In general, if we have some information about a domain, we can always incorporate the information into the program $\Pi(\mathcal{Q}, \mathcal{V})$ by adding the corresponding fact rules.

We may also obtain bindings by using some known domain knowledge. For example, suppose that we have a source view $student(name, dept, GPA)$ with the binding pattern bbf. That is, every query to this source must supply a name and a department of a student, so that the student's GPA can be returned. Assume we know that all the students at the source are in four departments: {*CS, EE, Physics, Chemistry*}. Then we can use these four departments as bindings for attribute *dept* to query the source, and we do not need other sources to retrieve *dept* bindings.

4.7.2 Computing a Partial Answer

In some cases a user may be interested in a partial answer to a query. Thus we do not need to compute the maximal answer, which may be expensive to retrieve. Theorem 4.4.1

suggests that if a connection is independent, its complete answer can be computed by using only the views in the query. If a connection T is not independent, we can find a kernel $\mathcal{K}$ of T. We access some sources in *b-closure*($\mathcal{K}$) to obtain bindings for the attributes in $\mathcal{K}$, and compute a partial answer for the connection. Notice that we may access only a subset of the backward-closure of $\mathcal{K}$, since we are not interested in the maximal answer for T. In addition, we need to consider the tradeoff between the number of results and the number of source accesses. The more sources we access, the more bindings we can retrieve, and the more answers we can compute for the connection. We decide how many source queries to send based on how many results the user is interested in.

4.7.3 Extending Results to Conjunctive Queries

Note that a conjunctive query might not be a connection query. The following query:

$$ans(A, B) \text{ :- } r(A, B), r(B, A)$$

is an example. Some results on connection queries in this chapter can be extended to arbitrary conjunctive queries. (1) We can construct a datalog program for a conjunctive query following the idea in Section 4.3.1. That is, we introduce an IDB predicate for each domain that can be shared by several attributes. The idea of using α-predicates and adding rules can be generalized naturally. (2) The decidability result in Section 4.6 can be extended to datalog programs for conjunctive queries.

4.8 Conclusions and Related Work

In this chapter we showed that sources not mentioned in a query can contribute to the query's result by providing useful bindings. We proposed a query-planning framework in the presence of source restrictions. In the framework, a user query and source descriptions are translated into a datalog program, which can be evaluated on the source relations to answer the query. We then solved optimization problems in this framework. In particular, we showed in what cases accessing off-query sources is necessary, and developed an algorithm for finding all useful sources for a query. We also solved the problem of testing whether the answer to a query is contained in the answer to another query. By using the results on monadic programs, we proved this containment problem is decidable. We developed an efficient algorithm for testing program boundedness in our framework. We can perform the containment test efficiently when one of the programs in the test is bounded.

Related Work

Taking the source-centric approach, Rajaraman, Sagiv, and Ullman [RSU95] proposed algorithms for answering queries using views with binding patterns. Duschka and Genesereth [DG97] studied the problem of answering datalog queries using views, and the plan can be a datalog program. Duschka and Levy [DL97] considered source restrictions by translating source binding patterns into rules in a datalog program, assuming that all attributes share the same domain. The paper did not discuss how to trim useless sources, thus it may generate programs that are not efficient to evaluate. Other studies [ASU79a, ASU79b, CM77, CKPS95, SY80] discussed conjunctive-query rewriting and optimization without considering the restrictions of retrieving information from relations.

Taking the query-centric approach, [LYV+98] showed how to generate a feasible plan of a query based on source restrictions. If the complete answer to the query cannot be retrieved, [LYV+98] would not answer the query, but would claim that a feasible plan does not exist. In this case, our approach can still compute a partial answer. Although we take the query-centric approach in this study, our techniques for finding useful sources are also applicable to the source-centric approach, since when source views are the same as global predicates, the query-centric approach in [DL97] and our framework generate equivalent datalog programs. Other related studies include how to optimize conjunctive queries with source restrictions [FLMS99, YLUGM99], how to describe source capabilities using a powerful language [VP97], how to compute mediator capabilities given source capabilities [YLGMU99], and how to convert data at mediators [CDSS98].

Chapter 5

Computing Complete Answers to Queries

In the previous chapter we studied how to compute the maximal answer to a query by using the useful bindings from other relations. In many cases we want to know whether the answer computed by a plan is really the *complete* answer, i.e., the answer that could be computed if we could retrieve all the tuples from all relations. In this chapter we study the following problem: given a query on relations with binding restrictions, can its complete answer be computed? If so, what is the execution plan? Since we can only retrieve *some* tuples from the relations due to their restrictions, we need to do reasoning about the completeness of the answer computed by a plan.

Chapter Organization

In Section 5.1 we motivate the study of this problem using an example. Section 5.2 introduces the notation used throughout the chapter. In particular, we say a query on relations with binding restrictions is *stable* if there exists a plan for the query that computes the query's complete answer independent of the database. Section 5.3 studies stability of conjunctive queries. Section 5.4 shows that for some conjunctive queries, whether their complete answers are computable depend on the relations. Section 5.5 studies stability of unions of conjunctive queries. Section 5.6 studies stability of conjunctive queries with arithmetic comparisons. Section 5.7 studies stability of datalog queries. In Section 5.8 we conclude the chapter and discuss related work.

5.1 Introduction

The following example shows that in some cases the complete answer to a query can be computed.

EXAMPLE 5.1.1 Assume that we have two relations. Relation $r(Star, Movie)$ has information about movies and their stars. Its only binding pattern, *bf*, says that each query to s_1 must specify a star name. Similarly, relation $s(Movie, Award)$ has a binding pattern *bf*. Consider the following query that asks for the awards of the movies in which Henry Fonda starred:

$$Q_1: \ ans(A) \text{ :- } r('Henry\ Fonda', M), s(M, A)$$

To answer Q_1, we first access relation r to retrieve the movies in which Henry Fonda starred. For each returned movie, access relation s to obtain its awards. Return all these awards as the answer to the query. Although only part of the two relations was retrieved, we can still claim that the computed answer is the complete answer. The reason is that all the tuples of relation r that satisfy the first subgoal were retrieved in the first step. Similarly, all the tuples of s that satisfy the second subgoal and join with the results of the first step were retrieved in the second step. □

Query Q_1 is called a "stable" query. A query is *stable* if for any instance of the relations mentioned in the query, the complete answer to the query is computable. That is, there exists a plan such that the answer computed by this plan is guaranteed to be the complete answer to the query. (The formal definition is in Section 5.2.) We show that if a conjunctive query is *feasible* (see Definition 3.2.1 in Section 3.2.1), then the query is stable, and its complete answer can be computed by a linear plan (see Section 3.2.1). In addition, the following example shows that a query can be stable even if it is not feasible.

EXAMPLE 5.1.2 Suppose that we have two relations: $r(A, B, C)$ with a binding pattern bff, and $s(D, E, F)$ with a binding pattern fbb. Consider the following query:

$$Q_2: \ ans(V, X) \text{ :- } r(a, V, Y), r(b, W, U), s(X, V, W), s(X, Z, W)$$

This query is not feasible, since the binding pattern fbb of relation s requires the second argument to be bound, but variable Z in subgoal $s(X, Z, W)$ cannot be bound by other subgoals. However, this subgoal is actually redundant, and Q_2 is equivalent to the following query (i.e., $Q_2' \equiv Q_2$):

$$Q_2' : \; ans(V, X) \;\text{:-}\; r(a, V, Y), r(b, W, U), s(X, V, W)$$

Q_2' is contained in Q_2, i.e., $Q_2' \sqsubseteq Q_2$, because there is a containment mapping from Q_2 to Q_2': $a \rightarrow a$, $b \rightarrow b$, $V \rightarrow V$, $Y \rightarrow Y$, $W \rightarrow W$, $U \rightarrow U$, $X \rightarrow X$, and $Z \rightarrow V$. Similarly, $Q_2 \sqsubseteq Q_2'$ because the identity mapping on subgoals gives us a containment mapping from Q_2' to Q_2. Since Q_2' has a feasible sequence of all its subgoals: $r(a, V, Y)$, $r(b, W, U)$, and $s(X, V, W)$, a linear plan following this sequence (see Section 3.2.1) can compute the complete answer, which is also the complete answer to Q_2. □

Example 5.1.2 suggests that we need to minimize a conjunctive query before checking its feasibility. However, even if the minimal equivalent of a query Q is not feasible, it is still not clear whether the query has an equivalent query that is feasible. Note that there is no a priori limit on the size of an equivalent of query Q, and some equivalents may look quite different from Q. Fortunately, in Section 5.3 we prove that if a minimal conjunctive query is not feasible, then no equivalent of the query can be feasible. We then show that a conjunctive query is stable if and only if its minimal equivalent is feasible. In particular, if its minimal equivalent is not feasible, then there can always be a database, such that the complete answer to the query cannot be computed. We propose two algorithms for testing stability of conjunctive queries, and we prove this problem is $\mathcal{NP}$-complete.

For a nonstable conjunctive query, whether its complete answer can be computed is data dependent. We categorize nonstable conjunctive queries into two classes based on whether the distinguished variables can be bound by the answerable subgoals in a query. We thus develop a decision tree (as shown in Figure 5.6) that guides the planning process to compute the complete answer to a conjunctive query. While traversing the decision tree, two planning strategies — a pessimistic strategy and an optimistic strategy — can be taken.

We then study unions of conjunctive queries, and show similar results as conjunctive queries (Section 5.5). In particular, we need to minimize a union of conjunctive queries before checking its stability. We show that a finite union of conjunctive queries $\mathcal{Q}$ is stable iff each query in the minimal equivalent of $\mathcal{Q}$ is stable. We also propose two algorithms for testing stability of unions of conjunctive queries. For conjunctive queries with arithmetic comparisons, stability testing becomes tricky, since a conjunctive query might not have a minimal equivalent consisting of a subset of its subgoals. We first show that if a query only includes comparisons $\{<, >\}$, we can generalize an algorithm for conjunctive queries for testing its stability. We then give an algorithm for testing stability of conjunctive queries with general arithmetic comparisons, including $\{\leq, \geq, =, \neq\}$.

Finally we study datalog queries, and show that if a set of rules and a query goal have a feasible rule/goal graph [Ull89], then the query is stable. We also show that stability of datalog queries is undecidable.

5.2 Preliminaries

In this section, we introduce the notation used in the chapter. The following observation serves as a starting point of our work.

Observation 1 *If a relation does not have an all-free binding pattern, then after some source queries are sent to the relation, there can always be some tuples in the relation that have not been retrieved, because we did not obtain the necessary bindings to retrieve them.* □

Definition 5.2.1 (**complete answer to a query**) Given a database D of relations with binding restrictions, the *complete* answer to a query Q is $ANS(Q, D)$, i.e., the query's answer that could be computed if we could retrieve all the tuples from the relations. □

Definition 5.2.2 (**stable query**) A query on relations with binding restrictions is *stable* if there exists a plan that accesses the relations using legal patterns, and the plan always computes the query's complete answer independent of the database. □

In this chapter we are especially interested in the two classes of plans: linear plans and exhaustive plans. As defined in Section 3.2.1, a linear plan computes the answer to a query by computing the supplementary relations following a feasible sequence of all the subgoals in the query. We call the plan in Section 4.3 an *exhaustive plan*, since it computes all the obtainable answers to the query by exhaustively retrieving tuples from relations. Here we make the binding assumptions in Section 4.3.2.

In general, an exhaustive plan for a CQ is more expensive than a linear plan, since an exhaustive plan often accesses relations not mentioned in the query to obtain bindings. Thus adding more relations to the database may help compute more results for the query. The previous chapter also discussed how to incorporate cached data into an exhaustive plan by adding the corresponding rules. As we will see in Section 5.4, exhaustive plans are especially useful for nonstable CQ's, since these queries cannot be answered by linear plans. Since the complete answer to a stable query can be computed by a linear plan, an exhaustive plan is not necessary for stable CQ's.

5.3 Stability of Conjunctive Queries

In this section we study stability of CQ's. We propose two algorithms for testing stability of CQ's, and prove that this problem is $\mathcal{NP}$-complete.

5.3.1 Feasible Conjunctive Queries

The following lemma shows that all feasible CQ's are stable.

Lemma 5.3.1 *If a CQ Q on relations with binding restrictions is feasible, then for any database D, $ANS(Q, D)$ can be computed by a linear plan. Thus every feasible CQ is stable.* □

Proof: Let Q have a feasible sequence $g_1(\bar{X}_1), \ldots, g_n(\bar{X}_n)$ of all its subgoals, and L be the corresponding linear plan of this sequence. For each tuple $t \in ANS(Q, D)$, suppose that t comes from the tuples $t_1, \ldots, t_n$ of the relations $g_1, \ldots, g_n$, respectively. For $j = 1, \ldots, n-1$, the tuple $t_1 \bowtie \cdots \bowtie t_j$ is included in the supplementary relation S_j after its values for the irrelevant variables are dropped. This tuple agrees with the tuple t_{j+1} on their common variables. Therefore, during the computation of the supplementary relation S_{j+1} in the plan L, no matter which binding pattern of the relation g_{j+1} is chosen, tuple t_{j+1} in g_{j+1} is retrieved by a source query to relation g_{j+1}. Based on the way S_{j+1} is computed, S_{j+1} also includes the tuple $t_1 \bowtie \cdots \bowtie t_{j+1}$ after the values for the irrelevant variables are dropped. Thus the supplementary relation S_n computed by the plan L includes the tuple t, which can derived from $t_1 \bowtie \cdots \bowtie t_n$ by dropping the values for the nondistinguished variables. ∎

Lemma 5.3.1 shows that the computability of the complete answer to a feasible CQ is *static*, because no matter what the relations mentioned in the query are, the complete answer can be computed by the same linear plan. As we will see in Section 5.4, the computability of the complete answer to a nonstable CQ is *dynamic*, since the computability is unknown until some plan is executed. Recall that the feasibility of a CQ can be checked by the Inflationary algorithm (Section3.2.1).

5.3.2 Minimal Equivalents of CQ's

Example 5.1.2 shows that even if a query is not feasible, it can still be stable, since its minimal equivalent may be feasible. A CQ is *minimal* if it has no redundant subgoals, i.e., removing any subgoal from the query will yield a nonequivalent query. It is known that each CQ has a unique minimal equivalent up to renaming of variables and reordering of subgoals, which can be obtained by deleting its redundant subgoals [CM77]. Thus Lemma 5.3.1 can be strengthened to the following corollary.

Corollary 5.3.1 *If a CQ has a minimal equivalent that is feasible, then the query is stable.* □

However, if the minimal equivalent Q_m of a CQ Q is not feasible, it is still not clear whether there exists an equivalent query that is feasible. In analogy with [RSU95], we might need to consider the possibility that by adding some redundant subgoals to query Q_m, we could have an equivalent query that is feasible. In principle, we have to consider all the equivalents of the query Q to check whether some of them are feasible. Note that there are infinite number of equivalents to a query, and some of them may look quite different from the query. Fortunately, we have the following lemma:

Lemma 5.3.2 *If a minimal CQ is not feasible, then it has no equivalent that is feasible.* □

Proof: Let Q be a minimal CQ that is not feasible. Suppose there is an equivalent CQ P that is feasible, and $\Theta_P = \langle e_1, \ldots, e_m \rangle$ is a feasible sequence of all the subgoals in P. Since the two queries are equivalent, there exist two containment mappings μ: $Q \rightarrow P$, and ν: $P \rightarrow Q$. Consider the targets in Q of the subgoals $e_1, \ldots, e_m$ under the mapping ν: $\nu(e_1), \ldots, \nu(e_m)$. Scan these subgoals from $\nu(e_1)$ to $\nu(e_m)$, and remove the subgoals with identical subgoals earlier in the sequence, and we have a sequence of *some* subgoals in Q: $\Theta_Q = \langle g_1, \ldots, g_n \rangle$, as shown in Figure 5.1. Now we prove that Θ_Q is a feasible sequence of *all* the subgoals in Q. That is, we need to show: (1) Θ_Q includes all the subgoals in query Q; (2) Θ_Q is a feasible sequence. Since query Q is not feasible, we can claim that the equivalent query P actually does not exist.

$$\begin{array}{ll} Q\text{:} & H \text{ :- } g_1(\ldots)\ \&\ \ldots\ \&\ g_t(\ldots,X,\ldots)\ \&\ \ldots\ \&\ g_i(\ldots,X,\ldots)\ \&\ \ldots\ \&\ g_n(\ldots) \\ & \\ P\text{:} & E \text{ :- } e_1(\ldots)\ \&\ \ldots\ \&\ e_j(\ldots,Y,\ldots)\ \&\ \ldots\ \&\ e_{k_i}(\ldots,Y,\ldots)\ \&\ \ldots\ \&\ e_m(\ldots) \end{array}$$

Figure 5.1: Proof of Lemma 5.3.2.

Claim (1) is correct because Q is minimal. If Θ_Q did not include all the subgoals in Q, let Q' be the head of Q and the subgoals in Θ_Q. Then $Q' \sqsubseteq P$ because of the containment mapping ν, and $P \sqsubseteq Q'$ because of the containment mapping μ. Thus Q' is equivalent to Q, and Q could not be minimal.

We now prove claim (2). Consider the first subgoal $g_1 = \nu(e_1)$ in Θ_Q. Since the containment mapping ν maps a variable to a variable or a constant, and maps a constant to the same constant, all the targets of the constant arguments in subgoal e_1 must also be constant arguments in subgoal g_1. Since e_1 is answerable by the relation e_1, subgoal g_1 is also answerable by the relation g_1, which is the same as relation e_1.

Consider each subgoal g_i in the sequence Θ_Q, and let $g_i = \nu(e_{k_i})$ for some $1 \leq k_i \leq m$. Since subgoal e_{k_i} is answerable in Θ_P, there is a binding pattern p of relation e_{k_i}, such that for each argument Y in subgoal e_{k_i} that is adorned b in binding pattern p, either Y is a constant, or Y is a variable bound by a previous subgoal e_j. Consider the argument $X = \nu(Y)$ in subgoal g_i. If Y is a constant, then X is also a constant. If Y is a variable, and X is not a constant, based on how Θ_Q was constructed, there exists a subgoal g_t before g_i in Θ_Q, such that $g_t = \nu(e_j)$. (If $\nu(e_j)$ was removed during the construction of Θ_Q, then g_t is a subgoal identical to $\nu(e_j)$.) Therefore, the variable X is also bound by the subgoal g_t. In summary, X is either a constant or a variable that is bound by a previous subgoal in Θ_Q. Subgoal g_i satisfies the binding requirements of the binding pattern p, and thus it is also answerable by the relation g_i. ∎

Lemma 5.3.3 *If the minimal equivalent of a CQ is not feasible, then the query is not stable.* □

Proof: Assume that query Q has a minimal equivalent Q_m that is not feasible. In order to prove that Q is not stable, we construct two databases, D_1 and D_2, such that

$ANS(Q, D_1) \neq ANS(Q, D_2)$, but D_1 and D_2 have the same observable tuples. Since we cannot tell which database we have by looking at the observable tuples, no plan for the query can guarantee that its computed answer is the complete answer to the query.

Let $X_1, \ldots, X_m$ be all the variables in Q_m. Each variable X_i is assigned a *distinct* value x_i. All the relations are empty initially. For each subgoal g_i in Q_m, add a tuple t_i to its relation. For each argument A in subgoal g_i, if A is a constant c, then the corresponding component in t_i is c. If A is a variable X_j, then the corresponding component in t_i is the distinct value x_j assigned to this variable. Let $\Phi_a = \{g_1, \ldots, g_k\}$ be the set of answerable subgoals of Q_m, and $\Phi_{na} = \{g_{k+1}, \ldots, g_n\}$ be the set of nonanswerable subgoals. Since Q_m is not feasible, Φ_{na} is not empty. Let this substitution turn the head of Q_m to a tuple t_h.

$$
\begin{array}{lll}
Q_m: & H \text{ :- } \overbrace{g_1, \ldots, g_k}^{\Phi_a}, \overbrace{g_{k+1}, \ldots, g_n}^{\Phi_{na}} \\
D_1: & \quad t_1, \ldots, t_k \\
D_2: & \quad t_1, \ldots, t_k, \; t_{k+1}, \ldots, t_n
\end{array}
$$

Figure 5.2: Proof of Lemma 5.3.3.

As shown in Figure 5.2, D_1 is constructed by adding the tuples $t_1, \ldots, t_k$ to the relations $g_1, \ldots, g_k$, respectively; D_2 is constructed by adding all the tuples $t_1, \ldots, t_n$ to the relations $g_1, \ldots, g_n$, respectively.[1] A relation may have multiple tuples, since it may appear in multiple subgoals of Q_m. All the relations that are not mentioned in the query are empty in both databases, so that these relations cannot provide any bindings.

Under both databases we can retrieve tuples $t_1, \ldots, t_k$ following a feasible sequence of the subgoals $g_1, \ldots, g_k$. Under database D_2, we cannot obtain the necessary bindings to retrieve the tuples $t_{k+1}, \ldots, t_n$. Thus D_1 and D_2 have the same observable tuples, i.e., the tuples in D_1. Clearly $t_h \in ANS(Q_m, D_2)$. Now we only need to prove that $t_h \notin ANS(Q_m, D_1)$.[2] Otherwise, there must be a substitution τ from a subset of the obtainable tuples $\{t_1, \ldots, t_k\}$ to all the subgoals $g_1, \ldots, g_n$, such that under τ each subgoal becomes true. Let Q'_m be a query with the head of Q_m plus the subgoals of the tuples used in τ. Since each variable was assigned with a distinct constant, these constants can represent their corresponding variables. Thus τ can be considered to be a containment mapping from Q_m to Q'_m, and $Q_m = Q'_m$. Then Q_m could not be minimal, since it has an equivalent query Q'_m that has

[1] Tuples $t_1, \ldots, t_n$ are called *canonical tuples* of query Q.

[2] Note that this claim might not be correct if Q_m is not minimal.

fewer subgoals. ∎

In general, if we want to prove a query Q is not stable, we need to show two databases D_1 and D_2, such that $ANS(Q, D_1) \neq ANS(Q, D_2)$, but these two databases have the same observable tuples.

5.3.3 Algorithm: CQstable

By Corollary 5.3.1 and Lemma 5.3.3 we have the following theorem.

Theorem 5.3.1 *A CQ is stable if and only if its minimal equivalent is feasible.* □

This theorem gives us an algorithm *CQstable* for testing the stability of a CQ, as shown in Figure 5.3.

Algorithm CQstable: Test stability of CQ's
Input: • Q: A conjunctive query.
• B: Binding restrictions of the relations used in Q.
Output: Decision about the stability of Q.
Method:
(1) Compute the minimal equivalent Q_m of Q by deleting its redundant subgoals.
(2) Based on B, use the algorithm Inflationary to test the feasibility of the query Q_m.
(3) If Q_m is feasible, then Q is stable; otherwise, Q is not stable.

Figure 5.3: Algorithm: CQstable.

Assume that a CQ Q has n subgoals, and its minimal equivalent Q_m has k subgoals. It is known that the minimization of CQ's is $\mathcal{NP}$-complete [CM77], so the first step takes exponential-time in the size of query Q. A number of papers (e.g., [ASU79a, ASU79b, JK83, Sar91]) considered special cases that have polynomial-time algorithms to minimize queries. The complexity of the algorithm Inflationary in the second step is $O(k^2)$. Since $k \leq n$, the total time complexity of the algorithm CQstable is exponential in the size of Q.

5.3.4 Algorithm: CQstable*

The exponential complexity of the algorithm CQstable comes from the fact that we need to minimize a CQ before its feasibility. There is a more efficient algorithm for testing stability of CQ's, which is based on the following theorem:

Theorem 5.3.2 *Let Q be a CQ on relations with binding restrictions. Let Q_a be its* answerable subquery*, i.e., the query that includes the head of Q and the answerable subgoals of Q. Then Q is stable iff $Q \equiv Q_a$.* □

Proof: *If*: Straightforward, since query Q_a is a stable query, and it has a feasible sequence of all its subgoals). *Only if*: The proof is essentially the same as the proof of Lemma 5.3.3, which is correct as long as the following conditions are satisfied: all subgoals in Q'_m are answerable, and $Q'_m \not\equiv Q_m$ as queries. ∎

Theorem 5.3.2 gives another algorithm *CQstable** for testing stability of CQ's, as shown in Figure 5.4.

Algorithm CQstable*: Test stability of CQ's
Input: • Q: A conjunctive query.
• B: Binding restrictions of the relations used in Q.
Output: Decision about the stability of Q.
Method:
(1) Compute the answerable subquery Q_a: use the algorithm Inflationary to find all the answerable subgoals of Q. Let Q_a be the query with these answerable subgoals and the head of Q.
(2) check whether these is a containment mapping from Q to Q_a.
(3) If such a containment mapping exists, then Q is stable; otherwise, Q is not stable.

Figure 5.4: Algorithm: CQstable*.

One advantage of algorithm CQstable* is that we do not need to minimize a CQ Q if all its subgoals are answerable. Note that if Q is stable, its answerable subquery Q_a may properly include the subgoals in Q's minimal equivalent, since some redundant subgoals in Q might be answerable. In addition, the complexity of step 1 is $O(n^2)$, where n is the number of subgoals in Q. However, if not all the subgoals are answerable in step 1, we still need to check the existence of a containment mapping from Q to Q_a.

Another advantage of algorithm CQstable*, as we will see in Section 5.6, is that we can extend it to CQ's with arithmetic comparisons (CQAC's for short). We cannot extend the algorithm CQstable to the case of CQAC's, since a CQAC does not necessarily have a unique minimal form (details in Section 5.6).

5.3.5 Complexity of Testing Stability of CQ's

We might want to find a polynomial-time algorithm for testing stability of CQ's. Unfortunately, the following theorem shows that this problem is $\mathcal{NP}$-complete.

Theorem 5.3.3 *Stability of a CQ is $\mathcal{NP}$-complete.* □

$$\begin{aligned} Q &: \; ans(\bar{X}) \text{ :- } g_1(\bar{X}_1), \ldots, g_k(\bar{X}_k), \ldots, g_n(\bar{X}_n) \\ Q' &: \; ans(\bar{X}) \text{ :- } g_1(\bar{X}_1), \ldots, g_k(\bar{X}_k) \end{aligned}$$

μ

(a) Testing $Q' \sqsubseteq Q$.

$$\begin{aligned} P &: \; ans(\bar{X}) \text{ :- } h(A), g_1(A, \bar{X}_1), \ldots, g_k(A, \bar{X}_k), g_{k+1}(B, \bar{X}_{k+1}), \ldots, g_n(B, \bar{X}_n) \\ P_a &: \; ans(\bar{X}) \text{ :- } h(A), g_1(A, \bar{X}_1), \ldots, g_k(A, \bar{X}_k) \end{aligned}$$

ν

(b) Testing $P_a \sqsubseteq P$.

Figure 5.5: Proof of Theorem 5.3.3.

Proof: Let P be a CQ on relations with binding restrictions. Let P_a be its answerable subquery. Clearly P_a can be computed in polynomial time. By Theorem 5.3.2, we only need to prove that the problem of testing $P \equiv P_a$ is $\mathcal{NP}$-complete. In particular, we need to show that $P_a \sqsubseteq P$ is $\mathcal{NP}$-complete. Clearly this problem is in $\mathcal{NP}$, since given a mapping from P to P_a, it takes polynomial time to verify if this mapping is a containment mapping.

Now we prove this problem is $\mathcal{NP}$-hard by reducing the following $\mathcal{NP}$-complete problem to it. Given a CQ Q and a CQ Q' that is a subset of the subgoals in Q, the problem of deciding whether $Q' \sqsubseteq Q$ is known to be $\mathcal{NP}$-complete [CM77]. Assume we have:

$$Q : ans(\bar{X}) \text{ :- } g_1(\bar{X}_1), \ldots, g_n(\bar{X}_n)$$

$$Q' : ans(\bar{X}) \text{ :- } g_1(\bar{X}_1), \ldots, g_k(\bar{X}_k)$$

where $k < n$. We construct a query P on relations with binding restrictions, such that $Q' \sqsubseteq Q$ iff $P_a \sqsubseteq P$, where P_a is the answerable subquery of P. Figure 5.5 shows how P is constructed. Let A and B be two new variables that do not appear in the subgoals of Q. For each relation g_i, introduce a new relation g_i' with one more attribute than g_i, and g_i' has only one binding pattern *bff…f*. Introduce a new monadic (i.e., 1-ary) relation

h with a binding pattern f. Let P be the query with the same head of Q and subgoals $h(A), g'_1(A, \bar{X}_1), \ldots, g'_k(A, \bar{X}_k)$, $g'_{k+1}(B, \bar{X}_{k+1}), \ldots, g'_n(B, \bar{X}_n)$.

Clearly the above construction of query P takes polynomial time in the size of Q. By the construction of the relations h and $g'_1, \ldots, g'_n$, the answerable subgoals of P are $h(A), g'_1(A, \bar{X}_1), \ldots, g'_k(A, \bar{X}_k)$. Let P_a be the answerable subquery with these answerable subgoals. It is easy to show that there is a containment mapping ν from P to P_a iff there is a containment mapping from Q to Q'. Note that any such containment mapping from P to P' must map both variables A and B to A. ∎

5.4 Nonstable Conjunctive Queries

For a nonstable CQ, in some cases we may still compute its complete answer. However, the computability of its complete answer is data dependent. In this section we discuss in what cases the complete answer to a nonstable CQ may be computed. We develop a decision tree that guides the planning process to compute the complete answer to a CQ. We discuss two planning strategies that can be taken while traversing the tree.

5.4.1 Dynamic Cases

The following example shows that even if a CQ is not stable, its complete answer may still be computable, and we do not know the computability until some plan is executed.

EXAMPLE 5.4.1 Suppose that we have a relation $r(A, B, C)$ with one binding pattern *bff*, a relation $s(C, D)$ with a binding pattern *fb*, and a relation $p(D, E)$ with a binding pattern *ff*. The attributes $A, B, \ldots, E$ have different domains. Consider the following two queries:

$$\begin{array}{lll} Q_1: & ans(B) & \text{:-}\ r(a, B, C), s(C, D) \\ Q_2: & ans(D) & \text{:-}\ r(a, B, C), s(C, D) \end{array}$$

The two queries have the same subgoals but different heads. They are not stable, since their minimal equivalents (themselves) are not feasible. However, we can still try to answer query Q_1 as follows: send a query $r(a, X, Y)$ to relation r. Assume this source query returns three tuples: $\langle a, b_1, c_1\rangle$, $\langle a, b_2, c_2\rangle$, and $\langle a, b_2, c_3\rangle$. The supplementary relation S_1 after the first subgoal has the schema BC, and contains three tuples: $\langle b_1, c_1\rangle$, $\langle b_2, c_2\rangle$, and $\langle b_2, c_3\rangle$.

Although we cannot use the new bindings $\{c_1, c_2, c_3\}$ for attribute C to query relation s directly due to its *fb* binding pattern, we may still use an exhaustive plan to retrieve tuples from relation s, e.g., using the D bindings provided by relation p, although p is not mentioned in the query.

If the exhaustive plan retrieves two tuples $\langle c_1, d_1\rangle$ and $\langle c_2, d_2\rangle$ from relation s, we can still claim that the complete answer is $\{b_1, b_2\}$. The reason is that the only distinguished variable B is bound by the supplementary relation S_1. Tuples $\langle b_1, c_1\rangle$, $\langle b_2, c_2\rangle$, and $\langle b_2, c_3\rangle$ are the only tuples in S_1, and their projection onto variable B is $\{b_1, b_2\}$. Tuples $\langle b_1, c_1\rangle$ and $\langle b_2, c_2\rangle$ in S_1 can join with tuples $\langle c_1, d_1\rangle$ and $\langle c_2, d_2\rangle$ in s, respectively, and their projection onto the variable B is also $\{b_1, b_2\}$. Thus, this projection is the complete answer. On the other hand, if the exhaustive plan retrieves only one tuple $\langle c_2, d_2\rangle$ from relation s, then we do not know whether $\langle b_2\rangle$ is the complete answer or not, since we do not know whether relation s has a tuple $\langle c_1, d\rangle$ (for some d value) that has not been retrieved. This tuple can join with the tuple $\langle b_1, c_1\rangle$ in S_1 to produce the value b_1 as an answer.

We can also try to answer query Q_2 in the same way. After the first subgoal is solved, the supplementary relation S_1 also includes three tuples $\langle b_1, c_1\rangle$, $\langle b_2, c_2\rangle$, and $\langle b_2, c_3\rangle$. However, even if an exhaustive plan is executed to retrieve tuples from relation s, we can never know the complete answer to Q_2, since there can always be a tuple $\langle c_1, d'\rangle$ in relation s that has not been retrieved, and d' is in the complete answer to Q_2. For both queries Q_1 and Q_2, if the supplementary relation S_1 is empty, then we can claim that their complete answers are both empty. □

An important observation on the two queries is that in query Q_1, the distinguished variable B can be bound by the answerable subgoal $r(a, B, C)$, while in query Q_2, the distinguished variable D cannot be bound by the answerable subgoal $r(a, B, C)$. In general, if a minimal CQ Q_m is not stable, we can use the algorithm Inflationary to find all its answerable subgoals Φ_a. If all the distinguished variables can be bound by Φ_a, i.e., the answerable subquery of Q_m is safe, we use a linear plan of a feasible sequence of Φ_a to compute the supplementary relation (denoted I_a) of these subgoals. There are two cases:

1. If I_a is empty, then the complete answer to the query is empty.
2. If I_a is not empty, let I_a^P be the projection of I_a onto the distinguished variables. Execute an exhaustive plan to retrieve tuples for the nonanswerable subgoals Φ_{na}.

(a) If for every tuple t^P in I_a^P, there is a tuple t_a in I_a, such that the projection of t_a onto the distinguished variables is t^P, and t_a can join with some tuples for all the subgoals Φ_{na} (tuple t^P is then called *satisfiable*), then I_a^P is the complete answer to the query.

(b) Otherwise, we do not know the complete answer to the query.

If not all the distinguished variables are bound by the answerable subgoals Φ_a, i.e., the answerable subquery of Q_m is not safe, then the complete answer is not computable, unless the supplementary relation I_a is empty. The following lemmas prove the claims above.

Lemma 5.4.1 *For a minimal CQ Q_m, if the supplementary relation I_a of the answerable subgoals Φ_a is empty, then the complete answer to the query is empty.* □

Proof: Otherwise, if the complete answer has a tuple t, consider the tuples for the answerable subgoals Φ_a that contribute to the tuple t. By using a linear plan of a feasible sequence of Φ_a, we can retrieve these tuples. Therefore, the join of these tuples, with the values for the irrelevant attributes dropped, must be in the supplementary relation I_a, which cannot be empty. ■

Lemma 5.4.2 *Assume that Q_m is a minimal CQ, and all the distinguished variables are bound by the answerable subgoals Φ_a. Let I_a^P be the projection of I_a onto the distinguished variables. (1) If every tuple t^P in I_a^P is satisfiable, then I_a^P is the complete answer to the query. (2) Otherwise, the complete answer is not computable.* □

Proof: (1) Let t be a tuple in the complete answer, and suppose t comes from tuples $t_1, \ldots, t_k$ of the answerable subgoals Φ_a and tuples $t_{k+1}, \ldots, t_n$ of the nonanswerable subgoals Φ_{na}. Tuples $t_1, \ldots, t_k$ must be retrieved by a linear plan of a feasible sequence of Φ_a during the computation of I_a. The projection of $t_1 \bowtie \cdots \bowtie t_k$ onto the distinguished variables is tuple t, since the distinguished variables are all bound by the subgoals Φ_a. Thus tuple t is in I_a^P. On the other hand, since every tuple t^P in I_a^P is satisfiable, t^P is in the answer to the query. Therefore, I_a^P is the complete answer.

(2) Let t^P be a tuple in I_a^P that is not satisfiable. Following the idea of the proof of Lemma 5.3.3, there can always be some tuples for the nonanswerable subgoals Φ_{na} that can join with a tuple in I_a that produces t^P, such that t^P is a tuple in the complete

answer. However, these tuples for Φ_{na} cannot be retrieved because of the restrictions of Φ_{na}. Without these tuples, t^P is not in the complete answer. Since we do not know whether these tuples for Φ_{na} exist or not, we do not know whether the complete answer includes t^P or not. ∎

Lemma 5.4.3 *For a minimal CQ Q_m, if not all the distinguished variables are bound by the answerable subgoals Φ_a, and the supplementary relation I_a is not empty, then the complete answer is not computable.* □

Proof: Let t_a be a tuple in I_a, and v be a distinguished variable that cannot be bound by Φ_a. Following the idea of the proof of Lemma 5.3.3, there can always be some tuples for the nonanswerable subgoals Φ_{na} that can join with the tuple t_a, such that the projection r of the join onto the distinguished variables (including v) is in the complete answer. However, these tuples cannot be retrieved because of the restrictions of Φ_{na}. Without these tuples, tuple r is not in the complete answer. Since we do not know whether these tuples for Φ_{na} exist or not, we do not know whether the complete answer includes tuple r or not. ∎

To summarize, whether the complete answer to a nonstable CQ is computable is dynamic or data dependent, since it is not known until some plan is executed, and some information about the relations becomes available.

5.4.2 The Decision Tree

We develop a decision tree (as shown in Figure 5.6) that guides the planning process to compute the complete answer to a CQ. The shaded nodes are where we can conclude about whether the complete answer is computable or not. Now we explain the decision tree in details. We first minimize a CQ Q by deleting its redundant subgoals, and compute its minimal equivalent Q_m (arc 1 in Figure 5.6). Then we test the feasibility of the query Q_m by calling the algorithm Inflationary; that is, we test whether Q_m has a feasible sequence of all its subgoals. If so (arc 2 in Figure 5.6), Q_m (thus Q) is stable, and its answer can be computed by a linear plan following a feasible sequence of all the subgoals in Q_m.

If Q_m is not feasible (arc 3), we compute all its answerable subgoals Φ_a by calling the algorithm Inflationary. Then we check if all the distinguished variables are bound by the subgoals Φ_a. There are two cases:

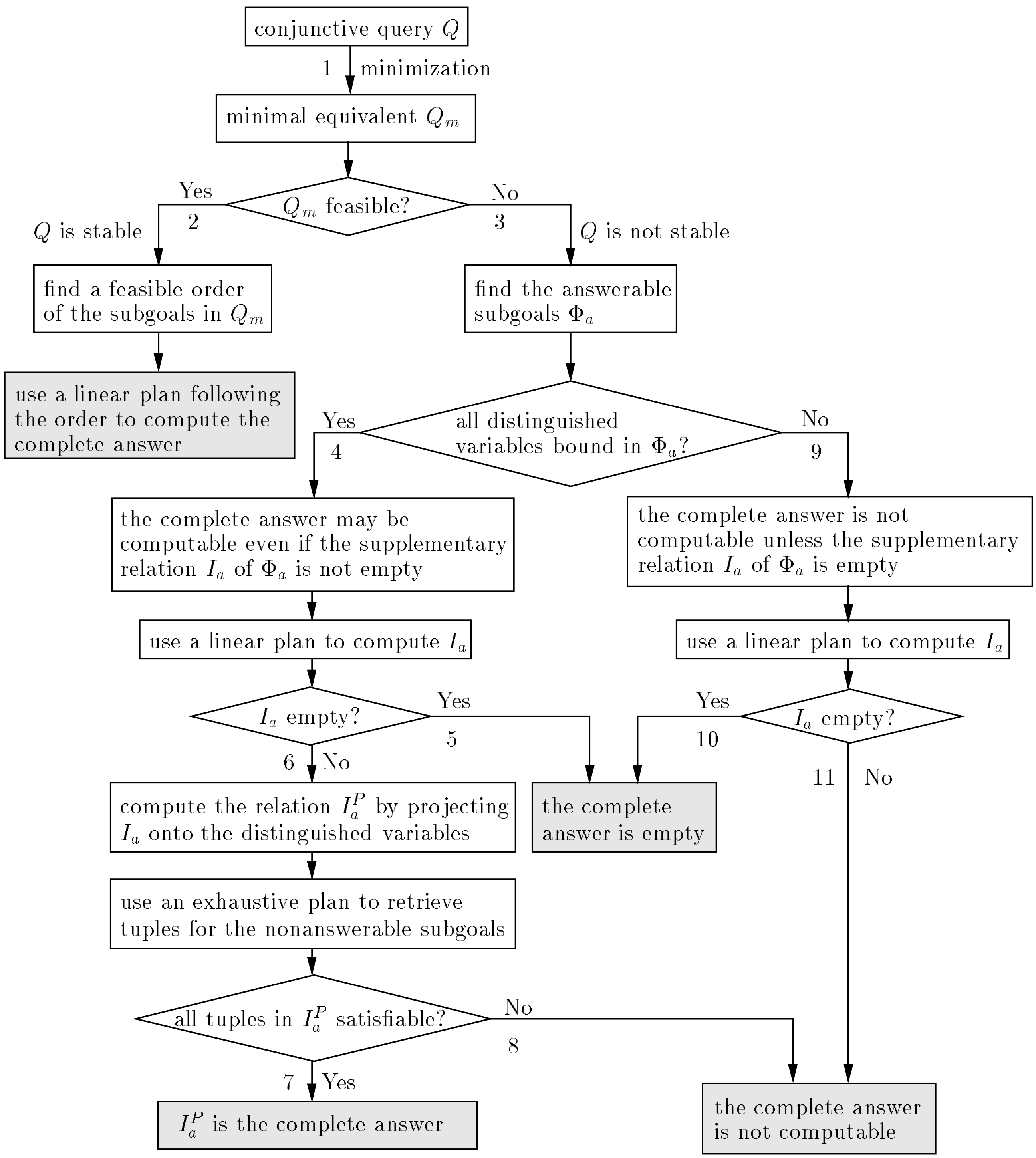


Figure 5.6: The decision tree of computing the complete answer to a CQ.

1. If all the distinguished variables are bound by the subgoals Φ_a (arc 4), then the complete answer to the query may be computed even if the supplementary relation I_a of subgoals Φ_a is not empty. We compute I_a by a linear plan following a feasible sequence of Φ_a.

   (a) If I_a is empty (arc 5), then the complete answer is empty.

   (b) If I_a is not empty (arc 6), we compute the relation I_a^P by projecting I_a onto the distinguished variables. We use an exhaustive plan to retrieve tuples for the nonanswerable subgoals Φ_{na}, and check whether all the tuples in I_a^P are satisfiable. If so (arc 7), then I_a^P is the complete answer. If not (arc 8), then the complete answer is not computable.

2. If some distinguished variables are not bound by the subgoals Φ_a (arc 9), then the complete answer is not computable, unless the supplementary relation I_a is empty. Similarly to the case of arc 4, we compute I_a by a linear plan. If I_a is empty (arc 10), then the complete answer is empty. Otherwise (arc 11), the complete answer is not computable.

While traversing the tree, if we reach a node where the complete answer is unknown, we still have some information about the lower bound and the upper bound of the answer. For instance, if arc (8) is reached, then the upper bound of the answer is I_a^P (i.e., the answer can only be a subset of I_a^P), and the lower bound is all the satisfiable tuples in I_a^P. If arc (11) is reached, the answer has the lower bound ϕ, while it has no upper bound. In the shaded nodes where we can compute the complete answer, the lower bound and upper bound converge. We can tell the user the information about the lower bound and upper bound for decision support and analysis by the user.

5.4.3 Pessimistic Planning and Optimistic Planning

While traversing the decision tree, we may reach a node where we do not know whether the complete answer is computable until we traverse one level down the tree. Two planning strategies can be adopted at this kind of nodes: a pessimistic strategy and an optimistic strategy. A *pessimistic* strategy gives up traversing the tree once the complete answer is unlikely to be computable. On the contrary, an *optimistic* strategy is optimistic about the possibility of computing the complete answer, and it traverses one more level by doing the corresponding operations.

For instance, if the minimal equivalent Q_m is not feasible, and all the distinguished variables are bound by the answerable subgoals Φ_a, we do not know whether the complete answer is computable before using a linear plan to compute the supplementary relation I_a and checking the emptiness of I_a. A pessimistic strategy gives up the planning process, but claims that the complete answer cannot be computed. An optimistic strategy continues traversing the tree by executing a linear plan to compute I_a. If I_a is empty, the complete answer is empty. Otherwise, we still have two options: a pessimistic strategy gives up answering the query, while an optimistic strategy executes an exhaustive plan to retrieve tuples for the nonanswerable subgoals Φ_{na}.

What strategy should be taken is application dependent. For instance, we should consider how "eager" the user is for the complete answer to a query, how expensive a linear plan and an exhaustive plan are, how likely the supplementary relation I_a is to be empty, and how likely it is that all the tuples in I_a^P are satisfiable. We may use statistics to answer these questions and then make the decision about what strategy to take.

5.5 Stability of Unions of Conjunctive Queries

In this section we study stability of unions of CQ's, and present similar results as CQ's. In particular, a union of CQ's is stable iff each query in its minimal equivalent is stable. We also propose two algorithms for testing of stability of unions of CQ's. Let $\mathcal{Q} = Q_1 \cup \cdots \cup Q_n$ be a finite union of conjunctive queries (UCQ for short), all of which have a common head predicate. It is known that there is a unique minimal subset of $\mathcal{Q}$ that is its minimal equivalent [SY80]. The following theorem is from [SY80]:

Theorem 5.5.1 *Let $\mathcal{Q} = Q_1 \cup \cdots \cup Q_m$ and $\mathcal{R} = R_1 \cup \cdots \cup R_n$ be two UCQ's. Then $\mathcal{Q} \sqsubseteq \mathcal{R}$ (i.e., $\mathcal{Q}$ is contained in $\mathcal{R}$ as queries) iff for any query Q_i in $\mathcal{Q}$, there is a query R_j in $\mathcal{R}$, such that $Q_i \sqsubseteq R_j$.* □

EXAMPLE 5.5.1 Let us see some examples of UCQ's and their stability. Suppose that we have three relations r, s, and p, and each relation has only one binding pattern *bf*. Consider the following three queries:

$$
\begin{array}{lll}
Q_1: & ans(X) & \text{:-}\ r(a,X) \\
Q_2: & ans(X) & \text{:-}\ r(a,X), p(Y,Z) \\
Q_3: & ans(X) & \text{:-}\ r(a,X), s(X,Y), p(Y,Z)
\end{array}
$$

Clearly $Q_3 \sqsubseteq Q_2 \sqsubseteq Q_1$. Queries Q_1 and Q_3 are both stable (since they are both feasible), while query Q_2 is not. Consider the following two UCQ's: $\mathcal{Q}_1 = Q_1 \cup Q_2 \cup Q_3$ and $\mathcal{Q}_2 = Q_2 \cup Q_3$. $\mathcal{Q}_1$ has a minimal equivalent Q_1, and $\mathcal{Q}_2$ has a minimal equivalent Q_2. Therefore, query $\mathcal{Q}_1$ is stable, and $\mathcal{Q}_2$ is not. □

5.5.1 Algorithm: UCQstable

In analogy with Theorem 5.3.1, we can prove a theorem that gives us a stability test for UCQ's.

Theorem 5.5.2 *Let $\mathcal{Q}$ be a UCQ on relations with binding restrictions. $\mathcal{Q}$ is stable iff each query in the minimal equivalent of $\mathcal{Q}$ is stable.* □

Proof: Let $\mathcal{Q} = Q_1 \cup \cdots \cup Q_n$. Without loss of generality, assume that the minimal equivalent $\mathcal{Q}_m$ has k queries, $Q_1, \ldots, Q_k$, where $k \leq n$.

If: Straightforward, since for any database D, we can compute $ANS(\mathcal{Q}, D)$ by computing $ANS(Q_i, D)$ for each Q_i $(1 \leq i \leq k)$, and taking the union of these answers.

Only If: If $\mathcal{Q}_m$ has a query, say Q_1, that is not stable. Without loss of generality, suppose that subgoals

$$g_1(\bar{X}_1), \ldots, g_p(\bar{X}_p)$$

are the answerable subgoals of query Q_1, and subgoals

$$g_{p+1}(\bar{X}_{p+1}), \ldots, g_q(\bar{X}_q)$$

are its nonanswerable subgoals. Let Q_1' be the answerable subquery of Q_1 with the p answerable subgoals. By Theorem 5.3.2, there is at least one nonanswerable subgoal (i.e., $p < q$), and $Q_1' \not\equiv Q_1$. Consider the canonical tuples $t_1, \ldots, t_k$ of query Q_1 (see Section 5.3 for the definition of canonical tuples). Let Q_1' be the query with the head of Q_1 and the answerable subgoals of Q_1. Let these tuples tuples turn the head of Q_1 to a tuple t_h.

Following the same idea in the proof of Theorem 5.3.2, to prove $\mathcal{Q}$ is not stable, we need to prove that given the obtainable tuples $t_1, \ldots, t_p$, the answer to $\mathcal{Q}$ (i.e., the answer to $\mathcal{Q}_m$) does not include tuple t_h. Suppose that the answer to $\mathcal{Q}_m$ includes tuple t_h. By

Theorem 5.3.2, t_h cannot come from query Q_1, since otherwise there will be a containment mapping from Q'_1 to Q_1, contradicting the fact that $Q_1 \not\equiv Q'_1$. Therefore, tuple t_h must be derived from another query Q_j in $\mathcal{Q}_m$. By the construction of the canonical tuples $t_1, \ldots, t_p$, it is easy to argue that Q_j produces t_h only if there is a symbol mapping from the arguments of Q_j to these p tuples. This mapping also serves as a containment mapping from Q_j to Q'_1. Thus, $Q'_1 \sqsubseteq Q_j$, and $Q_1 \sqsubseteq Q'_1 \sqsubseteq Q_j$. Then $\mathcal{Q}_m$ cannot be minimal, since it has a redundant query Q_1. ∎

Theorem 5.5.2 gives an algorithm *UCQstable* for testing stability of UCQs, as shown in Figure 5.7.

Algorithm UCQstable: Test stability of unions of conjunctive queries
Input: • $\mathcal{Q}$: A finite union of conjunctive queries.
• B: Binding restrictions of the relations used in $\mathcal{Q}$.
Output: Decision about the stability of $\mathcal{Q}$.
Method:
(1) Compute the minimal equivalent $\mathcal{Q}_m$ of $\mathcal{Q}$ by deleting its redundant queries.
(2) For each $Q_i \in \mathcal{Q}_m$:
• Use the algorithm CQstable or algorithm CQstable* to test the stability of Q_i;
• If Q_i is not stable, then query $\mathcal{Q}$ is not stable.
(3) Query $\mathcal{Q}$ is stable.

Figure 5.7: Algorithm: UCQstable.

5.5.2 Algorithm: UCQstable*

Similar to Theorem 5.3.2, we have the following theorem.

Theorem 5.5.3 *Let $\mathcal{Q}$ be a UCQ on relations with binding restrictions. Let $\mathcal{Q}_s$ be the union of all the stable queries in $\mathcal{Q}$. Then $\mathcal{Q}$ is stable iff $\mathcal{Q} \equiv \mathcal{Q}_s$, i.e., $\mathcal{Q}$ and $\mathcal{Q}_s$ are equivalent as queries.* □

Proof: *If*: Straightforward. If each CQ in $\mathcal{Q}_s$ is stable, for any database D, we can compute $ANS(\mathcal{Q}, D)$ by computing $ANS(Q_i, D)$ for each query $Q_i \in \mathcal{Q}_s$, and taking the union of these answers.

Only If: Suppose that $\mathcal{Q} \not\equiv \mathcal{Q}_s$, i.e., there is a nonstable query Q_u in $\mathcal{Q} - \mathcal{Q}_s$, such that Q_u is not contained in any query in $\mathcal{Q}_s$. Since containment between CQ's is transitive, there must be a query Q_i in $\mathcal{Q} - \mathcal{Q}_s$, such that Q_i is not contained in any query in $\mathcal{Q}_s$, and there is no query Q in $\mathcal{Q} - \mathcal{Q}_s$ such that $Q_i \sqsubset Q$ (i.e., $Q_i \sqsubseteq Q$ but $Q_i \not\equiv Q$). (Q_i can either be Q_u, or can be found by searching all the queries in $\mathcal{Q} - \mathcal{Q}_s$ that contain Q_u, and choosing the most containing one.)

Let Q_i' be the query with the answerable subgoals of Q_i. Since Q_i is not stable, by Theorem 5.5.3, $Q_i' \not\equiv Q_i$. Consider the canonical tuples for the subgoals in Q_i. Assume that these tuples turn the head of Q_i to a tuple t_h. Following the same idea in the proof of Theorem 5.5.2, to prove that $\mathcal{Q}$ is not stable, we need to prove that given the obtainable tuples for the answerable subgoals in Q_i, we cannot compute the tuple t_h. Otherwise, there can be three cases:

1. Tuple t_h is from Q_i, which can not be true by Theorem 5.3.2.

2. t_h is from a query Q_k in $\mathcal{Q}_s$. Then there is a symbol mapping from the arguments of Q_k to the obtainable tuples. This mapping also serves as a containment mapping from Q_k to Q_i'. Therefore, $Q_i \sqsubseteq Q_i' \sqsubseteq Q_k$, contradicting the fact that Q_i is not contained in any CQ in $\mathcal{Q}_s$.

3. t_h is from a query Q_j in $\mathcal{Q} - \mathcal{Q}_s$. Then $Q_i \sqsubset Q_i' \sqsubseteq Q_j$, contradicting the fact that no query in $\mathcal{Q} - \mathcal{Q}_s$ can properly contain Q_i.

∎

Theorem 5.5.3 gives another algorithm *UCQstable** for testing stability of UCQs, as shown in Figure 5.8. The advantage of this algorithm is that we might avoid testing the equivalence between the query $\mathcal{Q}_s$ and $\mathcal{Q}$ if all the queries in $\mathcal{Q}$ are stable.

Algorithm UCQstable*: Test stability of unions of conjunctive queries
Input: • $\mathcal{Q}$: A finite union of conjunctive queries.
• B: Binding restrictions of the relations used in $\mathcal{Q}$.
Output: Decision about the stability of $\mathcal{Q}$.
Method:
(1) Compute all the stable queries:
- $\mathcal{Q}_s = \phi$
- For each query Q_i in $\mathcal{Q}$:
  (a) Call the algorithm CQStable or algorithm CQStable* to test the stability of Q_i;
  (b) If Q_i is stable, add Q_i to $\mathcal{Q}_s$;

(2) Test whether $\mathcal{Q} \sqsubseteq \mathcal{Q}_s$ as queries;
(3) If $\mathcal{Q} \sqsubseteq \mathcal{Q}_s$, then query $\mathcal{Q}$ is stable; otherwise, query $\mathcal{Q}$ is not stable..

Figure 5.8: Algorithm: UCQstable*.

One corollary of Theorem 5.5.2 and Theorem 5.5.3 is that stability of bounded datalog queries [CGKV88] is decidable, because by definition, a bounded datalog program is equivalent to a UCQ. We can test the stability of a bounded datalog query by testing the stability its equivalent UCQ. It is known that boundedness of datalog programs is undecidable [GMSV93]. Several papers (e.g., [Ioa85, NS87, Sag85]) gave algorithms for detecting boundedness in several classes of datalog queries. Another corollary of the two theorems is that stability of a query with conjuncts and disjuncts is also decidable, since such a query can be translated into an equivalent UCQ. For instance, suppose we have a query with a condition

$$((Author = smith) \vee (Year = 1999)) \wedge (Subject = database)$$

we can rewrite the condition to a disjunctive form:

$$((Author = smith) \wedge (Subject = database)) \vee ((Year = 1999) \wedge (Subject = database))$$

Therefore, we test the stability of the original query by testing the stability of its corresponding UCQ.

5.6 Stability of Conjunctive Queries with Comparisons

In this section we study stability of CQ's with arithmetic comparisons (CQAC's for short). Let Q be a CQAC. Let $O(Q)$ be the set of ordinary (uninterpreted) subgoals of Q that do not have comparisons. Let $\mathcal{C}(Q)$ be the set of its subgoals that are arithmetic comparisons. We consider the following arithmetic comparisons: $<$, $\leq$, $=$, $>$, $\geq$, and $\neq$. In addition, we make the following assumptions about the comparisons:

1. Values for the arguments in the comparisons are chosen from an infinite, totally ordered set, such as the rationals or reals.

2. The comparisons are not contradictory, i.e., there exists an instantiation of the variables such that all the comparisons are true. All the comparisons safe, i.e., each variable in the comparisons appears in some ordinary subgoal.

We might be tempted to generalize the algorithm CQstable to test the stability of a CQAC. However, the following example from [Gup94] shows that a CQAC may not have a minimal equivalent that has a subset of its subgoals.

EXAMPLE 5.6.1 Consider the query

$$ans(X, Y) \text{ :- } p(X, Y), X \neq Y, X \leq Y$$

Clearly the query is not equivalent to the query formed from any subset of its subgoals, However,

$$ans(X, Y) \text{ :- } p(X, Y), X < Y$$

is an equivalent query with fewer subgoals. □

5.6.1 Answerable Subquery of a CQAC

Definition 5.6.1 (answerable subquery of a CQAC) Let Q be a CQAC on relations with binding restrictions. Its *answerable subquery*, denoted by Q_a, is the query that includes the head of Q, the answerable subgoals of Q, and all the comparisons of the bound variables that can be derived from $\mathcal{C}(Q)$. □

The answerable subquery Q_a of a CQAC Q can be computed as follows. First derive all the equalities from $\mathcal{C}(Q)$. That is, if Q contains equalities such as $X = Y$, or equalities that can be derived from inequalities (e.g., if we can derive $X \leq Y$ and $X \geq Y$, then we know $X = Y$), then we substitute variable X by Y. Then using the binding restrictions of the relations, compute all the answerable ordinary subgoals $\mathcal{A}(Q)$ of query Q using the Inflationary algorithm. Let $\mathcal{V}$ be the set of all the bound variables in $\mathcal{A}(Q)$. Derive all the inequalities $\mathcal{I}$ among the variables in $\mathcal{V}$ from $\mathcal{C}(Q)$. Q_a includes *all* the constraints of the variables in $\mathcal{V}$ that can be derived from $\mathcal{C}(Q)$. For instance, assume variable X is bound, and variable Y is not. If Q has comparisons $X < Y$ and $Y \leq 5$, then variable X in Q_a still needs to satisfy the constraint $X < 5$.

We might want to generalize the algorithm CQstable* as follows. Given a CQAC Q, we compute its answerable subquery Q_a. We test the stability of Q by testing whether $Q_a \sqsubseteq Q$, which can be tested using the algorithms in [GSUW94, ZO93] ("the GZO algorithm" for short). However, the following example shows that this "algorithm" does not always work.

EXAMPLE 5.6.2 Consider query

$$P : ans(Y) \text{ :- } p(X), r(X, Y), r(A, B), A < B, X \leq A, A \leq Y$$

where relation p has a binding pattern f, and relation r has a binding pattern bf. In the first step of the algorithm, we find all the answerable subgoals $p(X)$ and $r(X, Y)$ of query

P. With variables X and Y bound, we derive all the possible constraints these two variables must satisfy from the comparisons in P. The only derived comparison is $X \leq Y$. Thus we get a new query

$$ans(Y) \text{ :- } p(X), r(X, Y), X \leq Y$$

Using the GZO algorithm we know that $P_a \not\sqsubseteq P$. Therefore, we may claim that query P is not stable. However, actually query P *is* stable. As we will see in Section 5.6.3, query P is equivalent to the union of the following two queries. (Note that all the ordinary subgoals in these two queries are answerable.)

$$\begin{array}{lll} T_1: & ans(Y) & \text{:- } p(X), r(X, Y), X < Y \\ T_2: & ans(Y) & \text{:- } p(Y), r(Y, Y), r(Y, B), Y < B \end{array}$$

$\Box$

The reason the above "algorithm" fails is that, the only case where $P_a \not\sqsubseteq P$ is when $X = Y$. However, comparisons $X \leq A$ and $A \leq Y$ will then force A to be equal to X and Y, and the subgoal $r(A, B)$ becomes answerable. This example suggests that we need to use the idea in [Klu88] to test stability of CQAC's. That is, we need to consider *all* the total orders of the variables in the query.[3]

5.6.2 Algorithm: CQAC1stable

Before giving the algorithm for testing the stability of any CQAC, we first consider the case where the above "algorithm" works. It turns out that the above "algorithm" is correct when a CQAC does not include comparisons $\{\leq, \geq, =, \neq\}$. Figure 5.9 shows an algorithm *CQAC1stable* that tests stability of CQAC's without nonstrict comparisons $\{\leq, \geq, =, \neq\}$, i.e., their comparisons can only have $\{<, >\}$.

Before giving the proof of the correctness of the algorithm CQAC1stable, we review the following theorem from [GSUW94]:

Theorem 5.6.1 *Let Q_1 and Q_2 be two CQAC's. Assume that no variable appears twice among their ordinary subgoals, and no constant appears in their ordinary subgoals. Let $O(Q_1)$ (resp. $O(Q_2)$) and $\mathcal{C}(Q_1)$ (resp. $\mathcal{C}(Q_2)$) be the ordinary subgoals and comparisons of*

[3] Formally, a total order of the variables in the query is an order with some equalities, i.e., all the variables are partitioned to sets $S_1, \ldots, S_k$, such that each S_i is a set of equal variables, and for any two variables $X_i \in S_i$ and $X_j \in S_j$, if $i < j$, then $X_i < X_j$.

Algorithm CQAC1stable: Test stability of CQAC's with comparisons $\{<,>\}$
Input: • Q: A CQAC with comparisons $\{<,>\}$.
• B: Binding restrictions of the relations used in Q.
Output: Decision about the stability of Q.
Method:
(1) Compute the answerable subquery Q_a of Q:
• Based on B, compute all answerable ordinary subgoals $\mathcal{A}$ using algorithm Inflationary.
• Derive all the inequalities $\mathcal{I}$ among the bound variables in $\mathcal{A}$ from $\mathcal{C}(Q)$.
• Let Q_a be the query with $\mathcal{A}$, $\mathcal{I}$, and the head of Q.
(2) Test whether $Q_a \sqsubseteq Q$ using the GZO algorithm.
(3) If $Q_a \sqsubseteq Q$, then Q is stable; otherwise, Q is not stable.

Figure 5.9: Algorithm: CQAC1stable.

query Q_1 (resp. Q_2), respectively. Then $Q_2 \sqsubseteq Q_1$ if and only if the following holds. Let H be the set of all containment mappings from $O(Q_1)$ to $O(Q_2)$. Then H is nonempty, and $\mathcal{C}(Q_2)$ logically implies $\vee_{h \text{ in } H} h(\mathcal{C}(Q_1))$. □

Theorem 5.6.2 *The algorithm CQAC1stable correctly decides the stability of a CQAC with comparisons $\{<,>\}$.* □

$$
\begin{array}{ll}
Q' : H \text{ :- } O_1, \ldots, O_k, \ldots, O_n, C_1, \ldots, C_m & \Leftarrow \text{instantiation } f \text{ (database } D_2) \\
 & \qquad\qquad \big\uparrow \text{extend} \\
Q'_a : H \text{ :- } O_1, \ldots, O_k, C'_1, \ldots, C'_m & \Leftarrow \text{instantiation } s \text{ (database } D_1)
\end{array}
$$

Figure 5.10: Proof of the correctness of the algorithm CQAC1stable.

Proof: If query Q_a is equivalent to query Q, clearly query Q is stable, since for any database D, we can compute $ANS(Q, D)$ by computing $ANS(Q_a, D)$, which is computable since all the ordinary subgoals of Q_a are answerable.

Now, we prove that if $Q_a \not\sqsubseteq Q$, query Q cannot be stable. We need to construct two databases D_1 and D_2, such that $ANS(Q, D_1) \neq ANS(Q, D_2)$, but these two databases have the same observable tuples. Figure 5.10 shows the main idea of the construction. We first rewrite the queries Q and Q_a to queries Q' and Q'_a that satisfy the assumptions in Theorem 5.6.1. That is, no variable in Q' (resp. Q'_a) appears twice among its ordinary subgoals, and no constant appears in its ordinary subgoals. The rewriting can be done as follows: (1) if a variable appears twice in the ordinary subgoals, we use distinct variables and equate them by arithmetic equality contraints; (2) we replace constants in the ordinary subgoals by new variables and equate those variables to the desired constants.

Assume that Q' have n ordinary subgoals, $O_1, \ldots, O_n$. Without loss of generality, let $O_1, \ldots, O_k$ be the subgoals corresponding to the answerable subgoals of Q. These k subgoals are also all the ordinary subgoals of Q'_a. Since $Q_a \not\sqsubseteq Q$, we have $Q'_a \not\sqsubseteq Q'$. By Theorem 5.6.1, $\mathcal{C}(Q'_a)$ does not imply $\vee_{h\ in\ H} h(\mathcal{C}(Q'))$, where H includes all the containment mappings from Q' to Q'_a. Then there must be an instantiation s for the variables in Q'_a, such that $s(\mathcal{C}(Q'_a))$ is true, while no h in H can make $s\Big(h(\mathcal{C}(Q'))\Big)$ true.

Let database D_1 include tuples $t_1, \ldots, t_k$ under the instantiation s. Notice that: (1) Q_a does not include comparisons $\{\leq, \geq, =, \neq\}$, and (2) the comparisons in $\mathcal{C}(Q'_a)$ are *all* the inequality constraints that the variables in Q_a should satisfy. Therefore, we can always extend the instantiation s to an instantiation f of the variables in Q', by assigning new distinct values to the unbound variables in Q. This instantiation f also turns the head of Q_a to a tuple t_h. Let database D_2 include the tuples of Q under the instantiation f.

Since instantiation f uses new distinct values for the unbound variables in Q, the tuples for the nonanswerable subgoals of Q_a cannot be retrieved under D_2 given the binding restrictions of the relations. Therefore, tuples $t_1, \ldots, t_k$ are all the observable tuples under both databases D_1 and D_2. We only need to prove that $t_h \notin ANS(Q', D_1)$. Otherwise we can construct a containment mapping μ from $O(Q')$ to $O(Q'_a)$, such that $s\Big(\mu(\mathcal{C}(Q'))\Big)$ is true, contradicting the fact that no h in H can make $s\Big(h(\mathcal{C}(Q'))\Big)$ true. ∎

The algorithm CQAC1stable also shows how to compute the complete answer to a stable CQAC Q without comparisons $\{\leq, \geq, =, \neq\}$. For any database D, we compute $ANS(Q, D)$ by computing $ANS(Q_a, D)$. To compute $ANS(Q_a, D)$, we first use a linear plan following a feasible order of the subgoals $O(Q_a)$ to solve these subgoals. Then we filter out the tuples in the supplementary relation that do not satisfy the comparisons $\mathcal{C}(Q_a)$.

EXAMPLE 5.6.3 Consider query

$$P : ans(B) \text{ :- } p(B), r(A, B), r(A, C), A < C, C < B$$

where relation p has a binding pattern f, and relation r has a binding pattern fb. Since P does not have comparisons $\{\leq, \geq, =, \neq\}$, we can use the algorithm CQAC1stable to test its stability. Clearly subgoals $p(B)$ and $r(A, B)$ are answerable and the bound variables are A and B. We derive all the inequalities of A and B from $A < C$ and $C < B$. The only derived inequality of the two variables is $A < B$. Thus the following is the answerable subquery of Q:

$$P_a : ans(B) \text{ :- } p(B), r(A,B), A < B$$

We then test whether $P_a \sqsubseteq P$. We rewrite the queries P and P_a to the following queries P' and P'_a that satisfy the assumptions in Theorem 5.6.1.

$$\begin{array}{lll} P': & ans(B) & \text{:- } p(B), r(A,X), r(Y,C), X = B, Y = A, A < C, C < B \\ P'_a: & ans(B) & \text{:- } p(B), r(A,X), X = B, A < B \end{array}$$

The comparisons in query P' (i.e., $\mathcal{C}(P')$) are: $X = B$ & $Y = A$ & $A < C$ & $C < B$; the comparisons in query P'_a (i.e., $\mathcal{C}(P'_a)$) are: $X = B$ & $A < B$. There is only one containment mapping μ from P' to P'_a:

$$\mu(B) = B; \mu(A) = A; \mu(X) = X; \mu(Y) = A; \mu(C) = X.$$

Thus we need to verify whether $\mathcal{C}(P'_a)$ logically implies $\mu(\mathcal{C}(P'))$:

$$X = B \wedge A < B \Rightarrow X = B \wedge A = A \wedge A < X \wedge X < B$$

That is:

$$A < B \Rightarrow A < B \wedge B < B$$

which is false, and we have $P_a \not\sqsubseteq P$. Therefore, query P is not stable. The nonstability of query Q can also be proved by the following two databases: $D_1 = \{p(3), r(1,3), r(1,2)\}$, $D_2 = \{p(3), r(1,3)\}$. Clearly $ANS(P, D_1) = \{3\}$, and $ANS(P, D_2) = \phi$, but these two databases have the same observable tuples $\{p(3), r(1,3)\}$. Note that tuple $r(1,2)$ cannot be retrieved because "2" represents any constant that is between 1 and 3, but not equivalent to 1 and 3. □

5.6.3 Algorithm: CQACstable

Now we show how to test stability of CQAC's by giving the following theorem.

Theorem 5.6.3 *Let Q be a CQAC, and $\Omega(Q)$ be the set of all the total orders of the variables in Q that satisfy the comparisons of Q. For each total order $\lambda \in \Omega(Q)$, let Q^λ be the corresponding query that includes the ordinary subgoals of Q and all the inequalities and equalities of this order λ. Then query Q is stable if and only if for all $\lambda \in \Omega(Q)$, query $Q^\lambda_a \sqsubseteq Q$, where Q^λ_a is the answerable subquery of Q^λ.* □

$$Q \equiv \bigcup \left\{ \begin{array}{lllll} Q^{\lambda_1} & \sqsubseteq & Q_a^{\lambda_1} & \sqsubseteq & Q \\ Q^{\lambda_2} & \sqsubseteq & Q_a^{\lambda_2} & \sqsubseteq & Q \\ \vdots & & & & \\ Q^{\lambda_m} & \sqsubseteq & Q_a^{\lambda_m} & \sqsubseteq & Q \end{array} \right.$$

Figure 5.11: Proof of Theorem 5.6.3, "*If*" part.

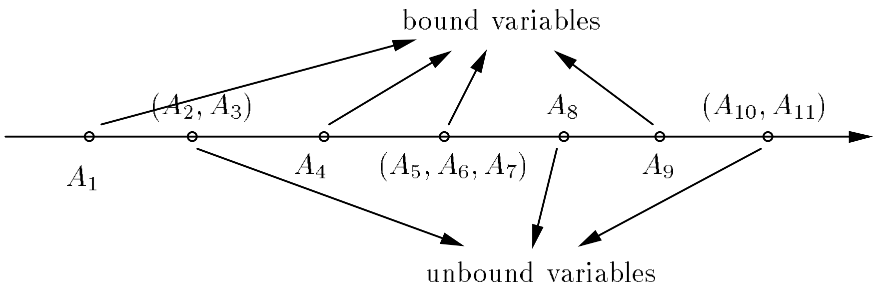


Figure 5.12: Proof of Theorem 5.6.3, "*Only If*" part.

Proof: *If*: Assume that for each $\lambda_i \in \Omega(Q)$, $Q_a^{\lambda_i} \sqsubseteq Q$. As shown in Figure 5.11, $Q \equiv \bigcup_{\lambda \in \Omega(Q)} Q^{\lambda}$. In addition, for each $\lambda_i \in \Omega(Q)$, $Q^{\lambda_i} \sqsubseteq Q_a^{\lambda_i}$. Then we have $Q \equiv \bigcup_{\lambda \in \Omega(Q)} Q^{\lambda} \sqsubseteq \bigcup_{\lambda \in \Omega(Q)} Q_a^{\lambda} \sqsubseteq Q$. Thus $Q \equiv \bigcup_{\lambda \in \Omega(Q)} Q_a^{\lambda}$. For any database D, we can compute $ANS(Q, D)$ by computing $ANS(Q_a^{\lambda}, D)$ for each total order $\lambda \in \Omega(Q)$. This answer is computable since all its subgoals are answerable. Then we take the union of these answers as $ANS(Q, D)$. Therefore, query Q is stable.

Only If: Assume there is a total order in $\Omega(Q)$, say λ_1, such that $Q_a^{\lambda_1} \not\sqsubseteq Q$. Figure 5.12 shows the main idea of a total order. The variables on the left side must be smaller than the variables on the fight side. Some variables must be equal to each other. For instance, the variables in the figure must satisfy:

$$A_1 < A_2 = A_3 < A_4 < A_5 = A_6 = A_7 < A_8 < A_9 < A_{10} = A_{11}$$

Some variables (i.e., variables $A_1, A_4, A_5, A_6, A_7, A_9$ in the figure) are bound by the answerable subgoals. Notice that we need to consider the equalities to compute all the answerable subgoals given the binding restrictions of the relations. That is because these equalities can help bind more variables.

Now we prove query Q cannot be stable. We need to construct two databases D_1 and D_2, such that $ANS(Q, D_1) \neq ANS(Q, D_2)$, but D_1 and D_2 have the same observable tuples. The construction is essential the same as the construction in the proof of Theorem 5.6.2. That is, let s be the instantiation of the variables in $Q_a^{\lambda_1}$ that makes $\mathcal{C}(P_a^{\lambda_1}) \Rightarrow \mathcal{C}(P)$ false, where $P_a^{\lambda_1}$ and P are the rewritten queries of $Q_a^{\lambda_1}$ and Q that satisfy the assumptions in Theorem 5.6.1. These variables correspond to the bound variables in the total order λ_1. Let D_1 include the tuples under the instantiation s.

Since $Q_a^{\lambda_1} \not\sqsubseteq Q$, $Q_a^{\lambda_1}$ must have fewer subgoals than Q, and some subgoals in Q are not answerable. In addition, some variables are not bound. For those unbound variables, we can choose new distinct values for them. That is, the unbound variables and the bound variables have different values. Therefore, we can always extend instantiation s to a new instantiation f for the variables in Q, such that f uses new distinct values for the unbound variables. Let D_2 include the tuples corresponding to the instantiation f. By the construction of D_1 and D_2, they have the same observable tuples (i.e., the tuples in D_1), since we chose new distinct values for the unbound variables. Following the same idea in the proof of Theorem 5.6.2, we can prove that $ANS(Q, D_1)$ does not include the tuple $t_h = f(G)$, where G is the head of Q. Therefore, query Q is not stable. ∎

Theorem 5.6.3 gives an algorithm *CQACstable* (shown in Figure 5.13) that tests the stability of any CQAC, even if the query has comparisons $\{\leq, \geq, =, \neq\}$. The algorithm considers all the total orders of the variables, including those with equalities.

Algorithm CQACstable: Test stability of CQAC's
Input: • Q: A conjunctive query with arithmetic comparisons.
• B: Binding restrictions of the relations used in Q.
Output: Decision about the stability of Q.
Method:
(1) Compute all the total orders $\Omega(Q)$ of the variables in Q that satisfy the comparisons in Q.
(2) For each $\lambda \in \Omega(Q)$:
- Compute the answerable subquery Q_a^{λ} of query Q^{λ};
- Test $Q_a^{\lambda} \sqsubseteq Q$ by calling the GZO algorithm;
- If $Q_a^{\lambda} \not\sqsubseteq Q$, then query Q is not stable.

(3) Query Q is stable.

Figure 5.13: Algorithm: CQACstable.

The algorithm CQACstable also shows how to compute the complete answer to a stable CQAC Q for a database D. That is, let $Q \equiv \bigcup_{\lambda \in \Omega(Q)} Q^{\lambda}$. For each query Q^{λ}, we compute

$ANS(Q_a^\lambda, D)$, where Q_a^λ is the answerable subquery of Q^λ. Since Q_a^λ has a feasible order of all its ordinary subgoals, we compute $ANS(Q_a^\lambda, D)$ by using a linear plan following this order, and filtering out the results using the comparisons in Q_a^λ. We take the union of the answers for all the total orders in $\Omega(Q)$ as $ANS(Q, D)$.

EXAMPLE 5.6.4 Consider the query P in Example 5.6.2. It has the following 8 total orders:

λ_1: $X < A = Y < B$	λ_5: $X = A = Y < B$
λ_2: $X < A < Y < B$	λ_6: $X = A < Y < B$
λ_3: $X < A < Y = B$	λ_7: $X = A < Y = B$
λ_4: $X < A < B < Y$	λ_8: $X = A < B < Y$

For each total order λ_i, we write its corresponding query P^{λ_i}. Here are all the 8 queries:

$$
\begin{array}{lll}
P^{\lambda_1}: & ans(Y) & :\!-\ p(X), r(X,Y), r(Y,B), X < Y, Y < B \\
P^{\lambda_2}: & ans(Y) & :\!-\ p(X), r(X,Y), r(A,B), X < A, A < Y, Y < B \\
P^{\lambda_3}: & ans(Y) & :\!-\ p(X), r(X,Y), r(A,Y), X < A, A < Y \\
P^{\lambda_4}: & ans(Y) & :\!-\ p(X), r(X,Y), r(A,B), X < A, A < B, B < Y \\
P^{\lambda_5}: & ans(Y) & :\!-\ p(Y), r(Y,Y), r(Y,B), Y < B \\
P^{\lambda_6}: & ans(Y) & :\!-\ p(X), r(X,Y), r(X,B), X < Y, Y < B \\
P^{\lambda_7}: & ans(Y) & :\!-\ p(X), r(X,Y), r(X,Y), X < Y \\
P^{\lambda_8}: & ans(Y) & :\!-\ p(X), r(X,Y), r(X,B), X < B, B < Y
\end{array}
$$

For each total order λ_i, we construct its corresponding answerable subquery $P_a^{\lambda_5}$. The following are the 8 answerable subqueries:

$$
\begin{array}{lll}
P^{\lambda_1}: & ans(Y) & :\!-\ p(X), r(X,Y), r(Y,B), X < Y, Y < B \\
P^{\lambda_2}: & ans(Y) & :\!-\ p(X), r(X,Y), X < Y \\
P^{\lambda_3}: & ans(Y) & :\!-\ p(X), r(X,Y), X < Y \\
P^{\lambda_4}: & ans(Y) & :\!-\ p(X), r(X,Y), X < Y \\
P^{\lambda_5}: & ans(Y) & :\!-\ p(Y), r(Y,Y), r(Y,B), Y < B \\
P^{\lambda_6}: & ans(Y) & :\!-\ p(X), r(X,Y), r(X,B), X < Y, Y < B \\
P^{\lambda_7}: & ans(Y) & :\!-\ p(X), r(X,Y), X < Y \\
P^{\lambda_8}: & ans(Y) & :\!-\ p(X), r(X,Y), r(X,B), X < B, B < Y
\end{array}
$$

Each of the 8 answerable subquery can be proved to be contained in P. Therefore, query P is stable. Actually, if we combine queries P^{λ_1}, $P_a^{\lambda_2}$, P^{λ_3}, P^{λ_4}, P^{λ_6}, $P_a^{\lambda_7}$, $P_a^{\lambda_8}$, and we have query

$$P^{\lambda_{1,2,3,4,6,7,8}} : ans(Y) \text{ :- } p(X), r(X,Y), r(A,B), X < Y, A < B, X \leq A, A \leq Y$$

and its answerable subquery is:

$$P_a^{\lambda_{1,2,3,4,6,7,8}} : ans(Y) \text{ :- } p(X), r(X,Y), X < Y$$

which is the query T_1 in Example 5.6.2. In addition, $P_a^{\lambda_5}$ is equivalent to the query T_2 in the example. We can prove that $P_a^{\lambda_{1,2,3,4,6,7,8}} \sqsubseteq P^{\lambda_{1,2,3,4,6,7,8}}$. Since query $P \equiv P^{\lambda_{1,2,3,4,6,7,8}} \cup P^{\lambda_5}$, we have $P \equiv T_1 \cup T_2$. □

5.7 Stability of Datalog Queries

In this section we study stability of datalog queries, i.e., Horn-clause programs without function symbols. We show that if a datalog query has a feasible rule/goal graph, then the query is stable. We also show that stability of datalog queries is not decidable.

EXAMPLE 5.7.1 Let us repeat the example in Section 2.1.2. Suppose $flight$ is a finite relation with a binding adornment bf, and $flight(F,T)$ means that there is a nonstop flight from airport F to airport T. An IDB relation *reachable* is defined by the following two rules:

$$\begin{array}{lll} r_1: & reachable(X,Y) & \text{:- } flight(X,Y) \\ r_2: & reachable(X,Y) & \text{:- } flight(X,Z), reachable(Z,Y) \end{array}$$

Let queries P_1 and P_2 be $reachable(sfo, X)$ and $reachable(X, sfo)$, respectively. That is, query P_1 asks for all the airports that are reachable from the airport sfo, while query P_2 asks for all the airports from which the airport sfo is reachable. Although we cannot retrieve all the flight facts, the answer to query P_1 can still be computed as follows: we query the relation to retrieve all the airports that are reachable from sfo via one nonstop flight. For each of these airports, we query the relation to retrieve its one-nonstop reachable airports. We repeat the process until no new airports are found. This process will terminate, since the $flight$ relation is finite. The set of the retrieved airports is the complete answer to query P_1. That is because for any airport a in the answer, there exists a chain of distinct airports $a_1 = sfo, a_2, \ldots, a_n = a$, such that for $i = 1, \ldots, n-1$, tuple $\langle a_i, a_{i+1} \rangle$ is in the $flight$

relation. Therefore, airport a_i can be retrieved in the $(i-1)$st step during the computation above.

For query P_2, we cannot compute its answer in the same way as P_2, because the $flight$ relation does not allow us to retrieve its facts in a "forward" way. In fact, we cannot know the complete answer to query P_2 at all, since there can always be an airport from which sfo is reachable, but this airport cannot be retrieved from the relation. □

5.7.1 Rule/goal Graphs

Given a set of rules and a query goal p^α with an adornment α (a string of b's and f's), a rule/goal graph (RGG for short) indicates the order in which subgoals are to be evaluated in these rules, and indicates the way in which variable bindings pass from one subgoal to another within a rule. (Details about rule/goal graphs are found in Chapter 12 in [Ull89].)

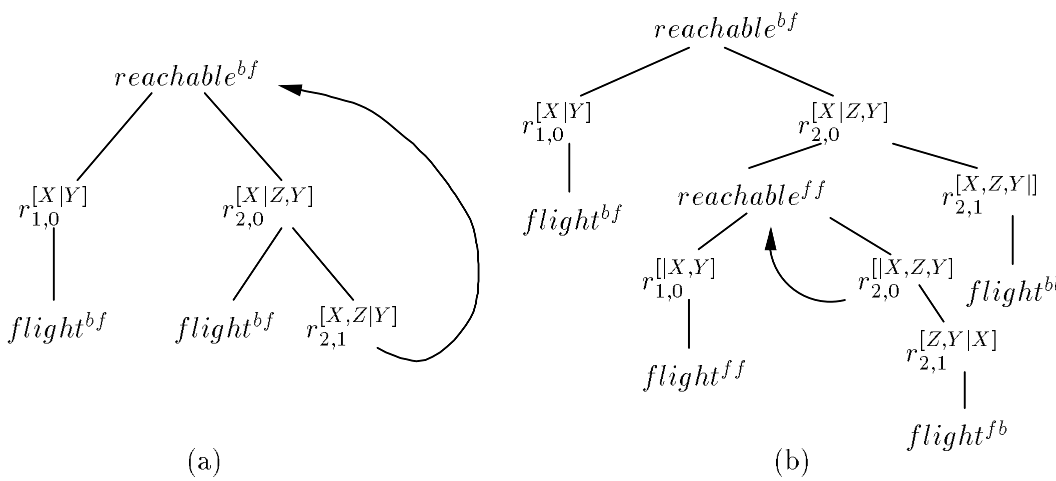


Figure 5.14: Two RGG's for the query goal $reachable^{bf}$.

EXAMPLE 5.7.2 Consider the query P_1 in Example 5.7.1. The query can be represented as a goal $reachable^{bf}$, i.e., the problem of determining, given a fixed value (i.e., sfo) for the first argument, the set of Y such that $reachable(sfo, Y)$ is true. Figure 5.14(a) shows an RGG of the goal. In the graph, there are two different kinds of nodes: goal nodes and rule nodes. A goal node is a predicate with a binding adornment that specifies which arguments are bound and which are not. For instance, the root node $reachable^{bf}$ is a goal node indicating that the first argument of predicate *reachable* has been bound, and the second argument is not.

A rule node indicates the binding status of the variables in a rule. Its subscript indicates the stage of processing the subgoals in the rule from left to right. Its superscript specifies what variables have been bound so far, either by a previous subgoal, or by the head of this rule. The superscript also specifies what variables have not been bound. For instance, the node $r_{1,0}^{[X|Y]}$ is a rule node. Its subscript "1, 0" means that it corresponds to the stage when no subgoal of rule r_1 has been processed. Its superscript "$[X|Y]$" means that at this stage, variable X is bound (by the head of the rule) and variable Y is not. Similarly, the rule node $r_{2,0}^{[X|Z,Y]}$ specifies that when no subgoal of rule r_2 has been processed, variable X is bound, and variables Y and Z are not. The rule node $r_{2,1}^{[X,Z|Y]}$ means that after the first subgoal of rule r_2 is processed, variables X and Z are bound, and Y is not. The RGG in Figure 5.14(a) is constructed respecting the order in which the subgoals are written. For a different order we may have a different RGG. For instance, if we switch the two subgoals in rule r_2, we will have a new RGG, as shown in Figure 5.14(b). □

We assume that a set of rules has the *unique binding pattern* property with respect to a given adorned goal. That is, when we construct the RGG starting with the adorned goal and following the order of the subgoals of the rules as written, no IDB predicate appears with two different adornments. If a set of rules does not have this property with respect to a query goal, we can call the Algorithm 12.7 in [Ull89] to rewrite these rules and the goal, and generate a revised set of rules that has the unique binding pattern property with respect to the query goal.

5.7.2 Feasible Rule/goal Graphs

Given a set of rules on EDB relations with binding restrictions, an RGG of a goal node p^{α} is *feasible* if all its EDB goals in the RGG use only the adornments that are permitted by the EDB relations. For instance, in Example 5.7.1, the RGG in Figure 5.14(a) is a feasible RGG, since all the EDB goals in the graph (the two nodes of $flight^{bf}$) are permitted by the $flight$ relation. However, the RGG in Figure 5.14(b) is not feasible, since it has two EDB goals ($flight^{ff}$ and $flight^{fb}$) that are not permitted by the $flight$ relation.

Theorem 5.7.1 *If a set of rules on EDB relations with binding restrictions has a feasible RGG with respect to a query goal, then the query is stable.* □

Proof: Assume that the set of rules and the query goal p^α have a feasible RGG. We apply the magic-sets transformation (as described in [Ull89, Chapter 13]) on the rules and the goal, and get a new set of rules $\mathcal{R}$ such that the relation for p is the answer to the query. For any instance of the EDB relations, consider the case where the relations did not have restrictions. The complete answer to the query p can be computed using a bottom-up evaluation of the rules $\mathcal{R}$.

Since the EDB relations do have binding restrictions, we should consider whether the bottom-up evaluation of $\mathcal{R}$ can be executed. Because G is a feasible RGG, during the bottom-up evaluation of $\mathcal{R}$, each time an EDB subgoal is evaluated, we have the necessary bindings to query the relation. Therefore, we can still execute the bottom-up evaluation. In addition, in each step of the evaluation, the facts we need to compute are the same as the facts we computed in the bottom-up evaluation if the EDB relations did not have any restrictions. Therefore, we can compute the complete answer to the query using a bottom-up evaluation of $\mathcal{R}$. ∎

The proof of Theorem 5.7.1 also gives an algorithm for computing the complete answer to a query goal if it has a feasible RGG. That is, we apply the magic-sets transformation to the rules and the goal to get a set of rules $\mathcal{R}$. We evaluate these rules using a bottom-up evaluation. In each step, we evaluate a rule following the order in which the RGG is constructed. By the construction of the rules $\mathcal{R}$, each time we solve an EDB subgoal, we have enough bindings to evaluate this subgoal.

[Mor88] gave an algorithm for testing the existence of a feasible RGG given a set of rules and a query goal. The algorithm is inherently exponential in time. However, if there is a bound on the arity of predicates, then the algorithm with this heuristic takes polynomial time [UV88].

5.7.3 What If a Feasible RGG Does not Exist

In some cases, even though a set of rules do not has a feasible RGG with respect to a query goal, the query may still be stable, since we may rewrite the rules to obtain a new set of rules that has a feasible RGG with respect to the query goal. The following example is a case in point.

EXAMPLE 5.7.3 If we add another subgoal $flight(W, Z)$ to rule r_2. Then the new set of rules does not have a feasible RGG with respect to the query goal of P_1 , since the variable W in the third subgoal $flight(W, Z)$. However, the new rules are equivalent to the old ones, and query P_1 on the new rules is still stable. □

Example 5.7.3 shows a similar phenomenon as CQ's; that is, we need to "minimize" datalog rules before checking the existence of a feasible RGG. However, minimizing datalog rules is much harder than minimizing CQ's and UCQ's. Shmueli [Shm93] showed that for a datalog program P, whether P is equivalent to datalog program P', where P' is produced by removing a subgoal from a rule of P, is undecidable. Not surprisingly, we have the following theorem:

Theorem 5.7.2 *Stability of datalog programs is undecidable.* □

Proof: Let P_1 and P_2 be two arbitrary datalog queries. We show that a decision procedure for the stability of datalog programs would allow us to decide whether $P_1 \sqsubseteq P_2$. Since containment of datalog programs is undecidable, we prove the claim.[4] Let all the EDB relations in the two queries have an all-free binding pattern; i.e., there is no restriction of retrieving tuples from these relations. Without loss of generality, we can assume that the goal predicates in P_1 and P_2, named p_1 and p_2 respectively, have arity m. Let $\mathcal{Q}$ be the datalog query consisting of all the rules in P_1 and P_2, and of the rules:

$$\begin{array}{lll} r_1: & ans(X_1, \ldots, X_m) & :\!\!-\ p_1(X_1, \ldots, X_m), e(Z) \\ r_2: & ans(X_1, \ldots, X_m) & :\!\!-\ p_2(X_1, \ldots, X_m) \end{array}$$

where e is a new 1-ary relation with the binding pattern b, and Z is a new argument that does not appear in $X_1, \ldots, X_m$. We show that $P_1 \sqsubseteq P_2$ if and only if query $\mathcal{Q}$ is stable.

"Only If": Assume $P_1 \sqsubseteq P_2$. Hence $\mathcal{Q} = P_2$. Since the EDB relations in P_2 can return all their tuples for free, P_2 (thus $\mathcal{Q}$) is stable.

"If": Assume $P_1 \not\sqsubseteq P_2$, we prove that query $\mathcal{Q}$ cannot be stable. Since P_1 is not contained in P_2, there exists a database D of the EDB relations in P_1 and P_2, such that $ANS(P_1, D) \not\subseteq ANS(P_2, D)$. That is, there is a tuple $t \in ANS(P_1, D)$, while $t \notin ANS(P_2, D)$. Now we construct two databases D_1 and D_2 of the EDB relations and the relation e, such that query $\mathcal{Q}$ has the same observable tuples under D_1 and D_2, but $ANS(\mathcal{Q}, D_1) \neq ANS(\mathcal{Q}, D_2)$.

[4] The idea of the proof is borrowed from [Dus97], Chapter 2.3.

Both D_1 and D_2 include D for the EDB relations in P_1 and P_2. However, in D_1, relation e is empty; in D_2, relation e has one tuple $\langle z \rangle$, while z is a new value that does not appear in any tuple in D. For both D_1 and D_2, the observable tuples are those in D, while we cannot get any tuple from relation e. Hence, rule r_1 cannot yield any answer to $\mathcal{Q}$, and the retrievable answer to $\mathcal{Q}$ is $ANS(P_2, D)$ for both D_1 and D_2. For D_1, since relation e is empty, $ANS(\mathcal{Q}, D_1) = ANS(P_2, D)$, which does not include tuple t. However, for D_2, relation e has a tuple $\langle z \rangle$, and $ANS(\mathcal{Q}, D_1) = ANS(P_1, D) \cup ANS(P_2, D)$, which includes tuple t. Therefore, $ANS(\mathcal{Q}, D_1) \neq ANS(\mathcal{Q}, D_2)$, and query $\mathcal{Q}$ is not stable. ∎

5.8 Conclusions and Related Work

In this chapter we studied the following problem of answering queries in the presence of binding restrictions: can the complete answer to a query be computed given the restrictions? If so, how to compute it? We first studied conjunctive queries, and showed that a conjunctive query is stable if and only if its minimal equivalent query Q_m has a feasible sequence of all its subgoals. We proposed two algorithms for testing stability of conjunctive queries, and proved this problem is $\mathcal{NP}$-complete.

For a nonstable conjunctive query, whether its complete answer can be computed is data dependent. We proposed a decision tree that guides the query planning process to compute the complete answer to a conjunctive query, if it can be computed at all. Two planning strategies — a pessimistic strategy and an optimistic strategy — can be taken while traversing the decision tree. We also studied stability of unions of conjunctive queries, and conjunctive queries with arithmetic comparisons. In both cases we proposed algorithms for testing stability of queries. Finally, we studied datalog queries, and proved that if a set of rules and a query goal have a feasible rule/goal graph, then the query is stable. We proved that stability of datalog queries is undecidable.

Related Work

Several works consider binding restrictions in the context of answering queries using views [DL97, LMSS95, Qia96]. Rajaraman, Sagiv, and Ullman [RSU95] proposed algorithms for answering queries using views with binding patterns. In that paper all solutions to a query compute the complete answer to the query; thus only stable queries are handled. Duschka and Levy [DL97] solved the same problem by translating source restrictions into recursive rules in a datalog program to obtain the maximally-contained rewriting of a query, but the rewriting does not necessarily compute the query's complete answer.

A query on relations with binding restrictions can be generated by a view-expansion process at mediators as in TSIMMIS. [LYV+98] studied the problem of generating an executable plan based on source restrictions. [FLMS99, YLUGM99] studied query optimization in the presence of binding restrictions. [YLGMU99] considered how to compute mediator restrictions given source restrictions. These four studies did not consider the possibility that removing subgoals may make an infeasible query feasible. Thus, they regard the query Q_2 in Example 5.1.2 as an unsolvable query, thus miss the chances of computing its complete answer.

In Chapter 4, we studied how to compute the maximal answer to a conjunctive query with binding restrictions by borrowing bindings from relations not in the query. We focused on how to trim irrelevant relations that do not help in obtaining bindings. However, the computed answer may not be the complete answer. As shown in Section 5.4, we can sometimes use this exhaustive approach in that paper to compute the complete answer to a nonstable conjunctive query.

The dynamic case of computing a complete answer to a nonstable query is different from the case of dynamic mediators discussed in [YLGMU99]. In [YLGMU99], source descriptions can specify a set of values that can be bound to an attribute at a source. The uncertainty of whether the mediator can answer a query comes from the fact that, before the query is executed, it is unknown whether the intermediate bindings are allowed by a source. In our case, as we saw in Section 5.4, the uncertainty comes from the fact that, before executing a plan, we do not know whether the tuples for the nonanswerable subgoals can join with all the tuples in the supplementary relations [BR87] of the answerable subgoals. That is why the computability of the complete answer is data dependent.

Chapter 6

Using Mediator Caching to Improve Performance

In information-integration systems, each access to a source could be expensive in terms of network traffic and delay, dynamic source availability, and possible source charges. Mediator caching can be used to reduce the number of source accesses. That is, we store the results of previous queries at the mediator, and then use the cached results to answer future queries. Since the mediator has limited resources (e.g., storage space), we may not be able to store all query results. Periodically we need replace some query results in the cache with new query results.

In this chapter we study the problem of deciding what query results (views) should be kept in the mediator cache, so that we can use the results to answer as many future queries as possible. We show that the traditional query-containment concept in [CM77] is *not* a good basis for deciding whether a view should be selected. Instead, we should minimize a view set while losing as little query-answering power as possible. To formalize this notion, we first introduce the concept of "p-containment." That is, a view set $\mathcal{V}$ is *p-contained* in another view set $\mathcal{W}$, if $\mathcal{W}$ can answer all the queries that can be answered by $\mathcal{V}$. Here we take the closed-world assumption (CWA) about views. We show that p-containment and the traditional query containment are *not* related. We then discuss how to minimize a view set while retaining its query-answering power. We develop the idea further by considering p-containment of two view sets with respect to a given set of queries, and consider their relationship in terms of maximally-contained rewritings of queries using the views.

Chapter Organization

Section 6.1 gives a motivating example to show what query results should be kept in a mediator cache. Section 6.2 introduces the concept of *p-containment*, which captures the fact that one view set is more powerful than another view set in terms of the possible queries they can answer. We compare p-containment with traditional query containment. We also discuss how to minimize a view set without losing its power to answer queries. Section 6.3 considers the relationship between two view sets with respect to a given set of queries. In particular, we consider infinite query sets defined by finite, parameterized queries.

Section 6.4 introduces the concept of *MCR-containment*, which describes the relative power of two view sets in terms of their maximally-contained rewritings of queries. Surprisingly, MCR-containment is essentially the same as p-containment. Section 6.5 extends the results to general queries, and proposes a framework for minimizing view sets without losing their query-answering power. Section 6.6 concludes the chapter and discusses related work.

6.1 Introduction

The following example shows that query results in a mediator cache can have redundancy. When we replace these query results, traditional query containment is *not* a good basis for deciding whether a view should be selected. Instead, we should consider the *query-answering* power of the views.

EXAMPLE 6.1.1 Suppose a source has the following base relation about books:

$$book(Title, Author, Pub, Price)$$

For example, the tuple $\langle$`databases`, `smith`, `prenhall`, \$60$\rangle$ at the source means that a book titled `databases` has an author `smith`, is published by Prentice Hall (`prenhall`), and has a current price of \$60. Assume that the mediator has seen the following three queries on the source, the answers of which have been cached locally. The cached data (or views), denoted by the view set $\mathcal{V} = \{V_1, V_2, V_3\}$, are:

$$\begin{array}{lll} V_1: & v_1(T, A, P) & \text{:- } book(T, A, B, P) \\ V_2: & v_2(T, A, P) & \text{:- } book(T, A, prenhall, P) \\ V_3: & v_3(A_1, A_2) & \text{:- } book(T, A_1, prenhall, P_1), book(T, A_2, prenhall, P_2) \end{array}$$

The view V_1 has author-title-price information about all books in the relation, while the view V_2 includes this information only about books published by Prentice Hall. The view V_3 has coauthor pairs for books published by Prentice Hall. Since the views have redundancy, we might want to eliminate a view from the mediator to save the cost of its maintenance and storage. At the same time, we want to be assured that such an elimination does not cause increased server accesses in response to future queries.

Clearly, $V_2 \sqsubseteq V_1$, that is, V_1 includes all the tuples in V_2, so we might be tempted to select $\{V_1, V_3\}$, and eliminate V_2 as a redundant view from the cache. However, with this selection, we cannot answer the query:

$$Q_1 : q_1(T, P) \text{ :- } book(T, smith, prenhall, P)$$

which asks for titles and prices of books written by `smith` and published by `prenhall`. The reason is that even though V_1 includes author-title-price information about all books in the base relation, the publisher attribute is projected out in the head of the view V_1. Thus, using V_1 only, we cannot tell which books are published by `prenhall`. On the other hand, the query Q_1 can be answered trivially using V_2:

$$q_1(T, P) \text{ :- } v_2(T, smith, P)$$

In other words, by dropping V_2 we have lost some *power* to answer queries. In addition, note that even though view V_3 is not contained in $V_1 \cup V_2$, it can be eliminated from $\mathcal{V}$ without changing the query-answering power of $\mathcal{V}$. The reason is that V_3 can be computed from V_2 as follows:

$$P_1 : v_3(A_1, A_2) \text{ :- } v_2(T, A_1, P_1), v_2(T, A_2, P_2)$$

To summarize, we should not select $\{V_1, V_3\}$ but $\{V_1, V_2\}$, even though the former includes all the tuples in $\mathcal{V}$, while the latter does not. The rationale is that the latter is as "powerful" as $\mathcal{V}$ while the former is not. One might hypothesize from this example that only projections in view definitions cause such a mismatch. We show in Section 6.2 that this hypothesis is wrong. □

In this chapter we discuss how to select query results in a mediator cache without losing their query-answering power. We first introduce the concept of *p-containment* between two sets of views, where "p" stands for query-answering *power* (Section 6.2). A view set $\mathcal{V}$ is *p-contained* in another view set $\mathcal{W}$, or $\mathcal{W}$ is *at least as powerful* as $\mathcal{V}$, if $\mathcal{W}$ can answer all the queries that can be answered using $\mathcal{V}$. Two view sets are called *equipotent* if they have the same power to answer queries. As shown in Example 6.1.1, two view sets may have the same tuples, yet have different query-answering power. That is, traditional view containment [CM77, SY80] does not imply p-containment. The example further shows that the reverse direction is not implied either.

Given a view set $\mathcal{V}$ on base relations, we show how to find a minimal subset of $\mathcal{V}$ that is equipotent to $\mathcal{V}$ (Section 6.2.4). As one might suspect, a view set can have several equipotent minimal subsets. In some scenarios, users are restricted in the queries they can ask. In such cases, equipotence may be determined *relative* to the expected (possibly infinite) set of queries. In Section 6.3, we investigate the above questions of equipotence testing given this extra constraint. In particular, we consider infinite query sets defined by parameterized queries, and develop algorithms for testing this relative p-containment.

In information integration, we often need to consider not only equivalent rewritings of a query using views, but also maximally-contained rewritings (MCR's). Analogous to p-containment, which requires equivalent rewritings, we introduce the concept of *MCR-containment* that is defined using maximally-contained rewritings (Section 6.4). Surprisingly, we show that p-containment implies MCR-containment, and vice-versa.

The different containments between two view sets discussed in this chapter are summarized in Table 6.1. We start our discussions with conjunctive queries, and generalize the results to more general queries in Section 6.5.

Containment	**Definition**	**How to test**
v-containment $\mathcal{V} \sqsubseteq_v \mathcal{W}$	For any database, a tuple in a view in $\mathcal{V}$ is in a view in $\mathcal{W}$.	Check if each view in $\mathcal{V}$ is contained in some view of $\mathcal{W}$.
p-containment $\mathcal{V} \preceq_p \mathcal{W}$	If a query is answerable by $\mathcal{V}$, then it is answerable by $\mathcal{W}$.	Check if each view in $\mathcal{V}$ is answerable by $\mathcal{W}$.
relative p-containment $\mathcal{V} \preceq_{\mathcal{Q}} \mathcal{W}$	For each query Q in a given set of queries $\mathcal{Q}$, if Q is answerable by $\mathcal{V}$, then Q is answerable by $\mathcal{W}$.	Test by the definition if $\mathcal{Q}$ is finite. See Section 6.3.3 for infinite set of queries.
MCR-containment $\mathcal{V} \preceq_{MCR} \mathcal{W}$	For each query Q, for any maximally -contained rewriting $MCR(Q,\mathcal{V})$ (resp. $MCR(Q,\mathcal{W})$) of Q using $\mathcal{V}$ (resp. $\mathcal{W}$), $MCR(Q,\mathcal{V}) \sqsubseteq MCR(Q,\mathcal{W})$.	Same as testing if $\mathcal{V} \preceq_p \mathcal{W}$, since $\mathcal{V} \preceq_p \mathcal{W} \Leftrightarrow \mathcal{V} \preceq_{MCR} \mathcal{W}$.

Table 6.1: Different containments between two sets of conjunctive views: $\mathcal{V}$ and $\mathcal{W}$.

Note that even though the results in this chapter are developed in the setting of mediator caching, they are applicable in many applications where views are used. In general, views are expensive to maintain: materialized views often compete for limited resources; views need to be kept up to date since base relations can change from time to time. By selecting an equipotent subset of views, we can reduce the maintanence cost of these views while retaining the same query-answering power.

6.2 Comparing Query-Answering Power of View Sets

Before discussing how to minimize a view set without losing its query-answering power, we first introduce the concept of *p-containment*. It captures the fact that one view set may be more powerful than another view set, in terms of the possible queries they can answer. We compare p-containment with traditional query containment. We also discuss how to minimize a view set without losing its power to answer queries.

6.2.1 p-containment

Definition 6.2.1 (p-containment and equipotence) A view set $\mathcal{V}$ is *p-contained* in another view set $\mathcal{W}$, or "$\mathcal{W}$ is at least as powerful as $\mathcal{V}$," denoted by $\mathcal{V} \preceq_p \mathcal{W}$, if any query answerable by $\mathcal{V}$ is also answerable by $\mathcal{W}$. Two view sets are *equipotent*, denoted by $\mathcal{V} \asymp_p \mathcal{W}$, if $\mathcal{V} \preceq_p \mathcal{W}$, and $\mathcal{W} \preceq_p \mathcal{V}$. □

In Example 6.1.1, the two view sets $\{V_1, V_2\}$ and $\{V_1, V_2, V_3\}$ are equipotent, since the latter can answer all the queries that can be answered by the former, and vice-versa. (We will give a formal proof shortly.) However, the two view sets, $\{V_1, V_3\}$ and $\{V_1, V_2, V_3\}$, are not equipotent, since the latter can answer the query Q_1, which cannot be answered by the former. The following theorem suggests an algorithm for testing p-containment.

Theorem 6.2.1 *Let $\mathcal{V}$ and $\mathcal{W}$ be two view sets. $\mathcal{V} \preceq_p \mathcal{W}$ iff for every view $V \in \mathcal{V}$, if treated as a query, V is answerable by $\mathcal{W}$.* □

Proof: "If": Clearly, each view $V \in \mathcal{V}$, if treated as a query, can be answered using the view V itself. By the definition of $\mathcal{V} \preceq_p \mathcal{W}$, view V is also answerable by $\mathcal{W}$. "Only If": For any query Q that is answerable using $\mathcal{V}$, assume Q can be written as

$$ans(\bar{X}) \text{ :- } v_{i_1}(\bar{X}_1), \ldots, v_{i_k}(\bar{X}_k)$$

where each v_{i_j} is the head of a view V_{i_j} in $\mathcal{V}$, which is equivalent to some conjunctive query using the views in $\mathcal{W}$. Unify $v_{i_j}(X_j)$ with the head of that conjunctive query to get a body that replaces $v_{i_j}(X_j)$. Then Q is equivalent to the resulting conjunctive query using the views in $\mathcal{W}$. ∎

The importance of this theorem is that given two view sets $\mathcal{V}$ and $\mathcal{W}$, we can test $\mathcal{V} \preceq_p \mathcal{W}$ simply by checking if every view in $\mathcal{V}$ is answerable by $\mathcal{W}$. That is, we can just consider a *finite* set of queries, even though $\mathcal{V} \preceq_p \mathcal{W}$ means that $\mathcal{W}$ can answer *all* the infinite number of queries that can be answered by $\mathcal{V}$. We can use the algorithms in [DG97, GM99a, LRO96, Qia96] to do the checking. It can be shown that the problem of checking p-containment is $\mathcal{NP}$-hard. It is also easy to see that the relationship "$\preceq_p$" is reflexive, antisymmetric, and transitive.

EXAMPLE 6.2.1 As we saw in Example 6.1.1, view V_3 is answerable by the view V_2. By Theorem 6.2.1, we have $\{V_1, V_2, V_3\} \preceq_p \{V_1, V_2\}$. Clearly the other direction is also true, so $\{V_1, V_2\} \asymp_p \{V_1, V_2, V_3\}$. On the other hand, V_2 cannot be answered using $\{V_1, V_3\}$, which means $\{V_1, V_2, V_3\} \not\preceq_p \{V_1, V_3\}$. □

We are interested in the relationship between p-containment and the traditional concept of query containment. Before making the comparisons, we first generalize the latter to a concept called *v-containment*.

6.2.2 v-containment

We use v-containment to capture the fact that one view set stores a superset of tuples of another view set. The motivation for introducing v-containment, rather than using the traditional concept of query containment (as in Definition 2.1.1), is to cover the cases where the views in a set have different schemas.

Definition 6.2.2 (v-containment and v-equivalence) A view set $\mathcal{V}$ is *v-contained* in another view set $\mathcal{W}$, denoted by $\mathcal{V} \sqsubseteq_v \mathcal{W}$, if the following holds. For any database D of the base relations, if tuple t is in $V(D)$ for a view $V \in \mathcal{V}$, then there exists a view $W \in \mathcal{W}$, such that $t \in W(D)$. The two sets are *v-equivalent*, if $\mathcal{V} \sqsubseteq_v \mathcal{W}$, and $\mathcal{W} \sqsubseteq_v \mathcal{V}$. □

In Example 6.1.1, the two view sets $\{V_1, V_2, V_3\}$ and $\{V_1, V_3\}$ are v-equivalent, while their views have different schemas. The definition of v-containment allows the views in the sets to have different schemas as well, unlike conventional query containment. Given two view sets $\mathcal{V}$ and $\mathcal{W}$, we want to test $\mathcal{V} \sqsubseteq_v \mathcal{W}$. The following theorem shows that this test can be conducted easily for conjunctive views.

Theorem 6.2.2 *Let $\mathcal{V}$ and $\mathcal{W}$ be two sets of conjunctive views. Then $\mathcal{V} \sqsubseteq_v \mathcal{W}$ if and only if for every view $V \in \mathcal{V}$, there is a view $W \in \mathcal{W}$, such that $V \sqsubseteq W$.* □

Proof: The "If" part is obvious. To prove the "Only If" part, we partition the views of $\mathcal{V}$ (resp. $\mathcal{W}$) into subsets of views $\mathcal{V}_1, \ldots, \mathcal{V}_m$ (resp. $\mathcal{W}_1, \ldots, \mathcal{W}_n$), such that the views in each subset have the same schema, i.e., the same number of arguments. For each subset $\mathcal{V}_i$ of $\mathcal{V}$, let $\mathcal{W}_i$ be the subset of $\mathcal{W}$ with the same schema as $\mathcal{V}_i$. Since $\mathcal{V} \sqsubseteq_v \mathcal{W}$, for any database, for any tuple t in a view in $\mathcal{V}_i$, there must exist a view W in $\mathcal{W}_i$ that yields t, since $\mathcal{W}_i$ includes all the views in $\mathcal{W}$ with the schema of t. Therefore, as unions of conjunctive queries, $\mathcal{V}_i$ is contained in $\mathcal{W}_i$. Based on the results in [SY80], for every view V in $\mathcal{V}_i$, there is a view W in $\mathcal{W}_i$, such that $V \sqsubseteq W$. ■

For more complicated queries, such as conjunctive queries with arithmetic comparisons, unions of conjunctive queries, and datalog queries, it becomes more challenging to test v-containment. For example, [Gup94] shows that if conjunctive views with arithmetic comparisons are considered, two views can "team up" to contain one view, while neither of them contains the view by itself.

6.2.3 Comparison Between p-containment and v-containment

Example 6.1.1 shows that v-containment does not imply p-containment, and vice-versa. One might guess that if we do not allow projections in the view definitions (i.e., all the variables in the body of a view appear in the head), then v-containment could imply p-containment. However, the following example shows that this guess is incorrect.

EXAMPLE 6.2.2 Let $e(X_1, X_2)$ be a base relation, where a tuple $e(x, y)$ means that there is an edge from vertex x to vertex y in a graph. Consider the following two view sets:

$$\begin{array}{lll} \mathcal{V} = \{V_1\}, & V_1: \quad v_1(A,B,C) & \text{:-}\ e(A,B), e(B,C), e(A,C) \\ \mathcal{W} = \{W_1\}, & W_1: \quad w_1(A,B,C) & \text{:-}\ e(A,B), e(B,C) \end{array}$$

As illustrated by Figure 6.1, view V_1 stores all the subgraphs shown in Figure 6.1(a), while view W_1 stores all the subgraphs shown in Figure 6.1(b). Although the two views do not have projections in their definitions, still $\mathcal{V} \sqsubseteq_v \mathcal{W}$, and $\mathcal{V} \not\sqsubseteq_p \mathcal{W}$, since V_1 cannot be answered using W_1. □

The following example shows that p-containment does not imply v-containment, even if the views in the sets have the same schemas.

(a) View V_1 (b) View W_1

Figure 6.1: Diagram for the two views in Example 6.2.2.

EXAMPLE 6.2.3 Let $r(X_1, X_2)$ and $s(Y_1, Y_2)$ be two base relations on which two view sets are defined:

$$\begin{array}{llll}
\mathcal{V} = \{V_1\}, & V_1: & v_1(A,C) & \text{:- } r(A,B), s(B,C) \\
\mathcal{W} = \{W_1, W_2\}, & W_1: & w_1(A,B) & \text{:- } r(A,B) \\
 & W_2: & w_2(B,C) & \text{:- } s(B,C)
\end{array}$$

Clearly $\mathcal{V} \not\sqsubseteq_v \mathcal{W}$, but $\mathcal{V} \preceq_p \mathcal{W}$, since there is a rewriting of V_1 using $\mathcal{W}$:

$$v_1(A,C) \text{ :- } w_1(A,B), w_2(B,C)$$

□

6.2.4 Minimizing View Set without Losing Power

Definition 6.2.3 (equipotent minimal subsets) A subset M of a view set $\mathcal{V}$ is an *equipotent minimal subset* (EMS for short) of $\mathcal{V}$ if $M \asymp_p \mathcal{V}$, and for any $V \in M : M - \{V\} \not\asymp_p \mathcal{V}$. □

Informally, an equipotent minimal subset of a view set $\mathcal{V}$ is a minimal subset that is as powerful as $\mathcal{V}$. For instance, in Example 6.1.1, the view set $\{V_1, V_2\}$ is an EMS of $\{V_1, V_2, V_3\}$. We can compute an EMS of $\mathcal{V}$ using the following Shrinking algorithm.

> Algorithm Shrinking initially sets $M = \mathcal{V}$. For each view $V \in M$, it checks if V is answerable by the views $M - \{V\}$. If so, it removes V from M. It repeats this process until no more views can be removed from M, and returns the resulting M as an EMS of $\mathcal{V}$.

It can be shown that the problem of finding an EMS is $\mathcal{NP}$-hard. The following example shows that, as suspected, a view set may have multiple EMS's.

EXAMPLE 6.2.4 Suppose $r(A, B)$ is a base relation, on which the following three views are defined:

$$\begin{array}{lll} V_1: & v_1(A) & \text{:- } r(A, B) \\ V_2: & v_2(B) & \text{:- } r(A, B) \\ V_3: & v_3(A, B) & \text{:- } r(A, X), r(Y, B) \end{array}$$

Let $\mathcal{V} = \{V_1, V_2, V_3\}$. As shown by the following rewritings, $\mathcal{V}$ has two EMS's: $\{V_1, V_2\}$, and $\{V_3\}$.

$$\begin{array}{llll} \text{rewrite } V_1 \text{ using } V_3: & P_1: & v_1(A) & \text{:- } v_3(A, B) \\ \text{rewrite } V_2 \text{ using } V_3: & P_2: & v_2(B) & \text{:- } v_3(A, B) \\ \text{rewrite } V_3 \text{ using } \{V_1, V_2\}: & P_3: & v_3(A, B) & \text{:- } v_1(A), v_2(B) \end{array}$$

For instance, P_3 is an equivalent rewriting of V_3 because its expansion:

$$v_3(A, B) \text{ :- } r(A, B'), r(A', B)$$

can be shown equivalent to V_3. □

In many applications, each view is associated with a cost, such as its storage space, or the number of Web pages that need to be crawled for the view [CGM00]. We often need to find an EMS that is optimal, i.e., the total cost of selected views is minimum. How to find an EMS with the lowest cost is still an open problem.

6.3 Testing p-containment Relative to a Query Set

Till now, we have considered p-containment between two view sets with respect to a "universal" set of queries, i.e., users can ask *any* query on the base relations. However, in some scenarios, users are restricted in the queries they can ask. In this section, we consider the relationship between two view sets with respect to a given set of queries. In particular, we consider infinite query sets defined by finite, parameterized queries.

6.3.1 Relative p-containment

Definition 6.3.1 (**relative p-containment**) Given a (possibly infinite) set of queries $\mathcal{Q}$, a view set $\mathcal{V}$ is p-contained in a view set $\mathcal{W}$ w.r.t. $\mathcal{Q}$, denoted by $\mathcal{V} \preceq_{\mathcal{Q}} \mathcal{W}$, if and only if for any query $Q \in \mathcal{Q}$ that is answerable by $\mathcal{V}$, Q is also answerable by $\mathcal{W}$. The two view sets are equipotent w.r.t. $\mathcal{Q}$, denoted by $\mathcal{V} \asymp_{\mathcal{Q}} \mathcal{W}$, if $\mathcal{V} \preceq_{\mathcal{Q}} \mathcal{W}$ and $\mathcal{W} \preceq_{\mathcal{Q}} \mathcal{V}$. □

EXAMPLE 6.3.1 Assume we have two relations $car(Make, Dealer)$ and $loc(Dealer, City)$ that store information about cars, their dealers, and located cities. Consider the following queries and views:

$$
\begin{array}{llll}
\text{Queries:} & Q_1: & q_1(D,C) & \text{:- } car(toyota, D), loc(D,C) \\
 & Q_2: & q_2(D,C) & \text{:- } car(honda, D), loc(D,C) \\
\text{Views:} & W_1: & w_1(D,C) & \text{:- } car(toyota, D), loc(D,C) \\
 & W_2: & w_2(D,C) & \text{:- } car(honda, D), loc(D,C) \\
 & W_3: & w_3(M,D,C) & \text{:- } car(M, D), loc(D,C)
\end{array}
$$

Let $\mathcal{Q} = \{Q_1, Q_2\}$, $\mathcal{V} = \{W_1, W_2\}$, and $\mathcal{W} = \{W_3\}$. Then $\mathcal{V}$ and $\mathcal{W}$ are equipotent w.r.t. $\mathcal{Q}$, since the following equivalent rewritings show that both Q_1 and Q_2 can be answered by $\mathcal{V}$ as well as $\mathcal{W}$, respectively.

$$
\begin{array}{llll}
\text{Rewritings using } \mathcal{V}: & Q_1: & q_1(D,C) & \text{:- } w_1(D,C) \\
 & Q_2: & q_2(D,C) & \text{:- } w_2(D,C) \\
\text{Rewritings using } \mathcal{W}: & Q_1: & q_1(D,C) & \text{:- } w_3(toyota, D, C) \\
 & Q_2: & q_2(D,C) & \text{:- } w_3(honda, D, C)
\end{array}
$$

Note that $\mathcal{V}$ and $\mathcal{W}$ are not equipotent in general. □

Given a view set $\mathcal{V}$ and a query set $\mathcal{Q}$, we define an *equipotent minimal subset* (EMS) of $\mathcal{V}$ w.r.t. $\mathcal{Q}$ as follows. A subset M of $\mathcal{V}$ is an EMS of $\mathcal{V}$ w.r.t. $\mathcal{Q}$ if $M \asymp_{\mathcal{Q}} \mathcal{V}$, and for any $V \in M$: $M - \{V\} \not\asymp_{\mathcal{Q}} \mathcal{V}$. We can compute an EMS of $\mathcal{V}$ w.r.t. $\mathcal{Q}$ in the same way as in Section 6.2.4, if we have a method to test relative p-containment. This testing is straightforward when $\mathcal{Q}$ is finite. By definition, we can check for each query $Q_i \in \mathcal{Q}$ that is answerable by $\mathcal{V}$, whether Q_i is also answerable by $\mathcal{W}$. However, if $\mathcal{Q}$ is infinite, testing relative p-containment becomes more challenging, since we cannot use this enumerate-and-test paradigm for all the queries in $\mathcal{Q}$. In the rest of this section, we consider ways to test relative p-containment w.r.t. infinite query sets generated by finite parameterized queries.

6.3.2 Parameterized Queries

A *parameterized query* is a conjunctive query that contains *placeholders* in the argument positions of its body, in addition to constants and variables. A placeholder is denoted by an argument name beginning with a "$" sign. The following is an example.

EXAMPLE 6.3.2 Consider the following parameterized query Q on the two relations in Example 6.3.1:

$$Q : q(D) \text{ :- } car(\$M, D), loc(D, \$C)$$

This query represents all the following queries: a user gives a car make m for the placeholder $\$M$, and a city c for the placeholder $\$C$, and asks for the dealers of the make m in the city c. For instance, the following are two instances of Q:

$$\begin{array}{lll} I_1: & q(D) & \text{:- } car(toyota, D), loc(D, sf) \\ I_2: & q(D) & \text{:- } car(honda, D), loc(D, sf) \end{array}$$

which respectively ask for the dealers of Toyota and Honda in San Francisco. □

In general, each *instance* of a parameterized query Q is obtained by assigning a constant from the corresponding domain to each placeholder. If a placeholder appears in different argument positions, then the same constant must be used in these positions. Let $IS(Q)$ denote the set of all instances of the query Q. We assume that the domains of placeholders are infinite (independent of an instance of the base relations), causing $IS(Q)$ to be infinite. Thus we can represent an infinite set of queries using a finite set of parameterized queries.

EXAMPLE 6.3.3 Consider the following three views:

$$\begin{array}{lll} V_1: & v_1(M, D, C) & \text{:- } car(M, D), loc(D, C) \\ V_2: & v_2(M, D) & \text{:- } car(M, D), loc(D, sf) \\ V_3: & v_3(M) & \text{:- } car(M, D), loc(D, sf) \end{array}$$

Clearly, view V_1 can answer all instances of Q, since it includes information for cars and dealers in all cities. View V_2 cannot answer all instances, since it has only the information about dealers in San Francisco. But it can answer instances of the following more restrictive parameterized query, which replaces the placeholder $\$C$ by sf:

$$Q' : q(D) \text{ :- } car(\$M, D), loc(D, sf)$$

That is, the user can only ask for information about dealers in San Francisco. Finally, view V_3 cannot answer any instance of Q, since it does not have the *Dealer* attribute in its head. □

Given a finite set of parameterized queries $\mathcal{Q}$ and two view sets $\mathcal{V}$ and $\mathcal{W}$, we want to test $\mathcal{V} \preceq_{IS(Q)} \mathcal{W}$. The example above suggests the following test strategy:

1. Deduce *all* instances of $\mathcal{Q}$ that can be answered by $\mathcal{V}$, represented by a finite set of parameterized queries.

2. Test if $\mathcal{W}$ can answer *all* such instances.

In the next two subsections we show how to perform each of these steps. We show that all answerable instances of a parameterized query for a given view set can be represented by a finite set of parameterized queries. We give an algorithm for deducing this set, and an algorithm for the second step. Although our discussion is based on one parameterized query, the results can be easily generalized to a finite set of parameterized queries.

6.3.3 Complete Answerability of Parameterized Queries

We first consider the problem of testing whether all instances of a parameterized query can be answered by a view set. If all instances of a parameterized query Q can be answered by a view set $\mathcal{V}$, we say that Q is *completely answerable* by $\mathcal{V}$.

Definition 6.3.2 (**canonical instance**) Let Q be a parameterized query and $\mathcal{V}$ be a view set. A *canonical instance* of Q (given $\mathcal{V}$) is an instance of Q, in which each placeholder is replaced by a new distinct constant that does not appear in Q and $\mathcal{V}$. $\Box$

Theorem 6.3.1 *Let Q be a parameterized query, and $\mathcal{V}$ be a view set. Q is completely answerable by $\mathcal{V}$ if and only if $\mathcal{V}$ can answer a canonical instance of Q (given $\mathcal{V}$).* $\Box$

Proof: Let Q_c be a canonical instance of Q given $\mathcal{V}$. The "Only if" part is obvious, since Q_c is an instance of Q. To prove the "If" part, we need to show that if there exists a rewriting P_c of the instance Q_c using $\mathcal{V}$, then for any instance I of query Q, there exists a rewriting P_I of I using $\mathcal{V}$. As shown in Figure 6.2, the rewriting P_I is constructed as follows. For each placeholder $\$A_i$ in Q, suppose it is replaced by a_i in Q_c, and by b_i in the instance I. Replace each a_i in the rewriting P_c with b_i, and we do this replacement for every placeholder in Q. Let P_I be the new rewriting after the replacements.

Now we prove that P_I is an equivalent rewriting of I using $\mathcal{V}$. Consider the expansion P_c^{exp} of the rewriting P_c. Since P_c^{exp} is equivalent to Q_c, there exists a containment mapping ν from Q_c to P_c^{exp}, and a containment mapping μ of the other direction. Note that a containment mapping must map a constant to the same constant, and map a variable to

Parameterized query Q:		$ans() \text{ :- } r_1(), \ldots, r_l(\ldots, \$A_i, \ldots), \ldots, r_n()$
Canonical instance Q_c:		$ans() \text{ :- } r_1(), \ldots, r_l(\ldots, a_i, \ldots), \ldots, r_n()$
Rewriting P_c:	μ	$ans() \text{ :- } v_1(), \ldots, v_p(\ldots, a_i, \ldots), \ldots, v_m()$ ν
Expansion P_c^{exp}:		$ans() \text{ :- } r_{j_1}(), \ldots, r_{j_q}(\ldots, a_i, \ldots), \ldots, r_{j_k}()$
Any instance I:		$ans() \text{ :- } r_1(), \ldots, r_l(\ldots, b_i, \ldots), \ldots, r_n()$
Rewriting P_I:	μ'	$ans() \text{ :- } v_1(), \ldots, v_p(\ldots, b_i, \ldots), \ldots, v_m()$ ν'
Expansion P_I^{exp}:		$ans() \text{ :- } r_{j_1}(), \ldots, r_{j_q}(\ldots, b_i, \ldots), \ldots, r_{j_k}()$

Figure 6.2: Proof of Theorem 6.3.1.

either a constant, or another variable. In addition, the expansion P_I^{exp} of P_I can be obtained from P_c^{exp} by replacing every occurrence of a_i with b_i.

We can construct a new mapping ν' from I to P_I^{exp} as follows. ν' is the same as ν except that for each constant b_i in I that replaces a placeholder $\$A_i$ of Q, we let $\nu'(b_i) = b_i$. By the construction of P_c and P_I, we can show that ν' is a containment mapping from I to P_I^{exp}. Similarly we can derive from μ a containment mapping μ' from P_I^{exp} to I. Thus P_I is an equivalent rewriting of I using $\mathcal{V}$. ∎

The theorem suggests an algorithm *TestComp* for testing whether all instances of a parameterized query Q can be answered by a view set $\mathcal{V}$.

> Algorithm TestComp first constructs a canonical instance Q_c of Q (given $\mathcal{V}$). Then it tests if Q_c can be answered using $\mathcal{V}$ by calling an algorithm of answering queries using views, such as those in [LRO96, GM99a, Qia96, DG97]. It outputs "yes" if $\mathcal{V}$ can answer Q_c; otherwise, it outputs "no."

EXAMPLE 6.3.4 Consider the parameterized query Q in Example 6.3.3. To test whether view V_1 can answer all instances of Q, we use two new distinct constants m_0 and c_0 to replace the two placeholders $\$M$ and $\$C$, and obtain the following canonical instance:

$$Q_c : q(D) \text{ :- } car(m_0, D), loc(D, c_0)$$

Clearly Q_c can be answered by view V_1, because of the following equivalent rewriting of Q_c:

$$P_c : q(D) \text{ :- } v_1(m_0, D, c_0)$$

By Theorem 6.3.1, view V_1 can answer all instances of Q. In addition, since V_2 cannot answer Q_c (which is also a canonical instance of Q given V_2), it cannot answer *some* instances of Q. The same argument holds for V_3. □

6.3.4 Partial Answerability of Parameterized Queries

As shown by the view V_2 and the query Q in Example 6.3.3, even if a view set cannot answer all instances of a parameterized query, it can still answer some of its instances. In general, we want to know what instances can be answered by the view set, and whether these instances can also be represented as a set of more "restrictive" parameterized queries. A parameterized query Q_1 is more *restrictive* than a parameterized query Q if every instance of Q_1 is also an instance of Q. For example, query

$$q(D) \text{ :- } car(\$M, D), loc(D, sf)$$

is more restrictive than query

$$q(D) \text{ :- } car(\$M, D), loc(D, \$C)$$

since the former requires the second argument of the loc subgoal to be sf, while the latter allows any constant for its corresponding placeholder $\$C$. For another example, query

$$q(M, C) \text{ :- } car(M, \$D_1), loc(\$D_1, C)$$

is more restrictive than query

$$q(M, C) \text{ :- } car(M, \$D_1), loc(\$D_2, C)$$

since the former has one placeholder in two argument positions, while the latter allows two different constants to be assigned to its two placeholders.

Clearly all the parameterized queries that are more restrictive than Q can be generated by adding the following two types of restrictions:

1. Type I: Some placeholders are assigned the same constant. Let $\{\$A_1, \ldots, \$A_k\}$ be *some* placeholders in Q. We can put a restriction $\$A_1 = \cdots = \$A_k$ on the query Q. That is, we can replace all these k placeholders with one placeholder.

2. Type II: For a placeholder $\$A_i$ in Q and a constant c in Q or $\mathcal{V}$, we put a restriction $\$A_i = c$ on Q. That is, the user can only assign constant c to this placeholder in an instance.

Consider all the possible (finite) combinations of these two types of restrictions. For example, suppose Q has two placeholders, $\{\$A_1, \$A_2\}$, and Q and $\mathcal{V}$ have one constant c. Then we consider the following restriction combinations: $\{\}$, $\{\$A_1 = \$A_2\}$, $\{\$A_1 = c\}$, $\{\$A_2 = c\}$, and $\{\$A_1 = \$A_2 = c\}$. Note that we allow a combination to have restrictions of only one type. In addition, each restriction combination is consistent, in the sense that it does not have a restriction $\$A_1 = \$A_2$ and two restrictions $\$A_1 = c_1$ and $\$A_2 = c_2$, while c_1 and c_2 are two different constants in Q and $\mathcal{V}$. For each restriction combination RC_i, let $Q(RC_i)$ be the parameterized query that is derived by adding the restrictions in RC_i to Q. Clearly $Q(RC_i)$ is a parameterized query that is more restrictive than Q. Let $\Phi(Q, \mathcal{V})$ denote all these parameterized queries that are more restrictive than Q.

Suppose I is an instance of Q that can be answered by $\mathcal{V}$. We can show that there exists a parameterized query $Q_i \in \Phi(Q, \mathcal{V})$, such that I is a canonical instance of Q_i. By Theorem 6.3.1, Q_i is completely answerable by $\mathcal{V}$. Therefore, we have proved the following theorem:

Theorem 6.3.2 *All instances of a parameterized query Q that are answerable by a view set $\mathcal{V}$ can be generated by a finite set of parameterized queries that are more restrictive than Q, such that all these parameterized queries are completely answerable by $\mathcal{V}$.* □

We propose the following algorithm GenPartial. Given a parameterized query Q and a view set $\mathcal{V}$, the algorithm generates all the parameterized queries that are more restrictive than Q, such that they are completely answerable by $\mathcal{V}$, and they generate all the instances of Q that are answerable by $\mathcal{V}$.

> Algorithm GenPartial first generates all the restriction combinations, and creates a parameterized query for each combination. Then it calls the algorithm TestComp to check if this parameterized query is completely answerable by $\mathcal{V}$. It outputs all the parameterized queries that are completely answerable by $\mathcal{V}$.

6.3.5 Testing p-containment Relative to Finite Parameterized Queries

Now we give an algorithm for testing p-containment relative to parameterized queries. Let $\mathcal{Q}$ be a query set with only one parameterized query Q. Let $\mathcal{V}$ and $\mathcal{W}$ be two view sets. The algorithm tests $\mathcal{V} \preceq_{IS(Q)} \mathcal{W}$ as follows. First call the algorithm GenPartial to find all the more restrictive parameterized queries of Q that are completely answerable by $\mathcal{V}$. For each of them, call the algorithm TestComp to check if it is also completely answerable by $\mathcal{W}$. By definition, $\mathcal{V} \preceq_{IS(Q)} \mathcal{W}$ iff all these parameterized queries that are completely answerable by $\mathcal{V}$ are also completely answerable by $\mathcal{W}$. The algorithm can be easily generalized to the case where $\mathcal{Q}$ is a finite set of parameterized queries.

6.4 MCR-containment

So far we have considered the query-answering power of views with respect to equivalent rewritings of queries. In information integration, we often need to consider maximally-contained rewritings of a query using views. (See Section 2.5 for the definition of maximally-contained rewritings.) In this section, we introduce the concept of *MCR-containment*, which describes the relative power of two view sets in terms of their maximally-contained rewritings of queries. Surprisingly, MCR-containment is essentially the same as p-containment.

Definition 6.4.1 (**MCR-containment**) A view set $\mathcal{V}$ is *MCR-contained* in another view set $\mathcal{W}$, denoted by $\mathcal{V} \preceq_{MCR} \mathcal{W}$, if for any query Q, we have $MCR(Q, \mathcal{V}) \sqsubseteq MCR(Q, \mathcal{W})$, where $MCR(Q, \mathcal{V})$ and $MCR(Q, \mathcal{W})$ are MCR's of Q using $\mathcal{V}$ and $\mathcal{W}$, respectively. The two sets are *MCR-equipotent*, denoted by $\mathcal{V} \asymp_{MCR} \mathcal{W}$, if $\mathcal{V} \preceq_{MCR} \mathcal{W}$, and $\mathcal{W} \preceq_{MCR} \mathcal{V}$. □

The following theorem shows that MCR-containment is essentially the same as p-containment.

Theorem 6.4.1 *For two view sets $\mathcal{V}$ and $\mathcal{W}$, $\mathcal{V} \preceq_p \mathcal{W}$ if and only if $\mathcal{V} \preceq_{MCR} \mathcal{W}$.* □

Proof: "If": Suppose $\mathcal{V} \preceq_{MCR} \mathcal{W}$. Consider each view $V \in \mathcal{V}$. Clearly V itself is an MCR of the query V using $\mathcal{V}$, since it is an equivalent rewriting of V. Let $MCR(V, \mathcal{W})$ be an MCR of V using $\mathcal{W}$. Since $\mathcal{V} \preceq_{MCR} \mathcal{W}$, we have $V \sqsubseteq MCR(V, \mathcal{W})$. On the other hand, by the definition of MCR's, $MCR(V, \mathcal{W}) \sqsubseteq V$. Thus $MCR(V, \mathcal{W})$ and V are equivalent, and $MCR(V, \mathcal{W})$ is an equivalent rewriting of V using $\mathcal{W}$. By Theorem 6.2.1, $\mathcal{V} \preceq_p \mathcal{W}$.

"Only if": Suppose $\mathcal{V} \preceq_p \mathcal{W}$. By Theorem 6.2.1, every view has an equivalent rewriting using $\mathcal{W}$. For any query Q, let $MCR(Q, \mathcal{V})$ and $MCR(Q, \mathcal{W})$ be MCR's of Q using $\mathcal{V}$ and $\mathcal{W}$, respectively. We replace each view in $MCR(Q, \mathcal{V})$ with its corresponding rewriting using $\mathcal{W}$, and obtain a new rewriting MCR' of query Q using $\mathcal{W}$, which is equivalent to $MCR(Q, \mathcal{V})$. By the definition of MCR's, we have $MCR' \sqsubseteq MCR(Q, \mathcal{W})$. Thus $MCR(Q, \mathcal{V}) \sqsubseteq MCR(Q, \mathcal{W})$, and $\mathcal{V} \preceq_{MCR} \mathcal{W}$. ∎

6.5 Minimizing View Sets of General Queries

Until now, we have discussed minimizations of conjunctive views using the concept of p-containment. In this section we extend the earlier results to more general queries, and propose a framework for minimizing view sets without losing their query-answering power.

Let $\mathcal{F}$ be one of the following three families of queries: conjunctive queries with arithmetic comparisons, unions of conjunctive queries, and datalog queries. Suppose $Q \in \mathcal{F}$ is a query on base relations, and $\mathcal{V}$ is a set of views in $\mathcal{F}$ on the same base relations.

Definition 6.5.1 (**general answerability**) We say Q is *answerable* by $\mathcal{V}$ if there is a query $P \in \mathcal{F}$ that uses only the views in $\mathcal{V}$, such that for any database D of the base relations, the answer computed by P on the instance of the views $\mathcal{V}(D)$ is equivalent to the answer computed by Q on the base relations. □

Definition 6.5.2 (**general p-containment and equipotence**) Let $\mathcal{V}$ and $\mathcal{W}$ be two sets of views in $\mathcal{F}$. We say $\mathcal{V}$ is *p-contained* in $\mathcal{W}$, denoted by $\mathcal{V} \preceq_p \mathcal{W}$, if any query in $\mathcal{F}$ that is answerable by $\mathcal{V}$ is also answerable by $\mathcal{W}$. The two view sets are *equipotent*, denoted by $\mathcal{V} \asymp_p \mathcal{W}$, if $\mathcal{V} \preceq_p \mathcal{W}$, and $\mathcal{W} \preceq_p \mathcal{V}$. □

6.5.1 Testing General p-containment

Theorem 6.5.1 *Under the definition of general p-containment, Theorem 6.2.1 holds. That is, for two sets of views $\mathcal{V}$ and $\mathcal{W}$, $\mathcal{V} \preceq_p \mathcal{W}$ if and only if for every view $V \in \mathcal{V}$, V is answerable by $\mathcal{W}$.* □

Suppose we have an algorithm $\mathsf{Check}_{\mathcal{F}}$ that can test the *answerability* of a query with respect to a set of views in the family $\mathcal{F}$. That is, given a query Q and a view set $\mathcal{V}$, as shown in Figure 6.3, the algorithm $\mathsf{Check}_{\mathcal{F}}$ tests if Q is answerable by $\mathcal{V}$. By Theorem 6.5.1,

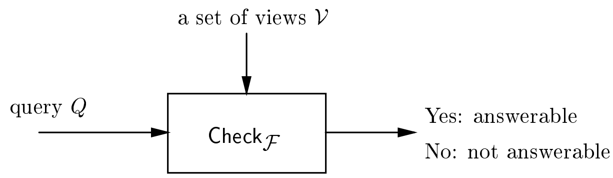


Figure 6.3: The algorithm $\mathsf{Check}_{\mathcal{F}}$ for a family $\mathcal{F}$ of queries.

we can test $\mathcal{V} \preceq_p \mathcal{W}$ as follows: for each view $V \in \mathcal{V}$, call the algorithm $\mathsf{Check}_{\mathcal{F}}$ to test whether V is answerable by $\mathcal{W}$. $\mathcal{V} \preceq_p \mathcal{W}$ is true iff all the views in $\mathcal{V}$ pass the test. It should be observed that algorithm $\mathsf{Check}_{\mathcal{F}}$ does not exist for the family of datalog queries [Dus97].

6.5.2 Minimizing View Sets Using General Equipotence

Given a view set $\mathcal{V}$, similarly to Definition 6.2.3, an equipotent minimal subset (EMS) of $\mathcal{V}$ is a minimal subset of $\mathcal{V}$ that is equipotent to $\mathcal{V}$. To compute an EMS of $\mathcal{V}$, we check for each view $V \in \mathcal{V}$ if V is answerable by $\mathcal{V} - \{V\}$ by calling the algorithm $\mathsf{Check}_{\mathcal{F}}$. If so, remove V from $\mathcal{V}$. We keep shrinking the view set until no views can be removed without changing the query-answering power, and the final set is an EMS of $\mathcal{V}$. As shown in Example 6.2.4, a view set can have multiple EMS's.

In some scenarios we need to answer queries using views in the presence of constraints (e.g., functional dependencies [Cod70], multivalued dependencies [Fag77, Del78]) and binding limitations on views [RSU95, DL97, Ull89, LC00, LYV+98, YLUGM99]. We can generalize our results by requiring that the algorithm $\mathsf{Check}_{\mathcal{F}}$ take constraints and binding limitations into consideration. [RSU95, Gry99] give algorithms for answering conjunctive queries using conjunctive views in these scenarios; [Dus97] provides answers for rewriting datalog queries using conjunctive views.

6.6 Conclusions and Related Work

In this chapter we studied how to use mediator caching to reduce the number of source accesses. One problem is to decide what cached query results to keep so that we can answer as many future queries as possible. We showed that when minimizing a set of query results (views), we should consider the query-answering power of the views, rather than using the traditional query-containment concept for view selection. We developed the concept of p-containment, and showed that it is not related to the traditional query containment. We discussed how to minimize a view set without losing its query-answering power. We also considered p-containment of two view sets with respect to a given (possibly infinite) set of queries, and with respect to maximally-contained rewritings of queries using the views. We developed algorithms for testing these containments, and discussed their relationships.

Related Work

There have been many studies on data caching (e.g., [ACPS96, Bas98, BCF+99]). One key difference between our work and the earlier studies is that we focus on how to keep query results in a mediator cache to maximize the chance of answering future queries at the mediator. In addition, our problem is closely related to the problem of selecting views to materialize. In [IK93, HGMW+95, Wid95, CW91], a data warehouse is modeled as a repository of integrated information available for querying and analysis. The materialized views are stored with an intention of making the frequent queries fast. The system accesses base relations for a query that cannot be answered using the materialized views. Each of the materialized views can be huge, resulting in the need for careful selection under the constraint of limited disk space [HRU96, GHRU97, RSS96, YKL97, BPT97, TS97]. Moreover, base relations can change over time. The updates have to be propagated to the materialized views, resulting in a maintenance cost [GHRU97, GM99b]. The study in this setting has, therefore, emphasized on modeling the view-selection problem as cost-benefit analysis. Thus, for a given set of queries, various sets of sub-queries are considered for materialization. Redundant views in a set that increase cost are deduced using query containment, and an optimal subset is chosen.

Such a model is feasible when all queries can be answered in the worst case by accessing the base relations, and not by views alone. This assumption is incorporated in the model by replicating base relations at the warehouse, without taking the costs of such a step into account. Thus, the base relations themselves are considered to be normalized, independent, and minimal. However, when real-time access to base relations is prohibitive, such an approach can lead to wrong conclusions, as was seen in Example 6.1.1. In such a scenario, it is essential to ensure the *computability* of queries using *only* maintained views.

Our results are applicable in the scenarios where the following assumptions hold. (1)

Real-time access to base relations is prohibitive, or possibly denied, and (2) cached views are expensive to maintain over time, because of the high costs of propagating changes from base relations to views. Therefore, while minimizing a view set, it is important to retain its query-answering power. We believe the power of answering queries and the benefit/costs of a view set are orthogonal issues, and their interplay would make an interesting work in its own right.

Recently, [CG00] has proposed solutions to the following problem of *database reformulation*: given a query on base relations, how can we materialize views to answer the query? The authors show that there can be infinite number of view sets that can answer the same query. Thus, their work starts with a given query but no views. In our framework, we assume that a set of views (query results) are given, and user queries can be arbitrary. We would like to deduce a minimal subset of views that can answer all queries answerable by the original set.

Chapter 7

Generating Efficient Plans for Queries Using Views

So far we have discussed query optimization in information-integration systems that take the query-centric approach. For systems that take the source-centric approach, one fundamental problem is: given a query on base relations and a set of views over the same relations, can we answer the query using only the answers to the views? Several algorithms have been proposed in the literature on this problem of answering queries using views.

In this chapter, we study the problem of generating efficient, equivalent rewritings using views to compute the answer to a query. We take the closed-world assumption, in which views are materialized from base relations, rather than views describing sources in terms of abstract predicates, as is common when the open-world assumption is used. In the closed-world model, there can be an infinite number of different rewritings that compute the same answer, yet have quite different performance. Query optimizers take a logical plan (a rewriting of the query) as an input, and generate efficient physical plans to compute the answer. Thus our goal is to generate a small subset of the possible logical plans without missing an optimal physical plan.

We first consider a cost model that counts the number of subgoals in a physical plan, and show a search space that is guaranteed to include an optimal rewriting, if the query has a rewriting in terms of the views. We also develop an efficient algorithm for finding rewritings with the minimum number of subgoals. We then consider a cost model that counts the sizes of intermediate relations of a physical plan, without dropping any attributes, and give a search space for finding optimal rewritings. Our final cost model allows attributes to be dropped in intermediate relations. We show that, by careful variable renaming, it is possible to do better than the standard supplementary-relation approach by dropping attributes that the latter approach would retain. Experiments show that our algorithm of generating optimal rewritings has good efficiency and scalability.

Chapter Organization

In Section 7.1 we give an example to show there can be several different rewritings for a query using the same set of views, and these rewritings could have quite different efficiency. Section 7.2 discusses efficiency of rewritings. In Section 7.3 we study how to find optimal rewritings under cost model M_1, i.e., rewritings with the minimum number of view subgoals. In Section 7.4 we develop an efficient algorithm, called *Corecover*, for finding rewritings with the minimum number of view subgoals.

In Section 7.5 we study cost model M_2 that considers sizes of view relations and intermediate relations in a physical plan. We show that the space of all minimal rewritings that use view tuples is guaranteed to include an optimal rewriting of a query under M_2, if the query has a rewriting. In Section 7.6 we study the search space for optimal rewritings under cost model M_3, and propose a heuristic that helps the optimizer drop more attributes without changing the final answer of the evaluation, thus producing a more efficient physical plan. We show our experimental results in Section 7.7. In Section 7.8 we conclude the chapter and discuss related work.

7.1 Introduction

The following example illustrates several issues in generating optimal rewritings using views for a query under the closed-world assumption:[1] (1) There can be an infinite number of rewritings for a query. (2) Traditional query-containment techniques [CM77] cannot find a rewriting with the minimum number of joins. (3) Adding more view relations to a rewriting could make the rewriting more efficient to evaluate.

EXAMPLE 7.1.1 Suppose we have the following three base relations:

- `car(Make,Dealer)`. A tuple `car(m,d)` means that dealer `d` sells cars of make `m`.

[1] We will refer to this example as the "car-loc-part example" throughout the chapter.

- `loc(Dealer,City)`. A tuple `loc(d,c)` means dealer `d` has a branch in the city `c`.
- `part(Store,Make,City)`. A tuple `part(s,m,c)` means that store `s` in city `c` sells parts for cars of make `m`.

Assume we have five sources, whose contents are defined by the following five views on the base relations:

$$
\begin{array}{lll}
V_1: & v_1(M,D,C) & \text{:- } car(M,D), loc(D,C) \\
V_2: & v_2(S,M,C) & \text{:- } part(S,M,C) \\
V_3: & v_3(S) & \text{:- } car(M,anderson), loc(anderson,C), part(S,M,C) \\
V_4: & v_4(M,D,C,S) & \text{:- } car(M,D), loc(D,C), part(S,M,C) \\
V_5: & v_5(M,D,C) & \text{:- } car(M,D), loc(D,C)
\end{array}
$$

Under CWA, these five views are computed from the three base relations. In particular, views V_1 and V_5 have the same definition, thus their view relations always have the same tuples for any base relations. Under OWA, however, we would only know that V_1 and V_5 contain only tuples in $car(M,D) \bowtie loc(D,C)$; either or both could even be empty.

A user submits the following query:

$$Q:\ q_1(S,C) \text{ :- } car(M,anderson), loc(anderson,C), part(S,M,C)$$

asking for cities and stores that sell parts for car makes in the `anderson` branch in this city.

We want to answer the query using the views. The following are some rewritings for the query. Notice that there is an infinite number of rewritings for the query, since each rewriting P has an infinite number of rewritings that are equivalent to P as queries.

$$
\begin{array}{lll}
P_1: & q_1(S,C) & \text{:- } v_1(M,anderson,C_1), v_1(M_1,anderson,C), v_2(S,M,C) \\
P_2: & q_1(S,C) & \text{:- } v_1(M,anderson,C), v_2(S,M,C) \\
P_3: & q_1(S,C) & \text{:- } v_3(S), v_1(M,anderson,C), v_2(S,M,C) \\
P_4: & q_1(S,C) & \text{:- } v_4(M,anderson,C,S) \\
P_5: & q_1(S,C) & \text{:- } v_1(M,anderson,C_1), v_5(M_1,anderson,C), v_2(S,M,C)
\end{array}
$$

We can show that all of these rewritings compute the answer to the query Q. However, some of them may lack an efficient physical plan. For instance, rewriting P_2 needs one access to the view relation V_1, while P_1 needs two accesses and also a join operation. In addition, we cannot easily minimize P_1 to generate P_2 using traditional query-containment techniques [CM77], since neither of the first two subgoals of P_1 is redundant. Furthermore, although

P_3 uses one more view V_3 than P_2, the former can still produce a more efficient execution plan if the view relation V_3 is very selective. That is, if there are very few stores that sell parts for cars that dealer `anderson` sells, and are located in the same city as `anderson`, then view V_3 can be used as a filtering relation. Rewriting P_4 could be an optimal rewriting, since it requires only one access to view V_4. □

In general, given a query at a mediator and a set of views that define the contents of sources, the following questions arise:

1. In what space should we search for optimal rewritings?
2. How do we find optimal rewritings efficiently?
3. How does an optimizer generate an efficient physical plan from a logical plan by considering the view definitions?

In this chapter we answer these questions by considering several cost models. We define search spaces for finding optimal rewritings, and develop efficient algorithms for finding optimal rewritings in each search space.

7.2 Efficiency of Rewritings

In this section, we introduce the three cost models used in this chapter. Recall Section 2.5 reviewed some concepts on answering queries using views. In this chapter, unless otherwise specified, the term "rewriting" means an "equivalent rewriting" of a query using views. For instance, in our car-loc-part example, both

$$\begin{array}{lll} P_1: & q_1(S,C) & \text{:- } v_1(M, anderson, C_1), v_1(M_1, anderson, C), v_2(S, M, C) \\ P_2: & q_1(S,C) & \text{:- } v_1(M, anderson, C), v_2(S, M, C) \end{array}$$

are two rewritings for the query

$$Q: \ q_1(S,C) \text{ :- } car(M, anderson), loc(anderson, C), part(S, M, C)$$

because their expansions

$$\begin{array}{lll} P_1^{exp}: & q_1(S,C) & \text{:- } car(M, anderson), loc(anderson, C_1), \\ & & \quad car(M_1, anderson), loc(anderson, C), part(S, M, C) \\ P_2^{exp}: & q_1(S,C) & \text{:- } car(M, anderson), loc(anderson, C), part(S, M, C) \end{array}$$

can be shown to be equivalent to Q. This example shows that two equivalent rewritings of the same query might not be equivalent as queries. That is, $P_1^{exp} \equiv P_2^{exp}$, but $P_1 \not\equiv P_2$. Notice that the test for $P_1 \equiv P_2$ involves containment mappings in *views*, while the test for $P_1^{exp} \equiv P_2^{exp}$ involves containment mappings in *base relations*. We say that two rewritings P_1 and P_2 are *equivalent as queries* if $P_1 \equiv P_2$. Whereas, we say that two rewritings P_1 and P_2 are *equivalent as expansions* if $P_1^{exp} \equiv P_2^{exp}$.

Let P be a rewriting of a query Q using views $\mathcal{V}$. We define three cost models, as shown in Table 7.1. For each of them, we define a *physical plan* for P and a *cost measure* on this physical plan.

Cost model	Physical plan	Cost measure
M_1	a set of subgoals	number of subgoals: n
M_2	a list of subgoals	$\sum_{i=1}^{n}(size(g_i) + size(IR_i))$
M_3	a list of subgoals annotated with projected attributes	$\sum_{i=1}^{n}(size(g_i) + size(GSR_i))$

Table 7.1: Three cost models.

Under cost model M_1, a physical plan of P is a set of the view subgoals in P, and the cost measure is the number of subgoals in the set. That is, the cost of a physical plan F is:

$$cost_{M_1}(F) = number\ of\ subgoals\ in\ F$$

The main motivation of cost model M_1 is to minimize the number of join operations, which tend to be expensive when a rewriting is evaluated.

Under cost model M_2, a physical plan F of rewriting P is a list $g_1, \ldots, g_n$ of the view subgoals in P. The views corresponding to these subgoals are joined in the order listed. After joining the first i subgoals in the list, the *intermediate relation* IR_i is the join result with all attributes retained [GMUW00]. The cost measure for F under M_2 is the sum of the sizes of the views joined, plus the sizes of the intermediate relations computed during the multiway join. More formally, The cost measure of F under M_2 is:

$$cost_{M_2}(F) = \sum_{i=1}^{n}(size(g_i) + size(IR_i))$$

where $size(g_i)$ is the size of the relation for the subgoal g_i, and $size(IR_i)$ is the size of the intermediate relation IR_i. The motivation of cost model M_2 is that, as shown in [GMUW00], the time of executing a physical plan is usually determined by the number of disk IO's, which is a function of the sizes of those relations used in the plan.

Cost model M_3 is motivated by the supplementary-relation approach [BR87], whose main idea is to drop attributes during the evaluation of a sequence of subgoals. Under M_3, a physical plan of rewriting P is a list $g_1^{\bar{X}_1}, \ldots, g_n^{\bar{X}_n}$ of the view subgoals in P, with each subgoal g_i annotated with a set $\bar{X}_i$ of *nonrelevant* attributes. All the attributes in $\bar{X}_i$ can be dropped after the first i subgoals are processed, while still being able to compute the answer to the original query after the evaluation terminates. The *generalized supplementary relation* ("GSR" for short) after the first i subgoals are processed, denoted GSR_i, is the intermediate relation IR_i with the attributes in $\bar{X}_i$ dropped.

The cost measure for M_3 is the sum of the sizes of the views joined, plus the sizes of the generalized supplementary relations computed during the multiway join. More formally, for a physical plan $F = g_1^{\bar{X}_1}, \ldots, g_n^{\bar{X}_n}$, its cost under M_3 is:

$$cost_{M_3}(F) = \sum_{i=1}^{n}(size(g_i) + size(GSR_i))$$

where $size(g_i)$ is the size of the relation for the subgoal g_i, and $size(GSR_i)$ is the size of the generalized supplementary relation GSR_i.

Notice that a special case of cost model M_3 is when the nonrelevant attributes in $\bar{X}_i$ are defined as the attributes in the join that are not used in either the query's head, or any subsequent subgoals after subgoal g_i. Then we get the supplementary relation as defined in the literature [BR87, Ull89]. However, as we will see in Section 7.6, by careful variable renaming, it is possible to drop more attributes than the traditional supplementary-relation approach.

Definition 7.2.1 (**efficiency of rewritings**) Under a cost model M, a rewriting P_1 of a query Q is *more efficient* than another rewriting P_2 of Q if the cost of an optimal physical plan of P_1 under cost model M is less than the cost of an optimal physical plan of P_2. A rewriting P is an *optimal rewriting* if it has a physical plan with the lowest cost in all the physical plans of rewritings of Q under M. □

7.3 Cost Model M_1: Number of View Subgoals

In this section we study how to find optimal rewritings under cost model M_1, i.e., rewritings with the minimum number of view subgoals. We first show, given a rewriting, how to minimize its view subgoals. However, this minimization step might miss optimal rewritings if it uses only traditional query-containment techniques. Then we analyze the internal structure of the space of rewritings, and give a space that is guaranteed to include a rewriting with the minimum number of subgoals, if the query has a rewriting.

7.3.1 Minimizing View Subgoals in a Rewriting

Suppose we are given a rewriting P of a query Q using views $\mathcal{V}$, and we want to minimize its number of subgoals while retaining its equivalence to Q. The first step to take is to find the minimal equivalent query of P (not P^{exp}) by removing its redundant subgoals. Let P_m be this minimal equivalent. However, even for the minimal rewriting P_m, we might still be able to remove some of its view subgoals while retaining its equivalence to Q, because we are really interested in rewritings *after* expansion of the views. For instance, P_3 in the car-loc-part example is a minimal rewriting, but we can still remove its subgoal $v_3(S)$ and obtain rewriting P_2 with fewer subgoals. Notice that P_2 and P_3 are not equivalent as queries, although they both compute the same answer to the query. Thus in the second minimization step, we keep removing subgoals from the minimal rewriting P_m, until we get a *locally-minimal rewriting* ("LMR" for short), denoted P_{LMR}. That is, P_{LMR} is a rewriting from which we cannot remove any subgoals and still retain equivalence to the query Q. For instance, the rewritings P_1 and P_2 are two LMRs of the query. The rewriting P_3 is a minimal rewriting, but not an LMR.

For the obtained rewriting P_{LMR}, we cannot remove further subgoals while retaining its equivalence to the query Q. For instance, neither of the first two subgoals in the rewriting P_1 can be removed and still retain its equivalence to the query Q. However, as we will see shortly, we can still reduce the number of view subgoals in an LMR by proper variable renaming. In addition, our goal is to find *globally-minimal rewritings* ("GMR" for short), i.e., rewritings with the minimum number of subgoals. For this goal we analyze the structure of the rewriting space of a query.

7.3.2 Structure of Rewriting Space

Consider the two LMRs P_1 and P_2 in the car-loc-part example. Notice that rewriting P_2 is properly contained in P_1 as queries (i.e., $P_2 \sqsubseteq P_1$), while P_2 has fewer subgoals than P_1. Surprisingly, we can generalize this relationship between containment of two LMRs and their numbers of subgoals as follows.

Lemma 7.3.1 *Let P_1 and P_2 be two LMRs of a query Q. If $P_1 \sqsubseteq P_2$ as queries, then the number of subgoals in P_1 is not greater than the number of subgoals in P_2.* □

Proof: Since $P_1 \sqsubseteq P_2$, there is a containment mapping μ from P_2 to P_1. Suppose that the number of subgoals in P_1 is greater than the number of subgoals in P_2. Then there is at least one subgoal of P_1 is not used in μ. Consider the expansions P_1^{exp} and P_2^{exp} of P_1 and P_2, respectively. The mapping μ implies a mapping from P_2^{exp} to P_1^{exp}. This mapping, together with a containment mapping from Q to P_2^{exp}, implies a mapping from Q to P_1^{exp}. The latter leads to a rewriting that uses only a proper subset of the subgoals in P_1, contradicting the fact that P_1 is an LMR. ∎

We say an LMR is a *containment-minimal rewriting* ("CMR" for short) if there is no other LMR that is properly contained in this rewriting as queries. For instance, the rewriting P_2 in the car-loc-part example is a CMR, while rewriting P_1 is not. However, a GMR might not be a CMR, as shown by the following query, views, and rewritings:

Query:	Q:	$q(X)$	:- $e(X, X)$
Views:	V:	$v(A, B)$	:- $e(A, A), e(A, B)$
Rewritings:	P_1:	$q(X)$	:- $v(X, B)$
	P_2:	$q(X)$	:- $v(X, X)$

The rewriting P_1 is a GMR, but it is not a CMR, since there is another rewriting P_2 (also a GMR) that is properly contained in P_1. We will give a space that is guaranteed to include a GMR of a query, if the query has a rewriting.

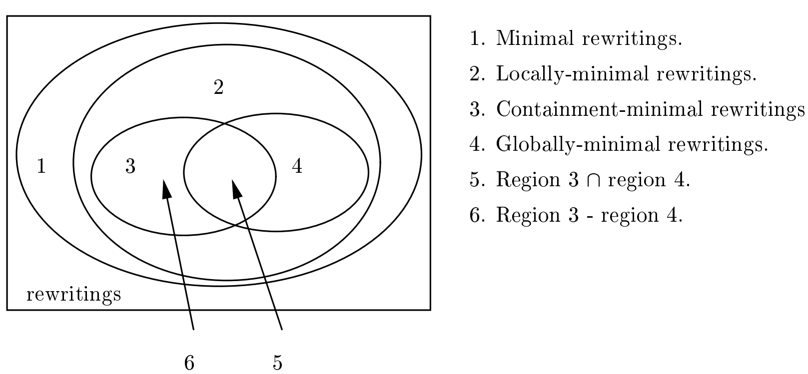


Figure 7.1: Relationship of rewritings of a query.

The relationship of different rewritings of a query Q is shown in Figure 7.1. It can be summarized as follows:

1. A minimal rewriting P does not include any redundant subgoals as a query.

2. A locally-minimal rewriting (LMR) is a minimal rewriting whose subgoals cannot be dropped and still retain equivalence to the query. As we will see shortly, all LMRs form a partial order in terms of their number of subgoals and containment relationship.

3. A containment-minimal rewriting (CMR) P is a locally-minimal rewriting with no other locally-minimal rewritings properly contained in P as queries.

4. A globally-minimal rewriting (GMR) is a rewriting with the minimum number of subgoals. A globally-minimal rewriting is also locally minimal. The subtlety here is that by Lemma 7.3.1, each GMR P has at least one CMR contained in P with the same number of subgoals. Thus, for each GMR in region 6 in Figure 7.1, there exists a GMR in region 5 that has the same number of subgoals. Therefore, we can just limit our search space to all CMRs for finding GMRs.

More formally, the following two propositions are corollaries of Lemma 7.3.1.

Proposition 7.3.1 *Each GMR P has at least one CMR that (1) is contained in P and (2) has the same number of subgoals as P.* □

Proposition 7.3.2 *The set of CMRs contains at least one GMR.* □

EXAMPLE 7.3.1 Consider the following query, view, and three rewritings.

$$\begin{array}{llll}
\text{Query:} & Q: & q(X,Y,Z) & \text{:- } e_1(X,c), e_2(Y,c), e_3(Z,c) \\
\text{View:} & V: & v(X,Y,Z,W) & \text{:- } e_1(X,W), e_2(Y,W), e_3(Z,W) \\
\text{Rewritings:} & P_1: & q(X,Y,Z) & \text{:- } v(X,Y,Z,c) \\
 & P_2: & q(X,Y,Z) & \text{:- } v(X,Y,Z_1,c), v(X_1,Y_1,Z,c) \\
 & P_3: & q(X,Y,Z) & \text{:- } v(X,Y_1,Z_1,c), v(X_2,Y,Z_2,c), v(X_3,Y_3,Z,c)
\end{array}$$

Clearly, LMR P_1 is properly contained in LMR P_2 as queries, which is properly contained in LMR P_3 as queries. Rewriting P_1 is a CMR. Notice we can generalize this example to m base relations $e_1, e_2, \ldots, e_m$ in the query, and get a partial order of *LMRs* that is a chain of length m. □

Since containment mapping is transitive, all the locally-minimal rewritings of a query form a partial order in terms of their containment relationships. The bottom elements in this partial order are the CMRs. In addition, by Lemma 7.3.1, the containment relationship between two LMRs also implies that the contained rewriting has no more subgoals than the containing rewriting. Figure 7.2(a) shows the partial order of the four LMRs (P_1, P_2, P_4, and P_5) in the car-loc-part example. Figure 7.2(b) shows the partial order of the rewritings in Example 7.3.1. Each edge in the figure represents a proper containment relationship: the upper rewriting properly contains the lower rewriting.

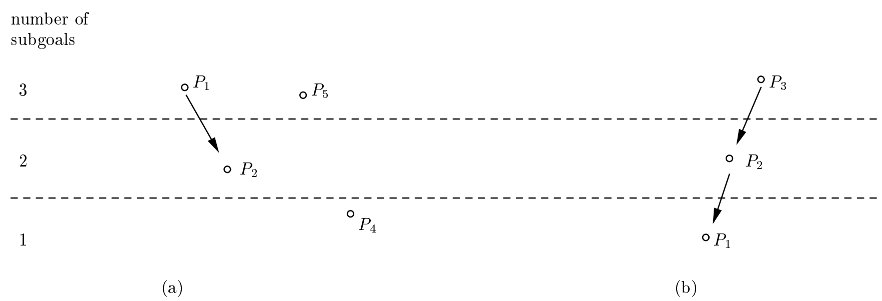


Figure 7.2: Partial order of locally-minimal rewritings of a query.

7.3.3 A Space Including Globally-Minimal Rewritings

The conclusion of the previous subsection is that we can search in the space of CMRs for a GMR, if the query has a rewriting. Now we define a search space in a more constructive way. We need first define several notations. Given a query Q, a *canonical database* D_Q of Q is obtained by turning each subgoal into a fact by replacing each variable in the body by a distinct constant, and treating the resulting subgoals as the only tuples in D_Q. Let $\mathcal{V}(D_Q)$ be the result of applying the view definitions $\mathcal{V}$ on database D_Q. For each tuple in $\mathcal{V}(D_Q)$, we restore each introduced constant back to the original variable of Q, and obtain a *view tuple* of the query given the views. Let $\mathcal{T}(Q, \mathcal{V})$ denote the set of all view tuples *after* the replacement.

In our car-loc-part example, a canonical database for the query Q is:

$$D_Q = \{car(m, anderson), loc(anderson, c), part(s, m, c)\}$$

where the variables M, C, and C are replaced by new distinct constants m, c, and s, respectively. By applying the five view definitions $\mathcal{V}$ on D_Q, we have

$$\mathcal{V}(D_Q) = \{v_1(m, anderson, c), v_2(s, m, c), v_3(s), v_4(m, anderson, c, s), v_5(m, anderson, c)\}$$

Thus the set of view tuples $\mathcal{T}(Q, \mathcal{V})$ is

$$\{v_1(M, anderson, C), v_2(S, M, C), v_3(S), v_4(M, anderson, C, S), v_5(M, anderson, C)\}$$

The following lemma, which is a rephrasing of a result in [LMSS95], helps us restrict the search space for finding globally-minimal rewritings for a query.

Lemma 7.3.2 *For any rewriting P*

$$q(\bar{X}) \text{ :- } p_1(\bar{Y_1}), \ldots, p_k(\bar{Y_k})$$

of a query Q using views $\mathcal{V}$, there is a rewriting P' of Q such that P' is in the form:

$$q(\bar{X}) \text{ :- } p_1(\bar{Y_1}'), \ldots, p_k(\bar{Y_k}')$$

In addition, each $p_i(\bar{Y_i}')$ is a view tuple in $\mathcal{T}(Q, \mathcal{V})$, and $P' \sqsubseteq P$. □

The main idea of the proof in [LMSS95] is to consider a containment mapping μ from P^{exp} to Q, and replace each variable X in P by its target variable $\mu(X)$ in Q. For instance, let us see how to transform P_1 in the car-loc-part example to the LMR P_2 that uses the view tuples only. Consider the containment mapping from P_1^{exp} to Q: $\{M_1 \rightarrow M, M \rightarrow M, anderson \rightarrow anderson, C_1 \rightarrow C, C \rightarrow C, S \rightarrow S\}$. Under this mapping, we transform P_1 to:

$$P_1' : q_1(M, C) \text{ :- } v_1(M, anderson, C), v_1(M, anderson, C), v_2(S, M, C)$$

After removing one duplicate subgoal from P_1', we have the rewriting P_2.

In Section 7.3.2, we showed that the set of CMRs contains a GMR. Below we define a search space for GMRs in a more constructive fashion. The following lemma shows that CMRs are contained in a set of rewritings defined constructively, hence we can regard this set as a search space for optimal rewritings under cost model M_1.

Lemma 7.3.3 *All LMRs of a query using views that use only view tuples of the query include all CMRs of the query.*[2] □

Proof: For any CMR P of a query Q, by Lemma 7.3.2, there is a CMR P' that uses only view tuples, such that $P' \sqsubseteq P$. By the definition of CMR, P cannot have any locally-minimal rewriting that is *properly* contained in P. Thus P must be equivalent to P' as

[2] *We assume two rewritings are the same if the only difference between them is variable renamings.*

queries. In addition, since both P and P' are minimal, they must be isomorphic to each other; i.e., the only difference between them is variable renamings. ∎

An immediate consequence is the following theorem that defines a restricted space for searching globally-minimal rewritings of a query.

Theorem 7.3.1 *By searching in the space of a query's LMRs that use only view tuples in $\mathcal{T}(Q, \mathcal{V})$, we guarantee to find a globally-minimal rewriting, if the query has a rewriting.* □

Theorem 7.3.1 suggests a naive algorithm that finds a globally-minimal rewriting of a query Q using views $\mathcal{V}$ as follows. We compute all the view tuples for the query. We start checking combinations of view tuples. We first check all combinations containing one view tuple, then all combinations containing two view tuples, and so on. Each combination could be a rewriting P. We test whether there is a containment mapping from Q to P^{exp}. (By the construction of the view tuples, there is always a containment mapping from P^{exp} to Q.). If so, then P is a GMR. It is known [LMSS95] that if there is a rewriting for the query, then there is one with at most n subgoals, where n is the number of subgoals in the query. Thus we stop after having considered all combinations of up to n view tuples.

7.4 An Algorithm for Finding Globally-Minimal Rewritings

In this section we develop an efficient algorithm, called *Corecover*, for finding optimal rewritings of a query under the cost model M_1, i.e., globally-minimal rewritings. The algorithm searches in the space of rewritings using view tuples for GMRs of the query. Intuitively, the algorithm considers each view tuple to see what query subgoals can be covered by this view tuple. The set of query subgoals covered by the view tuple is called *tuple-core*. The algorithm then uses the *minimum* number of view tuples to cover all query subgoals, and each cover yields a GMR of the query.

7.4.1 Tuple-Core: Query Subgoals Covered by a View Tuple

The algorithm Corecover first finds the set of query subgoals that can be "covered" by a view tuple, called *tuple-core*. Before giving the definition of tuple-core, we show a nice property of rewritings using view tuples for a minimal query. Note that for the rewritings we consider in this section, we may think as follows: All the variables of rewriting P (recall that P is generated out of view tuples) are also variables of Q, i.e., $Var(P) \subseteq Var(Q)$.

Lemma 7.4.1 *If Q is a minimal CQ, and ϕ is a containment mapping from Q to Q, then:*

1. *ϕ is one-to-one, i.e., different arguments are mapped to different arguments;*
2. *ϕ is onto, i.e., every subgoal in Q is an image of ϕ;*
3. *ϕ maps a variable to a variable.*

Therefore, ϕ can be reversed to yield another containment mapping from Q to Q. □

Proof: (1) If ϕ is not onto, the unused subgoals in $Q - \phi(Q)$ are redundant, contradicting the fact that Q is minimal. (2) If ϕ is not one-to-one, then there is a variable not used in $\phi(Q)$. Thus one subgoal in $\phi(Q)$ is not mapped to. Then $\phi(Q)$ has fewer subgoals than Q, but is still equivalent to Q, contradicting the fact that Q is minimal. (3) By definition, ϕ maps constants to constants, and maps variables to either variables to constants. If ϕ maps a variable to a constant, then there must be some left-over variables that are not in the image $\phi(Q)$, contradicting the fact that ϕ is onto. ∎

Lemma 7.4.2 *For a minimal query Q and a set of views $\mathcal{V}$, let P be a rewriting of Q that uses only view tuples in $\mathcal{T}(Q, \mathcal{V})$. There is a containment mapping μ from Q to P^{exp}, such that (1) μ is a one-to-one mapping, i.e., different arguments in Q are mapped to different arguments in P^{exp}; (2) For all arguments in Q that appear in P, they are mapped by μ as is the identity mapping on arguments, i.e., $\mu(X) = X$ for all $X \in Var(P)$.*[3] □

Proof: Consider a minimal equivalent query P_m^{exp} of P^{exp}. Notice that both Q and P_m^{exp} are minimal equivalents of the expansion P^{exp}. Thus their only difference is variable renamings. By the construction of the view tuples, there is a containment mapping from P^{exp} to Q, such that it maps each argument in P^{exp} that appears in P to itself. Let ν be the corresponding containment mapping from from P_m^{exp} to Q.

Since Q and P_m^{exp} are equivalent, there is a containment mapping τ from Q to P_m^{exp}. This mapping, together with ν, implies a containment mapping from Q to Q. Since Q is minimal, by Lemma 7.4.1, the composed containment mapping $\tau\nu$ should be one-to-one and onto. Thus ν should also be one-to-one and onto. Then we can reverse the mapping ν, and obtain a containment mapping $\mu = \nu^{-1}$ from Q to P_m^{exp}, such that μ is one-to-one, and maps the arguments in Q that appear in P under identity. ∎

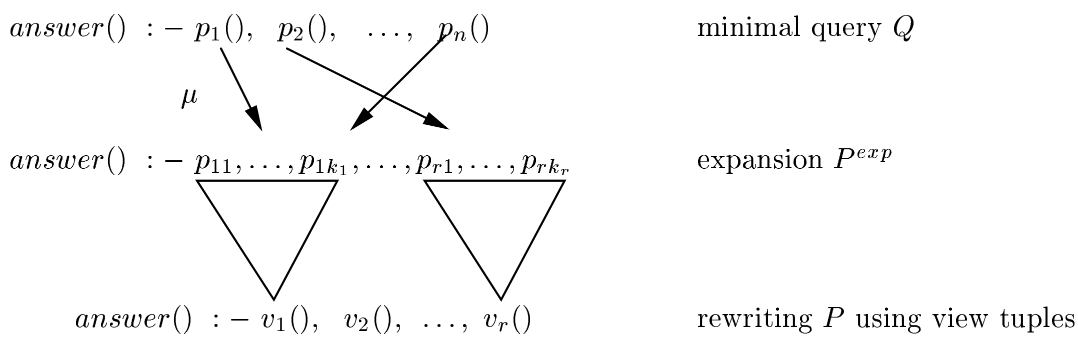


Figure 7.3: Proof of Lemma 7.4.2.

For instance, the rewriting P_2 in the car-loc-part example uses view tuples only. We have a containment mapping from the query Q to P_2^{exp}:

$$M \to M, anderson \to anderson, C \to C, S \to S$$

This containment mapping maps the arguments $\{M, anderson, C, S\}$ in Q that appear in P_2 to themselves.

In general, there can be different containment mappings from a minimal query to the expansion of a rewriting using view tuples. By Lemma 7.4.2, it turns out that we can just focus on a containment mapping that has the two properties in the lemma, and decide what query subgoals are covered by the *expansion* of each view tuple under this containment mapping. The expansion of a view tuple t_v, denoted t_v^{exp}, is obtained by replacing t_v by the base relations in this view definition. Existentially quantified variables in the definition are replaced by fresh variables in t_v^{exp}. Clearly this expansion t_v^{exp} will appear in the expansion of any rewriting using t_v.

Definition 7.4.1 (tuple-core) Let t_v be a view tuple of view v for a minimal query Q. A *tuple-core* of t_v is a *maximal* collection G of subgoals in the query Q, such that there is a containment mapping μ from G to the expansion t_v^{exp} of t_v, and μ has the following properties:

1. μ is a one-to-one mapping, and it maps the arguments in G that appear in t_v as is the identity mapping on arguments.

---

[3] *The definition of rewriting P guarantees a containment mapping from Q to P^{exp}, but this containment mapping might not have the two properties.*

2. Each distinguished variable X in G is mapped to a distinguished variable in t_v^{exp} (moreover, by Property (1), $\mu(X) = X$).

3. If a nondistinguished variable X in G is mapped under μ to an existential variable in t_v's expansion, then G includes all subgoals in Q that use this variable X.

□

The purpose of these properties is to make sure when we construct a rewriting using view tuples whose tuple-cores cover all query subgoals, the containment mappings of these core-tuples can be combined seamlessly to form a containment mapping from the query to the rewriting's expansion. In particular, Property (1) is based on Lemma 7.4.2. Properties (2) and (3), which are satisfied by any containment mapping from the query to a rewriting expansion, are also used in the MiniCon algorithm. A view tuple can have an empty tuple-core. As expected, we have:

Lemma 7.4.3 *A view tuple for a minimal query has a unique tuple-core.* □

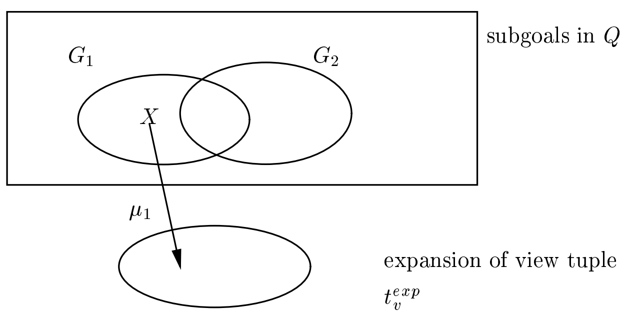


Figure 7.4: Uniqueness of tuple-core of a view tuple.

Proof: (Convention: We use the same names in Q and in t_v^{exp} for the distinguished variables of t_v^{exp} that are targets under μ_1 or μ_2.) Suppose a view tuple t_v for a minimal query Q has two distinct tuple-cores G_1 and G_2, with the corresponding mappings μ_1 and μ_2 in Definition 7.4.1. Let $H_1 = \mu_1(G_1)$ and $H_2 = \mu_2(G_2)$ be the targets (sets of subgoals in t_v^{exp}) respectively. Either $G_1 - G_2$ or $G_2 - G_1$ is not empty (otherwise G_1, G_2 are identical). Suppose $G_1 - G_2$ is not empty. As suggested by Figure 7.4, for each variable X used in $G_1 - G_2$, one of two cases applies:

1. $\mu_1(X) = X$. We will show that either X is not used in G_2, or $\mu_2(X) = X$.

2. $\mu_1(X) \neq X$. We will show that X cannot be used in G_2.

3. The mapping μ_2' is one-to-one.

The first two points show that the mappings μ_1 and μ_2 do not conflict with each other on their source variables in Q. Thus we can define a mapping μ_2' from $G_1 \cup G_2 = G_2 \cup (G_1 - G_2)$ onto $H_2' = \mu_1(G_1 - G_2) \cup \mu_2(G_2)$ as follows: if X is used in G_2, $\mu_2'(X) = \mu_2(X)$; if X is used in $G_1 - G_2$, $\mu_2'(X) = \mu_1(X)$.

Therefore, we have a larger set $G_1 \cup G_2$ of query subgoals that satisfies the conditions in Definition 7.4.1, contradicting the fact that G_2 is maximal.

We first prove case (1). Suppose X appears in G_2, and $\mu_2(X) \neq X$. Then $X = \mu_1(X)$ is a nondistinguished variable in t_v^{exp}, and $\mu_2(X)$ is a nondistinguished variable in the query. By G_2's definition, G_2 includes all query subgoals that use X, contradicting the fact that X appears in $G_1 - G_2$. Now we prove case (2). Suppose X is in G_2. By G_1's definition, X cannot be a variable in t_v, since $\mu_1(X) \neq X$. Then μ_2 can only map X to a nondistinguished variable in t_v's expansion. By G_2's definition, G_2 should include all the query subgoals that use X, contradicting the fact that X appears in $G_1 - G_2$.

In the rest of the proof, we prove claim (3). Since t_v is a view tuple, there is a mapping λ from t_v^{exp} to Q. Suppose mapping μ_2' is not one-to-one. Then, mapping $\mu_2'\lambda$ is a mapping from $G_1 \cup G_2$ to Q that is not one-to-one either. We show that we can extend $\mu_2'\lambda$ to a mapping from Q to Q that is *not* one-to-one, contradicting the fact that Q is minimal. For this extension to be feasible, we need to show: (i) No distinguished variable of Q is mapped to another distinguished variable of Q under $\mu_2'\lambda$ and (ii) If $G_1 \cup G_2$ shares a variable X with a subgoal not in $G_1 \cup G_2$, then $\mu_2'(X) = X$ If (i) and (ii) hold, then we easily extend $\mu_2'\lambda$ by having the variables not in $G_1 \cup G_2$ mapped each on itself.

We prove (i): If mapping μ_2' is not one-to-one, then, there exist variables X used in $G_1 - G_2$ and not used in G_2 and, X' used in G_2 such that $\mu_1(X) = \mu_2(X')$. Then either X or X' is a nondistinguished variable of Q.

We prove (ii): If $G_1 \cup G_2$ shares a variable X with a subgoal not in $G_1 \cup G_2$, then X is a variable in the view tuple t_v. Hence $\mu_2'(X) = X$. ∎

The unique tuple-core of a view tuple t_v is denoted by $\mathcal{C}(v_t)$.

EXAMPLE 7.4.1 Consider the following query and views:

$$\begin{array}{rll} \text{Query } Q\text{:} & q(X,Y) & \text{:- } a(X,Z), a(Z,Z), b(Z,Y) \\ \text{Views } V_1\text{:} & v_1(A,B) & \text{:- } a(A,B), a(B,B) \\ V_2\text{:} & v_2(C,D) & \text{:- } a(C,E), b(C,D) \end{array}$$

A canonical database of the query is $D_Q = \{a(x,z), a(z,z), b(z,y)\}$. By applying the view definitions on D_Q, we have $\mathcal{V}(D_Q) = \{v_1(x,z), v_1(z,z), v_2(z,y)\}$. Thus the set of view tuples is $\mathcal{T}(Q,\mathcal{V}) = \{v_1(X,Z), v_1(Z,Z), v_2(Z,Y)\}$. The table shows the tuple-cores for the three view tuples.

view tuple t_v	expansion t_v^{exp}	tuple-core $\mathcal{C}(t_v)$	mapping μ from $\mathcal{C}(t_v)$ to t_v^{exp}
$v_1(X,Z)$	$a(X,Z), a(Z,Z)$	$a(X,Z), a(Z,Z)$	$X \to X, Z \to Z$
$v_1(Z,Z)$	$a(Z,Z), a(Z,Z)$	$a(Z,Z)$	$Z \to Z$
$v_2(Z,Y)$	$a(Z,E), b(Z,Y)$	$b(Z,Y)$	$Z \to Z, Y \to Y$

Table 7.2: Tuple-cores for the three view tuples in example 7.4.1.

By using the three tuple-cores, the only minimum cover of the query subgoals is the union of the tuple-cores of $v_1(X,Z)$ and $v_2(Z,Y)$, which yields the following GMR of the query:

$$q(X,Y) \text{ :- } v_1(X,Z), v_2(Z,Y)$$

□

For another example, let us derive the tuple-cores of the five view tuples in the car-loc-part example. The tuple-cores for $v_1(M, anderson, C)$, $v_2(S,M,C)$, $v_4(M, anderson, C, S)$, and $v_5(M, anderson, C)$ are identical to the body of the corresponding rules, with variable D replaced by constant *anderson*. View tuple $v_3(S)$, though, has an empty tuple-core, since the only possible mapping from a collection of subgoals of Q to $v_3(S)^{exp}$ that satisfies property (3) of Definition 7.4.1, is: $M \to M_3, a \to a, C \to C_3, S \to S$. (To avoid confusion, in the definition of v_3, we replace variable M by variable M_3, and variable C by variable C_3.) However, this mapping does not satisfy property (2), since it maps a distinguished variable C in Q to a nondistinguished variable C_3 in $v_3(S)^{exp}$.

7.4.2 Using Tuple-Cores to Cover Query Subgoals

The second step of Corecover finds a minimum number of view tuples to cover query subgoals. This problem can be modeled as a classic *set-covering problem* [CLR90]. Notice by the construction of the tuple-cores, a containment-mapping check is not needed in this step. This step is based on the following theorem:

Theorem 7.4.1 *For a minimal query Q and a set of views $\mathcal{V}$, let P be a query that has the head of Q and uses only view tuples in $\mathcal{T}(Q,\mathcal{V})$ in its body. P is a rewriting of Q if and only if the union of the tuple-cores of its view tuples includes all the query subgoals in Q.* □

Proof: Let $t_1, \ldots, t_k$ be the view tuples used in P. For the "If" part, consider each t_i. Let $\mathcal{C}(t_i)$ be its tuple-cores, and μ_i be the containment mapping from $\mathcal{C}(t_i)$ to t_i^{exp} that defines the tuple-core. Assume $\mathcal{C}(t_1) \cup \ldots \cup \mathcal{C}(t_k)$ includes all query subgoals in Q. By the definitions of these tuple-cores, these containment mappings can be combined to form a containment mapping μ from $\mathcal{C}(t_1) \cup \ldots \cup \mathcal{C}(t_k)$, which includes all query subgoals in Q, to the body of P^{exp}. The mapping μ also maps the head variables in Q under identity. By the construction of these view tuples, there is also a mapping from P^{exp} to Q. Thus P^{exp} is equivalent to Q as queries, and P is a rewriting of Q using $\mathcal{V}$.

"Only If": Assume P is a rewriting of Q using $\mathcal{V}$. By Lemma 7.4.2, there is a one-to-one containment mapping μ from Q to P^{exp}, which maps all arguments in Q that appear in P under identity. Notice that the body of P^{exp} is the union of $t_1^{exp}, \ldots, t_k^{exp}$, thus μ partitions the query subgoals into k groups $G_1, \ldots, G_k$, such that each G_i is mapped by μ to t_i^{exp}. The subgoal set G_i and the "local" mapping μ satisfies the three properties in Definition 7.4.1, except that G_i might not be maximal. According to Lemma 7.4.3, the tuple core is unique, hence $G_i \subseteq \mathcal{C}(t_i)$. Thus the union of the k tuple-cores includes all query subgoals in Q. ■

Corollary 7.4.1 *For a minimal query Q and a set of views $\mathcal{V}$, each GMR of Q using view tuples in $\mathcal{T}(Q,\mathcal{V})$ corresponds to a minimum cover of the query subgoals using the tuple-cores of the view tuples.* □

For instance, consider the tuple cores of the view tuples in car-loc-part example. The minimum cover of the query subgoals uses the tuple core of view tuple $v_4(M, anderson, C, S)$, which yields the GMR P_4 of the query. Figure 7.5 summarizes the Corecover algorithm.

The complexity of the algorithm Corecover is exponential, since the problem of finding whether there exists a rewriting is $\mathcal{NP}$-hard [LMSS95]. The running time of the algorithm, though, depends mostly on the number of view tuples produced in the first step. Since this number tends to be small in practice, the algorithm performs efficiently in the later steps, as shown by our experimental results in Section 7.7.

Algorithm CoreCover: Find rewritings with minimum number of subgoals.
Input: • Q: A conjunctive query.
• $\mathcal{V}$: A set of conjunctive view.
Output: A set of rewritings using view tuples with minimum number of subgoals.
Method:
(1) Minimize Q by removing its redundant subgoals. Let Q_m be the minimal equivalent.
(2) Construct a canonical database D_{Q_m} for Q_m. Compute the view tuples $\mathcal{T}(Q_m, \mathcal{V})$ by applying the view definitions $\mathcal{V}_m$ on the database.
(3) For each view tuple $t \in \mathcal{T}(Q_m, \mathcal{V})$, compute its tuple-core $\mathcal{C}(t)$.
(4) Use the nonempty tuple-cores to cover the query subgoals in Q_m with minimum number of tuple-cores. For each cover, construct a rewriting by combining the corresponding view tuples.

Figure 7.5: The algorithm Corecover.

7.4.3 Comparison with the MiniCon Algorithm

Corecover and MiniCon [PL00] share the same observation of the Properties (2) and (3) in Definition 7.4.1, which should be satisfied by any mapping from query subgoals to a view subgoal that can be used in a rewriting. Since we want to find equivalent rewritings, rather than contained rewritings, the different goal gives us the chance to develop a more efficient algorithm. In particular, given the fact that there is a containment mapping from the expansion of an equivalent rewriting to the query, Corecover limits the search space for useful view literals by applying the view definitions on the canonical database of the query. In other words, this containment mapping helps Corecover not to consider *all* possible "head homomorphisms" (defined in [PL00]) on the views, which could be a huge set.

Another advantage that the new goal gives Corecover is that, each tuple-core of a view tuple includes the *maximal* subset of query subgoals that satisfy the three properties in Definition 7.4.1. Correspondingly, the "MCD" concept used in MiniCon includes a *minimal* subset of query subgoals. The reason MCD finds a minimal subset of query subgoals is that it tries to find maximally-contained rewritings, and each MCD should be as relaxing as possible, so that all MCDs can be combined. In our case, since we are finding equivalent rewritings, we are more aggressive to cover as many query subgoals as possible using a view tuple. As a consequence, in the last step of Corecover, the tuple-cores of a set of view tuples that form a rewriting can overlap. That is, a query subgoal can be covered by two tuple-cores. In the second step of MiniCon, the MCDs that form a contained rewriting do not overlap.

Since MiniCon does not aim at generating efficient rewritings, it may produce some rewritings with redundant subgoals, as shown by the following example.

EXAMPLE 7.4.2 Consider the following query and views:

$$
\begin{array}{llll}
\text{Query:} & Q\text{:} & q(X,Y) & \text{:- } a_1(X,Z_1), b_1(Z_1,Y), \\
 & & & \quad \vdots \\
 & & & \quad a_k(X,Z_k), b_k(Z_k,Y). \\
\text{Views:} & V\text{:} & v(X,Y) & \text{:- same as the body above} \\
 & V_1\text{:} & v_1(X,Y) & \text{:- } a_1(X,Z_1), b_1(Z_1,Y) \\
 & & & \quad \vdots \\
 & V_{k-1}\text{:} & v_{k-1}(X,Y) & \text{:- } a_{k-1}(X,Z_{k-1}), b_{k-1}(Z_{k-1},Y)
\end{array}
$$

For view V, algorithm Corecover computes only one view tuple $V(X,Y)$, whose tuple-core includes *all* the $2k$ subgoals in Q. In addition, Corecover also computes a view tuple $v_i(X,Y)$ for each of the rest $k-1$ views. Thus Corecover creates only one rewriting P with the minimum number of subgoals:

$$P: \; q(X,Y) \text{ :- } v(X,Y)$$

Correspondingly, for view V, MiniCon generates k different MCDs, each MCD covering two query subgoals $a_i(X,Z_i), b_i(Z_i,Y)$. In addition, MiniCon also produces an MCD for each of the rest $k-1$ views. Thus it produces rewritings with redundant subgoals.

[PL00] describes a minimization step that removes some subgoals in the generated rewritings. Different from our algorithm, this minimization step cannot guarantee to generate a rewriting with the minimum number of subgoals. One observation is that MiniCon may produce many inefficient rewritings with redundant subgoals. In particular, an implementation of the algorithm could have two choices that might not be efficient: (1) generate all rewritings and then look for P (we do not want this approach, since the space is too big); or (2) generate a single rewriting arbitrarily (then it is not good either, because we cannot produce P starting from an arbitrary rewriting, even if we use the minimization step). □

7.5 Cost Model M_2: Counting Sizes of Relations

In this section we study cost model M_2 that considers sizes of view relations and intermediate relations in a physical plan. We show that the space of all minimal rewritings that use view tuples is guaranteed to include an optimal rewriting of a query under M_2, if the query has a rewriting.

7.5.1 A Search Space for Optimal Rewritings under M_2

The following lemma helps us find a search space for optimal rewritings under M_2.

Lemma 7.5.1 *Under cost model M_2, for any rewriting P of a query Q using views $\mathcal{V}$, there is a minimal rewriting P' that uses only view tuples in $\mathcal{T}(Q, \mathcal{V})$, such that P' is at least as efficient as P.* □

Proof: Let F be an optimal physical plan of the rewriting P. By the proof of Lemma 7.3.2, there is a minimal rewriting P' that only uses view tuples in $\mathcal{T}(Q, \mathcal{V})$, and $P' \sqsubseteq P$. In addition, all the subgoals in P' are images of μ. Let μ be a containment mapping from P to P'. Now we construct a physical plan F' of P', such that $cost_{M_2}(F') \leq cost_{M_2}(F)$. Suppose $F = [g_1, \ldots, g_n]$. Let IR_i denote the intermediate relation after the first i subgoals in F, i.e., $IR_i = g_1 \bowtie \cdots \bowtie g_i$. We construct the physical plan F' of P' that processes the subgoals of P' in the sequence of $\mu(g_1), \ldots, \mu(g_n)$. If a subgoal $\mu(g_k)$ in the sequence has been processed earlier, we only keep its first occurrence in F'. Let $IR'_i = \mu(g_1) \bowtie \cdots \bowtie \mu(g_i)$ be the corresponding intermediate relation in plan F', with the duplicated subgoals dropped.

Because of the mapping μ from P to P', we have $IR'_i \subseteq IR_i$, thus $size(IR'_i) \leq size(IR_i)$. Also since all the subgoals P' are images of μ, plan F' includes all view subgoals in P'. In addition, all the view relations used in F are also used in F'. Thus F' is a physical plan of P', and $cost_{M_2}(F') \leq cost_{M_2}(F)$. ■

Under cost model M_2, plan P_2 in the car-loc-part example is at least as efficient as plan P_1, since there is a containment mapping from P_1 to P_2, such that all the subgoals of P'_2 are images under the mapping. By Lemma 7.5.1, we have the following theorem.

Theorem 7.5.1 *For a query Q and a set of views $\mathcal{V}$, the space of minimal writings using view tuples in $\mathcal{T}(Q, \mathcal{V})$ is guaranteed to include an optimal rewriting under cost model M_2, if the query has a rewriting.* □

By Theorem 7.4.1 in Section 7.4, we can modify the algorithm Corecover to get another algorithm Corecover* that finds all minimal rewritings using view tuples for a query. The only difference between these two algorithms is that in the last step, Corecover finds all *minimum* sets of view-tuples whose tuple-cores cover query subgoals, while Corecover* considers *all* sets of view-tuples to cover the query subgoals. The view tuples with an empty tuple-core are also used by Corecover*. By Theorem 7.5.1, these minimal rewritings guarantee to include an optimal rewriting under cost model M_2, if the query has a rewriting.

As shown by the minimal rewriting P_3 in the car-loc-part example, subgoal $v_3(S)$ can be used to improve the efficiency of the plan, although it does not cover any query subgoal. In general, some view subgoals in a minimal rewriting may be removed without changing the equivalence to the original query, but these view subgoals can serve as filtering subgoals to reduce the sizes of intermediate relations. The optimizer can do a cost-based analysis, and decide whether adding some filtering subgoals to a rewriting can make the rewriting more efficient.

7.5.2 Concise Representation of Minimal Rewritings

In the case where there are many views that can be used to answer a query, the number of view tuples could be large. For instance, consider the case where we have n views that are exactly the same as the query. Then there can be n view tuples, and each has a tuple-core that includes all the query subgoals. Then there can be $2^n - 1$ minimal rewritings of the query.

We propose the following solution to the problem. First, we partition all views into equivalence classes, such that all the views in each class are equivalent as queries. When we run the Corecover algorithm, we only select a view from each class as a representative. Second, after the view tuples are computed, we also partition these view tuples into equivalence classes, such that all the view tuples in each class have the same tuple-core, i.e., they cover the same set of query subgoals.

Our solution has several advantages. (1) There is a small number of groups of rewritings, with each group having specific properties that might facilitate a more efficient algorithm for the optimizer. (2) The number of view tuples that need to be considered by Corecover to cover the query subgoals is bounded by the number of query subgoals, thus it becomes *independent from the number of views*. (3) The optimizer can find efficient physical plans by considering the "representative rewritings," and then decide whether each rewriting can become more efficient by adding view tuples as filtering subgoals. The optimizer uses the information about the sizes of relations and selectivity of joins to make this decision. (4) The optimizer can replace a view tuple in a rewriting with another view tuple in the same equivalence view-tuple class, and yet get a new rewriting to the query. Our experiments in Section 7.7 will show that this solution helps the Corecover algorithm achieve good performance.

7.5.3 Generalization of Cost Model M_2

The key reason that cost model M_2 allows us to restrict the search space in minimal rewritings using view tuples is that M_2 has what we called the property of *containment monotonicity*. That is, a cost model M is *containment monotonic* if for any two rewritings P_1 and P_2, if the following two conditions

1. There is a containment mapping from P_1 to P_2;

2. All subgoals in P_2 are images under the mapping;

can imply $cost_M(P_2) \leq cost_M(P_1)$. Theorem 7.5.1 can be generalized to any cost model that is containment monotonic.

7.6 Cost Model M_3: Dropping Nonrelevant Attributes

Cost model M_3 improves M_2 by considering the fact that after computing an intermediate relation in a physical plan, some attributes can be dropped. In this section, we first give an example to show that if the optimizer uses the traditional supplementary-relation approach to decide what attributes to drop, the rewritings using view tuples might not yield an optimal physical plan under M_3. Then we propose a heuristic that can be taken by the optimizer to drop more attributes without changing the final answer of the evaluation, thus producing a more efficient physical plan.

7.6.1 Dropping Attributes Using Supplementary-Relation Approach

Recall that in cost model M_3, a physical plan F of a rewriting P is a list $g_1^{\bar{X}_1}, \ldots, g_n^{\bar{X}_n}$ of the subgoals in P, with each subgoal g_i annotated with a set of attributes $\bar{X}_i$ that can be dropped after subgoal g_i is processed in the sequence. Given a rewriting P, the optimizer considers all possible orderings of the subgoals, and decides the dropping strategy for each ordering. By taking the supplementary-relation approach, for an order of subgoals $g_1, \ldots, g_n$, after subgoal g_i is processed, the optimizer drops the nonrelevant arguments that are not used in subsequent subgoals or in the head of P. The corresponding supplementary relation SR_i is the $SR_{i-1} \bowtie g_i$ with the nonrelevant arguments dropped.

The following example shows that by taking this approach, the optimizer might miss an optimal physical plan under cost model M_3, if the rewriting generator passes to it only rewritings using view tuples.

EXAMPLE 7.6.1 Consider the following query, views, and rewritings:

Query:	Q:	$q(A)$	:- $r(A,A), t(A,B), s(B,B)$
Views:	V_1:	$v_1(A,B)$	:- $r(A,A), s(B,B)$
	V_2:	$v_2(A,B)$	:- $t(A,B), s(B,B)$
Rewritings:	P_1:	$q(A)$	:- $v_1(A,B), v_2(A,C)$
	P_2:	$q(A)$	:- $v_1(A,B), v_2(A,B)$

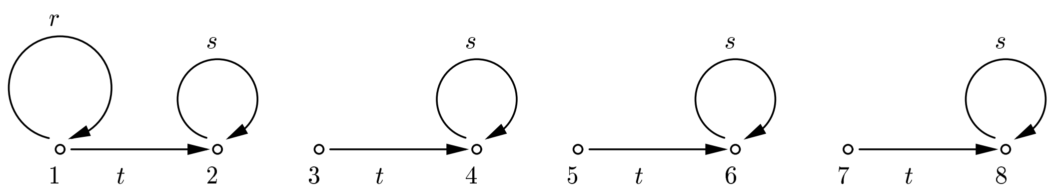


Figure 7.6: Base relations.

Rewriting P_2 is the only minimal rewriting of Q using the two view tuples $v_1(A,B)$ and $v_2(A,B)$, while rewriting P_1 uses a fresh variable C in its second subgoal. Consider the database shown in Figure 7.6. The three base relations (r, s, and t) and two view relations (v_1 and v_2) are:

r	s	t	v_1	v_2
$\langle 1,1\rangle$	$\langle 2,2\rangle$	$\langle 1,2\rangle$	$\langle 1,2\rangle$	$\langle 1,2\rangle$
	$\langle 4,4\rangle$	$\langle 3,4\rangle$	$\langle 1,4\rangle$	$\langle 3,4\rangle$
	$\langle 6,6\rangle$	$\langle 5,6\rangle$	$\langle 1,6\rangle$	$\langle 5,6\rangle$
	$\langle 8,8\rangle$	$\langle 7,8\rangle$	$\langle 1,8\rangle$	$\langle 7,8\rangle$

By taking the supplementary-relation approach, the physical plans of P_1 are more efficient than those of P_2. To see why, consider an order $O_2 = [v_1(A,B), v_2(A,B)]$ of subgoals in P_2, and a corresponding order $O_1 = [v_1(A,B), v_2(A,C)]$ of P_1. Order O_2 yields a physical plan $F_2 = [v_1(A,B)^{\{\}}, v_2(A,B)^{\{B\}}]$. In particular, its first supplementary relation needs to keep attributes A and B, since both will be used later. This supplementary relation includes all the four tuples in v_1. Order O_1 yields a physical plan $F_1 = [v_1(A,B)^{\{B\}}, v_2(A,C)^{\{C\}}]$, and its first supplementary relation does not keep attribute B, since B is not used by the second subgoal or the head. This supplementary relation has only one tuple $\langle 1 \rangle$. The rest costs of F_1 and F_2 are the same. Thus, $cost_{M_3}(F_1) < cost_{M_3}(F_2)$. If we reverse the two subgoals in the two orderings, the new physical plan of P_1 is still more efficient than that of P_2. $\square$

A minimal rewriting using view tuples may fail to generate an optimal physical plan under M_3, because the variables in the rewriting are made as restrictive as possible by only using the variables in the query. Then view literals in a rewriting might be removed while obtaining the equivalence to the query. However, if the optimizer takes the supplementary-relation approach to decide what attributes to drop, these restrictive variables might not be dropped, since some may be used later in a sequence of subgoals.

The reason that P_1 is more efficient than P_2 is that a physical plan of P_1 has the freedom to drop the second argument after processing its first subgoal. However, P_2 needs to keep the argument, since this argument will be used later in the second subgoal to do a comparison. Now we show that if the optimizer can be "smarter" by using the information about the query and views, it can do better than the supplementary-relation approach.

7.6.2 A Heuristic to Drop Attributes

We give a heuristic that helps the optimizer drop more attributes than the supplementary-relation approach. Intuitively, given a rewriting P of a query Q, the optimizer considers all orderings of the subgoals in P. For each ordering $O = g_1, \ldots, g_n$, it considers what attributes can be dropped after subgoal g_i is processed without changing the final result of the computation.

For a variable Y that appears in the intermediate relation IR_i, let us consider in what case we can drop Y without changing the result of the computation. As in the supplementary-relation approach, if Y does not appear in subsequent subgoals or the head, it can be dropped. However, even if Y appears in a subsequent subgoal, it might still be dropped, as shown by the variable B in rewriting P_2 in Example 7.6.1. Notice:

> Dropping Y will *not* change the result of the computation if and only if, should we rename Y in $g_1, \ldots, g_i$ with a fresh variable, the corresponding new query P' is still an equivalent rewriting of Q.

Therefore, for each variable Y that appears in $g_1, \ldots, g_i$, the optimizer adds Y to the annotation X_i (i.e., the set of attributes that can dropped) if one of the following conditions is satisfied:

1. If Y does not appear in subsequent subgoals or the head of P (as in the supplementary-relation approach);

2. If Y appears in a subsequent subgoal, but after replacing the Y instances in $g_1, \ldots, g_i$ with a fresh variable Y', the new query P' using views is still an equivalent rewriting of the original query Q. (This equivalence is done by testing the equivalence between P'^{exp} and Q.)

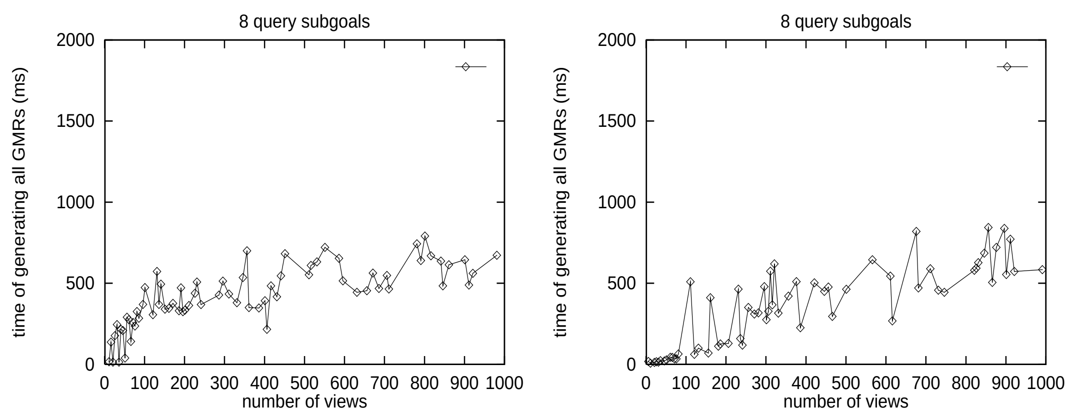
In the second case, by dropping a variable Y that appears in a subsequent subgoal $g_k(\ldots, Y, \ldots)$, we might remove an equality comparison between GSR_{k-1} and $g_k(\ldots, Y, \ldots)$, which could increase the size of GSR_k. Thus the optimizer needs to make the tradeoff between dropping Y and removing this comparison by using the information about the sizes of view relations and generalized supplementary relations.

7.7 Experimental Results

We did experiments to study the search spaces for optimal rewritings under cost models M_1 and M_2, and evaluate the performance of the Corecover algorithm. We studied different shapes of queries, such as chain queries, star queries, and randomly generated queries [SMK97]. We implemented a query generator that takes as input parameters such as: (1) number of base relations; (2) number of attributes in a base relation; (3) number of views; (4) number of subgoals in a view; (5) number of subgoals in a query; (6) shape of queries and views. In the experiments, queries and views were set to have the same parameters, except that they might have different numbers of subgoals. For the same number of views, we ran 40 queries and computed their average measures. The algorithm Corecover was implemented in Java. The experiments were run on a dual-processor Sun Ultra 2 workstation, running SunOS 5.6 with 256 MB memory.

7.7.1 Star Queries

We first considered star queries. Each query had 8 subgoals, and each view randomly had 1, 2, or 3 subgoals. We ignored queries that did not have rewritings. Figure 7.7 (a) shows the running time for Corecover to get all globally-minimal rewritings (GMRs) as the number of views increased, if all variables were distinguished. As the number of views increased, the time of finding all GMRs did not increase steadily. Instead, the time was bound in the range from 0ms to 1 second. On the average, it took Corecover about 500ms to generate all GMRs for a query. Even if there were 1000 views, the time was still less than 1 second. Figure 7.7 (b) shows the running time for Corecover to generate GMRs if one variable was distinguished.



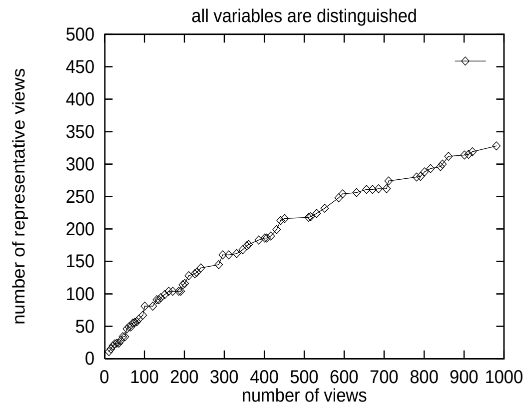
(a) All variables are distinguished.

(b) 1 variable is nondistinguished.

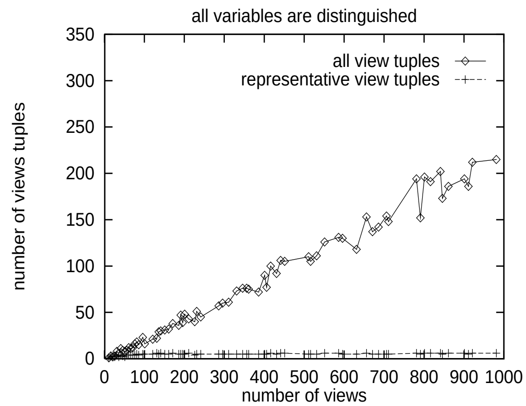
Figure 7.7: Time for Corecover to generate all GMRs for star queries.

The reason Corecover has good efficiency and scalability is that we can group views and view tuples into equivalence classes respectively. From each equivalence class of views, we selected only one representative that was equivalent as queries to other views in the class. Similarly, from each equivalence class of view tuples, we also selected one representative that had the same tuple-core as others. Therefore, the number of representative view tuples depends on the number of query subgoals only, and it is independent from the number of views. Notice that the running time includes the time of grouping views and view tuples into equivalence classes. Although in the early stage of Corecover, we paid extra cost to test view equivalence by testing query containments, this extra cost paid off later when the number of views was more than 100.

For instance, consider the case where all variables were distinguished. Figure 7.8 (a)



(a) Views.



(b) View tuples.

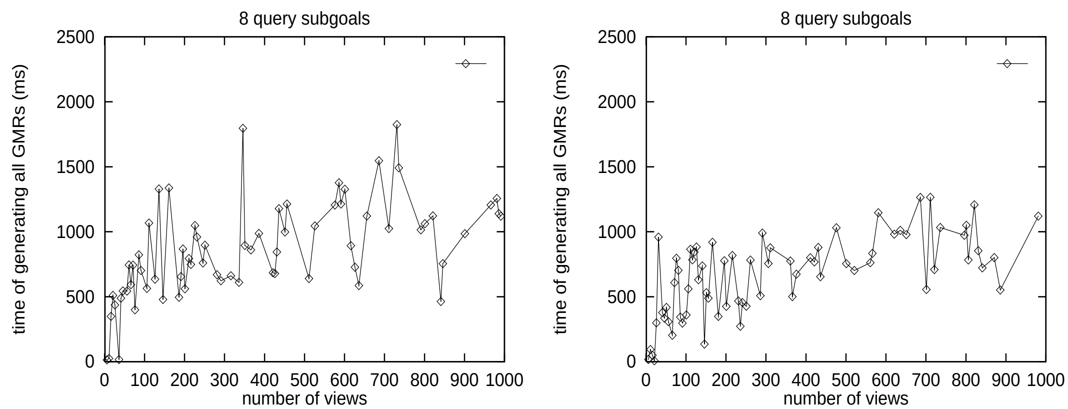
Figure 7.8: Number of equivalence classes for star queries.

shows that as the number of views increased, the number of view equivalence classes also increased, but with a decreasing slope. When there were 1000 views, there were only about 350 equivalent view classes. Figure 7.8 (b) shows that the number of equivalence classes of view tuples was almost a constant (less than 10) as the number of views increased, while the number of view tuples increased to more than 200.

7.7.2 Chain Queries

We then considered chain queries, and had the similar results. Each query had 8 subgoals, and each view had 1, 2, or 3 subgoals randomly. All relations were binary. If we only kept the head and tail variables of the chain as the head arguments of the query and views, then there are very few rewritings generated. Thus, we ran our experiments by first considering all variables as distinguished, and then let a few variables be nondistinguished. For the views that had only one subgoal, both variables were still distinguished. We ignored queries that did not have rewritings. Figure 7.9 (a) shows the running time of Corecover if all variables were distinguished, and Figure 7.9 (b) shows the running time if one variable was nondistinguished.

Again, the Corecover algorithm showed good efficiency and scalability. For instance, in the case where all variables were distinguished, it took the algorithm less than 2 seconds to



(a) All variables are distinguished. (b) 1 variable is nondistinguished.

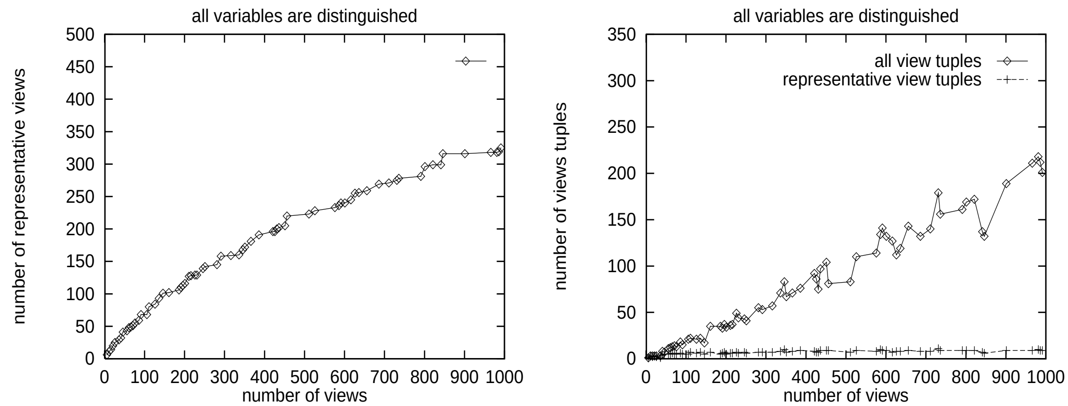
Figure 7.9: Time of generating all GMRs of chain queries.

generate all GMRs for a query when there were 1000 views. In the case where one variable was distinguished, it took the algorithm less than 1.4 seconds to generate all GMRs for a query when there were 1000 views. To illustrate the reason, Figure 7.10 (a) shows that as the number of views increased, the number of equivalence view classes increased with a decreasing slope. Figure 7.10 (b) shows that as the number of views increased, the number of representative view tuples was almost a constant.

In summary, our experiments illustrated two points. (1) The Corecover algorithm has good efficiency and scalability. (2) By grouping views and view tuples into equivalence classes respectively, we can reduce the number of views and view tuples used in the algorithm, thus the algorithm can perform efficiently.

7.8 Conclusions and Related Work

In this chapter, we studied the following important problem for mediators that take the query-centric approach to information integration: given a query on base relations and a set of source views over the same relations, how can we answer the query using the views *efficiently*? In other words, we want to know how to generate efficient rewritings using views to answer a query. We studied three cost models. Under the first cost model M_1 that



(a) Views. (b) View tuples.

Figure 7.10: Number of equivalence classes for chain queries.

considers the number of subgoals in a plan, we gave a search space for optimal rewritings for a query. We analyzed the internal relationship of all rewritings of a query using views, and developed an efficient algorithm, Corecover, for finding rewritings with the minimum number of subgoals.

We then considered a cost model M_2 that counts the sizes of relations in a physical plan. We also gave a search space for finding optimal rewritings under M_2. Surprisingly, we need to consider the fact that introduction of more view subgoals might make a rewriting more efficient. Finally, we considered a cost model M_3 that allows some nonrelevant attributes to be dropped during the evaluation of a plan without changing the result of the computation. We proposed a heuristic for an optimizer to drop more attributes than the traditional supplementary-relation approach. Experiments showed that the Corecover algorithm has good efficiency and scalability. Among other subtleties, this good result is also due to the fact that the algorithm (1) considers only a small number of *relevant* view tuples for the rewritings, and (2) uses a concise representation of these view tuples.

Related Work

The problem of finding whether there exists an equivalent rewriting for a query using views was studied in [LMSS95]. Recently, several algorithms have been developed for finding rewritings of queries using views, such as the bucket algorithm [GM99a, LRO96], the inverse-rule algorithm [DG97, Qia96, AGK99], the MiniCon algorithm [PL00], and the Shared-Variable-Bucket algorithm [Mit01]. (See [Lev00] for a survey.) These algorithms aim at generating *contained rewritings* for a query that compute a subset of the answer to the query, while we want to find *equivalent rewritings* that compute the same answer to a query. Another difference is that they take the open-world assumption, thus they have no optimization considerations, since two equivalent rewritings for a query can still produce different answers under the assumption. We take the closed-world assumption, under which two equivalent rewritings produce the same answer for any instance of the view database.

Our algorithms for generating optimal rewritings share some observations with the MiniCon algorithm. In addition, as we saw in Section 7.4, since we want to generate equivalent rewritings rather than contained rewritings, this different goal helps us develop more efficient algorithms by considering a containment mapping from the expansion of an equivalent rewriting to the query.

Another related work is [CKPS95], which also considers generating efficient plans using materialized views by replacing subgoals in a query with view literals. There are two differences between our work and that work. (1) We take a two-step approach by separating the rewriting generator and optimizer into two modules, while [CKPS95] combines them into one module. (2) [CKPS95] does not consider the possibility that introduction of new view literals can make a rewriting more efficient, as shown by the rewritings P_2 and P_3 in the car-loc-part example. Our work considers this possibility.

Chapter 8

Conclusions and Future Work

In this chapter we summarize the results of this thesis, and discuss future work on information integration.

8.1 Summary of Thesis Results

The motivation of information integration is to allow users to access heterogeneous databases just like accessing one large database. In this thesis we studied research problems on efficient query processing in information integration. We first addressed several problems on answering queries when relations have limited access patterns. Then we discussed how to use mediator caching to reduce the number of source accesses to answer a query. Finally we studied the problem of generating efficient plans for queries using views.

8.1.1 Query Processing in the Presence of Binding Patterns

In Chapters 3, 4, and 5 we studied optimization problems when relations have limited access patterns (i.e., binding patterns). Chapter 3 addressed the problem of generating feasible and efficient query plans for conjunctive queries when relations have binding patterns. One difference between this problem and traditional query-optimization problems is that we need to take the relation restrictions into consideration when generating plans.

We first considered a simple cost model that counts the number of source accesses in a physical plan, since each source access could be very expensive in mediation systems. Under this cost model, we developed two efficient algorithms for finding good plans. The first algorithm, called CHAIN, is based on a greedy strategy. We showed a linear bound on its worst-case performance compared to optimal plans. The second algorithm, called PARTITION, uses a partitioning scheme. It generates optimal plans in more scenarios than CHAIN, even though it has no bound on the margin by which it misses optimal plans. Our experimental results showed that both algorithms can find good plans very efficiently. We also discussed how to extend the results under this cost model to general cost models.

In Chapter 4 we showed how to compute the maximal answer to a query by borrowing bindings from other source relations not mentioned in the query. We focused on a special class of conjunctive queries, called "connection queries." We studied two optimization problems to trim useless source accesses. The first problem is to decide which relations should be accessed to compute the maximal answer to a query. We developed a polynomial-time algorithm for finding all relevant relations for a query. The second problem is to test query containment when relations have binding patterns. We proved this problem is decidable by using decidability results of monadic programs. We then proposed an efficient algorithm for testing the boundedness of the plan for computing the maximal answer to a query. If one program in the containment test is bounded, the test can be performed efficiently.

In Chapter 5 we answered the following question: given a query on relations with binding patterns, is there a plan that can compute the complete answer to the query by accessing the relations using legal patterns? If so, the query is called "stable." Since we cannot retrieve all tuples from relations due to their binding restrictions, we need to reason about whether the answer computed by a plan is complete or not. We studied this problem for a variety of queries. For conjunctive queries, we showed that testing the stability of a conjunctive query is $\mathcal{NP}$-complete. We then proposed two algorithms for testing the stability of a conjunctive query. Similarly, we developed two algorithms for testing the stability of a union of conjunctive queries, and an algorithm for conjunctive queries with arithmetic comparisons. Finally, we showed that stability of datalog queries is undecidable, and give a sufficient condition for computing the complete answer to a datalog query.

8.1.2 Mediator Caching

In Chapter 6 we studied how to use mediator caching to reduce the number of source accesses to answer a query. When the mediator has limited resources, we need to decide what query results (views) to keep in the mediator cache. Our goal is to use the cached results to answer as many future queries as possible. We introduced a new concept, called "query-answering power" of a view set, and discussed how to minimize a view set without losing its power to answer queries. We also studied the problem when there is a predefined set of queries a user can ask.

8.1.3 Using Views to Generate Efficient Plans for Queries

In Chapter 7 we discussed how to use views to generate efficient, equivalent plans to answer a conjunctive query. We take the closed-world assumption, under which views are materialized from base relations. Most previous algorithms on answering queries using views take the open-world assumptions, under which views describing sources are defined in terms of abstract predicates. We first considered a cost model that counts the number of subgoals in a physical plan, and showed a search space that is guaranteed to include an optimal rewriting. We also developed an efficient algorithm, called "CoreCover," for finding rewritings with the minimum number of subgoals. We then considered a cost model that counts the sizes of intermediate relations of a physical plan, and gave a search space for finding optimal rewritings. Our third cost model allows attributes to be dropped in intermediate relations. We showed that, by careful variable renaming, it is possible to do better than the standard supplementary-relation approach by dropping attributes that the latter approach would retain. Experiments showed that our algorithm of generating optimal rewritings has good efficiency and scalability.

8.2 Future Work

Now we discuss several research areas for future work.

8.2.1 Compute Maximal Answers to General Queries

Chapter 4 focused on how to compute maximal answers to connection queries. Section 4.7.3 extended some results to conjunctive queries. There are still several open problems on computing maximal answers to general queries that cannot be represented as connection queries (e.g., conjunctive queries). (1) For the datalog program of a conjunctive query that computes the maximal answers (see Section 4.3), it is still not clear how to decide which relations are relevant to the program. The "kernel" concept in Section 4.5 might give us a hint on how to trim useless source relations. (2) We want to know how to test the boundedness of the program of a conjunctive query. (3) In addition, it is an open problem how to evaluate the program of a query efficiently. We might use existing algorithms for evaluating datalog programs [BR86], such as the magic-sets techniques [BR87]. Finding efficient algorithms for evaluating the program of a query deserves investigation.

8.2.2 Finer-Granularity Mediator Caching

In Chapter 6 we assumed that the basic unit of mediator caching is the answer to a query. Some query results can overlap. For instance, query "finding Palo Alto car dealers" and query "finding Toyota dealers" share the Toyota dealers in Palo Alto. By considering finer-granularity mediator caching, we can avoid storing multiple copies of the same information. For example, we can cache each tuple in a relation, or cache common subexpressions [Fin82] that are shared by many queries. The finer the granularity is, the more effectively we can use the cache space. However, we may also need more cost to maintain the results, such as the meta information about these results. The cache replacement policy can be more complicated. We need to consider the tradeoff between the cache maintenance cost and cache storage effectiveness to decide the appropriate granularity.

8.2.3 Generating Efficient Plans Using Views for General Queries

Chapter 7 studied how to generate efficient rewritings using views for conjunctive queries. One future direction is to extend the results to general queries. For instance, it is still not clear how to extend the results to the case where queries and views have built-in predicates, and the case where we want to find maximally-contained rewritings of a query. In both cases, it is known that a rewriting of a query can be a union of conjunctive queries. Thus the challenge is how to evaluate the performance of a union of conjunctive queries, as shown by the following example borrowed from [LMSS95]. Consider the query and views:

$$\begin{array}{lll} \text{Query } Q\text{:} & q(X,Y,U,W) & \text{:- } p(X,Y), r(U,W), r(W,U) \\ \text{Views:} & v_1(A,B,C,D) & \text{:- } p(A,B), r(C,D), C \leq D \\ & v_2(E,F) & \text{:- } r(E,F) \end{array}$$

The following rewriting P_1 of Q using the two views is a union of two conjunctive queries, and it uses only the variables in Q:

$$\begin{array}{lll} P_1\text{:} & q(X,Y,U,W) & \text{:- } v_1(X,Y,U,W), v_2(W,U) \\ & q(X,Y,U,W) & \text{:- } v_1(X,Y,W,U), v_2(U,W) \end{array}$$

However, the following rewriting P_2 has only one conjunctive query, and it uses new variables C and D that are not in the query.

$$P_2: \; q(X,Y,U,W) \coloneq v_1(X,Y,C,D), v_2(U,W), v_2(W,U)$$

Notice that P_2 uses fewer conjunctive queries than P_1. However, this fact does not imply that P_2 is always more efficient than P_1, since P_2 uses three view subgoals, while each conjunctive query in P_1 uses only two view subgoals. It is an open problem how to compare the efficiency of two unions of conjunctive queries, and how to find efficient rewritings under certain cost models.

8.2.4 Extraction from Web Pages

In many data-integration applications such as comparison shopping, it is critical to write wrappers to extract data from Web pages. Even though several techniques (e.g., [Raj01]) have been proposed to do the extraction from HTML pages semi-automatically, it is an open issue how to reduce the efforts of human beings in this process. One possible direction is to use the similarity and difference among the pages from the same Web site, such as the book pages from AMAZON.COM and the electronic pages from EGGHEAD.COM. These pages are generated from a back-end database using the same template. By analyzing the difference of these pages, we might be able to extract the data while minimizing the amount of human efforts.

8.2.5 Dealing with Source Heterogeneity

In the thesis we assume that using wrapper technologies, data from different sources is translated into a uniform model. For the same information, different sources could have different representations. For example, the actor name "Keanu Reeves" can be represented as "Keanu Reeves," "Reeves, Keanu," "K. Reeves," or "Reeves" in different databases. How to use wrappers to deal with the heterogeneity of different sources still deserves future research. In addition, this thesis assumes that source data is translated into the relational data model. A future direction is to integrate different information sources that use semi-structured data model or XML.

Bibliography

[ACPS96] Sibel Adali, K. Selçuk Candan, Yannis Papakonstantinou, and V. S. Subrahmanian. Query caching and optimization in distributed mediator systems. In *SIGMOD*, pages 137–148, 1996.

[AD98] Serge Abiteboul and Oliver M. Duschka. Complexity of answering queries using materialized views. In *PODS*, pages 254–263, 1998.

[AGK99] Foto N. Afrati, Manolis Gergatsoulis, and Theodoros G. Kavalieros. Answering queries using materialized views with disjunctions. In *ICDT*, pages 435–452, 1999.

[AHU83] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *Data Structures and Algorithms*. Addison-Wesley Publishing Company, 1983.

[AHY83] Peter M. G. Apers, Alan R. Hevner, and S. Bing Yao. Optimization algorithms for distributed queries. *IEEE Transactions on Software Engineering (TSE)*, 9(1):57–68, 1983.

[ALU01] Foto Afrati, Chen Li, and Jeffrey D. Ullman. Generating efficient plans using views. In *SIGMOD*, pages 319–330, 2001.

[ASU79a] Alfred V. Aho, Yehoshua Sagiv, and Jeffrey D. Ullman. Efficient optimization of a class of relational expressions. *ACM Transactions on Database Systems (TODS)*, 4(4):435–454, 1979.

[ASU79b] Alfred V. Aho, Yehoshua Sagiv, and Jeffrey D. Ullman. Equivalences among relational expressions. *SIAM Journal on Computing*, 8(2):218–246, 1979.

[Bas98] Julie Basu. Associative caching in client-server databases. *Ph.D. Thesis, Computer Science Dept., Stanford Univ.*, 1998.

[BCF+99] Lee Breslau, Pei Cao, Li Fan, Graham Phillips, and Scott Shenker. Web caching and Zipf-like distributions: Evidence and implications. In *Proceedings of the INFOCOM'99 conference*, 1999.

[BGW+81] Philip A. Bernstein, Nathan Goodman, Eugene Wong, Christopher L. Reeve, and James B. Rothnie Jr. Query processing in a system for distributed databases (SDD-1). *ACM Transactions on Database Systems (TODS)*, 6(4):602–625, 1981.

[BPT97] Elena Baralis, Stefano Paraboschi, and Ernest Teniente. Materialized view selection in a multidimensional database. In *Proc. of VLDB*, 1997.

[BR86] François Bancilhon and Raghu Ramakrishnan. An amateur's introduction to recursive query processing strategies. In *SIGMOD*, pages 16–52, 1986.

[BR87] Catriel Beeri and Raghu Ramakrishnan. On the power of magic. In *PODS*, pages 269–283, 1987.

[C+94] Sudarshan S. Chawathe et al. The TSIMMIS project: Integration of heterogeneous information sources. *IPSJ*, pages 7–18, 1994.

[CDSS98] Sophie Cluet, Claude Delobel, Jerome Simeon, and Katarzyna Smaga. Your mediators need data conversion! In *SIGMOD*, pages 177–188, 1998.

[CG00] Rada Chirkova and Michael R. Genesereth. Linearly bounded reformulations of conjunctive databases. *DOOD*, 2000.

[CGKV88] Stavros S. Cosmadakis, Haim Gaifman, Paris C. Kanellakis, and Moshe Y. Vardi. Decidable optimization problems for database logic programs. *STOC*, pages 477–490, 1988.

[CGM00] Junghoo Cho and Hector Garcia-Molina. Synchronizing a database to improve freshness. *SIGMOD*, 2000.

[CKPS95] Surajit Chaudhuri, Ravi Krishnamurthy, Spyros Potamianos, and Kyuseok Shim. Optimizing queries with materialized views. In *ICDE*, pages 190–200, 1995.

[CLR90] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. MIT Presss, 1990.

[CM77] Ashok K. Chandra and Philip M. Merlin. Optimal implementation of conjunctive queries in relational data bases. *STOC*, pages 77–90, 1977.

[CM95] Sophie Cluet and Guido Moerkotte. On the complexity of generating optimal left-deep processing trees with cross products. In *ICDT*, pages 54–67, 1995.

[Cod70] E. F. Codd. A relational model of data for large shared data banks. *CACM*, 13(6):377–387, 1970.

[CV92] Surajit Chaudhuri and Moshe Y. Vardi. On the equivalence of recursive and nonrecursive datalog programs. In *PODS*, pages 55–66, 1992.

[CW91] Stefano Ceri and Jennifer Widom. Deriving production rules for incremental view maintenance. In *Proc. of VLDB*, 1991.

[Del78] Claude Delobel. Normalization and hierarchical dependencies in the relational data model. *TODS*, 3(3):201–222, 1978.

[DG97] Oliver M. Duschka and Michael R. Genesereth. Answering recursive queries using views. In *PODS*, pages 109–116, 1997.

[DL97] Oliver M. Duschka and Alon Levy. Recursive plans for information gathering. In *IJCAI*, 1997.

[Dus97] Oliver M. Duschka. Query planning and optimization in information integration. *Ph.D. Thesis, Computer Science Dept., Stanford Univ.*, 1997.

[ES80] Robert S. Epstein and Michael Stonebraker. Analysis of distributed data base processing strategies. In *Proc. of VLDB*, pages 92–101, 1980.

[Fag77] Ronald Fagin. Multivalued dependencies and a new normal form for relational databases. *TODS*, 2(3):262–278, 1977.

[Fin82] Sheldon J. Finkelstein. Common subexpression analysis in database applications. In *SIGMOD*, pages 235–245, 1982.

[FLMS99] Daniela Florescu, Alon Levy, Ioana Manolescu, and Dan Suciu. Query optimization in the presence of limited access patterns. In *SIGMOD*, pages 311–322, 1999.

[GHRU97] Himanshu Gupta, Venky Harinarayan, Anand Rajaraman, and Jeff Ullman. Index selection in olap. In *ICDE*, 1997.

[GJ79] Michael R. Garey and David S. Johnson. *Computers and Intractability : A Guide to the Theory of Np-Completeness*. Freeman, San Francisco, 1979.

[GKD97] Michael R. Genesereth, Arthur M. Keller, and Oliver M. Duschka. Infomaster: An information integration system. In *SIGMOD*, pages 539–542, 1997.

[GLPK94] César A. Galindo-Legaria, Arjan Pellenkoft, and Martin L. Kersten. Fast, randomized join-order selection - why use transformations? In *VLDB*, pages 85–95, 1994.

[GM99a] Gösta Grahne and Alberto O. Mendelzon. Tableau techniques for querying information sources through global schemas. In *ICDT*, pages 332–347, 1999.

[GM99b] Himanshu Gupta and Inderpal Mumick. Selection of views to materialize under a maintenance-time constraint. In *ICDT*, 1999.

[GMSV93] Haim Gaifman, Harry G. Mairson, Yehoshua Sagiv, and Moshe Y. Vardi. Undecidable optimization problems for database logic programs. *Journal of the ACM*, pages 683–713, 1993.

[GMUW00] Hector Garcia-Molina, Jeffrey D. Ullman, and Jennifer Widom. *Database System Implementation*. Prentice Hall, 2000.

[Gry99] Jarek Gryz. Query rewriting using views in the presence of functional and inclusion dependencies. *Information Systems*, 24(7):597–612, 1999.

[GSUW94] Ashish Gupta, Yehoshua Sagiv, Jeffrey D. Ullman, and Jennifer Widom. Constraint checking with partial information. In *PODS*, pages 45–55, 1994.

[Gup94] Ashish Gupta. Partial information based integrity constraint checking. *Ph.D. Thesis, Computer Science Dept., Stanford Univ.*, 1994.

[HGMN+97] Joachim Hammer, Hector Garcia-Molina, Svetlozar Nestorov, Ramana Yerneni, Markus M. Breunig, and Vasilis Vassalos. Template-based wrappers in the TSIMMIS system. *SIGMOD*, pages 532–535, 1997.

[HGMW+95] Joachim Hammer, Hector Garcia-Molina, Jennifer Widom, Wilburt Labio, and Yue Zhuge. The Stanford Data Warehousing Project. In *IEEE Data Eng. Bulletin, Special Issue on Materialized Views and Data Warehousing*, 1995.

[HKWY97] Laura M. Haas, Donald Kossmann, Edward L. Wimmers, and Jun Yang. Optimizing queries across diverse data sources. In *Proc. of VLDB*, pages 276–285, 1997.

[HRU96] Venky Harinarayan, Anand Rajaraman, and Jeff Ullman. Implementing data cubes efficiently. In *SIGMOD*, 1996.

[IK84] Toshihide Ibaraki and Tiko Kameda. On the optimal nesting order for computing n-relational joins. *TODS*, 9(3):482–502, 1984.

[IK90] Yannis E. Ioannidis and Younkyung Cha Kang. Randomized algorithms for optimizing large join queries. In *SIGMOD*, pages 312–321, 1990.

[IK93] W. H. Inmon and Chuck. Kelley. *Rdb/VMS: Developing the Data Warehouse.* QED Publishing Group, Boston, Massachussetts, 1993.

[Ioa85] Yannis E. Ioannidis. A time bound on the materialization of some recursively defined views. In Alain Pirotte and Yannis Vassiliou, editors, *Proc. of VLDB*, pages 219–226. Morgan Kaufmann, 1985.

[IW87] Yannis E. Ioannidis and Eugene Wong. Query optimization by simulated annealing. In Umeshwar Dayal and Irving L. Traiger, editors, *SIGMOD*, pages 9–22. ACM Press, 1987.

[JK83] David S. Johnson and Anthony C. Klug. Optimizing conjunctive queries that contain untyped variables. *SIAM Journal on Computing*, 12(4):616–640, 1983.

[KBZ86] Ravi Krishnamurthy, Haran Boral, and Carlo Zaniolo. Optimization of nonrecursive queries. In *VLDB*, pages 128–137, 1986.

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

[LBU01] Chen Li, Mayank Bawa, and Jeffrey D. Ullman. Minimizing view sets without losing query-answering power. In *ICDT*, pages 99–113, 2001.

[LC99] Chen Li and Edward Chang. Testing query containment in the presence of limited access patterns. *Technical report, Computer Science Dept., Stanford Univ.,* `http://dbpubs.stanford.edu:8090/pub/1999-12`, 1999.

[LC00] Chen Li and Edward Chang. Query planning with limited source capabilities. In *ICDE*, pages 401–412, 2000.

[LC01a] Chen Li and Edward Chang. Answering queries with useful bindings. In *ACM Transactions on Database Systems (TODS)*, 2001.

[LC01b] Chen Li and Edward Chang. On answering queries in the presence of limited access patterns. In *ICDT*, pages 99–113, 2001.

[Lev00] Alon Levy. Answering queries using views: A survey. *Technical report, Computer Science Dept., Washington Univ.*, 2000.

[LMSS95] Alon Levy, Alberto O. Mendelzon, Yehoshua Sagiv, and Divesh Srivastava. Answering queries using views. In *PODS*, pages 95–104, 1995.

[LRO96] Alon Levy, Anand Rajaraman, and Joann J. Ordille. Querying heterogeneous information sources using source descriptions. In *Proc. of VLDB*, pages 251–262, 1996.

[LYV+98] Chen Li, Ramana Yerneni, Vasilis Vassalos, Hector Garcia-Molina, Yannis Papakonstantinou, Jeffrey D. Ullman, and Murty Valiveti. Capability based mediation in TSIMMIS. In *SIGMOD*, pages 564–566, 1998.

[Mit01] Prasenjit Mitra. An algorithm for answering queries efficiently using views. In *Proceedings of the Australasian Database Conference*, 2001.

[MKW00] Prasenjit Mitra, Martin Kersten, and Gio Wiederhold. Graph-oriented model for articulation of ontology interdependencies. In *EDBT*, 2000.

[MLF00] Todd Millstein, Alon Levy, and Marc Friedman. Query containment for data integration systems. In *PODS*, 2000.

[Mor88] Katherine A. Morris. An algorithm for ordering subgoals in NAIL! In *PODS*, pages 82–88, 1988.

[MW97] David A. Maluf and Gio Wiederhold. Abstraction of representation for interoperation. *International Syposium on Methodologies for Intelligent Systems (ISMIS)*, pages 441–455, 1997.

[NS87] Jeffrey F. Naughton and Yehoshua Sagiv. A decidable class of bounded recursions. In *PODS*, pages 227–236. ACM, 1987.

[OL90] Kiyoshi Ono and Guy M. Lohman. Measuring the complexity of join enumeration in query optimization. In *Proc. of VLDB*, pages 314–325. Morgan Kaufmann, 1990.

[PGH96] Yannis Papakonstantinou, Ashish Gupta, and Laura M. Haas. Capabilities-based query rewriting in mediator systems. In *Proceedings of the Fourth International Conference on Parallel and Distributed Information Systems, December 18-20, 1996, Miami Beach, Florida, USA*, pages 170–181. IEEE Computer Society, 1996.

[PGLK97] Arjan Pellenkoft, César A. Galindo-Legaria, and Martin L. Kersten. The complexity of transformation-based join enumeration. In *Proc. of VLDB*, pages 306–315. Morgan Kaufmann, 1997.

[PGMW95] Yannis Papakonstantinou, Hector Garcia-Molina, and Jennifer Widom. Object exchange across heterogeneous information sources. In *ICDE*, pages 251–260, 1995.

[PL00] Rachel Pottinger and Alon Levy. A scalable algorithm for answering queries using views. In *Proc. of VLDB*, 2000.

[PS82] Christos H. Papadimitriou and Ken Steiglitz. *Combinatorial optimization: algorithms and complexity.* Prentice-Hall, 1982.

[Qia96] Xiaolei Qian. Query folding. In *ICDE*, pages 48–55, 1996.

[Raj01] Anand Rajaraman. Constructing virtual databases on the World-Wide Web. *Ph.D. Thesis, Computer Science Dept., Stanford Univ.*, 2001.

[RSS96] Kenneth A. Ross, Divesh Srivastava, and S. Sudarshan. Materialized view maintenance and integrity constraint checking: Trading space for time. In *SIGMOD*, 1996.

[RSU95] Anand Rajaraman, Yehoshua Sagiv, and Jeffrey D. Ullman. Answering queries using templates with binding patterns. In *PODS*, pages 105–112, 1995.

[SAC+79] Patricia G. Selinger, Morton M. Astrahan, Donald D. Chamberlin, Raymond A. Lorie, and Thomas G. Price. Access path selection in a relational database management system. In *SIGMOD*, pages 23–34, 1979.

[Sag85] Yehoshua Sagiv. On computing restricted projections of representative instances. In *PODS*, pages 171–180. ACM, 1985.

[Sar91] Yatin Saraiya. Subtree elimination algorithms in deductive databases. *Ph.D. Thesis, Computer Science Dept., Stanford Univ.*, 1991.

[SG88] Arun N. Swami and Anoop Gupta. Optimization of large join queries. In *SIGMOD*, pages 8–17, 1988.

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

[SM97] Wolfgang Scheufele and Guido Moerkotte. On the complexity of generating optimal plans with cross products. In *PODS*, pages 238–248. ACM Press, 1997.

[SMK97] Michael Steinbrunn, Guido Moerkotte, and Alfons Kemper. Heuristic and randomized optimization for the join ordering problem. *VLDB Journal*, 6(3):191–208, 1997.

[Swa89] Arun N. Swami. Optimization of large join queries: Combining heuristic and combinatorial techniques. In *SIGMOD*, pages 367–376, 1989.

[SY80] Yehoshua Sagiv and Mihalis Yannakakis. Equivalences among relational expressions with the union and difference operators. *Journal of the ACM*, 27(4):633–655, 1980.

[TS97] Dimitri Theodoratos and Timos Sellis. Data warehouse configuration. In *Proc. of VLDB*, 1997.

[Ull89] Jeffrey D. Ullman. *Principles of Database and Knowledge-base Systems, Volumes II: The New Technologies*. Computer Science Press, New York, 1989.

[Ull97] Jeffrey D. Ullman. Information integration using logical views. In *ICDT*, pages 19–40, 1997.

[UV88] Jeffrey D. Ullman and Moshe Y. Vardi. The complexity of ordering subgoals. In *PODS*, pages 74–81, 1988.

[Var99] Moshe Vardi. Personal communication, 1999.

[VM96] Bennet Vance and David Maier. Rapid bushy join-order optimization with cartesian products. In *SIGMOD*, pages 35–46, 1996.

[VP97] Vasilis Vassalos and Yannis Papakonstantinou. Describing and using query capabilities of heterogeneous sources. In *Proc. of VLDB*, pages 256–265, 1997.

[Wid95] Jennifer Widom. Research problems in data warehousing. In *Proc. of the Intl. Conf. on Information and Knowledge Management*, 1995.

[Wie92] Gio Wiederhold. Mediators in the architecture of future information systems. *IEEE Computer*, 25(3):38–49, 1992.

[YKL97] Jian Yang, Kamalakar Karlapalem, and Qing Li. Algorithms for materialized view design in data warehousing environment. In *Proc. of VLDB*, 1997.

[YLGMU99] Ramana Yerneni, Chen Li, Hector Garcia-Molina, and Jeffrey D. Ullman. Computing capabilities of mediators. In *SIGMOD*, pages 443–454, 1999.

[YLUGM99] Ramana Yerneni, Chen Li, Jeffrey D. Ullman, and Hector Garcia-Molina. Optimizing large join queries in mediation systems. In *ICDT*, pages 348–364, 1999.

[ZO93] Xubo Zhang and Meral Ozsoyoglu. On efficient reasoning with implication constraints. In *DOOD*, pages 236–252, 1993.