http://webdam.inria.fr/

Web Data Management

Data Integration

Serge Abiteboul
INRIA Saclay & ENS Cachan

Ioana Manolescu
INRIA Saclay & Paris-Sud University

Philippe Rigaux
CNAM Paris & INRIA Saclay

Marie-Christine Rousset
Grenoble University

Pierre Senellart
Télécom ParisTech

Copyright @2011 by Serge Abiteboul, Ioana Manolescu, Philippe Rigaux, Marie-Christine Rousset, Pierre Senellart;
to be published by Cambridge University Press 2011. For personal use only, not for distribution.

http://webdam.inria.fr/Jorge/

Contents

1 Introduction

The goal of data integration is to provide a uniform access to a set of autonomous and possibly heterogeneous data sources in a particular application domain. This is typically what we need when, for instance, querying the *deep web* that is composed of a plethora of databases accessible through Web forms. We would like to be able with a single query to find relevant data no matter which database provides it.

A first issue for data integration (that will be ignored here) is social: The owners of some data set may be unwilling to fully share it and be reluctant to participate in a data integration system. Also, from a technical viewpoint, the difficulty comes from the lack of interoperability between the data sources, that may use a variety of formats, specific query processing capabilities, different protocols. However, the real bottleneck for data integration is logical. It comes from the so-called *semantic heterogeneity* between the data sources. They typically organize data using different schemas even in the same application domain. For instance, each university or educational institution may choose to model students and teaching programs in its own way. A French university may use the social security number to identify students and the attributes NOM, PRENOM, whereas the Erasmus database about European students may use a European student number and the attributes FIRSTNAME, LASTNAME and HOME UNIVERSITY.

@ Serge Abiteboul, Ioana Manolescu, Philippe Rigaux, Marie-Christine Rousset, Pierre Senellart, 2011; to be published by Cambridge University Press 2011.

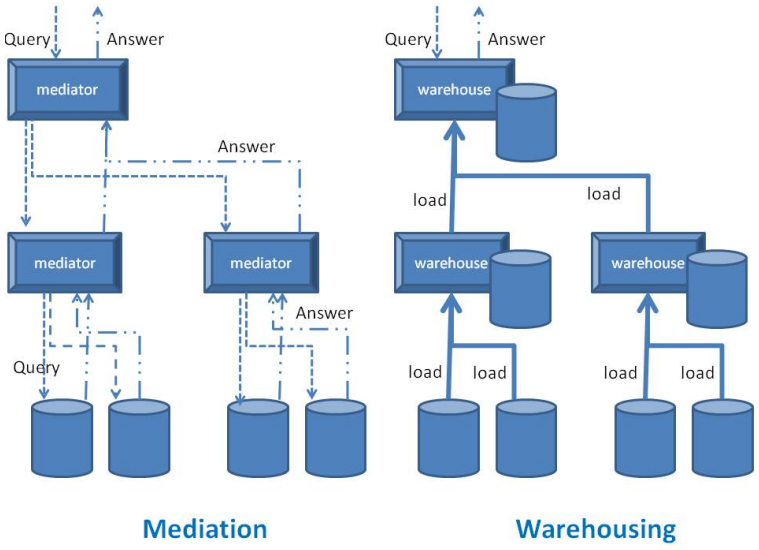

Figure 1: Virtual versus Materialized data integration

In this chapter, we study data integration in the *mediator* approach. In this approach, data remain exclusively in data sources and are obtained when the system is queried. One sometimes use the term *virtual* data integration. This is in contrast to a *warehousing* approach where the data is extracted from the data sources ahead of query time, transformed, and loaded in the warehouse. At query time, the warehouse is accessed but not the data sources. Warehouses approaches are typically preferred for very complex queries, e.g., for data mining. On the other hand, to have access to "fresh" information, a mediator approach is preferred since it avoids having to propagate in real time, data source updates to the warehouse. Figure 1 illustrates these two approaches of data integration.

In the *mediator* approach, one starts by designing a global schema (also called *mediated* schema) that serves as a unique entry point on which global queries are posed by users. A main issue is then to specify the relationships, namely *semantic mappings*, between the schemas of the data sources and the global schema. Based on these mappings, one can answer queries over the global schema using queries over the data sources. Typically, query answering in the mediator approach is performed as follows. First, independently of the data in the sources, the user's query posed over the global schema is transformed into *local queries* that refer to the schemas of the data sources. A *global query* combines the data provided by sources. Queries are optimized and transformed into query plans. The local query plans are executed and their results combined by the global query plan.

In the following, for presentation purposes, we consider that the global schema and the schemas of the data sources to integrate are all relational. In practice, each non-relational data source (e.g., XML or HTML) is abstracted as a relational database with the help of a *wrapper*. Wrappers are small programs that translate local relational queries into appropriate requests understood by specific data sources, and transform their results into relations. The role of wrappers is to allow the mediator to see each data source as relational, no matter which actual format it uses.

@ Serge Abiteboul, Ioana Manolescu, Philippe Rigaux, Marie-Christine Rousset, Pierre Senellart, 2011; to be published by Cambridge University Press 2011.

Let us consider in more detail the specification of semantic mappings between the data sources and the global schema. Let S_1, ..., S_n be the local schemas of n pre-existing data sources. To simplify the presentation, let us assume that each local schema S_i is made of a single relation that we denote also S_i. The relations S_1, ..., S_n are called the *local* relations. Suppose the global schema G consists of the *global* relations G_1, ..., G_m. The goal is to specify semantic relations between the local relations S_i and the global relations G_j. The G_j are logically (intentionally) defined by the S_i.

An example of simple relationship (not very interesting) is based on equality, e.g., $G_1 = S_1$. One can find more complicated relationships, e.g., $G_2 = S_1 \cup S_2$ or $G_3 = S_1 \bowtie S_3$. In these last two examples, a global relation is defined as a query over the local relations. In other words, the global relation is a *view* of the local relations. Indeed, one typically prefers more flexible constraints such as $G_3 \supseteq S_1 \bowtie S_3$. Using containment instead of equality leaves open the possibility for other sources of providing data about G_3, e.g., $G_3 \supseteq \sigma_{A="yes"}(S_4)$. Because global relations are constrained by views of the local relations, one uses the term *global-as-view* for such specifications.

In a somewhat dual manner, one can use *local-as-view* constraints such as: $S_4 \subseteq G_1 \bowtie G_3$. This leaves even more flexibility since the contribution of each data source can be specified (e.g., by its owner) independently of the other sources of the system. This kind of autonomy is typically well-adapted to a Web setting.

More generally, to express semantic mappings between $\{S_1,...,S_n\}$ and $\{G_1,...,G_m\}$, one can use inclusion statements, i.e., logical constraints, of the form $v(S_1,...,S_n) \subseteq v'(G_1,...,G_m)$, where v and v' are query expressions called views. All the constraints we consider in this chapter will be of this general form. Now, given an instance I of $\{S_1,...,S_n\}$ (i.e., an instance of the data sources), we don't know the instance J of the global schema. But we know that:

$$v(I(S_1),...,I(S_n)) \subseteq v'(J(G_1),...,J(G_m))$$

So, the story of mediator systems is essentially a story of logical constraints and incomplete information. In this general setting, given I, an *answer* to a global query q is a fact $q(a)$ that is true in any instance J that together with I satisfies the mapping constraints, i.e., a fact we can be sure of as a logical consequence of both the data stored in I and of the logical constraints expressed by the mappings. Not surprisingly, query answering is thus a complex reasoning problem that in general may be undecidable. We focus on two particular decidable cases, for which rewriting algorithms have been designed and implemented. They are based on semantic mappings that capture typical constraints found in many applications:

Global-As-View (GAV for short). The semantic mappings are of the form

$$V_i(S_1,...,S_n) \subseteq G_i$$

also equivalently denoted

$$G_i \supseteq V_i(S_1,...,S_n)$$

where each V_i is a view over the local schemas, i.e., a query built on local relations.

Local-As-View (LAV for short). The semantic mappings are of the form

$$S_i \subseteq V_i(G_1,...,G_m)$$

where each V_i is a view over the global schema, i.e., a query built on global relations.

@ Serge Abiteboul, Ioana Manolescu, Philippe Rigaux, Marie-Christine Rousset, Pierre Senellart, 2011; to be published by Cambridge University Press 2011.

In our development, we will consider conjunctive queries. Using negation in queries greatly complicates the issues. In the next section, we recall some standard material on *containment* of conjunctive queries, i.e., of the queries at the heart of our formal development. In Sections 3 and 4, we study GAV and LAV mediators, respectively. For each of these languages, we describe appropriate query rewriting algorithms. In Section 5, we show the impact on query rewriting of adding DL-LITE constraints in the global schema. Finally, in Section 6, we lay the basis of a peer-to-peer approach for data integration. In contrast with the mediator approach which offers a unique entry point to data, peer-to-peer data management systems (PDMS for short) are decentralized data integration systems.

2 Containment of conjunctive queries

In this section, we recall some basic notions on comparing conjunctive queries that we will use in the following.

We recall that a *conjunctive query* is an expression of the form:

$$q(x_1,...x_n) \text{ :- } A_1(\vec{u_1}),...,A_k(\vec{u_k})$$

where each A_i is an relation, $\vec{u_1},...,\vec{u_k}$ are vectors of constants and variables. Furthermore, we require that each x_i occurs in some $\vec{u_i}$. $q(x_1,...x_n)$ is called the *head* and $A_1(\vec{u_1}),...,A_k(\vec{u_k})$ the *body* of the query. The x_i variables are called *distinguished*. The other variables are called *existential*.

Given an instance I of the relations appearing in the body of the query, an *answer* is a tuple $\langle \nu(x_1),...,\nu(x_n)\rangle$ for some valuation ν of the variables in the query, such that for each i, $A_i(\nu(\vec{u_i}))$ holds in I. We denote *q(I)* the set of answers.

We sometimes denote this query $q(x_1,...x_n)$ when its body is understood. Observe that the interpretation of such a conjunctive query in logical terms is:

$$\{x_1,...,x_n \mid \exists y_1,...,\exists y_m(A_1(\vec{u_1}) \wedge ... \wedge A_k(\vec{u_k}))\}$$

where $y_1,...,y_m$ are the variables not occurring in the head.

The data integration techniques rely on *conjunctive query containment*. This problem has been extensively studied because it is at the core of query optimization. We use known techniques that we recall next.

A query q_1 is contained in q_2, denoted $q_1 \subseteq q_2$, if for each I, $q_1(I) \subseteq q_2(I)$. It is known that the containment between a conjunctive query q_1 and a conjunctive query q_2 can be tested by finding a "homomorphism" from q_2 to q_1.

Definition 2.1 *Let $q_1(x_1,...,x_n)$ and $q_2(y_1,...,y_n)$ be two conjunctive queries. A* (conjunctive query) homomorphism *from q_2 to q_1 is a mapping ψ from the variables of q_2 to the variables of q_1 such that:*

1. *For each i, $\psi(y_i) = x_i$; and*
2. *For each atom $R(\vec{u_i})$ in the body of q_2, $R(\psi(\vec{u_i}))$ is in the body of q_1.*

Example 2.2 *Consider the following queries:*

- $q_1(x_1,x_1') : -A_1(x_1,x_2,x_3),A_2(x_1',x_2,x_3)$

@ Serge Abiteboul, Ioana Manolescu, Philippe Rigaux, Marie-Christine Rousset, Pierre Senellart, 2011; to be published by Cambridge University Press 2011.

- $q_2(y_1, y_1') : -A_1(y_1, y_2, y_3), A_2(y_1', y_2, y_3')$

Consider a mapping ψ such that $\psi(y_i) = x_i$ for each i, $\psi(y_1') = x_1'$ and $\psi(y_3') = x_3$. Then the required conditions hold, and it follows that $q_1 \subseteq q_2$. Intuitively, q_2 joins A_1 and A_2 on the second attribute, whereas q_1 also *joins on the third one. The additional condition induces the containment.*

The following proposition states that the existence of a homomorphism is a necessary and sufficient condition for query containment.

Proposition 2.3 *(*Homomorphism theorem*) Let q_1 and q_2 be two conjunctive queries. Then q_1 is contained in q_2 if and only if there exists a homomorphism from q_2 to q_1.*

This provides a simple algorithm for testing conjunctive query containment. In the general case, deciding whether a conjunctive query is contained in another one is NP-complete in the size of the two queries. In fact, in many practical cases, there are polynomial-time algorithms for query containment.

Algorithm 1 checks whether a query q_1 is contained in a query q_2.

Algorithm 1: The *Query containment algorithm*
$QC(q1, q_2)$
Input: Two conjunctive queries:
$q_1(\vec{x})$:- $g_1(\vec{x_1}), \ldots, g_n(\vec{x_n})$
$q_2(\vec{y})$:- $h_1(\vec{y_1}), \ldots, h_m(\vec{y_m})$
Output: Yes if $q_1 \subseteq q_2$; no otherwise
(1) **freeze** q_1**:** construct a canonical instance $D_{can} = \{g_i(\nu(\vec{x_i})) \mid 1 \leq i \leq n\}$
(2) for some valuation ν mapping each variable in q_1
(3) to a distinct constant
(4) **if** $\nu(\vec{x}) \in q_2(D_{can})$ **return** yes
(5) **else return** no.

Example 2.4 *Consider the queries of Example 2.2. The canonical instance D_{can} is $A_1(a, b, c), A_2(a', b, c)$. It is easily verified that $q_2(D_{can}) = (a, a')$, which is $\nu(x, x')$.*

3 Global-as-view mediation

The main advantage of GAV is its conceptual and algorithmic simplicity. The global schema is simply defined using views over the data sources and specifies how to obtain tuples of the global relation G_i from tuples in the sources.

Definition 3.1 (GAV mapping) *A* GAV mapping *is an expression of the form: $R(x_1, ..., x_n) \supseteq q(x_1, ..., x_n)$, where $q(x_1, ..., x_n)$:- $A_1(\vec{u_1}), ..., A_k(\vec{u_k})$ is a conjunctive query of the same arity as R. The* semantics *of this mapping is:*

$$\forall x_1, ..., x_n (\exists y_1, ..., y_m (A_1(\vec{u_1}), ..., A_k(\vec{u_k}) \Rightarrow R(\vec{u})))$$

where $y_1, ..., y_m$ are of variables occurring in the body of the rule and not its head.

@ Serge Abiteboul, Ioana Manolescu, Philippe Rigaux, Marie-Christine Rousset, Pierre Senellart, 2011; to be published by Cambridge University Press 2011.

We write alternatively this GAV mapping as:

$$R(x_1,...,x_n) \supseteq A_1(\vec{u_1}),...,A_k(\vec{u_k})$$
$$R(x_1,...,x_n) \supseteq q(x_1,...,x_n)$$
$$R \supseteq q$$

by omitting information that is either not needed or that is clear from the context. When we want to stress which are the existential variables, we write it $R(\vec{x}) \supseteq q(\vec{x},\vec{y})$ where $\vec{y}$ is the vector of existential variables.

Example 3.2 *Consider the following four data sources:*

- *The source relation* `S1` *is a catalog of teaching programs offered in different French universities with master programs.*

  ```
  S1.Catalogue(nomUniv, programme).
  ```

- *The source relation* `S2` *provides the names of European students enrolled in courses at some university within the Erasmus exchange program:*

  ```
  S2.Erasmus(student, course, univ).
  ```

- *The source relation* `S3` *provides the names of foreign students enrolled in programs of some French university:*

  ```
  S3.CampusFr(student, program, university).
  ```

- *The source relation* `S4` *provides the course contents of international master programs:*

  ```
  S4.Mundus(program,course)
  ```

Now, suppose we define a global *schema with the following unary and binary relations:*

```
MasterStudent(studentName),      University(uniName),
MasterProgram(title),            MasterCourse(code),
EnrolledIn(studentName,title),   RegisteredTo(studentName, uniName).
```

These relations are defined in terms of the local *relations by the following GAV mappings:*

```
MasterStudent(N) ⊇ S2.Erasmus(N,C,U),S4.Mundus(P,C)
MasterStudent(N) ⊇ S3.CampusFr(N,P,U),S4.Mundus(P,C)
University(U) ⊇ S1.Catalogue(U,P)
University(U) ⊇ S2.Erasmus(N,C,U)
University(U) ⊇ S3.CampusFr(N,P,U)
MasterProgram(T) ⊇ S4.Mundus(T,C)
MasterCourse(C) ⊇ S4.Mundus(T,C)
EnrolledIn(N,T) ⊇ S2.Erasmus(N,C,U),S4.Mundus(T,C)
EnrolledIn(N,T) ⊇ S3.CampusFr(N,T,U),S4.Mundus(T,C)
RegisteredTo(N,U) ⊇ S3.CampusFr(N,T,U)
```

@ Serge Abiteboul, Ioana Manolescu, Philippe Rigaux, Marie-Christine Rousset, Pierre Senellart, 2011; to be published by Cambridge University Press 2011.

Note that in a warehousing approach, one would simply evaluate all the queries that define the global view, and populate the warehouse using standard relational query evaluation. In a mediator approach, we try to only derive data that is relevant to a specific query posed on the global view by a user. We show how to rewrite a global query into queries over the local relations and combine their results. This is achieved by a technical trick that consists in *unfolding* the atoms of the global query.

Observe the first two mappings. They specify (using joins) how to obtain tuples of the unary relation `MasterStudent`. Now consider the following global query asking for universities with registered master students:

```
q(x) :- RegisteredTo(s,x), MasterStudent(s)
```

The rewriting of this query into source queries is obtained by *unfolding*, i.e., by replacing each atom which can be matched with the head of some view, by the body of the corresponding view. (For readers familiar with logic programming, this is some very simple form of resolution.)

In the example, there is a single mapping whose head can be matched with $RegisteredTo(s,x)$, and two mappings that match $MasterStudent(s)$. Thus, we obtain the following two unfoldings:

```
q1(x) :- S3.CampusFr(s,v1,x), S2.Erasmus(s,v2,v3), S4.Mundus(v4,v2)
q2(x) :- S3.CampusFr(s,v5,x), S3.CampusFr(s,v6,v7), S4.Mundus(v6,v8)
```

Observe that `q2` can be simplified. Replacing the conjunction of its first two atoms by the single atom `S3.CampusFr(s,v6,x)` leads to an equivalent query. We thus obtain the following two GAV rewritings of the initial query:

```
r1(x) :- S3.CampusFr(s,v1,x), S2.Erasmus(s,v2,v3), S4.Mundus(v4,v2)
r2(x) :- S3.CampusFr(s,v6,x), S4.Mundus(v6,v8)
```

The result is obtained by computing `r1` $\cup$ `r2`. Now, observe that each r_ℓ is a conjunctive query. It can be optimized using standard query optimization to obtain an optimized physical query plan. Of course, the choice of the particular physical query plan that is selected depends on the statistics that are available and the capabilities of the sources. For instance, a plan may consist in querying `S3` and then for each value `a` of `v6` (i.e., a particular university program), asking the query `q(X) :- S4.Mundus(a,X)` to `S4`.

We now formalize the simple and intuitive notion of query unfolding.

Definition 3.3 (Query unfolding) *Let $q(\vec{x})$:- $G_1(\vec{z}_1),\ldots,G_n(\vec{z}_n)$ be a query and for each i, $G_i(\vec{x}_i) \supseteq q_i(\vec{x}_i,\vec{y}_i)$ be a GAV mapping. An* unfolding *of q is the query u obtained from q by replacing, for each i, each conjunct $G_i(\vec{z}_i)$ by $q_i(\psi_i(\vec{x}_i,\vec{y}_i))$ where ψ_i is a function that maps $\vec{x}_i$ to $\vec{z}_i$, and the existential variables $\vec{y}_i$ to new fresh variables.*

The renaming of the existential variables into fresh ones is necessary to avoid the introduction of unnecessary constraints in the unfolding. Indeed, consider an existential variable y occurring in two distinct atoms, say G_i and G_j. Then, the two atoms should be understood

@ Serge Abiteboul, Ioana Manolescu, Philippe Rigaux, Marie-Christine Rousset, Pierre Senellart, 2011; to be published by Cambridge University Press 2011.

as $\exists ...y...G_i(\vec{z}_i)$ and $\exists ...y...G_j(\vec{z}_j)$. The scopes of y in both are disjoint and nothing requires that the two occurrences of y take the same value. Hence the renaming using fresh variables.

Example 3.4 *Suppose we have the two mappings:*

$$F(x,y) \supseteq S(x,z), S(y,z) \qquad G(x) \supseteq S(x,y)$$

and the query $q(x)$:- $F(x,y), G(y)$. Then we get the following unfolding:

$$q(x) \text{ :- } S(x,v_1), S(y,v_1), S(y,v_2)$$

The variable v_1 corresponds to the renaming of the existential variable z in the view defining F, whereas v_2 comes from the renaming of the existential variable y in the view defining G.

We next establish that each unfolding of a query computes a part of the desired results, and that their union computes the whole set of answers. To do so, we use two propositions. The first one ignores unfolding and focuses on the "materialization" of the global relations.

Proposition 3.5 *Let $S_1,...,S_n$ be a set of source relations; $G_1,...,G_m$ a global schema defined by a set $\mathcal{G}$ of GAV mappings over $S_1,...,S_n$; and I be an instance over $S_1,...,S_n$. Let J be the instance over $G_1,...,G_m$ defined by, for each j,*

$$J(G_j) = \cup\{V(I) \mid G_j \supseteq V(S_1,...,S_n) \in \mathcal{G}\}$$

Then for each query q over $G_1,...,G_m$, the answer of q is $q(J)$.

Proof (sketch). Let u be an answer. Then, by definition, $q(u)$ is true in each instance J' over $G_1,...,G_m$ such that I and J' together satisfy the mappings. In particular, u belongs to $q(J)$. Conversely, let u be in $q(J)$. Let J' be an instance such that I and J' together satisfy the mappings. Since J' satisfies the mappings, $J \subseteq J'$. Since conjunctive queries are monotone, $q(J) \subseteq q(J')$. Thus $u \in J'$. Since u belongs to all such J', u is an answer. □

The second proposition deals with unfoldings.

Proposition 3.6 *Let* **S** *be a set of source relations and* **G** *a set of global relations defined by a set $\mathcal{G}$ of GAV mappings over* **S**. *Consider the query $q(\vec{z})$:- $G_{i_1}(\vec{z_{i_1}}),\ldots,G_{i_n}(\vec{z_{i_n}})$ over* **G** *and the set $\{r_\ell\}$ of unfoldings of q given $\mathcal{G}$. Then for each I over $S_1,...,S_n$, the answer of q is given by $\cup r_\ell(I)$.*

Proof (sketch). Let J be as in Proposition 3.5. By the same proposition, it suffices to show that $q(J) = \cup r_\ell(I)$.

Soundness. Let $u \in \cup r_\ell(I)$. Then u is $r_\ell(I)$ for some unfolding r_ℓ. Suppose r_ℓ results from the unfolding defined by selecting for each j, the mapping $G_{i_j}(\vec{x_{i_j}}) \supseteq q_{i_j}(\vec{x_{i_j}}, \vec{y_{i_j}})$. It follows that $u \in q(\{\vec{u_1}\},...,\{\vec{u_n}\})$ where for each j, $\vec{u_j}$ is derived by $G_{i_j}(\vec{x_{i_j}}) \supseteq q_{i_j}(\vec{x_{i_j}}, \vec{y_{i_j}})$. Thus, each $\vec{u_j}$ is in $J(G_{i_j})$ and $u \in q(J(G_{i_1}),...,J(G_{i_n})) = q(J)$. Therefore, $\cup r_\ell(I) \subseteq q(J)$.

Completeness. Conversely, consider u in $q(J)$. Then, there exists $\vec{u_1}$ in $J(G_{i_1})$, ..., $\vec{u_j}$ in $J(G_{i_j})$, ...$\vec{u_n}$ in $J(G_{i_n})$ such that $u \in q(\{\vec{u_1}\},...,\{\vec{u_n}\})$. By construction of J, for each j there is some mapping $G_{i_j}(\vec{x_{i_j}}) \supseteq q_{i_j}(\vec{x_{i_j}}, \vec{y_{i-1}})$ such that $\vec{u_j}$ is in $q_{i_j}(\vec{x_{i_j}}, \vec{y_{i-1}})$. Consider the unfolding r_ℓ defined by selecting for each j, this particular mapping. One can verify that u is $r_\ell(I)$. Hence, $u \in \cup r_\ell(I)$ and $q(J) \subseteq \cup r_\ell(I)$.

@ Serge Abiteboul, Ioana Manolescu, Philippe Rigaux, Marie-Christine Rousset, Pierre Senellart, 2011; to be published by Cambridge University Press 2011.

□

We can compute the answer using the unfoldings (also called the GAV rewritings). These unfoldings can be simplified by removing redundant conjuncts that may have been introduced by the technique. This simplification relies on checking conjunctive query containment. Given a conjunctive query with body $A_1(\vec{u_1}),...,A_m(\vec{u_m})$, we verify whether each query obtained by removing some $A_i(\vec{u_i})$ is equivalent to the initial one. If yes, the atom is redundant and can be removed. We keep doing this until the query is "minimal". This simplification test is costly but the resulting query may be much less expensive to evaluate that the initial one.

We must evaluate all the unfoldings to obtain the complete answer. If we are aware of some constraints on the local schemas or on the global one, this can be further simplified. For instance, the constraints may imply that the result of a particular unfolding is empty, in which case this particular unfolding needs not be evaluated. Also, the constraints may imply that the result of some unfolding, say r_ℓ, is always included in another one. Then r_ℓ needs not be evaluated. For instance, in the previous example, if it is known that students obtained from the source *S2* are European students, while those obtained from the source *S3* are non European students, we can be sure that the GAV rewriting r_ℓ obtained by unfolding will not provide any answer. This requires expressing and exploiting disjointness constraints over the local relations. Inclusion constraints on local relations would, on the other hand, permit to detect in advance that a given query plan provides answers that are redundant with those obtained by another query plan.

A main limitation of GAV is that adding or removing data sources to the integration system may require deeply revising all the views defining the global schema. In a Web context where sources may come and go, e.g., because of (non) availability of servers, this is really too constraining. The LAV approach does not suffer from this disadvantage.

4 Local-as-view mediation

The LAV approach takes a dual approach. The local relations are defined as views over global relations. The goal is to define the global schema in such a way that individual definitions do not change when data sources join or leave the integration system except for the definitions of the sources that are involved in the change.

Definition 4.1 (LAV mapping) *A LAV mapping is a* mapping *of the form: $S \subseteq q$, for some conjunctive query $q(x_1,...,x_n)$:- $A_1(\vec{u_1}),...,A_k(\vec{u_k})$ over the global relations. Its semantics is:*

$$\forall x_1,...,x_n[S(x_1,...,x_n) \Rightarrow (\exists y_1,...,y_m A_1(\vec{u_1}),...,A_k(\vec{u_k}))]$$

where $y_1,...,y_m$ are the existential variables.

Again, $S(x_1,...,x_n)$ is called the *head* of the view, whereas $A_1(\vec{u_1}),...,A_k(\vec{u_k})$ is called the *body* of the view.

Example 4.2 *We define the global schema as consisting of the following relations:*

@ Serge Abiteboul, Ioana Manolescu, Philippe Rigaux, Marie-Christine Rousset, Pierre Senellart, 2011; to be published by Cambridge University Press 2011.

```
Student(studentName),                EuropeanStudent(studentName),
University(uniName),                 NonEuropeanStudent(studentName),
FrenchUniversity(uniName),           EuropeanUniversity(uniName),
NonEuropeanUniversity(uniName),      Program(title),
MasterProgram(title),                EnrolledInProgram(studentName,title),
Course(code),                        EnrolledInCourse(studentName,code),
PartOf(code, title),                 RegisteredTo(studentName, uniName),
OfferedBy(title, uniName).
```

The four data sources considered in the previous example can be described by the following LAV mappings:

```
m1: S1.Catalogue(U,P) ⊆    FrenchUniversity(U),Program(P),OfferedBy(P,U),
                           OfferedBy(P',U),MasterProgram(P')
m2: S2.Erasmus(S,C,U) ⊆    Student(S),EnrolledInCourse(S,C),PartOf(C,P),
                           OfferedBy(P,U),EuropeanUniversity(U),
                           EuropeanUniversity(U') RegisteredTo(S,U'),
                           U ≠ U'
m3: S3.CampusFr(S,P,U) ⊆   NonEuropeanStudent(S),Program(P),
                           EnrolledInProgram(S,P),OfferedBy(P,U),
                           FrenchUniversity(U),RegisteredTo(S,U)
m4: S4.Mundus(P,C) ⊆       MasterProgram(P),OfferedBy(P,U),
                           OfferedBy(P,U'),EuropeanUniversity(U),
                           NonEuropeanUniversity(U'),PartOf(C,P)
```

LAV mappings enable quite fine-grained descriptions of the contents of data sources. For example, we are able to specify precisely the students that can be found in the Erasmus source: they are European students enrolled in courses of a given (European) university that is different from their home (European) University in which they remain registered.

LAV mappings express loose coupling between local and global relations, which is important for flexibility and robustness when the participating data sources change frequently. If we are interested in Master students, we do not need to know in advance (unlike the GAV approach) how to join two sources. We just define them as a global query:

```
MasterStudent(E) :-  Student(E),EnrolledInProgram(E,M),
                     MasterProgram(M).
```

The local sources that must be queried and combined to get the Master students will be discovered by the rewriting process. Recall that, in the GAV approach, they were predefined by the two mappings given in Example 3.2.

The price to pay for the flexibility of LAV compared to GAV is that the rewritings are more complicated to find. We describe three algorithms that achieve this rewriting. The Bucket algorithm and the Minicon algorithm follow the same approach. They first determine the local relations that are *relevant* to the query, then consider their combinations as *candidate* rewritings and verify whether they are indeed correct. Minicon is actually an optimization of Bucket that avoids the last verification step by a trickier first step. The third algorithm, namely the Inverse-rules algorithm, follows a completely different approach: it consists in

@ Serge Abiteboul, Ioana Manolescu, Philippe Rigaux, Marie-Christine Rousset, Pierre Senellart, 2011; to be published by Cambridge University Press 2011.

transforming the logical rules supporting the LAV mappings (which are unsafe rules) into a set of safe rules with a single global relation. The global query is unfolded using these rules.

4.1 The Bucket algorithm

The principle of the Bucket algorithm is quite simple. It proceeds in three steps:

1. the first step constructs for each atom g of the global query body its *bucket*, which groups the view atoms from which g can be inferred;
2. the second step consists in building a set of *candidate* rewritings that are obtained by combining the view atoms of each bucket;
3. in a last step, we check whether each candidate rewriting is valid.

Bucket creation

Let g be a query atom. The atoms in $bucket(g)$ are the heads of mappings having in their body an atom from which g can be inferred. Intuitively, data comes from source relations, and a (global) query atom is satisfied by (local) data only if it can be matched to a (global) atom in the body of a mapping whose head can be matched to source facts. A match between g and some atom in the body of a mapping is thus an indication that the corresponding data source provides a relevant information for this particular query.

There is an extra constraint that has to be considered to guarantee that g can indeed be logically inferred, as illustrated next. In fact, the bucket of a query atom g includes a view atom v only if an atom in the body of v can be matched with g by a variable mapping such that the variables mapped to the *distinguished* variables of g are also *distinguished* variables in the view defining the mapping.

Let us illustrate this on an example. Consider the LAV mappings of Example 4.2, and the global query:

```
q(x) :- RegisteredTo(s,x), EnrolledInProgram(s,p), MasterProgram(p)
```

Let us consider the query atom `g= RegisteredTo(s,x)`, in which the variable `x` is *distinguished*.

We can find two mappings (`m2` and `m3`) in which a body atom can be matched to `RegisteredTo(s,x)`.

First, consider the mapping `m3`:

```
m3: S3.CampusFr(S,P,U) ⊆  NonEuropeanStudent(S),Program(P),
                          EnrolledInProgram(S,P),OfferedBy(P,U),
                          FrenchUniversity(U),RegisteredTo(S,U)
```

The atom `RegisteredTo(s,x)` matches the atom `RegisteredTo(S,U)` with the variable mapping $\{S/s, U/x\}$, where U is *distinguished* in the view defining the mapping (it occurs in the head of this LAV mapping).

Therefore, applying the variable mapping $\{S/s, U/x\}$ to the head `S3.CampusFr(S,P,U)` of the mapping `m3` enforces the matching of `RegisteredTo(S,U)` with the query atom `RegisteredTo(s,x)`, and then:

@ Serge Abiteboul, Ioana Manolescu, Philippe Rigaux, Marie-Christine Rousset, Pierre Senellart, 2011; to be published by Cambridge University Press 2011.

$$\texttt{S3.CampusFr(s,v1,x)} \wedge \texttt{FOL(m3)} \models \exists s\ \texttt{RegisteredTo(s,x)}$$

Thus `S3.CampusFr(s,v1,x)` is added in *Bucket*(g). Note that `v1` is simply a fresh variable mapped to the variable `P` appearing in `S3.CampusFr(S,P,U)` but not in the variable mapping $\{S/s, U/x\}$.

On the other hand, consider the mapping `m2`:

$$\begin{aligned}\texttt{m2: S2.Erasmus(S,C,U)} \subseteq\ & \texttt{Student(S),EnrolledInCourse(S,C),PartOf(C,P),}\\ & \texttt{OfferedBy(P,U),EuropeanUniversity(U),}\\ & \texttt{EuropeanUniversity(U') RegisteredTo(S,U'),U} \neq \texttt{U'}\end{aligned}$$

The match this time is between $g = \texttt{RegisteredTo(s,x)}$ and `RegisteredTo(S,U')` by the variable mapping $\{S/s, U'/x\}$. The difference with the previous situation is that the variable U' is *existentially* quantified in the view defining the mapping. Applying the variable mapping $\{S/s, U'/x\}$ to the head `S2.Erasmus(S,C,U)` of the mapping `m2` *do not* enforce the matching of `RegisteredTo(S,U')` in its body with the query atom `RegisteredTo(s,x)`.

More formally:

$$\texttt{S2.Erasmus(s,v2,v3)} \wedge \texttt{FOL(m2)} \not\models \exists s\ \texttt{RegisteredTo(s,x)}.$$

To see why, consider the LAV mapping `m2` and its logical meaning `FOL(m2)`:

$$\begin{aligned}\texttt{FOL(m2):}\ & \forall S \forall C \forall U\ [\ \texttt{S2.Erasmus(S,C,U)} \Rightarrow \exists\ \texttt{P}\ \exists\ \texttt{U'}\ (\\ & \texttt{EuropeanStudent(S)} \wedge \texttt{EnrolledInCourse(S,C)} \wedge\\ & \texttt{PartOf(C,P)} \wedge \texttt{OfferedBy(P,U)}\\ & \wedge \texttt{EuropeanUniversity(U)} \wedge \texttt{RegisteredTo(S,U')} \wedge \texttt{U} \neq \texttt{U'})\ ]\end{aligned}$$

From the fact that `S2.Erasmus(s,v2,v3)`, it follows that:

$$\exists\ \texttt{s}\ \exists\ \texttt{U'}\ \texttt{RegisteredTo(s,U')}.$$

However, this is a strictly weaker statement than $\exists$ `s RegisteredTo(s,x)` where `x` is fixed. We prove this next. Consider an instance I over the domain $\Delta = \{s, v2, v3, v4, v5, x\}$ defined by:

$$\begin{aligned} &I(\texttt{S2.Erasmus}) = \{\langle \texttt{s, v2, v3}\rangle\} && I(\texttt{EuropeanStudent}) = \{\texttt{s}\}\\ &I(\texttt{EnrolledInCourse}) = \{\langle \texttt{s, v2}\rangle\} && I(\texttt{PartOf}) = \{\langle \texttt{v2, v4}\rangle\}\\ &I(\texttt{OfferedBy}) = \{\langle \texttt{v4, v3}\rangle\} && I(\texttt{EuropeanUniversity}) = \{\texttt{v3, v5}\}\\ &I(\texttt{RegisteredTo}) = \{\langle \texttt{s, v5}\rangle\} \end{aligned}$$

By the valuation that instantiates respectively the variables `S` to the constant `s`, `C` to the constant `v2`, `U` to the constant `v3`, `P` to the constant `v4` and `U'` to the constant `v5`, we see that I satisfies the fact `S2.Erasmus(s,v2,v3)` and the formula `FOL(m2)`, but that $\exists$ `s RegisteredTo(s,x)` is not satisfied in I.

As a consequence, `S2.Erasmus(s,v2,v3)` does not belong to the bucket and:

$$\texttt{Bucket}(\texttt{RegisteredTo}(\texttt{s,x})) = \{\texttt{S3.CampusFr}(\texttt{s,v1,x})\}.$$

Algorithm 2 constructs the buckets. Proposition 4.3 is a logical characterization of the view atoms put in the buckets of the atoms of the global query.

@ Serge Abiteboul, Ioana Manolescu, Philippe Rigaux, Marie-Christine Rousset, Pierre Senellart, 2011; to be published by Cambridge University Press 2011.

Algorithm 2: The *Bucket* algorithm
Bucket(g, q, M)
Input: An atom $g = G(u_1,...,u_m)$ of the query q and a set of LAV mappings
Output: The set of view atoms from which g can be inferred
(1) $Bucket(g) : \emptyset$
(2) **for each** LAV mapping $S(\vec{x}) \subseteq q(\vec{x},\vec{y})$
(3) **if** there exists in $q(\vec{x},\vec{y})$ an atom $G(z_1,...,z_m)$ such that
(4) z_i is distinguished for each i such that u_i is distinguished in q;
(5) **let** ψ the variable mapping $\{z_1/u_1,....,z_m/u_m\}$
(6) extended by mapping the head variables in $\vec{x}$ not
(7) appearing in $\{z_1,...,z_m\}$ to new fresh variables;
(8) **add** $S(\psi(\vec{x}))$ to $Bucket(g)$;
(9) **return** $Bucket(g)$;

Proposition 4.3 *Let $G(u_1,...,u_m)$ be an atom of the global query. Let $\vec{u}$ be the (possibly empty) subset of existential variables in $\{u_1,...,u_m\}$. Let m: $S(\vec{x}) \subseteq q(\vec{x},\vec{y})$ be a LAV mapping. Then*

$$S(\vec{v}),\ FOL(m) \models \exists\vec{u}G(u_1,...,u_m)$$

iff there exists a view atom in $Bucket(g)$ that is equal to $S(\vec{v})$ (up to a renaming of the fresh variables).

The proof is tedious and left as exercise.

In the worst-case, the Bucket algorithm applied to each atom of a query has a time complexity in $O(N \times M \times V)$ and produces N buckets containing each at most $M \times V$ view atoms, where N is the size of the query, M is the maximal size of the LAV mappings and V is the number of LAV mappings.

Returning to the example, we obtain by the Bucket algorithm, the following buckets for the three atoms of the query q.

`RegisteredTo(s,x)`	`EnrolledInProgram(s,p)`	`MasterProgram(p)`
`S3.CampusFr(s,v1,x)`	`S3.CampusFr(s,p,v2)`	`S1.Catalogue(v3,v4)`
		`S4.Mundus(p,v5)`

Construction of candidate rewritings

The *candidate* rewritings of the initial global query are then obtained by combining the view atoms of each bucket. In the worst-case, the number of candidate rewritings is in $O((M \times V)^N)$. For instance, in our example, we obtain two *candidate* rewritings for the query q:

```
r1(x) :- S3.CampusFr(s,v1,x), S3.CampusFr(s,p,v2), S1.Catalogue(v3,v4)
r2(x) :- S3.CampusFr(s,v1,x), S3.CampusFr(s,p,v2), S4.Mundus(p,v5)
```

A *candidate* rewriting may not be a valid rewriting of the query. By Proposition 4.3, we only know that each candidate rewriting entails each atom of the query *in isolation*, i.e., without taking into account the possible bindings of the existential variables within the query.

@ Serge Abiteboul, Ioana Manolescu, Philippe Rigaux, Marie-Christine Rousset, Pierre Senellart, 2011; to be published by Cambridge University Press 2011.

It turns out that, in our example, the first candidate rewriting `r1` is not a valid rewriting of the query q: the body of q is not logically entailed by the conjunction of the view atoms in the body of `r1`.

To see why, we first apply to each view atom in the body of `r1` the corresponding LAV mapping to obtain the logical global expression (i.e., built on global relations). This step is called *expanding* `r1`, and its result, the *expansion* of `r1`. In our case, the expansion of `r1` is the following query expression:

```
Exp_r1(x) :- NonEuropeanStudent(s), Program(v1), EnrolledInProgram(s,v1),
             OfferedBy(v1,x), FrenchUniversity(x), RegisteredTo(s,x),
             Program(p), EnrolledInProgram(s,p), OfferedBy(p,v2),
             FrenchUniversity(v2), RegisteredTo(s,v2),
             FrenchUniversity(v3), Program(v4),OfferedBy(v4,v3),
             OfferedBy(v5,v3), MasterProgram(v5)
```

Note that new existential variables may be introduced by the expansion of some view atoms. For instance, the LAV mapping defining `S1.Catalogue(v3,v4)` contains the existential variable denoted `P'` in the LAV mapping definition. Such variables are renamed with new fresh variables to avoid unnecessary constraints between the variables. In our example, this corresponds to variable `v5` in the body of `Exp_r1(x)`.

To check whether a rewriting is correct, it suffices to check with the Conjunctive Query Containment algorithm whether the query `Exp_r1(x)` is *contained* in the query `q(x)`. For each variable v, let the corresponding constant, i.e., $\psi(v)$, be `"v"`. The canonical database obtained from `r1` is given in Figure 2.

NonEuropean-Student	Program	EnrolledIn-Program	OfferedBy	French-University	RegisteredTo	Master-Program
"s"	"v1" "p" "v4"	("s","v1") ("s","p")	("v1","x") ("p","v2") ("v4","v3") ("v5","v3")	"x" "v2" "v3"	("s","x") ("s","v2")	"v5"

Figure 2: The canonical database resulting from freezing `r1`

The evaluation of `q(x)` over this canonical database yields an empty result because there is no way of assigning the existential variables `s` and `p` to constants of the canonical database which satisfies the binding of the existential variable `p` between the two last atoms of the body of the query.

Expanding the rewriting `r2` and checking that it is contained into the query `q` is left in exercise. This shows that among the two candidate rewritings, only `r2` is a valid rewriting:

```
r2(x) :- S3.CampusFr(s,v1,x), S3.CampusFr(s,p,v2),S4.Mundus(p,v5)
```

Remark 4.4 In spite of the apparent redundancy of the two first atoms, this rewriting cannot be simplified to

```
  r2.1(x) :- S3.CampusFr(s,p,x), S4.Mundus(p,v5)
```

@ Serge Abiteboul, Ioana Manolescu, Philippe Rigaux, Marie-Christine Rousset, Pierre Senellart, 2011; to be published by Cambridge University Press 2011.

It is true that `r2.1(x)` is contained into `r2(x)`. However, the two queries are not equivalent. For some data sets, it may be the case that there is a student `s` and there is a university `x` such that (based on `S3.CampusFr`), `s` is registered in `x` and also enrolled in a Mundus master program offered by another university. The containment would hold under a constraint that would forbid a student to be registered in more than one universities.

One can prove that each rewriting finally obtained does indeed provide answers and that their union constitutes the complete answer.

4.2 The Minicon algorithm

The idea underlying Minicon is to avoid putting in a bucket an atom that will only generate invalid rewritings. As we saw in the discussion of Bucket, the reason for an atom to be useless is that its binding of a variable does not match with the binding of other occurrences of that variable. This explains why a candidate rewriting (like `r1`) is not valid.

We now illustrate the Minicon algorithm by example. Consider the query `q`:

```
q(x) :- U(y,z), R(x,z), T(z,y), R(y',x)
```

and the two LAV mappings:

$$\texttt{V1(u,v)} \subseteq \texttt{T(w,u),U(v,w),R(v,u)}$$
$$\texttt{V2(u,v,v')} \subseteq \texttt{T(w,u),U(v,w),R(v',w)}$$

Minicon proceeds in two steps that correspond to the first two steps of Bucket.

First step of Minicon: creation of MCDs

Minicon scans each atom in the query, but instead of creating buckets for them, it builds MCDs (short name for *Minicon Descriptions*). The first iteration of Minicon determines the relevance of the different LAV mappings to rewrite the first query atom `U(y,z)`:

- The Bucket algorithm would put `V1(v1,y)` in the bucket of `U(y,z)` (where `v1` is a fresh variable), because the variable mapping $\{v/y, w/z\}$ allows the match between the atom `U(v,w)` in the expansion of `V1(u,v)` and the query atom `U(y,z)`.

  Minicon does not consider the query atom `U(y,z)` in isolation. Instead, since the variable `w` is *existential* in the view defining the mapping, and mapped to the variable `z` that has several occurrences in the query, it checks whether the variable mapping $\{v/y, w/z\}$ also *covers all* the query atoms involving variable `z`, i.e., can be extended to also match `R(x,z)` and `T(z,y)`. Because variable `w` is *existential* in the expansion of `V1(u,v)` (i.e., `w` does not appear in the head of the mapping), it is the only way to enforce the several occurrences of `z` in the query to be mapped to by the same variable `w`. Here, matching the query atom `R(x,z)` with an atom of the form `R(_,w)` in the expansion of `V1(v1,y)` is not possible: there does not exist such an atom in the expansion of `V1(v1,y)`. Therefore, no MCD is created from `V1` for covering the query atoms including an occurrence of the variable `z`.

@ Serge Abiteboul, Ioana Manolescu, Philippe Rigaux, Marie-Christine Rousset, Pierre Senellart, 2011; to be published by Cambridge University Press 2011.

- When considering `V2`, though Minicon starts from the same variable mapping $\{v/y, w/z\}$ to match `U(v,w)` in the expansion of `V2(u,v,v')` and the query atom `U(y,z)`, the situation is different for checking whether it can be extended to cover the other query atoms `R(x,z)` and `T(z,y)` containing occurrences of the variable `z`. Extending the variable mapping $\{v/y, w/z\}$ to match `R(x,z)` with the atom `R(v',w)` is possible by adding the variable mapping v'/x. Now, extending the variable mapping $\{v/y, w/z, v'/x\}$ to match `T(z,y)` with the atom `T(w,u)` is also possible by adding the variable mapping u/y. The resulting variable mapping is: $\{v/y, w/z, v'/x, u/y\}$. And, `V2(y,y,x)` is retained as a rewriting of the corresponding part of the query: a MCD is created for it, with in addition the positions of the atoms in the query it covers:

MCD1 = (`V2(y,y,x)`) , {1,2,3})

The last iteration of building MCDs corresponds to the last query atom: `R(y',x)`. The LAV mapping `V1` has in its expansion the atom `R(v,u)` that can be matched to it by the variable mapping $\{v/y', u/x)\}$. Since the distinguished variable `x` in the query is assigned to the distinguished variable (same condition as for adding to a bucket), and since the existential variable `y'` of the query atom has a single occurrence in the query, the following MCD is created:

MCD2 = ((`V1(x,y')`, {4})

In contrast, there is no MCD created for `R(y',x)` with the second LAV mapping: in the variable mapping $\{v'/y', w/x)\}$ that allows to match the query atom `R(y',x)` with the atom `R(v',w)` in the expansion of `V2`, the distinguished variable `x` in the query is assigned to the variable `w` which is not distinguished in the expansion of of `V2`. As for adding a view atom in a bucket, a MCD is created for a query atom g *only* if the variables mapped to the distinguished variables of g are also distinguished variables in the view defining the mapping.

.

Second step of Minicon: combination of the MCDs

The second step of Minicon replaces the combination of the buckets by the combination of the MCDs. More precisely, the rewritings are obtained by combining MCDs that cover mutually disjoint subsets of query atoms, while together covering all the atoms of the query.

Because of the way in which the MCDs were constructed, the rewritings obtained that way are guaranteed to be valid. No containment checking is needed, unlike in the Bucket Algorithm. In our example, we would therefore obtain as single rewriting of `q(x)`:

```
r(x) :- V2(y,y,x), V1(x,y')
```

4.3 The Inverse-rules algorithm

This algorithm takes a radically different approach. It transforms the LAV mappings into GAV mappings (called inverse rules) so that the complex operation of query rewriting using LAV mappings can then be replaced by the much simpler operation of query unfolding. A

@ Serge Abiteboul, Ioana Manolescu, Philippe Rigaux, Marie-Christine Rousset, Pierre Senellart, 2011; to be published by Cambridge University Press 2011.

LAV mapping is replaced by several GAV mappings, one for each atom in the body of the rule. The subtlety is to keep bindings between the different occurrences of the same existential variable in the body. This is realized using a simple trick from first-order logic, namely by introducing *Skolem functions*.

Let us explain the Inverse-rules algorithm on the example we used for Minicon. A first important point that distinguishes it from the Bucket and Minicon algorithms is that the Inverse-rules algorithm is independent of the query. It only considers as input the set of LAV mappings:

$$\texttt{V1(u,v)} \subseteq \texttt{T(w,u),U(v,w),R(v,u)}$$
$$\texttt{V2(u,v,v')} \subseteq \texttt{T(w,u),U(v,w),R(v',w).}$$

Consider the first LAV mapping and recall that its logical meaning mapping is the formula:

$$\forall \texttt{u}\, \forall \texttt{v} \big[\texttt{V1(u,v)} \Rightarrow \exists w \big(\texttt{T(w,u)} \wedge \texttt{U(v,w)} \big) \wedge \texttt{R(v,u)} \big) \big]$$

Suppose we know that `(a,b)` belongs to the source relation `V1`. From the fact `V1(a,b)`, we can infer the fact `R(b,a)`, i.e., that the tuple `(b,a)` is in the extension of the global relation `R`, and thus that, for instance, `b` is an answer for the global query `q(x) :- R(x,y)`.

But we can infer much more. We can also infer that there exists some constant `d1` such that `T(d1,a)` and `U(b,d1)` are both true. We do not know the exact value of that constant `d1`, but we know it exists and that, in some way, it depends on the constants `a,b`. Since this dependency comes from the first rule, we denote this unknown `d1` value: `f1(a,b)`.

Creating the inverse rules This motivates the construction of three following GAV mappings for which we give also the FOL translation.

$$\texttt{IN11 : V1(u,v)} \subseteq \texttt{T(f1(u,v),u)} \quad \texttt{FOL(IN11)} : \forall u \forall v \big[\texttt{V1(u,v)} \Rightarrow \texttt{T(f1(u,v),u)} \big]$$
$$\texttt{IN12 : V1(u,v)} \subseteq \texttt{U(v,f1(u,v))} \quad \texttt{FOL(IN12)} : \forall u \forall v \big[\texttt{V1(u,v)} \Rightarrow \texttt{U(v,f1(u,v))} \big]$$
$$\texttt{IN13 : V1(u,v)} \subseteq \texttt{R(v,u)} \quad \texttt{FOL(IN13)} : \forall u \forall v \big[\texttt{V1(u,v)} \Rightarrow \texttt{R(v,u)} \big]$$

They are called the *inverse rules* of the corresponding LAV mapping.

In the previous rules, the symbol `f1` is a Skolem function of arity 2, and `f1(u,v)` is a Skolem term denoting some constant that depends on the values instantiating the variables `u,v`. Given two distinct Skolem terms, e.g. `f1(1,2)` and `f1(2,v3)`, we cannot tell whether they refer to the same constant or not.

The Inverse-rules algorithm just scans the LAV mappings and creates n GAV mappings for each LAV mapping having n atoms. The result of this algorithm applied to the second LAV mappings in the example is:

$$\texttt{IN21 : V2(u,v,v')} \subseteq \texttt{T(f2(u,v,v'),u)}$$
$$\texttt{IN22 : V2(u,v,v')} \subseteq \texttt{U(v,f2(u,v,v'))}$$
$$\texttt{IN23 : V2(u,v,v')} \subseteq \texttt{R(v',f2(u,v,v'))}$$

Obtaining the rewritings by unfolding: The rewritings of any global query is now obtained by unfolding the query atoms using the (Inverse-rules) GAV mappings corresponding to the initial set of LAV mappings. The unfolding operation here is a bit trickier than the unfolding defined in Definition 3.3, because of the Skolem terms. In Definition 3.3, the unfolding

@ Serge Abiteboul, Ioana Manolescu, Philippe Rigaux, Marie-Christine Rousset, Pierre Senellart, 2011; to be published by Cambridge University Press 2011.

was based on matching each query atom `G(x1, .., xm)` with an atom (in the right-hand side of a GAV mapping) of the form `G(z1, .., zm)` by equating each pair (`zi`, `xi`) of variables. Proposition 3.6 showed that unfolding each atom of the query *in isolation* builds valid rewritings of the query, i.e., conjunctions of view atoms which logically implies the conjunction of the query atoms. It is not the case anymore when atoms in the right-hand side of GAV mappings contain Skolem terms.

The *unification* of two atoms with functions is more complex than just equating variables, and it may fail. It may require the substitution of some variables with functional terms (in our case, Skolem terms). This may make impossible to unify the other atoms of the query with atoms in the right-hand side of GAV mappings.

Let us illustrate on our example the subtleties of unfolding queries in presence of functional terms. Consider again the same query `q`:

```
q(x) :- U(y,z), R(x,z), T(z,y), R(y',x).
```

The query atom `U(y,z)` can be unified with the atom `U(v,f1(u,v))` in the right-hand side of the GAV mappings `IN12` using a so-called *most general unifier (mgu)*. In this case, the mgu is the substitution:

$$\sigma = \{\texttt{y/v1, v/v1, z/f1(v2,v1), u/v2}\}$$

where `v1` and `v2` are new fresh variables introduced in order to avoid name conflict between variables that would generate unnecessary constraints. The substitution σ is a unifier of the two expressions `U(y,z)` and `U(v,f1(u,v))` because the replacement in the two expressions of the occurrences of the variables `y`, `v`, `z` and `u` by the corresponding term (variable or Skolem term) in σ results in two identical expressions:

$$\sigma(\texttt{U(y,z)}) = \sigma(\texttt{U(v,f1(u,v))})$$

This substitution that makes the unfolding of the first query atom possible, now constrains the other occurrences in the query of the variables y and z for the unfolding of the other query atoms. After the application of σ to the whole body of the query and the unfolding of the first query atom made possible by σ, we obtain the following (partial) query rewriting:

```
pr1(x) :- V1(v2,v1), R(x,f1(v2,v1)), T(f1(v2,v1),v1), R(y',x).
```

The unfolding of the second atom `R(x,f1(v2,v1))` yields `V1(f1(v2,v1),x)`, and we obtain the (partial) rewriting:

```
pr2(x) :- V1(v2,v1), V1(f1(v2,v1),x), T(f1(v2,v1),v1), R(y',x).
```

It is useless to continue unfolding the remaining query atoms of `pr2(x)`. As soon as a given unfolding has produced a view atom with Skolem terms, we can be sure that the evaluation of the query plan under construction will not produce any answer: there is no way to match `V1(f1(v2,v1),x)` with any fact in the data source which are of the form `V1(a,b)` where `a,b` are constants. Since we don't know `f1(v2,v1)`, there is absolutely no reason to believe that it is equal to `a`.

@ Serge Abiteboul, Ioana Manolescu, Philippe Rigaux, Marie-Christine Rousset, Pierre Senellart, 2011; to be published by Cambridge University Press 2011.

Using the inverse rule `IN23` to unfold `R(x,f1(v2,v1))` does not help because unifying `R(x,f1(v2,v1))` and `R(v',f2(u,v,v'))` fails because of the two different Skolem functions. Thus, the (partial) rewriting issued from unfolding `U(y,z)` using the inverse rule `IN12` is abandoned.

Let us try now to unfold `U(y,z)` using `IN22` made possible by the substitution

$$\sigma' = \{\texttt{y/v1, v/v1, z/f2(v2,v1,v3), u/v2, v'/v3}\}.$$

We obtain the following (partial) query rewriting:

```
pr'1(x) :- V2(v2,v1,v3), R(x,f2(v2,v1,v3)), T(f2(v2,v1,v3),v1), R(y',x).
```

Now, unfolding `R(x,f2(v2,v1,v3))` using the inverse rule `IN23` is possible thanks to the substitution

$$\sigma'' = \{\texttt{v'/x,v3/x,u/v2,v/v1}\}.$$

This leads to the (partial) rewriting:

```
pr'2(x) :- V2(v2,v1,x), V2(v2,v1,x), T(f2(v2,v1,x),v1), R(y',x),
```

in which one of the first two atoms can be dropped.

Now, we examine the unfolding of the query atom `T(f2(v2,v1,x),v1)`, which requires checking whether `T(f2(v2,v1,x),v1)` and `T(f2(u,v,v'),u)` are unifiable. This is the case thanks to the substitution $\{v2/v3, u/v3, v1/v3, v/v3, v'/x\}$, which leads to the (partial) rewriting:

```
pr'3(x) :- V2(v2,v1,x), V2(v3,v3,x), R(y',x),
```

Again, we can remove the first atom that is redundant and obtain the equivalent (partial) rewriting:

```
pr'4(x) :- V2(v3,v3,x), R(y',x).
```

Finally the unfolding of `R(y',x)` using `IN23` leads to the final rewriting:

```
r1(x) :- V2(v3,v3,x), V1(x,y').
```

4.4 Discussion

The three algorithms have the same (worst-case) complexity and they guarantee to provide the correct answer. Some experiments have shown that in practice Minicon outperforms both Bucket and Inverse-rules. The main advantage of the Inverse-rules algorithm over the Bucket and Minicon algorithms is that the step producing the inverse rules is done independently of the queries. Furthermore, the unfolding step can also be applied to *Datalog* queries, i.e., to *recursive* queries.

@ Serge Abiteboul, Ioana Manolescu, Philippe Rigaux, Marie-Christine Rousset, Pierre Senellart, 2011; to be published by Cambridge University Press 2011.

The common limitation of the three algorithms is that they do not handle additional knowledge (ontology statements) that can be known about the domain of application. In the next section, we see how to extend both the Local-As-Views and Global-As-Views approaches with DL-LITE ontologies, i.e., we consider global schemas that include constraints expressed as DL-LITE axioms.

5 Ontology-based mediators

We first show a negative result: as soon as we add functionality constraints over the global schema, the number of conjunctive rewritings of a query to be considered, may become infinite. This is a severe limitation for extending the LAV or GAV approaches since such constraints are rather natural. So these approaches to data integration fail when we consider the DL-LITE$_{\mathcal{F}}$ dialect of previous chapters. On the positive side, we show how to extend the GAV and LAV approaches to constraints expressible in DL-LITE$_{\mathcal{R}}$.

5.1 Adding functionality constraints

We illustrate on an example the problem raised by taking into account functionality constraints in the global schema. Let us consider a global schema with one unary relation C and two binary relations R and R'. In both R and $R1$, we impose that the first attribute is a key. Let us consider two LAV mappings:

$$\texttt{V1:}\quad \texttt{S(P,N)} \subseteq \texttt{R(P,A)},\ \texttt{R1(N,A)}$$
$$\texttt{V2:}\quad \texttt{V(P)} \subseteq \texttt{C(P)}$$

and the following query:

$$q(x) \text{ :- } R(x,z), R(x_1,z), C(x_1).$$

The three previous algorithms (Bucket, Minicon, and Inverse-rules) would return no rewriting at all for q. The proof is left as an exercise. However, we next show that the following rewriting is valid:

$$r_1(x) \text{ :- } S(x,v_1), S(x_1,v_1), V(x_1)$$

To prove it, we expand $r_1(x)$ and show that the resulting expansion *together with the logical axiom expressing the functionality of R1* logically implies the conjunction of atoms in the body of the query. The expansion of $r_1(x)$ is:

$$Exp\_r_1(x) \text{ :- } R(x,y_1), R1(v_1,y_1), R(x_1,y'_1), R1(v_1,y'_1), C(x_1)$$

Now, if we ignore the functional dependencies, it is not true that $Exp\_r_1 \subseteq q$. But knowing them, the inclusion holds. Indeed, the logical axiom expressing the functionality of *R1* is:

$$\forall y \forall z1 \forall z2 \, [R1(y,z1) \wedge R1(y,z2) \Rightarrow z1 = z2]$$

Therefore, it can be inferred from $R1(v_1,y_1)$ and $R1(v_1,y'_1)$ in the body of $Exp\_r_1(x)$ that $y_1 = y'_1$, and thus:

$$Exp\_r_1(x) \text{ :- } R(x,y_1), R1(v_1,y_1), R(x_1,y_1), R1(v_1,y_1), C(x_1)$$

@ Serge Abiteboul, Ioana Manolescu, Philippe Rigaux, Marie-Christine Rousset, Pierre Senellart, 2011; to be published by Cambridge University Press 2011.

Hence $Exp\_r_1 \subseteq q$ with ψ mapping x, x_1, z to x, x_1, y_1, respectively. Thus $r_1(x)$ is a *valid rewriting* of $q(x)$.

It is important to note that to properly check containment, the standard query containment algorithm seen in the previous section would have to be modified in a standard manner to take into account functional dependencies. Intuitively, one would have to proceed pretty much as we did in the example, equating variables as implied by the functional dependencies.

It turns out that the situation is even more subtle. Surprisingly, this rewriting $r_1(x)$ is not the only one. In fact there exists an infinite number of different rewritings for $q(x)$. Let $k \geq 2$. The following query is a *valid rewriting* of $q(x)$:

$$r_k(x) : S(x,v_k), S(x_k,v_k), S(x_k,v_{k-1}), S(x_{k-1},v_{k-1}), \ldots, S(x_2,v_1), S(x_1,v_1), V(x_1)$$

The expansion of $r_k(x)$ is:

$$\begin{array}{lllll} Exp\_r_k(x) \text{ :-} & R(x,y_k), & R1(v_k,y_k), & & \\ & R(x_k,y'_k), & R1(v_k,y'_k), & & \\ & R(x_k,y_{k-1}), & R1(v_{k-1},y_{k-1}), & & \\ & R(x_{k-1},y'_{k-1}), & R1(v_{k-1},y'_{k-1}), & & \\ & \ldots, & \ldots & & \\ & R(x_2,y_1), & R1(v_1,y_1), & & \\ & R(x_1,y'_1), & R1(v_1,y'_1), & C(x_1). & \end{array}$$

To show that this expansion is logically contained in q, we exploit the axioms of functionality of both R and R_1. Since R_1 is functional, we get: $y_k = y'_k$, and since R is functional, we get: $y'_k = y_{k-1}$. By induction, we obtain $y_k = y'_k = y_{k-1} = y'_{k-1} = \cdots = y_1 = y'_1$, and in particular: $y_k = y'_1$. Thus $Exp\_r_k \subseteq q(x)$. This implies that $r_k(x)$ is a *valid rewriting* of $q(x)$.

One can also show that for each k, each such rewriting may return answers that are not returned with $k-1$. Thus, there exists an infinite number of non redundant conjunctive rewritings. The reader familiar with Datalog will observe that this infinite collection of rewritings can be captured in Datalog by the following *recursive* rewriting:

$$\begin{array}{l} r(x) \text{ :- } S(x,v), S(x_1,v), V(x_1) \\ r(x) \text{ :- } S(x,v), P(v,u), S(x_1,u), V(x_1) \\ P(v,u) \text{ :- } S(z,v), S(z,u) \\ P(v,u) \text{ :- } S(z,v), S(z,w), P(w,u) \end{array}$$

The question of building automatically such conjunctive rewritings is out of the scope of this book (see Section 7).

5.2 Query rewriting using views in DL-LITE$_\mathcal{R}$

Querying data through DL-LITE$_\mathcal{R}$ ontologies has been detailed in the previous chapter. It has been shown how the positive and negative constraints expressed in the ontology are exploited both for data consistency checking and for query answering. In particular, the first step of query answering is the *query reformulation* step which is performed by the *PerfectRef* algorithm: using the positive constraints, called the PIs, it computes a set of *reformulations*, which are then evaluated over the data to produce the answer set of the original query. The negative constraints, called the NIs, are used to check data consistency, by translating each

@ Serge Abiteboul, Ioana Manolescu, Philippe Rigaux, Marie-Christine Rousset, Pierre Senellart, 2011; to be published by Cambridge University Press 2011.

(declared or entailed) NI into a Boolean conjunctive query q_{unsat} that must be evaluated over the data.

In this section, we show how to extend both the LAV and GAV approaches to rewrite queries in term of views when the global schema includes some DL-LITE$_\mathcal{R}$ Tbox.

Two observations explain how this can be realized:

1. First, one can obtain the answer set of a query $\mathtt{q}(\vec{x})$ by computing the union of the answer sets returned by the evaluation over the local data sources of the (GAV or LAV) *relational rewritings* of each *reformulation* of $\mathtt{q}(\vec{x})$ as computed by $\mathtt{PerfectRef}(\mathtt{q}(\vec{x}), PI)$.

2. The rewritings that are obtained may be inconsistent with the negative constraints NI declared or inferred in the Tbox. Therefore, the consistency of each rewriting $r(\vec{x})$ has to be checked. This can be done by checking containment between the Boolean query $\exists \vec{x} Exp\_r(\vec{x})$ (where $Exp\_r(\vec{x})$ is the expansion of $r(\vec{x})$) and each of the Boolean queries q_{unsat} obtained from the NIs.

These two observations follow from the completeness of the *PerfectRef* and *Consistent* algorithms for DL-LITE$_\mathcal{R}$ presented in the previous chapter, and that of the rewriting algorithms of Sections 3 and 4; namely *Unfolding* for GAV, *Minicon*, *Bucket* or *Inverse-rules* for LAV.

The argument may be somewhat too abstract for some readers. We next illustrate these two points with examples. We use the global schema considered in Example 4.2 page 10, enriched with the DL-LITE$_\mathcal{R}$ Tbox of Figure 3. Note in particular that we add the subclass *College* of the class *University*, the subproperty *EnrolledInCollege* of the property *RegisteredTo*, for which the domain is declared as being the class *MasterStudent*. In addition, we add the property *EnrolledInMasterProgram* that we declare as a subproperty of the property *EnrolledInProgram*. Finally, we declare a mandatory property for the class *College*: any college must have students enrolled in it.

DL notation	**FOL notation**
PIs:	
$MasterStudent \sqsubseteq Student$	$MasterStudent(X) \Rightarrow Student(X)$
$EuropeanStudent \sqsubseteq Student$	$EuropeanStudent(X) \Rightarrow Student(X)$
$NonEuropeanStudent \sqsubseteq Student$	$NonEuropeanStudent(X) \Rightarrow Student(X)$
$College \sqsubseteq University$	$College(X) \Rightarrow University(X)$
$FrenchUniversity \sqsubseteq University$	$FrenchUniversity(X) \Rightarrow University(X)$
$EuropeanUniversity \sqsubseteq University$	$EuropeanUniversity(X) \Rightarrow University(X)$
$NonEuropeanUniversity \sqsubseteq University$	$NonEuropeanUniversity(X) \Rightarrow University(X)$
$\exists EnrolledInCollege \sqsubseteq MasterStudent$	$EnrolledInCollege(X,Y) \Rightarrow MasterStudent(X)$
$College \sqsubseteq \exists EnrolledInCollege^-$	$College(X) \Rightarrow \exists Y EnrolledInCollege(Y,X)$
$EnrolledInCollege \sqsubseteq RegisteredTo$	$EnrolledInCollege(X,Y) \Rightarrow RegisteredTo(X,Y)$
$MasterStudent \sqsubseteq \exists EnrolledInMasterProgram$	$MasterStudent(X) \Rightarrow \exists Y EnrolledInMasterProgram(X,Y)$
$\exists EnrolledInMasterProgram^- \sqsubseteq MasterProgram$	$EnrolledInMasterProgram(X,Y) \Rightarrow MasterProgram(Y)$
$EnrolledInMasterProgram \sqsubseteq EnrolledInProgram$	$EnrolledInMasterProgram(X,Y) \Rightarrow EnrolledInProgram(X,Y)$
NIs:	
$NonEuropeanStudent \sqsubseteq \neg EuropeanStudent$	$NonEuropeanStudent(X) \Rightarrow \neg EuropeanStudent(X)$
$NonEuropeanUniversity \sqsubseteq \neg EuropeanUniversity$	$NonEuropeanUniversity(X) \Rightarrow \neg EuropeanUniversity(X)$
$NonEuropeanUniversity \sqsubseteq \neg FrenchUniversity$	$NonEuropeanUniversity(X) \Rightarrow \neg FrenchUniversity(X)$

Figure 3: A DL-LITE$_\mathcal{R}$ Tbox enriching the global schema of Example 4.2

@ Serge Abiteboul, Ioana Manolescu, Philippe Rigaux, Marie-Christine Rousset, Pierre Senellart, 2011; to be published by Cambridge University Press 2011.

5.2.1 GAV and DL-LITE$_{\mathcal{R}}$

We revisit Example 3.2 by adding the data source `S5` giving the list of French so-called *Grandes Ecoles*. Its local schema is made of the local relation: `S5.GrandeEcole(nomEcole)`. According to this new source and also to the enriched global schema of Figure 3, we add the following GAV mappings to the ones already considered in Example 3.2:

```
College(U) ⊇ S5.GrandeEcole(U)
EuropeanStudent(N) ⊇ S2.Erasmus(N,C,U)
NonEuropeanStudent(N) ⊇ S3.CampusFr(N,P,U)
```

Consider again the global query looking for universities with registered master students:

```
q(x) :- RegisteredTo(s,x), MasterStudent(s)
```

It is left as an exercise to show that the application of the *PerfectRef(q(x), PI)* algorithm returns, in addition to `q(x)` itself, the reformulation:

```
q1(x) :- College(x)
```

By unfolding `q(x)`, we obtain the same two rewritings as in Example 3.2:

```
r1(x) :- S3.CampusFr(s,v1,x), S2.Erasmus(s,v2,v3), S4.Mundus(v4,v2)
r2(x) :- S3.CampusFr(s,v6,x), S4.Mundus(v6,v8)
```

By unfolding the reformulation `q1(x)`, we get the additional rewriting:

```
r3(x) :- S5.GrandeEcole(x)
```

It is important to note that even if we had the GAV mapping

```
College(U) ⊇ S5.GrandeEcole(U),
```

the rewriting `r3(x)` would not have been obtained without reformulated first the initial query `q(x)` into `q1(x)`.

Now, in contrast with the standard GAV approach, we have to check the consistency of each of these rewritings. To do so:

- We first compute the closure of the NI and we translate them into Boolean queries q_{unsat} (as explained in detail in Section **??** of Chapter **??**). This is independent of the rewritings and can be performed at compile time given the Tbox. From the Tbox in Figure 3, we obtain only three Boolean queries q_{unsat}:

q^1_{unsat} :- `NonEuropeanStudent(x), EuropeanStudent(x)`
q^2_{unsat} :- `NonEuropeanUniversity(x), EuropeanUniversity(x)`
q^3_{unsat}, :- `NonEuropeanUniversity(x), FrenchUniversity(x)`

@ Serge Abiteboul, Ioana Manolescu, Philippe Rigaux, Marie-Christine Rousset, Pierre Senellart, 2011; to be published by Cambridge University Press 2011.

- At query time, we check the consistency of each rewriting by applying the *Consistent* algorithm to the canonical instance obtained by expanding each rewriting and freezing its variables (as explained in detail in Section **??** of Chapter**??**).

We illustrate the consistency check by checking the consistency of the rewriting `r1(x)`. First, its expansion replaces each of its local atoms $S(\vec{z})$ with the conjunction of global atoms of the form $G(\vec{z})$ that can be produced by a GAV mapping $G(\vec{x}) \supseteq S(\vec{x})$, if such GAV mappings exist. For expanding `r1(x)`, we apply the following GAV mappings:

```
NonEuropeanStudent(N) ⊇ S3.CampusFr(N,P,U)
University(U) ⊇ S3.CampusFr(N,P,U)
RegisteredTo(N,U) ⊇ S3.CampusFr(N,P,U)
EuropeanStudent(N) ⊇ S2.Erasmus(N,C,U)
University(U) ⊇ S2.Erasmus(N,C,U)
MasterProgram(T) ⊇ S4.Mundus(T,C)
MasterCourse(C) ⊇ S4.Mundus(T,C)
```

As a result, we obtain the following expansion for `r1(x)`:

```
Exp_r1(x) :- NonEuropeanStudent(s), University(x), RegisteredTo(s,x),
             EuropeanStudent(s), University(x), MasterProgram(v4),
             MasterCourse(v2)
```

We then apply the *Consistent* algorithm. For this, we evaluate q^1_{unsat}, q^2_{unsat} and q^3_{unsat} over the body of `Exp_r1(x)` seen as a relational database; i.e., we freeze its atoms to obtain a canonical instance. Query q^1_{unsat} returns *true*, so an inconsistency has been detected and the rewriting `r1(x)` is rejected.

5.2.2 LAV and DL-LITE$_\mathcal{R}$

We revisit Example 4.2 by adding the same data source `S5` as in Section 5.2.1. The GAV mapping is also a LAV mapping: `S5.GrandeEcole(U)` $\subseteq$ `College(U)`

Consider again the global query considered in Section 4.1:

```
q(x) :- RegisteredTo(s,x), EnrolledInProgram(s,p), MasterProgram(p)
```

It is left as an exercise to show that the application of the *PerfectRef(q(x), PI)* algorithm returns, in addition to `q(x)` itself, the following reformulation:

```
q1(x) :- College(x)
```

By applying the Minicon algorithm[1] to the initial query `q(x)`, we obtain the following rewriting (as shown in Section 4.1):

```
r2(x) :- S3.CampusFr(s,v1,x), S3.CampusFr(s,p,v2),S4.Mundus(p,v5)
```

[1]The same holds for the Bucket or Inverse-rules algorithm.

@ Serge Abiteboul, Ioana Manolescu, Philippe Rigaux, Marie-Christine Rousset, Pierre Senellart, 2011; to be published by Cambridge University Press 2011.

By applying the Minicon algorithm to the reformulation `q1(x)` of the initial query, we obtain the additional rewriting:

```
r3(x) :- S5.GrandeEcole(x)
```

As for the extended GAV approach, the consistency of LAV rewritings is not guaranteed because of the NIs in the Tbox. We follow the same approach: at compile time, the closure of the NIs is computed and each (declared or inferred) NI is compiled into a Boolean query q_{unsat}. At query time, each of these q_{unsat} queries is evaluated over the canonical instance corresponding to each rewriting.

6 Peer-to-Peer Data Management Systems

In contrast with the centralized mediator model, a Peer-to-Peer data management system (PDMS for short) implements a decentralized view of data integration, in which data sources collaborate without any central authority. In a PDMS, each collaborating data source can also play the role of a mediator, so is at the same time a data server and a client for other data sources. Thus each participant to the system is a *peer* and there are mappings relating data from the different peers. A PDMS architecture is therefore very flexible in the sense that there is no need for a global schema defining in advance a set of terms to which each data source needs to adhere. Over time, data sources can join or leave the PDMS just by adding or removing mappings between them. PDMS are inspired by P2P file sharing systems but they enable answering fine-grained queries. Like in the mediator model, answering queries is performed by reformulating queries based on the mappings, but in a decentralized manner.

Each peer in a PDMS has a peer schema composed of peer relations and peer mappings that relate its schema to the schemas of other peers. To avoid confusing relations from different peers, we assume that each relation of peer p is of the form $r@p$ for some local relation name r. A query to a PDMS is posed using the peer schema of one of the peers. A query is asked to a particular peer, as a query over his particular schema. It is reformulated using the peer mappings into a set of queries that may refer to other peer relations. This captures the intuition that we want to use the information available in the entire P2P system to answer the query.

For designing the mappings, the distinction made in the mediator model between local and global relations does not make sense anymore, since each peer relation may play the role at different times both of a local relation and of a global relation. Therefore, the notions of GAV and LAV mappings are relaxed to the more appropriate symmetric notion of GLAV mappings.

Definition 6.1 (GLAV mapping) *Let S@i and S@j be the peer schemas of two peers i and j. A* GLAV mapping *between these two peers is an inclusion axiom of the form: $q_i(\vec{x}) \subseteq q_j(\vec{x})$, where $q_i(\vec{x})$ and $q_j(\vec{x})$ are conjunctive queries over the peer schema S@i, S@j, respectively.*

Let $q_i(\vec{x},\vec{y}_i)$ and $q_j(\vec{x},\vec{y}_j)$ be the bodies of $q_i(\vec{x})$ and $q_j(\vec{x})$), respectively. The semantics of the GLAV mapping $q_i(\vec{x}) \subseteq q_j(\vec{x})$ is: $\forall\vec{x}[\exists\vec{y}_i\, q_i(\vec{x},\vec{y}_i) \Rightarrow \exists\vec{y}_j\, q_j(\vec{x},\vec{y}_j)]$.

In database terms, a GLAV mapping $q_i(\vec{x}) \subseteq q_j(\vec{x})$ expresses that answers obtained by asking $q_i(\vec{x})$ at peer i should also be considered as answers to $q_j(\vec{x})$ asked at peer j. Note

@ Serge Abiteboul, Ioana Manolescu, Philippe Rigaux, Marie-Christine Rousset, Pierre Senellart, 2011; to be published by Cambridge University Press 2011.

that with this semantics, each local query is assumed to be *incompletely* answered with local data since external data may bring in new information to it. As already mentioned, such an open-world assumption is fully appropriate for Web data.

We next show one negative and one positive result for PDMSs. In Section 6.1, we show that in general, answering queries with GLAV mappings is undecidable, so without further restriction, answering queries in a PDMS is undecidable. In Section 6.2, we show that if we restrict the peer mappings to be DL-LITE$_\mathcal{R}$ inclusion axioms, a decentralized version of the algorithm for DL-LITE$_\mathcal{R}$ can be used to answer queries in DL-LITE$_\mathcal{R}$ PDMSs.

6.1 Answering queries using GLAV mappings is undecidable

We show that the *Dependency Implication Problem* (more precisely, the problem of the implication of an inclusion dependency from a set of inclusion and functional dependencies) can be reduced to the *GLAV Query Answering Problem*, i.e., the problem of answering queries in presence of GLAV mappings. Since the *Dependency Implication Problem* is known to be undecidable, this shows that the *GLAV Query Answering Problem* is also undecidable.

The reduction technique is standard for proving undecidability results. We first recall how it works and also recall the *Dependency Implication Problem*. We believe that these notions are important to know beyond the scope of this book. Finally, we use a reduction to show the undecidability of answering queries using GLAV mappings.

Reduction from a decision problem B to a decision problem B′

Let B be a Boolean function over a set X. The *decision problem* B is *decidable* if there exists an algorithm (in any computation model equivalent to Turing machines) that terminates on any input $x \in B$ and returns "true" *if and only if* $B(x)$ is true.

Let B, B' be two decision problems. A *reduction* from B to B' is an algorithm f computing a function (also denoted f) from X to X' such that: $B(x)$ is true $\Leftrightarrow$ $B'(f(x))$ is true.

It is immediate to see that if there is a reduction f from B to B':

- if B' is decidable then B is. Suppose B' is decidable. Let $f_{B'}$ be an algorithm that given some $x' \in X'$, decides whether $B'(x')$ holds. Then for each x, $B(x)$ is true if $f_{B'}(f(x))$ is true. This provides an algorithm for deciding for any x if $B(x)$ is true.
- (The contraposite) if B is undecidable, then B' is also undecidable.

The Dependency Implication Problem

We recall the class of dependencies that are used. Let R be a relation of arity n. Then:

Functional dependencies. A *functional dependency* over R is an expression $R : i_1 ... i_m \rightarrow j$, where $1 \leq i_1, ..., i_m, j \leq n$, for $n = arity(R)$. An instance I over R satisfies $R : i_1 ... i_m \rightarrow j$ if for each tuples $\langle a_1, ... a_n \rangle$, $\langle b_1, ... b_n \rangle$ in I,

$$\text{if for each } k \in [1..m],\ a_{i_k} = b_{i_k}, \text{ then } a_j = b_j.$$

Inclusion dependencies. An *inclusion dependency* over R_1, R_2 is an expression $R_1 : i_1 ... i_m \subseteq R_2 : j_1 ... j_m$, where the i_k are distinct, the j_k are distinct, $1 \leq i_1, ..., i_m \leq arity(R_1)$, $1 \leq$

@ Serge Abiteboul, Ioana Manolescu, Philippe Rigaux, Marie-Christine Rousset, Pierre Senellart, 2011; to be published by Cambridge University Press 2011.

$j_1,...,j_m \leq arity(R_2)$. An instance I over $\{R_1, R_2\}$ satisfies $R_1 : i_1...i_m \subseteq R_2 : j_1...j_m$ if for each tuple $\langle a_1,...,a_n\rangle$ in $I(R_1)$, there exists a tuple $\langle b_1,...,b_{n'}\rangle$ in $I(R_2)$ such that for each k, $1 \leq k \leq m$, $a_{i_k} = b_{j_k}$.

We will use the following known result:

Theorem 6.2 (Undecidability of the *Dependency Implication Problem*). Let $\mathbf{R} = \{R_1,...,R_n\}$ be a relational schema. Given a set Σ of functional and inclusion dependencies and an inclusion dependency σ over relations in $\mathbf{R}$, one cannot decide whether $\Sigma \models \sigma$ (i.e., whether each instance over $\mathbf{R}$ satisfying Σ also satisfies σ).

The problem is undecidable when Σ contains both functional and inclusion dependencies. Note that the implication problem is decidable for functional dependencies alone, and for inclusion dependencies alone. Undecidability arises when they are considered together.

Undecidability of the GLAV Query Answering Problem

The *GLAV Query Answering Problem* is to decide, given a PDMS $\mathcal{N}$ defined using a set of GLAV mappings and a query asked at one of the peers whether some particular tuple is in its answer.

Let us define a reduction from the *Dependency Implication Problem* to the *GLAV Query Answering Problem*. If we show that such a reduction exists, since the *Dependency Implication Problem* is undecidable, this will show that the *GLAV Query Answering Problem* is undecidable.

Surprisingly, we can show the reduction for a PDMS with a single peer. To do that, we will use some GLAV mapping of the form $q@P \supseteq q'@P$, where both sides of the mapping involve the same peer. Note that the undecidability still holds if such "self" mappings are forbidden. Indeed, we can simulate such a mapping by using "clones" of relations. For instance, suppose that we want to enforce the mapping $R@P(x_1,...,x_n) \supseteq R'@P(y_1,...,y_n)$. Then we can use a dummy site $\widehat{P}$ and a copy $\widehat{R}@\widehat{P}$ of $R@P$ with the mappings:

$$R@P(x_1,...,x_n) \supseteq \widehat{R}@\widehat{P}(x_1,...,x_n)$$
$$\widehat{R}@\widehat{P}(x_1,...,x_n) \supseteq R@P(x_1,...,x_n)$$
$$\widehat{R}@\widehat{P}(x_1,...,x_n) \supseteq R'@P(y_1,...,y_n)$$

So, in the rest of this proof, we consider a single peer, say P, with possibly self mappings. To simplify a relation $R@P$ is simply denoted R.

Let (Σ,σ) be an instance over $\{R_1,...,R_n\}$ of the *Dependency Implication Problem* with Σ a finite set of functional and inclusion dependencies, and σ an inclusion dependency. We build a PDMS $\mathcal{N}$ defined as follows:

- For each relation R_i, the peer P has a relation R_i.
- For each inclusion dependency $R_1 : i_1...i_m \subseteq R_2 : j_1...j_m$ in Σ, we add the GLAV mapping $q_1 \subseteq q_2$, where:

$$q_1(x_1,...,x_m) \text{ :- } R_1(\vec{u})$$
$$q_2(x_1,...,x_m) \text{ :- } R_2(\vec{v})$$

  where $\vec{u}$ has x_k in position i_k for each k and some existential variable x_j^i in each other position j; and similarly for $\vec{v}$ and j_k.

@ Serge Abiteboul, Ioana Manolescu, Philippe Rigaux, Marie-Christine Rousset, Pierre Senellart, 2011; to be published by Cambridge University Press 2011.

- For each functional dependency $R_i : i_1 ... i_m \rightarrow j$ in Σ, we add the GLAV mapping $q \subseteq q'$ where q, q' are defined by:

$$\begin{array}{lcl} q(x_{i_1},...,x_{i_k},x_j,x'_j) & :- & R_i(x_1,...,x_k), R_i(x'_1,...,x'_k), x_{i_1} = x'_{i_1},...,x_{i_k} = x'_{i_k} \\ q'(x_{i_1},...,x_{i_k},x_j,x'_j) & :- & R_i(x_1,...,x_k), R_i(x'_1,...,x'_k), x_{i_1} = x'_{i_1},...,x_{i_k} = x'_{i_k}, \mathbf{x_j = x'_j} \end{array}$$

  for some distinct sets $x_1,...,x_k$ and $x'_1,...,x'_k$ of variables.

It is easy to see that the GLAV mappings force each R_i to satisfy the functional dependencies of R_i, and each R_i, R_j to satisfy the inclusion dependencies between R_i and R_j.

Let us assume that $\sigma = R_i : i_1 \subseteq R_j : j_1$ for some R_i of arity n. (This is without loss of generality since the implication is already undecidable when σ is unary).

Let $Ext(R_i)$ be the set of tuples t of arity n with values in $[1..n]$ such that:

- $t[i_1] = 1$, for every tuple t in $Ext(R_i)$,
- each tuple t in $Ext(R_i)$ represents an equality pattern between values in tuples of size n.

For instance if $n = 3$ and $i_1 = 2$, $Ext(R_i) = \{\langle 1,1,1\rangle, \langle 1,1,2\rangle, \langle 2,1,1\rangle, \langle 2,1,2\rangle, \langle 2,1,3\rangle\}$.

We construct an instance $(\mathcal{N}, Ext(R_i), q)$ of the *GLAV Query Answering Problem* where q is the query defined by $q(x)$:- $R_j(y_1,...,x,...y_k)$ where the distinguished variable x is in position j_1, and the existential variables y_i are pairwise distinct.

We show that $\Sigma \models \sigma$ iff 1 is an answer to q in the PDMS $\mathcal{N}$ in which the only data is $Ext(R_i)$.

($\Rightarrow$) Suppose that $\Sigma \models \sigma$. Let I be a model of $Ext(R_i)$ satisfying the GLAV mappings of $\mathcal{N}$. By construction of those GLAV mappings, I is a model of Σ. Because $\Sigma \models \sigma$, I is a model of σ, and thus for each tuple $\langle a_1,...,a_n\rangle$ in $I(R_i)$, there exists a tuple $\langle b_1,...,b_k\rangle$ in $I(R_j)$ such that $a_{i_1} = 1 = b_{j_1}$. Therefore, $I \models \exists y_1,...,y_k R_j(y_1,...,1,...y_k)$, i.e., $I \models q(1)$. Thus 1 is an answer to q given the GLAV mapping of $\mathcal{N}$ and the extension $Ext(R_i)$.

($\Leftarrow$) Conversely, suppose that 1 is an answer to q given the GLAV mapping of $\mathcal{N}$ and the extension $Ext(R_i)$. Note that 1 is also an answer to q if the extension of R_i is reduced to any tuple of the original $Ext(R_i)$. Suppose that $\Sigma \not\models \sigma$: there exists an interpretation I that satisfies Σ in which σ is not satisfied. This means that there exists a tuple $\langle e_1,...,e,...e_n\rangle$ (where e is in position i_1) in $I(R_i)$ such that there does not exists a tuple in $I(R_j)$ with the value e in position j_1. Let t be the tuple of $Ext(R_i)$ which corresponds to the equality pattern between values of $\langle e_1,...,e,...e_n\rangle$. By extending I to interpret each value of t by the element e_i at the same position in $\langle e_1,...,e,...e_n\rangle$, we obtain a new interpretation I' that satisfies Σ and thus each GLAV mapping of $\mathcal{N}$, and $R_i(t)$. Since 1 is an answer to q given the GLAV mapping of $\mathcal{N}$ and $R_i(t)$, $I' \models q(1)$, i.e., $I'(1) \in I'(R_j[j_1])$. Since $I'(1) = e$ and $I'(R_j) = I(R_j)$, it means that there exists a tuple in $I(R_j)$ with the value e in position j_1, which contradicts our assumption that σ is not satisfied in I. Hence $\Sigma \models \sigma$. □

6.2 Decentralized DL-LITE$_\mathcal{R}$

If we restrict the GLAV mappings in a PDMS to be inclusion statements that are expressible in DL-LITE$_\mathcal{R}$, we get what we will call a DL-LITE$_\mathcal{R}$ PDMS. The decidability of query answering over a DL-LITE$_\mathcal{R}$ PDMS results from the algorithmic machinery described in the previous chapter for answering queries over DL-LITE$_\mathcal{R}$ knowledge bases. Given a query posed to a

@ Serge Abiteboul, Ioana Manolescu, Philippe Rigaux, Marie-Christine Rousset, Pierre Senellart, 2011; to be published by Cambridge University Press 2011.

given peer, the application of the *PerfectRef* algorithm to the set of all the GLAV mappings in the PDMS provides a set of reformulations. The union of the answer sets obtained by evaluating each reformulation provides the answer set of the initial query. Note that a reformulation is of the form:

$$R_1@i_1(\vec{z_1}), \ldots, R_k@i_k(\vec{z_k})$$

where the different conjuncts $R_j@i_j(\vec{z_j})$ may refer to relations of different peer schemas. Therefore, the evaluation of each reformulation may require the interrogation of different peers and the combination of the answers returned by each such sub-queries.

This provides a centralized algorithm for computing the reformulations of answering queries over a *decentralized* DL-LITE$_\mathcal{R}$ knowledge base. We next present a *decentralized* algorithm that computes exactly the same thing, i.e., we present a decentralized version of the *PerfectRef* algorithm seen in the previous chapter in order to deploy effectively DL-LITE$_\mathcal{R}$ PDMSs that avoids having to centralize all the GLAV mappings.

We denote $PerfectRef^i(q)$ the reformulation algorithm running on the peer P_i applied to a query q (asked to the peer P_i). The main procedure is the decentralized reformulation of each atom of the query using the positive inclusion statements that are distributed over the whole PDMS. Let us denote $AtomRef^i(g)$ the reformulation algorithm running on the peer P_i to reformulate the atom g (built on a relation of the schema of the peer P_i).

Within each peer P_i we distinguish the *local* positive inclusion axioms of the form $C_i \subseteq D_i$ where C_i and D_i are built over relations in the schema of the peer P_i, from the *mappings* which are positive inclusion mappings of the form $C_j \subseteq D_i$ or $D_i \subseteq C_j$ where C_j denotes a relation of another peer P_j (while D_i refers to a relation in the schema of the peer P_i).

Let us denote $LocalRef(g, PI_i)$ the result of the reformulation of the atom g using the set PI_i of local positive inclusion atoms of the peer P_i. We refer to the previous chapter (Definition **??**, Section **??**) for the computation of $LocalRef(g, PI_i)$ by backward application of the local PIs.

We just recall here that $gr(g, I)$ denotes the reformulation of the atom g using the positive inclusion axiom I. We also recall that the atoms g that can be found as conjunct of a query q over a DL-LITE$_\mathcal{R}$ PDMS are of the following forms:

- $A@i(x)$ where $A@i$ is a unary relation in the schema of a peer P_i and x an (existential or qualified) variable
- $P@i(x,\_)$, $P@i(\_,x)$ or $P@i(x,y)$ where $P@i$ is a binary relation in the schema of a peer P_i, and _ denotes an unbounded existential variable of the query, while x and y denote qualified variables or existential variables which are bounded in the query.

Running the algorithm $AtomRef^i$ on the peer P_i for reformulating the atom g consists first in computing the set $LocalRef(g, PI_i)$ of local reformulations of g, and then, for each mapping m with a peer P_j applicable to a local reformulation g', in triggering the application of $AtomRef^j(gr(g', m))$ on P_j (by sending a message to P_j). Other peers P_k may be solicited in turn to run locally $AtomRef^k$.

A loop may occur if a request of reformulation of an atom g initiated by a given peer P generates a branch of requests reaching a peer P' which in turn requests P to reformulate g. Such loops can be easily handled by transmitting with every request the *history* of the current reasoning branch. More precisely, an history *hist* is a sequence $[(g_k, P_k), \ldots, (g_1, P_1)]$ of pairs

@ Serge Abiteboul, Ioana Manolescu, Philippe Rigaux, Marie-Christine Rousset, Pierre Senellart, 2011; to be published by Cambridge University Press 2011.

(g_i, P_i) where g_i is an atom of a peer P_i such that for each $i \in [1..k-1]$, g_{i+1} is a reformulation of g_i using a mapping between P_i and P_{i+1}.

This is summarized in Algorithm 3, which is the atom reformulation algorithm with history running on Peer i.

Algorithm 3: The *decentralized* algorithm with history for reformulating atoms
$AtomRefHist^i(g, hist)$
Input: An atom g in the vocabulary of the peer P_i, an history *hist*
Output: The set of its reformulations in the PDMS: R
(1) $R \leftarrow \emptyset$
(2) **if** $(g, P_i) \in hist$ **return** R
(3) **else**
(4) Let PI_i be the local PIs of the peer P_i
(5) Let M_i be the mappings between the peer P_i and other peers
(6) **for each** $g' \in LocalRef(g, PI_i)$
(7) **for each** mapping $m \in M_i$ between P_i and a peer P_j applicable to g'
(8) $R \leftarrow R \cup AtomRefHist^j(gr(g', m), [(g, P_i)|hist])$

Algorithm 4 is the atom reformulation algorithm (denoted $AtomRef^i$) running on peer P_i, which just calls $AtomRefHist^i$ with an empty history.

Algorithm 4: The *decentralized* algorithm for reformulating atoms
$AtomRef^i(g)$
Input: An atom g in the vocabulary of the peer P_i
Output: The set of its reformulations in the PDMS
(1) $AtomRefHist^i(g, \emptyset)$

The decentralized version of the *PerfectRef* algorithm that computes all the reformulations of a conjunctive query q is provided in Algorithm 5. The main difference with the centralized version is that the simplification of the produced reformulations (which is required for making some PIs applicable) are delayed after (decentralized) computation of the reformulations of all the atoms in the query.

We recall here the notation used for denoting the simplification of some atoms within a query under reformulation, which were introduced in the previous chapter when describing the *PerfectRef* algorithm:

- The notation $q[g/gr(g, I)]$ denotes the replacement of the atom g in the body of the query q with the result $gr(g, I)$ of the backward application of the PI I to the atom g,
- The operator $reduce(q, g, g')$ denotes the simplification of the body of q obtained by replacing the conjunction of its two atoms g and g' with their *most general unifier* (if g and g' can be unified),
- The operator τ replaces in the body of a query all the possibly new *unbounded* existential variables with the anonymous variable denoted _.

@ Serge Abiteboul, Ioana Manolescu, Philippe Rigaux, Marie-Christine Rousset, Pierre Senellart, 2011; to be published by Cambridge University Press 2011.

For each atom in the query, it computes first (in the decentralized manner explained previously) the set of all of its reformulations, and then a first set of reformulations of the original query by building all the conjunctions between the atomic reformulations (denoted $\oplus_{i=1}^{n} AtomRef^i(g_i)$ at Line 5). These reformulations are then possibly simplified by unifying some of their atoms (Lines 8 to 11), and the reformulation process is iterated on these newly produced reformulations until no simplification is possible (general loop starting on Line 4).

Algorithm 5: The decentralized *PerfectRef* algorithm running on the peer P_i
$PerfectRef^i(q)$
Input: a conjunctive query q over the schema of the peer P_i
Output: a set of reformulations of the query using the union of PIs and mappings in the PDMS

(1) $PR := \{q\}$
(2) $PR' := PR$
(3) **while** $PR' \neq \emptyset$
(4) (a) **foreach** $q' = g_1 \wedge g_2 \wedge \ldots \wedge g_n \in PR'$
(5) $PR'' := \oplus_{i=1}^{n} AtomRef^i(g_i)$
(6) $PR' := \emptyset$
(7) (b) **foreach** $q'' \in PR''$
(8) **foreach** $g_1', g_2' \in q''$
(9) **if** g_1' and g_2' unify
(10) $PR' := PR' \cup \{\tau(reduce(q'', g_1', g_2'))\}$
(11) $PR := PR \cup PR' \cup PR''$
(12) **return** PR

One can prove that the decentralized algorithm computes the same set of facts as the centralized one, and thus is correct. The proof results (1) from the observation that the centralized version of $PerfectRef^i$ (in which $AtomRef^i(g_i)$ is computed by iterating the one-step application of PIs on each atom g_i of the query) produces the same results than the original *PerfectRef*, and (2) from the completeness of $AtomRef^i(g_i)$ ensuring the decentralized computation of all the reformulations of g_i.

7 Further reading

The Bucket and Minicon algorithms can be extended ([LRO96, PH01]) to handle (union of) conjunctives queries with *interpreted predicates*. When a query q includes interpreted predicates, finding all answers to q given the LAV mappings is co-NP hard in the size of the data. This complexity result shows that answering such queries cannot be fully realized with a finite set of conjunctive rewriting (unlike what we showed here in absence of interpreted predicates). The Inverse-rule algorithm does not handle interpreted predicates but is able to build *recursive* query plans for data integration [DGL00].

A survey on answering queries using views can be found in [Hal01], and a survey on query containment for data integration systems in [MHF03].

More material can be found on PDMS in [HIST03, HIDT05].

Distributed reasoning in a peer to peer setting has been investigated in [ACG+06] as a

@ Serge Abiteboul, Ioana Manolescu, Philippe Rigaux, Marie-Christine Rousset, Pierre Senellart, 2011; to be published by Cambridge University Press 2011.

basis for querying distributed data through distributed ontologies [GR03, AGR09]. The subtle point that we have not treated in this chapter concerns consistency checking. In contrast with the centralized case, the global consistency of the PDMS cannot be checked at query time since the queried peer does not know all the peers in the PDMS. However, it can get the identifiers of the peers involved in a reformulation of the query. Then the (local) consistency of the union of the corresponding knowledge bases can be checked in a decentralized manner. The important point is that it can be shown that this local consistency is sufficient to guarantee that the answers obtained by evaluating the reformulations (computed by the decentralized algorithm that we have described) are *well-founded*.

The undecidability of the *Dependency Implication Problem* is shown in [CV85] even if σ is a unary inclusion dependency. More on this topic may be found in [ARV95])

8 Exercices

Exercise 8.1 *By applying the query containment algorithm (see Algorithm 1), determine which query is contained in which one among the three following queries. Are there equivalent queries ? (two queries q and q′ are equivalent if q is contained in q′ and q′ is contained in q).*

```
q1(x) :- A(x,y), B(x,y'), A(y,z')

q2(x) :- A(x,y'), A(y',z), B(x,x)

q3(x) :- B(x,y), A(x,y'), B(z,z'), A(y',u)
```

Exercise 8.2 *Consider a global schema defined by the following relations:*
emp(E): E is an employee
phone(E,P): E has P as phone number
office(E,O): E has O as office
manager(E,M): M is the manager of E
dept(E,D): D is the department of E
Suppose that the three following data sources are available for providing data:
Source1 provides the phone number and the manager for some employees. It is modeled by the local relation s1(E,P,M).
Source2 provides the office and the department for some employees. It is modeled by the local relation s2(E,O,D).
Source3 provides the phone number of employees of the 'toy' department. It is modeled by the local relation s3(E,P).

1. *Model the content of these sources by GAV mappings.*
2. *Model the content of these sources by LAV mappings.*
3. *Consider the global query asking for Sally's phone number and office:*

```
q(x,y) :- phone('sally', x), office('sally', y)
```

@ Serge Abiteboul, Ioana Manolescu, Philippe Rigaux, Marie-Christine Rousset, Pierre Senellart, 2011; to be published by Cambridge University Press 2011.

Compute the reformulation of the query in terms of local relations:

- *by applying the query unfolding algorithmto the GAV mappings of Question 1,*
- *by applying the Bucket algorithm to LAV mappings of Question 2.*

Which algorithm is easier ?

4. *Now Source1 disappears (becomes unavailable) and a new source comes in, that provides the phone number of their manager for some employees. Do the updates in the GAV and LAV mappings that are required to take into account these changes. What is the approach (GAV or LAV) for which updating the mappings between the global and local relations is easier ?*

Exercise 8.3 *Consider the three following LAV mappings:*
V1(x) $\subseteq$ cite(x,y), cite(y,x)
V2(x,y) $\subseteq$ sameTopic(x,y)
V3(x,y) $\subseteq$ cite(x,z), cite(z,x), sameTopic(x,z)

1. *Provide the FOL semantics of these LAV mappings*

2. *Suppose that the global relation cite(x,y) means that the paper x cites the paper y, and that the global relation sameTopic(x,y) means that the two papers x and y have the same topic. Suppose that each LAV mapping models the content of different available data sources. Express with an english sentence which information on papers each data source provides.*

3. *Apply in turn the Bucket, Minicon and Inverse-rule algorithms to compute the different rewritings of the following query asking for papers that cite and are cited by a paper having the same topic:*

```
q(u) :- cite(u,v), cite(v,u), sameTopic(u,v)
```

[ACG+06] P. Adjiman, P. Chatalic, F. Goasdoué, M.-C. Rousset, and L. Simon. Distributed reasoning in a peer-to-peer setting. *Journal of Artificial Intelligence Research*, 25, 2006.

[AGR09] Nada Abdallah, Francois Goasdoué, and Marie-Christine Rousset. DL-LITE$_\mathcal{R}$ in the Light of Propositional Logic for Decentralized Data Management. In *Proc. Intl. Joint Conference on Artificial Intelligence (IJCAI)*, 2009.

[ARV95] S. Abiteboul, R.Hull, and V. Vianu. *Foundations of Databases*. Addison-Wesley, 1995.

[CV85] A.K. Chandra and M. Vardi. The implication problem for functional and inclusion dependencies is undecidable. *SIAM Journal on Computing*, 14(3):671–677, 1985.

[DGL00] O.M. Duschka, M.R. Genesereth, and A. Y. Levy. Recursive query plans for data integration. *Journal of Logic Programming*, 43(1):49–73, 200.

[GR03] F. Goasdoué and M.-C. Rousset. Querying distributed data through distributed ontologies: A simple but scalable approach. *IEEE Intelligent Systems (IS)*, 18(5):60–65, 2003.

[Hal01] A. Y. Halevy. Answering queries using views: A survey. *Very Large Databases Journal (VLDBJ)*, 10(4):270–294, 2001.

@ Serge Abiteboul, Ioana Manolescu, Philippe Rigaux, Marie-Christine Rousset, Pierre Senellart, 2011; to be published by Cambridge University Press 2011.

[HIDT05] A.Y. Halevy, Z.G Ives, D.Suciu, and I. Tatarinov. Schema Mediation for Large-Scale Semantic Data Sharing. *Very Large Databases Journal (VLDBJ)*, 14(1):68–83, 2005.

[HIST03] A. Y. Halevy, Z. G. Ives, D. Suciu, and I. Tatarinov. Schema mediation in peer data management systems. In *Proc. Intl. Conf. on Data Engineering (ICDE)*, 2003.

[LRO96] A. Y. Levy, A. Rajaraman, and J. Ordille. Querying heterogeneous information sources using source descriptions. In *Proc. Intl. Conf. on Very Large Databases (VLDB)*, 1996.

[MHF03] Todd D. Millstein, Alon Y. Halevy, and Marc Friedman. Query containment for data integration systems. *Journal of Computer and System Sciences*, 66(1):20–39, 2003.

[PH01] R. Pottinger and A. Y. Halevy. Minicon: A scalable algorithm for answering queries using views. *Very Large Databases Journal (VLDBJ)*, 10(2-3):182–198, 2001.

@ Serge Abiteboul, Ioana Manolescu, Philippe Rigaux, Marie-Christine Rousset, Pierre Senellart, 2011; to be published by Cambridge University Press 2011.