Chapter 5

Describing Data Sources

In order for a data integration system to process a query over a set of data sources, the system must know which sources are available, what data exists in each source and how each source can be accessed. The *source descriptions* in a data integration system encode this information. In this chapter we study the different components of source descriptions and identify the tradeoffs involved in designing formalisms for source descriptions.

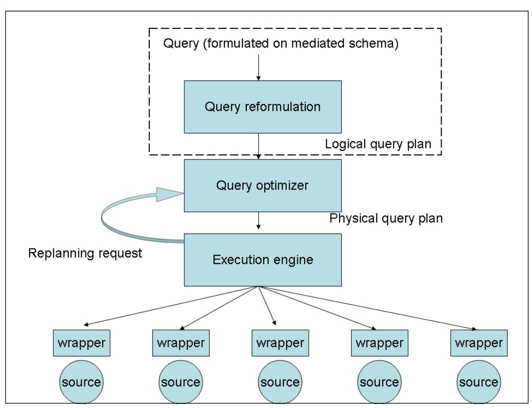

Figure 5.1: Query processing in a data integration system. This chapter focuses on the reformulation step, highlighted in the dashed box.

To put the topic of this chapter in context, consider the architecture of a data integration system, redrawn in Figure 5.1. Recall that a user or an application pose a query to the data integration system using the relations and attributes of the mediated schema. The system then reformulates the query into a query over the data sources. The result of the reformulation is called a *logical query* *plan*. The logical query plan is later optimized so it runs efficiently. In this chapter we show how source descriptions are expressed and how the system uses them to reformulate the user's query into a logical query plan.

5.1 Overview and Desiderata

Before we begin our technical discussion of source descriptions, it is instructive to highlight the goals we that these descriptions are trying to achieve, and outline basic desiderata from a source description formalism.

To understand the requirements from source descriptions, we use a scenario that includes the following mediated schema and data sources. Note that the first data source contains four tables, while the others each include a single table. We refer to a relation in a data source by the relation name, prefixed by the source name (e.g., S1.Movie).

Mediated schema:
Movie(title, director, year, genre), Actors(title, name)
Plays(movie, location, startTime), Reviews(title, rating, description)

Data sources:
S1:
  Movie(MID, title)
  Actor(AID, firstName, lastName, nationality, yearOfBirth)
  ActorPlays(AID, MID)
  MovieDetail(MID, director, genre, year)

S2:
  Cinemas(place, movie, start)

S3:
  NYCCinemas(name, title, startTime)

S4:
  Reviews(title, date, grade, review)

S5:
  MovieGenres(title, genre)

S6:
  MovieDirectors(title, dir)

S7:
  MovieYears(title, year)

A source description needs to convey several pieces of information. The main component of a source description, called a *schema mapping*, is a specification of *what* data exists in the source and how the terms used in the source schema relate to the terms used in the mediated schema. The schema mapping needs to be able to handle the following discrepancies between the source schemata and the mediated schema:

- **Relation and attribute names:** The relation and attribute names in the mediated schema are likely to be different than they are in the sources, even if they are speaking of the same terms. For example, the attribute description in the mediated schema refers to the text description of a review, which is the same as the attribute review in source S4. Similarly, if the same relation or attribute name are used in the mediated schema and in the source that does not necessarily entail that they mean the same thing. For example the attribute name appears in both the Actors relation in the mediated schema and in S3, but refers to actor names in one case and to cinema names in the other.

- **Tabular organization:** The tabular organization of data can be different between the mediated schema and the source. For example, in the mediated schema, the relation Actor stores the relationship between actor names and movie titles. In contrast, in source S1, actors are modeled with IDs, some of their data is stored in the relation Actor, and the relationship with movies is stored in the relation ActorPlays. Hence, the schema mapping needs to be able to specify that a join of two tables in the source corresponds to a relation in the mediated schema and vice versa.

- **Domain coverage:** The coverage and level of detail of the two schemas may differ. For instance, source S1 models actors in more detail than the mediated schema. The source models the year of birth and nationality of actors in addition to their name.

- **Data-level variations:** The schemas may be assuming different conventions for specifying data values. In the simple case, there may be a difference in the scales used to specify values (e.g, GPA's on a letter scale versus a numeric scale). In other cases, names of people or companies may be written differently. For example, in S1 names of actors are broken up into two columns, whereas in the mediated schema the full name is in one column.

Collectively, these differences between the mediated schema and the source schema (or between any pair of schema) are called *semantic heterogeneity*. Bridging semantic heterogeneity is considered to be one of the key challenges in data integration, and will be discussed extensively. We cover schema mappings in Sections 5.2.

In addition to schema mappings, the source descriptions also specify information that enables the data integration system to optimize queries to the sources and to avoid illegal access patterns. Specifically, the following two are common.

Access-pattern limitations: Data sources may differ on which access patterns they support. In the best case, a data source is a full fledged database system, and we can send it any SQL query. However, many data sources are much more limited. For example, a data source whose interface is a web form constrains the possible access patterns. In order to get tuples from the source, the data integration system needs to supply some set of legal input parameters. We discuss limitations on access patterns to sources in Section 5.3. We postpone the discussion leveraging processing power of data sources to Chapter 9.

Source completeness: it is often important to know whether a source is complete w.r.t. the contents it's purported to have. When a data source is known to be complete, the data integration system can save work by not accessing other data sources that have overlapping data. For example, if S2.Cinemas is known to have playing times of all movies in the country, then we can ignore S3 and S4 for many queries. In some cases, the data source may be complete w.r.t. a subset of its contents. Given partially-complete sources, we also want to know whether answers to our queries are guaranteed to be complete. We discuss how a data integration handles knowledge about completeness in Section 5.5.

5.2 Schema mapping languages

Formally, a schema mapping is a set of expressions that describe a relationship between a set of schemata (typically two). In our context, the schema mappings describe a relationship between the mediated schema and the schema of the sources. When a query is formulated in terms of the mediated schema, we use the mappings to reformulate the query into appropriate queries on the sources. The result of the reformulation is a *logical query plan*.

Schema mappings are also used in other contexts. In the context of data exchange and data warehousing (which we discuss in Chapter 16), schema mappings express a relationship between a source database and a target database. In this context, we use the schema mappings to map the data from the source database into the target database (which is often a data warehouse). A schema mapping may also be used to describe the relationship between two databases that store data. In this context, the goal of the schema mapping is typically to merge the two databases into one. We discuss this case briefly in Chapter 18.

5.2.1 Principles of schema mapping languages

In this chapter we use query expressions as the main mechanism for specifying schema mappings, and we leverage the algorithms described in Chapter 4 for query reformulation. In our description, we denote the mediated schema by G, and the source schemata by $S_1, \ldots, S_n$.

Semantics of schema mappings

The semantics of a schema mapping formalism are specified by defining which instances of the mediated schema are consistent with given instances of the data sources. Specifically, a semantic mapping M defines a relation M_R over

$$I(G) \times I(S_1) \times \ldots \times I(S_n)$$

where $I(G)$ denotes the possible instances of the mediated schema, and $I(S_1), \ldots, I(S_n)$ denote the possible instances of the source relations $S_1, \ldots, S_n$, respectively. If $(g, s_1, \ldots, s_n) \in M_R$, then g is a possible instance of the mediated schema when the source relation instances are $s_1, \ldots, s_n$. The semantics of queries over the mediated schema are based on the relation M_R.

Definition 5.1: *(Certain answers) Let M be a schema mapping between a mediated schema G and source schemata $S_1, \ldots, S_n$, that defines the relation M_R over $I(G) \times I(S_1) \times \ldots \times I(S_n)$.*

Let Q be a query over G, and let $s_1, \ldots, s_n$ be instances of the source relations. We say that $\bar{t}$ is a certain answer of Q w.r.t. M and $s_1, \ldots, s_n$, if $\bar{t} \in Q(g)$ for every instance g of G such that $(g, s_1, \ldots, s_n) \in M_R$. $\square$

Logical query plans

To obtain the certain answers, the data integration system will create a logical query plan as a result of reformulation. The logical query plan is a query expression that refers only to the relations in the data sources. As we see later, it is not always possible to create a query plan that generates all the certain answers. Hence, in our discussion, we will analyze two different algorithmic problems: finding the best possible logical query plan and finding all the certain answers.

As we discuss the different schema mapping languages, we keep the following properties in mind:

- **Flexibility:** given the significant differences between disparate schemata, the schema mapping languages should be very flexible. That is, the language should be able to express a wide variety of relationships between schemata.
- **Efficient reformulation:** since our goal is to use the schema mapping to reformulate queries, we should be able to develop reformulation algorithms whose properties are well understood and are efficient in practice. This requirement is often at odds with that of flexibility, because more expressive languages are typically harder to reason about.
- **Easy update:** for a formalism to be useful in practice, it needs to be easy to add and remove sources. If adding a new data source potentially requires inspecting all other sources, the resulting system will be hard to manage as it scales up to a large number of sources.

We discuss three schema mapping languages: Global-as-View (Section 5.2.2), Local-as-View (Section 5.2.3), and Global-Local-as-View (Section 5.2.4) that combines the features of the two previous ones. For historical reasons, the formalism names leave some room for further explanation, and we do so as we go along. We note that the topic of *creating* schema mappings is the subject of Chapter 7.

5.2.2 Global-as-View

The first formalism we consider, Global-as-View (GAV), takes a very intuitive approach to specifying schema mappings: GAV defines the mediated schema as a set of views over the data sources. The mediated schema is often referred to as a global schema, hence the name of the formalism.

Syntax and semantics

The syntax of GAV source descriptions is defined as follows.

Definition 5.2: *(GAV schema mappings) Let G be a mediated schema, and let $\bar{S} = \{S_1, \ldots, S_n\}$ be schemata of n data sources. A Global-as-View schema mapping $\bar{M}$ is a set of expressions of the form $G_i(\bar{X}) \supseteq Q(\bar{S})$ or $G_i(\bar{X}) = Q(\bar{S})$, where*

- *G_i is a relation in G, and appears in at most one expression in $\bar{M}$, and*

- *$Q(\bar{S})$ is a query over the relations in $\bar{S}$.* □

Expressions with $\supseteq$ make the open-world assumption. That is, the data sources, and therefore the instances computed for the relations in the mediated schema, are assumed to be incomplete. Expressions with $=$ make the closed-world assumption in that the instances computed for the relations in the mediated schema are assumed to be complete. In Section 5.5 we show how to state more refined notions of completeness for data sources.

Example 5.1: The following is a GAV schema mapping for some of the sources in our running example. For readability, when illustrating GAV schema mappings, we typically abuse the notation by not showing the head of the query over the data sources, $Q(\bar{S})$ since it is the same as the relation G_i.

The first expression shows how to obtain tuples for the Movie relation by joining relations in S1. The second expression obtains tuples for the Movie relation by joining data from sources S5, S6 and S7. Hence, the tuples that would be computed for Movie are the result of the union of the first two expressions. The third and fourth expressions generate tuples for the Plays relation by taking the union of S2 and S3.

Movie(title, director, year, genre) $\supseteq$ S1.Movie(MID, title),
    S1.MovieDetail(MID, director, genre, year)
Movie(title, director, year, genre) $\supseteq$ S5.MovieGenres(title, genre),
    S6.MovieDirectors(title, director),
    S7.MovieYears(title, year)
Plays(movie, location, startTime) $\supseteq$ S2.Cinemas(location, movie, startTime)
Plays(movie, location, startTime) $\supseteq$ S3.NYCCinemas(location, movie, startTime)

□

The following definition specifies the relation M_R entailed by a GAV schema mapping M, thereby defining its semantics.

Definition 5.3: *(GAV semantics) Let $\bar{M} = M_1, \ldots, M_l$ be a GAV schema mapping between G and $\bar{S} = \{S_1, \ldots, S_n\}$, where M_i is of the form $G_i(\bar{X}) \supseteq Q_i(\bar{S})$, or $G_i(\bar{X}) = Q_i(\bar{S})$.*

Let g be an instance of the mediated schema G, and let $\bar{s} = s_1, \ldots, s_n$ be instances of $S_1, \ldots, S_n$, respectively. The tuple of instances $(g, s_1, \ldots, s_n)$ is in M_R if for every $1 \leq i \leq l$, the following holds:

- *if M_i is a $=$ expression, then the extension of G_i in g is equal to the result of evaluating Q_i on $\bar{s}$,*
- *if M_i is a $\supseteq$ expression, then the extension of G_i in g is a superset of result of evaluating Q_i on $\bar{s}$.* $\square$

Reformulation in GAV

The main advantage of GAV is its conceptual simplicity. The mediated schema is simply a view over the data sources. To reformulate a query posed over the mediated schema, we simply need to unfold the query with the view definitions (see Section 4.1). Furthermore, the reformulation resulting from the unfolding is guaranteed to find all the certain answers. Hence, the following theorem summarizes the complexity of reformulation and query answering in GAV.

Theorem 5.1: *Let $\bar{M} = M_1, \ldots, M_l$ be a GAV schema mapping between G and $\bar{S} = \{S_1, \ldots, S_n\}$, where M_i is of the form $G_i(\bar{X}) \supseteq Q_i(\bar{S})$, or $G_i(\bar{X}) = Q_i(\bar{S})$. Let Q be a query over G.*

If Q and the Q_i's in $\bar{M}$ are conjunctive queries or unions of conjunctive queries, even with interpreted and negated atoms, then the problem of finding all certain answers to Q is in PTIME in the size of the data sources, and the complexity of reformulation is in PTIME in the size of the query and source descriptions. $\square$

Example 5.2: Suppose we have the following query over the mediated schema, asking for comedies starting after 8pm:

Q(title, location, startTime) :- Movie(title, director, year, "comedy"),
Plays(title, location, st), st $\geq$ 8pm

Reformulating Q with the source descriptions in Example 5.1 would yield the following four logical query plans:

Q'(title, location, startTime) :- S1.Movie(MID, title),
S1.MovieDetail(MID, director, "comedy" year),
S2.Cinemas(location, movie, st), st $\geq$ 8pm
Q'(title, location, startTime) :- S1.Movie(MID, title),
MovieDetail(MID, director, "comedy" year),
S3.NYCCinemas(location, title, st), st $\geq$ 8pm

Q′(title, location, startTime) :- S5.MovieGenres(title, genre), S6.MovieDirectors(title, director)
S7.MovieYears(title, year), S2.Cinemas(location, movie, st), st ≥
Q′(title, location, startTime) :- S5.MovieGenres(title, genre), S6.MovieDirectors(title, director)
S7.MovieYears(title, year), S3.NYCCinemas(location, title, st), s

We note two points about the above reformulation. First, the reformulation may not be the most efficient method to answer the query. For example, in this case it may be better to factor common subexpressions (namely, Movie and Plays) in order to reduce the number of joins necessary to evaluate the query. We discuss query optimization for data integration in Chapter 9.

Second, note that in the last two reformulations, the subgoals of S6.MovieDirectors and S7.MovieYears are redundant, since all we really need from the Movies relation is the genre of the movie. However, there is no way of reaching this conclusion with GAV descriptions since the way Movie is defined requires a tuple for all three of S5, S6 and S7. □

Discussion

In terms of modeling, GAV source descriptions specify directly how to compute tuples of the mediated schema relations from tuples in the sources. We've already seen one limitation of GAV in Example 5.2 when we could not remove the redundant subgoals of S6 and S7. The following is a more extreme example of where translating data sources into the mediated schema is limiting.

Example 5.3: Suppose we have a data source S8 that stored pairs of (actor, director) who worked together on movies. The only way to model this source in GAV is with the following two descriptions that use NULL liberally.

Actors(NULL, actor) ⊇ S8(actor, director)
Movie(NULL, director, NULL, NULL) ⊇ S8(actor, director)

Note that these descriptions essentially create tuples in the mediated schema that include NULLs in all columns except one. For example, if the source S8 would include the tuples {Keaton, Allen} and {Pacino, Coppola}, then the tuples computed for the mediated schema would be:

Actors(NULL, Keaton), Actors(NULL, Pacino)
Movie(NULL, Allen, NULL, NULL), Movie(NULL, Coppola, NULL, NULL)

Now suppose we have the following query that essentially recreates S8:

Q(actor, director) :- Actors(title, actor), Movie(title, director, genre, year)

We would not be able to retrieve the tuples from S8 because the source descriptions lost the relationship between actor and director. □

A final limitation of GAV is that adding and removing sources involves considerable work and knowledge of the sources. For example, suppose we discover another source that includes only movie directors (similar to S6). In order to update the source descriptions, we need to specify exactly which sources it needs to be joined with in order to produce tuples of Movie. Hence, we need to be aware of all sources that contribute movie years and movie genres (of which there may also be many). In general, if adding a new source requires that we are familiar with all other sources in the system, then the system is unlikely to scale to a large number of sources.

5.2.3 Local-as-View

Local-as-view (LAV) takes an opposite approach to GAV. Instead of specifying how to compute tuples of the mediated schema, LAV focuses on describing each data source as precisely as possible and *independently* of any other sources.

Syntax and semantics

LAV expressions describe data sources as queries over the mediated schema.

Definition 5.4: *(LAV schema mappings) Let G be a mediated schema, and let $\bar{S} = \{S_1, \ldots, S_n\}$ be schemata of n data sources. A Local-as-View schema mapping M is a set of expressions of the form $S_i(\bar{X}) \subseteq Q_i(G)$ or $S_i(\bar{X}) = Q_i(G)$, where*

- *Q_i is a query over the mediated schema G, and*
- *S_i is a source relation and it appears at most one expression in M.* □

As with GAV, LAV expressions with $\subseteq$ make the open-world assumption and expressions with $=$ make the closed-world assumption. However, LAV descriptions make completeness statements about data sources, not about the relations in the mediated schma.

Example 5.4: In LAV, sources S5–S7 would be described as follows, simply as projection queries over the Movie relation in the mediated schema. Note that here too we abuse the notation for clarity and omit the head of the Q_i's.

S5.MovieGenres(title, genre) $\subseteq$ Movie(title, director, year, genre)
S6.MovieDirectors(title, dir) $\subseteq$ Movie(title, director, year, genre)
S7.MovieYears(title, year) $\subseteq$ Movie(title, director, year, genre)

With LAV we can also model the source S8 as a join over the mediated schema:

S8(actor, dir) $\subseteq$ Movie(title, director, year, genre), Actors(title, actor)

Furthermore, we can also express constraints on the contents of data sources. For example, we can describe the following source that includes movies produced after 1970 and are all comedies:

S9(title, year, "comedy") $\subseteq$ Movie(title, director, year, "comedy"), year $\geq$ 1970. □

As with GAV, the semantics of a LAV schema mapping are defined by specifying the relation M_R defined by M.

Definition 5.5: *(LAV semantics) Let $M = M_1, \ldots, M_l$ be a LAV schema mapping between G and $\bar{S} = \{S_1, \ldots, S_n\}$, where M_i is of the form $S_i(\bar{X}) \subseteq Q_i(G)$ or $S_i(\bar{X}) = Q_i(G)$.*

Let g be an instance of the mediated schema G, and let $\bar{s} = s_1, \ldots, s_n$ be instances of $S_1, \ldots, S_n$, respectively. The tuple of instances $(g, s_1, \ldots, s_n)$ is in M_R if for every $1 \leq i \leq l$, the following holds:

- *if M_i is a $=$ expression, then the result of evaluating Q_i over g is equal to s_i,*
- *if M_i is a $\subseteq$ expression, then the result of evaluating Q_i over g is a subset s_i.*

□

Reformulation in LAV

The main advantage of LAV is that data sources are described in isolation and the system, not the designer, is responsible for finding ways of combining data from multiple sources. As a result, it is easier for a designer to add and remove sources.

Example 5.5: Consider the following query asking for comedies produced after 1960:

Q(title) :- Movie(title, director, year, "comedy"), year $\geq$ 1960.

Using the sources S5–S7, we would generate the following reformulation from the LAV source descriptions:

Q'(title) :- S5.MovieGenres(title, "comedy"), S7.MovieYears(title, year), year $\geq$ 1960.

Note that unlike GAV, the reformulation here did not require a join with the MovieDirectors relation in S6. Using the source S9, we would also create the following reformulation:

Q'(title) :- S9(title, year,"comedy")

Note that here the reformulation does not need to apply the predicate on year, because S9 is already known to contain only movies produced after 1970. □

Of course, to obtain this flexibility, we need to develop more complex query reformulation algorithms. Fortunately, the techniques for answering queries using views (Section 4.3) give us a framework for establishing results for reformulation in LAV.

To see why answering queries using views applies in our context, simply consider the following. The mediated schema represents a database whose tuples are unknown. The data sources in LAV are described as view expressions over the mediated schema. The extensions of the views are the data stored in the data sources. For example, S8 is described as a join over the mediated schema. Hence, to answer a query formulated over the mediated schema, we need to reformulate it into a query over the known views, i.e., the data sources. Unlike the traditional setting of answering queries using views, here the original database (i.e., the mediated schema) never had any tuples stored in it. However, that makes no difference to the query reformulation algorithms.

The above formulation of the problem immediately yields a plethora of algorithms and complexity results regarding reformulation for LAV, as summarized by the following theorem. The proof of the theorem is a corollary of the results in Chapter 4.

Theorem 5.2: *Let $M = M_1, \ldots, M_l$ be a LAV schema mapping between G and $\bar{S} = \{S_1, \ldots, S_n\}$, where M_i is of the form $S_i(\bar{X}) \subseteq Q_i(G)$ or $S_i(\bar{X}) = Q_i(G)$. Let Q be a conjunctive query over G.*

- *If all the Q_i's in M are conjunctive queries with no interpreted predicates or negation, and all the M_i's are $\subseteq$ expressions, then all the certain answers can be found in time polynomial in the size of the data and of the size of M.*

- *If all the Q_i's in M are conjunctive queries with no interpreted predicates or negation, and some of the expressions in M are $=$ expressions, then finding all the certain answers to Q is co-NP hard in the size of the data.*

- *If some of the Q_i's include interpreted predicates, then finding all then finding all certain answers to Q is co-NP hard in the size of the data.* □

We also note that finding all certain answers is co-NP hard in the size of the data if the Q_i's include unions or negated predicates.

We generate logical query plans for LAV schema mappings using any of the algorithms described in Chapter 4 for finding a maximally-contained rewriting of a query using a set of views. The computational complexity of finding the maximally-contained rewriting is polynomial in the number of views and the size of the query. Checking whether the maximally-contained rewriting is equivalent to the original query is NP-complete. We note that in practice, even when the complexity of finding all the certain answers is co-NP hard, the logical query plan created by the algorithms will typically find all the certain answers.

Discussion

The added flexibility of LAV is also the reason for the increased computational complexity of answering queries. Fundamentally, the reason is that LAV enables expressing incomplete information. Given a set of data sources, GAV mappings define a *single* instance of the mediated schema that is consistent with the sources, and therefore query answering can simply be done on that instance. For that reason, the complexity of query answering is similar to that of query evaluation over a database. In contrast, given a set of LAV source descriptions, there are a *set* of instances of the mediated schema that are consistent with the data sources. As a consequence, query answering in LAV amounts to querying incomplete information, which is computationally more expensive.

Finally, we note a shortcoming of LAV. Consider the relations S1.Movie(MID, title) and S1.MovieDetail(MID, director, genre, year). The join between these two relations requires the MID key, which is internal to S1 and not modeled in the mediated schema. Hence, while it is possible to model the fact that

MovieDetail contains directors, genres and years of movies, LAV descriptions would lose the connection of those attributes with the movie title. The only way to circumvent that is to introduce an identifier for movies in the mediated schema. However, identifiers are typically not meaningful across multiple data sources, and hence we'd need to introduce a special identifier for every source where it is needed.

5.2.4 Global-and-Local-as-View

Fortunately, the two formalisms described above can be combined into one formalism that has the expressive power of both (with the sole cost of inventing another unfortunate acronym).

Syntax and semantics

In the Global-and-Local-as-View formalism (GLAV) the expressions in the schema mapping include a query over the data sources on the left hand side, and a query on the mediated schema on the right-hand side. Formally, GLAV is defined as follows.

Definition 5.6: *(GLAV schema mapping) Let G be a mediated schema, and let $\bar{S} = \{S_1, \ldots, S_n\}$ be schemata of n data sources. A GLAV schema mapping M is a set of expressions of the form $Q^S(\bar{X}) \subseteq Q^G(\bar{X})$ or $Q^S(\bar{X}) = Q^G(\bar{X})$ where:*

- *Q^G is a query over the mediated schema G whose head variables are $\bar{X}$, and*
- *Q^S is a query over the data sources whose head variables are also $\bar{X}$.* □

Example 5.6: Suppose source S1 was known to have only comedies produced after 1970, then we could describe it using the following GLAV expression. Note that here too we abuse the notation by omitting the heads of the Q^G's and the Q^S's:

S1.Movie(MID, title), S1.MovieDetail(MID, director, genre, year) $\subseteq$
Movie(title, director, "comedy", year), year $\geq$ 1970. □

The semantics of GLAV are defined by specifying the relation M_R defined by M.

Definition 5.7: *(GLAV semantics) Let $M = M_1, \ldots, M_l$ be a GLAV schema mapping between G and $\bar{S} = \{S_1, \ldots, S_n\}$, where M_i is of the form $Q^S(\bar{X}) \subseteq Q^G(\bar{X})$ or $Q^S(\bar{X}) = Q^G(\bar{X})$.*

Let g be an instance of the mediated schema G, and let $\bar{s} = s_1, \ldots, s_n$ be instances of $S_1, \ldots, S_n$, respectively. The tuple of instances $(g, s_1, \ldots, s_n)$ is in M_R if for every $1 \leq i \leq l$, the following holds:

- *if M_i is a $=$ expression, then $S_i(\bar{s}) = Q_i(g)$,*
- *if M_i is a $\subseteq$ expression, then then $S_i(\bar{s}) \subseteq Q_i(g)$.* □

Reformulation in GLAV

Reformulation in GLAV amounts to composing the LAV techniques with the GAV techniques. Given a query Q, it can be reformulated in the following two steps:

- Find a rewriting Q' of the query Q using the views $Q_1^G, \ldots, Q_l^G$,
- Create Q'' by replacing every occurrence of Q_i^G in Q' with Q_i^S, and unfolding the result so it mentions only the source relations.

Applying Q'' to the source relations will yield all the certain answers in the cases specified in the theorem below. Consequently, the complexity of finding the certain answers and of finding a logical query plan in GLAV is the same as that for LAV.

Theorem 5.3: *Let $\bar{M} = M_1, \ldots, M_l$ be a GLAV schema mapping between a mediated schema G and source schemas $\bar{S} = \{S_1, \ldots, S_n\}$, where M_i is of the form $Q^S(\bar{X}) \subseteq Q^G(\bar{X})$ or $Q^S(\bar{X}) = Q^G(\bar{X})$, and assume that each relation in the mediated schema or in the sources appears in at most one M_i. Let Q be a conjunctive query over G.*

Assume that the Q_i^S's are conjunctive queries or unions of conjunctive queries, even with interpreted predicates and negated predicates.

- *If all the Q_i^G's in M are conjunctive queries with no interpreted predicates or negation, and all the M_i are $\subseteq$ expressions, then all the certain answers can be found in time polynomial in the size of the data and of the size of M, and the complexity of reformulation is polynomial in the number of data sources.*

- *If all the Q_i^G's in M are conjunctive queries with no interpreted predicates or negation, and some of the M_i's are $=$ expressions, then finding all the certain answers to Q is co-NP hard in the size of the data.*

- *If some of the Q_i^G's include interpreted predicates, then finding all then finding all certain answers to Q is co-NP hard in the size of the data.* □

The reformulations created as described above produce only certain answers when some of the relations occur in more than one M_i, but may not produce all the certain answers. We note that in practice, the real power of GLAV is the ability to use both GAV and LAV descriptions, even if none of the source descriptions uses the power of both.

5.3 Access-pattern limitations

Thus far, the logical plans we have generated assumed that we can access the relations in the data sources in any way we want. In particular, this means the data integration system can choose any *order* it deems most efficient to access the data sources, and is free to pose any query to each source. In practice, there are often significant limitations on the allowable access patterns to data sources. The primary examples of such limitations involve sources served by forms on the web and data available through specific interfaces defined by web services. Typically, such interfaces define a set of required inputs that must be given in order to obtain an answer, and it is rarely possible to obtain *all* the tuples from such sources. In some other cases, limitations on access patterns are imposed in order to restrict the queries that can be asked of a source and therefore limit the load on the source.

This section begins by discussing how to model limitations on access patterns to data sources, and then describes how to refine a logical query plan into an *executable* plan that adheres to these limitations. We will see that access-pattern limitations can have subtle effects on query plans.

5.3.1 Modeling access-pattern limitations

We model access-pattern limitations by attaching *adornments* to relations of data sources. Specifically, if a source relation has n attributes, then an adornment consists of a string of length n, composed of the letters b and f. The meaning of the letter b in an adornment is that the source *must* be given values for the attribute in that position. An f adornment means that the source does not need a value for that position. If there are multiple sets of allowable inputs to the source, we attach several adornments to the source.

Example 5.7: To illustrate the concepts in this section we use an example in the domain of publications and citations. Consider a mediated schema that includes the following relations: Cites stores pairs of publications identifiers (X, Y), where publication X cites publication Y, AwardPaper stores the identifiers of papers that received an award, and DBPapers stores the identifiers of papers in the field of Databases.

The following LAV expressions also express the access-pattern limitations to the sources:

S1: CitationDB^{bf}(X,Y) $\subseteq$ Cites(X,Y)
S2: CitingPapers^{f}(X) $\subseteq$ Cites(X,Y)
S3: DBSource^{f}(X) $\subseteq$ DBpapers(X)
S4: AwardDB^{b}(X) $\subseteq$ AwardPaper(X)

The first source stores pairs of citations where the first paper cites the second, but requires that the citing paper be given as input (hence the *bf* adornment). The second source stores all the papers that cite some paper, and enables to query for all such papers. The third source stores papers in the database field, but does not have any access restrictions, and the fourth source stores all the papers that won awards, but requires that the identifier of the paper be given as input. That is, you can ask the source if a particular paper won an award, but cannot ask for all award papers. □

5.3.2 Generating executable plans

Given a set of access-pattern limitations, we need to generate logical plans that are executable. Intuitively, an executable query plan is one in which we can always supply values to data sources when they are required. Hence, a key aspect of executable plans is the order of its subgoals. Formally, executable query plans are defined as follows.

Definition 5.8: *(Executable query plans) Let $q_1(\bar{X}_1), \ldots, q_n(\bar{X}_n)$ be a conjunctive query plan over a set of data sources, and let BF_i be the set of adornments describing the access-pattern limitations to the source q_i.*

We say that $q_1(\bar{X}_1), \ldots, q_n(\bar{X}_n)$ is an executable plan if there is a choice of adornments $bf_1, \ldots, bf_n$, such that

- $bf_i \in BF_i$ *and*
- *if the variable* X *appears in position* k *of* $q_i(\bar{X}_i)$ *and the* k*'th letter of* bf_i *is* b*, then* X *appears in a subgoal* $q_j(\bar{X}_j)$ *where* $j < i$. □

Note that changing the order of the subgoals in the plan does not affect the results. Figure 5.2 shows how to find an executable re-ordering of a given logical query plan with a simple greedy algorithm. Intuitively, the algorithm orders the subgoals in the plan beginning with those who have a completely free adornment (i.e., all f's), and then iteratively adds subgoals whose requirements are satisfied by subgoals earlier in the plan. If the algorithm manages to insert all the subgoals into the plan, then it is executable. Otherwise, there is no executable ordering of the subgoals.

Algorithm FindExecutablePlan(Q**,** BF**)**

/$*$ Q is a logical query plan of the form: $g_1(\bar{X}_1), \ldots, g_n(\bar{X}_n)$
/$*$ $BF = BF_1, \ldots, BF_n$, where BF_i is a set of adornments for g_i.

EP = empty list. /$*$ EP is the resulting plan.
for $i = 1, \ldots, n$, initialize $AD_i = BF_i$.
    /$*$ As we add subgoals to the plan, AD_i records their new allowable access patterns.

until no new subgoals can be added to EP **do**
    Choose a subgoal $q_i(\bar{X}_i) \in Q$, such that AD_i has an adornment that is all f's and $q_i(\bar{X}_i) \notin EP$.
    Add $q_i(\bar{X}_i)$ to the end of EP.
    for every variable $X \in \bar{X}_i$
        if X appears in position k of $g_l(\bar{X}_l)$ and position k of an adornment $ad \in AD_i$ is b,
            then change position k to f.

if all the subgoals in Q are in EP, **return** EP as an executable plan
else return No executable ordering.

Figure 5.2: An algorithm for finding an executable ordering of a logical query plan.

When we cannot find an executable ordering of the subgoals in a query plan, then the question is whether we can *add* subgoals to make the plan executable, and whether the new plan is guaranteed to find all the certain answers.

Example 5.8: Consider the following query over our mediated schema, asking for all the papers citing paper with identifier #001:

Q(X) :- Cites(X,001)

Ignoring the access-pattern limitations, the following plan would suffice

Q′(X) :- CitationDB(X,001).

However, that plan is not executable because CitationDB requires an input to its first field. Fortunately, the following longer plan is executable:

q(X) :- CitingPapers(X), CitationDB(X,001).

□

The above example showed that it is possible to add subgoals to the plan to obtain an executable plan. The following example shows that there may not be any limit on the length of such a plan!

Example 5.9: Consider the following query, asking for all papers that won awards, and let's ignore S2 for the purpose of this example.

Q(X) :- AwardPaper(X).

Since the view AwardDB requires its input to be bound, we cannot query it without a binding. Instead, we need to find candidate award papers. One way to find candidates is to query the source DBSource, obtaining all database papers, and feed these papers to the source AwardDB. Another set of candidates can be computed by papers cited by database papers, i.e., joining DBSource with the source CitationDB. In fact, for any integer n, we can begin by finding papers reachable by citations chains of length n starting from database papers, and then querying AwardDB to see whether any of these papers won an award. As we see below, we can create a logical query plan for each such n, and we can never decide without querying the data that we can stop at a certain n. Hence, since we need to create the executable query plan based on the source descriptions and the query, there is no bound on the length of such a plan.

Q′(X) :- DBSource(X), AwardDB(X)
Q′(X) :- DBSource(V), CitationDB(V,X_1), $\ldots$, CitationDB(X_n,X), AwardDB(X).

□

Fortunately, even if there is no bound on the length of a query plan, there is a compact *recursive* query plan that is executable and that will obtain all the possible answers. A recursive query plan is a datalog program whose base predicates are the data sources, and is allowed to compute intermediate relations in addition to the query relation. Let us first see how to construct a recursive plan for our example.

Example 5.10: The key to constructing the recursive plan is to define a new intermediate relation papers whose extension is the set of all papers reachable by citation chains from database papers. The papers relation is defined by the first two rules below. The third rule joins the papers relation with the AwardDB relation. Note that each of the rules in the plan is executable.

papers(X) :- DBsource(X)
papers(X) :- papers(Y), CitationDB(Y,X)
Q′(X) :- papers(X), AwardDB(X).

□

We now describe how to build such a recursive plan in the general case. We describe the construction for the case in which every source has a single adornment, but the generalization for multiple adornments is straightforward. Given a logical query plan Q over a set of sources $S_1, \ldots, S_n$, we create an executable query plan in two steps.

Step 1: We define an intermediate (IDB) relation Dom that will include all the constants in the domain that we can obtained from the sources. Let $S_i(X_1, \ldots, X_k)$ be a data source, and assume without loss of generality that the adornment of S_i requires that arguments $X_1, \ldots, X_l$ (for $l \leq k$) must be bound and the rest are free. We add the following rules, for $l + 1 \leq j \leq k$:

Dom(X_j) :- Dom(X_1), . . . , Dom(X_l), S$_i(X_1, \ldots, X_k)$

Note that at least one of the sources must have an adornment that is all f's, otherwise we cannot answer any query. Those sources will provide the base case rules for Dom.

Step 2: We modify the original query plan by inserting atoms of the Dom relation as necessary. Specifically, for every variable X in the plan, let k be the first subgoal in which it appears. If the adornment of $q_k(\bar{X}_k)$ has a b in any of the positions that X appears in $q_k(\bar{X}_k)$, then we insert the atom Dom(X) in front of $q_k(\bar{X}_k)$. □

The above algorithm is obviously inefficient in many cases. In practice, the Dom relation need only include values for columns that need to be bound in some source. Furthermore, we can refine the Dom relation into multiple relations, each one containing the constants relevant to a particular column (e.g., we can create one relation for movie names and another for city names). Finally, we note that in practice we often have a reasonable list of constants in particular domains. For example, there are many geographical databases with lists of country and city names, and good collections of movie names. In these cases, we can use these lists in place of Dom.

5.4 Integrity constraints on the mediated schema

When we design a mediated schema, we often have additional knowledge about the domain. We express such knowledge in the form of integrity constraints, such as functional dependencies or inclusion constraints. This section shows how the presence of integrity constraints in the mediated schema affects the query plans we need to create in order to obtain all certain answers. Integrity constraints affect both LAV and GAV source descriptions.

5.4.1 LAV with Integrity Constraints

The following example illustrates the complications that arise when we have integrity constraints in LAV.

Example 5.11: Consider a mediated schema that includes a single relation representing flight schedule information, including the pilot and aircraft that are planned for each flight.

Schedule(airline, flightNum, date, pilot, aircraft)

Suppose we have the following functional dependencies on the Schedule relation:

Pilot → Airline and Aircraft → Airline.

The first functional dependency expresses the fact that pilots work for only one airline, and second specifies that there is no joint ownership of aircraft between airlines. Suppose we have the following LAV schema mapping of a source S:

$$\mathsf{S(date, pilot, aircraft)} \subseteq \mathsf{schedule(airline, flightNum, date, pilot, aircraft)}$$

The source S records the dates on which pilots flew different aircraft. Now suppose a user asks for pilots that work for the same airline as Mike:

$$\mathsf{q(p) \text{ :- } schedule(airline, flightNum, date, ``mike", aircraft), schedule(airline, f, d, p, a)}$$

Source S doesn't record the airlines that pilots work for, and therefore, without any further knowledge, we cannot compute any answers to the query q. Nonetheless, using the functional dependencies of relation schedule, conclusions can be drawn on 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 then Ann and John work for the same airline because of the functional dependency Aircraft $\rightarrow$ Airline. Hence, we can infer that John and Mike work for the same airline. In general, we can consider any logical query plan, q'_n, of the following form:

$$\begin{aligned} q'_n(p) \text{ :- } & S(D_1, ``mike", C_1), S(D_2, p_2, C_1), S(D_3, p_2, C_2), S(D_4, p_3, C_2), \ldots, \\ & S(D_{2n-2}, p_n, C_{n-1}), S(D_{2n-1}, p_n, C_n), S(D_{2n}, p, C_n) \end{aligned}$$

For any n, q'_n may yield results that shorter plans did not, and therefore there is no limit on the length of a logical query plan that we need to consider. Fortunately, as in the case of access-pattern limitations, recursive query plans come to our rescue. We now describe the construction of a recursive query plan that is guaranteed to produce all certain answers even in the presence of functional dependencies. □

date	pilot	aircraft
1/1	Mike	#111
5/2	Ann	#111
1/3	Ann	#222
4/3	John	#222

Figure 5.3: A database of pilots' schedules.

The input to the construction is the logical query plan, q', generated by the Inverse-rules Algorithm (Section 4.3.5). The inverse rule created for source S is shown below. Recall that f_1 and f_2 are Skolem functions and they are used to represent objects about which we have incomplete information.

schedule(f_1(d,p,a), f_2(d,p,a), d, p, a) :- S(d, p, a)

The inverse rules alone don't take into account the presence of the functional dependencies. For example, applying the inverse rule on the on the table shown in Figure 5.3 would yield the following tuples:

schedule(f_1(1/1, Mike, #111), f_2(1/1, Mike, #111), 1/1, Mike, #111)
schedule(f_1(5/2, Ann, #111), f_2(5/2, Ann, #111), 5/2, Ann, #111)
schedule(f_1(1/3, Ann, #222), f_2(1/3, Ann, #222), 1/3, Ann, #222)
schedule(f_1(4/3, John, #222), f_2(4/3, John, #222), 4/3, John, #222)

Because of the functional dependencies on the schedule relation, it is possible to conclude that f_1(1/1, Mike, #111) is equal to f_1(5/2, Ann, #111), and that both are equal to f_1(1/3, Ann, #222) and f_1(4/3, John, #222).

We enable the recursive query plan to make such inferences 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 constants 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),...,e(A_n,A'_n)).

Definition 5.9: *(chase rules) Let* $\bar{A} \rightarrow$ B *be a functional dependency satisfied by a relation* p *in the mediated schema. Let* $\bar{C}$ *be the attributes of* p *that are not in* $\bar{A}, B$. *The* chase rule *corresponding to* $\bar{A} \rightarrow B$ *is the following:*

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

□

Given a set of functional dependencies Σ on the mediated schema, 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(X,Y) :- schedule(X, F, P, D, A), schedule(Y, F′, P′, D′, A′), e(A, A′)
e(X,Y) :- schedule(X, F, P, D, A), schedule(Y, F′, P′, D′, A′), e(P, P′)

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

e(f_1(1/1, Mike, #111), f_1(5/2, Ann, #111))
e(f_1(5/2, Ann, #111), f_1(1/3, Ann, #222))
e(f_1(1/3, Ann, #222), f_1(4/3, John, #222))

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, we add the following rule that enforces the transitivity of e.

```
T: e(X,Y) :- e(X,Z), e(Z,Y).
```

The final step in the construction is to rewrite the query q′ in a way that it can use the equivalences derived in relation e. We initialize q″ to q′ and apply the following transformations:

1. If c is a constant in one of the subgoals of 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 q″, we replace X in the body of 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 q″ appears in two subgoals of 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 q″. In our example query q′ would be rewritten to:

```
q″(P) :- schedule(A, F, D, M, C), schedule(A′, F′, D′, P′, C′), e(M, "mike"),
         e(P′, P), e(A, A′)
```

The resulting query plan includes q″, the chase rules, and the transitivity rule T. It can be shown that the above construction is guaranteed to yield all the certain answers to the query in the presence of functional dependencies. The bibliographic notes include a reference to the full proof of this claim.

5.4.2 GAV with Integrity Constraints

A key property of GAV schema mappings is that unlike LAV mappings, they do not model incomplete information. Given a set of data sources and a GAV schema mapping, there is a single instance of the mediated schema that is consistent with the sources, thereby considerably simplifying query processing. With integrity constraints, as the following example illustrates, this property no longer holds.

flight

flightNum	origin	destination
222	Seattle	SFO
333	SFO	Saigon

schedule

airline	flightNum	date	pilot	aircraft
United	#111	1/1	Mike	Boeing777-15
Singapore Airlines	#222	1/3	Ann	Boeing777-17

Table 5.1: Pilot and flight schedules.

Example 5.12: Suppose that in addition to the schedule relation, we also had a relation flight(flightNum, origin, destination) storing the beginning and ending points of every flight. In addition, we have the following integrity constraint stating that every flight number appearing in the schedule relation must also appear in the flight relation:

schedule(flightNum) $\subseteq$ flight(flightNum).

Assume that we have two sources whose mappings are trivial (see Table 5.1). Each source provides tuples for one of the relations in the mediated schema. Consider the query asking for all flight numbers:

q(fN) :- schedule(airline, fN, date, pilot, aircraft), flight(fN, origin, destination)

In GAV we would answer this query by unfolding it and joining the two tables in Table 5.1. However, that join would not yield the correct answer. Specifically, flight #111 appears in the schedule relation, but not in flight, and therefore would not appear in the result. We do not know exactly the flight details of flight#111, but we know that one has to exist, and therefore the answer to q should include #111. Note that this situation arises if we make the open-world assumption. If we made the closed-world assumption, the data sources in this example would be inconsistent with the schema mapping. □

Using techniques similar to those in LAV, there is a way to extend the logical query plans to ensure that we obtain all the certain answers. We refer the reader to the bibliographic notes for futher details.

5.5 Answer Completeness

We've already seen that the schema mapping formalisms can express completeness of data sources, and we've also seen how completeness of data sources can affect the complexity of finding the certain answers. Knowing that sources are complete is useful for creating more efficient query answering plans. In particular, if there are several sources that contain similar data, then unless we know that one of them is complete, we need to query them all in order to get all the possible answers. This section considers a more refined notion of completeness, called *local completeness*.

Local completeness

In practice, sources are often *locally* complete. For example, a movie database may be complete w.r.t. more recent movies, but incomplete w.r.t. earlier ones. The following example shows how we can extend LAV expressions with local completeness information. Similarly, we can also describe local completeness in GAV.

Example 5.13: Recall that in LAV, sources S5–S7 were described as follows:

S5.MovieGenres(title, genre) $\subseteq$ Movie(title, director, year, genre)
S6.MovieDirectors(title, dir) $\subseteq$ Movie(title, director, year, genre)
S7.MovieYears(title, year) $\subseteq$ Movie(title, director, year, genre)

We can add the following local completeness (LC) descriptions:

LC(S5.MovieGenres(title, genre), genre="comedy")
LC(S6.MovieDirectors(title, dir), American(director))
LC(S7.MovieYears(title, year), year $\geq$ 1980)

The above assertions express that S5 is complete w.r.t. comedies, S6 is complete for American directors (where American is a relation in the mediated schema), and that S7 is complete w.r.t. movies produced in 1980 or after. □

Formally, we specify local-completeness statements by specifying a constraint on the tuples of a source.

Definition 5.10: *(Constraint) Let* M *be a LAV expression of the form* S($\bar{X}$) $\subseteq$ Q($\bar{X}$)*, where* S *is a data source and* Q($\bar{X}$) *is a conjunctive query over the mediated schema. A constraint C on* M *is a conjunction of atoms of relations in the mediated schema or of atoms of interpreted predicates that does not include relation names mentioned in* Q*. The atoms may include variables in $\bar{X}$ or new ones. We denote the complement of C by $\neg C$.* □

The semantics of the local-completeness expression LC(S, C) is that in addition to the original expression, we also have the following expression in the schema mapping. Note that we add the conjuncts of C to Q.

S($\bar{X}$) = Q($\bar{X}$), C.

When schema mappings can include local-completeness statements, a natural question to ask is the following: given a query over the mediated schema, is the answer to the query guaranteed to be complete?

Example 5.14: Consider the following two queries over the sources in Example 5.13:

q$_1$(title) :- Movie(title, director, genre, "comedy"), year $\geq$ 1990, American(director)
q$_2$(title) :- Movie(title, director, genre, "comedy"), year $\geq$ 1970, American(director)

The answer to q$_1$ is guaranteed to be complete because it only touches on complete parts of the sources: comedies by American directors produced after 1980. On the other hand, the answer to q$_2$ may not be complete if the source S7 is missing movies produced between 1970 and 1980. □

Formally, we define answer-completeness as follows. Intuitively, the definition says that whenever two instances of the data sources agree on the tuples for which the sources are known to be locally complete, they will have the same certain answers.

Definition 5.11: *(answer-completeness) Let M be a LAV schema mapping for sources $S_1, \ldots, S_n$ that includes a set of expressions of the form $S_i(\bar{X}_i) \subseteq Q_i(\bar{X}_i)$, and a set of local-completeness assertions of the form: $LC(S_i, C_i)$. Let Q be a conjunctive query over the mediated schema.*

The query Q is answer-complete w.r.t. M if for any pair instances d_1, d_2 of the data sources, such that for every i, if d_1 and d_2 have the same tuples of S_i satisfying C_i, then the certain answers to Q over d_1 are the same as the certain answers to Q over d_2. □

In the context of GAV, an important special case of answer completeness is to determine whether we have complete data for a relation in the mediated schema given the sources.

Detecting answer-completeness

We now describe an algorithm for deciding when a query is answer complete. To focus on the interesting aspects of the algorithm, we consider a simplified setting. We assume that data sources correspond directly to relations in the mediated schema, augmented with a conjunction of interpreted atoms. Specifically, we assume LAV expressions of the form:

$$S_i(\bar{X}) \subseteq R_i(\bar{X}), C'$$

where R_i is a relation in the mediated schema and C' is a conjunction of atoms of interpreted predicates.

Figure 5.4 shows the algorithm for determining answer completeness by reducing it to the problem of query containment. The intuition behind the algorithm is the following. Since the sources S_i are complete for tuples satisfying C_i, the only tuples in S_i that may be missing are ones that satisfy $\neg C_i$. We define the view V_i to be the relation that includes the tuples of S_i that satisfy C_i and tuples that may be missing from S_i. The view V_i obtains the tuples satisfying C_i from S_i (since S_i has them all) and the tuples satisfying $\neg C_i$ from a new relation E_i whose tuples we don't know. Note that with appropriate extensions for the E_i's it is possible to create an instance of V_i that is equal to any possible instance of S_i.

The algorithm then compares the query Q with the query Q' in which occurrences of S_i are replaced with V_i. If the two queries are equivalent, then for any instance of the S_i's and the E_i's we obtain the same result. Since Q does not depend on the E_i's, this means that Q is completely determined by the tuples of S_i that satisfy C_i.

The following theorem establishes the correctness of Algorithm **decide-completeness**.

Theorem 5.4: *Let M be a LAV schema mapping for sources $S_1, \ldots, S_n$ that includes the expressions $S_i(\bar{X}_i) \subseteq R(\bar{X}_i), C'_i$, and a set of local-completeness assertions $LC(S_i, C_i)$. Let Q be a conjunctive query over the mediated schema.*

Algorithm **decide-completeness** *returns* **yes** *if and only Q is answer-complete w.r.t. M.* □

Proof: For the first direction, suppose Q_1 is not equivalent to Q. We show that Q is not is answer-complete w.r.t. M.

Since $Q_1 \not\equiv Q$, there is a database instance d on which Q and Q_1 return different answers. Let d_1 be the instance of the data sources where the extension

algorithm decide-completeness(Q, M)
/* Q is a conjunctive query over the sources $S_1, \ldots, S_n$;
/* M includes the LAV expressions: $S_i(\bar{X}_i) \subseteq R(\bar{X}_i), C'_i$.
and the local-completeness assertions: $LC(S_i, C_i)$.
The procedure returns *yes* if and only if Q is answer-complete w.r.t. M. */

Let $E_1, \ldots, E_n$ be new relation symbols.
Define the views $V_1, \ldots, V_n$ as follows:
$V_i(\bar{X}_i) : -E_i(\bar{X}_i), \neg C_i$.
$V_i(\bar{X}_i) : -S_i(\bar{X}_i), C_i$.
Let Q_1 be the query in which every occurrence of S_i in Q is replaced by V_i.
return *yes* if and only if Q is equivalent to Q_1.
end.

Figure 5.4: An algorithm for detecting answer-completeness of a query.

of S_i is its extension in d. Let d_2 be the instance of the data sources in which the extension of S_i is the extension of V_i in d. The instances d_1 and d_2 agree on the tuples of S_i that satisfy C_i, for $1 \leq i \leq n$, but do not agree on the certain answers to Q, and therefore Q is not answer-complete w.r.t. M.

For the other direction, assume $Q_1 \equiv Q$. Let d_1 and d_2 be two instances of the sources that agree on the tuples of S_i that satisfy C_i. Define d_3 to be the restriction of d_1 (and hence also d_2) that include only the tuples of S_i that satisfy C_i. Since $Q_1 \equiv Q$ it is easy to see that $Q(d_1) = Q(d_3)$ and similarly that $Q(d_2) = Q(d_3)$. Hence, $Q(d_1) = Q(d_2)$. □

5.6 Data-Level Heterogeneity

The schema mappings described thus far assumed that whenever an expression in the mapping requires joining tuples from different sources, then the join columns will have comparable values. For example, in the movie domain, we assumed that whenever a movie occurs in a source, it occurs with the same title string as in all other sources in which it appears.

In practice, sources differ not only on the structure of their schema, but they also differ considerably on how they represent values and objects in the world. We refer to these differences as *data-level heterogeneity*. Data-level heterogeneity can be broadly classified into two types.

Differences of scale

The first kind of data-level heterogeneity occurs when there is some mathematical transformation between the values in one source and the other. Examples of this type of heterogeneity are when one source represents temperatures in Celsius while does it in Fahrenheit, or when one source represents course grades on a numerical ladder while the other uses letter grades. In some cases the transformation may require values from different columns. For example, one source may represent first name and last name in two different columns while another may concatentate them in one column. In other cases, the transformation may require deeper knowledge of the semantics. For example, in one database prices may include local taxes while in another they do not. It is important to keep in mind that these transformations are not always reversible, therefore limiting the queries that can be answered in some cases.

This kind of data-level heterogeneity can be reconciled by adding the transformation function to the expression in the schema mapping. For example, the first expression below translates the temperatures in the source from Fahrenheit to Celsius, and in the second expression, we adjust the price obtained from the source to include the local taxes.

S(city, temp -32 * 5/9, month) $\subseteq$ Weather(city, temp, humidity, month)
CDStore(cd, price) $\subseteq$ CDPrices(cd, state, price * (1+rate)), LocalTaxes(state, rate).

Multiple references to the same entity

The second kind of data-level heterogeneity occurs when there are multiple ways of referring to the same object in the real world. Common examples of this case include different ways of referring to companies (e.g., IBM versus International Business Machines, or Google versus Google Inc.), and people (e.g., Jack M. Smith versus J. M. Smith). Reconciling multiple references to the same real-world entity gets more complicated when referring to complex objects. For example a reference to a publication includes a list of references to authors, a title, a year of publication, and a reference to venue of the publication. Furthermore, data need not always be clean, complicating the reconciliation problem even further. In some cases, we don't even know the exact truth. For example, biologists have many ways of referring to genes or species and it's not even known how to resolve all the references.

To resolve this kind of heterogeneity we create a *concordance table*, whose rows include multiple references to the same object. Specifically, the first column includes references from the first source and the second includes references from the second source. When we join the two sources, we perform an intermediate join with the concordance table to obtain the correct answers. For example, Figure 5.6 shows a concordance table for two sources that describe different attributes of countries.

Clearly, the hard problem is to create the concordance table in applications where there are large number of rows. We focus the problem of automatic reference reconciliation in Chapter 8.

Country GDPs	**Country Water Access**
Congo, Republic of the	Congo (Dem. Rep.)
Korea, South	South Korea
Isle of Man	Man, Isle of
Burma	Myanmar
Virgin Islands	Virgin Islands of teh US
Palestinian Territories	West Bank

5.7 Bibliographic Notes

Global-as-View was used in the earliest data integration systems (e.g., Multibase [197]) and several systems since then [61, 116, 147, 279]. Local-as-View was introduced by the Information Manifold System [206] and used in [99, 194] and others. GLAV was introduced in [122], and is the main formalism used in data exchange systems (see Chapter 16). In fact, GLAV is essentially a formalism for specifying tuple-generating dependencies [4], and hence some of the theory on integrity constraints can be applied to data integration and exchange. The development of multiple schema mapping languages lead to several comparisons between the formalisms [200, 203, 290].

Restrictions on access patterns were first discussed by Rajaraman et al. [261]. They considered the problem of answering queries using views when views have adornments describing the limitations on access patterns. They showed that in the case of looking for an *equivalent* rewriting of a query using views, the bound on the length of the rewriting established by [204] (namely, the number of subgoals in the query) no longer holds. Instead, they established a new bound which is the sum of the number of subgoals and of the number of variables in the query. Kwok and Weld [194] showed that if we are looking for the maximally-contained rewriting of a query in the presence of access-pattern limitations, then there may be no bound on its length. The recursive query plan that generates all the certain answers was described by Duschka et al. [97, 100]. Friedman and Weld [121] and Lambrecht et al. [196] describe how to further optimize the recursive query plan. Manolescu et al. [114] show the effects of binding patterns on traditional System-R style query optimization. Levy et al. [206] describe a more complex model for access-pattern limitations, where in addition to modeling the input, we also model the output. Then, if the sources have more than one capability record, they show that the problem of finding an executable plan is NP-Complete.

Integrity constraints in the mediated schema and their effects on finding the certain answers was first discussed in the context of LAV in Duschka et al. [97, 99]. There the authors show that it is possible to find all the certain answers to a query even if there are functional dependencies and/or full dependencies on the mediated schema. Our examples are taken from [97]. Integrity constraints on GAV were considered by Cali et al. [47], where the authors showed that it is possible to find all the certain answers in the presence of key constraints and inclusion dependencies.

The study of answer-completeness in databases goes back to Motro [241]. Etzioni et al. [105] introduced the local-completeness notation in the context of information gathering agents and showed some of its basic properties. The completeness algorithm we described is based on [202]. There it is shown that the problem of determining completeness can be reformulated as the problem of determining independence of queries from updates [38, 101, 102, 207], which, in turn, can be reduced to a containment checking problem [207]. Floresecu et al. [113] described a formalism for expressing degree of completeness via a probability that a tuple will be in a source. Probabilistic approaches to schema mapping are now gaining attention again, and we discuss them further in Chapter 13.