17 Peer-to-Peer Integration

In all the structured query-based data integration and warehousing architectures we have described so far in this book, one must create a mediated schema or centralized database for the domain of the data integration application, before any queries may be posed. This task requires significant modeling, maintenance, and coordination effort among the owners of the data sources. In particular, creating a mediated schema entails that the *scope* of the domain be well defined and that there exists a clearly identifiable schema for it, which is not necessarily true. For example, consider data sharing in a scientific context, where data may involve scientific findings from multiple disciplines, such as genomic data, diagnosis and clinical trial data, bibliographic data, and drug-related and clinical trial data. Each discipline may have its own way of conceptualizing the domain. Hence, attempting to create a single mediated schema that covers all of these topics is likely to fail.

There are an increasing number of scenarios in which owners of data sources want to collaborate, but without any central authority and global standardization. The collaboration may start with two or three data sources wanting to exchange their data, and then it may grow as new data sources become available or as the requirements change. However, the owners of the data might not want to explicitly create a mediated schema that defines the entire scope of the collaboration and a set of terms to which every data source owner needs to map.

These problems of decentralized collaboration have become a recent emphasis in the data integration literature. In this chapter we focus on how one enables decentralized, collaborative querying of data that are of high quality, through a variety of different schemas. Chapter 18, particularly in Section 18.4, builds upon many of these ideas to support scenarios where data are of nonuniform quality and may be subject to updates.

The basic approach of *peer data management systems*, or PDMSs, is to eliminate the reliance on a central, authoritative mediated schema. Rather, each *participant* or *peer* in the system has the ability to define its own schema, both for its source data (*stored relations*) and for integrated querying of the data (*peer relations*). This model is conceptually related to the notion of peer-to-peer computing, in that all participants are peers who provide resources in the form of tuples, schemas, and query processing and who consume resources by posing queries. In the PDMS the participants need not have any broad consensus on an integrated view of the data; rather, they must instead come to a limited number of local (e.g., pairwise) agreements about how to map data from one schema to another. These *peer mappings* define the semantic relationships between peers. A query is always posed with respect to a specific peer's schema, and query answering is recursively

Principles of Data Integration, DOI: 10.1016/B978-0-12-416044-6.00017-X

reformulated in terms of the schemas of the "neighboring" peers, who can then reformulate it onto their neighbors, and so on — thereby following paths in the network. In this way, local semantic relationships give rise to data sharing among peers farther apart in the network.

The PDMS makes data transformations more local and modular, in that collaborating data sources can create local mappings between pairs (or small sets) of data sources. These mappings can be specialized to the particular collaboration needs the data owners may have. Over time, collaborators can grow these local mappings either by extending the scope of the data they are sharing or by adding new collaborators with data sets. Data sharing proceeds by reformulating queries over these local mappings and by following *mapping paths* to connect far-flung collaborators in the network.

17.1 Peers and Mappings

A PDMS is composed of a set of peers and two kinds of mappings: storage descriptions, which specify how to map source data into the PDMS, and peer mappings, which relate the schemas of different peers. See Figure 17.1 for an example. Each peer in a PDMS has a peer schema that is composed of a set of *peer relations*. The peer schema is the schema

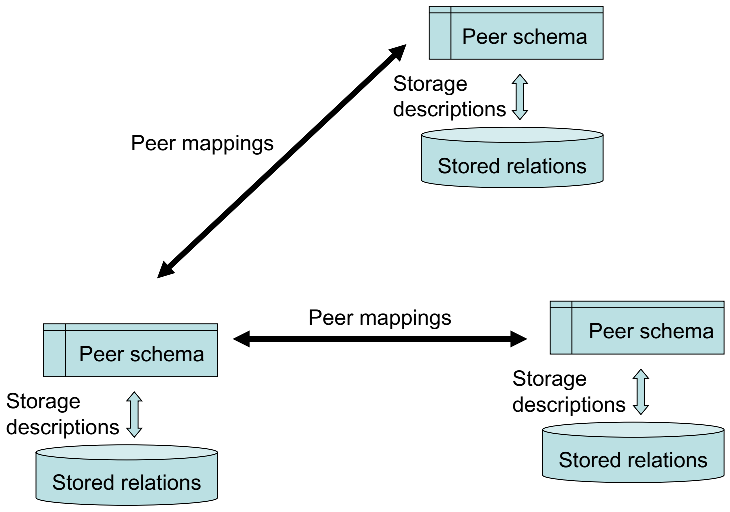


FIGURE 17.1 The structure of a PDMS. Each peer has a peer schema, and peer mappings relate the schemas of multiple peers. The data at each peer are stored in the stored relations, and the storage descriptions are semantic mappings between the stored relations and the peer schema.

that is visible to other peers in the PDMS. Peers typically also have their own data in the form of (instances of) stored relations. The storage descriptions are semantic mappings that relate the schema of the stored relations to the peer schema. The peer mappings are semantic mappings that relate schemas of multiple peers.

Each peer may hide quite a bit of complexity. For example, a peer may be a local data integration system that offers access to multiple sources and exposes the mediated schema to other peers. In some cases, a peer may not have any data and only serve the purpose of mediating among other peers.

A query to a PDMS is posed over the peer schema of one of the peers. In general, each peer relation is "namespaced" or prefixed with a unique identifier corresponding to the peer, making it easy to determine the peer schema. Following this convention, we will name all peer and stored relations in this chapter using a peer-name.relation-name syntax.

A query over a peer schema is reformulated by using the peer mappings and the storage descriptions, and the result of the reformulation is a query that refers only to the stored relations in the PDMS.

Example 17.1

Figure 17.2 illustrates a PDMS for coordinating emergency response. Relations listed near the rectangles are peer relations, and those listed near the cylinders are stored relations. Lines between peers illustrate that there is a peer mapping between those peers (which we describe in more detail later).

On the left side of the figure, we see a set of peers with data from Oregon. Stored relations containing actual data are provided by the hospitals and fire stations (the FH, LH, PFD, and VFD peers). The two fire services peers (PFD and VFD) can share data because there are mappings between their peer relations. Additionally, the FS peer provides a uniform view of all fire services data, but note that it does not have any data of its own. Similarly, H provides a unified view of hospital data. The 911 Dispatch Center (9DC) peer unites all emergency services data.

On the right side of the figure, we see a set of peers with data from the state of Washington. The power of the PDMS becomes evident when an earthquake occurs: the Earthquake Command Center (ECC) and other related peers from Washington can join the Oregonian system. Once we provide peer mappings between the ECC and the existing 911 Dispatch Center, queries over *either* the 9DC or ECC peers will be able to make use of *all* of the source relations.

We now formally describe the language we use for storage descriptions and for peer mappings. The language we describe is based on schema mapping languages from Chapter 3 and combines the features of Global-as-View and Local-as-View. In a sense, the language we describe below extends GAV and LAV from a two-tiered architecture that includes sources and a mediated schema to our more general network of peers that can form an arbitrary graph.

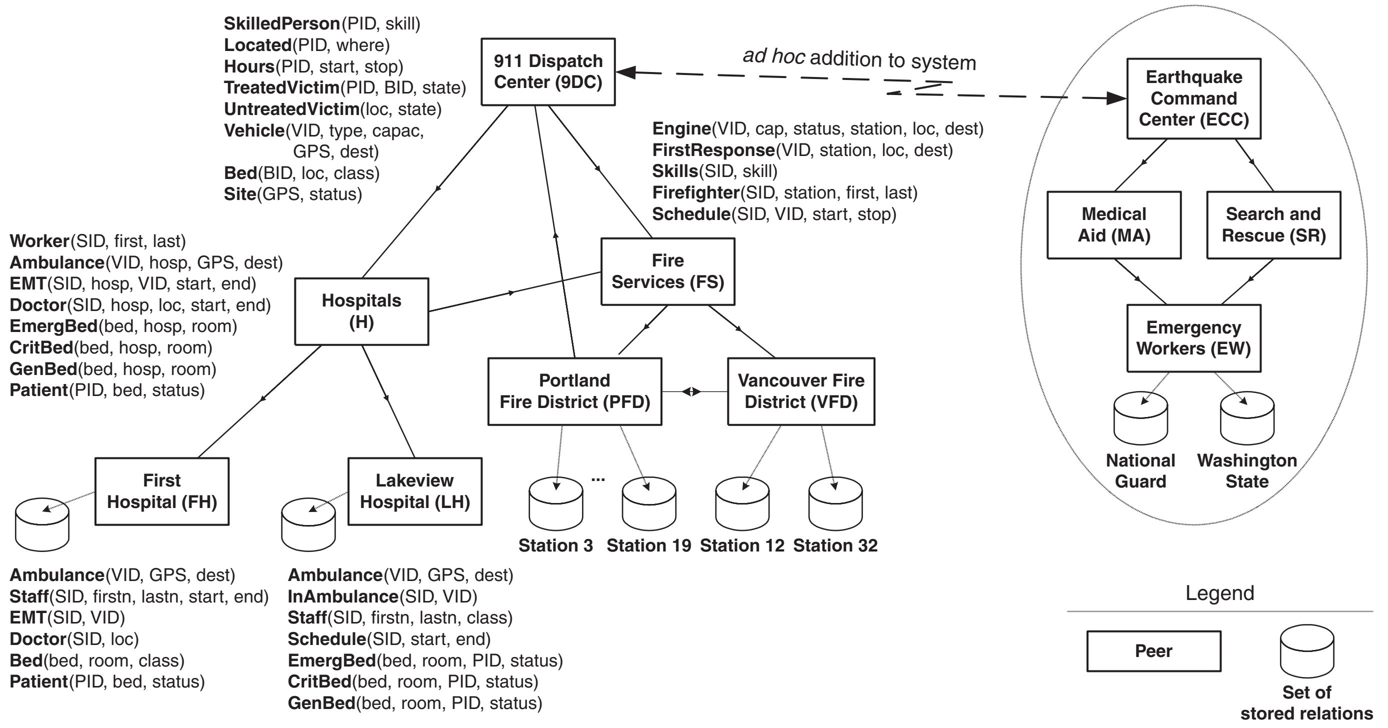

SkilledPerson(PID, skill)
Located(PID, where)
Hours(PID, start, stop)
TreatedVictim(PID, BID, state)
UntreatedVictim(loc, state)
Vehicle(VID, type, capac, GPS, dest)
Bed(BID, loc, class)
Site(GPS, status)
911 Dispatch Center (9DC)
ad hoc addition to system
Earthquake Command Center (ECC)
Engine(VID, cap, status, station, loc, dest)
FirstResponse(VID, station, loc, dest)
Skills(SID, skill)
Firefighter(SID, station, first, last)
Schedule(SID, VID, start, stop)
Medical Aid (MA)
Search and Rescue (SR)
Emergency Workers (EW)
National Guard
Washington State
Worker(SID, first, last)
Ambulance(VID, hosp, GPS, dest)
EMT(SID, hosp, VID, start, end)
Doctor(SID, hosp, loc, start, end)
EmergBed(bed, hosp, room)
CritBed(bed, hosp, room)
GenBed(bed, hosp, room)
Patient(PID, bed, status)
Hospitals (H)
Fire Services (FS)
Portland Fire District (PFD)
Vancouver Fire District (VFD)
First Hospital (FH)
Lakeview Hospital (LH)
Station 3
...
Station 19
Station 12
Station 32
Ambulance(VID, GPS, dest)
Staff(SID, firstn, lastn, start, end)
EMT(SID, VID)
Doctor(SID, loc)
Bed(bed, room, class)
Patient(PID, bed, status)
Ambulance(VID, GPS, dest)
InAmbulance(SID, VID)
Staff(SID, firstn, lastn, class)
Schedule(SID, start, end)
EmergBed(bed, room, PID, status)
CritBed(bed, room, PID, status)
GenBed(bed, room, PID, status)
Legend
Peer
Set of stored relations

FIGURE 17.2

PDMS for coordinating emergency response in the states of Oregon and Washington. Arrows indicate that there is (at least a partial) mapping between the relations of the peers.

STORAGE DESCRIPTIONS

Each peer contains a (possibly empty) set of storage descriptions that specify which data it actually stores. The storage descriptions are semantic mappings that relate the stored relations with the peer relations. Formally, a storage description for a peer A is of the form

$$A.R = Q \text{ or } A.R \subseteq Q$$

R is a stored relation at peer A, and Q is a query over the peer schema of A. We refer to storage descriptions with $=$ as *equality descriptions* and those with $\subseteq$ as *containment descriptions*. An equality description specifies that A stores in relation R the result of the query Q over its schema, and a containment description specifies that R is a subset of the result of the query Q over its schema, thereby expressing an open-world semantics.

Example 17.2

An example storage description might relate stored doctor relations at First Hospital to the peer relations.

$$\begin{aligned} \mathsf{doc(sid, last, loc)} &\subseteq \mathsf{FH.Staff(sid, f, last, s, e), FH.Doctor(sid, loc)} \\ \mathsf{sched(sid, s, e)} &\subseteq \mathsf{FH.Staff(sid, f, last, s, e), FH.Doctor(sid, loc)} \end{aligned}$$

PEER MAPPINGS

Peer mappings semantically relate the schemas of different peers. We describe two types of peer mappings. The first kind is *equality* and *inclusion* mappings that are similar to GLAV descriptions in data integration. The second kind is *definitional mappings* that are essentially datalog programs.

Equality and inclusion peer mappings have the following forms:

$$\begin{aligned} Q_1(\bar{\mathcal{A}}_1) &= Q_2(\bar{\mathcal{A}}_2) && \text{(equality mappings)} \\ Q_1(\bar{\mathcal{A}}_1) &\subseteq Q_2(\bar{\mathcal{A}}_2) && \text{(inclusion mappings)} \end{aligned}$$

where Q_1 and Q_2 are conjunctive queries with the same arity and $\bar{\mathcal{A}}_1$ and $\bar{\mathcal{A}}_2$ are *sets* of peers. Query Q_1-can refer to any of the peer relations in $\bar{\mathcal{A}}_1$ (and the same for $\mathcal{Q}_2$ and $\bar{\mathcal{A}}_2$, respectively).

Intuitively, such a mapping states that evaluating Q_1 over the peers $\bar{\mathcal{A}}_1$ will always produce the same answer (or a subset in the case of inclusions) as evaluating Q_2 over $\bar{\mathcal{A}}_2$. Note that by "evaluating" here we consider data retrieved from other peers as well, not only from the peer being queried.

Definitional mappings are datalog rules whose relations (both head and body) are peer relations. We distinguish definitional mappings for two reasons. First, as we see later, the complexity of answering queries when equality mappings are restricted to being definitional is more attractive than the general case. Second, definitional mappings can easily express disjunction (as we see in the example below), which cannot be done with GLAV mappings.

Example 17.3

We can use the inclusion mappings to specify the Lakeview Hospital peer relations as views over the relations of the peer H that mediates data from multiple hospitals. This is especially convenient in this scenario because the peer H may eventually mediate between many hospitals, and hence LAV-type descriptions are more appropriate.

```
LH.CritBed(bed, hosp, room, PID, status) ⊆
                        H.CritBed(bed, hosp, room), H.Patient(PID, bed, status)
LH.EmergBed(bed, hosp, room, PID, status) ⊆
                        H.EmergBed(bed, hosp, room), H.Patient(PID, bed, status)
LH.GenBed(bed, hosp, room, PID, status) ⊆
                        H.GenBed(bed, hosp, room), H.Patient(PID, bed, status)
```

The 911 Dispatch Center's SkilledPerson peer relation, which mediates hospital and fire services relations, may be expressed using the definitional mappings below. The definition specifies that SkilledPerson in the 9DC is obtained by a union over the H and FS schemas.

```
9DC.SkilledPerson(PID, "Doctor") :- H.Doctor(SID, h, l, s, e)
9DC.SkilledPerson(PID, "EMT") :- H.EMT(SID, h, vid, s, e)
9DC.SkilledPerson(PID, "EMT") :- FS.Schedule(PID, vid),
                               FS.1stResponse(vid, s, l, d), FS.Skills(PID, "medical")
```

17.2 Semantics of Mappings

We now define the semantics of mappings in PDMS in the same spirit that we defined semantics for mappings in virtual data integration systems. In Section 3.2.1 we defined the set of *certain answers* for a query Q. The certain answers are the ones that hold in *every* instance of the mediated schema that is consistent with the data in the sources.

In the context of PDMS, we look at instances of the peer relations in all the peers instead of looking at instances of the mediated schema. To be a consistent instance, the extensions of the peer relations need to satisfy the peer mappings and storage descriptions with respect to the contents of the stored relations.

Formally, we assume that we are given a PDMS, $\mathcal{P}$, and an instance for the stored relations, D. The instance D assigns a set of tuples, $D(R)$, to every stored relation in $R \in \mathcal{P}$. A *data instance*, I, for a PDMS $\mathcal{P}$ is an assignment of a set of tuples $I(R)$ to each relation in $\mathcal{P}$. Note that I assigns a set of tuples to the peer relations as well as the stored relations. We denote by $Q(I)$ the result of computing the query Q over the instance I.

Intuitively, a data instance I is consistent with $\mathcal{P}$ and D if it describes *one* possible state of the world (i.e., an extension for the relations in $\mathcal{P}$) that is allowable given D, the storage descriptions, and the peer mappings.

Definition 17.1 (Consistent Data Instance). *Let I be a data instance for a PDMS, $\mathcal{P}$, and let D be an instance for the stored relations in $\mathcal{P}$. The instance, I, is said to be consistent with $\mathcal{P}$ and D if*

- *for every storage description $r \in \mathcal{P}$, if r is of the form $A.R = Q_1$ ($A.R \subseteq Q_1$), then $D(R) = Q_1(I)$ ($D(R) \subseteq Q_1(I)$).*
- *for every peer description in $r \in \mathcal{P}$,*
  - *if r is of the form $Q_1(\mathcal{A}_1) = Q_2(\mathcal{A}_2)$, then $Q_1(I) = Q_2(I)$,*
  - *if r is of the form $Q_1(\mathcal{A}_1) \subseteq Q_2(\mathcal{A}_2)$, then $Q_1(I) \subseteq Q_2(I)$,*
  - *if r is a definitional description whose head predicate is p, then let $r_1, \ldots, r_m$ be all the definitional mappings with p in their head, and let $I(r_i)$ be the result of evaluating the body of r_i on the instance I. Then $I(p) = I(r_1) \cup \ldots \cup I(r_m)$.* □

We define the certain answers to be the answers that hold in *every* consistent data instance:

Definition 17.2 (Certain Answers in PDMS). *Let Q be a query over a PDMS $\mathcal{P}$, and let D be an instance of the stored relations of $\mathcal{P}$. A tuple $\bar{a}$ is a certain answer to Q w.r.t. D if $\bar{a} \in Q(I)$ for every data instance that is consistent with $\mathcal{P}$ and D.* □

Note that in the last bullet of Definition 17.1 we did not require that the extension of p be the least fixed point model of the datalog rules. However, since we defined certain answers to be those that hold for *every* consistent data instance, we will actually get only the ones that are satisfied in the least fixed point model.

17.3 Complexity of Query Answering in PDMS

We now consider the problem of finding all the certain answers to a query posed on a PDMS. As we will soon see, the complexity of the problem depends on the properties of the mappings. We consider the computational complexity of query answering in terms of the size of the data and the size of the PDMS. One of the important properties in determining the computational complexity is whether the mappings are *cyclic*.

Definition 17.3 (Acyclic Inclusion Peer Mappings). *A set $\mathcal{L}$ of inclusion peer mappings is said to be acyclic if the following directed graph, G, is acyclic.*

The nodes in G are the peer relations mentioned in $\mathcal{L}$. There is an arc in G from the relation R to the relation S if there is a peer mapping in $\mathcal{L}$ of the form $Q_1(\bar{\mathcal{A}}_1) \subseteq Q_2(\bar{\mathcal{A}}_2)$, where R appears in Q_1 and S appears in Q_2. □

The following theorem characterizes two extreme cases of query answering in PDMS:

Theorem 17.1. *Let $\mathcal{P}$ be a PDMS, and let Q be a conjunctive query over $\mathcal{P}$.*

1. *The problem of finding all certain answers to Q is undecidable.*
2. *If $\mathcal{P}$ includes only inclusion storage descriptions and peer mappings, and the peer mappings are acyclic, then Q can can be answered in polynomial time in the size of $\mathcal{P}$ and the stored relations in $\mathcal{P}$.* □

The difference in complexity between the first and second statements shows that the presence of cycles is the culprit for achieving efficient query answering in a PDMS. The next section describes a query-answering algorithm that completes in polynomial time when the PDMS satisfies the conditions of the second statement of Theorem 17.1. The proof of the first statement of the theorem is based on a reduction from the implication problem for functional and inclusion dependencies.

It is interesting to note how Theorem 17.1 is an extension of the results concerning GLAV reformulation (Section 3.2.4). The theorem in Section 3.2.4 shows that LAV and GAV reformulation can be combined by applying one level of LAV followed by one level of GAV. Theorem 17.1 implies that under certain conditions, LAV and GAV can be combined an arbitrary number of times.

17.3.1 Cyclic PDMS

Acyclic PDMSs may be too restrictive for practical applications. One case of particular interest is *data replication*: when one peer maintains a copy of the data stored at a different peer. For example, in the example in Figure 17.2, the Earthquake Command Center may wish to replicate the 911 Dispatch Center's Vehicle table for reliability, using an expression such as

$$\text{ECC.vehicle}(vid, t, c, g, d) = \text{9DC.vehicle}(vid, t, c, g, d)$$

To express replication, we need equality descriptions that, in turn, introduce cycles into the PDMS. While, in general, query answering in the presence of cycles is undecidable, it becomes decidable when equalities are projection-free, as in this example. The following theorem shows an important special case where answering queries on a PDMS is tractable even in the presence of cycles in the peer descriptions.

Theorem 17.2. *Let $\mathcal{P}$ be a PDMS for which all inclusion peer mappings are acyclic, but which may also contain equality peer mappings. Let Q be a conjunctive query over $\mathcal{P}$. Suppose the following two conditions hold:*

- *whenever a storage description or peer mapping in $\mathcal{P}$ is an equality description, it does not contain projections, i.e., all the variables that appear on the left-hand side appear on the right-hand side and vice versa, and*

- *a peer relation that appears in the head of a definitional description does not appear on the right-hand side of any other description.*

Then, finding all certain answers to a Q can be done in polynomial time in the size of the data and the PDMS. □

A further generalization to a specific class of cyclic mappings, called *weakly acyclic* mappings, can also be made using a query reformulation algorithm different from the one presented in this chapter. This is discussed in Section 18.4.

17.3.2 Interpreted Predicates in Peer Mappings

As we saw in Chapter 3, the complexity of answering queries can change if mappings include interpreted predicates. The following theorem characterizes the effect of interpreted predicates on query answering in a PDMS.

Theorem 17.3. *Let $\mathcal{P}$ be a PDMS satisfying the same conditions as the first bullet of Theorem 17.2, and let Q be a conjunctive query over $\mathcal{P}$.*

1. *If interpreted predicates appear only in storage descriptions or in the bodies of definitional mappings, but not in Q, then finding all the certain answers to Q can be done in polynomial time.*
2. *Otherwise, if either the query contains interpreted predicates or interpreted predicates appear in nondefinitional peer mappings, then the problem of deciding whether a tuple $\bar{t}$ is a certain answer to Q is co-NP-complete.* □

17.4 Query Reformulation Algorithm

We now describe an algorithm for query reformulation in PDMSs. The input to the algorithm is a PDMS, $\mathcal{P}$, with its storage descriptions and its peer mappings, and a conjunctive query Q over $\mathcal{P}$. The algorithm outputs a reformulation of Q onto the stored relations in $\mathcal{P}$, i.e., an expression, Q', that refers *only* to stored relations at the peers. To answer Q we need to evaluate Q' over the stored relations, which can be done with the techniques described in Chapter 8.

The algorithm we describe has two properties relating it to our previous discussion on computational complexity. First, the algorithm is sound: evaluating Q' on the stored relations will produce *only* certain answers to Q. Second, the algorithm is complete in the sense that when the peer descriptions are acyclic, then evaluating Q' will produce *all* the certain answers to Q.

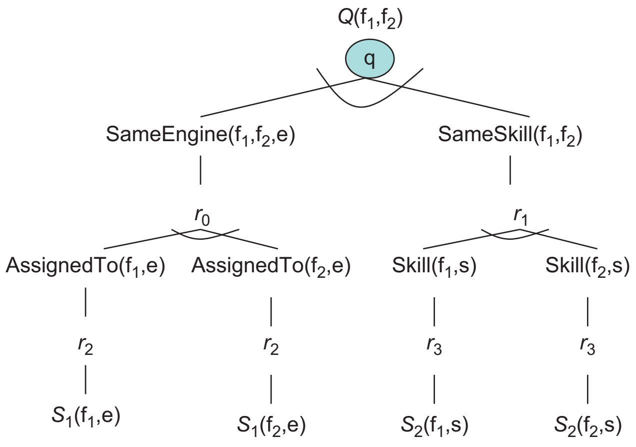
In the input of the algorithm, we assume that every peer mapping in $\mathcal{P}$ of the form $Q_1 = Q_2$ has been replaced by the pair of mappings $Q_1 \subseteq Q_2$ and $Q_1 \supseteq Q_2$. We also assume that left-hand sides of the inclusion dependencies have a single atom. We can easily achieve that by replacing any description of the form $Q_1 \subseteq Q_2$ by the pair $V \subseteq Q_2$ and V :- Q_1, where V is a new predicate name that appears nowhere else in the PDMS.

The key challenge in developing the reformulation algorithm is that we need to interleave reformulation of definitional mappings, which require query unfolding (in the style of GAV reformulation), with reformulation of inclusion mappings, which require algorithms for answering queries using views (in the style of LAV reformulation). Before we discuss how to combine these two kinds of reformulation, let us first briefly consider each one in isolation in the PDMS setting.

Consider a PDMS in which all peer mappings are definitional. In this case, a reformulation algorithm is a simple construction of a *rule-goal tree*. A rule-goal tree has *goal nodes*, labeled with atoms of the peer relations (for the inner nodes) and stored relations (for the leaves), and *rule nodes*, labeled with peer mappings or storage descriptions (see Figure 17.3). The root of the tree is the goal node representing the head of the query Q. The algorithm proceeds by expanding each goal node in the tree with the definitional mappings whose head unifies with the goal node. When none of the nodes of the tree can be expanded with peer mappings, we consider the storage descriptions and create the leaves of the tree.

Example 17.4

Consider the example in Figure 17.3. The root of the tree is the query, and its children are its conjuncts. Each of the conjuncts is expanded with a peer mapping (r_0 and r_1, respectively), resulting in four subgoals. These are then reformulated using the storage descriptions r_2 and r_3. The resulting reformulation is the conjunction of leaves in the tree.



Query:
$Q(f_1,f_2)$:– SameEngine(f_1,f_2,e),
SameSkill(f_1,f_2)

Peer descriptions:
r_0 : SameEngine (f_1,f_2,e) :– AssignedTo(f_1,e)
AssignedTo(f_2,e)
r_1 : SameSkill(f_1,f_2) :– Skill(f_1,s),Skill(f_2,s)

Storage descriptions:
r_2 : $S_1(x,y) \subseteq$ AssignedTo(x,y)
r_3 : $S_2(x,y) \subseteq$ SameSkill(x,y)

FIGURE 17.3 A rule-goal tree for a PDMS with only definitional peer mappings.

Now consider a PDMS whose peer mappings are all inclusions. In this case, we begin with the query subgoals and apply an algorithm for answering queries using views (as in Section 3.2.3). Here, we may take a set of subgoals and replace them by a single subgoal.

Example 17.5

As a simple example, suppose we have the query

$Q(f_1, f_2)$:- SameEngine(f_1, f_2, e), Skill(f_1, s), Skill(f_2, s)

and the peer mapping

SameSkill(f_1, f_2) $\subseteq$ Skill(f_1, s), Skill(f_2, s)

The first step of reformulation would replace two subgoals in the query with a single subgoal of SameSkill.

$Q'(f_1, f_2)$:- SameEngine(f_1, f_2, e), SameSkill(f_1, f_2)

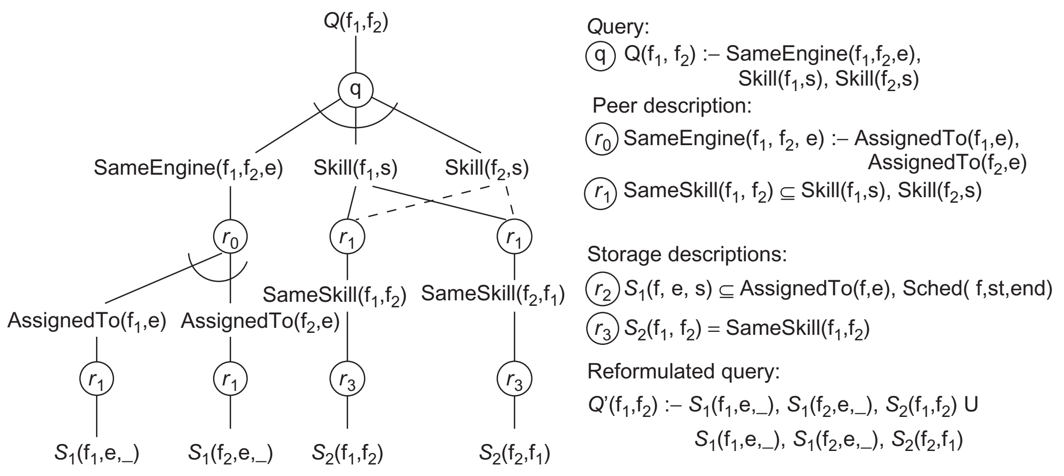
We apply such reformulation steps until we can no longer reformulate any peer relations, and then, as in the previous algorithm, we consider the storage descriptions. Hence, the key difference between these two kinds of reformulation is that while one reformulation replaces a single subgoal with a set of subgoals (definitional reformulation), the other reformulation replaces a set of subgoals with a single subgoal (inclusion reformulation). The algorithm will combine the two types of reformulation by building a rule-goal tree, but in that tree certain nodes will be marked as covering their *uncle* nodes in the tree. We illustrate the algorithm first with an example.

Example 17.6

Figure 17.4 shows an example of the operation of the reformulation algorithm. The query, Q, asks for firefighters with matching skills riding in the same engine. The query Q is expanded into its three subgoals, each of which appears as a goal node. The SameEngine peer relation (indicating which firefighters are assigned to the same engine) is involved in a single definitional peer description, r_0; hence we expand the SameEngine goal node with the rule r_0, and its children are two goal nodes of the AssignedTo peer relation (each specifying an individual firefighter's assignment).

The Skill relation appears on the right-hand side of an inclusion peer description, r_1. Hence, we expand Skill(f1,s) with the rule node r_1, and we create a child node for it that is labeled with the left-hand side of r_1, SameSkill(f_1, f_2). However, in doing so, we realize that the SameSkill atom also covers the subgoal Skill(f_2, s), which is the uncle of the node r_1. In the figure, this covering is annotated by a dashed line, but in the algorithm we will denote the covering with the relation *unc*.

Since the peer relation Skill is involved in a single peer description, we do not need to expand the subgoal Skill(f2,s) any further. Note, however, that we must apply description r_1 a second



$Q(f_1,f_2)$
q
SameEngine(f_1,f_2,e)
Skill(f_1,s)
Skill(f_2,s)
r_0
r_1
r_1
SameSkill(f_1,f_2)
SameSkill(f_2,f_1)
AssignedTo(f_1,e)
AssignedTo(f_2,e)
r_1
r_1
r_3
r_3
$S_1(f_1,e,\_)$
$S_1(f_2,e,\_)$
$S_2(f_1,f_2)$
$S_2(f_2,f_1)$
Query:
q Q(f_1, f_2) :– SameEngine(f_1,f_2,e), Skill(f_1,s), Skill(f_2,s)
Peer description:
r_0 SameEngine(f_1, f_2, e) :– AssignedTo(f_1,e), AssignedTo(f_2,e)
r_1 SameSkill(f_1, f_2) $\subseteq$ Skill(f_1,s), Skill(f_2,s)
Storage descriptions:
r_2 S_1(f, e, s) $\subseteq$ AssignedTo(f,e), Sched(f,st,end)
r_3 S_2(f_1, f_2) = SameSkill(f_1,f_2)
Reformulated query:
$Q'(f_1,f_2)$:– $S_1(f_1,e,\_)$, $S_1(f_2,e,\_)$, $S_2(f_1,f_2)$ U $S_1(f_1,e,\_)$, $S_1(f_2,e,\_)$, $S_2(f_2,f_1)$

FIGURE 17.4 Reformulation rule-goal tree for the Emergency Services domain. Dashed lines represent nodes that are included in the *unc* label.

time with the head variables reversed, since SameSkill may not be symmetric (because it is $\subseteq$ rather than $=$).

At this point, since we cannot reformulate the peer mappings any further, we consider the storage descriptions. We find stored relations for each of the peer relations in the tree (S_1 and S_2) and produce the final reformulation. In this simple example, our reformulation involves only one level of peer mappings, but in general, the tree may be arbitrarily deep.

The first step of the algorithm, shown in Algorithm 15, builds a rule-goal tree starting from the query definition. The algorithm expands goal nodes with either definitional or inclusion mappings until it reaches the stored relations. The labels on the nodes indicate which ancestor and uncle subgoals are covered by each expansion. Note that in order to avoid cycles, the algorithm never uses the same peer mapping more than once in any path down the tree.

When the algorithm considers inclusion mappings, it creates MCDs. Recall from the description of the MiniCon Algorithm (Section 2.4.4) that when we consider a conjunctive query Q and a view V, an MCD is the smallest set of subgoals in Q that will be covered by V if V is used in the rewriting of Q. The other atoms in the MCD are going to be uncles of the rule node in the tree, and we mark those with the *unc* label.[1]

The second step of the algorithm, shown in Algorithm 16, constructs the reformulation from the rule-goal tree T. The solution is a union of conjunctive queries over the stored

[1]We note that in some cases, an MCD may cover cousins or uncles of its father node, not only its own uncles. For brevity of exposition, we ignore this detail in our discussion. However, we note that we do not compromise completeness as a result. In the worst case, we obtain conjunctive rewritings that contain redundant atoms.

Algorithm 15. PDMS-Build-Tree: Build a reformulation rule-goal tree.

```
Input: PDMS $\mathcal{P}$; conjunctive query $Q$ over $\mathcal{P}$. Output: A rule-goal tree.
  // Create rule-goal tree $T$
  Add a goal node as the root of $T$, $root(T)$, labeled with $Q(\bar{X})$
  Add a child $r$ to $root(T)$ and label it with the rule defining the query $Q(\bar{X})$
  for every subgoal, $g$, of $Q$ do
    Create a rule-goal child for $r$ labeled with $g$
  end for
  while goal nodes in $T$ can be expanded do
    Let $n$ be a leaf goal node in $T$ whose label is $l(n) = p(\bar{Y})$, and $p$ is not a stored relation
    if $p$ appears in the head of a definitional description $r$ and $r$ is not an ancestor of $n$ in $T$ then
      // definitional expansion
      Let $r'$ be the result of unifying $p(\bar{Y})$ with the head of $r$
      Create a child rule node $n_r$ with $l(n_r) = r'$
      Create a child goal node for $n_r$ for every subgoal, $g$, of $r'$ with $l(g) = g$
    else if $p$ appears in the right-hand side of an inclusion description or storage description $r$ of
    the form $V \subseteq Q_1$ and $r$ is not an ancestor of $n$ in $T$ then
      // inclusion expansion
      Let $n_1, \ldots, n_m$ be the children of the father node of $n$, and $p_1, \ldots, p_m$ be their corresponding
      labels
      for every MCD that can be created for $p(\bar{Y})$ w.r.t. $p_1, \ldots, p_m$ and $r$ do
        Let $V(\bar{Z})$ be the atom created by the MCD
        Create a child rule node, $n_r$, for $n$ labeled with $r$, and a child goal node, $n_g$, for $n_r$ labeled
        with $V(\bar{Z})$
        Set $unc(n_g)$ to be the set of subgoals covered by the MCD
      end for
    end if
  end while
  return $T$
```

relations. Each of these conjunctive queries represents one way of obtaining answers to the query from the relations stored at peers. Each of them may yield different answers unless we know that some sources are exact replicas of others.

Let us first consider the simple case where only definitional mappings are used. In this case, the reformulation would be the union of conjunctive queries, each with head $Q(\bar{X})$ and a body that can be constructed as follows. Let T' be a subset of the leaves of T constructed by traversing the tree top-down and choosing a *single* child at every goal node and all the children for a given rule node. The body of a conjunctive query is the conjunction of all the nodes in T'.

To accommodate inclusion expansions, we create the conjunctive queries as follows. In creating T's we still choose a single child for every goal node. This time, we do not necessarily have to choose *all* the children of a rule node g. Instead, given a rule node g, we need

Algorithm 16. PDMS-Construct-Reformulation: Create the output PDMS reformulation.

```
Input: a rule-goal tree constructed by Algorithm PDMS-Build-Tree. Output: a set of conjunctive
  queries.
  Let $A = \emptyset$
  Add to $A$ any conjunctive query of the form $Q(\bar{X})$ :- $B$, where $B$ is a conjunction of subgoals that can
  be created as follows
  Initialize $s$ to be a list containing the root node of $T$
  while the list contains nonleaf nodes in $T$ do
    Let $g$ be a nonleaf node in $s$
    Remove $g$ from $s$
    if $g$ is a goal node then
      Choose one child of $g$ and add it to $s$
    else if $g$ is a definitional rule node then
      Add all the children of $g$ to $s$
    else if $g$ is an inclusion rule node then
      Choose a subset of the children of $g, g_1, \ldots, g_l$, where $unc(g_1) \cup \ldots \cup unc(g_l)$ includes all of the
      children of $g$
      Add $g_1, \ldots, g_l$ to $s$
    end if
  end while
  return $A$
```

to choose a subset of the children $g_1, \ldots, g_l$ of g, such that $unc(g_1) \cup \ldots \cup unc(g_l)$ includes all of the children of g.

17.5 Composing Mappings

The reformulation algorithm we described above builds a rule-goal tree that can get very large in a PDMS with many nodes and many peer mappings. To answer queries efficiently in a PDMS, multiple optimizations are needed. For example, we need to intelligently *order* how we expand nodes in the tree, be able to *prune* paths in the tree that yield only redundant results, and apply *adaptive* methods (as described in Chapter 8) to provide results to the user in a streaming fashion.

Another important optimization is to *compose* peer mappings in order to compress longer paths into single edges in the PDMS. Composing mappings saves in reformulation time and also enables us to prune paths that may look good initially but turn out to yield nothing ultimately. In fact, composing schema mappings is an interesting problem in its own right from a logical point of view. It is also important in the context of data exchange (Section 10.2) when data exchange may span longer paths across multiple systems.

Informally, the problem of composing schema mappings is the following. Suppose we have sources A, B, and C and mappings M_{AB} between A and B and M_{BC} between B and C.

Our goal is to find a mapping, M_{AC}, between A and C that will always give the same answers as following the mappings M_{AB} and then M_{BC}.

In the formal definition of composition, recall that a schema mapping, M, between sources A and B defines a binary relation, M^R, between the instances of A and the instances of B. The relation M^R contains pairs (a, b), where $a \in I(A)$ and $b \in I(b)$ and where a and b are consistent with each other under the mapping M (Section 3.2.1). The formal definition requires that M_{AC} be the join of the relations M_{AB} and M_{BC}.

Definition 17.4 (Mapping Composition). *Let A, B, and C be data sources, and let M_{AB} be a mapping between A and B and M_{BC} be a mapping between B and C. A mapping M_{AC} is said to be a composition of M_{AB} and M_{BC} if*

$$M^R_{AC} = M^R_{AB} \times M^R_{BC}$$

□

Finding the composition of mappings turns out to be rather tricky. The following examples illustrate why.

Example 17.7

Consider the following schema mappings, where each source has a single relation A, B, and C, respectively.

$$M_{AB} \quad \mathsf{A(x,z), A(z,y)} \subseteq \mathsf{B(x,y)}$$

$$M_{BC} \quad \mathsf{B(x,z), B(z,y)} \subseteq \mathsf{C(x,y)}$$

Each relation can be thought of as storing the set of edges in a directed graph. The mapping M_{AB} states that paths of length two in A are a subset of edges in B, and M_{BC} says that paths of length two in B are a subset of the edges in C.

It is easy to verify that the following mapping represents the composition of M_{AB} and M_{BC}. The mapping states that paths of length four in A are a subset of edges in C.

$$M \qquad \mathsf{A(x,z_1), (z_1,z_2), A(z_2,z_3), A(z_3,y)} \subseteq \mathsf{C(x,y)}$$

In Example 17.7 we were able to express the composition with a mapping in GLAV, i.e., in the same language as the mappings being composed. A natural question is whether a language for expressing mappings is closed under composition. In particular, is it always possible to express the composition of GLAV mappings in GLAV?

The answer to this question depends in subtle ways on the form of the mappings being composed. In Example 17.7, the mappings did not introduce any new existential variables. Specifically, all the variables that appeared on the right-hand side of the mappings also appear on the left-hand side. As it turns out, this condition is crucial in order to express the composition of GLAV mappings in GLAV. We state the following theorem without proof (see the bibliographic notes for more detail).

In the theorem, we say that a formula in a GLAV schema mapping is a *full* formula if the right-hand side of the mapping does not have variables that do not appear on the left-hand side.[2]

Theorem 17.4. *Let M_{AB} and M_{BC} be mappings, each specified by a finite set of GLAV mappings. If all the formulas in M_{AB} are full formulas, then the composition of M_{AB} and M_{BC} can be represented by a finite set of GLAV formulas. Deciding whether a GLAV formula f is in the composition of M_{AB} and M_{BC} can be done in polynomial time.* □

In fact, as we show below, Theorem 17.4 describes a rather tight condition on when composition can be computed efficiently. The following example shows that a composition of two finite GLAV mappings may require an *infinite* number of formulas.

Example 17.8

In this example, source B has two relations, R and G, representing red and green edges, respectively. Sources A and C also have two relations each. Consider the following mappings:

$$\begin{aligned} M_{AB} \quad & \{\mathsf{A_{rg}}(x,y) \subseteq \mathsf{R}(x,x_1), \mathsf{G}(x_1,y) \\ & \mathsf{A_{gg}}(x,y) \subseteq \mathsf{G}(x,x_1), \mathsf{G}(x_1,y)\} \\ M_{BC} \quad & \{\mathsf{R}(x,x_1), \mathsf{G}(x_1,x_2), \mathsf{G}(x_2,y) \subseteq \mathsf{C_{rgg}}(x,y) \\ & \mathsf{G}(x,x_1), \mathsf{G}(x_1,y) \subseteq \mathsf{C_{gg}}(x,y)\} \end{aligned}$$

The relation $\mathsf{A_{rg}}$ is a subset of the node pairs in B with red-green paths. The other relations $\mathsf{A_{gg}}, \mathsf{B_{rgg}}$, and $\mathsf{C_{gg}}$ can be described similarly. Observe that the following infinite sequence of formulas are all in the composition of M_{AB} and M_{BC}:

$$\mathsf{A_{gg}}(x,y) \subseteq \mathsf{C_{gg}}(x,y) \tag{17.1}$$

$$\mathsf{A_{rg}}(x,x_1), \mathsf{A_{gg}}(x_1,x_2) \subseteq \mathsf{C_{rgg}}(x,y_1) \tag{17.2}$$

$$\mathsf{A_{rg}}(x,x_1), \mathsf{A_{gg}}(x_1,x_2), \mathsf{A_{gg}}(x_2,x_3) \subseteq \mathsf{C_{rgg}}(x,y_1), \mathsf{C_{gg}}(y_1,y_2) \tag{17.3}$$

$$\ldots$$

$$\mathsf{A_{rg}}(x,x_1), \mathsf{A_{gg}}(x_1,x_2), \ldots, \mathsf{A_{gg}}(x_n,x_{n+1}) \subseteq \mathsf{C_{rgg}}(x,y_1), \mathsf{C_{gg}}(y_1,y_2), \ldots, \mathsf{C_{gg}}(y_{n-1},y_n) \tag{17.4}$$

[2]As noted earlier, GLAV formulas are similar to tuple-generating dependencies. In that terminology, a full formula is a full dependency.

The sequence is infinite. Each equation in the infinite sequence says that a path comprising one red (R) edge followed by $2n+1$ green (G) edges (the query over A) is contained within a path comprising a red edge followed by $2n+2$ green edges (the query over C). None of them can be expressed in terms of the others. For example, Formula 17.3 is not implied by Formulas 17.1 and 17.2.

The following theorem formalizes the intuition of Example 17.8 and shows that whenever M_{AB} may have existential variables on the right-hand side of the formulas, the composition may not be definable by a finite set of GLAV formulas.

Theorem 17.5. *There exist mappings, M_{AB} and M_{BC}, where M_{AB} is a finite set of GLAV formulas and M_{BC} is a finite set of full GLAV formulas, such that the composition of M_{AB} and M_{BC} is not definable by a finite set of GLAV formulas but is definable by an infinite set of GLAV formulas.* □

In practice, it is possible to compute efficiently the compositions of peer mappings. However, it is important that compositions be used judiciously in query answering, because they can also cause the search space of possible plans to grow further.

17.6 Peer Data Management with Looser Mappings

So far we have discussed peer data management as a generalization of data integration systems. The system is based on peer mappings that are specified between pairs (or small sets) of peers, rather than mappings between data sources and a mediated schema. As described, the system already provides quite a bit more flexibility than virtual data integration systems. However, to obtain these benefits, the collaborators must still create schema mappings, which may be a labor-intensive task.

In this section we describe two looser models for collaboration among multiple parties that do not require full peer mappings. Of course, with less precise specification of relationships between peers, the queries that can be specified and the accuracy of the results obtained will be diminished. However, in some applications, getting *some* data from other peers may be much better than nothing.

In the first method we describe, relationships between peers are *inferred* via approximate techniques (in the spirit of Chapter 5). In the second method, mappings are specified at the data level only with *mapping tables*. Of course, in practice, we may have a system that includes precise mappings in some places and uses a combination of the techniques we describe below in others.

17.6.1 Similarity-Based Mappings

Consider a PDMS where each peer, n, has a *neighborhood* of peers that are the peers with which n can communicate. However, we do not necessarily have peer mappings between

Algorithm 17. PDMS-Similarity: Compute PDMS similarity. The function *sim* computes similarity between pairs of relation names or attribute names. The parameter τ is a similarity threshold.

```
Input: PDMS $\mathcal{P}$; node $n$ from $\mathcal{P}$; query $Q(\bar{X})$ on $\mathcal{P}$; (initially empty) accumulator for result $\mathcal{R}$.
Output: $\mathcal{R}$ contains final result.
Let $Q(\bar{X})$ be of the form $p(\bar{X})$, where $p$ is a relation in the schema of $n$, and $A_1,\ldots,A_l$ are the attributes of $p$
Evaluate $Q(\bar{X})$ on the stored relations of $n$; add the results to $\mathcal{R}$
for every neighbor $n_1$ of $n$ do
    Let $sim(r,p)$ denote be the similarity between the relation $r \in n_1$ and $p$
    Let $sim(A_i,B)$ be the similarity between an attribute $A_i$ and an attribute $B$ in $n_1$
    for every atom $r(B_1,\ldots,B_j)$ where $sim(r,p), sim(B_1,A_1),\ldots,sim(B_k,A_k) \geq \tau$ do
        Call PDMS-Similarity($\mathcal{P}$, $r(B_1,\ldots,B_j), n_1$, $\mathcal{R}$)
    end for
end for
```

n and its neighbors, and therefore reformulation and query answering will proceed by approximate mappings between schema elements of n and of its neighbors.

Specifically, suppose a user poses an atomic query $p(\bar{X})$ on n, and let n_1 be a neighbor of n. We can compute a similarity measure between the relation names of n and the relation names of n' and similarity values between the attribute names of p and the attribute names in n'. If we find a relation $p' \in n'$ that has high similarity to p, and we can find attributes $B_1,\ldots,B_k$ of p' that are similar to $A_1,\ldots,A_j$ of p, respectively, then we can formulate the query $p(\bar{X})$ as a query of the form $p'(\bar{Y})$. Note that we do not need to cover all the attributes of p in order for this to be useful. If p' does not have attributes corresponding to each of the attributes of p, then the reformulation ignores the missing attributes. Algorithm 17 describes the algorithm based on such a reformulation strategy.

We can continue this process recursively with neighbors of n_1 to obtain more answers. Note that since we may not always find all the attributes of the reformulated query, as the path gets longer the resulting tuples will be narrower, and possibly empty.

17.6.2 Mapping Tables

The second mechanism involves describing only correspondences at the data level. As described in Chapter 7, data-level mappings are often crucial to data integration because data sources often refer to the same real-world object in different ways, and in order to integrate data we must know which values match up.

In this context we can make broader use of mapping tables. In the simplest case, mapping tables can specify correspondences between different references to the same real-world objects. Going one step further, mapping tables can be used to describe correspondences between different lexicons. For example, suppose we are given two airlines' flight tables, shown in Figure 17.5. The mapping table in Figure 17.6 describes the mapping that is necessary for the airlines to code-share flights. We can go even further and create mapping tables that describe correspondences between elements in completely *different*

(a)

UnitedFlights

flightNum	origin	destination	departure	arrival
UA292	JFK	SFO	8AM	11:30AM
UA200	EWR	LAX	9AM	12:30PM
UA404	YYZ	SFO		

(b)

AirCanadaFlights

flightID	origin	dest	depTime	arrivalTime	aircraft
AC543	JFK	SFO	8AM	11:30AM	Boeing 747
AC505	EWR	LAX	9AM	12:30PM	Airbus 320
UA404	YYZ	SFO	8AM	1PM	Airbus 320

FIGURE 17.5 Tables of flights for two different peers (United Airlines and Air Canada).

FlightCodeMapping

UnitedFlightCode	AirCanadaFlightCode
UA292	AC543
UA200	AC505
UA404	AC303
v - {UA001 – UA500}	v - {CA001 – CA500}

FIGURE 17.6 A mapping table for the tables shown in Figure 17.5.

domains. For example, when sharing biological data, we can create a mapping table that relates gene IDs to related proteins.

Intuitively, a mapping table represents a possible join between tables in different peers. By following such joins, a user at one peer can obtain data from other peers without the need for schema mappings. For example, the table in Figure 17.5(a) shows flights for United Airlines and Figure 17.5(b) shows flights for Air Canada. Note that United Airlines does not store the aircraft for the flight (or perhaps it stores the aircraft for its own flights in a different table). A user can query for all the information for flight UA404 at the United Airlines peer. With a mapping table, the system will reformulate the query on the Air Canada peer, with flight ID 303. The Air Canada peer does store the aircraft.

More formally, we define a mapping table M between two relations R_1 and R_2. We assume the table M has two attributes A_1 and A_2, where A_1 is assumed to be an attribute of R_1 and A_2 is assumed to be an attribute of R_2. In principle, M can have any number of attributes, mapping projections of tuples in R_1 with projections of tuples in R_2, but we consider the simple case here for brevity. We denote the domain of A_1 by D_1 and the domain of A_2 by D_2. We also assume there is an alphabet of variables, $\mathcal{V}$, that is disjoint from $D_1 \cup D_2$.

Definition 17.5 (Mapping Tables). *A mapping table, M, between two relations R_1 and R_2 is a table with two attributes A_1 and A_2, where the values in columns can be as follows:*

- *values in D_1 (in the A_1 column) or values in D_2 (in the A_2 column), or*
- *a variable $v \in \mathcal{V}$, or*

- *an expression of the form $v - C$, where C is a finite set of values D_1 (in A_1) or C is a finite set of values for D_2 (in A_2).*

We assume that every variable appears in at most one tuple in M. □

Example 17.9

Consider the mapping table in Figure 17.6. The first three rows specify correspondences between specific flights of United Airlines and Air Canada. The fourth row is a succinct way of specifying that for flights whose number is greater than 500, the codes of the two airlines are identical. In fact, the tuple (x,x) can be used to specify that any time the same value appears in both tables, they can map to each other.

We now consider the semantics of mapping tables, which is based on valuations. A valuation for a tuple in a mapping table is the result of substituting values for the variables. We can substitute a value in D_1 for a variable that appears in the left column and a value from D_2 for a variable that appears in the right column. Of course, if the tuple does not have any variables, then the tuple is unchanged.

The formal definition of a mapping table specifies a subset of the Cartesian product of R_1 and R_2 that is consistent with the mapping.

Definition 17.6 (Mapping Table Semantics). *Let $M(A_1,A_2)$ be a mapping table between R_1 and R_2. A tuple t in $R_1 \times R_2$ is consistent with M if and only if there is a tuple, t', in M and a variable substitution, τ, such that $\Pi_{A_1,A_2}(t) = \tau(t')$.* □

Mapping tables are interesting subjects of study in themselves. As with schema mappings, an interesting question that arises is whether they can be composed to discover new mapping tables, and whether a set of mapping tables is consistent with each other. We comment on these properties in the bibliographic notes.

Bibliographic Notes

The emergence of of peer-to-peer file sharing systems inspired the data management research community to consider P2P architectures for data sharing. Some of the systems that were built in this spirit were [74, 283, 309, 346, 433, 459, 544, 585]. Connections have also been made between PDMSs and architectures for the Semantic Web [2, 10, 287].

The language for describing peer mappings in PDMSs and the query reformulation algorithm are taken from [288], describing the Piazza PDMS. Optimization algorithms, including methods for ignoring redundant paths in the reformulation and the effect of mapping composition, are described in [288, 541].

The problem of composing schema mappings was initially introduced by Madhavan and Halevy [408]. In that paper, composition was defined to hold w.r.t. a class of queries and several restricted cases where composition can be computed were identified. They also showed that compositions of GLAV mappings may require an infinite number of formulas. The definition we presented is based on the work of Fagin et al. [218], and Theorems 17.4 and 17.5 are also taken from there. Interestingly, Fagin et al. also showed that composition of GLAV formulas can always be expressed as a finite number of *second-order* tuple-generating dependencies, where relation names are also variables. Nash et al. [452] present more complexity results regarding composition, and in [75] Bernstein et al. describe a practical algorithm for implementing composition. A variant on the notion of composition is to merge many partial specifications of mappings, as is done in the work on the MapMerge operator [22].

The looser architectures we described in Section 17.6 are from [2, 346, 459]. In particular, the similarity-based reformulation is described in more detail in [459]. Mapping tables were introduced in [346], where the Hyperion PDMS is described. That paper considers several other interesting details concerning mapping tables. First, the paper considers two semantics for mapping tables – the open-world semantics and the closed-world semantics. In the closed-world semantics, the presence of a pair (X, Y) in the mapping table implies that X (or Y) cannot be associated with any other value, while in the open-world assumption they can. The semantics also differ on the meaning of a value *not* appearing in the mapping table. Second, the paper considers the problem of composing mapping tables and determining whether a set of mapping tables is consistent and shows that in general, the problem is NP-complete in the size of the mapping tables. Another related idea is that of update coordination across parties with different levels of abstraction, studied in [368]. One means of propagating such updates is via triggers [341].

The Orchestra system [268, 544] takes PDMSs one step further and focuses on *collaboration* among multiple data owners. To facilitate collaboration, users need to be able to manage their data independently of others, propagate updates as necessary, track the provenance of the data, and create and apply *trust* conditions on which data they would like to import locally. We touch on some of these issues in more detail in Chapter 18.