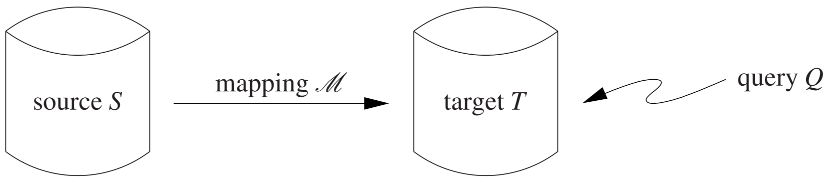
PART ONE
GETTING STARTED

1
Data exchange by example

Data exchange is the problem of finding an instance of a target schema, given an instance of a source schema and a specification of the relationship between the source and the target. Such a target instance should correctly represent information from the source instance under the constraints imposed by the target schema, and should allow one to evaluate queries on the target instance in a way that is semantically consistent with the source data.

Data exchange is an old problem that re-emerged as an active research topic recently due to the increased need for exchange of data in various formats, often in e-business applications.

The general setting of data exchange is this:



We have fixed source and target schemas, an instance S of the source schema, and a mapping $\mathscr{M}$ that specifies the relationship between the source and the target schemas. The goal is to construct an instance T of the target schema, based on the source and the mapping, and answer queries against the target data in a way consistent with the source data.

The goal of this introductory chapter is to make precise some of the key notions of data exchange: schema mappings, solutions, source-to-target dependencies, and certain answers. We do it by means of an example we present in the next section.

1.1 A data exchange example

Suppose we want to create a database containing three relations:

- `ROUTES(flight#,source,destination)`

  This relation has information about routes served by several airlines: it has a `flight#` attribute (e.g., AF406 or KLM1276), as well as `source` and `destination` attributes (e.g., Paris and Santiago for AF406).

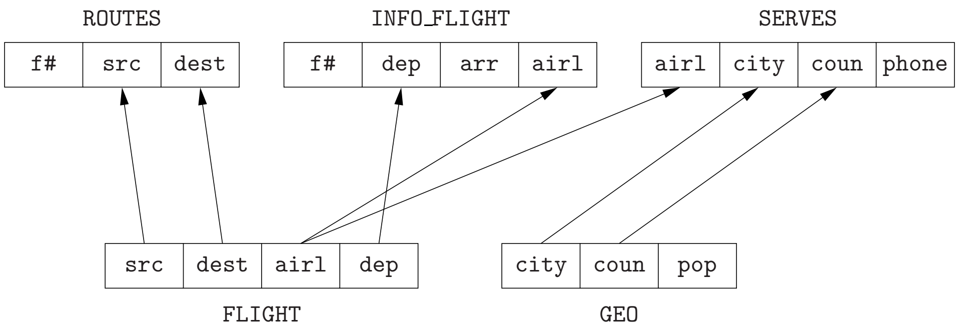


Figure 1.1 Schema mapping: a simple graphical representation

- INFO_FLIGHT(flight#,departure_time,arrival_time,airline)
  This relation provides additional information about the flight: departure and arrival times, as well as the name of an airline.
- SERVES(airline,city,country,phone)
  This relation has information about cities served by airlines: for example, it may have a tuple (*AirFrance*, *Santiago*, *Chile*, *5550000*), indicating that Air France serves Santiago, Chile, and its office there can be reached at 555-0000.

We do not start from scratch: there is a source database available from which we can transfer information. This source database has two relations:

- FLIGHT(source,destination,airline,departure)
  This relation contains information about flights, although not all the information needed in the target. We only have source, destination, and airline (but no flight number), and departure time (but no arrival time).
- GEO(city,country,population)
  This relation has some basic geographical information: cities, countries where they are located, and their population.

As the first step of moving the data from the source database into our target, we have to specify a *schema mapping*, a set of relationships between the two schemas. We can start with a simple graphical representation of such a mapping shown in Figure 1.1. The arrows in such a graphical representation show the relationship between attributes in different schemas.

But simple connections between attributes are not enough. For example, when we create records in ROUTES and INFO_FLIGHT based on a record in FLIGHT, we need to ensure that the values of the flight# attribute (abbreviated as f# in the figure) are the same. This is indicated by a curved line connecting these attributes. Likewise, when we populate table SERVES, we only want to include cities which appear in table FLIGHT – this is indicated by the line connecting attributes in tables GEO and FLIGHT.

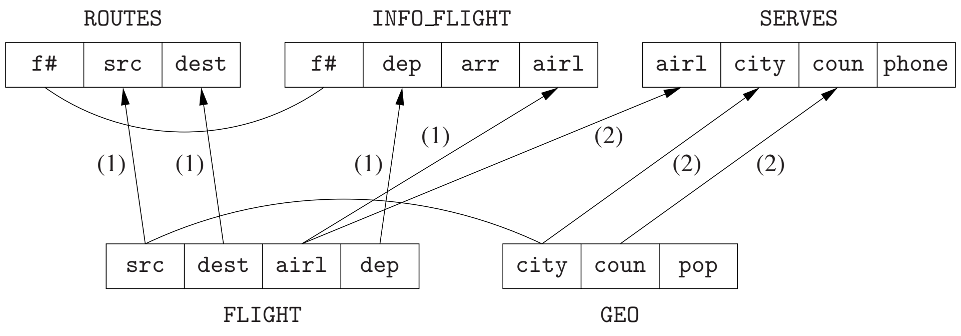


Figure 1.2 Schema mapping: a proper graphical representation

Furthermore, there are several *rules* in a mapping that help us populate the target database. In this example, we can distinguish two rules. One uses table `FLIGHT` to populate `ROUTES` and `INFO_FLIGHT`, and the other uses both `FLIGHT` and `GEO` to populate `SERVES`. So in addition we annotate arrows with names or numbers of rules that they are used in. Such a revised representation is shown in Figure 1.2.

While it might be easy for someone understanding source and target schemas to produce a graphical representation of the mapping, we need to translate it into a formal specification. Let us look at the first rule which says:

whenever we have a tuple `(src,dest,airl,dep)` *in relation* `FLIGHT`*, we must have a tuple in* `ROUTES` *that has* `src` *and* `dest` *as the values of the second and the third attributes, and a tuple in* `INFO_FLIGHT` *that has* `dep` *and* `airl` *as the second and the fourth attributes.*

Formally, this can be written as:

$$\begin{array}{l}\texttt{FLIGHT(src,dest,airl,dep)} \longrightarrow \\ \qquad \texttt{ROUTES(\_,src,dest), INFO\_FLIGHT(\_,dep,\_,airl)}.\end{array}$$

This is not fully satisfactory: indeed, as we lose information that the flight numbers must be the same; hence, we need to explicitly mention the names of all the variables, and produce the following rule:

$$\begin{array}{l}\texttt{FLIGHT(src,dest,airl,dep)} \longrightarrow \\ \qquad \texttt{ROUTES(f\#,src,dest), INFO\_FLIGHT(f\#,dep,arr,airl)}.\end{array}$$

What is the meaning of such a rule? In particular, what are those variables that appear in the target specification without being mentioned in the source part? What the mapping says is that values for these variables must *exist* in the target, in other words, the following must be satisfied:

$$\begin{array}{l}\texttt{FLIGHT(src,dest,airl,dep)} \longrightarrow \\ \qquad \exists \texttt{f\#}\, \exists \texttt{arr} \Big(\texttt{ROUTES(f\#,src,dest)} \\ \qquad\qquad\qquad \wedge\ \texttt{INFO\_FLIGHT(f\#,dep,arr,airl)} \Big).\end{array}$$

To complete the description of the rule, we need to clarify the role of variables `src`, `dest`, `airl` and `dep`. The meaning of the rule is that *for every* tuple `(src,dest,airl,dep)` in table `FLIGHT` we have to create tuples in relations `ROUTES` and `INFO_FLIGHT` of the target schema. Hence, finally, the meaning of the first rule is:

$$\begin{aligned}\forall \mathtt{src}\, \forall \mathtt{dest}\, \forall \mathtt{airl}\, \forall \mathtt{dep} \Big(& \mathtt{FLIGHT(src,dest,airl,dep)} \longrightarrow \\ & \exists \mathtt{f\#}\, \exists \mathtt{arr} \big(\mathtt{ROUTES(f\#,src,dest)} \\ & \qquad \wedge \mathtt{INFO\_FLIGHT(f\#,dep,arr,airl)} \big) \Big).\end{aligned}$$

Note that this is a query written in relational calculus, without free variables. In other words, it is a sentence of first-order logic, over the vocabulary including both source and target relations. The meaning of this sentence is as follows: given a source S, a target instance we construct is such that together, S and T satisfy this sentence.

We now move to the second rule. Unlike the first, it looks at two tuples in the source: `(src,dest,airl,dep)` in `FLIGHT` and `(city,country,popul)` in `GEO`. If they satisfy the join condition `city=scr`, then a tuple needs to be inserted in the target relation `SERVES`:

$$\begin{aligned}&\mathtt{FLIGHT(src,dest,airl,dep)}, \mathtt{GEO(city,country,popul)}, \mathtt{city{=}src} \\ &\quad \longrightarrow \mathtt{SERVES(airl,city,country,phone)}.\end{aligned}$$

As with the first rule, the actual meaning of this rule is obtained by explicitly quantifying the variables involved:

$$\begin{aligned}&\forall \mathtt{city}\, \forall \mathtt{dest}\, \forall \mathtt{airl}\, \forall \mathtt{dep}\, \forall \mathtt{country}\, \forall \mathtt{popul} \Big(\\ &\quad \mathtt{FLIGHT(city,dest,airl,dep)} \wedge \mathtt{GEO(city,country,popul)} \longrightarrow \\ &\qquad\qquad \exists \mathtt{phone}\, \mathtt{SERVES(airl,city,country,phone)} \Big).\end{aligned}$$

We can also have a similar rule in which the destination city is moved in the `SERVES` table in the target:

$$\begin{aligned}&\forall \mathtt{city}\, \forall \mathtt{dest}\, \forall \mathtt{airl}\, \forall \mathtt{dep}\, \forall \mathtt{country}\, \forall \mathtt{popul} \Big(\\ &\quad \mathtt{FLIGHT(src,city,airl,dep)} \wedge \mathtt{GEO(city,country,popul)} \longrightarrow \\ &\qquad\qquad \exists \mathtt{phone}\ \mathtt{SERVES(airl,city,country,phone)} \Big).\end{aligned}$$

These rules together form what we call a *schema mapping*: a collection of rules that specify the relationship between the source and the target. When we write them, we actually often omit universal quantifiers $\forall$, as they can be reconstructed by the following rule:

- every variable mentioned in one of the source relations is quantified universally.

With these conventions, we arrive at the schema mapping $\mathscr{M}$, shown in Figure 1.3.

Now, what does it mean to have a target instance, given a source instance and a mapping? Since mappings are logical sentences, we want target instances to satisfy these sentences, with respect to the source. More precisely, note that mappings viewed as logical sentences

$$
\begin{array}{ll}
(1) & \texttt{FLIGHT(src,dest,airl,dep)} \longrightarrow \\
 & \qquad \exists\texttt{f\#}\ \exists\texttt{arr}\ \big(\ \ \texttt{ROUTES(f\#,src,dest)} \\
 & \qquad\qquad\qquad\quad \wedge\ \texttt{INFO\_FLIGHT(f\#,dep,arr,airl)}\big) \\
 & \\
(2) & \texttt{FLIGHT(city,dest,airl,dep)} \wedge \texttt{GEO(city,country,popul)} \\
 & \qquad \longrightarrow\ \exists\texttt{phone SERVES(airl,city,country,phone)} \\
 & \\
(3) & \texttt{FLIGHT(src,city,airl,dep)} \wedge \texttt{GEO(city,country,popul)} \\
 & \qquad \longrightarrow\ \exists\texttt{phone SERVES(airl,city,country,phone)}
\end{array}
$$

Figure 1.3 A schema mapping

mention both source and target schemas. So possible target instances T for a given source S must satisfy the following condition:

For each condition φ of the mapping $\mathscr{M}$, the pair (S,T) satisfies φ.

We call such instances T *solutions for S under $\mathscr{M}$*. Look, for example, at our mapping $\mathscr{M}$, and assume that the source S has a tuple $(\textit{Paris, Santiago, AirFrance}, 2320)$ in FLIGHT. Then every solution T for S under $\mathscr{M}$ must have tuples

$$
\begin{array}{rll}
(x, \textit{Paris, Santiago}) & \text{in} & \texttt{ROUTES} \text{ and} \\
(x, 2320, y, \textit{AirFrance}) & \text{in} & \texttt{INFO\_FLIGHT}
\end{array}
$$

for some values x and y, interpreted as flight number and arrival time. The mapping says nothing about these values: they may be real values (constants), e.g., $(406, \textit{Paris, Santiago})$, or *nulls*, indicating that we lack this information at present. We shall normally use the symbol $\bot$ to denote nulls, so a common way to populate the target would be with tuples $(\bot, \textit{Paris, Santiago})$ and $(\bot, 2320, \bot', \textit{AirFrance})$. Note that the first attributes of both tuples, while being unknown, are nonetheless the same. This situation is referred to as having *marked nulls*, or *naïve* nulls, as they are used in naïve tables, studied extensively in connection with incomplete information in relational databases. At the same time, we know nothing about the other null $\bot'$ used: nothing prevents it from being different from $\bot$ but nothing tells us that it should be.

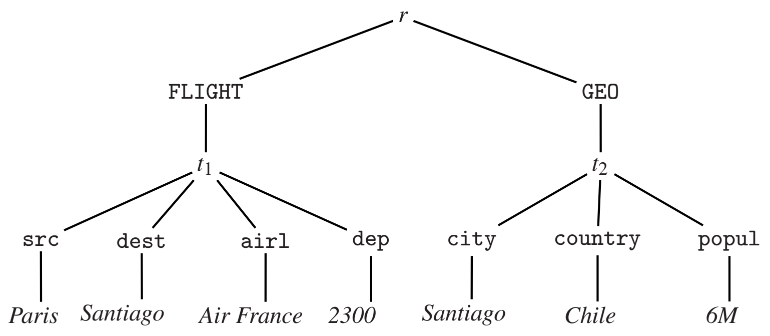
Note that already this simple example leads to a crucial observation that makes the data exchange problem interesting: *solutions are not unique*. In fact, there could be infinitely many solutions: we can use different marked nulls, or can instantiate them with different values.

If solutions are not unique, how can we answer queries? Consider, for example, a Boolean (yes/no) query *"Is there a flight from Paris to Santiago that arrives before 10am?"*. The answer to this query has to be "no", even though in some solutions we shall have tuples with arrival time before 10am. However, in others, in particular in the one with null values, the comparison with 10am will not evaluate to true, and thus we have to return "no" as the answer.

On the other hand, the answer to the query *"Is there a flight from Paris to Santiago?"* is "yes", as the tuple including Paris and Santiago will be in every solution. Intuitively, what we want to do in query answering in data exchange is to return answers that will be true in every solution. These are called *certain answers*; we shall define them formally shortly.

XML data exchange

Before outlining the key tasks in data exchange, we briefly look at the XML representation of the above problem. XML is a flexible data format for storing and exchanging data on the Web. XML documents are essentially trees that can represent data organized in a way more complex than the usual relational databases. But each relational database can be encoded as an XML document; a portion of our example database, representing information about the Paris–Santiago flight and information about Santiago, is shown in the picture below.

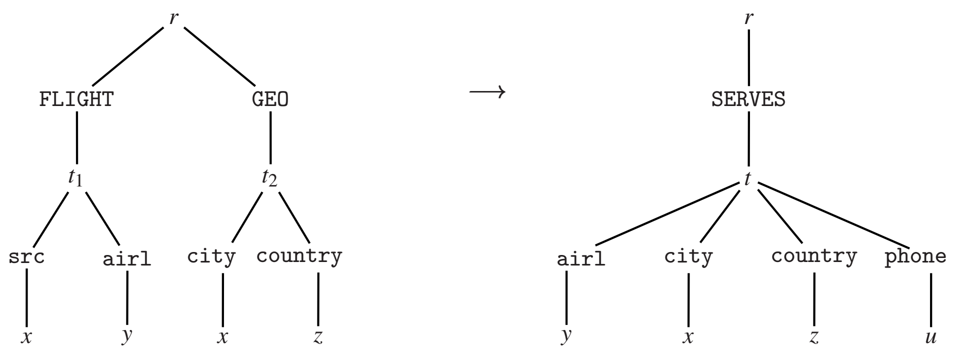


The tree has a root r with two children, corresponding to relations FLIGHT and GEO. Each of these has several children, labeled t_1 and t_2, respectively, corresponding to tuples in the relations. We show one tuple in each relation in the example. Each t_1-node has four children that correspond to the attributes of FLIGHT and each t_2-node has three children, with attributes of GEO. Finally, each of the attribute nodes has a child holding the value of the attribute.

To reformulate a rule in a schema mapping in this language, we show how portions of trees are restructured. Consider, for example, the rule

$$\begin{aligned}&\texttt{FLIGHT(city,dest,airl,dep)} \wedge \texttt{GEO(city,country,popul)} \longrightarrow \\ &\qquad\qquad \exists \texttt{phone}\ \texttt{SERVES(airl,city,country,phone)}\end{aligned}$$

We restate it in the XML context as follows:



That is, if we have tuples in `FLIGHT` and `GEO` that agree on the values of the `source` and `city` attributes, we grab the values of the `airline` and `country` attributes, invent a new value u for `phone` and create a tuple in relation `SERVES`.

The rules of XML schema mappings are thus represented via *tree patterns*. Essentially, they say that if a certain pattern occurs in a source document, some other pattern, obtained by its restructuring, must occur in the target.

This view of XML schema mappings is not surprising if we note that in our relational examples, the rules are obtained by using a relational pattern – i.e., a conjunction of source atoms – and rearranging them as a conjunction of target atoms. Conjunctions of atoms are natural analogs of tree patterns. Indeed, the pattern on the right-hand side of the above rule, for example, can be viewed as the conjunction of the statements about existence of the following edge relations: between the root and a node labeled `SERVES`, between that node and a node labeled t, between the t-node and nodes labeled `airl`, `city`, `country`, and `phone`, respectively, and between those nodes and nodes carrying attribute values y, x, z, and u.

Of course we shall see when we describe XML data exchange that patterns could be significantly more complicated: they need not be simple translations of relational atoms. In fact one can use more complicated forms of navigation such as the horizontal ordering of siblings in a document, or the descendant relation. But for now our goal was to introduce the idea of tree patterns by means of a straightforward translation of a relational example.

1.2 Overview of the main tasks in data exchange

The key tasks in many database applications can be roughly split into two groups:

1. *Static analysis.* This mostly involves dealing with schemas; for example, the classical relational database problems such as dependency implication and normalization fall into this category. Typically, the input one considers is (relatively) small, e.g., a schema or a set of constraints. Therefore, somewhat higher complexity bounds are normally toler-

ated: for example, many problems related to reasoning about dependencies are complete for complexity classes such as NP, or coNP, or PSPACE.

2. *Dealing with data.* These are the key problems such as querying or updating the data. Of course, given the typically large size of databases, only low-complexity algorithms are tolerated when one handles data. For example, the complexity of evaluating a fixed relational algebra query is very low (AC^0, to be precise), and even more expressive languages such as Datalog stay in PTIME.

In data exchange, the key tasks too can be split into two groups. For static analysis tasks, we treat schema mappings as first-class citizens. The questions one deals with are generally of two kinds:

- *Consistency*. For these questions, the input is a schema mapping $\mathcal{M}$, and the question is whether it makes sense: for example, whether there exists a source S that has a solution under $\mathcal{M}$, or whether all sources of a given schema have solutions. These analyses are important for ruling out "bad" mappings that are unlikely to be useful in data exchange.
- *Operations on mappings*. Suppose we have a mapping $\mathcal{M}$ from a source schema $\mathbf{R_s}$ to a target schema $\mathbf{R_t}$, and another mapping $\mathcal{M}'$ that uses $\mathbf{R_t}$ as the source schema and maps it into a schema $\mathbf{R_u}$. Can we combine these mappings into one, the composition of the two, $\mathcal{M} \circ \mathcal{M}'$, which maps $\mathbf{R_s}$ to $\mathbf{R_u}$? Or can we invert a mapping, and find a mapping $\mathcal{M}^{-1}$ from $\mathbf{R_t}$ into $\mathbf{R_s}$, that undoes the transformation performed by $\mathcal{M}$ and recovers as much original information about the source as possible? These questions arise when one considers schema evolution: as schemas evolve, so do the mappings between them. And once we understand when and how we can construct mappings such as $\mathcal{M} \circ \mathcal{M}'$ or $\mathcal{M}^{-1}$, we need to understand their properties with respect to the "existence of solutions" problem.

Tasks involving data are generally of two kinds.

- *Materializing target instances*. Suppose we have a schema mapping $\mathcal{M}$ and a source instance S. Which target instance do we materialize? As we already saw, there could be many – perhaps infinitely many – target instances which are solutions for S under $\mathcal{M}$. Choosing one we should think of three criteria:
  1. it should faithfully represent the information from the source, under the constraints imposed by the mapping;
  2. it should not contain (too much) redundant information;
  3. the computational cost of constructing the solution should be reasonable.
- *Query answering*. Ultimately, we want to answer queries against the target schema. As we explained, due the existence of multiple solutions, we need to answer them in a way that is consistent with the source data. So if we have a materialized target T and a query Q, we need to find a way of evaluating it to produce the set of certain answers. As we shall see, sometimes computing $Q(T)$ does *not* give us certain answers, so we may need to change Q into another query Q' and then evaluate Q' on a chosen solution to get the