QUERY PLANNING AND OPTIMIZATION IN INFORMATION INTEGRATION

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

By
Oliver M. Duschka
December 1997

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

Michael R. Genesereth
(Principal Adviser)

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

Jeffrey D. Ullman

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

Yehoshua Sagiv

Approved for the University Committee on Graduate Studies:

Abstract

Information integration systems, also knows as mediators, information brokers, or information gathering agents, provide uniform user interfaces to varieties of different information sources. With corporate databases getting connected by intranets, and vast amounts of information becoming available over the Internet, the need for information integration systems is increasing steadily.

Our work focuses on query planning in such systems. Query planning is the task of transforming a user query, represented in the user's interface language and vocabulary, into queries that can be executed by the information sources. Every information source might require a different query language and might use different vocabularies. The resulting answers of the information sources need to be translated and combined before the final answer can be reported to the user.

We show that query plans with a fixed number of database operations are insufficient to extract all information from the sources, if functional dependencies or limitations on binding patterns are present. Dependencies complicate query planning because they allow query plans that would otherwise be invalid. We present an algorithm that constructs query plans that are guaranteed to extract all available information in these more general cases. This algorithm is also able to handle datalog user queries.

We examine further extensions of the languages allowed for user queries and for describing information sources: disjunction, recursion and negation in source descriptions, negation and inequality in user queries. For these more expressive cases, we determine the data complexity required of languages able to represent "best possible" query plans.

Acknowledgments

I would like to thank my advisor, Michael R. Genesereth, the members of my reading committee, Yehoshua Sagiv and Jeffrey D. Ullman, the members of my oral exam committee, Russ B. Altman, Alon Y. Levy and John C. Mitchell, and the coauthor of one of my papers, Serge Abiteboul. I want to especially thank Yehoshua Sagiv for making it possible for me to visit him at the Hebrew University in Jerusalem. Thanks to Werner Nutt for several very interesting discussions during this visit. This work couldn't have been done without the intellectual and social environment created by the members of the MUGS research group at Stanford University: Amit Agarwal, Winton Davies, Don Geddis, Elham Ghassemzadeh, Rob Hoskins, Arthur Keller, Prasad Kodur, Petros Maniatisarajan, Sheri Mason, Ofer Matan, Illah Nourbakhsh, Terry Rodriguez, Josefina Sierra, Vishal Sikka, Narinder Singh, Mustafa Syed, H. Scott Roy, and Sanjay Verma. Special thanks to Whitney Carrico and Harish Devarajan for many hours of fruitful discussions. My work was supported by the German National Scholarship Foundation, the Fulbright Commission, the Hermann-Schlosser-Foundation, and a grant from Schlumberger.

Contents

Chapter 1

Introduction

1.1 Information integration

The problem of information integration (a.k.a. information gathering agents) has recently received considerable attention due to the growing number of structured information sources available online. The goal of information integration systems (e.g., TSIMMIS [13, 27], HERMES [1], the Internet Softbot [24], SIMS [4], the Information Manifold [38], Disco [25,49], TransFER [48], Occam [34], Razor [26], Infomaster [19]) is to provide a *uniform* query interface to multiple information sources, thereby freeing the user from having to locate the relevant sources, query each one in isolation, and combine manually the information from the different sources.

Information integration systems are based on the following general architecture. The user interacts with a uniform interface in the form of a set of *global* relation names that are used in formulating queries. These relations are called *world relations*. The actual data is stored in external sources, called the *source relations*. In order for the system to be able to answer queries, we must specify a *mapping* between the world relations and the source relations. A common method to specify these mappings (employed in [38, 34]) is to describe each source relation as the result of a *conjunctive query* (i.e., a single Horn rule) over the world relations. For example, an information source storing nonstop flights offered by United Airlines would be described as follows:

```
CREATE VIEW Flights_by_United
   SELECT number, from, to
   FROM Nonstop
   WHERE airline = 'UA'
```

The relation `Nonstop` is a world relation and can be used in formulating queries, and relation `Flights_by_United` is a source relation.

Given a query from the user, formulated in terms of the world relations, the system must translate it to a query that mentions *only* the source relations, because only these relations are actually available. That is, the system needs to find a query expression, that mentions only the source

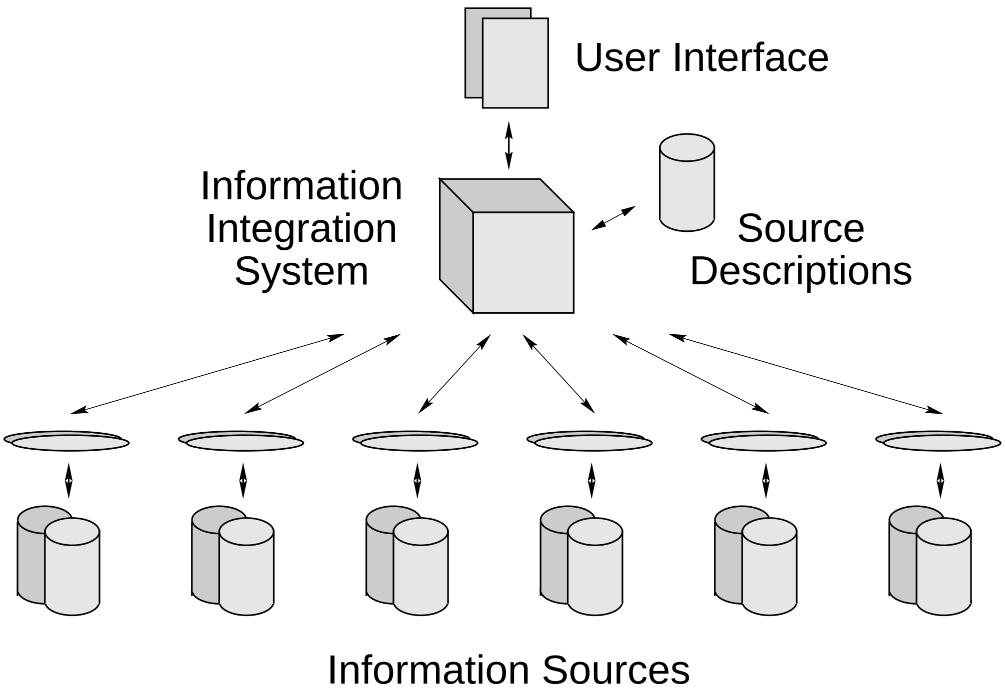


Figure 1.1: *General architecture of an information integration system. The user interface and the information sources can be modeled for the purpose of query planning by sets of relations, called world relations and source relations respectively. Source descriptions relate source and world relations.*

relations, and is equivalent to the original query. The new query is called a *query plan*. The problem of finding a query plan is the same as the problem of *answering queries using views*. In this context, the views are the relations in the sources. The problem of answering queries using views has also been investigated in the database literature because of its importance for query optimization and data warehousing [57,50,10,37,44,43,17].

Most previous work has considered the problem of finding query plans where the query plan is required to be *equivalent* to the original query. In practice, the collection of available information sources may not contain *all* the information needed to answer a query, and therefore, we need to resort to *maximally-contained* query plans. A maximally-contained plan provides *all* the answers that are possible to obtain from the sources, but the expression describing the plan may not be equivalent to the original query. For example, if we only have the `Flights_by_United` source available, and our query asks for all flights departing from San Francisco International Airport, then the following is a maximally-contained query plan:

```
SELECT 'UA',number,to
FROM Flights_by_United
WHERE from = 'SFO'
```

1.2 Answering queries using views

A *view* is the result of evaluating a query. The query that generates the view is called *view definition*. One application of views in current database systems is *security*. For example, a student is only allowed to see his or her own grades in the university database system, but not the grades of other students. This limited access to the database is implemented by defining a view for every student that contains exactly the information that the student is allowed to see. Students then are only allowed to query their view, but not the underlying database. We will use views in order to describe information available from information sources. Data stored by information sources can be seen as views over a global schema.

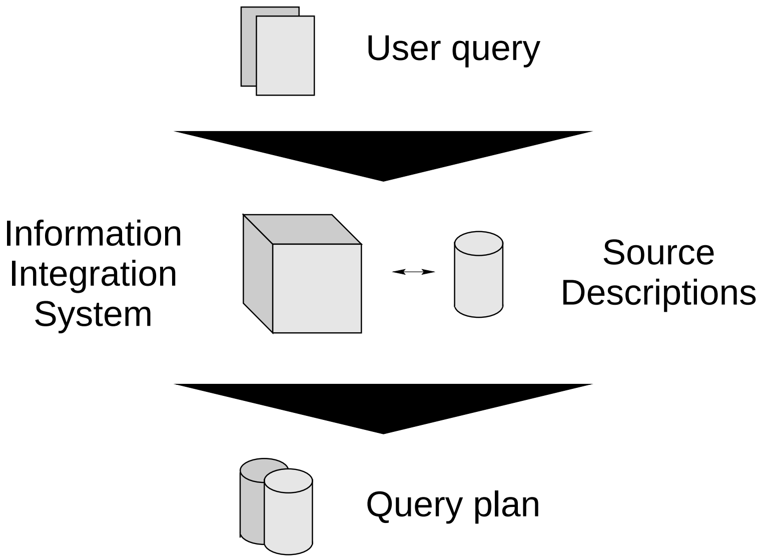


Figure 1.2: *Query planning in information integration systems is the task of transforming a user query, represented in the user's interface language and vocabulary, into queries that can be executed by the information sources. Every information source might require a different query language and might use different vocabularies. The resulting answers of the information sources need to be translated and combined before the final answer can be reported to the user.*

Example 1.2.1 The examples in this section use a global schema with the relation

`Nonstop(airline,number,from,to)`.

The intended meaning of the tuple ⟨`UA`,`2021`,`SFO`,`LAX`⟩ in this relation, for example, is that United Airlines (`UA`) flight 2021 is a nonstop flight from San Francisco (`SFO`) to Los Angeles (`LAX`). Consider the following two views:

```
CREATE VIEW Flights_by_United
```

```
   SELECT from, to
   FROM Nonstop
   WHERE airline = 'UA'

CREATE VIEW Flights_from_SFO
   SELECT airline, number, to
   FROM Nonstop
   WHERE from = 'SFO'
```

View `Flights_by_United` stores the nonstop flights operated by United Airlines, and the view `Flights_from_SFO` stores the nonstop flights out of San Francisco. The first view might represent the data that is available from a United Airlines database, whereas the second view might be stored in a database at San Francisco International Airport. □

1.3 Equivalent query plans

The query planning problem in information integration systems is very closely related to the problem of answering queries using views. User queries are posed in terms of a global schema. Information sources can be seen as views over this global schema. In order to answer a user query from the data available from the information sources, the query must be rewritten so that it only uses the views.

Example 1.3.1 A user might be interested to know which nonstop flights from San Francisco to Los Angeles are offered. The following is the corresponding SQL query:

```
SELECT airline, number
FROM Nonstop
WHERE from = 'SFO' AND to = 'LAX'
```

It is easy to rewrite her query into an equivalent query that uses only the views defined in Example 1.2.1. View `Flights_from_SFO` stores all nonstop flights out of San Francisco, and therefore also all nonstop flights from San Francisco to Los Angeles. It follows that the query rewrite

```
SELECT airline, number
FROM Flights_from_SFO
WHERE to = 'LAX'
```

answers the user's query. The rewriting requires the view `Flights_from_SFO`, but does not require any relations from the global schema. Therefore, it can be answered by querying the database at San Francisco International Airport. □

We will refer to queries that can be answered by using the views as *query plans*.

1.4 Maximally-contained query plans

The user in Example 1.3.1 was lucky because her query could be answered exactly from the available views. In general, however, the available views might not provide all the information that is needed to answer exactly the user's query. In these cases, we still want to give the user some answer, indeed the "best" answer that can be given using only the available views. The query plan that computes this "best" answer is called *maximally-contained* query plan.

Example 1.4.1 The user who was interested in flights from San Francisco to Los Angeles might want to continue her flight to Phoenix. Therefore, she might ask for nonstop flights from Los Angeles to Phoenix:

```
SELECT airline, number
FROM Nonstop
WHERE from = 'LAX' AND to = 'PHX'
```

Given only the two available sources — the United Airlines flight database and the database of San Francisco International Airport — there is no way to find a query plan that only uses these sources and is equivalent to the original query. Nonetheless, it is possible to extract some information out of the available sources. Indeed, if United Airlines offers nonstop flights from Los Angeles to Phoenix, then this information would be available from the United Airlines flight database. The query plan

```
SELECT 'UA', number
FROM Flights_by_United
WHERE from = 'LAX' AND to = 'PHX'
```

requests this information from the United Airlines database. □

1.5 Functional dependencies

Frequently, relations in databases satisfy functional dependencies. For example, consider the relation

`Schedule(airline,number,date,pilot,aircraft)`.

The intended meaning of the tuple ⟨`UA,2021,08/21,Mike,#111`⟩ in this relation is that on August 21st United Airline flight 2021 has Mike as pilot and is using the aircraft with identification number #111. Pilots work for one airline only, and airlines don't have joint ownership of aircraft. Therefore, relation `Schedule` satisfies the functional dependencies `pilot` → `airline` and `aircraft` → `airline`. If we know that pilot Mike works for United Airlines, for example, then we can be sure that Mike does not work for American Airlines as well.

Using functional dependencies, more information might be extracted from information sources than would be possible otherwise. Algorithms that don't take functional dependencies into account in the query planning process might fail to produce all correct answers, i.e. the generated query plans might not be maximally-contained in the user query.

Example 1.5.1 Assume that there is an additional view available described by the following definition:

```
CREATE VIEW Work_Schedule
   SELECT date, from, to, pilot, aircraft
   FROM Nonstop, Schedule
   WHERE Nonstop.airline = Schedule.airline AND
         Nonstop.number = Schedule.number
```

This view would store, for example, the tuple ⟨`08/21`,`SFO`,`LAX`,`Mike`,`#111`⟩ expressing that on August 21st pilot Mike flies from San Francisco to Los Angeles on aircraft #111. A user might be interested in all the pilots that work for the same airline as Mike. The corresponding SQL query is:

```
SELECT S1.pilot
FROM Schedule AS S1, Schedule AS S2
WHERE S1.airline = S2.airline AND
      S2.pilot = 'Mike'
```

View `Work_Schedule` is the only view that stores any information about relation `Schedule`. Unfortunately, this view doesn't store the `airline` attribute explicitly. Therefore, without any consideration of functional dependencies, this query couldn't be answered at all. The functional dependencies `pilot` → `airline` and `aircraft` → `airline` though can be used to extract more information from this view. To see that these functional dependencies might really yield more answers, consider the following instance of view `Work_Schedule`:

date	from	to	pilot	aircraft
08/21	SFO	LAX	Mike	#111
09/05	PHX	ATL	Ann	#111

In this example, Mike and Ann fly the same aircraft. Because of the functional dependency `aircraft` → `airline` it follows that Mike and Ann work for the same airline. In general, the query plan

```
SELECT W1.pilot
FROM Work_Schedule AS W1, Work_Schedule AS W2
WHERE W1.aircraft = W2.aircraft AND
      S2.pilot = 'Mike'
```

is contained in the user query if this functional dependency holds, but is not contained in the user query otherwise. □

Example 1.5.1 showed that functional dependencies might lead to more query plans that are contained in the given user query. Indeed, we will show that there is no maximally-contained query plan in this case if the queries are restricted to use join, projection, selection, and union only. More precisely, for every query plan that is contained in the user query, there is another query plan contained in the user query that might produce some new answers. In order to get a maximally-contained query plan it is necessary to consider query plans that contain *recursion*.

Example 1.5.2 Using recursion there is a query plan that is maximally-contained in the query of Example 1.5.1. The maximally-contained recursive query plan is the following:

```
WITH
   RECURSIVE Pilots AS
      ( SELECT pilot
        FROM Work_Schedule
        WHERE pilot = 'Mike' )
      UNION
      ( SELECT pilot
        FROM Work_Schedule, Aircraft
        WHERE Work_Schedule.aircraft = Aircraft.aircraft ),
   RECURSIVE Aircraft AS
      ( SELECT aircraft
        FROM Work_Schedule
        WHERE pilot = 'Mike' )
      UNION
      ( SELECT aircraft
        FROM Work_Schedule, Pilots
        WHERE Work_Schedule.pilot = Pilots.pilot ),
SELECT pilot
FROM Pilots
```

For the convenience of the reader who might be more used to representing recursive queries in datalog notation instead of SQL, we also give the datalog query that corresponds to the SQL query. Note that in datalog it is conventional to denote relations and constants by lower case names, and variables by upper case names.

$$
\begin{array}{ll}
q(Pilot) & :-\ pilots(Pilot) \\
pilots(Pilot) & :-\ work\_schedule(Date, From, To, mike, Aircraft) \\
pilots(Pilot) & :-\ work\_schedule(Date, From, To, Pilot, Aircraft),\ aircraft(Aircraft) \\
aircraft(Aircraft) & :-\ work\_schedule(Date, From, To, mike, Aircraft) \\
aircraft(Aircraft) & :-\ work\_schedule(Date, From, To, Pilot, Aircraft),\ pilots(Pilot)
\end{array}
$$

The query computes two intermediate relations, `Pilots` and `Aircraft`. Relation `Pilots` stores all the pilots that work for the same airline as Mike, and relation `Aircraft` stores all the aircraft that are owned by the airline that Mike works for. Obviously, if the tuple ⟨08/21,SFO,LAX,Mike,#111⟩ is in view `Work_Schedule`, then Mike is one of the pilots in relation `Pilots`, and aircraft #111 is one of the aircraft in relation `Aircraft`. More interestingly, if ⟨09/12,ORD,JFK,John,#222⟩ is in view `Work_Schedule` and aircraft #222 is in relation `Aircraft`, then John is also one of the pilots in relation `Pilots` because of the functional dependency `aircraft` → `airline`. Similarly, if ⟨09/23,BOS,SEA,Sue,#333⟩ is in view `Work_Schedule` and Sue is a pilot in relation `Pilots`, then aircraft #333 is also one of the aircraft in relation `Aircraft` because of the functional dependency `pilot` $\rightarrow$ `airline`. □

1.6 Limitations on binding patterns

We model information sources by describing the data that they store. It might be the case though that sources do not support arbitrary queries on this data. Instead, they might require that queries provide values for some of the attributes. We refer to these restrictions as *limitations on binding patterns*. One of the reason for limitations on binding patterns is security. A directory server of a company, for example, usually doesn't allow users to ask for all employees, but only for the phone number of a specific employee. Another reason for limitations on binding patterns is performance. For example, a query for the phone number given a name might be supported by an index, but a query for a name given a phone number might require to scan the entire relation. We express limitations on binding patterns by stating which arguments have to be *bound*. In the directory example, the name attribute is likely required to be bound.

Unlike in the presence of functional dependencies, limitations on binding patterns do not increase the number of possible query plans. A query plan that respects limitations on binding patterns is also a query plan if we disregard these limitations. However, some query plans that are contained in the user query might not respect the limitations on binding patterns, and are therefore not executable. Especially, a maximally-contained query plan might turn out to be not executable.

Example 1.6.1 Consider again the view `Flights_by_United` from Example 1.2.1. Assume a user asks for all nonstop flights offered by United Airlines:

```
SELECT number, from, to
FROM Nonstop
WHERE airline = 'UA'
```

Without limitations on binding patterns, this query has a simple maximally-contained query plan:

```
SELECT number, from, to
FROM Flights_by_United
```

However, the United Airlines database might require that the `from` attribute is bound in all queries. The query plan above violates this limitation on binding patterns. Therefore, it could not be executed. A query plan that is executable is for example the following:

```
SELECT number, from, to
FROM Flights_by_United
WHERE from = 'SFO'
```

□

Functional dependencies increase the number of possible query plans, and limitations on binding patterns decrease the number of possible query plans. Both cases complicate query planning. There might be *no* conjunctive query plan that respects the limitations on binding patterns and is maximally-contained in the user query. As in the presence of functional dependencies, recursive query plans might be required as maximally-contained query plans that respect the limitations on binding patterns.

Example 1.6.2 Continuing with Example 1.6.1, the following recursive query is a maximally-contained query plan of the user query that respects the limitations on binding patterns:

```
WITH
   RECURSIVE Flights AS
      ( SELECT number, from, to
        FROM Flights_by_United
        WHERE from = 'SFO' )
      UNION
      ( SELECT F.number, F.from, F.to
        FROM Flights, Flights_by_United AS F
        WHERE F.from = Flights.to )
SELECT number, from, to
FROM Flights
```

$$
\begin{array}{ll}
q(Number, From, To) & :- \ flights(Number, From, To) \\
flights(Number, sfo, To) & :- \ flights\_by\_united(Number, sfo, To) \\
flights(Number, From, To) & :- \ flights(N, F, From),\ flights\_by\_united(Number, From, To)
\end{array}
$$

The example assumes that the symbol for San Francisco International Airport, `SFO`, is known and no other airport symbols are known. The query plan is executable because in the first part of the query, attribute `from` of `Flights_by_United` is bound to `SFO`, and in the second part of the query, this attribute is bound to the values of the `to` attribute of relation `Flights`. The query computes all nonstop flights offered by United Airlines that depart from an airport that can be reached from San Francisco on United flights. Under the given limitations on binding patterns, and assuming that `SFO` is the only known airport symbol, this query plan is maximally-contained in the user query. □

1.7 Recursive user queries

So far, we have restricted users to ask only nonrecursive queries. This query language is sufficient for many practical applications. One might consider though allowing users to issue more expressive queries that include recursion. It is obvious that query plans that answer recursive user queries also might require recursion. The following is an example:

Example 1.7.1 Assume a user has gathered plenty of frequent flyer miles on United Airlines. Now she wants to know which destinations she can reach from her home in San Francisco by flying United.

All cities that United flies to nonstop from San Francisco are clearly possible destinations. But there are more. Other cities can be reached by taking two nonstop flights in a row, or three, or four, ...

The SQL query that corresponds to the user's request requires recursion, because it is not known in advance how many nonstop flights are required to reach all the possible destinations from San Francisco on United. The following is the query that computes the desired answer:

```
WITH
   RECURSIVE Flights AS
      (  SELECT to
         FROM Nonstop
         WHERE airline = 'UA' AND from = 'SFO' )
      UNION
      (  SELECT Nonstop.to
         FROM Flights, Nonstop
         WHERE airline = 'UA' AND
               Flights.to = Nonstop.from )
SELECT to
FROM Flights
```

$$
\begin{array}{ll}
q(To) & :- \ flights(To) \\
flights(To) & :- \ nonstop(ua, Number, sfo, To) \\
flights(To) & :- \ flights(Stopover),\ nonstop(ua, Number, Stopover, To)
\end{array}
$$

If the only available view is `Flights_by_United`, then the maximally-contained query plan not surprisingly also requires recursion:

```
WITH
   RECURSIVE Flights AS
      (  SELECT to
         FROM Flights_by_United
         WHERE from = 'SFO' )
      UNION
      (  SELECT Flights_by_United.to
         FROM Flights, Flights_by_United
         WHERE Flights.to = Flights_by_United.from )
SELECT to
FROM Flights
```

$$
\begin{array}{ll}
q(To) & :- \ flights(To) \\
flights(To) & :- \ flights\_by\_united(Number, sfo, To) \\
flights(To) & :- \ flights(Stopover),\ flights\_by\_united(Number, Stopover, To)
\end{array}
$$

□

1.8 Inequality

We showed in Sections 1.5, 1.6, and 1.7 that query plans with recursion might be required in the presence of functional dependencies, limitations on binding patterns, or recursive user queries. There always exists a maximally-contained recursive query plan in these cases. The recursive query plans we are considering can be expressed in a language called *datalog*. One of the features of datalog programs is that they can be executed in polynomial time.

In this section, we consider inequality constraints in user queries. It turns out that inequality constraints make it much harder to compute all the answers to queries using the views. Indeed, this computation cannot be done in polynomial time (unless P = NP). The following example illustrates the increased difficulty of the reasoning that is necessary.

Example 1.8.1 Assume two views are available. The first view is `Flights_from_SFO` defined in Example 1.2.1. The second view stores departure airport, stopover airport and arrival airport of two-leg flights that are operated by the same airline on both legs of the flight.

```
CREATE VIEW Flights_with_Stopover
  SELECT F1.from, F1.to AS stopover, F2.to
  FROM Nonstop AS F1, Nonstop AS F2
  WHERE F1.to = F2.from AND
        F1.airline = F2.airline
```

A user might be interested to know all the airlines with direct competition on at least one of their nonstop flights. The following is the corresponding SQL query:

```
SELECT F1.airline
FROM Nonstop AS F1, Nonstop AS F2
WHERE F1.from = F2.from AND F1.to = F2.to AND
      F1.airline <> F2.airline
```

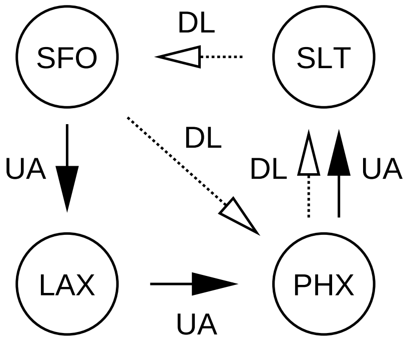
To see that computing all answers for this query that can be concluded from the views is quite complicated, consider the instance of the relation `Nonstop` and the views `Flights_from_SFO` and `Flights_with_Stopover` in Figure 1.8.1.

For this instance, "United Airlines" is an answer to the query, because it is in competition with Delta Airlines (`DL`) on the route from Phoenix to Salt Lake City. Let us see how we can be sure that United Airlines has competition on one of its nonstop flights.

We know because of the tuple ⟨LAX,UA⟩ in view `Flights_from_SFO` that United Airlines flies from San Francisco to Los Angeles. Moreover, because of the tuple ⟨SFO,LAX,PHX⟩ in view `Flights_with_Stopover` there is some airline flying from San Francisco via Los Angeles to Phoenix. If this airline is not United, then United has competition on the San Francisco to Los Angeles route. So let us assume that this airline is again United. Because of the tuple ⟨LAX,PHX,SLT⟩ in view `Flights_with_Stopover` we know that there is some airline flying from Los Angeles via Phoenix to Salt Lake City. If this airline is not United, then United has competition on the route from Los

`Nonstop`

from	to	airline
SFO	LAX	UA
LAX	PHX	UA
PHX	SLT	UA
SFO	PHX	DL
PHX	SLT	DL
SLT	SFO	DL



`Flights_from_SFO`

to	airline
LAX	UA
PHX	DL

`Flights_with_Stopover`

from	stopover	to
SFO	LAX	PHX
LAX	PHX	SLT
SFO	PHX	SLT
PHX	SLT	SFO
SLT	SFO	PHX

Figure 1.3: *An instance of the relation and views in Example 1.8.1*

Angeles to Phoenix. If this airline is again United, then we can continue correspondingly with tuple ⟨PHX,SLT,SFO⟩ in `Flights_with_Stopover` and ⟨SLT,SFO,PHX⟩ in `Flights_with_Stopover` to conclude that either United has competition on the Phoenix to Salt Lake City or Salt Lake City to San Francisco route respectively, or United flies from Phoenix via Salt Lake City to San Francisco and from Salt Lake City via San Francisco to Phoenix. But in the latter case, United has competition on the San Francisco to Phoenix route because of tuple ⟨PHX,DL⟩ in `Flights_from_SFO`. We can conclude from this case analysis that United Airlines must have a competitor for at least one of its nonstop flights. □

1.9 More expressive languages

As we indicated in the previous section, adding to the expressive power of the query languages used might complicate the query planning process. In this thesis we are going to consider a variety of further extensions of these languages. For example, we examine the effect of allowing union and recursion in the view definitions.

1.10 Organization of thesis

The material in this thesis is organized in five chapters. Chapter 2 covers recursive query plans. As we saw in Examples 1.5.2, 1.6.2, and 1.7.1, recursive query plans are necessary in order to have maximally-contained query plans in the presence of functional dependencies, limitations on binding patterns, and recursive user queries. We show how maximally-contained recursive query plans can be constructed in all three cases.

The source descriptions that we consider in Chapter 2 are restricted to be conjunctive. In Chapter 3 we extend the ideas introduced in Chapter 2 to sources that contain disjunctive data.

Chapter 4 examines the effect of extending the languages used for describing information sources and allowed in user queries on the complexity required for maximally-contained query plans. The results show that relatively small extensions of these languages require query plans with more than polynomial (assuming $P \neq NP$) data complexity. As a consequence we can conclude that datalog query plans are not powerful enough in these cases.

Chapter 5 considers the problem of optimizing query plans. Two query plans that produce identical results might differ drastically in the number of resources they require. Clearly, query plans that require fewer resources but produce the same results are preferable. Usually in information integration it is assumed that data provided by information sources is incomplete. The chapter examines how knowledge of partial completeness of sources can be used for query optimization.

In Chapter 6 we present the Infomaster system as an example of an information integration system.

Some of the material presented in this thesis appears in previous conference publications. The material in Sections 2.3, 2.7 and 2.8, is covered in [17]. Sections 2.4 and 2.6 are presented in [20]. Chapter 5 is covered in [16]. Finally, the material in Chapter 6 appears in [18].

Chapter 2

Recursive Query Plans

Generating query-answering plans for information integration systems requires to translate a user query, formulated in terms of world relations, to a query that uses relations that are actually stored in information sources. Previous solutions to the translation problem produced unions of *conjunctive plans*, and were therefore limited in their ability to handle recursive queries and to exploit information sources with binding-pattern limitations and functional dependencies that are known to hold in the world schema. As a result, these plans were incomplete w.r.t. sources encountered in practice (i.e., produced only a subset of the possible answers). We describe the novel class of *recursive* query answering plans, which enables us to settle three open problems. First, we describe an algorithm for finding a query plan that produces the maximal set of answers from the sources for arbitrary recursive queries. Second, we extend this algorithm to use the presence of full generalized dependencies in the world schema. Third, we describe an algorithm for finding the maximal query plan in the presence of binding-pattern restrictions in the sources. In all three cases, recursive plans are necessary in order to obtain a maximal query plan.

2.1 Introduction

In this chapter we consider several important extensions of the problem of finding a maximally-contained plan for a query using a set of information sources. In all of these extensions we show that it is not possible to find a maximally-contained plan if we restrict ourselves to nonrecursive plans. Hence we introduce a new class of *recursive* query plans and show the following results:

- We describe an algorithm for finding a maximally-contained plan for cases in which the user query is recursive. We show that the problem of finding an equivalent plan in this case is undecidable.
- We describe an algorithm for finding a maximally-contained plan when full generalized dependencies are present in the world schema. Full generalized dependencies are a large class of dependencies, including functional dependencies, for example. The presence of dependencies

further complicates the rewriting problem, because it allows rewritings that are not valid otherwise. Furthermore, we show that in this context there does not always exist a nonrecursive maximally-contained query plan.

- In practice, many information sources have limitations on the ways they can be accessed. For example, a name server of an institution, holding the addresses of its employees, will not provide the list of all employees and their addresses. Instead, it will provide the address for a *given* name. We extend our algorithms to the case in which there are limitations on sources, and they are described by the set of allowed binding patterns. In this case it is known that recursive plans may be necessary [34]. We describe an algorithm that constructs a recursive maximally-contained query plan.

Another significant advantage of our algorithms is that they are *generative*, rather than *search-based*. Our algorithms generate the rewriting in time that is polynomial in the size of the query. In contrast, previous methods [37,44] search the space of possible candidate rewritings, and propose heuristics for reducing the size of this space [34,38].[1] These methods combine the process of finding a rewriting with the process of checking whether it is equivalent to the original query (which is NP-hard). In contrast, our method isolates the process of generating the maximally-contained rewriting, which can be done much more efficiently.

2.1.1 Organization of chapter

This chapter is organized as follows. Section 2.2 explains the basic terms we use in the discussion. Section 2.3 describes the construction of *inverse rules*, which is the basis for all the algorithms we describe in this chapter. This section also shows that the construction of the inverse rules suffices in order to compute maximally-contained query plans for recursive queries. Section 2.4 describes the extension of the algorithm in the presence of functional dependencies. The algorithm is extended further to handle the presence of full generalized dependencies in Section 2.5. Section 2.6 describes the algorithm for the case of limitations on binding patterns. The inverse rules described in Section 2.3 use a set of function symbols. In Section 2.7 we show how these function symbols can be removed, to obtain query plans that are datalog queries.

2.2 Preliminaries

2.2.1 Relations and queries

We model the world schema and the information sources by sets of relations. For every relation, we associate an *attribute name* to each of its arguments. For example, the attribute names of the

[1] The algorithm in [38] checks whether the plans can be executed given the binding-pattern restrictions, but is not guaranteed to produce the maximally-contained rewriting when these restrictions are present. The algorithm in [34] produces only conjunctive plans that are guaranteed to adhere to the limitations on binding patterns, but is not guaranteed to compute the maximally-contained plan.

binary relation *author* may be *Paper* and *Person*. For a tuple t of a relation r with attribute A, we denote by $t[A]$ the value of the attribute A in t.

We consider datalog queries over sets of relations. A datalog query is a is set of function-free Horn rules of the form

$$p(\bar{X}) :- \ p_1(\bar{X}_1), \ \ldots, \ p_n(\bar{X}_n)$$

where p, and $p_1, \ldots, p_n$ are predicate names, and $\bar{X}$, $\bar{X}_1$, ..., $\bar{X}_n$ are tuples of variables or constants. The *head* of the rule is $p(\bar{X})$, and its *body* is $p_1(\bar{X}_1)$, ..., $p_n(\bar{X}_n)$. Each $p_i(\bar{X}_i)$ is a *subgoal* of the rule. We require that the rules be safe, i.e., every variable in the head of a rule must also occur in the body of the rule. A predicate is an *intensional database predicate*, or *IDB predicate*, in a query $\mathcal{Q}$ if it appears as the head of some rule in $\mathcal{Q}$. Predicates not appearing in any head are *extensional database predicates*, or *EDB predicates*. We assume that every query has an IDB predicate q, called the query predicate, that represents the result of $\mathcal{Q}$.

The input of a datalog query $\mathcal{Q}$ consists of a database D storing extensions of all EDB predicates in $\mathcal{Q}$. Given such a database D, a bottom-up evaluation is one in which we start with the ground EDB facts in D and apply the rules to derive facts for the IDB predicates. The output of $\mathcal{Q}$, denoted $\mathcal{Q}(D)$, is the set of ground facts generated for the query predicate in the bottom-up evaluation.

As an intermediate result of our algorithms, we will construct datalog programs with function symbols. That is, some of the arguments in the bodies or the heads of the rules are functional terms. When datalog queries contain function symbols we will refer to them as *logic queries*. In general, the bottom-up evaluation of a logic query may not terminate. As it turns out, we introduce function symbols in a controlled fashion, and in particular, the evaluation of our logic queries is guaranteed to terminate. Furthermore, we show in Section 2.7 how to remove the function symbols.

Given a query, we can define a *dependency graph*, whose nodes are the predicate names appearing in the rules. There is an edge from the node of predicate p_i to the node of predicate p if p_i appears in the body of a rule whose head predicate is p. The query is *recursive* if there is a cycle in the dependency graph. A *conjunctive query* is a single nonrecursive function-free Horn rule. A recursive datalog query can be seen as a finite encoding of a potentially infinite set of conjunctive queries.

Example 2.2.1 Consider the following datalog query:

$$\begin{array}{ll} \mathcal{Q}: & q(X,Y) :- \ edge(X,Z), \ edge(Z,Y), \ black(Z) \\ & q(X,Y) :- \ edge(X,Z), \ black(Z), \ q(Z,Y) \end{array}$$

Predicates *edge* and *black* are EDB predicates, and predicate q is an IDB predicate. The query is recursive because its dependency graph has a self-loop on predicate q. Datalog query $\mathcal{Q}$ encodes the following infinite set of conjunctive queries:

$$q(X,Y) :- \ edge(X,Z_1), \ edge(Z_1,Y), \ black(Z_1)$$

$$\begin{array}{l} q(X,Y) :- \ edge(X,Z_1), \ edge(Z_1,Z_2), \ edge(Z_2,Y), \\ \qquad\qquad black(Z_1), \ black(Z_2) \end{array}$$

$$q(X,Y) :- \ edge(X,Z_1), \ edge(Z_1,Z_2), \ edge(Z_2,Z_3), \ edge(Z_3,Y),$$

$$black(Z_1),\ black(Z_2),\ black(Z_3)$$
$$\vdots$$

□

2.2.2 Containment

A datalog query $\mathcal{Q}'$ is *contained* in a datalog query $\mathcal{Q}$ if, for all databases D, $\mathcal{Q}'(D)$ is a subset of $\mathcal{Q}(D)$. Datalog queries $\mathcal{Q}'$ and $\mathcal{Q}$ are *equivalent* if $\mathcal{Q}'$ and $\mathcal{Q}$ are contained in one another. The problem of determining whether a datalog query $\mathcal{Q}'$ is contained in a datalog query $\mathcal{Q}$ is in general undecidable [47]. The problem remains decidable if either $\mathcal{Q}'$ or $\mathcal{Q}$ are nonrecursive [45,12]. In our discussion we use the following algorithm from [45] to test when a union of conjunctive queries $\mathcal{Q}'$ is contained in a recursive query $\mathcal{Q}$.[2] First, replace all variables in $\mathcal{Q}'$ by distinct constants. Consider the database D_c that contains exactly the tuples corresponding to the subgoals in the "frozen" bodies of the rules in $\mathcal{Q}'$. D_c is called the *canonical database* of $\mathcal{Q}'$. Evaluate $\mathcal{Q}$ on the canonical database. $\mathcal{Q}'$ is contained in $\mathcal{Q}$ if and only if the "frozen" heads of the rules in $\mathcal{Q}'$ are contained in $\mathcal{Q}(D_c)$.

Example 2.2.2 Let $\mathcal{Q}$ be the following datalog query:

$$\begin{array}{ll} \mathcal{Q}: & q(X,Y) :- \ edge(X,Z),\ edge(Z,Y),\ black(Z) \\ & q(X,Y) :- \ edge(X,Z),\ black(Z),\ q(Z,Y) \end{array}$$

To determine whether the nonrecursive datalog query

$$\begin{array}{ll} \mathcal{Q}': & q(X,Y) :- \ edge(X,Z),\ edge(Z,Y),\ black(X),\ black(Z) \\ & q(X,Y) :- \ edge(X,V),\ edge(V,W),\ edge(W,Y),\ black(V),\ black(W) \end{array}$$

is contained in $\mathcal{Q}$, we replace the variables in the two rules by distinct constants:

$$q(c_1,c_3) :- \ edge(c_1,c_2),\ edge(c_2,c_3),\ black(c_1),\ black(c_2)$$
$$q(c_4,c_7) :- \ edge(c_4,c_5),\ edge(c_5,c_6),\ edge(c_6,c_7),\ black(c_5),\ black(c_6)$$

The following is the corresponding canonical database:

$\underline{edge}$
$\langle c_1,c_2\rangle,\ \langle c_2,c_3\rangle,\ \langle c_4,c_5\rangle,\ \langle c_5,c_6\rangle,\ \langle c_6,c_7\rangle$
$\underline{black}$
$\langle c_1\rangle,\ \langle c_2\rangle,\ \langle c_5\rangle,\ \langle c_6\rangle$

The output of datalog query $\mathcal{Q}$ on the canonical database is $\langle c_1,c_3\rangle$, $\langle c_4,c_6\rangle$, $\langle c_5,c_7\rangle$, and $\langle c_4,c_7\rangle$. Because this output contains $\langle c_1,c_3\rangle$ and $\langle c_4,c_7\rangle$, $\mathcal{Q}'$ is contained in $\mathcal{Q}$. □

[2] Recall that every nonrecursive datalog program can be translated into an equivalent union of conjunctive queries.

2.2.3 Functional dependencies

An instance of a relation p satisfies the *functional dependency* $A_1, \ldots, A_n \rightarrow B$ if for every two tuples t and u in p with $t.A_i = u.A_i$ for $i = 1, \ldots, n$, also $t.B = u.B$. We will abbreviate a set of attributes $A_1, \ldots, A_n$ by $\bar{A}$.

When the relations satisfy a set of functional dependencies Σ, we refine our notion of containment to *relative containment*: Query $\mathcal{Q}'$ is *contained* in query $\mathcal{Q}$ *relative to* Σ, denoted $\mathcal{Q}' \subseteq_\Sigma \mathcal{Q}$, if for each database D satisfying the functional dependencies in Σ, $\mathcal{Q}'(D) \subseteq \mathcal{Q}(D)$.

In order to decide containment of conjunctive queries in the presence of functional dependencies, Aho et al. [2] show that it suffices to precede the containment algorithm by applying the *chase* algorithm to the contained query. A step in applying the chase to the body of a conjunctive query $\mathcal{Q}$ is the following. If the functional dependency $\bar{A} \rightarrow B$ holds for a relation p, and a conjunctive query $\mathcal{Q}$ has two subgoals of p, g_1 and g_2, with the same variables or values for the attributes $\bar{A}$, and g_1 has a variable X for attribute B, then we replace the occurrences of X in $\mathcal{Q}$ by the value or variable for B in g_2.

2.2.4 Full generalized dependencies

Functional dependencies are a special form of a more general kind of dependencies, called full generalized dependencies[3]. A *full generalized dependency* δ is a first-order formula of the form

$$\forall \bar{X}\ [\ \phi(\bar{X}) \Rightarrow \psi(\bar{Y})\ ]$$

where $\phi(\bar{X})$ is a conjunction of relations and equality assertions with variables $\bar{X}$, $\psi(\bar{Y})$ is a relation or an equality assertion with variables $\bar{Y}$, and $\bar{Y} \subseteq \bar{X}$. If ψ is an equality assertion, then δ is called an *equality generating dependency*. If ψ is a relation, then δ is called a *tuple generating dependency*. In examples, we will omit the universal quantification for the sake of brevity. A functional dependency $A \rightarrow B$ of relation $p(A, B, C)$ is an equality generating dependency because it can be written in the form

$$\forall X\, \forall Y\, \forall Z\, \forall Y'\, \forall Z'\ [\ p(X, Y, Z) \wedge p(X, Y'Z') \Rightarrow Y = Y'\ ].$$

Query $\mathcal{Q}'$ is *contained* in query $\mathcal{Q}$ *relative to* a set of full generalized dependencies Δ, denoted $\mathcal{Q}' \subseteq_\Delta \mathcal{Q}$, if for each database D satisfying the full generalized dependencies in Δ, $\mathcal{Q}'(D) \subseteq \mathcal{Q}(D)$.

2.2.5 Information sources and query plans

The schema of an information integration system includes a set of *virtual* relations. The relations in the mediator are virtual because their extensions are not actually stored. Their role is to provide the user a uniform interface to a multitude of information sources. We refer to the schema of the mediator as the *world schema*, and to the relations in the world schema as *world relations*. The

[3]Full generalized dependencies include also two other well-known dependencies, namely multivalued dependencies and join dependencies.

actual data is stored in a set of external information sources. We model each source by containing the extension of a *source relation.* The set of source relations is disjoint from the set of world relations.

To answer user queries, the mediator must also have a mapping between the global and source relations. We follow the approach taken in [38,34,19], where the mappings (a.k.a. source descriptions) are specified by a set of conjunctive queries, one for every source relation. The predicates in the heads of the conjunctive queries are source relations, and the predicates in their bodies are world relations. The meaning of such a mapping is that all the tuples that are found in the information source satisfy the query over the world relations. Several authors have distinguished the case in which the source contains *all* the tuples that satisfy the query from the case in which some tuples may be missing from the source [22,23,16,36]. We will examine this distinction in detail in Chapters 4 and 5. For the discussion in this chapter this distinction does not matter.

Example 2.2.3 Consider a world schema that includes the relations $parent$, $male$ and $female$. The source descriptions below say that the source relations v_1, v_2, and v_3 store the father, the daughter, and the grandmother relation, respectively.

$$
\begin{aligned}
&v_1(X,Y) :- \ parent(X,Y),\ male(X)\\
&v_2(X,Y) :- \ parent(Y,X),\ female(X)\\
&v_3(X,Y) :- \ parent(X,Z),\ parent(Z,Y),\ female(X)
\end{aligned}
$$

□

Given a query $\mathcal{Q}$ from the user, the mediator needs to formulate a *query plan,* which is a query that bottoms out in the source relations and produces answers to $\mathcal{Q}$. A query plan is a set of Horn rules whose EDB predicates include *only* the source relations. The *expansion* $\mathcal{P}^{exp}$ of a query plan $\mathcal{P}$ is obtained from $\mathcal{P}$ by replacing all source relations with their corresponding source descriptions. Existentially quantified variables in view definitions are replaced by new variables in the expansion.

Example 2.2.4 The following query plan determines all grandparents of ann from the source described in Example 2.2.3:

$$
\begin{aligned}
&p(X,Y) &&:- \ v_1(X,Y)\\
&p(X,Y) &&:- \ v_2(Y,X)\\
&q(X) &&:- \ p(X,Z),\ p(Z,ann)\\
&q(X) &&:- \ v_3(X,ann)
\end{aligned}
$$

The expansion of this query plan is the following datalog query:

$$
\begin{aligned}
&p(X,Y) &&:- \ parent(X,Y),\ male(X)\\
&p(X,Y) &&:- \ parent(Y,X),\ female(X)\\
&q(X) &&:- \ p(X,Z),\ p(Z,ann)\\
&q(X) &&:- \ parent(X,Z),\ parent(Z,Y),\ female(X)
\end{aligned}
$$

□

2.2.6 Equivalent vs. maximally-contained query plans

A query plan $\mathcal{P}$ is contained in a datalog query $\mathcal{Q}$ if $\mathcal{P}^{exp}$ is contained in $\mathcal{Q}$, and is equivalent to $\mathcal{Q}$ if $\mathcal{P}^{exp}$ is equivalent to $\mathcal{Q}$. A query plan $\mathcal{P}$ is *maximally-contained* in a datalog query $\mathcal{Q}$ if $\mathcal{P}$ is contained in $\mathcal{Q}$, and for every query plan $\mathcal{P}'$ that is contained in $\mathcal{Q}$, $\mathcal{P}'$ is already contained in $\mathcal{P}$. Containment and maximal containment relative to a set of functional dependencies Σ or relative to a set of full generalized dependencies Δ is defined accordingly. Note that the notion of maximal containment is relative to a fixed set of source relations.

Ideally, the mediator would try to find a query plan that is equivalent to the user query. However, in practice we may not have sufficient information sources to completely answer the user query. Hence, the mediator tries to find the maximally-contained query plan. In a sense, the maximally-contained query plan produces *all* the answers to the query that could be retrieved from the available sources. Of course, if there exists a plan that is equivalent to the user query then it will be a maximally-contained plan.

In this chapter we focus on finding maximaly-contained plans. As it turns out, in the cases we consider in this chapter, the maximally-contained query plan may have to be a recursive datalog program. Furthermore, we show that if the query $\mathcal{Q}$ is recursive, then finding an equivalent query plan is undecidable, while finding a maximally-contained query plan is decidable.

2.3 Inverse rules and recursive queries

In this section we first describe how to compute a set of *inverse rules* from a given set of source descriptions. Intuitively, inverse rules can be viewed as query plans for the world relations. Inverse rules are common to all the constructions we describe in this chapter. We then show that the inverse rules themselves, together with a recursive datalog query $\mathcal{Q}$ provide a maximally-contained plan for $\mathcal{Q}$. It should be noted that previous work considered the construction of query plans given source descriptions for nonrecursive datalog user queries only. Finally, we show that the problem of finding an equivalent query plan for recursive queries is undecidable.

As explained below, in constructing the inverse rules we use function symbols. These function symbols can later be eliminated, as we will show in Section 2.7. We use the following set of function symbols in inverse rules. For every source relation v with a variable Z_i in the body but not in the head of its source description, we have a function symbol $f_{v,i}$.

Definition 2.3.1 (**inverse rules**) Let v be a source relation defined by the view definition

$$v(\bar{X}) :- p_1(\bar{X}_1),\ \ldots,\ p_n(\bar{X}_n).$$

Then for $j = 1, \ldots, n$,

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

is an *inverse rule* of v, denoted v^{-1}. We modify $\bar{X}_j$ to obtain the tuple $\bar{X}'_j$ as follows: if X is a constant or is a variable in $\bar{X}$, then X is unchanged in X'_j. Otherwise, X is one of the variables Z_i appearing in the body of v but not in $\bar{X}$, and X is replaced by $f_{v,i}(\bar{X})$ in $\bar{X}'_j$. □

We denote the set of inverse rules of the view definitions in $\mathcal{V}$ by $\mathcal{V}^{-1}$.

Example 2.3.1 The inverse of the view definitions

$$\begin{array}{ll} v_1(X,Y) & :- \ edge(X,Z_1),\ edge(Z_1,Z_2),\ edge(Z_2,Y) \\ v_2(X) & :- \ edge(X,Z_1) \end{array}$$

is the following set of rules:

$$\begin{array}{ll} edge(X, f_{v_1,1}(X,Y)) & :- \ v_1(X,Y) \\ edge(f_{v_1,1}(X,Y), f_{v_1,2}(X,Y)) & :- \ v_1(X,Y) \\ edge(f_{v_1,2}(X,Y), Y) & :- \ v_1(X,Y) \\ edge(X, f_{v_2,1}(X)) & :- \ v_2(X) \end{array}$$

□

In the following we will abbreviate the function symbols $f_{v,i}$ by function symbols like f, g, h, f_1, f_2, etc.

Given a datalog query $\mathcal{Q}$ and a set of conjunctive source descriptions $\mathcal{V}$, the construction of the query plan is quite simple. We delete all rules from $\mathcal{Q}$ that contain world relations that do not appear in any of the source descriptions. To the resulting query, denoted $\mathcal{Q}^-$, we add the rules of $\mathcal{V}^{-1}$, and call the query so obtained $(\mathcal{Q}^-, \mathcal{V}^{-1})$. Notice that the EDB predicates of the remaining rules of $\mathcal{Q}$ are IDB predicates in $(\mathcal{Q}^-, \mathcal{V}^{-1})$, because they appear in heads of the rules in $\mathcal{V}^{-1}$. Because naming of IDB predicates is arbitrary, one could rename the IDB predicates in $(\mathcal{Q}^-, \mathcal{V}^{-1})$ so that their names differ from the names of the corresponding EDB predicates in $\mathcal{Q}$. For ease of exposition, we will not do it here.

Example 2.3.2 Consider the recursive query

$$\mathcal{Q}: \quad \begin{array}{l} q(X,Y) :- \ edge(X,Y) \\ q(X,Y) :- \ edge(X,Z),\ q(Z,Y) \end{array}$$

which determines the transitive closure of the relation *edge*. Assume there is only one information source available:

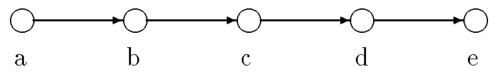
$$v(X,Y) :- \ edge(X,Z),\ edge(Z,Y)$$

View v stores endpoints of paths of length two. Just using this view, there is no way to determine the transitive closure of the relation *edge*. The best one can hope to achieve is to compute the endpoints of paths of *even* lengths. Relation *edge*, the only EDB predicate in $\mathcal{Q}$, appears in the definition of v. Therefore, $(\mathcal{Q}^-, \mathcal{V}^{-1})$ is just $\mathcal{Q}$ with the rules of v^{-1} added:

$$(\mathcal{Q}^-, \mathcal{V}^{-1}): \quad \begin{array}{ll} q(X,Y) & :- \ edge(X,Y) \\ q(X,Y) & :- \ edge(X,Z),\ q(Z,Y) \\ edge(X, f(X,Y)) & :- \ v(X,Y) \\ edge(f(X,Y), Y) & :- \ v(X,Y) \end{array}$$

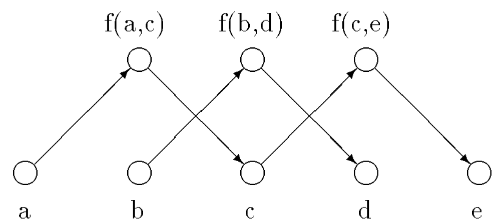
$(\mathcal{Q}^-, \mathcal{V}^{-1})$ indeed yields all endpoints of paths of even length in its result. For example, assume that an instance of the EDB predicate *edge* in $\mathcal{Q}$ represents the following graph:

$G:$



$(\mathcal{Q}^-, \mathcal{V}^{-1})$ introduces three new constants, named $f(a,c)$, $f(b,d)$, and $f(c,e)$. The IDB predicate *edge* in $\mathcal{V}^{-1}$ represents the following graph:

$G':$



$\mathcal{Q}^-$ computes the transitive closure of G'. Notice that the pairs in the transitive closure of G' that do not contain any of the new constants are exactly the endpoints of paths of even length in the original graph G. □

The query $(\mathcal{Q}^-, \mathcal{V}^{-1})$ is a logic query because the inverse rules contain function symbols. In order to show that it is the maximally-contained plan of of $\mathcal{Q}$, we first show that the evaluation of $(\mathcal{Q}^-, \mathcal{V}^{-1})$ will terminate on every database.

Bottom-up evaluation of logic queries is not guaranteed to terminate in general. It might be possible to generate terms with arbitrarily deeply nested function symbols. For example, bottom-up evaluation of the logic query

$$\begin{array}{ll} q(X) & :- \; p(X) \\ q(f(X)) & :- \; q(X) \end{array}$$

contains the infinite number of terms a, $f(a)$, $f(f(a))$, $\ldots$ in its answer, if the EDB predicate p contains the constant a.

In contrast, our construction produces logic queries whose bottom-up evaluation *is* guaranteed to terminate. The key observation is that function symbols are *only* introduced in inverse rules. Because inverse rules are not recursive, no terms with nested function symbols can be generated.

Lemma 2.3.1 *For every datalog query $\mathcal{Q}$, every set of conjunctive source descriptions $\mathcal{V}$, and all finite instances of the source relations, the logic query $(\mathcal{Q}^-, \mathcal{V}^{-1})$ has a unique finite minimal fixpoint. Therefore, bottom-up evaluation is guaranteed to terminate, and produces this unique fixpoint.*

Proof. $\mathcal{Q}^-$ is recursive, but does not introduce function symbols. On the other hand, $\mathcal{V}^{-1}$ introduces function symbols, but is not recursive. Moreover, the IDB predicates of $\mathcal{V}^{-1}$ depend only on the EDB predicates. Therefore, every bottom-up evaluation of $(\mathcal{Q}^-, \mathcal{V}^{-1})$ will necessarily progress in two stages. In the first stage, the extensions of the IDB predicates in $\mathcal{V}^{-1}$ are determined. The second stage will then be a standard datalog evaluation of $\mathcal{Q}^{-}$. Because datalog queries have unique finite minimal fixpoints, this proves the claim. $\square$

Given extensions for its EDB predicates, a logic query might produce tuples containing function symbols in its result. Because the extensions of EDB predicates do not contain any function symbols, no datalog query produces tuples in its result containing function symbols. Hence, in order to compare between the result of evaluating $\mathcal{Q}$ to that of evaluating $(\mathcal{Q}^{-}, \mathcal{V}^{-1})$ on a set of information sources, we need to define a filter that gets rid of all extraneous tuples with functional terms. If $\mathcal{D}$ is a set of sources containing tuples of the EDB predicates of a query plan with function symbols $\mathcal{P}$, then let $\mathcal{P}(\mathcal{D})\downarrow$ be the set of all tuples in $\mathcal{P}(\mathcal{D})$ that do not contain function symbols. Let $\mathcal{P}\downarrow$ be the plan that given the sources $\mathcal{D}$ computes $\mathcal{P}(\mathcal{D})\downarrow$.

The following theorem shows that the simple construction of adding the inverse rules to $\mathcal{Q}$ yields a logic query that uses the source relations in the best possible way. That is, after discarding all tuples containing function symbols, the result of $(\mathcal{Q}^{-}, \mathcal{V}^{-1})$ is contained in $\mathcal{Q}$. Moreover, the result of every query plan that is contained in $\mathcal{Q}$ is already contained in $(\mathcal{Q}^{-}, \mathcal{V}^{-1})$.

Theorem 2.3.1 *For every datalog query $\mathcal{Q}$ and every set of conjunctive source descriptions $\mathcal{V}$, the query plan $(\mathcal{Q}^{-}, \mathcal{V}^{-1})\downarrow$ is maximally-contained in $\mathcal{Q}$. Moreover, $(\mathcal{Q}^{-}, \mathcal{V}^{-1})$ can be constructed in time polynomial in the size of $\mathcal{Q}$ and $\mathcal{V}$.*

Proof. First we prove that $(\mathcal{Q}^{-}, \mathcal{V}^{-1})\downarrow$ is contained in $\mathcal{Q}$. Let $E_1, \ldots, E_n$ be instances of the EDB predicates in $\mathcal{Q}$. $E_1, \ldots, E_m$ determine the instances of the source relations in $\mathcal{V}$ which in turn are the EDB predicates of $(\mathcal{Q}^{-}, \mathcal{V}^{-1})$. Assume that $(\mathcal{Q}^{-}, \mathcal{V}^{-1})$ produces a tuple t that does not contain any function symbols. Consider the derivation tree of t in $(\mathcal{Q}^{-}, \mathcal{V}^{-1})$. All the leaves are source relations because source relations are the only EDB predicates of $(\mathcal{Q}^{-}, \mathcal{V}^{-1})$. Removing all leaves from this tree produces a tree with the original EDB predicates from $\mathcal{Q}$ as new leaves. Because the instances of the source relations are derived from $E_1, \ldots, E_n$, there are constants in $E_1, \ldots, E_n$ such that consistently replacing function terms with these constants yields a derivation tree of t in $\mathcal{Q}$. Therefore, $(\mathcal{Q}^{-}, \mathcal{V}^{-1})\downarrow$ is contained in $\mathcal{Q}$.

Let $\mathcal{P}$ be an arbitrary query plan contained in $\mathcal{Q}$. We have to prove that $\mathcal{P}$ is also contained in $(\mathcal{Q}^{-}, \mathcal{V}^{-1})$. Let c_v be an arbitrary conjunctive query generated by $\mathcal{P}$. If we can prove that c_v is contained in $(\mathcal{Q}^{-}, \mathcal{V}^{-1})$, then $\mathcal{P}$ is contained in $(\mathcal{Q}^{-}, \mathcal{V}^{-1})$, which proves the claim. Let D_c be the canonical source of c_v^{exp}. Because c_v^{exp} is contained in $\mathcal{Q}$, $c_v^{exp}(D_c)$ is contained in the output of $\mathcal{Q}$ applied to D_c. Let c be the conjunctive query generated by $\mathcal{Q}$ that produces $c_v^{exp}(D_c)$. Because all predicates of query c are also in c_v^{exp}, and all predicates in c_v^{exp} appear in some view definition, c is also generated by $\mathcal{Q}^{-}$. Because c_v^{exp} is contained in c, there is a containment mapping h from c to c_v^{exp} [6]. Every variable Z in c_v^{exp} that does not appear in c_v is existentially quantified in some view definition $v_i(X_1, \ldots, X_m)$ in c_v. Let k be the mapping that maps every such variable Z to the corresponding term $f(X_1, \ldots, X_m)$ used in v_i^{-1} in the expansion of c_v. Because $\mathcal{Q}^{-}$ can derive c, $\mathcal{Q}^{-}$ can also derive the more specialized conjunctive query $k(h(c))$. Using rules in $\mathcal{V}^{-1}$, the derivation of $k(h(c))$ in $\mathcal{Q}^{-}$ can be extended to a derivation of a conjunctive query c' that contains only source relations. The identity mapping is a containment mapping from c' to c_v. Therefore, $\mathcal{P}$ is contained in $(\mathcal{Q}^-, \mathcal{V}^{-1})$.

$(\mathcal{Q}^-, \mathcal{V}^{-1})$ can be constructed in time polynomial in the size of $\mathcal{Q}$ and $\mathcal{V}$, because every subgoal in a view definition in $\mathcal{V}$ corresponds to exactly one inverse rule in $\mathcal{V}^{-1}$. □

As stated earlier, if there exists an equivalent plan for a query $\mathcal{Q}$, it will be a maximally-contained plan. However, since equivalence of datalog programs is undecidable in general, we cannot test whether $(\mathcal{Q}^-, \mathcal{V}^{-1})$ is an equivalent plan by testing whether it is equivalent to $\mathcal{Q}$. Moreover, the following theorem shows that the problem of whether there exists a query plan equivalent to $\mathcal{Q}$ is undecidable.

Theorem 2.3.2 *Given a datalog query $\mathcal{Q}$ and conjunctive view definitions, it is undecidable whether there is a query plan $\mathcal{P}$ equivalent to $\mathcal{Q}$.*

Proof. Let $\mathcal{Q}_1$ and $\mathcal{Q}_2$ be two arbitrary datalog queries. We show that a decision procedure for the above problem would allow us to decide whether $\mathcal{Q}_1$ is contained in $\mathcal{Q}_2$. Because the containment problem for datalog queries is undecidable, this proves the claim. Without loss of generality we can assume that there are no IDB predicates with the same name in $\mathcal{Q}_1$ and $\mathcal{Q}_2$, and that the query predicates in $\mathcal{Q}_1$ and $\mathcal{Q}_2$, named q_1 and q_2 respectively, have arity m. Let $\mathcal{Q}$ be the datalog query consisting of all the rules in $\mathcal{Q}_1$ and $\mathcal{Q}_2$, and of the rules

$$q(X_1, \ldots, X_m) :- \; q_1(X_1, \ldots, X_m), \; e()$$
$$q(X_1, \ldots, X_m) :- \; q_2(X_1, \ldots, X_m)$$

where e is a new zero-ary global relation. For every global relation $e_i(X_1, \ldots, X_{k_i})$ in $\mathcal{Q}_1$ and $\mathcal{Q}_2$ (but not for e) assume there is a source relation described by the view definition

$$v_i(X_1, \ldots, X_{k_i}) :- \; e_i(X_1 \ldots, X_{k_i}).$$

We show that $\mathcal{Q}_1$ is contained in $\mathcal{Q}_2$ if and only if there is a query plan $\mathcal{P}$ equivalent to $\mathcal{Q}$.

$'' \Rightarrow''$: Assume $\mathcal{Q}_1$ is contained in $\mathcal{Q}_2$. Then $\mathcal{Q}$ is equivalent to the query plan $\mathcal{P}$ consisting of all the rules of $\mathcal{Q}_2$ with e_i's replaced by the corresponding v_i's, and the additional rule

$$q(X_1, \ldots, X_m) :- \; q_2(X_1, \ldots, X_m).$$

$'' \Leftarrow''$: Assume there is a query plan $\mathcal{P}$ equivalent to $\mathcal{Q}$. Then for any instantiation of the global relations, $\mathcal{Q}$ and $\mathcal{P}^{exp}$ yield the same result, especially for instantiations where e is the empty relation, and where e contains the empty tuple. If e is the empty relation then $\mathcal{Q}$ produces exactly the tuples produced by $\mathcal{Q}_2$, and therefore $\mathcal{P}^{exp}$ does likewise. If e contains the empty tuple then $\mathcal{Q}$ produces the union of the tuples produced by $\mathcal{Q}_1$ and $\mathcal{Q}_2$, and hence $\mathcal{P}^{exp}$ produces this union. No view definition contains relation e. Therefore $\mathcal{P}^{exp}$ does not contain relation e. It follows that $\mathcal{P}^{exp}$ will produce the same set of tuples regardless of the instantiation of e. It follows that $\mathcal{Q}_2$ is equivalent to the union of $\mathcal{Q}_1$ and $\mathcal{Q}_2$. Therefore, $\mathcal{Q}_1$ is contained in $\mathcal{Q}_2$. □

2.4 Functional dependencies

In this section we consider the problem of generating a maximally-contained plan for a query $\mathcal{Q}$ in the presence of functional dependencies in the world schema. We begin by describing an algorithm for generating a maximally-contained plan, and in the end of the section we show recursive plans may be *necessary* in this context.

We use the following example throughout this section to illustrate the difficulties introduced by functional dependencies and to present our algorithm. Suppose we have the following world relations

conference(Paper, Conference),
year(Paper, Year),
location(Conference, Year, Location)

The relations describe the conference at which a paper was presented, the publication year of a paper, and the location a conference was held at in a given year. A paper is only presented at one conference and published in one year. Also, in a given year a conference is held at a specific location. Therefore we have three functional dependencies:

$$\begin{array}{ll} \textit{conference}: & \textit{Paper} \rightarrow \textit{Conference} \\ \textit{year}: & \textit{Paper} \rightarrow \textit{Year} \\ \textit{location}: & \textit{Conference},\ \ \textit{Year} \rightarrow \textit{Location} \end{array}$$

Suppose we have the following information sources:

$$\begin{array}{ll} v_1(P, C, Y) & :-\ \textit{conference}(P, C),\ \textit{year}(P, Y) \\ v_2(P, L) & :-\ \textit{conference}(P, C),\ \textit{year}(P, Y),\ \textit{location}(C, Y, L) \end{array}$$

v_1 tells us in which conference and year a paper was presented, and v_2 stores the location of the presentation of a paper directly with the paper. Assume a user wants to know where PODS '89 was held:

$$q(L) :-\ \textit{location}(\textit{pods}, 1989, L)$$

The following plan would answer the query:

$$q(L) :-\ v_1(P, \textit{pods}, 1989),\ v_2(P, L)$$

Informally, the query plan proceeds as follows. It first finds *some* paper presented at PODS '89 using v_1 and then finds the location of the conference this paper was presented at using v_2. This plan is correct *only* because every paper is presented at one conference and in one year. In fact, if these dependencies were not to hold, there would be no way of answering this query using the sources. It is also important to note that source relation v_1 is needed in the query plan, even though the predicates in v_1, *conference* and *year*, don't appear in the query $\mathcal{Q}$ at all. Without functional dependencies, only source descriptions that contain predicates appearing in the user query need to be considered [37].

In the following we are going to give a construction of query plans that is guaranteed to be maximally-contained in the given queries, even in the presence of functional dependencies. As in the previous section, we begin by computing the set of inverse rules, whose purpose is to recover tuples of the world relations from the source relations. The inverse rules for v_1 and v_2 in our example are:

$$\begin{array}{lll}
r_1: & conference(P,C) & :-\ v_1(P,C,Y) \\
r_2: & year(P,Y) & :-\ v_1(P,C,Y) \\
r_3: & conference(P, f_1(P,L)) & :-\ v_2(P,L) \\
r_4: & year(P, f_2(P,L)) & :-\ v_2(P,L) \\
r_5: & location(f_1(P,L), f_2(P,L), L) & :-\ v_2(P,L)
\end{array}$$

For example, rule r_5 extracts from v_2 that *some* conference in *some* year was held in location L. Suppose that v_1 stores the information that the paper "Bottom-Up Beats Top-Down for Datalog" (abbreviated as *datalog*) was presented at PODS '89, and v_2 stores the information that "Bottom-Up Beats Top-Down for Datalog" was presented in Philadelphia. The inverse rules derive the following facts:

$\underline{conference}$

$\langle datalog,\ pods\rangle$	(with r_1)
$\langle datalog,\ f_1(datalog, philadelphia)\rangle$	(r_3)

$\underline{year}$

$\langle datalog,\ 1989\rangle$	(r_2)
$\langle datalog,\ f_2(datalog, philadelphia)\rangle$	(r_4)

$\underline{location}$

$\langle f_1(datalog, philadelphia),\ f_2(datalog, philadelphia),\ philadelphia\rangle$	(r_5)

The inverse rules don't take into account the presence of the functional dependencies. For example, because of the functional dependency $Paper \rightarrow Conference$ in relation *conference* it is possible to conclude that function term $f_1(datalog, philadelphia)$ must actually be the same as the constant *pods*. We model this inference by introducing a new binary relation e. The intended meaning of e is that $e(c_1, c_2)$ holds if and only if c_1 and c_2 must be equal under the given functional dependencies. Hence, the extension of e includes the extension of $=$ (i.e., for every X, $e(X,X)$), and the tuples that can be derived by the following chase rules ($e(\bar{A}, \bar{A}')$ is a shorthand for $e(A_1, A_1'), \ldots, e(A_n, A_n')$):[4]

Definition 2.4.1 (chase rules) Let $\bar{A} \rightarrow B$ be a functional dependency satisfied by a world relation p. Let $\bar{C}$ be the attributes of p that are not in $\bar{A}$, B. The *chase rule* corresponding to $\bar{A} \rightarrow B$, denoted $chase(\bar{A} \rightarrow B)$, is the following rule:

$$e(B, B') :-\ p(\bar{A}, B, \bar{C}),\ p(\bar{A}', B', \bar{C}'),\ e(\bar{A}, \bar{A}').$$

[4] We only require relation e to be reflexive for ease of exposition. For every rule r having a subgoal $e(X,Y)$ in its body, we could add a modified version of rule r with subgoal $e(X,Y)$ removed and X replaced by Y. The resulting set of rules wouldn't require e to be reflexive.

□

We denote by $chase(\Sigma)$ the set of chase rules corresponding to the functional dependencies in Σ. In our example, the chase rules are

$$e(C, C') :- conference(P, C),\ conference(P', C'),\ e(P, P')$$
$$e(Y, Y') :- year(P, Y),\ year(P', Y'),\ e(P, P')$$
$$e(L, L') :- location(C, Y, L),\ location(C', Y', L'),\ e(C, C'),\ e(Y, Y')$$

The chase rules allow us to derive the following facts in relation e:

e
$\langle f_1(datalog, philadelphia),\ pods\rangle$
$\langle f_2(datalog, philadelphia),\ 1989\rangle$

The extension of e is reflexive by construction, and is symmetric because of the symmetry in the chase rules. To guarantee that e is an equivalence relation, it is still needed to enforce transitivity of e. The following rule, denoted by $\mathcal{T}$, is sufficient for guaranteeing transitivity of relation e:

$$e(X, Y) :- e(X, Z),\ e(Z, Y).$$

The final step in the construction is to rewrite query $\mathcal{Q}$ in a way that it can use the equivalences derived in relation e. We define the *rectified* query $\bar{\mathcal{Q}}$ by modifying $\mathcal{Q}$ iteratively as follows:

1. If c is a constant in one of the subgoals of $\mathcal{Q}$, we replace it by a new variable Z, and add the subgoal $e(Z, c)$.

2. If X is a variable in the head of $\mathcal{Q}$, we replace X in the body of $\mathcal{Q}$ by a new variable X', and add the subgoal $e(X', X)$.

3. If a variable Y that is not in the head of $\mathcal{Q}$ appears in two subgoals of $\mathcal{Q}$, we replace one of its occurrences by Y', and add the subgoal $e(Y', Y)$.

We apply the above steps until no additional changes can be made to the query. In our example query $\mathcal{Q}$ would be rewritten to

$$\bar{q}(L) :- location(C, Y, L'),\ e(C, pods),\ e(Y, 1989),\ e(L', L)$$

Note that evaluating query $\bar{\mathcal{Q}}$ on the reconstructed world relations and the derived equivalence relation e yields the desired result: PODS '89 was held in Philadelphia.

Given a query $\mathcal{Q}$, a set of source descriptions $\mathcal{V}$, and a set of functional dependencies Σ, the constructed query plan includes $\bar{\mathcal{Q}}$, the inverse rules $\mathcal{V}^{-1}$, the chase rules $chase(\Sigma)$ and the transitivity rule $\mathcal{T}$. The following theorem shows that this query plan is maximally-contained in $\mathcal{Q}$ relative to Σ.

Theorem 2.4.1 *Let Σ be a set of functional dependencies, $\mathcal{V}$ a set of conjunctive source descriptions, and let $\mathcal{Q}$ be a datalog query over the world relations. Let $\mathcal{R}$ denote the set of rules $\mathcal{V}^{-1} \cup chase(\Sigma) \cup \mathcal{T}$. Then, $(\bar{\mathcal{Q}}, \mathcal{R})\downarrow$ is maximally-contained in $\mathcal{Q}$ relative to Σ. Furthermore, $(\bar{\mathcal{Q}}, \mathcal{R})$ can be constructed in time polynomial in the size of $\mathcal{Q}$, $\mathcal{V}$, and Σ.* $\square$

Proof. The key to the proof is to show that for every conjunctive query plan $\mathcal{P} \subseteq_\Sigma \mathcal{Q}$, $\mathcal{P} \subseteq_\Sigma (\bar{\mathcal{Q}}, \mathcal{R})$. Because recursive query plans can be seen as an encoding of the union of infinitely many conjunctive query plans, it suffices to prove the claim for all conjunctive query plans. We prove the following statement by induction on k: if $\mathcal{Q}$ is a query, $\mathcal{P}$ is a conjunctive query plan, and $e_1, \ldots, e_k$ is a sequence of queries with $e_1 = \mathcal{P}^{exp}$, $e_k \subseteq \mathcal{Q}$, and e_{i+1} results from e_i by applying a chase step, then $\mathcal{P} \subseteq_\Sigma (\bar{\mathcal{Q}}, \mathcal{R})$. This statement would prove that $(\bar{\mathcal{Q}}, \mathcal{R})$ is maximally-contained in $\mathcal{Q}$ relative to Σ.

For $k = 1$, $\mathcal{P}^{exp}$ is contained in $\mathcal{Q}$. As shown in Theorem 2.3.1, it follows that $\mathcal{P}$ is contained in $(\mathcal{Q}, \mathcal{V}^{-1})$. It follows that $\mathcal{P}$ is contained in $(\bar{\mathcal{Q}}, \mathcal{R})$ relative to Σ.

For the induction step, let $k > 1$ and assume $e_{k-1} \not\subseteq \mathcal{Q}$. Let $\bar{A} \rightarrow B$ be the functional dependency that holds for relation p and that is applied from e_{k-1} to e_k. Then e_{k-1} contains two subgoals of p, g_1 and g_2, with the same values/variables for the attributes in $\bar{A}$, and g_1 contains a variable X for attribute B that is replaced by some value/variable in e_k. Let h be the containment mapping [6] that shows that $\mathcal{Q}$ contains e_k. Replace every value/variable X_i in an argument position in $\mathcal{Q}$ that is mapped by h to an argument position in e_k that used to be variable X in e_{k-1} by a new variable X_i'. For each of the new variables X_i', add two subgoals of p to $\mathcal{Q}$ with the identical new variables for the corresponding attributes $\bar{A}$, X_i and X_i' for attribute B respectively, and new variables for the remaining attributes. We can now find a containment mapping from query $\mathcal{Q}'$ to query e_{k-1}. This mapping shows that e_{k-1} is contained in $\mathcal{Q}'$. Therefore, $\mathcal{P}^{exp} \equiv e_1, \ldots, e_{k-1}$ is a chase sequence with $e_{k-1} \subseteq \mathcal{Q}'$. By the induction hypothesis we have that $\mathcal{P} \subseteq (\bar{\mathcal{Q}}', \mathcal{R})^{exp}$. Using the chase rule $chase(\bar{A} \rightarrow B)$, the transitivity rule, and the reflexivity of relation e, we can show that $(\bar{\mathcal{Q}}', \mathcal{R}) \subseteq (\bar{\mathcal{Q}}, \mathcal{R})$. It follows that $\mathcal{P} \subseteq_\Sigma (\bar{\mathcal{Q}}, \mathcal{R})$.

Query $\bar{\mathcal{Q}}$ contains all subgoals in $\mathcal{Q}$, and at most as many additional subgoals of e as the sum of all arities of the subgoals in $\mathcal{Q}$. Also, there are as many inverse rules as there are subgoals in all view definitions in $\mathcal{V}$ together. Finally, there are exactly as many chase rules as there are functional dependencies in Σ. We can conclude that $(\bar{\mathcal{Q}}, \mathcal{R})$ can be constructed in time polynomial in the size of $\mathcal{Q}$, $\mathcal{V}$ and Σ. $\square$ $\square$

Example 2.4.1 The following is the query plan for the running example that results from the construction described in this section. It consists of the rectified user query, the chase rules, the transitivity rule, and the inverse rules. Relation e is assumed to be reflexive.

$$
\begin{array}{lll}
\bar{q}(L) & :- & location(C,Y,L'),\ e(C,pods),\ e(Y,1989),\ e(L',L) \\
e(C,C') & :- & conference(P,C),\ conference(P',C'),\ e(P,P') \\
e(Y,Y') & :- & year(P,Y),\ year(P',Y'),\ e(P,P') \\
e(L,L') & :- & location(C,Y,L),\ location(C',Y',L'),\ e(C,C'),\ e(Y,Y') \\
e(X,Y) & :- & e(X,Z),\ e(Z,Y)
\end{array}
$$

$$
\begin{array}{lll}
conference(P, C) & :- & v_1(P, C, Y) \\
year(P, Y) & :- & v_1(P, C, Y) \\
conference(P, f_1(P, L)) & :- & v_2(P, L) \\
year(P, f_2(P, L)) & :- & v_2(P, L) \\
location(f_1(P, L), f_2(P, L), L) & :- & v_2(P, L)
\end{array}
$$

If relation e cannot be assumed to be reflexive, for example because the query plan execution engine doesn't allow to treat relation e differently from other IDB predicates, then the following rules must be added to the above query plan:

$$
\begin{array}{lll}
e(C, C') & :- & conference(P, C),\ conference(P, C') \\
e(Y, Y') & :- & year(P, Y),\ year(P, Y') \\
e(L, L') & :- & location(C, Y, L),\ location(C, Y, L') \\
e(L, L') & :- & location(C, Y, L),\ location(C, Y', L'),\ e(Y, Y') \\
e(L, L') & :- & location(C, Y, L),\ location(C', Y, L'),\ e(C, C')
\end{array}
$$

Theorem 2.4.1 showed that all results produced by this query plan that do not contain any function symbols are answers to the user query. Moreover, the result of every datalog query plan that is guaranteed to produce only answers to the user query is contained in the result of the above query plan. In Section 2.7 we will show how to eliminate the function symbols from this query plan. The resulting datalog query plan is maximally-contained in the user query. □

We showed that recursive query plans are expressive enough to extract the maximal amount of information from the information sources even in the presence of functional dependencies. Still, one might ask whether it is somehow possible to do without recursion in the plans. The following example shows that recursion is really needed in order not to miss any answers.

Example 2.4.2 Suppose we have the following world relation

schedule(Airline,Flight_no,Date,Pilot,Aircraft)

which represents the pilot that is scheduled for a certain flight, and the aircraft that is used for this flight. The functional dependencies that we consider for this example are

$Pilot \rightarrow Airline$ and
$Aircraft \rightarrow Airline$

expressing that pilots work for only one airline, and that there is no joint ownership of aircraft between airlines. The following information source is available:

$$v_3(D, P, C) :- schedule(A, N, D, P, C)$$

v_3 records on which date which pilot flies which aircraft. Assume a user asks for pilots that work for the same airline as Mike:

$$q(P) :- schedule(A, N, D, mike, C),\ schedule(A, N', D', P, C')$$

Source v_3 doesn't record the airlines that pilots work for. Nonetheless, using the functional dependencies of relation *schedule*, conclusions can be drawn about which pilots work for the same airline as Mike. For example, if both Mike and Ann are known to have flown aircraft #111, then Ann works for the same airline as Mike because of the functional dependency *Aircraft* $\rightarrow$ *Airline*. Moreover, if Ann is known to have flown aircraft #222, and John has flown aircraft #222 as well, then John works also for the same airline as Mike. This time, both functional dependencies were used to draw this conclusion. In general, the query plan $\mathcal{P}_n$ given by

$$\begin{aligned} q_n(P) :- \ & v_3(D_1, mike, C_1),\ v_3(D_2, P_2, C_1),\ v_3(D_3, P_2, C_2),\ v_3(D_4, P_3, C_2),\ \ldots, \\ & v_3(D_{2n-2}, P_n, C_{n-1}),\ v_3(D_{2n-1}, P_n, C_n),\ v_3(D_{2n}, P, C_n) \end{aligned}$$

is contained in the user query for each n. Moreover, each $\mathcal{P}_n$ is not contained in any shorter query plan. Therefore, any query plan with a fixed number of subgoals cannot be maximally-contained in the user query. $\square$

2.5 Full generalized dependencies

In this section we generalize the algorithm of the previous section to arbitrary full generalized dependencies. The added expressive power of full generalized dependencies allows us, for example, to express constraints between different relations. As an example, assume that United Airlines as a rule always uses one specific aircraft for every connection, in both directions. This can be expressed by the following full generalized dependencies:

$$schedule(ua, N, D, P, C) \wedge schedule(ua, N, D', P', C') \ \Rightarrow\ C = C'$$

$$\begin{aligned} & flight(ua, N, F, T) \wedge flight(ua, N', T, F) \wedge \\ & \quad schedule(ua, N, D, P, C) \wedge schedule(ua, N', D', P', C') \ \Rightarrow\ C = C' \end{aligned}$$

The first full generalized dependency expresses that United Airlines operates only one aircraft for every flight number. The second full generalized dependency states that the aircraft used in both directions are the same.

The key to generalizing our algorithm is to define chase rules for these more general dependencies. Let δ be a full generalized dependency. The rectified full generalized dependency $\bar{\delta}$ can be obtained from δ by rectifying the antecedent of its implication using the same procedure as for rectifying queries presented in Section 2.4. For example, the rectified version of first the full generalized dependency above is the following full generalized dependency:

$$\begin{aligned} & schedule(A, N, D, P, C) \wedge schedule(A', N', D', P', C') \wedge \\ & \quad A = ua \wedge A' = ua \wedge N = N' \ \Rightarrow\ C = C' \end{aligned}$$

For every full generalized dependency there is an equivalent rectified full generalized dependency. Therefore, it suffices to define generalized chase rules for rectified full generalized dependencies only.

Definition 2.5.1 (generalized chase rules) Let

$$\forall \bar{X}\ [\ p_1(\bar{X}_1) \wedge \ldots \wedge p_{n-1}(\bar{X}_{n-1})\ \Rightarrow\ p_n(\bar{X}_n)\ ]$$

be a rectified full generalized dependency, where $p_1, \ldots, p_n$ are either world relations or equality assertions. The *generalized chase rule* corresponding to this full generalized dependency is the following rule:

$$\bar{p}_n(X_n)\ :-\ \bar{p}_1(\bar{X}_1),\ \ldots,\ \bar{p}_{n-1}(\bar{X}_{n-1}).$$

If p_i is a world relation, then $\bar{p}_i$ is p_i. Otherwise, p_i is an equality assertion $Y_i = Z_i$, and $\bar{p}_i$ is defined to be $e(Y_i, Z_i)$. □

We denote by $chase(\Delta)$ the set of generalized chase rules corresponding to the full generalized dependencies in Δ. The generalized chase rule corresponding to the rectified full generalized dependency mentioned above is the following rule:

$$\begin{aligned} e(C, C')\ :-\ & schedule(A, N, D, P, C),\ schedule(A', N', D', P', C'), \\ & e(A, ua),\ e(A', ua),\ e(N, N') \end{aligned}$$

Note that for functional dependencies, generalized chase rules are identical to the corresponding chase rules defined in Section 2.4. To generate a maximally-contained plan in the presence of full generalized dependencies, we follow the same algorithm as in Section 2.4, except that we replace the chase rules by the generalized chase rules. The following theorem generalizes Theorem 2.4.1.

Theorem 2.5.1 *Let Δ be a set of full generalized dependencies, $\mathcal{V}$ a set of conjunctive source descriptions, and let $\mathcal{Q}$ be a query over the world relations. Let $\mathcal{R}$ denote the set of rules $\mathcal{V}^{-1} \cup chase(\Delta) \cup \mathcal{T}$. Then, $(\bar{\mathcal{Q}}, \mathcal{R})\downarrow$ is maximally-contained in $\mathcal{Q}$ relative to Δ. Furthermore, $(\bar{\mathcal{Q}}, \mathcal{R})$ can be constructed in time polynomial in the size of $\mathcal{Q}$, $\mathcal{V}$, and Δ.* □

The dependencies that we consider in this chapter are called *full* generalized dependencies because all the variables that appear on the right hand side of the implication in a dependency must already occur on the left hand side. This restrictions is essential. The following dependency is *not* a full generalized dependency:

$$flight(A, N, F, T)\ \Rightarrow\ \exists N'\, flight(A, N', T, F)$$

This kind of dependency is usually refered to as an *inclusion dependency* because it asserts that the set of values appearing for some attribute is included in the set of values appearing for some other attribute. The dependency expresses that if an airline offers a flight between two cities, then the airline offers the flight in both directions. If we allowed this kind of dependency, the corresponding chase rule would be

$$flight(A, f(A, N, F, T), T, F)\ :-\ flight(A, N, F, T).$$

But this rule is recursive and introduces new function terms. Therefore, naive bottom-up evaluation of a query containing this rule wouldn't terminate. The question of whether it is possible to build the maximally-contained query plan in the presence of general — including nonfull — dependencies remains open.

2.6 Limitations on binding patterns

The last case we consider in this chapter is the presence of limitations on access to information sources. In practice, some information sources cannot answer arbitrary atomic queries on the relation they store. In particular, the information source may require that some of the arguments of its relations be given as input. To model source capabilities, we attach to each source relation an *adornment* (see [51], Chap. 12), specifying which binding patterns the source supports.[5] An adornment of a view definition of v is a string of b's and f's of length n, where n is the arity of v. The meaning of the adornment is that the source only supports queries in which the arguments with b adornments are bound. The other arguments may be either bound or free. For example, the adornment v^{bf} means that the first argument must be bound in queries on v. We define an *executable* query plan as follows.

Definition 2.6.1 (executable query plan) Let $\mathcal{V}$ be a set of source descriptions with binding adornments, and let $\mathcal{P}$ be the following conjunctive query plan:

$$q(\bar{X}) :- v_1(\bar{X}_1), \ldots, v_n(\bar{X}_n)$$

Query plan $\mathcal{P}$ is *executable* if the following holds for $i = 1, \ldots, n$: let j be an argument position of v_i that has a b adornment, and let α be the j'th element in $\bar{X}_i$. Then, either α is a constant, or α appears in $\bar{X}_1 \cup \ldots \cup \bar{X}_{i-1}$. □

A datalog query plan includes source relations and IDB relations, which we model as having the all-free adornment (i.e., f^n, where n is the relation's arity). A query plan $\mathcal{P}$ is executable if for every rule $r \in \mathcal{P}$, r is executable.

In [44] it is shown that the number of literals in a query plan that is equivalent to the user query is bounded. However, as the following example, adapted from [34] shows, when sources have limitations on binding patterns, there may *not* be a finite maximally-contained query plan, if we restrict ourselves to query plans without recursion.

Example 2.6.1 Consider the following sources:

$$\begin{array}{ll} v_1^f(X) & :- podsPapers(X) \\ v_2^{bf}(X,Y) & :- cites(X,Y) \\ v_3^b(X) & :- awardPaper(X) \end{array}$$

The first source stores PODS papers, the second is a citation database, but only accepts queries where the first argument is bound, and the third source will tell us whether a *given* paper won an award. Suppose our query is to find all the award papers:

$$q(X) :- awardPaper(X)$$

For each n, the following is an executable conjunctive query plan $\mathcal{P}_n$ that is contained in $\mathcal{Q}$:

[5] For simplicity of exposition, we assume that each source relation has a single adornment.

$$q_n(Z_n) :- v_1(Z_0),\ v_2(Z_0, Z_1),\ \ldots, v_2(Z_{n-1}, Z_n),\ v_3(Z_n).$$

Furthermore, for each n, $\mathcal{P}_n$ may produce answers that are not obtained by any other $\mathcal{P}_i$, for any i. Intuitively, a paper will be in the answer to $\mathcal{P}_i$ if the number of links that need to be followed from a PODS paper is i. Therefore, there is no bound on the size of the conjunctive queries in the maximally-contained plan.

We now show that by allowing recursive plans we *can* produce a maximally-contained query plan. On our example, the construction will yield the following query plan. The construction is based on inventing a new recursively-defined relation, *papers*, whose extension will be the set of all papers that can be reached from the papers in v_1.

$$\begin{array}{ll} papers(X) & :- v_1^f(X) \\ papers(X) & :- papers(Y),\ v_2^{bf}(Y, X) \\ q(X) & :- papers(X),\ v_3^b(X). \end{array}$$

□

We now describe the construction for a given set of adorned source relations $\mathcal{V}$ and a query $\mathcal{Q}$. The recursive plan includes a unary relation *dom* whose intended extension is the set of all constants that appear in the query or in the view definitions, or that can be obtained by iteratively querying the sources. The rules involving *dom* are the following.

Definition 2.6.2 (domain rules) Let $v \in \mathcal{V}$ be a source relation of arity n. Suppose the adornment of v says that the arguments in positions $1, \ldots, l$ need to be bound, and the arguments $l+1, \ldots, n$ can be free. Then for $i = l+1, \ldots, n$, the following rule is a *domain rule*:

$$dom(X_i) :- dom(X_1),\ \ldots,\ dom(X_l),\ v(X_1, \ldots, X_n).$$

Also, if c is a constant appearing in the view definitions in $\mathcal{V}$ or in query $\mathcal{Q}$, then the fact $dom(c)$ is a *domain rule*. □

We denote by $domain(\mathcal{V}, \mathcal{Q})$ the set of rules described above for defining the predicate *dom*. Notice that all domain rules are executable, and that relation *dom* has adornment f. Every query plan $\mathcal{P}$ can be transformed to an executable query plan by inserting the literal $dom(X)$ before subgoals g in $\mathcal{P}$ that have a variable X in an argument position that is required to be bound, and X does not appear in the subgoals to the left of g in the body. The resulting query plan, denoted by $\mathcal{P}^{exec}$, is executable. Moreover, we can show that $\mathcal{P}^{exec}$ is equivalent to $\mathcal{P}$. Combining this result with the one of the previous section, we can conclude with the following theorem:

Theorem 2.6.1 *Let Δ be a set of full generalized dependencies, $\mathcal{V}$ a set of conjunctive source descriptions with binding adornments, and let $\mathcal{Q}$ be a query over the global relations. Then $\bar{\mathcal{Q}} \cup chase(\Sigma) \cup \mathcal{T} \cup domain(\mathcal{V}, \mathcal{Q}) \cup (\mathcal{V}^{-1})^{exec}$ is maximally-contained in $\mathcal{Q}$ relative to Δ.* □

Finally, we note that the query plan can be constructed in time polynomial in the size of $\mathcal{Q}$, $\mathcal{V}$ and Δ.

2.7 Eliminating function symbols

Although in Section 2.3 we demonstrated an efficient procedure to answer a datalog query as well as possible given only source relations, it is desirable to transform the constructed logic query to a datalog query that represents this answer. Indeed, we are looking for a datalog query that is equivalent to $(\mathcal{Q}^-, \mathcal{V}^{-1})\downarrow$. The key observation underlying the construction of such a datalog query is that there are only finitely many function symbols in $(\mathcal{Q}^-, \mathcal{V}^{-1})$. Because nested function expressions can never be generated using bottom-up evaluation, it is possible, with a little bit of bureaucracy, to keep track of function terms produced by $(\mathcal{Q}^-, \mathcal{V}^{-1})$ without actually generating tuples containing function terms.

The transformation proceeds in a bottom-up fashion, starting with the inverse rules. Function terms like $f(X_1, \ldots, X_k)$ in the IDB predicates of $\mathcal{V}^{-1}$ are eliminated by replacing them by the list of variables $X_1, \ldots, X_k$ that occur in them. The IDB predicate names need to be annotated to indicate that $X_1, \ldots, X_k$ belonged to the function term $f(X_1, \ldots, X_k)$. For instance, in Example 2.3.2 the rule

$$edge(X, f(X,Y)) :- v(X,Y)$$

is replaced by the rule

$$edge^{\langle \star, f(\star,\star)\rangle}(X, X, Y) :- v(X,Y)$$

The annotation $\langle \star, f(\star,\star)\rangle$ represents the fact that the first argument in $edge^{\langle \star, f(\star,\star)\rangle}$ is identical to the first argument in $edge$, and that the second and third argument in $edge^{\langle \star, f(\star,\star)\rangle}$ combine to a function term with the function symbol f as the second argument of $edge$. If bottom-up evaluation of $(\mathcal{Q}^-, \mathcal{V}^{-1})$ can yield a function term for an argument of an IDB predicate in $\mathcal{Q}^-$, then a new rule is added with correspondingly expanded and annotated predicates. The following definition states this construction formally. $\bar{X}$ is a shorthand for a list of variables or constants, and $\langle\bar{\beta}\rangle$ is a shorthand for an adornment. $\bar{X}[i]$ and $\bar{\beta}[i]$ stand for the ith position in $\bar{X}$ and $\bar{\beta}$ respectively.

Definition 2.7.1 (predicate splitting) Let $\mathcal{P}$ be a query plan with function symbols. We are going to define a query plan $\mathcal{P}^{split}$ that encodes exactly the derivations in $\mathcal{P}$, but doesn't contain function symbols. The transformation from $\mathcal{P}$ to $\mathcal{P}^{split}$ is called *predicate splitting*, because an IDB predicate in $\mathcal{P}$ might be represented by several IDB predicates in $\mathcal{P}^{split}$.
If

$$p(\alpha_1, \ldots, \alpha_n) :- v(\bar{X})$$

is an inverse rule in $\mathcal{P}$, then the query plan $\mathcal{P}^{split}$ contains the rule

$$p^{\langle\beta_1,\ldots,\beta_n\rangle}(Y_1, \ldots, Y_{\gamma_n}) :- v(\bar{X})$$

with $\gamma_0 = 0$ and for $i = 1, \ldots, n$,

$$|\alpha_i| = \begin{cases} 1 & : \text{ if } \alpha_i \text{ is a variable or a constant} \\ \text{arity of } f & : \text{ if } \alpha_i \text{ is a function term with function symbol } f, \end{cases}$$

$\gamma_i = \gamma_{i-1} + |\alpha_i|$,

$$\beta_i = \begin{cases} \star & : \text{ if } \alpha_i \text{ is a variable or a constant} \\ f(\underbrace{\star, \ldots, \star}_{|\alpha_i|}) & : \text{ if } \alpha_i \text{ is a function term with function symbol } f, \end{cases}$$

and for $k_i = 1, \ldots, |\alpha_i|$:

$$Y_{\gamma_{i-1}+k_i} = \begin{cases} \alpha_i & : \text{ if } \alpha_i \text{ is a variable or a constant} \\ X & : \text{ if } \alpha_i \text{ is a function term with } X \text{ as its } k\text{th argument.} \end{cases}$$

If

$$p(\bar{X}) :- p_1(\bar{X}_1), \ \ldots, \ p_m(\bar{X}_m)$$

is a rule in $\mathcal{P}$, and

1. the query plan $\mathcal{P}^{split}$ contains rules that have $p_1^{\langle \bar{\beta}_1 \rangle}, \ldots, p_m^{\langle \bar{\beta}_m \rangle}$ as heads,
2. if for some i, j, i', j', $\bar{X}_j[i]$ is identical to $\bar{X}_{j'}[i']$, then $\bar{\beta}_j[i] = \bar{\beta}_{j'}[i']$, and
3. if for some i, j, $\bar{X}_j[i]$ is a constant, then $\bar{\beta}_j[i] = \star$,

then the query plan $\mathcal{P}^{split}$ contains the rule

$$p^{\langle \bar{\beta} \rangle}(\bar{Y}) :- p_1^{\langle \bar{\beta}_1 \rangle}(\bar{Y}_1), \ \ldots, \ p_m^{\langle \bar{\beta}_m \rangle}(\bar{Y}_m)$$

such that for all i, $\bar{\beta}[i] = \bar{\beta}_j[k]$ for some j, k with $\bar{X}[i] = \bar{X}_j[k]$, and if a variable X that occurs at $\bar{X}_j[k]$ for some j, k occurs anywhere else, then the variables and constants that represent X in $\bar{Y}_j$ are the same as the variables and constants that represent X in the other places. □

The following example shows this transformation.

Example 2.7.1 The logic query from Example 2.3.2 is transformed to the following datalog query. The lines indicate the stages in the generation of the datalog rules.

$$edge^{\langle \star, f(\star,\star) \rangle}(X, X, Y) :- v(X, Y)$$
$$edge^{\langle f(\star,\star), \star \rangle}(X, Y, Y) :- v(X, Y)$$

$$q^{\langle \star, f(\star,\star) \rangle}(X, Y_1, Y_2) :- edge^{\langle \star, f(\star,\star) \rangle}(X, Y_1, Y_2)$$
$$q^{\langle f(\star,\star), \star \rangle}(X_1, X_2, Y) :- edge^{\langle f(\star,\star), \star \rangle}(X_1, X_2, Y)$$

$$q^{\langle \star, \star \rangle}(X, Y) :- edge^{\langle \star, f(\star,\star) \rangle}(X, Z_1, Z_2), \ q^{\langle f(\star,\star), \star \rangle}(Z_1, Z_2, Y)$$
$$q^{\langle f(\star,\star), f(\star,\star) \rangle}(X_1, X_2, Y_1, Y_2) :- edge^{\langle f(\star,\star), \star \rangle}(X_1, X_2, Z), \ q^{\langle \star, f(\star,\star) \rangle}(Z, Y_1, Y_2)$$

$$q^{\langle f(\star,\star),\star\rangle}(X_1, X_2, Y) \quad :- \; edge^{\langle f(\star,\star),\star\rangle}(X_1, X_2, Z),\; q(Z, Y)$$

$$q^{\langle \star,f(\star,\star)\rangle}(X, Y_1, Y_2) \quad :- \; edge^{\langle \star,f(\star,\star)\rangle}(X, Z_1, Z_2),\; q^{\langle f(\star,\star),f(\star,\star)\rangle}(Z_1, Z_2, Y_1, Y_2)$$

□

The generated datalog query shows explicitly in which arguments the original logic query was able to produce function terms.

Some tuples with function symbols might never have been able to contribute to an answer without function symbols. Using our exact bookkeeping of function symbols, we are able to eliminate the derivations of these useless tuples. In the following we are going to present two optimizations.

Define a predicate p to be *relevant* if there is a path in the dependency graph from p to the query predicate q. If a predicate p is not relevant, then no derivation of a tuple in the answer requires p. Therefore, all rules for irrelevant predicates can be dropped without losing any answers.

Example 2.7.2 The dependency graph for the datalog query plan in Example 2.7.1 is shown in Figure 2.1. There are no paths from predicates $q^{\langle \star,f(\star,\star)\rangle}$ and $q^{\langle f(\star,\star),f(\star,\star)\rangle}$ to q. Therefore, these

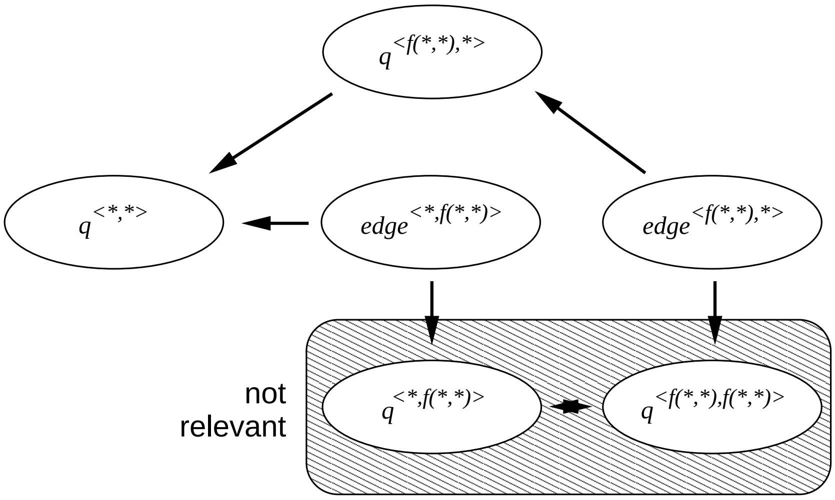


Figure 2.1: *Dependency graph for the query plan in Example 2.7.1.*

two predicates are irrelevant. The three rules defining the irrelevant predicates can be dropped. The following is the resulting datalog query:

$$edge^{\langle \star,f(\star,\star)\rangle}(X, X, Y) :- \; v(X, Y)$$

$$edge^{\langle f(\star,\star),\star\rangle}(X, Y, Y) :- \; v(X, Y)$$

$$q^{\langle f(\star,\star),\star\rangle}(X_1, X_2, Y) \quad :- \; edge^{\langle f(\star,\star),\star\rangle}(X_1, X_2, Y)$$

$$q^{\langle f(\star,\star),\star\rangle}(X_1, X_2, Y) \quad :- \; edge^{\langle f(\star,\star),\star\rangle}(X_1, X_2, Z),\; q(Z, Y)$$

$$q^{\langle\star,\star\rangle}(X,Y) \quad :- \; edge^{\langle\star,f(\star,\star)\rangle}(X,Z_1,Z_2), \; q^{\langle f(\star,\star),\star\rangle}(Z_1,Z_2,Y)$$

□

The second optimization doesn't reduce the number of derivations, but is an easy way to save unneccessary copying of data during the evaluation of the datalog program. If p is a predicate in a datalog query that has only one rule, and the body of this rule has only one subgoal, then predicate p can be eliminated from the query. For every rule having p as one of its subgoals, unify this subgoal with the head of the rule of p, and replace the subgoal by the corresponding body of the rule of p.

Example 2.7.3 Predicates $edge^{\langle\star,f(\star,\star)\rangle}$ and $edge^{\langle f(\star,\star),\star\rangle}$ in Example 2.7.1 have only one rule and only one subgoal in the bodies of their rules, and can therefore be eliminated. The following is the resulting datalog query:

$$\begin{aligned}
&q^{\langle f(\star,\star),\star\rangle}(X,Y,Y) &&:- \; v(X,Y)\\
&q^{\langle f(\star,\star),\star\rangle}(X,Z,Y) &&:- \; v(X,Z), \; q(Z,Y)\\
&q(X,Y) &&:- \; v(X,Z), \; q^{\langle f(\star,\star),\star\rangle}(X,Z,Y)
\end{aligned}$$

□

Because we keep track of function symbols in $(\mathcal{Q}^-,\mathcal{V}^{-1})^{split}$, we know that the resulting instance of the query predicate q with the all "$\star$" adornment is exactly the subset of the result of $(\mathcal{Q}^-,\mathcal{V}^{-1})$ that does not contain function symbols. The following is therefore an immediate corollary of Theorem 2.3.1.

Corollary 2.7.1 *For every datalog query $\mathcal{Q}$ and every set of conjunctive source descriptions $\mathcal{V}$ over the EDB predicates of $\mathcal{Q}$, the query plan $(\mathcal{Q}^-,\mathcal{V}^{-1})^{split}$ is maximally-contained in $\mathcal{Q}$. Moreover, if there exists a query plan that is equivalent to $\mathcal{Q}$, then $(\mathcal{Q}^-,\mathcal{V}^{-1})^{split}$ is equivalent to $\mathcal{Q}$.*

2.8 Comparison with other algorithms

In the following, we are going to compare the construction presented in this chapter with two other algorithms for answering queries using views: the *bucket algorithm* [38, 39], and the *unification-join algorithm* [43]. Because these algorithms cannot handle recursive queries, dependencies, or limitations on binding patterns, we will illustrate the differences between our construction and these two algorithms using a nonrecursive example query without dependencies and without limitations on binding patterns.

Example 2.8.1 Assume that three sources are available which are described by the following view definitions:

$$\begin{aligned}
&v_1(F,T) &&:- \; \mathit{flight}(F,T,wn)\\
&v_2(F,T) &&:- \; \mathit{flight}(F,T,ua)\\
&v_3(F,T,C) &&:- \; \mathit{flight}(F,Z,C), \; \mathit{flight}(Z,T,C)
\end{aligned}$$

The first and second source store the pairs of cities between which Southwest Airlines (wn) and United Airlines (ua) respectively offer direct flights. The third source stores pairs of cities that are connected by flights with one stop-over, together with the airlines that offer these flights. As an example query, assume a user wants to know the airlines that fly from San Francisco to New York with at most one stop-over:

$$q(C) :- \mathit{flight}(sfo, jfk, C) \qquad (\alpha)$$
$$q(C) :- \mathit{flight}(sfo, Z, C),\ \mathit{flight}(Z, jfk, C) \qquad (\beta)$$

Using the construction in Section 2.3, the following maximally-contained logic query plan can be obtained in polynomial time:

$$\begin{array}{ll}
\mathit{flight}(F, T, wn) :- v_1(F, T) & \\
\mathit{flight}(F, T, ua) :- v_2(F, T) & \\
\mathit{flight}(F, g(F, T, C), C) :- v_3(F, T, C) & (*) \\
\mathit{flight}(g(F, T, C), T, C) :- v_3(F, T, C) & (*) \\
q(C) :- \mathit{flight}(sfo, jfk, C) & \\
q(C) :- \mathit{flight}(sfo, Z, C),\ \text{it flight}(Z, jfk, C) &
\end{array}$$

The transformation presented in Section 2.7 would remove the two rules marked with (*), and would add the following rule:

$$q(C) :- v_3(F, T, C)$$

□

2.8.1 Bucket algorithm

The bucket algorithm is the algorithm used for query planning in the Information Manifold system [33,38,39]. For each subgoal p_i in the user query, a "bucket" B_i is created. If v_j is a view definition containing a predicate r unifiable with p_i, then $v_j\sigma$ is inserted into B_i, where σ is a most general unifier of p_i and r preferring the variables in p_i. For each conjunctive user query c separately, the bucket algorithm constructs conjunctive query plans with the same head as c, and all possible combinations of source relations taken from the buckets corresponding to the subgoals of c as bodies. For each of these query plans, the algorithm checks whether it can add a constraint C to the body, such that the expansion of the resulting query is contained in c. All conjunctive query plans that pass this containment test, will be evaluated to find the answer to the user query.

Example 2.8.2 Applied to the query in Example 2.8.1, the bucket algorithm creates three buckets B_1, B_2, and B_3 for the three subgoals $\mathit{flight}(sfo, jfk, C)$, $\mathit{flight}(sfo, Z, C)$, and $\mathit{flight}(Z, jfk, C)$ respectively. The buckets are filled as follows:

B_1	B_2	B_3
$v_1(sfo, jfk)$	$v_1(sfo, Z)$	$v_1(Z, jfk)$

$$\begin{array}{lll} v_2(sfo, jfk) & v_2(sfo, Z) & v_2(Z, jfk) \\ v_3(sfo, T_1, C) & v_3(sfo, T_2, C) & v_3(Z, T_3, C) \\ v_3(F_1, jfk, C) & v_3(F_2, Z, C) & v_3(F_3, jfk, C) \end{array}$$

For each of the four query plans

$$q(C) :- \ v_1(sfo, jfk) \qquad (1)$$
$$q(C) :- \ v_2(sfo, jfk) \qquad (2)$$
$$q(C) :- \ v_3(sfo, T_1, C)$$
$$q(C) :- \ v_3(F_1, jfk, C)$$

the algorithm checks whether, after adding some constraints, its expansion is contained in the conjunctive user query (α). Further, the following sixteen query plans are checked to see whether, after adding some constraints, their expansion is contained in the conjunctive user query (β):

$$q(C) :- \ v_1(sfo, Z), \ v_1(Z, jfk) \qquad (3)$$
$$q(C) :- \ v_1(sfo, Z), \ v_2(Z, jfk)$$
$$q(C) :- \ v_1(sfo, Z), \ v_3(Z, T_3, C) \qquad (4)$$
$$q(C) :- \ v_1(sfo, Z), \ v_3(F_3, jfk, C) \qquad (5)$$
$$q(C) :- \ v_2(sfo, Z), \ v_1(Z, jfk)$$
$$q(C) :- \ v_2(sfo, Z), \ v_2(Z, jfk) \qquad (6)$$
$$q(C) :- \ v_2(sfo, Z), \ v_3(Z, T_3, C) \qquad (7)$$
$$q(C) :- \ v_2(sfo, Z), \ v_3(F_3, jfk, C) \qquad (8)$$
$$q(C) :- \ v_3(sfo, T_2, C), \ v_1(Z, jfk)$$
$$q(C) :- \ v_3(sfo, T_2, C), \ v_2(Z, jfk)$$
$$q(C) :- \ v_3(sfo, T_2, C), \ v_3(Z, T_3, C) \qquad (9)$$
$$q(C) :- \ v_3(sfo, T_2, C), \ v_3(F_3, jfk, C) \qquad (10)$$
$$q(C) :- \ v_3(F_2, Z, C), \ v_1(Z, jfk) \qquad (11)$$
$$q(C) :- \ v_3(F_2, Z, C), \ v_2(Z, jfk) \qquad (12)$$
$$q(C) :- \ v_3(F_2, Z, C), \ v_3(Z, T_3, C) \qquad (13)$$
$$q(C) :- \ v_3(F_2, Z, C), \ v_3(F_3, jfk, C) \qquad (14)$$

For each numbered query plan, a constraint can be added to its body such that it passes the containment test. For example, the constraint that needs to be added to query (1) is $''C = wn''$, and the constraint that needs to be added to query (10) is $''T_2 = jfk''$. □

As the example shows, the bucket algorithm has to perform a lot of containment tests. This is quite expensive, especially because testing containment of conjunctive queries is NP-complete.

2.8.2 Unification-join algorithm

The first step of the unification-join algorithm is the same as the first step of the construction given in this chapter, namely the generation of inverse rules. However, whereas our construction transforms the original query together with the inverse rules into a query plan, the unification-join algorithm constructs a set of conjunctive query plans using the so-called unification-join as a central step.

For each subgoal p_i in the user query, a "label" L_i is created. If $r :- v$ is one of the inverse rules, and r and p_i are unifiable, then the pair $(\sigma \downarrow p_i,\ v\sigma)$ is inserted into L_i provided that $\sigma \downarrow q$ does not contain any function terms. Here, σ is a most general unifier of p_i and r, and $\sigma \downarrow p_i$ and $\sigma \downarrow q$ are the restriction of σ to the variables in p_i and to the variables in the query predicate q respectively. The unification-join of two labels L_1 and L_2, denoted $L_1 \overset{u}{\bowtie} L_2$, is defined as follows. If L_1 contains a pair (σ_1, t_1) and L_2 contains a pair (σ_2, t_2), then $L_1 \overset{u}{\bowtie} L_2$ contains the pair $(\sigma_1\sigma \cup \sigma_2\sigma, (t_1 \wedge t_2)\sigma)$ where σ is a most general substitution such that $\sigma_1\sigma \downarrow \sigma_2 = \sigma_2\sigma \downarrow \sigma_1$, provided there is such a substitution σ, and provided $\sigma_1\sigma \downarrow q$, $\sigma_2\sigma \downarrow q$, and $(t_1 \wedge t_2)\sigma$ do not contain any function terms. If $(\sigma, v_{i_1} \wedge \ldots \wedge v_{i_n})$ is in the unification-join of all labels corresponding to the subgoals in one of the conjunctive user queries, and this user query has head h, then the query plan with head $h\sigma$ and body $v_{i_1}, \ldots, v_{i_n}$ is part of the result.

Example 2.8.3 Applied to the query in Example 2.8.1, the unification-join algorithm generates three labels corresponding to the three subgoals of the query, $\mathit{flight}(sfo, jfk, C)$, $\mathit{flight}(sfo, Z, C)$, and $\mathit{flight}(Z, jfk, C)$, respectively:

$$
\begin{array}{l}
\underline{L_1} \\
(\{C \rightarrow wn\},\ v_1(sfo, jfk)) \\
(\{C \rightarrow ua\},\ v_2(sfo, jfk)) \\
\underline{L_2} \\
(\{C \rightarrow wn\},\ v_1(sfo, Z)) \\
(\{C \rightarrow ua\},\ v_2(sfo, Z)) \\
(\{Z \rightarrow g(sfo, T, C)\},\ v_3(sfo, T, C)) \\
\underline{L_3} \\
(\{C \rightarrow wn\},\ v_1(Z, jfk)) \\
(\{C \rightarrow ua\},\ v_2(Z, sfo)) \\
(\{Z \rightarrow g(F, jfk, C)\},\ v_3(F, jfk, C))
\end{array}
$$

The unification-join of L_2 and L_3 is

$$
\begin{array}{l}
\underline{L_2 \overset{u}{\bowtie} L_3} \\
(\{C \rightarrow wn\},\ v_1(sfo, Z) \wedge v_1(Z, jfk)) \\
(\{C \rightarrow ua\},\ v_2(sfo, Z) \wedge v_2(Z, jfk)) \\
(\{Z \rightarrow g(sfo, jfk, C)\},\ v_3(sfo, jfk, C)).
\end{array}
$$

The labels corresponding to conjunctive queries (α) and (β) are L_1 and $L_2 \overset{u}{\bowtie} L_3$ respectively. The conjunctive query plans that can be constructed from L_1 and $L_2 \overset{u}{\bowtie} L_3$ are:

$$
\begin{array}{l}
q(wn) :- v_1(sfo, jfk) \\
q(ua) :- v_2(sfo, jfk)
\end{array}
$$

$$\begin{array}{ll} q(wn) & :- \; v_1(sfo, Z),\; v_1(Z, jfk) \\ q(ua) & :- \; v_2(sfo, Z),\; v_2(Z, jfk) \\ q(C) & :- \; v_3(sfo, jfk, C) \end{array}$$

□

The unification-join algorithm doesn't require any containment tests. However, it might generate an exponential number of conjunctive queries in cases when our algorithm generates a small datalog query. As an example, assume that there are k view definitions of the form

$$v_i(X, Y) :- \; p(X, Y),\; p_i(X, Y) \quad \text{for } i = 1, \ldots, k.$$

Given the user query

$$q(X_0, X_n) :- \; p(X_0, X_1),\; p(X_1, X_2),\; \ldots,\; p(X_{n-1}, X_n)$$

the constructed maximally-contained datalog query is the following:

$$\begin{array}{ll} p(X, Y) & :- \; v_1(X, Y) \\ & \vdots \\ p(X, Y) & :- \; v_k(X, Y) \\ q(X_0, X_n) & :- \; p(X_0, X_1),\; p(X_1, X_2),\; \ldots,\; p(X_{n-1}, X_n) \end{array}$$

Evaluating this datalog query requires $k - 1$ unions and $n - 1$ joins. On the other hand, the unification-join algorithm yields the following k^n conjunctive query plans:

$$q(X_0, X_n) :- \; v_{j_1}(X_0, X_1),\; v_{j_2}(X_1, X_2),\; v_{j_n}(X_{n-1}, X_n)$$
$$\text{for all } j_1, \ldots, j_n \in \{1, \ldots, k\}.$$

Evaluating these conjunctive queries requires $k^n - 1$ unions and $(n - 1)k^n$ joins.

2.9 Conclusions and related work

We introduced a novel approach to creating information gathering plans, that allows for recursive plans. We have shown that recursive plans enable us to solve three open problems. We described algorithms for obtaining a maximally-contained query plan in the case of recursive user queries, in the presence of dependencies and in the presence of limitations on binding patterns. Our results are also of practical importance because dependencies and limitations on binding patterns occur very frequently in information sources in practice (e.g., the WWW).

Recursive information gathering plans have another important methodological advantage. Query plans can be constructed by *describing* a set of inferences that the mediator needs to make in order to obtain data from its sources. As a consequence, it is simpler to construct these plans, and we believe that it is easier to extend our methods to other contexts.

Previous work on this problem did not consider cases where the queries are recursive and where full generalized dependencies exist in the world schema. The first theoretical investigations of the problem concentrated on showing a bound on the size of the resulting query plan [37,44]. These results establish the complexity of the rewriting problem, but yield only nondeterministic algorithms for its solution. As stated above, the algorithms in [34,38] propose heuristics for searching the space of candidate plans. Huyn [29] proposed "pseudo-equivalent" rewritings in the case that no equivalent rewritings exist. These ideas were used in [43] to give an algorithm for rewriting conjunctive queries given source relations described by view definitions.

The problem of finding query plans in the presence of binding-pattern limitations is considered in [44], but only an algorithm for finding an equivalent plan is presented. Later, Kwok and Weld [34] showed that if we restrict our plans to be unions of conjunctive queries, then there may *not be* a finite maximally-contained rewriting in the presence of binding-pattern limitations. More complex query capabilities in sources are considered in [40]. Complex capabilities are modeled by the ability of a source to answer a potentially infinite number of conjunctive queries. Hence, [40] considered how to answer queries given an infinite number of conjunctive views definitions.

Several authors have considered the problem of rewriting queries using views for query optimization [57,10,50]. In this context, one usually requires a query plan that is equivalent to the original query. The algorithms described in [10,50] also explain how to combine the search for query plans with a traditional System-R style query optimizer. Another use of rewriting queries using views is explored in [1] for the purpose of deciding which cached answers can be used by a mediator. The algorithms described in [1] are aimed at capturing frequently occurring cases which can be detected efficiently.

Chapter 3

Disjunctive Sources

We examine the query planning problem in data integration systems in the presence of sources that contain disjunctive data. We show that datalog, the language of choice for representing query plans in data integration systems, is not sufficiently expressive in this case. We prove that disjunctive datalog with inequality, on the other hand, is sufficiently expressive by presenting a construction of query plans that are guaranteed to extract all available information from disjunctive sources.

3.1 Introduction

We examine the query planning problem in data integration systems in the presence of sources that contain disjunctive data. The query planning problem in such systems can be formally stated as the problem of answering queries using views as described in Chapter 2. View definitions describe the data stored by sources, and query planning requires rewriting a query into one that only uses these views. In this chapter we are going to extend the algorithm for answering queries using conjunctive views introduced in Section 2.3 so that it can handle disjunction in the view definitions as well.

Example 3.1.1 Assume a data source stores flight information. More precisely, the source stores nonstop flights by United Airlines (*ua*) and Southwest Airlines (*sw*), and flights out of San Francisco International Airport (*sfo*) with one stopover. The data stored by this source can be described as being a view over a database with a relation *flight* that stores all nonstop flights. The view definition that describes this source is the following:

$$
\begin{array}{lll}
v(ua, From, To) & :- & flight(ua, From, To) \\
v(sw, From, To) & :- & flight(sw, From, To) \\
v(Airline, sfo, To) & :- & flight(Airline, sfo, Stopover), \\
 & & flight(Airline, Stopover, To)
\end{array}
$$

A user might be interested in all cities that have nonstop flights to Seattle (*sea*):

$$
Q: \quad q(From) \ :- \ flight(Airline, From, sea)
$$

If $\langle ua,jfk,sea\rangle$ is a tuple stored by the data source, then there is clearly a nonstop flight from New York (*jfk*) to Seattle. On the other hand, if the tuple $\langle ua,sfo,sea\rangle$ is stored by the data source then a nonstop flight from San Francisco to Seattle does not necessarily exist. Indeed, this tuple might be stored because there is a flight with one stopover from San Francisco to Seattle. The task of query planning in data integration systems is to find a query plan, i.e. a query that only requires views, that extracts as much information as possible from the available sources. All flights to Seattle stored by the data source with the exception of flights departing from San Francisco International Airport are nonstop flights. Therefore, the query plan is the following:

$$\mathcal{P}: \quad q(From) \; :- \; v(Airline,From,sea), \;\; From \neq sfo$$

Note that without the use of the inequality constraint "$From \neq sfo$" it wouldn't be possible to guarantee that all cities returned by the query plan indeed have nonstop flights to Seattle. □

In Chapter 2, we showed that the expressive power of datalog is both required and sufficient to represent "good" query plans in data integration systems when view definitions are restricted to be conjunctive. As we have seen in Example 3.1.1, the presence of disjunctive sources in addition requires the use of inequality constraints in query plans. So far, there are no algorithms that generate query plans with inequality constraints. But the differences between conjunctive sources and disjunctive sources are much more extensive. We will see in Example 3.1.2 that the expressive power of datalog, even with inequality, is insufficient to represent query plans that extract all available data from disjunctive sources.

Example 3.1.2 Assume that there are two data sources available which are described by the following view definitions:

$$\begin{array}{ll} v_1(X) & :- \; color(X, red) \\ v_1(X) & :- \; color(X, green) \\ v_1(X) & :- \; color(X, blue) \\ v_2(X,Y) & :- \; edge(X,Y) \end{array}$$

View v_1 stores vertices that are colored red, green, or blue. View v_2 stores pairs of vertices that are connected by an edge. Assume a user wants to know whether there is a pair of vertices of the same color that are connected by an edge:

$$\mathcal{Q}: \quad q('yes') :- \; edge(X,Y), \;\; color(X,Z), \;\; color(Y,Z).$$

Consider the graphs G_1, G_2, and G_3 in Figure 3.1. All of these graphs are not three-colorable, i.e. for every possible coloring of the vertices with at most three colors, there will be one edge that connects vertices with the same color. Therefore, every graph that contains G_1, G_2, or G_3 as a subgraph contains an edge that connects two vertices with the same color if the vertices in G_1, G_2, and G_3 are colored by at most three colors. Query plans $\mathcal{P}_1$, $\mathcal{P}_2$, and $\mathcal{P}_3$ output 'yes' exactly when the input graph contains G_1, G_2, or G_3 respectively as a subgraph and when the vertices in G_1, G_2, and G_3 respectively are colored by at most three colors:

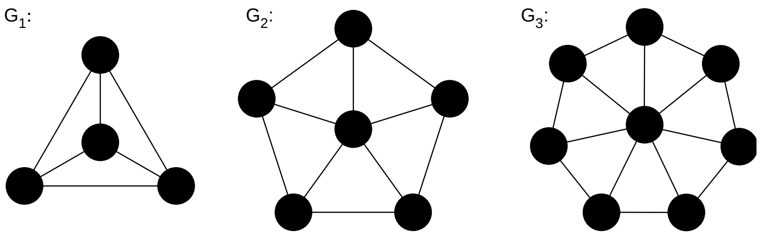


Figure 3.1: *Examples of graphs that are not 3-colorable.*

$$\begin{array}{ll}\mathcal{P}_1: & q('yes') :- v_1(X_1),\ v_1(X_2),\ v_1(X_3),\ v_1(Y),\ v_2(X_1,X_2),\ v_2(X_2,X_3),\\ & \qquad v_2(X_3,X_1), v_2(X_1,Y),\ v_2(X_2,Y),\ v_2(X_3,Y)\end{array}$$

$$\begin{array}{ll}\mathcal{P}_2: & q('yes') :- v_1(X_1),\ v_1(X_2),\ v_1(X_3),\ v_1(X_4),\ v_1(X_5),\ v_1(Y),\\ & \qquad v_2(X_1,X_2),\ v_2(X_2,X_3),\ v_2(X_3,X_4),\ v_2(X_4,X_5),\ v_2(X_5,X_1),\\ & \qquad v_2(X_1,Y).\ v_2(X_2,Y),\ v_2(X_3,Y),\ v_2(X_4,Y),\ v_2(X_5,Y)\end{array}$$

$$\begin{array}{ll}\mathcal{P}_3: & q('yes') :- v_1(X_1),\ v_1(X_2),\ v_1(X_3),\ v_1(X_4),\ v_1(X_5),\ v_1(X_6),\ v_1(X_7),\\ & \qquad v_1(Y),\ v_2(X_1,X_2),\ v_2(X_2,X_3),\ v_2(X_3,X_4),\ v_2(X_4,X_5),\\ & \qquad v_2(X_5,X_6),\ v_2(X_6,X_7), v_2(X_7,X_1),\ v_2(X_1,Y).\ v_2(X_2,Y),\\ & \qquad v_2(X_3,Y),\ v_2(X_4,Y),\ v_2(X_5,Y),\ v_2(X_6,Y), v_2(X_7,Y)\end{array}$$

It follows that query plans $\mathcal{P}_1$, $\mathcal{P}_2$, and $\mathcal{P}_3$ are contained in query $\mathcal{Q}$. More generally, *every* query plan that checks that the input graph contains a not three-colorable subgraph, and that all the vertices in the subgraph are colored by at most three colors, is contained in $\mathcal{Q}$.

It is well known that deciding whether a graph is three-colorable is NP-complete [31]. Because the problem of evaluating a datalog program has polynomial data complexity [56], this shows that there is *no* datalog query plan that contains *all* the query plans that are contained in $\mathcal{Q}$. Intuitively, the reason is that for every datalog query plan $\mathcal{P}$ that is contained in $\mathcal{Q}$, an additional conjunctive query that tests for one more not three-colorable graph can be added to create a query plan that is still contained in $\mathcal{Q}$, but that is not contained in $\mathcal{P}$. □

Example 3.1.2 showed that the expressive power of datalog is insufficient to represent query plans that extract all available data from disjunctive sources. In this chapter, we will present a construction of query plans formulated in *disjunctive datalog with inequality* that do guarantee to extract all data. Example 3.1.3 shows the query plan resulting from our construction when applied to the query planning problem in Example 3.1.2.

Example 3.1.3 Let us continue Example 3.1.2. The disjunctive datalog query plan that contains all query plans contained in query $\mathcal{Q}$ is the following:

$$\begin{array}{ll}\mathcal{P}: & q('yes')\ :-\ v_2(X,Y),\ c(X,Z),\ c(Y,Z)\\ & c(X,red) \vee c(X,green) \vee c(X,blue)\ :-\ v_1(X)\end{array}$$

□

3.2 Preliminaries

3.2.1 Disjunctive datalog

A *disjunctive Horn rule* is an expression of the form

$$p_1(\bar{X}_1) \vee \ldots \vee p_n(\bar{X}_n) :- r_1(\bar{Y}_1),\ \ldots,\ r_m(\bar{Y}_m) \qquad (*)$$

where $p_1, \ldots, p_n$, and $r_1, \ldots, r_m$ are predicate names, and $\bar{X}_1$, ..., $\bar{X}_n$, $\bar{Y}_1$, ..., $\bar{Y}_m$ are tuples of variables, constants, and function terms. The *head* of the rule is $p_1(\bar{X}_1) \vee \ldots \vee p_n(\bar{X}_n)$, and its *body* is $r_1(\bar{X}_1)$, ..., $r_m(\bar{X}_m)$. Every variable in the head of a rule must also occur in the body of the rule. A *disjunctive logic query* is a set of disjunctive Horn rules, and a *disjunctive datalog query* is a set of function-free disjunctive Horn rules. If the predicates that appear in the bodies of the rules are allowed to contain the built-in inequality predicate ($\neq$), then the query language is called disjunctive datalog *with inequality*. Disjunctive Horn rules with inequality have to satisfy the additional constraint that every variable in the body appears at least once in an uninterpreted predicate. A *positive query* is a union of conjunctive queries with the same predicate as head. In this chapter, view definitions, abbreviated as $\mathcal{V}$, are sets of positive queries, and user queries are formulated in datalog.

3.2.2 Semantics

Various semantics have been given to disjunctive datalog queries. The semantic that we are going to present here is commonly known as cautious minimal model semantics [21]. As we will see, disjoint datalog with inequality and cautious minimal model semantics is sufficiently expressive to represent "good" query plans in the presence of disjunctive sources. We will formalize the notion of a "good" query plan in section 3.3.

The input of a disjunctive datalog query $\mathcal{Q}$ consists of a database D storing instances of all EDB predicates in $\mathcal{Q}$. A *model* $\mathcal{M}$ of a query $\mathcal{Q}$ and an input database D, denoted as $\mathcal{M} \models \mathcal{Q}(D)$, is an instance of the predicates in $\mathcal{Q}$ such that

1. $\mathcal{M}$ contains the input database $\mathcal{D}$, and

2. whenever there is an instantiation σ of a rule $(*)$ in $\mathcal{Q}$ such that $r_1(\bar{Y}_1)\sigma$, ..., $r_m(\bar{Y}_m)\sigma$ are in $\mathcal{M}$, then $p_i(\bar{X}_i)\sigma$ is in $\mathcal{M}$ for at least one $i \in \{1, \ldots, n\}$.

We denote the instance of the query predicate q in a model $\mathcal{M}$ by $\mathcal{M}_q$. The output of $\mathcal{Q}$, denoted $\mathcal{Q}(D)$, is the largest instance of the query predicate q that occurs in all models of $\mathcal{Q}$ and D, i.e.

$$\mathcal{Q}(D) = \bigcap_{\mathcal{M} : \mathcal{M} \models \mathcal{Q}(D)} \mathcal{M}_q.$$

For the class of disjunctive queries that we are considering in this chapter, there is an efficient — although in the worst case co-NP-complete — method to compute $\mathcal{Q}(D)$ using conditional tables [30]. However, we are not going to present this evaluation technique here. A query $\mathcal{Q}'$ is *contained* in a query $\mathcal{Q}$ if, for all databases D, $\mathcal{Q}'(D)$ is contained in $\mathcal{Q}(D)$.

3.3 Maximal containment vs. certain answers

In this section we are looking more closely at the question of what makes a query plan a "good" query plan. The most basic requirement on a query plan $\mathcal{P}$ is that it produces answers that are asked for in the corresponding query $\mathcal{Q}$ – and nothing else, i.e. that the expansion of $\mathcal{P}$ is contained in $\mathcal{Q}$. Clearly, two query plans $\mathcal{P}_1$ and $\mathcal{P}_2$ can both satisfy this condition, and still $\mathcal{P}_1$ might be better than $\mathcal{P}_2$ because it might be the case that $\mathcal{P}_1$ always produces more answers than $\mathcal{P}_2$, i.e. $\mathcal{P}_2$ is contained in $\mathcal{P}_1$. In Chapter 2, we therefore focused on the notion of *maximally-contained* query plans.

Definition 3.3.1 (maximal containment) Let $\mathcal{L}$ be the language for representing query plans. Given a query $\mathcal{Q}$ and view definitions $\mathcal{V}$, a *maximally-contained* query plan w.r.t. $\mathcal{Q}$, $\mathcal{V}$, and $\mathcal{L}$, denoted by $max_{\mathcal{Q},\mathcal{V},\mathcal{L}}$, is a query plan that contains all query plans whose expansion is contained in $\mathcal{Q}$, i.e.

$$max_{\mathcal{Q},\mathcal{V},\mathcal{L}} \equiv \bigcup_{\mathcal{P}\in\mathcal{L}\,:\,\mathcal{P}^{exp}\subseteq\mathcal{Q}} \mathcal{P}.$$

□

Depending on the language used for query plans, maximally-contained query plans might not be guaranteed to exist. For example, if query plans are restricted to be formulated in datalog, then no maximally-contained query plan exists for the query and the view definitions in Example 3.1.2. The reason is that there is no (finite) datalog query that is equivalent to the infinite union of conjunctive query plans whose expansion is contained in the user query. Maximally-contained query plans can be found by adding expressive power to the language used to formulate query plans. As seen in Example 3.1.3, moving from datalog to disjunctive datalog as the language for query plans is sufficient to represent a maximally-contained query plan.

The notion of maximal containment depends on the concrete language chosen to represent query plans. Indeed, it would be preferable to have a notion of a "good" query plan that is independent from specific languages. The following definition gives such a characterization.

Definition 3.3.2 (certain answers) Given a query $\mathcal{Q}$ and view definitions $\mathcal{V}$, the function that computes the set of certain answers w.r.t. $\mathcal{Q}$ and $\mathcal{V}$, denoted by $cert_{\mathcal{Q},\mathcal{V}}$, is the function that maps a view instance to the tuples that are in all results of evaluating $\mathcal{Q}$ on databases consistent with the view instance and the view definitions, i.e. for every instance I of the views,

$$cert_{\mathcal{Q},\mathcal{V}}(I) = \bigcap_{D\,:\,I\subseteq\mathcal{V}(D)} \mathcal{Q}(D).$$

□

Maximal containment is a syntactic, proof-theoretic notion. In order to prove that a query plan $\mathcal{P}$ is maximally-contained in a query $\mathcal{Q}$ it is necessary to show that an arbitrarily chosen query plan whose expansion is contained in $\mathcal{Q}$ is also contained in $\mathcal{P}$. On the other hand, the concept of computing certain answers is a semantic, model-theoretic notion. To prove that a function computes all certain answers one has to consider every database that is consistent with the view instance and the view definitions. As in the case of, for example, derivability of a first-order logic formula and its validity, there is also a duality between a query plan being maximally-contained in a query and this query plan computing exactly the certain answers. Theorem 3.3.1 formally states this duality.

Theorem 3.3.1 *Let $\mathcal{Q}$ be a query, let $\mathcal{V}$ be a set of view definitions, and let $\mathcal{L}$ be a language such that $max_{\mathcal{Q},\mathcal{V},\mathcal{L}}$ exists. Then $cert_{\mathcal{Q},\mathcal{V}}$ is contained in $max_{\mathcal{Q},\mathcal{V},\mathcal{L}}$. Moreover, if $\mathcal{L}$ is monotone then $cert_{\mathcal{Q},\mathcal{V}}$ is equivalent to $max_{\mathcal{Q},\mathcal{V},\mathcal{L}}$.*

Proof. Let I be an arbitrary view instance. We have to show that

$$\bigcap_{D\,:\,I\subseteq\mathcal{V}(D)} \mathcal{Q}(D) \;\subseteq\; \bigcup_{\mathcal{P}\in\mathcal{L}\,:\,\mathcal{P}^{exp}\subseteq\mathcal{Q}} \mathcal{P}(I).$$

Let t be a tuple in $\bigcap_{D\,:\,I\subseteq\mathcal{V}(D)} \mathcal{Q}(D)$. Consider the following query plan

$$\mathcal{P}: \quad q(t) \;:-\; v_1(t_{11}),\ \ldots,\ v_1(t_{1k_1}),\ \ldots,\ v_n(t_{n1}),\ \ldots,\ v_n(t_{nk_n})$$

where $t_{11},\ldots,t_{1k_1},\ldots,t_{n1},\ldots,t_{nk_n}$ are the tuples in the view instance I. We know that for every database D with $I\subseteq\mathcal{V}(D)$,

$$\mathcal{P}^{exp}(D) \;=\; \mathcal{P}(\mathcal{V}(D)) \;=\; \{t\} \;\subseteq\; \mathcal{Q}(D),$$

and for every database D with $I\not\subseteq\mathcal{V}(D)$,

$$\mathcal{P}^{exp}(D) \;=\; \mathcal{P}(\mathcal{V}(D)) \;=\; \{\} \;\subseteq\; \mathcal{Q}(D).$$

Therefore, $\mathcal{P}^{exp}\subseteq\mathcal{Q}$. It follows that t is also a tuple in $\bigcup_{\mathcal{P}\in\mathcal{L}\,:\,\mathcal{P}^{exp}\subseteq\mathcal{Q}} \mathcal{P}(I)$.

In order to show equivalence in the case of monotone $\mathcal{L}$, let t be a tuple in $\bigcup_{\mathcal{P}\in\mathcal{L}\,:\,\mathcal{P}^{exp}\subseteq\mathcal{Q}} \mathcal{P}(I)$, and let D be a database with $I\subseteq\mathcal{V}(D)$. There exists at least one query plan $\mathcal{P}$ with $\mathcal{P}^{exp}\subseteq\mathcal{Q}$ and $t\in\mathcal{P}(I)$. Because of the monotonicity of $\mathcal{P}$ we can conclude that

$$\mathcal{P}(I) \;\subseteq\; \mathcal{P}(\mathcal{V}(D)) \;=\; \mathcal{P}^{exp}(D) \;\subseteq\; \mathcal{Q}(D).$$

Therefore, tuple t is also in $\bigcap_{D\,:\,I\subseteq\mathcal{V}(D)} \mathcal{Q}(D)$. □

Theorem 3.3.1 shows that maximally-contained query plans compute exactly the certain answers if the language representing query plans is monotone. The following example shows that $max_{\mathcal{Q},\mathcal{V},\mathcal{L}}$ is not guaranteed to be contained in $cert_{\mathcal{Q},\mathcal{V}}$ if query plans are allowed to be nonmonotone.

Example 3.3.1 Consider the following view definitions and view instances

$$\mathcal{V}: \quad \begin{array}{lcl} v_1(X) & :- & p(X) \\ v_2(X) & :- & p(X) \\ v_2(X) & :- & r(X) \end{array} \qquad\qquad I: \quad \begin{array}{l} v_1 = \{a\} \\ v_2 = \{a, b\} \end{array}$$

and the following query:

$$\mathcal{Q}: \quad q(X) \;:-\; r(X)$$

The expansion of the nonmonotone query plan

$$\mathcal{P}: \quad q(X) \;:-\; v_2(X),\ \neg v_1(X)$$

is contained in $\mathcal{Q}$. Therefore, $b \in max_{\mathcal{Q},\mathcal{V},\mathcal{L}}(I)$. On the other hand, $b \notin cert_{\mathcal{Q},\mathcal{V}}(I)$ because the database D with $p = \{a, b\}$ and $r = \{\}$ satisfies $I \subseteq \mathcal{V}(D)$, and $b \notin \mathcal{Q}(D)$. Therefore, $max_{\mathcal{Q},\mathcal{V},\mathcal{L}}$ is not contained in $cert_{\mathcal{Q},\mathcal{V}}$. □

3.4 Generalization of construction

In this section, we are going to generalize the construction from Section 2.3 so that it can produce maximally-contained query plans in the presence of disjunctive sources. As we have seen in Example 3.1.2, datalog — and any other language with polynomial data complexity — is not sufficiently expressive to represent maximally-contained query plans in this case. Our construction will therefore produce query plans in a more expressive language, namely disjunctive datalog with inequality.

The central part of the construction of maximally-contained query plans is the generalization of inverse rules, introduced in Section 2.3, to *disjunctive inverse rules*. Before we can proceed to this definition, we have to define some technical concepts. Let $\mathcal{Q}_1 \cup \ldots \cup \mathcal{Q}_n$ be a positive view definition with

$$\begin{array}{llcl} \mathcal{Q}_1: & v(\bar{X}_1) & :- & p_{11}(\bar{X}_{11}),\ \ldots,\ p_{1m_1}(\bar{X}_{1m_1}) \\ & & \cdots & \\ \mathcal{Q}_n: & v(\bar{X}_n) & :- & p_{n1}(\bar{X}_{n1}),\ \ldots,\ p_{nm_n}(\bar{X}_{nm_n}). \end{array}$$

We can assume without loss of generality that the sets of variables $\bar{X}_1, \ldots, \bar{X}_n$ are all mutually disjoint. Given a tuple t in an instance of v, we have to determine which of the conjunctive queries $\mathcal{Q}_1, \ldots, \mathcal{Q}_n$ might have generated t. If there is a tuple t such that t can be generated by any of the queries $\mathcal{Q}_{i_1}, \ldots, \mathcal{Q}_{i_k}$, then these queries are called *truly disjunctive*. More formally, queries $\mathcal{Q}_{i_1}, \ldots, \mathcal{Q}_{i_k}$ are called truly disjunctive if there is a substitution σ such that $\bar{X}_{i_1}\sigma = \bar{X}_{i_2}\sigma = \ldots = \bar{X}_{i_k}\sigma$. $\bar{X}_{i_1}\sigma$ is a *witness* of $\mathcal{Q}_{i_1}, \ldots, \mathcal{Q}_{i_k}$ being truly disjunctive.

Let the arity of v be α, and let $\pi_1, \ldots, \pi_\alpha$ be new constants. A conjunction of inequalities φ involving only the new constants $\pi_1, \ldots, \pi_\alpha$ and the constants in $\bar{X}_1, \ldots, \bar{X}_n$ is called an *attribute constraint*. A conjunctive query $\mathcal{Q}_i$ *satisfies* an attribute constraint φ if all inequalities in φ hold after replacing each π_j in φ by the corresponding $\bar{X}_i[j]$. If queries $\mathcal{Q}_{i_1}, \ldots, \mathcal{Q}_{i_k}$ are truly disjunctive with most general witness $\bar{W}$, and there is an attribute constraint φ satisfied by $\bar{X}_{i_1}, \ldots, \bar{X}_{i_k}$, but not satisfied by any $\bar{X}_j$ with $j \in \{1, \ldots, n\} - \{i_1, \ldots, i_k\}$ and $\bar{X}_j$ unifiable with $\bar{W}$, then φ is called an *exclusion condition* for $\mathcal{Q}_{i_1}, \ldots, \mathcal{Q}_{i_k}$.

Example 3.4.1 Let us continue Example 3.1.1. The following is the positive view definition we considered there with head variables renamed appropriately:

$$
\begin{array}{lll}
\mathcal{Q}_1: & v(ua,F_1,T_1) & :- \ flight(ua,F_1,T_1) \\
\mathcal{Q}_2: & v(sw,F_2,T_2) & :- \ flight(sw,F_2,T_2) \\
\mathcal{Q}_3: & v(A_3,sfo,T_3) & :- \ flight(A_3,sfo,S), \ \ flight(A_3,S,T_3)
\end{array}
$$

Here is a list of truly disjunctive queries together with their most general witness and their most general exclusion condition:

$\mathcal{Q}_1$	$\langle ua,F_1,T_1\rangle$	$\pi_2 \neq sfo$
$\mathcal{Q}_2$	$\langle sw,F_2,T_2\rangle$	$\pi_2 \neq sfo$
$\mathcal{Q}_3$	$\langle A_3,sfo,T_3\rangle$	$\pi_1 \neq ua \ \& \ \pi_1 \neq sw$
$\mathcal{Q}_1, \mathcal{Q}_3$	$\langle ua,sfo,T_1\rangle$	*true*
$\mathcal{Q}_2, \mathcal{Q}_3$	$\langle sw,sfo,T_2\rangle$	*true*

This list tells us that a tuple of the form $\langle ua,F,T\rangle$, for example, with $F \neq sfo$ must have been generated by query $\mathcal{Q}_1$, and a tuple of the form $\langle sw,sfo,T\rangle$ must have been generated by either query $\mathcal{Q}_2$ or query $\mathcal{Q}_3$. □

We are now able to define the central concept of disjunctive inverse rules. Intuitively, disjunctive inverse rules describe all the databases that are consistent with the view definitions given a specific view instance.

Definition 3.4.1 (Disjunctive inverse rules) Let

$$
\begin{array}{lll}
\mathcal{Q}_1: & v(\bar{X}_1) & :- \ p_{11}(\bar{X}_{11}), \ \ldots, \ p_{1m_1}(\bar{X}_{1m_1}) \\
 & & \ldots \\
\mathcal{Q}_n: & v(\bar{X}_n) & :- \ p_{n1}(\bar{X}_{n1}), \ \ldots, \ p_{nm_n}(\bar{X}_{nm_n})
\end{array}
$$

be a positive view definition with disjoint sets of head variables $\bar{X}_1, \ldots, \bar{X}_n$, and variables $X_1, \ldots, X_s$ in the bodies but not in $\bar{X}_1, \ldots, \bar{X}_n$. Let $f_1, \ldots, f_s$ be new function symbols. Then for every set of truly disjunctive queries $\mathcal{Q}_{i_1}, \ldots, \mathcal{Q}_{i_k}$ with most general witness $\bar{W}$ and most general exclusion condition φ, the following rules are *disjunctive inverse rules*:

$$p_{i_1\delta_1}(\bar{X}'_{i_1\delta_1}) \vee \ldots \vee p_{i_k\delta_k}(\bar{X}'_{i_k\delta_k}) \ :- \ v(\bar{W}), \ \varphi'$$

with $\delta_l \in \{1, \ldots, m_{i_l}\}$ for $l = 1, \ldots, k$, and

$$
\bar{X}'_{\beta\gamma}[j] = \begin{cases} \bar{W}[j'] & : \ \text{if } (\bar{X}_{\beta\gamma})[j] = \bar{X}_\beta[j'] \text{ for some } j' \\ \bar{X}_{\beta\gamma}[j] & : \ \text{if } (\bar{X}_{\beta\gamma})[j] \text{ is a constant} \\ f_\kappa(\bar{W}) & : \ \text{if } (\bar{X}_{\beta\gamma})[j] = X_\kappa \end{cases}
$$

for all β, γ, j. Condition φ' is generated from φ by replacing each constant π_j in φ by the corresponding variable or constant $\bar{W}[j]$. □

We denote the set of disjunctive inverse rules of a set $\mathcal{V}$ of view definitions by $\mathcal{V}^{-1}$.

Example 3.4.2 The disjunctive inverse rules of the positive view definition in Example 3.4.1 are the following rules:

$$
\begin{array}{ll}
\mathit{flight}(ua,F_1,T_1) & :- \ v(ua,F_1,T_1),\ F_1 \neq sfo \\
\mathit{flight}(sw,F_2,T_2) & :- \ v(sw,F_2,T_2),\ F_2 \neq sfo \\
\mathit{flight}(A_3,sfo,f(A_3,sfo,T_3)) & :- \ v(A_3,sfo,T_3),\ A_3 \neq ua,\ A_3 \neq sw \\
\mathit{flight}(A_3,f(A_3,sfo,T_3),T_3) & :- \ v(A_3,sfo,T_3),\ A_3 \neq ua,\ A_3 \neq sw \\
\mathit{flight}(ua,sfo,T_1) \vee \mathit{flight}(ua,sfo,f(ua,sfo,T_1)) & :- \ v(ua,sfo,T_1) \\
\mathit{flight}(ua,sfo,T_1) \vee \mathit{flight}(ua,f(ua,sfo,T_1),T_1) & :- \ v(ua,sfo,T_1) \\
\mathit{flight}(sw,sfo,T_1) \vee \mathit{flight}(sw,sfo,f(sw,sfo,T_2)) & :- \ v(sw,sfo,T_2) \\
\mathit{flight}(sw,sfo,T_1) \vee \mathit{flight}(sw,f(sw,sfo,T_2),T_2) & :- \ v(sw,sfo,T_2)
\end{array}
$$

□

In the following we will consider the query plan consisting of the rules of a datalog query $\mathcal{Q}$ together with the disjunctive inverse rules $\mathcal{V}^{-1}$. Disjunctive inverse rules contain function symbols. Therefore, the output of a query plan $\mathcal{Q} \cup \mathcal{V}^{-1}$ can contain tuples with function symbols. Given a query plan $\mathcal{P}$ and an instance I, let us denote by $P(I)\downarrow$ the subset of $\mathcal{P}(I)$ that doesn't contain function symbols. As shown in 2.7 for datalog query plans, it is possible to transform a query plan of the form $\mathcal{Q} \cup \mathcal{V}^{-1}$ into a datalog query plan, denoted as $(\mathcal{Q} \cup \mathcal{V}^{-1})\downarrow$, that computes only the tuples without function symbols, i.e.

$$(\mathcal{Q} \cup \mathcal{V}^{-1})(I)\downarrow \quad = \quad (\mathcal{Q} \cup \mathcal{V}^{-1})\downarrow(I)$$

for all instances I. This transformation can easily be generalized to disjunctive datalog query plans.

The following theorem shows that the query plan $(\mathcal{Q} \cup \mathcal{V}^{-1})\downarrow$ is guaranteed to be maximally-contained in $\mathcal{Q}$. The proof of the theorem crucially uses the duality between maximal containment and certain answers discussed in Section 3.3.

Theorem 3.4.1 *For every datalog query $\mathcal{Q}$ and every set of positive view definitions $\mathcal{V}$, the disjunctive datalog query plan $(\mathcal{Q} \cup \mathcal{V}^{-1})\downarrow$ is maximally-contained in $\mathcal{Q}$.*

Proof. Let I be a view instance. Because the $\mathcal{Q}$ part of query plan $\mathcal{Q} \cup \mathcal{V}^{-1}$ does not contain any EDB predicates, and because all the predicates in the bodies of $\mathcal{V}^{-1}$ are EDB predicates, every bottom-up evaluation of $\mathcal{Q} \cup \mathcal{V}^{-1}$ necessarily first has to evaluate $\mathcal{V}^{-1}$ before evaluating $\mathcal{Q}$. Therefore,

$$(\mathcal{Q} \cup \mathcal{V}^{-1})(I) \quad = \quad \bigcap_{\mathcal{M}\,:\,\mathcal{M} \models \mathcal{V}^{-1}(I)} \mathcal{Q}(\mathcal{M}).$$

Since disjunctive datalog queries are monotone, it suffices by Theorem 3.3.1 to show that

$$\underbrace{(\bigcap_{\mathcal{M}\,:\,\mathcal{M}\models\mathcal{V}^{-1}(I)} \mathcal{Q}(\mathcal{M})\,)\downarrow}_{A} \;=\; \underbrace{\bigcap_{D\,:\,I\subseteq\mathcal{V}(D)} \mathcal{Q}(D)}_{B}.$$

Let $\mathcal{M}$ be a model of $\mathcal{V}^{-1}$ and I. By the construction of $\mathcal{V}^{-1}$ we know that $I \subseteq \mathcal{V}(\mathcal{M})$. Therefore, $B \subseteq A$. Because B doesn't contain function symbols it follows that $B \subseteq A\downarrow$.

Let D be a database with $I \subseteq \mathcal{V}(D)$. Consider all the models of $\mathcal{V}^{-1}$ and $\mathcal{V}(D)$. One of these models coincides with D with the only difference that some function symbols in the model are replaced by constants in D. Let $\mathcal{M}$ be this model, and let t be a tuple without function symbols in $\mathcal{M}_q$. Because datalog queries are monotone when constants in the input database are made equal, it follows that $\mathcal{Q}(\mathcal{M})\downarrow \subseteq \mathcal{Q}(D)$. Therefore, $A\downarrow \subseteq B$. □

Theorem 3.4.2 *For every datalog query $\mathcal{Q}$ and every set of positive view definitions $\mathcal{V}$, the disjunctive datalog query plan $(\mathcal{Q} \cup \mathcal{V}^{-1})\downarrow$ can be evaluated in co-NP time (data complexity).*

Proof. Let t be a tuple that is *not* in $(\mathcal{Q} \cup \mathcal{V}^{-1})\downarrow(I)$ for some instance I. Then there is some model $\mathcal{M}$ of $\mathcal{V}^{-1}$ and I such that t is not in $\mathcal{Q}(\mathcal{M})$. If I contains n tuples and the longest conjunct in $\mathcal{V}$ has m literals, then there is a submodel $\mathcal{M}'$ of $\mathcal{M}$ with at most $n \times m$ tuples that is still a model of $\mathcal{V}^{-1}$. Because of the monotonicity of $\mathcal{Q}$, t is also not in $\mathcal{M}'$. Moreover, checking that $\mathcal{M}'$ is a model of $\mathcal{V}^{-1}$, and that t is not in $\mathcal{Q}(\mathcal{M})$ can be done in polynomial time. □

3.5 Conclusions and future work

We considered the problem of answering queries using views with positive view definitions. We showed that datalog is not expressive enough to represent maximally-contained query plans in this case. One the other hand, disjunctive datalog with inequality *is* expressive enough. We presented a construction of maximally-contained query plans in this more expressive language.

The data complexity of evaluating disjunctive datalog queries with inequality in general is co-NP-complete. However, it seems like there are subcases that might allow polynomial time evaluation. The following subcases are likely candidates: (i) $\mathcal{Q}$ has no projections, (ii) $\mathcal{Q}$ is conjunctive and $\mathcal{V}$ has no projections, and (iii) all view definitions in $\mathcal{V}$ have at most two disjuncts. Future work needs to be devoted to look more closely at these subcases.

Chapter 4

Complexity of Answering Queries Using Views

We study the data complexity of the problem of computing certain answers given view definitions, an instance of the views, and a query. We consider conjunctive queries, conjunctive queries with inequality, positive queries, datalog, and first order queries as query and view definition languages. We show that the choice between the assumption that views are complete and the assumption that some tuples might be missing has a considerable impact on the complexity of the problem. Our results imply that in many cases datalog query plans are not expressive enough to answer queries using views in a best possible way.

4.1 Introduction

In Chapter 2 we showed that even for query languages as expressive as datalog it is easy to compute query plans that extract as much information as possible from the views. We called these query plans *maximally-contained* query plans. Here we examine whether it is possible to extend this result to more expressive view definition languages than conjunctive queries, and to other query languages. We show that any query language with polynomial data complexity, for example relational calculus, datalog, or even datalog with well-founded negation, is *not sufficiently expressive* for maximally-contained query plans in these more general cases. For example, a conjunctive query with a single inequality constraint might require maximally-contained query-plans to have more than polynomial data complexity.

We derive these strong negative results by determining the data complexity of the problem whether a given tuple is a certain answer of a query given view definitions and instances of these views. As language for view definitions and queries we examine conjunctive queries, conjunctive queries with inequality, positive queries, datalog, and first order logic.

4.2 Preliminaries

4.2.1 Queries and views

If the body of a conjunctive query is allowed to contain the inequality predicate ($\neq$), then the query is called a *conjunctive query with inequality*. Every variable in a query with inequality must occur at least once in a relational predicate, i.e. in a predicate other than the inequality predicate. A *positive query* is a union of conjunctive queries with the same head predicate. Finally, a *first order query* is a query whose body is a first order formula. A *materialized view*, also called *view instance*, is the stored result of a previously executed query. A query that corresponds to a view instance is called *view definition*. In this chapter we will use the following abreviations:

Conjunctive queries or view definitions:	CQ
Conjunctive queries or view definitions with inequality:	$CQ^{\neq}$
Positive queries or view definitions:	PQ
Datalog queries or view definitions:	$datalog$
First order queries or view definitions:	FO

4.2.2 Open and closed world assumption

Given view definitions $\mathcal{V}$ and an instance I of these views, we are interested in answering queries on the underlying database D. Under the *closed world assumption* we can be sure that instance I stores *all* the tuples that satisfy the view definitions in $\mathcal{V}$, i.e. $I = \mathcal{V}(D)$. Under the *open world assumption*, on the other hand, instance I might only store *some* of the tuples that satisfy the view definitions in $\mathcal{V}$, i.e. $I \subseteq \mathcal{V}(D)$. As we can see from the following example, in reasoning about the underlying database it makes a difference whether we are using the closed or the open world assumption.

Example 4.2.1 Consider the two view definitions

$$v_1(X) :- p(X,Y)$$
$$v_2(Y) :- p(X,Y)$$

and assume that v_1 stores the tuple $\langle a\rangle$ and v_2 stores the tuple $\langle b\rangle$. Under the open world assumption we only know that some p tuple has value a as its first component, and some (possibly different) p tuple has value b as its second component. Under the closed world assumption, however, we can conclude that all p tuples have value a as their first component and value b as their second component, i.e. p contains exactly the tuple $\langle a, b\rangle$. □

Given some view definitions, a corresponding instance of the views, and a query, the question is which answers of the query can be guaranteed to be correct. As we have seen in Example 4.2.1, this question has to be answered differently depending on whether we are assuming an open or a closed world. The following definition formalizes the concept of a certain answer under these two assumptions:

Definition 4.2.1 **(certain answer)** Let $\mathcal{V}$ be a set of view definitions, let I be an instance of the views, and let $\mathcal{Q}$ be a query. A tuple t is a *certain answer under the open world assumption* if t is an element of $\mathcal{Q}(D)$ for every database D with $I \subseteq \mathcal{V}(D)$. A tuple t is a *certain answer under the closed world assumption* if t is an element of $\mathcal{Q}(D)$ for every database D with $I = \mathcal{V}(D)$. □

We will be interested in the *data complexity* [56] of the problem of computing certain answers under the open and the closed world assumption. Data complexity is the complexity of the problem as a function of the size of the instance of the views. We will also refer to the *query complexity* of the problem. Query complexity is the complexity of the problem as a functions of the size of the view definitions $\mathcal{V}$ and the query $\mathcal{Q}$. In the remaining of the chapter, when we discuss complexity, we will always mean data complexity unless specified otherwise.

An answer that is certain under the open world assumption is also a certain answer under the closed world assumption, but not necessarily vice versa. In fact, we will show that computing certain answers under the closed world assumption is harder than under the open world assumption. In Section 4.3 we are going to examine the complexity of the problem of computing certain answers under the open world assumption. Section 4.4 then establishes the complexity results for this problem under the closed world assumption.

4.3 Open world assumption

Figure 4.1 gives an overview of the complexity of computing certain answers under the open world assumption.

			— query —		
views	CQ	$CQ^{\neq}$	PQ	$datalog$	FO
CQ	PTIME	co-NP	PTIME	PTIME	undec.
$CQ^{\neq}$	PTIME	co-NP	PTIME	PTIME	undec.
PQ	co-NP	co-NP	co-NP	co-NP	undec.
$datalog$	co-NP	undec.	co-NP	undec.	undec.
FO	undec.	undec.	undec.	undec.	undec.

Figure 4.1: *Complexity of computing certain answers under the open world assumption.*

Under the open world assumption, the problem of computing certain answers is closely related to the query containment problem. Therefore, decidability and undecidability results carry over in both directions. As shown in Theorem 4.3.1, if the problems are decidable, then their *query complexity* is the same.

Theorem 4.3.1 *Let $\mathcal{L}_1, \mathcal{L}_2 \in \{CQ, CQ^{\neq}, PQ, datalog, FO\}$ be a view definition language and query language respectively. Then the problem of computing certain answers under the open world assumption of a query $\mathcal{Q} \in \mathcal{L}_2$ given view definitions $\mathcal{V} \subseteq \mathcal{L}_1$ and instances of the views is decidable if and* *only if the containment problem of a query in $\mathcal{L}_1$ in a query in $\mathcal{L}_2$ is decidable. Moreover, if the problems are decidable then their query complexity is identical, and the data complexity of the problem of computing certain answers under the open world assumption is at most this query complexity.*

Proof. We establish the claim by giving reductions between the two problems. We start with a reduction from the problem of computing certain answers under the open world assumption to the query containment problem. Let $\mathcal{V} = \{v_1, \ldots, v_k\} \subseteq \mathcal{L}_1$ be a set of view definitions, let $\mathcal{Q} \in \mathcal{L}_2$ be a query, let I be an instance of the views, and let t be a tuple of the same arity as the head of $\mathcal{Q}$. Let $\mathcal{Q}'$ be a query consisting of the rules of the definitions in $\mathcal{V}$ together with the rule

$$q'(t) :- v_1(t_{11}),\ \ldots,\ v_1(t_{1n_1}), \ldots, v_k(t_{k1}),\ \ldots,\ v_k(t_{kn_k})$$

where I is the instance $v_1 = \{t_{11}, \ldots, t_{1n_1}\}$, $\ldots$, $v_k = \{t_{k1}, \ldots, t_{kn_k}\}$. If $\mathcal{L}_1$ is CQ or $CQ^{\neq}$, then the view definitions in $\mathcal{V}$ can be substituted in for the view literals in this new rule. The result is a single conjunctive query. If $\mathcal{L}_1$ is PQ, *datalog*, or FO, then no substitution is necessary. In all cases, $\mathcal{Q}'$ is in $\mathcal{L}_1$. We are going to show that tuple t is a certain answer of $\mathcal{Q}$ given $\mathcal{V}$ and I if and only if $\mathcal{Q}'$ is contained in $\mathcal{Q}$.

"$\Rightarrow$": Assume that t is a certain answer under the open world assumption. Let D be a database. If $I \not\subseteq \mathcal{V}(D)$, then $\mathcal{Q}'(D) = \{\}$, and therefore $\mathcal{Q}'(D)$ is trivially contained in $\mathcal{Q}(D)$. If $I \subseteq \mathcal{V}(D)$, then $\mathcal{Q}'(D) = \{t\}$ and $t \in \mathcal{Q}(D)$. Again, $\mathcal{Q}'(D)$ is contained in $\mathcal{Q}(D)$.

"$\Leftarrow$" Assume that $\mathcal{Q}'$ is contained in $\mathcal{Q}$. Let D be a database with $I \subseteq \mathcal{V}(D)$. Then $\mathcal{Q}'(D) = \{t\}$, and therefore $t \in \mathcal{Q}(D)$. Hence, t is a certain answer.

The remaining part of the proof is a reduction from the query containment problem to the problem of computing certain answers under the open world assumption. Let $\mathcal{Q}_1 \in \mathcal{L}_1$ and $\mathcal{Q}_2 \in \mathcal{L}_2$ be two queries. Let p be a new predicate. Consider as view definition the rules of $\mathcal{Q}_1$ and the additional rule

$$v(c) :- q_1(X),\ p(X)$$

together with the instance I with $v = \{\langle c \rangle\}$. Let the query $\mathcal{Q}$ be all the rules of $\mathcal{Q}_2$ together with the following rule:

$$q(c) :- q_2(X),\ p(X).$$

Again, if $\mathcal{L}_1$ or $\mathcal{L}_2$ are CQ or $CQ^{\neq}$, then the definition of v and query $\mathcal{Q}$ respectively can be transformed into a conjunctive query. Therefore, $v \in \mathcal{L}_1$ and $\mathcal{Q} \in \mathcal{L}_2$. We are going to show that $\mathcal{Q}_1$ is contained in $\mathcal{Q}_2$ if and only if $\langle c \rangle$ is a certain answer of $\mathcal{Q}$ given v and I.

"$\Rightarrow$": Suppose that $\langle c \rangle$ is not a certain answer. Then there is a database D with $I \subseteq v(D)$, and $\mathcal{Q}(D)$ does not contain $\langle c \rangle$. It follows that $\mathcal{Q}_1(D)$ contains a tuple that $\mathcal{Q}_2(D)$ does not contain. Therefore, $\mathcal{Q}_1$ is not contained in $\mathcal{Q}_2$.

"$\Leftarrow$": Assume that $\mathcal{Q}_1$ is not contained in $\mathcal{Q}_2$. Then there is a database D such that $\mathcal{Q}_1(D)$ contains a tuple t that is not contained in $\mathcal{Q}_2(D)$. Database D can be assumed to have $p(D) = \{t\}$. Then $v(D) = I$ and $\mathcal{Q}(D) = \{\}$. Therefore, $\langle c \rangle$ is not a certain answer. □

The previous theorem allows to conclude the query complexity of the problem of computing certain answers under the open world assumption from the query complexity of the corresponding query containment problem. In order to get answers to the question whether maximally-contained datalog query plans can exists, on the other hand, only data complexity results are helpful. Indeed, query complexity results can be misleading. For example, the query complexity of the containment of a conjunctive query in a datalog query is EXPTIME-complete, while the containment problem of a conjunctive query in a conjunctive query with inequality is considerably easier, namely Π_2^p-complete [55]. In comparison, the data complexity of the problem of computing certain answers under the open world assumption for conjunctive view definitions and datalog queries is polynomial, while it is considerably harder, namely co-NP-complete, for conjunctive view definitions and conjunctive queries with inequality.

4.3.1 Conjunctive view definitions

In this section we consider the complexity of the problem of computing certain answers under the open world assumption in the case of conjunctive view definitions. We will consider queries of different expressive power.

Polynomial cases

The main tool for proving polynomial time bounds is the notion of maximally-contained query plans. The relevant definitions can be found in Section 2.2.

Intuitively, a maximally-contained query plan is the best of all query plans in using the information available from the view instances. We showed in Chapter 2 that it is easy to construct these maximally-contained query plans in the case of conjunctive view definitions. Theorem 3.3.1 shows that maximally-contained query plans compute exactly the certain answers under the open world assumption.

As we have shown in Chapter 2, for all $\mathcal{V} \subseteq CQ$ and $\mathcal{Q} \in datalog$, corresponding maximally-contained query plans can be constructed. Because the data complexity of evaluating datalog queries is polynomial [56], it follows that the problem of computing certain answers under the open world assumption can be done in polynomial time.

Corollary 4.3.1 *For $\mathcal{V} \subseteq CQ$ and $\mathcal{Q} \in datalog$, the problem of computing certain answers under the open world assumption can be done in polynomial time.*

Inequality

Theorem 4.3.2 shows that adding inequality just to the view definition doesn't add any expressive power. The certain answers are exactly the same as if the inequalities in the view definitions were omitted. As a consequence, the maximally-contained datalog query plan constructed from the query and the view definitions but disregarding the inequality constraints computes exactly the certain answers. Therefore, the problem is polynomial. On the other hand, Theorem 4.3.3 shows that adding inequality to queries does add expressive power. A single inequality in a conjunctive query, even combined with purely conjunctive view definitions, suffices to make the problem co-NP-hard.

Van der Meyden proved a similar result in [53], namely co-NP hardness for the case $\mathcal{V} \subseteq CQ^{<}$ and $\mathcal{Q} \in CQ^{<}$. Our theorem strengthens this result to $\mathcal{V} \subseteq CQ$ and $\mathcal{Q} \in CQ^{\neq}$.

Theorem 4.3.2 *Let $\mathcal{V} \subseteq CQ^{\neq}$ and $\mathcal{Q} \in$ datalog. Define $\mathcal{V}^{-}$ to be the same view definitions in $\mathcal{V}$ but with the inequality constraints deleted. Then a tuple t is a certain answer under the open world assumption given $\mathcal{V}$, $\mathcal{Q}$ and an instance I of the views if and only if t is a certain answer under the open world assumption given $\mathcal{V}^{-}$, $\mathcal{Q}$ and I.*

Proof.

"$\Rightarrow$": Assume that t is a certain answer under the open world assumption given $\mathcal{V}$, $\mathcal{Q}$ and I. Let D be a database with $I \subseteq \mathcal{V}^{-}(D)$. If also $I \subseteq \mathcal{V}(D)$, then it follows immediately that t is in $\mathcal{Q}(D)$. Otherwise, there is a view definition v in $\mathcal{V}$ and a tuple $s \in I$ such that $s \in v^{-}(D)$, but $s \notin v(D)$. Let $C \neq C'$ be an inequality constraint in v that disabled the derivation of s in $v(D)$. Because we can assume that s is in $v(D')$ for some database D', at least one of C or C' must be an existentially quantified variable X. Add tuples to D that correspond to the tuples that generate s in $v^{-}(D)$, but with the constant that X binds to replaced by a new constant. These new tuples then satisfy the inequality constraint $C \neq C'$. By repeating this process for every such inequality constraint $C \neq C'$ and every such tuple s, we arrive at a database D'' with $I \subseteq \mathcal{V}(D'')$. Because t is a certain answer given $\mathcal{V}$, it follows that t is in $\mathcal{Q}(D'')$. Therefore, there are tuples $t_1, \ldots, t_k \in D''$ that derive t. If any t_i contains one of the new constants, replace it by the tuple $t'_i \in D$ that it was originally derived from. Because t doesn't contain any new constants, and because $\mathcal{Q}$ cannot test for inequality, it follows that t is also derived from $t'_1, \ldots, t'_k$. Hence t is in $\mathcal{Q}(D)$.

"$\Leftarrow$": Assume that t is a certain answer under the open world assumption given $\mathcal{V}^{-}$, $\mathcal{Q}$ and I. Let D be a database with $I \subseteq \mathcal{V}(D)$. Because $\mathcal{V}$ is contained in $\mathcal{V}^{-}$, it follows that $I \subseteq \mathcal{V}^{-}(D)$, and therefore t is in $\mathcal{Q}(D)$. □

The following theorem establishes that the data complexity of the problem of computing certain answers can be non-polynomial (unless P = NP). This increased complexity is due to a single inequality added in the query. The view definitions are purely conjunctive. By Theorem 3.3.1, we know that maximally-contained query plans compute exactly the certain answers under the open world assumption. Because evaluating datalog queries has polynomial data complexity [56], it follows that in general there are *no* datalog query plans that are maximally-contained in a conjunctive query with inequality.

Theorem 4.3.3 *For $\mathcal{V} \subseteq CQ$, $\mathcal{Q} \in CQ^{\neq}$, the problem of determining whether a tuple is a certain answer under the open world assumption given an instance of the views is co-NP-hard.*

Proof. Let φ be a 3CNF formula with variables $x_1, \ldots, x_n$ and conjuncts $c_1, \ldots, c_m$. Consider the conjunctive view definitions

$$\begin{array}{ll} v_1(X,Y,Z) & :-\ p(X,Y,Z) \\ v_2(X) & :-\ r(X,Y) \\ v_3(Y) & :-\ p(X,Y,Z),\ r(X,Z) \end{array}$$

and the instance I of the views with

$$\begin{aligned} v_1 &= \{\langle i,j,1\rangle \mid x_i \text{ occurs in } c_j\} \ \cup \ \{\langle i,j,0\rangle \mid \bar{x}_i \text{ occurs in } c_j\} \\ v_2 &= \{\langle 1\rangle,\ldots,\langle n\rangle\} \\ v_3 &= \{\langle 1\rangle,\ldots,\langle m\rangle\} \end{aligned}$$

and the following query:

$$q(c) :- \ r(X,Y), \ r(X,Y'), \ Y \neq Y'.$$

We are going to show that tuple $\langle c\rangle$ is a certain answer under the open world assumption if and only if formula φ is *not* satisfiable. Because the problem of testing a 3CNF formula for satisfiability is NP-complete [15], this implies the claim.

"$\Rightarrow$": Assume that φ is satisfiable. Then there is an assignment ν from $x_1,\ldots,x_n$ to $true$ and $false$ such that every conjunct of φ contains at least one variable x_i with $\nu(x_i) = true$ or one negated variable $\bar{x}_i$ with $\nu(x_i) = false$. Consider the database D with

$$\begin{aligned} p &= \{\langle i,j,1\rangle \mid x_i \text{ occurs in } c_j\} \ \cup \ \{\langle i,j,0\rangle \mid \bar{x}_i \text{ occurs in } c_j\} \\ r &= \{\langle i,\nu(x_i)\rangle \mid i \in \{1,\ldots,n\}\} \end{aligned}$$

Instance I is contained in $\mathcal{V}(D)$, and $\mathcal{Q}(D)$ doesn't derive $\langle c\rangle$. Therefore, $\langle c\rangle$ is not a certain answer.

"$\Leftarrow$": Assume that $\langle c\rangle$ is not a certain answer. Then there is a database D with $I \subseteq \mathcal{V}(D)$ such that $\mathcal{Q}(D)$ is the empty set. It follows that for $i = 1,\ldots,n$, database D contains exactly one r tuple $\langle i,d_i\rangle$. Consider the assignment ν with $\nu(x_i) = true$ if D contains the r tuple $\langle i,1\rangle$, and with $\nu(x_i) = false$ otherwise. Let c_j be one of the conjuncts. Because $\langle j\rangle$ is contained in v_3, there must be a p tuple $\langle i,j,d_i\rangle$ and an r tuple $\langle i,d_i\rangle$. If $d_i = 1$, then c_j contains a variable x_i with $\nu(x_i) = true$. If $d_i = 0$, then c_j contains a negated variable $\bar{x}_i$ with $\nu(x_i) = false$. Since ν satisfies each c_j, φ is satisfiable. □

So far, we have only proved co-NP-hardness. The following theorem establishes that the problem is indeed solvable in co-NP. Therefore, the problem of computing certain answers under the open world assumption in co-NP-complete for $\mathcal{V} \subseteq CQ$ and $\mathcal{Q} \in CQ^{\neq}$. The theorem applies for the more general case of positive view definitions with inequality ($PQ^{\neq}$), and datalog queries with inequality ($datalog^{\neq}$). For these cases it will therefore suffice later in the chapter to prove co-NP-hardness in order to establish co-NP-completeness.

Theorem 4.3.4 *For $\mathcal{V} \subseteq PQ^{\neq}$, $\mathcal{Q} \in datalog^{\neq}$, the problem of determining whether a tuple is a certain answer under the open or the closed world assumption given an instance of the views is in co-NP.*

Proof. We prove the claim first for the open world assumption. Assume that t is not a certain answer. Then there is a database D with $I \subseteq \mathcal{V}(D)$ and t is not in $\mathcal{Q}(D)$. Let n be the total number of tuples in I and let k be the maximal length of conjuncts in the view definitions. Each tuple in I can be generated by at most k tuples in D. Therefore, there is a database $D' \subseteq D$ with at most nk tuples such that still $I \subseteq \mathcal{V}(D')$. Because t is not in $\mathcal{Q}(D)$ and $\mathcal{Q}$ is monotone, t is also not in

$\mathcal{Q}(D')$. It follows that there is a database D' whose size is polynomially bounded in the size of I and $\mathcal{V}$ such that $I \subseteq \mathcal{V}(D')$, and t is not in $\mathcal{Q}(D')$. Moreover, checking that $I \subseteq \mathcal{V}(D')$ and that t is not in $\mathcal{Q}(D')$ can be done in polynomial time.

For the closed world assumption, the proof is essentially the same with $I = \mathcal{V}(D)$ in place of $I \subseteq \mathcal{V}(D)$. □

First order queries

We saw that adding recursion to positive queries leaves the data complexity of the problem of computing certain answers under the open world assumption still polynomial. On the other hand, adding negation to positive queries makes the problem undecidable, as the following theorem shows.

Theorem 4.3.5 *For $\mathcal{Q} \in FO$, the problem of determining whether a tuple is a certain answer under the open or the closed world assumption given a set of view definitions together with an instance of the views is undecidable, even if the instance is empty.*

Proof. Let ϕ_1 and ϕ_2 be first order formulas. Consider the query

$$q(c) :- \phi_1 \Rightarrow \phi_2.$$

Clearly, $\langle c \rangle$ is a certain answer if and only if ϕ_1 implies ϕ_2. As shown in [7,41], the implication problem for functional and inclusion dependencies is undecidable. Because functional and inclusion dependencies can be formulated in first order logic, this proves the claim. □

4.3.2 Positive view definitions

In the previous section we proved that adding inequality to the query results in co-NP-completeness of the problem of computing certain answers under the open world assumption. The following theorem shows that allowing disjunction in view definitions has the same effect on the data complexity. The same result was proved by van der Meyden in [54] while studying indefinite databases. We include the theorem for the sake of completeness. The proof is a formalization of the ideas presented in Example 3.1.2.

Theorem 4.3.6 *For $\mathcal{V} \subseteq PQ$, $\mathcal{Q} \in CQ$ the problem of determining whether a tuple is a certain answer under the open world assumption given an instance of the views is co-NP-hard.*

Proof. Let $G = (V, E)$ be an arbitrary graph. Consider the view definitions

$$\begin{aligned} v_1(X) &:- color(X, red) \vee color(X, green) \vee color(X, blue) \\ v_2(X,Y) &:- edge(X,Y) \end{aligned}$$

and the instance I with $v_1 = V$ and $v_2 = E$. We will show that the query

$$q(c) :- edge(X,Y),\ color(X,Z),\ color(Y,Z)$$

has the tuple $\langle c\rangle$ as a certain answer if and only if graph G is not 3-colorable. Because testing a graph's 3-colorability is NP-complete [31], this implies the claim. For every database D with $I \subseteq \mathcal{V}(D)$, relation *edge* contains at least the edges from E, and relation *color* relates at least the vertices in V to either *red*, *green*, or *blue*. It follows that the databases D with $I \subseteq \mathcal{V}(D)$ are all the assignments of supersets of the vertex set V to colors such that the vertices in V are assigned to *red*, *green*, or *blue*.

"$\Rightarrow$": Assume that $\langle c\rangle$ is a certain answer of the query. It follows that for every assignment of the vertices to *red*, *green*, and *blue*, there is an edge $\langle e_1, e_2\rangle$ in E such that e_1 and e_2 are assigned to the same color. Therefore, there is not a single assignment of vertices to the three colors *red*, *green*, and *blue* such that all adjacent vertices are assigned to different colors. Hence G is not 3-colorable.

"$\Leftarrow$": Assume G is not 3-colorable. Then for every assignment of supersets of the vertex set V to *red*, *green*, and *blue* there is at least one edge $\langle e_1, e_2\rangle$ such that e_1 and e_2 are assigned to the same color. It follows that the query will produce $\langle c\rangle$ for every database D with $I \subseteq \mathcal{V}(D)$, i.e. the query has $\langle c\rangle$ as a certain answer. □

4.3.3 Datalog view definitions

Theorem 4.3.4 established that the problem can be solved in co-NP for $\mathcal{V} \subseteq PQ^{\neq}$ and $\mathcal{Q} \in datalog^{\neq}$. Here we examine the effect on the complexity of the problem of computing certain answers if we allow datalog as view definition language. For positive queries, the problem stays in co-NP as was shown by van der Meyden in [54]. However, theorems 4.3.7 and 4.3.8 respectively establish that the problem becomes undecidable for conjunctive queries with inequality and datalog queries.

Inequality

In the case of conjunctive view definitions, adding inequality to the query increased the complexity of the problem of computing certain answers under the open world assumption from polynomial to co-NP. With datalog view definitions, adding inequality to the query raises the problem from co-NP complexity to undecidability. In [53], van der Meyden showed undecidability for the case of $\mathcal{V} \subseteq datalog$ and $\mathcal{Q} \in PQ^{\neq}$. The following theorem proves that the problem is already undecidable for *conjunctive* queries with inequality.

Theorem 4.3.7 *For $\mathcal{V} \subseteq datalog$, $\mathcal{Q} \in CQ^{\neq}$, the problem of determining whether a tuple is a certain answer under the open world assumption given an instance of the views is undecidable.*

Proof. The proof is by reduction of the Post Correspondence Problem [42] to the problem in the claim.

Let $w_1, \ldots, w_n, w'_1, \ldots, w'_n$ be words over the alphabet $\{a, b\}$. Consider the following datalog query that defines view v:

$$v(0,0) \;:-\; s(e,e,f)$$
$$v(X,Y) \;:-\; v(X_0,Y_0),\; s(X_0,X_1,\alpha_1),\ldots,s(X_{k-1},X,\alpha_k),$$

$s(Y_0, Y_1, \beta_1), \ldots, s(Y_{l-1}, Y, \beta_l)$
where $w_i = \alpha_1 \ldots \alpha_k$ and $w'_i = \beta_1 \ldots \beta_l$; one rule for each $i \in \{1, \ldots, n\}$.

$$s(X, Y, Z) :- \ p(X, X, Y), \ p(X, Y, Z)$$

and the following query:

$$q(c) :- \ p(X, Y, Z), \ p(X, Y, Z'), \ Z \neq Z'$$

Assume that the instance I of view v is $\{\langle e, e\rangle\}$. We will show that there exists a solution to the instance of the Post Correspondence Problem given by the words $w_1, \ldots, w_n, w'_1, \ldots, w'_n$ if and only if $\langle c\rangle$ is *not* a certain answer under the open world assumption. The result then follows from the undecidability of the Post Correspondence Problem [42].

"$\Rightarrow$": Assume that the instance of the Post Correspondence Problem given by the words $w_1, \ldots, w_n, w'_1, \ldots, w'_n$ has a solution $i_1, \ldots, i_k$. Then $w_{i_1} \ldots w_{i_k} = w'_{i_1} \ldots w'_{i_k} = \gamma_1 \ldots \gamma_m$ for some characters $\gamma_1, \ldots, \gamma_m \in \{a, b\}$. Consider the database D with

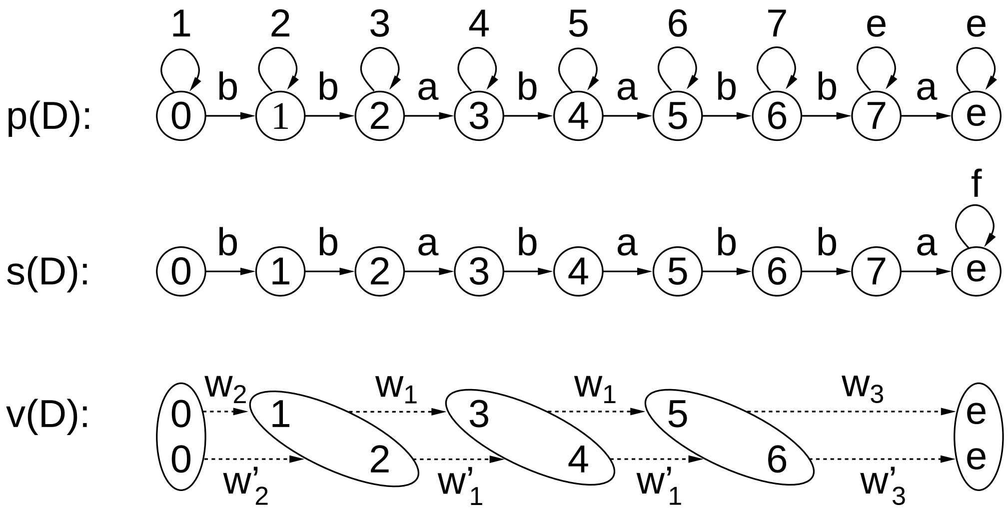


Figure 4.2: *The instance of the Post Correspondence Problem given by the words $w_1 = ba$, $w_2 = b$, $w_3 = bba$, $w'_1 = ab$, $w'_2 = bb$, and $w'_3 = ba$ has solution "2113" because $w_2 w_1 w_1 w_3 = bbababba = w'_2 w'_1 w'_1 w'_3$. The figure shows a database D with $\langle e, e\rangle \in v(D)$, but $\mathcal{Q}(D) = \{\}$.*

$$p(D) = \{\langle 0, 1, \gamma_1\rangle, \ldots, \langle m-2, m-1, \gamma_{m-1}\rangle, \langle m-1, e, \gamma_m\rangle, \langle e, e, f\rangle,$$
$$\langle 0, 0, 1\rangle, \ldots, \langle m-2, m-2, m-1\rangle, \langle m-1, m-1, e\rangle, \langle e, e, e\rangle\}.$$

Clearly, $\mathcal{Q}(D) = \{\}$. Moreover, $s(D)$ and $v(D)$ can be computed to be the following:

$$s(D) = \{\langle 0, 1, \gamma_1\rangle, \ldots, \langle m-2, m-1, \gamma_{m-1}\rangle, \langle m-1, e, \gamma_m\rangle, \langle e, e, f\rangle\}$$

$$\begin{aligned}v(D) = \{&\langle 0, 0\rangle,\\ &\langle |w_{i_1}|, |w'_{i_1}|\rangle,\\ &\langle |w_{i_1}| + |w_{i_2}|, |w'_{i_1}| + |w'_{i_2}|\rangle, \ldots,\\ &\langle |w_{i_1}| + \ldots + |w_{i_{k-1}}|, |w'_{i_1}| + \ldots + |w'_{i_{k-1}}|\rangle,\\ &\langle e, e\rangle\}\end{aligned}$$

Since $I \subseteq v(D)$ and $\mathcal{Q}(D) = \{\}$, it follows that $\langle c\rangle$ is not a certain answer.

"$\Leftarrow$": Assume that $\langle c\rangle$ is not a certain answer under the open world assumption. Then there is a database D with $I \subseteq v(D)$ such that $\mathcal{Q}(D) = \{\}$. Because tuple $\langle e, e\rangle$ is in $v(D)$, there must be constants $c_0, c_1, \ldots, c_m$ with $c_0 = 0$ and $c_m = e$ and characters $\gamma_1, \ldots, \gamma_m \in \{a, b\}$ such that

$$\langle c_0, c_1, \gamma_1\rangle,\ \langle c_1, c_2, \gamma_2\rangle, \ldots, \langle c_{m-1}, c_m, \gamma_m\rangle \in s(D). \qquad (*)$$

Let $d_0, d_1, \ldots, d_{m'}$ be constants with $d_0 = 0$ and $\delta_1, \ldots, \delta_{m'} \in \{a, b\}$ be characters such that

$$\langle d_0, d_1, \delta_1\rangle,\ \langle d_1, d_2, \delta_2\rangle, \ldots, \langle d_{m'-1}, d_{m'}, \delta_{m'}\rangle \in s(D).$$

We are going to show by induction on m' that for $m' \leq m$, $d_i = c_i$ and $\delta_i = \gamma_i$ for $i = 0, \ldots, m'$. The claim is trivially true for $m' = 0$. For the induction case, let $m' > 0$. We know that $\langle c_{i-1}, c_i, \gamma_i\rangle \in s(D)$ and $\langle d_{i-1}, d_i, \delta_i\rangle \in s(D)$, and that $c_{i-1} = d_{i-1}$. By definition of s, this implies that the tuples $\langle c_{i-1}, c_{i-1}, c_i\rangle$, $\langle c_{i-1}, c_{i-1}, d_i\rangle$, $\langle c_{i-1}, c_i, \gamma_i\rangle$, and $\langle c_{i-1}, d_i, \delta_i\rangle$ are all in $p(D)$. Because $\mathcal{Q}(D) = \{\}$, it follows that $d_i = c_i$ and $\delta_i = \gamma_i$.
Assume for the sake of contradiction that $m' > m$. Then there exists a tuple $\langle d_m, d_{m+1}, \gamma_{m+1}\rangle \in s(D)$, and therefore $\langle d_m, d_m, d_{m+1}\rangle, \langle d_m, d_{m+1}, \gamma_{m+1}\rangle \in p(D)$. Because $\langle e, e, f\rangle \in s(D)$, it follows that $\langle e, e, e\rangle, \langle e, e, f\rangle \in p(D)$. Since $d_m = c_m = e$ this implies that $d_{m+1} = e$ and $\gamma_{m+1} = f$, which contradicts the fact that $\gamma_{m+1} \in \{a, b\}$. Hence, $m' = m$.
We proved that there is exactly one chain of the form in (*). Because $\langle e, e\rangle \in v(D)$, there is a sequence $i_1 \ldots i_k$ with $i_1, \ldots, i_k \in \{1, \ldots, n\}$ such that $w_{i_1} \ldots w_{i_k} = \gamma_1 \ldots \gamma_m$ and $w'_{i_1} \ldots w'_{i_k} = \gamma_1 \ldots \gamma_m$. Therefore, $i_1, \ldots, i_k$ is a solution to the instance of the Post Correspondence Problem given by $w_1, \ldots, w_n, w'_1, \ldots, w'_n$.

□

Theorem 4.3.7 has an interesting consequence for the containment problem of a recursive datalog query in a nonrecursive datalog query with inequality. It shows that the technique in [11] to prove decidability of the containment problem of a datalog query in a nonrecursive datalog query does not carry to datalog with inequality. Indeed, it is an easy corollary of Theorems 4.3.1 and 4.3.7 that the problem is undecidable.

Corollary 4.3.2 *The containment problem of a datalog query (even without inequality) in a conjunctive query with inequality is undecidable.*

Datalog queries

As we saw, there is a close relationship between the problem of computing certain answers under the open world assumption and query containment. Not surprisingly it is therefore the case that the problem becomes undecidable for datalog view definitions and datalog queries.

Theorem 4.3.8 *For $\mathcal{V} \subseteq datalog$, $\mathcal{Q} \in datalog$, the problem of determining whether a tuple is a certain answer under the open world assumption given an instance of the views is undecidable.*

Proof. The containment problem of a datalog query in another datalog query is undecidable [47]. Therefore, the claim follows directly from Theorem 4.3.1. □

4.3.4 First order view definitions

Theorem 4.3.5 showed that adding negation in queries leads to undecidability. The following theorem now shows that the same is true for adding negation to view definitions.

Theorem 4.3.9 *For $\mathcal{V} \in FO$, $\mathcal{Q} \in CQ$, the problem of determining whether a tuple is a certain answer under the open or the closed world assumption given an instance of the views is undecidable.*

Proof. Let φ be a first order formula, and let p be a new predicate. Consider the view definition

$$v(c) :- \varphi(X) \vee p(X)$$

together with the instance I with $v = \{\langle c \rangle\}$ and the query

$$q(c) :- p(X).$$

We will show that $\langle c \rangle$ is a certain answer under the open or closed world assumption if and only if formula φ is not satisfiable. A formula φ is not satisfiable if and only if the formula $\neg\varphi$ is a tautology. Since by Church's Theorem [14] testing whether a first order formula is a tautology is undecidable, this implies the claim.

"$\Rightarrow$": Suppose that φ is satisfiable. Then there is a database D such that $\varphi(D)$ is satisfied, and such that $p(D)$ is empty. For this database, $I = v(D)$ and $\mathcal{Q}(D) = \{\}$. Therefore, $\langle c \rangle$ is not a certain answer.

"$\Leftarrow$": Suppose that $\langle c \rangle$ is not certain. Then there is a database D with $I \subseteq \mathcal{V}(D)$ (or with $I = \mathcal{V}(D)$) such that $\langle c \rangle$ is not in $\mathcal{Q}(D)$. Since $p(D)$ is empty, $\varphi(D)$ must be satisfied. Therefore, formula φ is satisfiable. □

4.4 Closed world assumption

Figure 4.3 gives an overview of the complexity of computing certain answers under the closed world assumption. Computing certain answers under the closed world assumption is harder than computing certain answers under the open world assumption. Whereas the problem is polynomial for $\mathcal{V} \subseteq CQ^{\neq}$ and $\mathcal{Q} \in datalog$ under the open world assumption, the problem is already co-NP-complete for $\mathcal{V} \subseteq CQ$ and $\mathcal{Q} \in CQ$ under the closed world assumption. Moreover, whereas the problem is decidable for $\mathcal{V} \subseteq datalog$ and $\mathcal{Q} \in PQ$ under the open world assumption, the problem is already undecidable for $\mathcal{V} \subseteq datalog$ and $\mathcal{Q} \in CQ$ under the closed world assumption.

	— query —				
views	CQ	$CQ^{\neq}$	PQ	$datalog$	FO
CQ	co-NP	co-NP	co-NP	co-NP	undec.
$CQ^{\neq}$	co-NP	co-NP	co-NP	co-NP	undec.
PQ	co-NP	co-NP	co-NP	co-NP	undec.
$datalog$	undec.	undec.	undec.	undec.	undec.
FO	undec.	undec.	undec.	undec.	undec.

Figure 4.3: *Complexity of computing certain answers under the closed world assumption.*

4.4.1 Conjunctive view definitions

As in Theorem 4.3.6, the proof is a reduction of the problem of 3-colorability of a graph to the problem of computing certain answers.

Theorem 4.4.1 *For $\mathcal{V} \subseteq CQ$, $\mathcal{Q} \in CQ$, the problem of determining whether a tuple is a certain answer under the closed world assumption given an instance of the views is co-NP-hard.*

Proof. Let $G = (V, E)$ be an arbitrary graph. Consider the view definitions

$$\begin{aligned} v_1(X) &:- \ color(X, Y) \\ v_2(Y) &:- \ color(X, Y) \\ v_3(X, Y) &:- \ edge(X, Y) \end{aligned}$$

and the instance I with $v_1 = V$, $v_2 = \{red, green, blue\}$ and $v_3 = E$. We will show that under the closed world assumption the query

$$q(c) :- \ edge(X, Y), \ color(X, Z), \ color(Y, Z)$$

has the tuple $\langle c \rangle$ as a certain answer if and only if graph G is not 3-colorable. Because testing a graph's 3-colorability is NP-complete [31], this implies the claim. For every database D with $I = \mathcal{V}(D)$, relation $edge$ contains exactly the edges from E, and relation $color$ relates all vertices in V to either red, $green$, or $blue$.

"$\Rightarrow$": Assume that $\langle c \rangle$ is a certain answer of the query. It follows that for every assignment of the vertices to red, $green$, and $blue$, there is an edge $\langle e_1, e_2 \rangle$ in E such that e_1 and e_2 are assigned to the same color. Therefore, there is not a single assignment of vertices to the three colors red, $green$, and $blue$ such that all adjacent vertices are assigned to different colors. Hence G is not 3-colorable.

"$\Leftarrow$": Assume G is not 3-colorable. Then for every assignment of vertices in V to red, $green$, and $blue$ there is at least one edge $\langle e_1, e_2 \rangle$ such that e_1 and e_2 are assigned to the same color. It follows that the query will produce $\langle c \rangle$ for every database D with $I = \mathcal{V}(D)$, i.e. the query has $\langle c \rangle$ as a certain answer. $\square$

When we were studying the problem of computing certain answers under the open world assumption in Section 4.3, we were able to conclude from a co-NP-hardness result that maximally-contained datalog query plans cannot exist in general. The same reasoning is not true under the closed world assumption. Indeed, we know that for $\mathcal{V} \subseteq CQ$ and $\mathcal{Q} \in CQ$ maximally-contained datalog query plans do exist. By Theorem 3.3.1 we know that these maximally-contained query plans compute exactly the certain answers under the open world assumption. These answers are also certain under the closed world assumption, but there might be certain answers under the closed world assumption that are not computed. The reason why maximally-contained datalog query plans are unable to compute all certain answer under the closed world assumption is the monotonicity of datalog queries. The following example illustrates this point.

Example 4.4.1 Consider the view definitions

$$\begin{array}{ll} v_1(X) & :- \; color(X,Y) \\ v_2(Y) & :- \; color(X,Y) \\ v_3(X,Y) & :- \; edge(X,Y) \end{array}$$

and the query

$$q(c) :- \; edge(X,Y),\; color(X,Z),\; color(Y,Z).$$

Assume that for this choice of $\mathcal{V}$ and $\mathcal{Q}$ there exists a query plan $\mathcal{P}$ that computes exactly the certain answers under the closed world assumption. Then $\mathcal{P}$ outputs $\langle c\rangle$ when applied to an instance I with $v_1 = V$, $v_2 = \{c_1, \ldots, c_k\}$ and $v_3 = E$ for some graph $G = (V, E)$ exactly when G is not k-colorable. Let G be a graph that is not 3-colorable, but that is 4-colorable. Then $\mathcal{P}$ outputs $\langle c\rangle$ for $v_2 = \{c_1, c_2, c_3\}$, but does not output $\langle c\rangle$ for $v_2 = \{c_1, c_2, c_3, c_4\}$. It follows that $\mathcal{P}$ is not monotone. Therefore, $\mathcal{P}$ cannot be a datalog query plan. □

4.4.2 Datalog view definitions

Theorem 4.4.2 *For $\mathcal{V} \subseteq$ datalog, $\mathcal{Q} \in CQ$ the problem of determining whether t is a certain answer under the closed world assumption given an instance of the views is undecidable.*

Proof. Let $\mathcal{Q}_1$ and $\mathcal{Q}_2$ be two datalog queries with answer predicate q_1 and q_2 respectively. Consider the two recursive views

$$\begin{array}{l} v_1(c) :- \; r(X) \\ v_1(c) :- \; q_1(X),\; p(X) \\ v_2(c) :- \; q_2(X),\; p(X) \end{array}$$

where p and r are two relations not appearing in $\mathcal{Q}_1$ and $\mathcal{Q}_2$, and the instance I with $v_1 = \{\langle c\rangle\}$ and $v_2 = \{\}$. Assume the user query is

$$q(c) :- \; r(X).$$

If $\mathcal{Q}_1 \subseteq \mathcal{Q}_2$, then for every database D with $\mathcal{V}(D) = I$,

$$q_1(D) \cap p(D) \subseteq q_2(D) \cap p(D) = v_2(I) = \{\}.$$

Therefore,

$$r(D) = v_1(I) = \{\langle c\rangle\},$$

i.e. $\langle c\rangle$ is a certain answer under the closed world assumption.

On the other hand, if $\mathcal{Q}_1 \not\subseteq \mathcal{Q}_2$, then there is a database D such that some tuple $\langle b\rangle$ is in $\mathcal{Q}_1(D)$, but not in $\mathcal{Q}_2(D)$. By extending D such that $p(D) = \{\langle b\rangle\}$ and $r(D) = \{\}$, we have that $\mathcal{V}(D) = I$. Because $\mathcal{Q}(D) = \{\}$, $\langle c\rangle$ is not a certain answer under the closed world assumption.

We established that $\langle c\rangle$ is a certain answer under the closed world assumption if and only if $\mathcal{Q}_1$ is contained in $\mathcal{Q}_2$. The claim now follows from the undecidability of containment of datalog queries [47]. □

4.5 Conclusions and related work

We established the data complexity of the problem of computing certain answers given view definitions, view instances, and a query, both under the open and the closed world assumption. The query and view definition languages we considered were conjunctive queries, conjunctive queries with inequality, positive queries, datalog, and first order logic. In general, the problem is harder under the closed than under the open world assumption. Under the open world assumption certain answers in the conjunctive view definitions/datalog queries case can be computed in polynomial time. Indeed, datalog queries that compute exactly the certain answers in this case can be easily constructed as shown in Chapter 2 and [17]. On the other hand, already the conjunctive view definitions/conjunctive queries case is co-NP-complete under the closed world assumption. But even under the open world assumption, adding inequalities to the queries, or disjunction to the view definitions makes the problem co-NP-hard. We were able to conclude that, unless P = NP, maximally-contained query plans do not exist if these more expressive query and view definition languages are used.

The problem of computing certain answers under the open world assumption is closely related to the problem of querying indefinite databases. The complexity of this problem was studied by van der Meyden in [53,54,55]. As we showed in this chapter, the *query* complexity of the problem of computing certain answers under the open world assumption is identical to the query complexity of the corresponding query containment problem. The containment problem for conjunctive queries was proved to be NP-complete in [6]. Containment of datalog queries is known to be undecidable [47], and the containment of a datalog query in a non-recursive datalog query was shown to be 2EXPTIME-complete [12].

The problem of answering queries using views has been studied intensively [37,10,17,5,57,44,40]. Its applications range from query optimization to information integration [52,20,16,19,39]. In [37], the query complexity of the problem of answering queries using views was studied for the case of conjunctive queries and conjunctive view definitions, possibly with built-in predicates.

Chapter 5

Query Optimization Using Local Completeness

In this chapter we consider the problem of query plan optimization in information integration systems. It is unrealistic to assume that data stored by information sources is complete. Therefore, current implementations of information integration systems query all possibly relevant information sources in order not to miss any answers. This approach is very costly. We show how a weaker form of completeness, local completeness, can be used to minimize the number of accesses to information sources.

5.1 Introduction

We consider the problem of query plan optimization in information integration. The goal of information integration is to provide the illusion that data stored by distributed information sources is stored in a single "global" database. Users can pose queries in terms of the global database schema. These queries then need to be translated into queries that can be answered by the information sources. There are basically two approaches to information integration. Either the relations of the global schema are defined in terms of the relations stored by the information sources (query-centric approach), or the relations stored by the information sources are described in terms of the global schema (source-centric approach).

The TSIMMIS project [13] investigates the query-centric approach to information integration. Query planning is very efficient using this approach, because user queries simply have to be matched against query templates to find the corresponding predefined query plans. However, the query-centric approach has two major disadvantages. The number of possible user queries is restricted, and adding new information sources requires rewriting all related query templates.

The Information Manifold [33], Infomaster [28], Occam [34], Razor [26], and Emerac [35] follow the more flexible source-centric approach. This approach is very well suited for dynamic environments like the Internet, because adding, removing, or changing an information source only requires adding, removing, or changing the description of this respective information source. Because query plans have to be computed at query time, efficient query plan generation and optimization become crucial.

While query plan generation in the source-centric approach has been studied extensively [37, 44, 38, 39, 17, 19, 20], little work has been done on query plan optimization. We show how *local completeness* information as introduced in [22] and explained in the following can be used for query plan optimization in the source-centric approach to information integration.

5.1.1 Local completeness

Information integration systems communicate with users in terms of a global schema consisting of a set of *world relations* $p_1, p_2, \ldots, p_m$. The system has access to a number of information sources. We refer to these sources as $IS_1, \ldots, IS_n$. Each information source IS_i is assumed to store a *source relation* s_i. Ideally, a source relation would be a materialized view defined in terms of world relations. This view then would concisely describe the data stored by the information source. However, this requirement is seldom satisfiable in real world applications.

Example 5.1.1 Assume an information integration system wants to integrate classified ads from several newspapers. One of the world relations then might be a relation

$$cars\_for\_sale(Manufacturer, Model, Year, Mileage, Price, Phone\_number)$$

representing information found in used car classifieds. Because a specific used car classified can appear in any of the newspapers — the newspapers have overlapping markets — no newspaper is "complete" on some part of the $cars\_for\_sale$ relation. The best one can do in describing the information sources is to state that the data they store is contained in the $cars\_for\_sale$ relation. □

Because frequently source relations do not correspond to materialized views in terms of the world relations, implementations of information integration systems [33,28] consider the data stored by an information source to be *contained* in the corresponding view. Using this interpretation, however, an information integration system might be forced to retrieve much redundant information. If several information sources store data that might be relevant to a user query, then all of these sources need to be queried, although data stored by one information source might be completely stored by another.

In some applications it is impossible to improve on this situation. If for example a user asks for used red sports cars, then there is no way to tell which newspaper might provide matching classifieds. In many application domains, however, it is known that some subset of the data that an information source stores is complete, although the entire data stored by the information source might not be complete. This so called *local completeness information* can be used to minimize the number of information sources that need to be queried. We represent an information source therefore by *two* views: a *conservative view* v_i^c as a lower bound of s_i, and a *liberal view* v_i^l as an upper bound of s_i. Conservative views describe the subsets of the data that are known to be complete, i.e. they encode local completeness information.

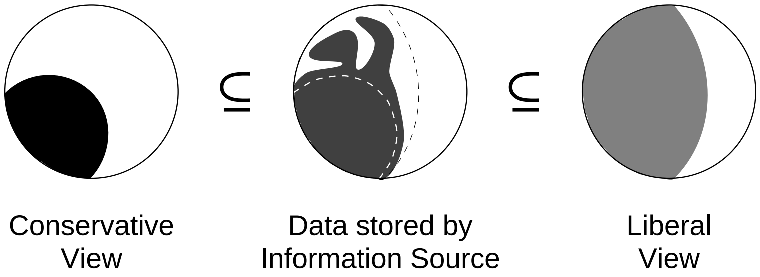


Figure 5.1: *The relationship between data stored by an information source and the corresponding conservative and liberal views. Conservative views represent local completeness information.*

Conservative and liberal views have the same schema as the corresponding source relations. If t is a tuple belonging to the conservative view, then t is indeed stored by the information source. If t is a tuple stored by the information source, then t also belongs to the liberal view. In the special case that an information source indeed stores a materialized view in terms of world relations, the corresponding conservative and liberal views are identical.

Example 5.1.2 Assume that the information integration system also wants to integrate sources that provide information on the current market value of cars. The information integration system might export a world relation like

$$bluebook(Manufacturer, Model, Year, Value).$$

Assume an infomation source stores current market values for cars, and guarantees that it has all information for models built after 1990. This information source can be descibed as follows:

$$v^c_{info}(Ma, Mo, Ye, Va) :- \ bluebook(Ma, Mo, Ye, Va), \ Ye > 1990$$
$$v^l_{info}(Ma, Mo, Ye, Va) :- \ bluebook(Ma, Mo, Ye, Va)$$

A second information source accessible by the information integration system might be a database of the car manufacturer BMW. This database stores information on all BMW models, and nothing else. The completeness of this database can be expressed by coinciding conservative and liberal views:

$$v^{c,l}_{bmw}(bmw, Mo, Ye, Va) :- \ bluebook(bmw, Mo, Ye, Va)$$

If a user requests information on a car build after 1990 or built by BMW, then only information source IS_{info} or IS_{bmw} respectively needs to be queried. On the other hand, if a user asks for all cars with market value over $\$50,000$, then both information sources have to be queried in order not to miss any answers. □

5.1.2 Semantical correctness

In the following, we are going to define three important properties of query plans: semantical correctness, source-completeness, and view-minimality. The notion of maximally-contained query plans, as defined in Chapter 2, corresponds exactly to query plans that are semantically correct and source-complete. We refine the notion of maximally-containment here for the ease of exposition. The algorithms presented in [43, 38, 39, 17, 19] generate semantically correct and source-complete query plans. We will show how these algorithms can be extended to also guarantee view-minimality.

The most basic requirement a query plan $\mathcal{P}$ must satisfy in order to qualify as an answer to a user query $\mathcal{Q}$ is that every tuple reported to the user by executing $\mathcal{P}$ does satisfy $\mathcal{Q}$. A query plan $\mathcal{P}$ is *semantically correct* with respect to a user query $\mathcal{Q}$, if $\mathcal{P}$ is contained in $\mathcal{Q}$ for all instances of the source relations $s_1, \ldots, s_n$ consistent with the given conservative and liberal views.

Example 5.1.3 Assume a user asks for used cars built in 1991 that are offered for sale below their current market value:

$$q(Ma, Mo, Mi, Pr, Ph) :- \; cars\_for\_sale(Ma, Mo, 1991, Mi, Pr, Ph), \\ bluebook(Ma, Mo, 1991, Va), \;\; Pr < Va$$

In addition to the two information sources in Example 5.1.2, the information integration system might have access to the used car classifieds of the San Francisco Chronicle and the San Jose Mercury News. These two information sources don't guarantee any local completeness, and are therefore only described by the following liberal views:

$$v^l_{sfc}(Ma, Mo, Ye, Mi, Pr, Ph) :- \; cars\_for\_sale(Ma, Mo, Ye, Mi, Pr, Ph)$$
$$v^l_{sjmn}(Ma, Mo, Ye, Mi, Pr, Ph) :- \; cars\_for\_sale(Ma, Mo, Ye, Mi, Pr, Ph)$$

The query plan

$$q_1(Ma, Mo, Mi, Pr, Ph) :- \; s_{sfc}(Ma, Mo, 1991, Mi, Pr, Ph), \\ s_{bmw}(Ma, Mo, 1991, Va), \;\; Pr < Va$$

is semantically correct with respect to $\mathcal{Q}$. It is essential to add the selection on year and price range in order for the query to be semantically correct. We assume that information sources have the capability of equality selection. Therefore, the selection of used cars built in 1991 can be pushed to the sources. However, the selection on the price range needs to be added as a post-processing step in the information integration system. □

5.1.3 Source-completeness

A user will hardly be satisfied by an answer from an information integration system that is guaranteed merely to be semantically correct. For example, answering with the empty set is always semantically correct. Indeed, users require that they obtain all information from the system that they could get by manually checking the sources. The notion of source-completeness formalizes this demand for a "best possible" query plan. A query plan $\mathcal{P}$ is *source-complete* if every semantically correct query plan $\mathcal{P}'$ is contained in $\mathcal{P}$ for all instances of the source relations $s_1, \ldots, s_n$ consistent with the given conservative and liberal views.

Example 5.1.4 The query plan in Example 5.1.4 is *not* source-complete. A used BMW offered for sale in the San Jose Mercury News, for example, will not be contained in the answer of $\mathcal{P}_1$ although it might be in the answer of the query plan

$$\begin{aligned} q_2(Ma, Mo, Mi, Pr, Ph) :- \; & s_{sjmn}(Ma, Mo, 1991, Mi, Pr, Ph), \\ & s_{bmw}(Ma, Mo, 1991, Va), \;\; Pr < Va. \end{aligned}$$

The union of the query plans $\mathcal{P}_1$ and $\mathcal{P}_2$ is still not source-complete, because all cars in the answer are manufactured by BMW. Information source IS_{info} can be used to also consider cars of other manufacturers:

$$\begin{aligned} q_3(Ma, Mo, Mi, Pr, Ph) :- \; & s_{sfc}(Ma, Mo, 1991, Mi, Pr, Ph), \\ & s_{info}(Ma, Mo, 1991, Va), \;\; Pr < Va. \\ q_4(Ma, Mo, Mi, Pr, Ph) :- \; & s_{sfmn}(Ma, Mo, 1991, Mi, Pr, Ph), \\ & s_{info}(Ma, Mo, 1991, Va), \;\; Pr < Va. \end{aligned}$$

The query plan returns the union of $\mathcal{P}_1$, $\mathcal{P}_2$, $\mathcal{P}_3$, and $\mathcal{P}_4$ is indeed source-complete. □

5.1.4 View-minimality

Getting a semantically correct and source-complete answer is the main concern of the user. The information integration system, however, needs also to be concerned with the cost of coming up with this answer. By just executing *all* semantically correct query plans, assuming there is only a finite number, and reporting the union of all answers to the user, the given answer would be guaranteed to be source-complete. On the other hand, much information might be retrieved redundantly from several information sources. A query plan requiring considerably fewer information sources might still be source-complete. A query plan $\mathcal{P}$ is *view-minimal* if every semantically correct and source-complete query plan $\mathcal{P}'$ queries at least as many information sources as $\mathcal{P}$.

Example 5.1.5 Because IS_{info} is guaranteed to store all information for cars built after 1990, there is no information in the BMW database for cars built in 1991 that could not be found in IS_{info}. Therefore, query plans $\mathcal{P}_1$ and $\mathcal{P}_2$ are redundant. The query plan that is semantically correct, source-complete, and view-minimal is the union of $\mathcal{P}_3$ and $\mathcal{P}_4$. □

5.2 Computing with source descriptions

In this chapter, liberal views are conjunctive queries. Conservative views and user queries can be unions of conjunctive queries. Given conservative and liberal views $v_1^c, \ldots, v_n^c$, $v_1^l, \ldots, v_n^l$ and a user query $\mathcal{Q}$, the goal is to generate a query $\mathcal{P}$ that satisfies the following four conditions:

- *(query plan)* $\mathcal{P}$ uses only $s_1, \ldots, s_n$ and built-in predicates.

- *(semantical correctness)* For every database D over the world relations and every instance I of the source relations with $v^c(D) \subseteq I \subseteq v^l(D)$, $\mathcal{P}(I)$ is contained in $\mathcal{Q}(D)$.
- *(source-completeness)* For every database D over the world relations and every instance I of the source relations with $v^c(D) \subseteq I \subseteq v^l(D)$, $\mathcal{P}'(I)$ is contained in $\mathcal{P}(I)$ for every semantically correct query plan $\mathcal{P}'$.
- *(view-minimality)* $|\mathcal{P}| \leq |\mathcal{P}''|$ for every semantically correct and source-complete query plan $\mathcal{P}''$.

Here $|\mathcal{P}|$ denotes the number of information sources required in the query plan $\mathcal{P}$.

The test of both semantical correctness and source-completeness is relative to instances I of the source relations with the restriction that $v^c(D) \subseteq I \subseteq v^l(D)$ The only way to effectively test semantical correctness and source-completeness is to use the descriptions of the source relations given by the conservative and liberal views. We therefore have to develop criteria for semantical correctness and source-completeness that do not refer to a restricted set of instances of the source relations.

5.2.1 Syntactic criterion for semantical correctness

In order to test semantical correctness it is necessary to test containment of a query plan in a user query. Query plans are formulated in terms of source relations. User queries, on the other hand, are formulated in terms of world relations. In order to compare queries in different languages, we have to translate one into the language of the other. Because conservative and liberal views express source relations in terms of world relations, it is easier to translate query plans into the world schema. Let us denote a query plan $\mathcal{P}$ requiring source relations $s_{i_1}, \ldots, s_{i_k}$ as $\mathcal{P}[s_{i_1}, \ldots, s_{i_k}]$. Replacing each occurrence of s_{i_j} by the corresponding body of the definition of $v^l_{i_j}$ yields $\mathcal{P}[v^l_{i_1}, \ldots, v^l_{i_k}]$ which we denote as $\mathcal{P}[s \mapsto v^l]$.

Example 5.2.1 If $\mathcal{P}$ is the query plan

$$q(X,Y) :- \ s_1(X,Z),\ s_2(Z,Y,c),\ X < 100$$

and the liberal views corresponding to source relations s_1 and s_2 are

$$\begin{aligned} v^l_1(X,Y) \quad &:-\ p_1(X,Y,Z,d) \quad \text{and} \\ v^l_2(X,Y,Z) &:-\ p_2(X,Y),\ p_3(Y,Z), \end{aligned}$$

then $\mathcal{P}[s \mapsto v^l]$ denotes the query

$$q[s \mapsto v^l](X,Y) :- \ p_1(X,Z,Z',d),\ p_2(Z,Y),\ p_3(Y,c),\ X < 100.$$

□

Semantical correctness requires that the answer of a query plan is guaranteed to satisfy the given user query. Because the source relations themselves are unknown to the information integration system, they must be assumed to be possibly as large as indicated by the liberal views. This intuition motivates the following syntactic characterization of semantical correctness:

Theorem 5.2.1 *A query plan $\mathcal{P}$ is semantically correct with respect to $\mathcal{Q}$ if and only if $\mathcal{P}[s \mapsto v^l]$ is contained in $\mathcal{Q}$.*

Proof. If $\mathcal{P}$ is semantically correct with respect to $\mathcal{Q}$, then for every database D over the world relations, $\mathcal{P}(I) \subseteq \mathcal{Q}(D)$ for the instance $I = v^l(D)$ of the source relations, and therefore $\mathcal{P}[s \mapsto v^l] \subseteq \mathcal{Q}$. If $\mathcal{P}$ is not semantically correct with respect to $\mathcal{Q}$, then there is a database D over the world relations, an instance I of the source relations with $v^c(D) \subseteq I \subseteq v^l(D)$, and a tuple t in $\mathcal{P}(I)$ that is not in $\mathcal{Q}(D)$. Since unions of conjunctive queries are monotone, adding tuples to I until $I = v^l(D)$ will not delete t from $\mathcal{P}(I)$. Therefore, $\mathcal{P}[s \mapsto v^l]$ is not contained in $\mathcal{Q}$. □

One should notice that Theorem 5.2.1 fails if we allow non-monotone queries. Consider for example the query

$$q(X) :- p_1(X),\ \neg p_2(X)$$

and assume source relation s and s' are bounded by the liberal views

$$v_1^l(X) :- p_1(X) \quad \text{and}$$
$$v_2^l(X) :- p_2(X)$$

respectively. The query plan

$$q(X) :- s_1(X),\ \neg s_2(X)$$

satisfies $\mathcal{P}[s \mapsto v^l] \subseteq \mathcal{Q}$. But consider the instance with $p_1 = p_2 = s_1 = \{\langle a \rangle\}$ and $s_2 = \{\}$. Whereas $\mathcal{Q}$ would return no answer, $\mathcal{P}$ returns the answer a. Therefore, $\mathcal{P}$ is not semantically correct.

5.2.2 Syntactic criterion for source-completeness

Intuitively, a query plan $\mathcal{P}$ is source-complete if all information asked for by a user query and available from source relations is retrieved. No other semantically correct query plan should be able to retrieve more information than $\mathcal{P}$. We were able to formulate a syntactic criterion for semantical correctness by replacing all occurrences of source relations by their liberal descriptions. One might hope to find a similar criterion for source-completeness. If $\mathcal{P}'[s \mapsto s^l]$ is contained in $\mathcal{P}[s \mapsto v^c]$, then $\mathcal{P}'$ does not need to be retrieved, because all information that might possibly be retrieved using $\mathcal{P}'$ is guaranteed to be retrieved using $\mathcal{P}$. This observation suggests that source-completeness of $\mathcal{P}$ might be equivalent to the condition "$\mathcal{P}'[s \mapsto v^l]$ is contained in $\mathcal{P}[s \mapsto v^c]$ for all semantically correct query plans $\mathcal{P}'$". This condition is sufficient for source-completeness. It is not necessary, however, as can be seen from the following example.

Example 5.2.2 Assume there are three source relations described by the following conservative and liberal views:

$$v_1^c(X) :- p_1(X),\ p_2(X, a)$$
$$v_1^l(X) :- p_1(X),\ p_2(X, Y)$$

$$v_2^c(X) :- p_1(X),\ p_2(X,Y)$$
$$v_2^l(X) :- p_1(X)$$
$$v_3^c(X) :- p_3(X,a)$$
$$v_3^l(X) :- p_3(X,Y)$$

Given the user query

$$q(X) :- p_1(X),\ p_3(X,Z)$$

the two query plans

$$q_1(X) :- s_1(X),\ s_3(X) \quad \text{and}$$
$$q_2(X) :- s_2(X),\ s_3(X)$$

are semantically correct with respect to $\mathcal{Q}$. Query plans $\mathcal{P}_2$ is source-complete, because source relation s_1 is guaranteed to be contained in source relation s_2. However,

$$q_1[s \mapsto v^l](X) :- p_1(X),\ p_2(X,Y),\ p_3(X,Z)$$

is not contained in

$$q_2[s \mapsto v^c](X) :- p_1(X),\ p_2(X,Y),\ p_3(X,a).$$

□

The problem in Example 5.2.2 is that source relation s_3 appears both in $\mathcal{P}_1$ and in $\mathcal{P}_2$. Although s_3 is of course contained in s_3, v_3^l is not contained in v_3^c. A small variation on this idea, however, provides us with a syntactic criterion for source-completeness. If $\mathcal{P}$ is a query plan, then let $\mathcal{P}[s \mapsto s \wedge v^l]$ be the result of replacing every source relation s_i in $\mathcal{P}$ with the conjunction of s_i and the corresponding body of the definition of v_i^l. Let $\mathcal{P}[s \mapsto s \vee v^c]$ be the result of replacing every source relation s_i in $\mathcal{P}$ with the disjunction of s_i and the corresponding body of the definition of v_i^c.

Example 5.2.3 Continuing with Example 5.2.2, $\mathcal{P}_1[s \mapsto s \wedge v^l]$ denotes the query

$$q_1[s \mapsto s \wedge v^l](X) :- s_1(X),\ s_3(X),\ p_1(X)\ p_2(X,Y),\ p_3(X,Z)$$

and $\mathcal{P}_2[s \mapsto s \vee v^c]$ denotes the query $\mathcal{P}_{21} \cup \mathcal{P}_{22} \cup \mathcal{P}_{23} \cup \mathcal{P}_{24}$ with

$$q_{21}(X) :- s_2(X),\ s_3(X)$$
$$q_{22}(X) :- s_2(X),\ p_3(X,a)$$
$$q_{23}(X) :- p_1(X),\ p_2(X,Y),\ s_3(X)$$
$$q_{24}(X) :- p_1(X),\ p_2(X,Y),\ p_3(X,a)$$

Because $\mathcal{P}_1[s \mapsto s \wedge v^l]$ is contained in $\mathcal{P}_{23}$, $\mathcal{P}_1[s \mapsto s \wedge v^l]$ is contained in $\mathcal{P}_2[s \mapsto s \vee v^c]$. As we will show in the following theorem, this implies that $\mathcal{P}_1$ is redundant. □

Although both $\mathcal{P}[s \mapsto s \wedge v^l]$ and $\mathcal{P}[s \mapsto s \vee v^c]$ still contain source relations, we can use these two notions for a syntactic criterion for source-completeness. The reason is that instance I is no longer constrained to satisfy $v^c(D) \subseteq I \subseteq v_i^l(D)$, but can be chosen arbitrarily. This means that each source relation can be treated as just another world relation, and containment can be tested without referring to a restricted set of instances of the source relations.

Theorem 5.2.2 *A query plan $\mathcal{P}$ is source-complete if and only if for every query plan $\mathcal{P}'$ with $\mathcal{P}'[s \mapsto v^l] \subseteq \mathcal{Q}$, $\mathcal{P}'[s \mapsto s \wedge v^l]$ is contained in $\mathcal{P}[s \mapsto s \vee v^c]$.*

Proof. Let $\mathcal{P}$ be a source-complete query plan, and assume $\mathcal{P}'$ is a query plan with $\mathcal{P}'[s \mapsto v^l] \subseteq \mathcal{Q}$. Let D be an arbitrary database over the world relations and I an arbitrary instance of the source relations, and let I' be $(I \cup v^c(D)) \cap v^l(D)$. Then I' satisfies $v^c(D) \subseteq I' \subseteq v^l(D)$. Because by Theorem 5.2.1, $\mathcal{P}'$ is semantically correct it follows that $\mathcal{P}'[s \mapsto s \wedge v^l](D, I) \subseteq \mathcal{P}'(I') \subseteq \mathcal{P}(I') \subseteq \mathcal{P}[s \mapsto s \vee v^c](D, I)$. Therefore, $\mathcal{P}'[s \mapsto s \wedge v^l]$ is contained in $\mathcal{P}[s \mapsto s \vee v^c]$. For the opposite direction, assume $\mathcal{P}'[s \mapsto s \wedge v^l]$ is contained in $\mathcal{P}[s \mapsto s \vee v^c]$ for every query plan with $\mathcal{P}'[s \mapsto v^l] \subseteq \mathcal{Q}$. Let $\mathcal{P}''$ be a semantically correct query plan and let D be a database over the world relations and I be an instance of the source relations with $v^c(D) \subseteq I \subseteq v^l(D)$. By Theorem 5.2.1, $\mathcal{P}''[s \mapsto v^l] \subseteq \mathcal{Q}$ and therefore $\mathcal{P}''(I) \equiv \mathcal{P}''[s \mapsto s \wedge v^l](D, I) \subseteq \mathcal{P}[s \mapsto s \vee v^c](D, I) \equiv \mathcal{P}(I)$. Therefore, $\mathcal{P}$ is source-complete. $\square$

5.3 Query plan optimization

In general, there will be infinitely many query plans $\mathcal{P}'$ with $\mathcal{P}'[s \mapsto v^l] \subseteq \mathcal{Q}$. It therefore seems as if the criterion for source-completeness is not effective. However, it is sufficient to only consider the finite number of conjunctive query plans generated by the algorithm in, for example, [43]. Applied to a conjunctive query $\mathcal{Q}$ and the liberal views $v_1^l, \ldots, v_n^l$, Qian's algorithm produces a set of conjunctive query plans, denoted $folding(\mathcal{Q}, v^l)$, with the following properties:

1. $\mathcal{P}[s \mapsto v^l] \subseteq \mathcal{Q}$ for every $\mathcal{P} \in folding(\mathcal{Q}, v^l)$.
2. For every conjunctive query plan $\mathcal{P}'$ with $\mathcal{P}'[s \mapsto v^l] \subseteq \mathcal{Q}$, $\mathcal{P}' \subseteq \mathcal{P}$ for some $\mathcal{P} \in folding(\mathcal{Q}, v^l)$.

By Theorem 5.2.1, the first property guarantees that each conjunctive query plan in $folding(\mathcal{Q}, v^l)$ is semantically correct with respect to $\mathcal{Q}$. Therefore, the union of all conjunctive query plans in $folding(\mathcal{Q}, v^l)$, denoted $\bigcup folding(\mathcal{Q}, v^l)$, is semantically correct. The second property states that for every semantically correct conjunctive query plan $\mathcal{P}'$ and *every* instance I of the source relations, $\mathcal{P}'(I)$ is contained in $\bigcup folding(\mathcal{Q}, v^l)(I)$. It follows that specifically for instances I satisfying $v^c(D) \subseteq I \subseteq v^l(D)$ for some database D over the world relations, $\mathcal{P}'(I)$ is contained in $\bigcup folding(\mathcal{Q}, v^l)(I)$. Therefore $\bigcup folding(\mathcal{Q}, v^l)$ is source-complete. The second property has a further implication. A query $\mathcal{P}'$ that is contained in a query $\mathcal{P}$ requires the same source relations as $\mathcal{P}$, and possibly more. It follows that there is a semantically correct, source-complete, and view-minimal query plan of the form $\bigcup_{i \in I} \mathcal{P}_i$ with $\mathcal{P}_i \in folding(\mathcal{Q}, v^l)$ for all $i \in I$.

Information integration systems that do not have any local completeness information have to retrieve a query equivalent to $\bigcup folding(\mathcal{Q}, v^l)$ in order to guarantee source-completeness. However, $\bigcup folding(\mathcal{Q}, v^l)$ is in general not view-minimal. By using the local completeness information given by the conservative views, conjunctive queries can be removed from $folding(\mathcal{Q}, v^l)$ without losing source-completeness. The following theorem gives the crucial criterion for identifying the proper subset of $folding(\mathcal{Q}, v^l)$ that is both source-complete and view-minimal.

Theorem 5.3.1 *If $\bigcup_{j \in J} \mathcal{P}_j$ is a semantically correct query plan that contains every semantically correct query plan, then for every $I \subseteq J$ satisfying*

$$\bigcup_{j \in J-I} \mathcal{P}_j[s \mapsto s \wedge v^l] \subseteq \bigcup_{i \in I} \mathcal{P}_i[s \mapsto s \vee v^c], \qquad (*)$$

the query plan $\bigcup_{i \in I} \mathcal{P}_i$ is source-complete. Moreover, if I is chosen as the set minimizing the number of information sources required in $\bigcup_{i \in I} \mathcal{P}_i$ and satisfying (*), then $\bigcup_{i \in I} \mathcal{P}_i$ is view-minimal.

Proof. Because $\bigcup_{j \in J} \mathcal{P}_j$ contains all semantically correct query plans we have

$$\begin{aligned}
\mathcal{P}'(I) &\subseteq \bigcup_{j \in J} \mathcal{P}_j(I) \\
&\equiv \bigcup_{j \in J-I} \mathcal{P}_j[s \mapsto s \wedge v^l](D, I) \cup \bigcup_{i \in I} \mathcal{P}_i[s \mapsto s \vee v^c](D, I) \\
&\subseteq \bigcup_{i \in I} \mathcal{P}_i[s \mapsto s \vee v^c](D, I) \\
&\equiv \bigcup_{i \in I} \mathcal{P}_i(I)
\end{aligned}$$

for all databases D and I over the world and source relations respectively with $v^c(D) \subseteq I \subseteq v^l(D)$. Therefore, $\bigcup_{i \in I} \mathcal{P}_i$ is source-complete.

Let $\bigcup_{k \in K} \mathcal{P}'_k$ be a union of conjunctive query plans that is semantically correct, source-complete, and view-minimal. Because $\bigcup_{j \in J} \mathcal{P}_j$ contains all semantically correct query plans, it follows from [46] that for every $k \in K$ there is a $j_k \in J$ such that $\mathcal{P}'_k$ is contained in $\mathcal{P}_{j_k}$. Therefore, for each $k \in K$ there is a containment mapping from $\mathcal{P}_{j_k}$ to $\mathcal{P}'_k$. This implies that for each $k \in K$, all source relations of $\mathcal{P}_{j_k}$ are also required in $\mathcal{P}_k$. We established that $\bigcup_{k \in K} \mathcal{P}_{j_k}$ contains $\bigcup_{k \in K} \mathcal{P}'_k$ and requires at most as many source relations as $\bigcup_{k \in K} \mathcal{P}'_k$. Hence, $\bigcup_{k \in K} \mathcal{P}_{j_k}$ is source-complete and view-minimal. $\bigcup_{k \in K} \mathcal{P}_{j_k}$ is also semantically correct because it is contained in $\bigcup_{j \in J} \mathcal{P}_j$ which is semantically correct. The set $I' = \{j_k | k \in K\}$ is a subset of J and satisfies condition (*) by Theorem 5.2.2. Because $\bigcup_{i \in I} \mathcal{P}_i$ requires at most as many source relations as $\bigcup_{i \in I'} \mathcal{P}_i$, and $\bigcup_{i \in I'} \mathcal{P}_i$ is view-minimal, it follows that $\bigcup_{i \in I} \mathcal{P}_i$ is also view-minimal. □

Theorem 5.3.1 suggests an algorithm for finding semantically correct, source-complete and view-minimal query plans given *conjunctive* user queries. First, compute $folding(\mathcal{Q}, v^l)$, and then find a subset $\mathcal{R}$ of $folding(\mathcal{Q}, v^l)$ requiring the least number of source relations such that

$$\bigcup(folding(\mathcal{Q}, v^l) - R)[s \mapsto s \wedge v^l]$$

is contained in $\bigcup R[s \mapsto s \vee v^c]$. Then $\bigcup R$ is the desired query plan.

The conjunctive case can be easily generalized to handle unions of conjunctive queries as user queries. User queries then are of the form

$$q \equiv \bigcup_{k \in K} \mathcal{Q}_k$$

where the $\mathcal{Q}_k$'s are conjunctive queries in terms of world relations. In this case,

$$\bigcup_{k \in K} (\bigcup folding(\mathcal{Q}_k, v^l))$$

is semantically correct with respect to $\mathcal{Q}$ and contains all query plans that are semantically correct with respect to $\mathcal{Q}$. Again, Theorem 5.3.1 can be applied to find a subset R of the set of conjunctive query plans in this union such that $\bigcup R$ is source-complete and view-minimal. The general optimization algorithm is shown in Figure 5.2.

Input:
- User query $\mathcal{Q}$. The query can be a union of conjunctive queries.
- Liberal views $v_1^l, \ldots, v_n^l$. View definitions are conjunctive queries.
- Conservative views $v_1^c, \ldots, v_n^c$. View definitions are unions of conjunctive queries.

Output: Semantically correct, source-complete, and view-minimal query plan for $\mathcal{Q}$.

(1) Convert $\mathcal{Q}$ into the form $\bigcup_{k \in K} \mathcal{Q}_k$ where the $\mathcal{Q}_k$'s are conjunctive queries.

(2) For each $k \in K$, compute $folding(\mathcal{Q}_k, v^l)$.

(3) Let $\bigcup_{j \in J} \mathcal{P}_j$ be the union of all conjunctive query plans $\mathcal{P}_j$ with $\mathcal{P}_j \in folding(\mathcal{Q}_k, v^l)$, for all $k \in K$.

(4) Let $I^* \subseteq J$ be a set minimizing the number of information sources required in $\bigcup_{i \in I^*} \mathcal{P}_i$ such that $\bigcup_{i \in J - I^*} \mathcal{P}_i[s \mapsto s \wedge v^l] \subseteq \bigcup_{i \in I^*} \mathcal{P}_i[s \mapsto s \vee v^c]$;

(5) return($\bigcup_{i \in I^*} \mathcal{P}_i$);

Figure 5.2: *An algorithm for generating semantically correct, source-complete and view-minimal query plans.*

5.4 Conclusions and related work

We considered the problem of query plan optimization in the source-centric approach to information integration. We showed how local completeness information can be used to avoid redundant accesses to information sources. Our algorithm proceeds in two steps. In the first step, a semantically correct and source-complete query plan is generated using one of the algorithms in [43, 38, 39, 17, 19]. In the second step, redundant parts of this query plan are eliminated. The resulting query plan is guaranteed to be semantically correct, source-complete, and view-minimal.

Lambrecht and Kambhampati [35] apply the query optimization technique presented in this chapter to minimize the recursive query plans introduced in Chapter 2. Levy [36] uses local completeness information to test whether a query plan is complete, but doesn't consider query optimization. In [33], Kirk et al. give an algorithm that makes use of local completeness information, but doesn't guarantee view-minimality. Their algorithm first determines the part of a user query that is known to be stored completely by some information sources. It selects a minimal set of information sources that provide this part of the query. In a second step, every information source that might contribute some data to the remaining part of the query is added. This algorithm doesn't guarantee view-minimality, however, as can be seen from the following counterexample. Consider three information sources IS_{new}, IS_{bmw}, and IS_{honda} that store fragments of the *bluebook* relation from Example 5.1.2. IS_{new} stores all information for cars built in 1997, and nothing else. IS_{bmw} and IS_{honda} store information for cars built by BMW and Honda respectively. They are complete for information on cars built by BMW and Honda respectively in 1997. Suppose a user requests current market values for cars built by BMW and Honda. The part of the query that is guaranteed to be stored is the fragment of the *bluebook* relation for the year 1997. IS_{new} alone guarantees to provide this fragment. For the remaining part of the query though, IS_{bmw} and IS_{honda} have to be included. Therefore, the query plan resulting from the algorithm in [33] accesses all three information sources. However, this query plan is not view-minimal, because it is sufficient to only request information from the BMW and the Honda database.

Approximating a relation by two views is studied in the context of predicate caching [32] and in relation to the question of whether datalog programs can be approximated by unions of conjunctive queries [8,9]. Our terminonolgy of "conservative" and "liberal" views is adopted from [32]. The work of Chaudhuri in [8] and Chaudhuri and Kolaitis in [9] is of interest here because it points to limitations of the source-centric approach. If for example an information source stores the transitive closure of a predicate p, then there are no nonrecursive views that could be used as close approximations of this source relation.

Chapter 6

The Infomaster System

We present the Infomaster system, an information integration tool developed and tested at Stanford University. The Infomaster system makes it possible to bridge differences in schemata and terminology between existing databases. The query planning component of Infomaster applies essentially the algorithms presented in previous chapters. There are two main differences: First, there is a third level in the abstraction hierarchy, called *interface* relations, in addition to world relations and source relations. This additional level makes it possible to decouple the language used for describing information sources from the user's query language. Second, the Infomaster system is able to handle built-in predicates like "$<$" and "$*$".

6.1 Architecture

The Infomaster system is a generic information integration tool for integrating existing information sources. Information sources that can be integrated can vary from expressive SQL databases, over Z39.50 sources — a standard used for library information, to semistructured data that can be found on the WWW. Each type of information source requires a unique program, called *wrapper*, that translates between the language spoken by the information source and the language spoken by the core Infomaster system. The internal content language used by the Infomaster system is the *knowledge interchange format* (KIF), a language for representing first order logic expressions. For the sake of compatibility with previous chapters we will continue to use datalog notation.

6.2 Tested application areas

The Infomaster system was developed as a research project at Stanford University. It has been tested in a variety of application domains, some of which we are going to present here.

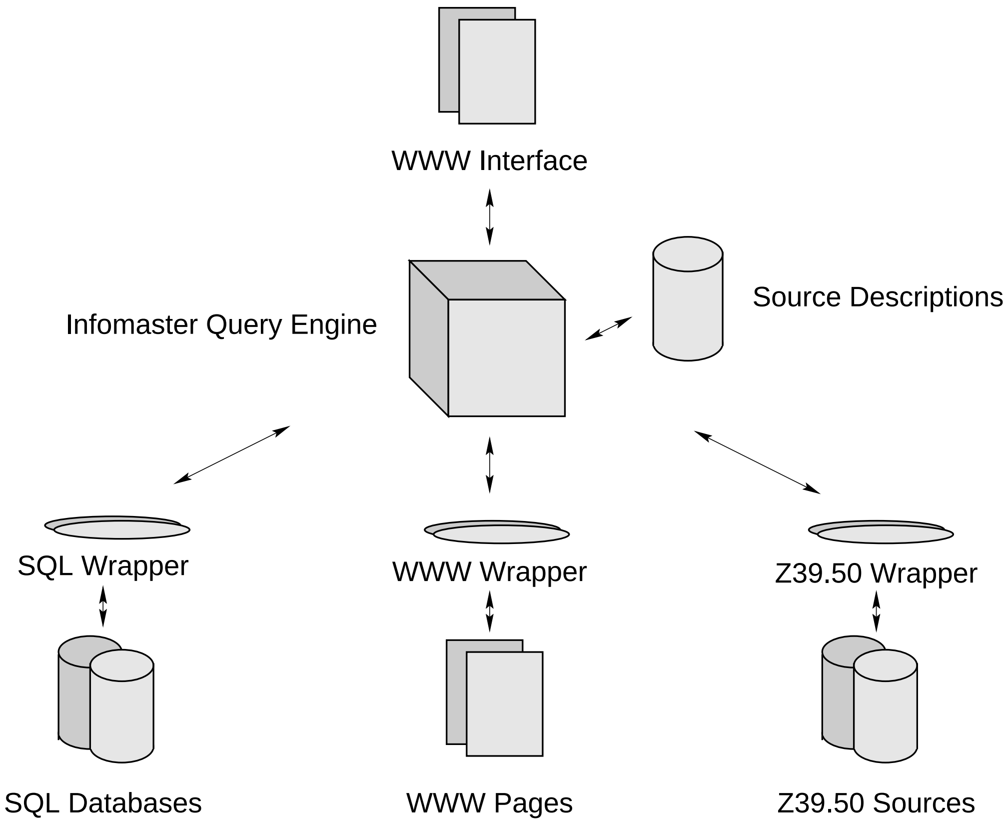


Figure 6.1: *Architecture of the Infomaster System*

6.2.1 Newspaper classifieds

Several newspapers are published in the San Francisco Bay Area. All of them have rental and used car classified ads. The Infomaster system has been applied to provide a uniform search interface to this information. For example, users can search for 2 bedroom apartments in Palo Alto, Menlo Park, or Portola Valley under \$1000. The system then gathers the corresponding classifieds from all relevant newspapers.

6.2.2 Product catalogs

The Infomaster system has been deployed in integrating electronic product catalogs and catalogs for houseware items from several vendors. This application requires a lot of terminology translation. For example, one houseware items vendor refers to its version of Teflon as Maxalon X2000. Clearly, a user shouldn't be required to know all these details when searching for a non-stick pan.

6.2.3 Campus databases

Stanford University itself has a wide collection of databases. The Infomaster system provides a uniform interface to databases on people, courses, and library information.

6.3 Abstraction hierarchy

Both the user interface and the available information sources are modeled by a set of *relations*. The WWW forms that users can use to enter their queries are abstracted as so-called *interface relations*. Data available from an information source can also be seen as a relation, which we call *source relation*. The information integration problem can be reduced in this framework to the problem of relating the interface and the source relations in an appropriate way. Figure 6.2 shows an example

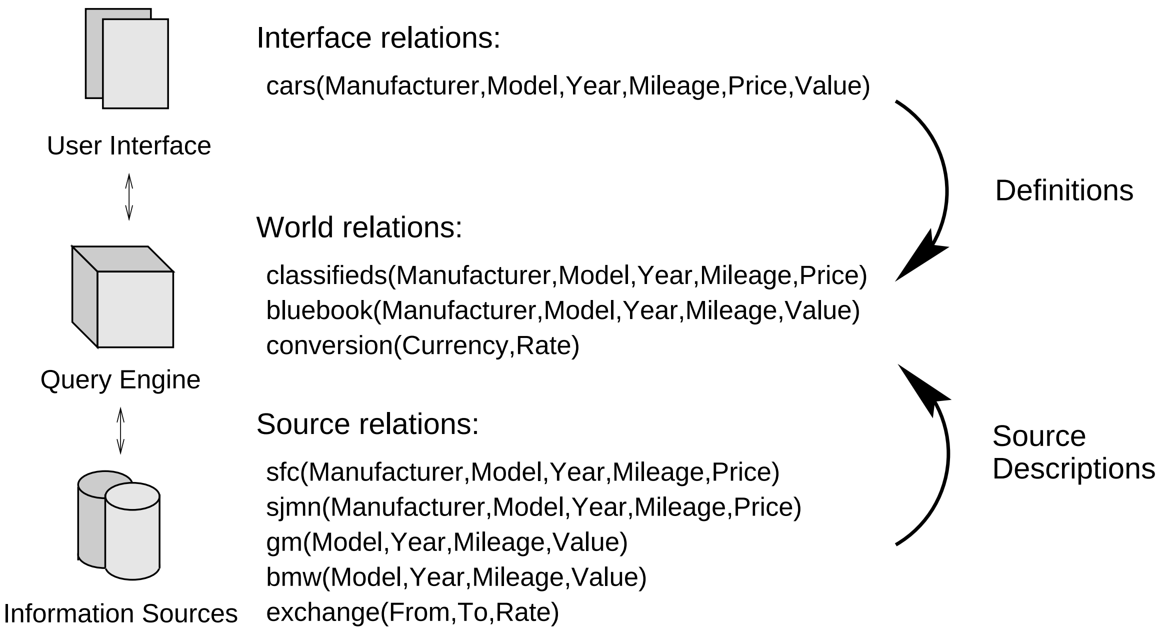


Figure 6.2: *The information integration problem is abstracted as describing the relationships between three kinds of relations: Interface relations conceptualize the interaction with a user through a WWW based user interface. Source relations represent the data that is actually stored in the available information sources. World relations are used as means to describe both interface and source relations and are crucial in the query planning process.*

of interface and source relations. The application domain in the example is the following: The San Francisco Chronicle and the San Jose Mercury News both contain a used car classifieds section in which cars are offered for sale. Moreover, we assume that we have access to information of car manufacturers General Motors and BMW. Both manufacturers provide data on the average market value of their cars for a given model, year, and mileage. Our goal is to provide a WWW interface to users that integrates all this information. Source relations *sfc* and *sjmn* model the information parsed from the used car classifieds in the San Francisco Chronicle and the San Jose Mercury News respectively. Source relations *gm* and *bmw* represent the information available from the two car manufacturers. Interface relation *cars* represents the WWW form presented to the user.

If we were sure that we would never add new information sources and that already integrated information sources never changed their content, then we could relate interface relations directly to source relations. However, we want to be more flexible in our design. For example, it is likely that we want to improve our service in the future by also including classifieds published in the Palo Alto Weekly, the Sacramento Bee, or the Los Angeles Times. In order to simplify adding new information sources and accommodating the changes in content of existing ones, we introduce a new level into our abstraction hierarchy. We call these new relations *world relations*. We express both interface relations and source relations in the terms of world relations. This allows us to easily integrate new information sources. Also, we can easily change the user interface without having to integrate the information sources anew.

World relations are chosen to be the basic building blocks of the application domain at hand. In our example, there are basically two kinds of information: classified ads and general information on car models. We represent all classified ads by the world relation *classifieds* and the market value information on car models by the world relation *bluebook*.

The example contains one further world relation called *conversion* that provides the current exchange rate from a given currency into dollar. There is also an additional source relation *exchange* that represents information provided by some currency exchange.

6.4 Descriptions of relationships

The previous section described how the information integration problem can be broken down into an abstraction hierarchy of source, world, and interface relations. Here we will show how these different abstraction levels can be related to each other. The descriptions of these relationships will be used by the query planner to translate user queries into queries that can be answered by the information sources, and to combine the resulting answers.

We describe both interface relations and source relations in terms of world relations. Interface relation *cars* combines information from classified ads with the average market value of the corresponding model from the *bluebook* world relation. This can be expressed in the following definition:

$$\begin{aligned}&cars(Manufacturer, Model, Year, Mileage, Price, Value) :-\\&\qquad classifieds(Manufacturer, Model, Year, Mileage, Price),\\&\qquad bluebook(Manufacturer, Model, Year, Mileage, Value)\end{aligned}$$

Both the San Francisco Chronicle and the San Jose Mercury News provide classified ads. However, none of them publishes *all* classified ads. Therefore, the *sfc* and *sjmn* source relations are *contained* in (and not equivalent to) the *classifieds* world relation. This containment is expressed by the following liberal source descriptions:

$$sfc^{l}(Manufacturer, Model, Year, Mileage, Price) :- \\ classifieds(Manufacturer, Model, Year, Mileage, Price)$$

$$sjmn^{l}(Manufacturer, Model, Year, Mileage, Price) :- \\ classifieds(Manufacturer, Model, Year, Mileage, Price)$$

General Motors and BMW, on the other hand, do provide all information on their respective car models. Therefore, source relations *gm* and *bmw* are equivalent to the corresponding fragments of the *bluebook* world relation. The distinction between containment and equivalence is important for the query optimization process. For example, even if there were another information source storing bluebook information for General Motors cars, it would be unnecessary to access both this information source and the original General Motors source because the original source is guaranteed to store all relevant information. On the other hand, a classified ad could be published in the San Francisco Chronicle or the San Jose Mercury News. None of the two information sources is complete. Equivalence is expressed by identical conservative and liberal source descriptions.

$$gm^{c,l}(Model, Year, Mileage, Value) :- \\ bluebook(gm, Model, Year, Mileage, Value)$$

$$bmw^{c,l}(Model, Year, Mileage\_in\_km, Value\_in\_DM) :- \\ bluebook(bmw, Model, Year, Mileage, Value), \\ conversion(dm, Rate), \\ Mileage = Mileage\_in\_km * 1.6, \\ Value = Value\_in\_DM * Rate$$

Information provided from the BMW information source is stored in *km* and *DM* instead of *miles* and *Dollars*. Therefore, the "mileage" in *km* has to be related to the mileage (in *miles*), and the market value in *DM* has to be related to the corresponding value in *Dollars*. For the second translation, the current exchange rate has to be obtained. Finally, the rule

$$exchange^{c,l}(From, dollar, Rate) :- \\ conversion(From, Rate)$$

expresses that the currency exchange provides data of exchange rates from a given currency into *Dollar*.

6.5 Query processing

Query processing in the Infomaster system is a three-step process. Assume the user asks a query Q. This query is expressed in terms of interface relations. In a first step, query Q is rewritten into a query in terms of world relations. We call this step *reduction*. In a second step, the descriptions of the source relations have to be used to translate the rewritten query into a query in terms of source relations. This second step involves the *query planning* described in Chapter 2. The query in terms of source relations is an executable *query plan*, because it only refers to data that is actually available

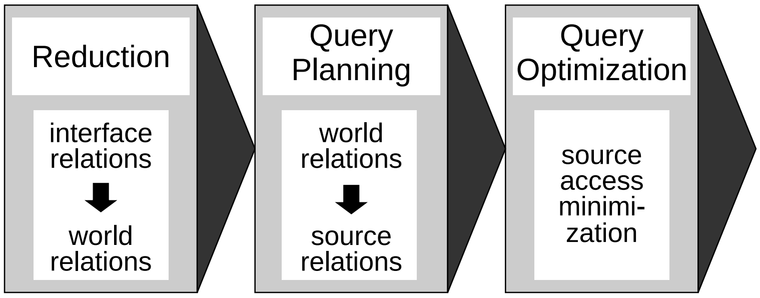


Figure 6.3: *The three steps of Infomaster's query processing.*

from the information sources. However, the generated query plan might be inefficient. Using the conservative source descriptions, the query plan can be optimized as described in Chapter 5.

As an example query, assume that a user asks for BMWs built in 1996 that are for sale for a price below their average market value:

$$\begin{aligned}&q(Model, Mileage, Price) :- \\ &\qquad cars(bmw, Model, 1996, Mileage, Price, Value), \\ &\qquad Price < Value\end{aligned}$$

We will discuss the three steps of the query planning process in the following.

6.5.1 Reduction

The reduction step in the query processing sequence is very simple. It is essentially a macro expansion. The user query is given in terms of interface relations, and interface relations are defined in terms of world relations. Therefore, the reduction step requires only to substitute interface relations by the corresponding definitions. In our example, the user query is rewritten to the following query in terms of world relations:

$$\begin{aligned}&q(Model, Mileage, Price) :- \\ &\qquad classifieds(bmw, Model, 1996, Mileage, Price), \\ &\qquad bluebook(bmw, Model, 1996, Mileage, Value), \\ &\qquad Price < Value\end{aligned}$$

6.5.2 Query planning

The interesting step in the query processing sequence is the query planning step. It requires to translate a query in terms of world relations into a query in terms of source relations. This is more complicated than the reduction step, because source relations are expressed in terms of world relations and not vice versa[1]. Chapter 2 presents algorithms for solving this problem for the case that both source descriptions and user queries do not contain any built-in predicates. In a "real" system like Infomaster though, built-in predicates like "$*$" and "$<$" are a must. As we showed in Chapter 4, maximally-contained query plans might not exist in the presence of built-in predicates. For all practical purposes though, it is sufficient to consider the built-in predicates as "ordinary" predicates during query planning. For example, the condition *Value = Value_in_DM * Rate* is translated into *times(Value, Value_in_DM, Rate)* and the condition *Price < Value* is translated into *less(Price, Value)*. Each subset of two attributes of relation *times* functionally determines the remaining attribute. Therefore, there are three chase rules for relation *times*. The resulting query plan in our example is the following:

$$
\begin{array}{l}
q(Mo_1,Mi_1,Pr_1) :- \ classifieds(Ma_1,Mo_2,Ye_1,Mi_2,Pr_2),\\
\qquad bluebook(Ma_2,Mo_3,Ye_2,Mi_3,Va_1),\ less(Pr_3,Va_2),\\
\qquad e(Mo_1,Mo_2),\ e(Mo_2,Mo_3),\ e(Mi_1,Mi_2),\ e(Mi_2,Mi_3),\\
\qquad e(Pr_1,Pr_2),\ e(Pr_2,Pr_3),\ e(Ma_1,bmw),\ e(Ma_2,bmw),\\
\qquad e(Va_1,Va_2),\ e(Ye_1,1996),\ e(Ye_2,1996)\\
classifieds(Ma,Mo,Ye,Mi,Pr) :- \ sfc(Ma,Mo,Ye,Mi,Pr)\\
classifieds(Ma,Mo,Ye,Mi,Pr) :- \ sjmn(Ma,Mo,Ye,Mi,Pr)\\
bluebook(gm,Mo,Ye,Mi,Va) \quad :- \ gm(Mo,Ye,Mi,Va)\\
bluebook(bmw,Mo,Ye,f_1(Mo,Ye,Mi\_km,Va\_DM),f_2(Mo,Ye,Mi\_km,Va\_DM))\\
\qquad :- \ bmw(Mo,Ye,Mi\_km,Va\_DM)\\
conversion(dm,f_3(Mo,Ye,Mi\_km,Va\_DM)) :- \ bmw(Mo,Ye,Mi\_km,Va\_DM)\\
times(f_1(Mo,Ye,Mi\_km,Va\_DM),Mi\_km,1.6) :- \ bmw(Mo,Ye,Mi\_km,Va\_DM)\\
times(f_2(Mo,Ye,Mi\_km,Va\_DM),Va\_DM,f_3(Mo,Ye,Mi\_km,Va\_DM))\\
\qquad :- \ bmw(Mo,Ye,Mi\_km,Va\_DM)\\
conversion(From,Rate) \quad :- \ exchange(From,Dollar,Rate)\\
e(Y_1,Y_2) :- \ conversion(X_1,Y_1),\ conversion(X_2,Y_2),\ e(X_1,X_2)\\
e(X_1,X_2) :- \ times(X_1,Y_1,Z_1),\ times(X_2,Y_2,Z_2),\ e(Y_1,Y_2),\ e(Z_1,Z_2)\\
e(Y_1,Y_2) :- \ times(X_1,Y_1,Z_1),\ times(X_2,Y_2,Z_2),\ e(X_1,X_2),\ e(Z_1,Z_2)\\
e(Z_1,Z_2) :- \ times(X_1,Y_1,Z_1),\ times(X_2,Y_2,Z_2),\ e(X_1,X_2),\ e(Y_1,Y_2)\\
e(X,Y) \quad :- \ e(X,Z),\ e(Z,Y)
\end{array}
$$

6.5.3 Query optimization

The query plan generated in the query planning phase can be simplified a lot before it is executed. The final result of the simplification is the following plan:

[1] It is of course possible to define world relations in terms of source relations. This would simplify the query planning step. However, adding an information source or accommodating the change in content of an information source then would become more difficult.

$$
\begin{aligned}
&q(Model, Mileage, Price) :- \\
&\qquad (sfc(bmw, Model, 1996, Mileage, Price) \\
&\qquad \lor\ sjmn(bmw, Model, 1996, Mileage, Price)), \\
&\qquad bmw(Model, 1996, Mileage\_in\_km, Value\_in\_DM), \\
&\qquad exchange(dm, dollar, Rate), \\
&\qquad Mileage = Mileage\_in\_km * 1.6, \\
&\qquad Value = Value\_in\_DM * Rate, \\
&\qquad Price < Value
\end{aligned}
$$

6.6 Conclusions and related work

We have given an overview of the Infomaster system. We presented the overall system architecture, some tested application areas, and showed how information sources can be described declaratively using an abstraction hierarchy of source, world, and interface relations. Finally, we explained the query planning process in Infomaster, consisting of a reduction, query planning, and query optimization phase.

The design of the Infomaster system builds upon extensive work in the field of information integration. Related efforts to integrate distributed information sources are the Information Manifold project [39], the SIMS project [3], the Occam project [34], and the TSIMMIS project [13]. The Information Manifold project and the SIMS project explore the use of descriptions logics for describing information sources. The Occam project uses general AI planning techniques to generate information gathering plans. Finally, the TSIMMIS project uses pattern matching techniques to match user queries against predefined queries with stored query plans.

Bibliography

[1] Sibel Adalı, K. S. Candan, Yannis Papakonstantinou, and V.S. Subrahmanian. Query caching and optimization in distributed mediator systems. In *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data*, pages 137–148, June 1996.

[2] Alfred V. Aho, Yehoshua Sagiv, and Jeffrey D. Ullman. Equivalences among relational expressions. *SIAM Journal on Computing*, 8(3):218–246, May 1979.

[3] Yigal Arens, Chin Y. Chee, Chun-Nan Hsu, and Craig A. Knoblock. Retrieving and integrating data from multiple information sources. *International Journal of Intelligent & Cooperative Information Systems*, 2(2):127–58, June 1993.

[4] Yigal Arens, Craig A. Knoblock, and Wei-Min Shen. Query reformulation for dynamic information integration. *International Journal on Intelligent and Cooperative Information Systems*, 6(2/3):99–130, June 1996.

[5] Catriel Beeri, Alon Y. Levy, and Marie-Christine Rousset. Rewriting queries using views in description logics. In *Proceedings of the Sixteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, PODS '97*, pages 99 – 108, Tucson, AZ, May 1997.

[6] Ashok K. Chandra and Philip M. Merlin. Optimal implementation of conjunctive queries in relational data bases. In *Proceedings of the Ninth Annual ACM Symposium on the Theory of Computing*, pages 77–90, 1977.

[7] Ashok K. Chandra and Moshe Y. Vardi. The implication problem for functional and inclusion dependencies is undecidable. *SIAM Journal on Computing*, 14(3):671–677, August 1985.

[8] Surajit Chaudhuri. Finding nonrecursive envelopes for datalog predicates. In *Proceedings of the Twelfth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 135 – 146, 1993.

[9] Surajit Chaudhuri and Phokion G. Kolaitis. Can datalog be approximated? In *Proceedings of the Thirteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 86 – 96, Minneapolis, MN, May 1994.

[10] Surajit Chaudhuri, Ravi Krishnamurthy, Spyros Potamianos, and Kyuseak Shim. Optimizing queries with materialized views. In *Proceedings of the Eleventh International Conference on Data Engineering*, IEEE Comput. Soc. Press, pages 190–200, Los Alamitos, CA, 1995.

[11] Surajit Chaudhuri and Moshe Y. Vardi. On the equivalence of recursive and nonrecursive datalog programs. In *Proceedings of the Eleventh ACM SIGACT-SIGMOD-SIGART Symposium on the Principles of Database Systems*, pages 55–66, San Diego, CA, June 1992.

[12] Surajit Chaudhuri and Moshe Y. Vardi. One the equivalence of recursive and nonrecursive datalog programs. *Journal of Computer and System Sciences*, 54(1):61 – 78, February 1997.

[13] Sudarshan Chawathe, Hector Garcia-Molina, Joachim Hammer, Kelly Ireland, Yannis Papakonstantinou, Jeffrey Ullman, and Jennifer Widom. The TSIMMIS project: Integration of heterogeneous information sources. In *Proceedings of the 100th Anniversary Meeting*, pages 7–18, Tokyo, Japan, October 1994. Information Processing Society of Japan.

[14] Alonzo Church. A note on the Entscheidungsproblem. *Journal of Symbolic Logic*, 1:40–41, 101–102, 1936.

[15] Stephen A. Cook. The complexity of theorem proving procedures. In *Proceedings of the Third Annual ACM Sympoisum on Theory of Computing*, pages 151 – 158, Shaker Heights, OH, May 1971.

[16] Oliver M. Duschka. Query optimization using local completeness. In *Proceedings of the Fourteenth National Conference on Artificial Intelligence, AAAI-97*, pages 249–255, Providence, RI, July 1997.

[17] Oliver M. Duschka and Michael R. Genesereth. Answering recursive queries using views. In *Proceedings of the Sixteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, PODS '97*, pages 109 – 116, Tucson, AZ, May 1997.

[18] Oliver M. Duschka and Michael R. Genesereth. Infomaster — an information integration tool. In *Proceedings of the International Workshop on Intelligent Information Integration during the 21st German Annual Conference on Artificial Intelligence, KI-97*, Freiburg, Germany, September 1997.

[19] Oliver M. Duschka and Michael R. Genesereth. Query planning in Infomaster. In *Proceedings of the 1997 ACM Symposium on Applied Computing*, San Jose, CA, February 1997.

[20] Oliver M. Duschka and Alon Y. Levy. Recursive plans for information gathering. In *Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence, IJCAI*, Nagoya, Japan, August 1997.

[21] Thomas Eiter, Georg Gottlob, and Heikki Mannila. Adding disjunction to datalog. In *Proceedings of the Thirteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 267 – 278, Minneapolis, MN, May 1994.

[22] Oren Etzioni, Keith Golden, and Daniel Weld. Tractable closed world reasoning with updates. In *Proceedings of the 4th International Conference on Principles of Knowledge Representation and Reasoning*, pages 178–189, San Francisco, CA, June 1994.

[23] Oren Etzioni, Keith Golden, and Daniel Weld. Sound and efficient closed-world reasoning for planning. *Journal of Artificial Intelligence*, 89(1–2):113–148, January 1997.

[24] Oren Etzioni and Daniel S. Weld. A softbot-based interface to the internet. *Communications of the ACM*, 37(7):72–76, 1994.

[25] Daniela Florescu, Louiqua Raschid, and Patrick Valduriez. A methodology for query reformulation in CIS using semantic knowledge. *International Journal of Intelligent & Cooperative Information Systems, special issue on Formal Methods in Cooperative Information Systems*, 5(4), 1996.

[26] Marc Friedman and Daniel S. Weld. Efficiently executing information-gathering plans. In *Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence, IJCAI*, Nagoya, Japan, August 1997.

[27] Hector Garcia-Molina, Yannis Papakonstantinou, Dallan Quass, Anand Rajaraman, Yehoshua Sagiv, Jeffrey D. Ullman, Vasilis Vassalos, and Jennifer Widom. The TSIMMIS approach to mediation: Data models and languages. *Journal of Intelligent Information Systems*, 8(2):117–132, 1997.

[28] Donald F. Geddis, Michael R. Genesereth, Arthur M. Keller, and Narinder P. Singh. Infomaster: A virtual information system. In *Intelligent Information Agents Workshop at CIKM '95*, Baltimore, MD, December 1995.

[29] Nam Quan Huyn. A more aggressive use of views to extract information. Technical Report STAN-CS-TR-96-1577, Department of Computer Science, Stanford University, 1996.

[30] Tomasz Imielinski and Witold Lipski Jr. Incomplete information in relational databases. *J. ACM*, 31(4):761–791, 1984.

[31] Richard M. Karp. Reducibility among combinatorial problems. *Complexity of Computer Computations*, pages 85 – 104, 1972.

[32] Arthur M. Keller and Julie Basu. A predicate-based caching scheme for client-server database architectures. *The VLDB Journal*, 5:35–47, January 1996.

[33] Thomas Kirk, Alon T. Levy, Yehoshua Sagiv, and Divesh Srivastava. The Information Manifold. In *Proceedings of the AAAI Spring Symposium on Information Gathering in Distributed Heterogeneous Environments*, Stanford, CA, March 1995.

[34] Chung T. Kwok and Daniel S. Weld. Planning to gather information. In *Proceedings of the AAAI Thirteenth National Conference on Artificial Intelligence*, 1996.

[35] Eric Lambrecht and Subbarao Kambhampati. Minimizing recursive information gathering plans. Technical Report TR-97-040, Department of Computer Science and Engineering, Arizona State University, 1997.

[36] Alon Y. Levy. Obtaining complete answers from incomplete databases. In *Proceedings of the 22nd International Conference on Very Large Databases*, pages 402–412, Bombay, India, 1996.

[37] Alon Y. Levy, Alberto O. Mendelzon, Divesh Srivastava, and Yehoshua Sagiv. Answering queries using views. In *Proceedings of the 14th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, San Jose, CA, May 1995.

[38] Alon Y. Levy, Anand Rajaraman, and Joann J. Ordille. Query-answering algorithms for information agents. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence, AAAI-96*, Portland, OR, August 1996.

[39] Alon Y. Levy, Anand Rajaraman, and Joann J. Ordille. Querying heterogeneous information sources using source descriptions. In *Proceedings of the 22nd International Conference on Very Large Databases*, pages 251–262, Bombay, India, 1996.

[40] Alon Y. Levy, Anand Rajaraman, and Jeffrey D. Ullman. Answering queries using limited external processors. In *Proceedings of the 15th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, Montreal, Canada, 1996.

[41] John C. Mitchell. The implication problem for functional and inclusion dependencies. *Information and Control*, 56(3):154 – 173, March 1983.

[42] Emil Post. A variant of a recursively unsolvable problem. *Bulletin American Mathematical Society*, 52:264–268, 1946.

[43] Xiaolei Qian. Query folding. In *Proceedings of the 12th International Conference on Data Engineering*, pages 48–55, New Orleans, LA, February 1996.

[44] Anand Rajaraman, Yehoshua Sagiv, and Jeffrey D. Ullman. Answering queries using templates with binding patterns. In *Proceedings of the 14th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, 1995.

[45] Raghu Ramakrishnan, Yehoshua Sagiv, Jeffrey D. Ullman, and Moshe Y. Vardi. Proof-tree transformation theorems and their applications. In *Proceedings of the Eighth ACM SIGACT-SIGMOD-SIGART Symposium on Princicples of Database Systems*, pages 172 – 181, Philadelphia, PA, March 1989.

[46] Yehoshua Sagiv and Mihalis Yannakakis. Equivalence among relational expressions with the union and difference operators. *J. ACM*, 27(4):633–655, 1980.

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

[48] Walter Sujansky and Russ B. Altman. Towards a standard query model for sharing decision-support applications. In *Proceedings of the Eighteenth Annual Symposium on Computer Applications in Medical Care*, pages 325 – 331, Washington, DC, 1994.

[49] Anthony Tomasic, Louiqua Raschid, and Patrick Valduriez. A data model and query processing techniques for scaling access to distributed heterogeneous databases in Disco. *IEEE Transactions on Computers, special issue on Distributed Computing Systems*, 1997.

[50] Odysseas G. Tsatalos, Marvin H. Solomon, and Yannis E. Ioannidis. The GMAP: A versatile tool for physical data independence. *VLDB Journal*, 5(2):101–118, 1996.

[51] Jeffrey D. Ullman. *Principles of Database and Knowledgebase Systems*, volume 2. Computer Science Press, 1989.

[52] Jeffrey D. Ullman. Information integration using logical views. In *Proceedings of the Sixth International Conference on Database Theory*, January 1997.

[53] Ron van der Meyden. The complexity of querying indefinite information: Defined relations, recursion and linear order. Technical report, Rutgers University, 1992.

[54] Ron van der Meyden. Recursively indefinite databases. *Theoretical Computer Science*, 116(1):151–194, August 1993.

[55] Ron van der Meyden. The complexity of querying indefinite data about linearly ordered domains. *Journal of Computer and System Sciences*, 54(1):113 – 135, February 1997.

[56] Moshe Y. Vardi. The complexity of relational query languages. In *Proceedings of the Fourteenth Annual ACM Symposium on Theory of Computing*, pages 137 – 146, San Francisco, CA, May 1982.

[57] H. Z. Yang and P.-Å. Larson. Query transformation for PSJ-queries. In *Proceedings of the Thirteenth International Conference on Very Large Data Bases*, pages 245–254, Los Altos, CA, 1987.