5 Schema Matching and Mapping

In Chapter 3 we described formalisms for specifying source descriptions and algorithms that use these descriptions to reformulate queries. To create the source descriptions, we typically begin by creating semantic matches. The matches specify how the elements of the source schemas and the mediated schema semantically correspond to one another. Examples include "attribute name in one source corresponds to attribute title in another," and "location is a concatenation of city, state, and zipcode." In the next step we elaborate the matches into semantic mappings, which are typically structured queries written in a language such as SQL. The mappings specify how to translate data across the sources and the mediated schema. The above two steps are often referred to as *schema matching* and *schema mapping*, respectively, and are the focus of this chapter.

In practice, creating the matches and mappings consumes quite a bit of the effort in setting up a data integration application. These tasks are often difficult because they require a deep understanding of the semantics of the schemas of the data sources and of the mediated schema. This knowledge is typically distributed among multiple people, including some who may no longer be with the organization. Furthermore, since the people who understand the meaning of the data are not necessarily database experts, they need to be aided by others who are skilled in writing formal transformations between the schemas.

This chapter describes a set of techniques that helps a designer create semantic matches and mappings. Unlike the algorithms presented in previous chapters, the task we face here is inherently a heuristic one. There is no algorithm that will take two arbitrary database schemas and flawlessly produce correct matches and mappings between them. Our goal is thus to create tools that significantly *reduce* the time it takes to create matches and mappings. The tools can help the designer speed through the repetitive and tedious parts of the process, and provide useful suggestions and guidance through the semantically challenging parts.

5.1 Problem Definition

We begin by defining the notion of schema element, semantic mapping, and semantic match. Let S and T be two relational schemas. We refer to the attributes and tables of S as the *elements* of S, and those of T as the *elements* of T, respectively.

Principles of Data Integration, DOI: 10.1016/B978-0-12-416044-6.00005-3

Example 5.1

Consider the two schemas in Figure 5.1. Schema DVD-VENDOR belongs to a vendor of DVDs and consists of three tables. The first table, **Movies**, describes the details of the movies themselves, while the remaining two, **Products** and **Locations**, describe the DVD products and the sale locations, respectively. Schema AGGREGATOR belongs to a shopping aggregation site. Unlike the individual vendor, the aggregator is not interested in all the details of the product, but only in the attributes that are shown to its customers, as captured in table **Items**.

Schema DVD-VENDOR has 14 elements: 11 attributes (e.g., id, title, and year) and three tables (e.g., **Movies**). Schema AGGREGATOR has five elements: four attributes and one table.

DVD-VENDOR
Movies(id, title, year)
Products(mid, releaseDate, releaseCompany, basePrice, rating, saleLocID)
Locations(lid, name, taxRate)

AGGREGATOR
Items(name, releaseInfo, classification, price)

FIGURE 5.1 Example of two database schemas. Schema DVD-VENDOR belongs to a DVD vendor, while AGGREGATOR belongs to a shopping site that aggregates products from multiple vendors.

5.1.1 Semantic Mappings

As described in Chapter 3, a *semantic mapping* is a query expression that relates a schema S with a schema T. Depending on the formalism used for source descriptions, the direction of the semantic mapping can differ.

Example 5.2

The following semantic mapping expresses the fact that the title attribute of the **Movies** relation in the DVD-VENDOR schema is the name attribute in the **Items** table in the AGGREGATOR schema.

```
SELECT name as title
FROM Items
```

The following semantic mapping goes in the reverse direction. It shows that to obtain the price attribute of the **Items** relation in the AGGREGATOR schema we need to join the **Products** and **Locations** tables in the DVD-VENDOR schema.

```
SELECT (basePrice * (1 + taxRate)) AS price
FROM Products, Locations
WHERE Products.saleLocID = Locations.lid
```

The previous two mappings obtain a single attribute of a relation. The following semantic mapping shows how to obtain an entire tuple for the **Items** table of the AGGREGATOR schema from data in the DVD-VENDOR schema:

```
SELECT title AS name, releaseDate AS releaseInfo, rating AS
   classification, basePrice * (1 + taxRate) AS price
FROM Movies, Products, Locations
WHERE Movies.id = Products.mid AND Products.saleLocID = Locations.lid
```

Given two schemas S and T, our goal is to create a semantic mapping between them.

Example 5.3

Consider building a system that integrates two data sources: a DVD vendor and a book vendor, with source schemas DVD-VENDOR and BOOK-VENDOR, respectively. Assume we use the schema AGGREGATOR in Figure 5.1 as the mediated schema.

If we use the Global-as-View (GAV) approach to relate the schemas, we need to describe the **Items** relation in the AGGREGATOR as a query over the source schemas. We would proceed as follows. We create an expression m_1 that specifies how to obtain tuples of **Items** from tuples of the DVD-VENDOR schema (this is the last expression in Example 5.2) and create an expression m_2 that shows how to obtain tuples of **Items** from relations in the BOOK-VENDOR schema. Finally, we return the SQL query (m_1 `UNION` m_2) as the GAV description of the **Items** table.

If we use the Local-as-View (LAV) approach, we would create an expression for every relation in the source schemas specifying how to obtain tuples for it from the **Items** relation.

5.1.2 Semantic Matches

A *semantic match* relates a set of elements in schema S to a set of elements in schema T, without specifying in detail (to the level of SQL queries) the exact nature of the relationship. The simplest type of semantic matches is *one-to-one matches*, such as

$$\textbf{Movies}.\text{title} \approx \textbf{Items}.\text{name}$$

$$\textbf{Movies}.\text{year} \approx \textbf{Items}.\text{year}$$

$$\textbf{Products}.\text{rating} \approx \textbf{Items}.\text{classification}.$$

An example of more complex matches is

$$\textbf{Items}.\text{price} \approx \textbf{Products}.\text{basePrice} * (1 + \textbf{Locations}.\text{taxRate})$$

We call such matches *one-to-many matches*, because they relate one element in schema S to multiple elements in schema T. Conversely, matches that relate multiple elements in schema *S* to one element in schema *T* are called *many-to-one matches*. Finally, *many-to-many matches* relate multiple elements in *S* to multiple elements in *T*.

5.1.3 Schema Matching and Mapping

To create the descriptions of data sources, we often begin by creating semantic matches, then elaborate the matches into mappings. These two tasks are often referred to as *schema matching* and *schema mapping*, respectively.

The main reason that we begin by creating semantic matches is that such matches are often easier to elicit from designers. For example, it may be easy for the designer to glean from their domain knowledge that price in AGGREGATOR is a function of basePrice and taxRate in DVD-VENDOR. Thus the designer can create the semantic match price $\approx$ basePrice * (1 + taxRate).

This semantic match for price specifies the functional relationship between basePrice and taxRate. But it does not specify which data values of basePrice should be combined with which data values of taxRate. Thus the match cannot be used to obtain data for price. In the next step, a schema mapping system takes the match as the input and elaborates it into a semantic mapping by filling in the missing details:

```
SELECT (basePrice * (1 + taxRate)) AS price
FROM Product, Location
WHERE Product.saleLocID = Location.lid
```

This semantic mapping combines a value *x* of basePrice with a value *y* of taxRate (to obtain a value for price) only if *x* and *y* come from tuples *p* and *q* of **Products** and **Locations**, respectively, such that *p*.saleLocID = *q*. lid.

Thus, in a sense, by supplying the semantic match, the designer supplies the SELECT and FROM clauses of the semantic mapping. The subsequent schema mapping system only has to create the WHERE clause, a far easier job than if the system were to start from scratch.

In practice, finding semantic matches is often already quite difficult. So the designer often employs a schema matching system to find the matches, verifies and corrects the matches, then employs a schema mapping system to elaborate the matches into the mappings. Even in such cases, breaking the overall process into the matching and mapping steps is still highly beneficial, because it allows the designer to verify and correct the matches, thus reducing the complexity of the overall process.

5.2 Challenges of Schema Matching and Mapping

As described, a schema matching and mapping system needs to reconcile the heterogeneity between the disparate schemas. Such heterogeneity arises in multiple forms. First, the table and attribute names could be different in the schemas even when they refer to the same underlying concept. For example, the attributes rating and classification both refer to the rating assigned to the movie by the Motion Picture Association of America. Second, in some cases multiple attributes in one schema correspond to a single attribute in the other. For example, basePrice and taxRate from the vendor schema are used to compute the value of price in the AGGREGATOR schema. Third, the tabular organization of the schemas is different. The aggregator only needs a single table, whereas the DVD vendor requires three. Finally, the coverage and level of details of the two schemas are different. The DVD vendor models details such as releaseDate and releaseCompany, whereas the aggregator does not.

The underlying reason for semantic heterogeneity is that schemas are created by different people whose tastes and styles are likely to be different. We see the same phenomenon when comparing computer programs written by different programmers. Even if two programmers write programs that behave exactly the same, it is likely that they will structure the programs differently, and certainly name variables in different ways. Another fundamental source of heterogeneity is that disparate databases are rarely created for the *exact* same purpose. In our example, even though both schemas model movies, the DVD vendor is managing its inventory, while the aggregator is concerned only with customer-facing attributes.

Reconciling the semantic heterogeneity between the schemas is difficult for several reasons:

- **The semantics is not fully captured in the schemas:** In order to correctly map between the schemas, the schema mapping system needs to understand the intended semantics of each schema. However, a schema itself does not completely describe its full meaning. A schema is a set of symbols that represent some model that was in the mind of the designer, but does not fully capture the model. For example, the name of the attribute rating in schema DVD-VENDOR does not convey the information that this is the rating by the Motion Picture Association of America, not the rating by the customers. In some cases, elements of the schema are accompanied by textual descriptions of their intent. However, these descriptions are typically partial at best, and even then, they are in natural language, making it hard for a program to understand.
- **Schema clues can be unreliable:** Since the semantics is not fully captured in the schemas, to perform adequately, the schema mapping system must glean enough of the semantics of the schema from both its formal specification, such as names, structures, types, and data values, and any other clues that may accompany the schema, such as text descriptions and the way the schema is used. However, such specification and clues can be unreliable. Two schema elements may share the same name and yet refer to different real-world concepts. For example, attribute name in schema DVD-VENDOR refers to the name of the sale location, whereas attribute name in schema AGGREGATOR refers to the name of the movie. Conversely, two attributes with different names, such as rating and classification, can refer to the same real-world concept.

- **Intended semantics can be subjective:** Sometimes it is difficult to decide whether two attributes should match. For example, a designer may think that attribute plot-summary matches attribute plot-synopsis, whereas another designer may not. In certain schema mapping applications (e.g., in military domains), it is not uncommon to have a committee of experts voting on such issues.
- **Correctly combining the data is difficult:** It is difficult not just to find the semantic matches, but also to elaborate the matches into mappings. When elaborating matches into mappings, the designer must find the correct way to combine the data values of the various attributes, typically using a join path coupled with various filtering conditions. Consider, for example, the semantic match

$$\textbf{Items}.\mathsf{price} \approx \textbf{Products}.\mathsf{basePrice} * (1 + \textbf{Locations}.\mathsf{taxRate})$$

  When elaborating this match into a mapping for price, the designer must decide how to join the two tables **Products** and **Locations** and whether any filtering condition should be applied to the join result. In practice, numerous join paths can exist between any two tables, and even after the designer has decided on a join path, there are still many possible ways to perform the join (e.g., full outer join, left outer join, right outer join, inner join). To decide which combination is correct, the designer often must examine a large amount of data in both schemas, in an error-prone and labor-intensive process.

A NOTE ON STANDARDS

It is tempting to argue that a natural solution to the heterogeneity problem is to create standards for every conceivable domain and encourage database designers to follow these standards. Unfortunately, this argument fails in practice for several reasons. It is often hard to agree on standards since some organizations are already entrenched in particular schemas and there may not be enough of an incentive to standardize. But more fundamentally, creating a standard is possible when data are used for the same purposes. But, as the above examples illustrated, uses of data vary quite a bit, and therefore so do the schemas created for them.

A second challenge to standardization is the difficulty of precisely delineating domains. Where does one domain end and another begin? For example, imagine trying to define a standard for the domain of people. Of course, the standard should include attributes such as name, address, phone number, and maybe employer. But beyond those attributes, others are questionable. People play many roles, such as employees, students (or alumni), bank account holders, moviegoers, and coffee consumers. Each of these roles can contribute more attributes to the standard, but including them all is clearly not practical. As another example, consider modeling scientific information, beginning with genes and proteins. We would also like to model the diseases they are linked to, the medications associated with the diseases, the scientific literature reporting findings related to the diseases, etc. There is no obvious point at which we can declare the end of this domain.

In practice, standards work for very limited use cases where the number of attributes is relatively small and there is strong incentive to agree on them (e.g., if exchanging data is critical for business processes). Examples include data that need to be exchanged between banks or in electronic commerce and medical data about vital signs of patients. Even in these cases, while the data may be *shared* in a standardized schema, each data source models the data internally with a different schema.

Before we proceed, we emphasize that heterogeneity appears not just at the schema level, but also at the data level. For example, two databases may each have a column named companyName but use different strings to refer to the same company (e.g., IBM vs. International Business Machines or HP vs. Hewlett Packard). We defer the discussion of data-level heterogeneity to Chapter 7.

5.3 Overview of Matching and Mapping Systems

We now outline the different components of matching and mapping systems. In what follows, we will use the terms "match" and "correspondence" interchangeably, when there is no ambiguity.

5.3.1 Schema Matching Systems

The goal of a schema matching system is to produce a set of matches, i.e., correspondences, between two given schemas S and T. One solution is to provide the designer a convenient graphical user interface to specify correspondences manually, but this chapter focuses on how to *automatically* create such correspondences that are then validated by a designer.

We begin by considering matching systems that produce one-to-one matches, such as title $\approx$ name and rating $\approx$ classification. The first observation from our DVD example is that no *single* heuristic is guaranteed to yield accurate matches. One set of heuristics is based on examining the similarities between the names of the schema elements. Using such heuristics, we may be able to infer that attribute releaseInfo in schema AGGREGATOR matches either attribute releaseDate or attribute releaseCompany in schema DVD-VENDOR, but we do not know which one.

Another set of heuristics examines the similarities between the data values when they are available. For example, suppose that releaseInfo specifies the year in which the DVD is released, and so does releaseDate. Then by examining the data values, we may be able to infer that releaseInfo matches either attribute releaseDate of table **Products** or attribute year of table **Movies**, but again we do not know which one. Other clues we may consider are the

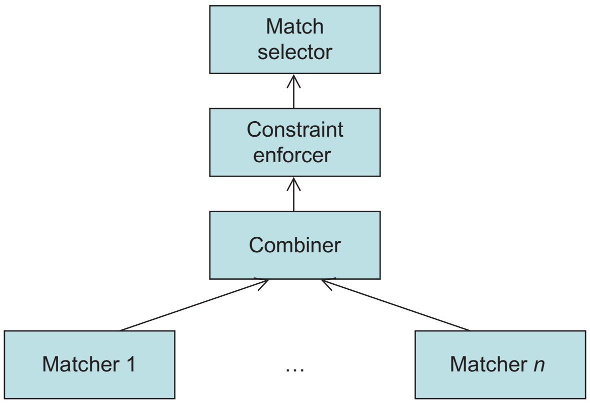

FIGURE 5.2 The components of a typical schema matching system.

proximity of attributes to each other or how attributes are used in queries. But again, none of these clues by itself is likely to produce good matches for all attributes.

On the other hand, by combining multiple clues, we can improve matching accuracy. For example, by examining both the names and the data values of the attributes, we can infer that releaseInfo is more likely to match releaseDate than releaseCompany and year.

The variety of available clues and heuristics and the need to exploit all of them to maximize matching accuracy motivate the architecture of a schema matching system, shown in Figure 5.2. A typical schema matching system consists of the following components:

- *Matchers (schemas $\rightarrow$ similarity matrix):* A matcher takes two schemas S and T and outputs a similarity matrix, which assigns to each element pair s of S and t of T a number between 0 and 1. The higher this number, the more confident the matcher is that the match $s \approx t$ is correct. Each matcher is based on a certain set of heuristics and, as a result, exploits a certain kind of clues. For example, a matcher may compare the similarities of the names of schema elements, while another matcher may compare the data values. The matching system employs several such matchers, and we describe several common types of matchers in Section 5.4.
- *Combiner (matrix $\times \ldots \times$ matrix $\rightarrow$ matrix):* A combiner merges the similarity matrices output by the matchers into a single one. Combiners can take the average, minimum, maximum, or a weighted sum of the similarity scores. More complex types of combiners use machine learning techniques or elaborate handcrafted scripts. We describe several types of combiners in Section 5.5.
- *Constraint enforcer (matrix $\times$ constraints $\rightarrow$ matrix):* In addition to clues and heuristics, domain knowledge plays an important role in pruning candidate matches. For example, knowing that many movie titles contain four words or more, but most location names do not, can help us guess that **Items**. name is more likely to match

Movies.title than **Locations**.name. Hence, the third component of the schema matching system, the *constraint enforcer*, enforces such constraints on the candidate matches. In particular, it transforms the similarity matrix produced by the combiner into another one that better reflects the true similarities. We describe several techniques to enforce constraints in Section 5.6.

- *Match selector (matrix → matches):* The last component of the matching system produces matches from the similarity matrix output by the constraint enforcer. The simplest selection strategy is thresholding: all pairs of schema elements with similarity scores exceeding a given threshold are returned as matches. More complex strategies include formulating the selection as an optimization problem over a weighted bipartite graph. We describe common selector types in Section 5.7.

Schema matching tasks are often repetitive, such as matching a mediated schema to the schemas of tens to hundreds of data sources. Section 5.8 describes a set of methods based on machine learning techniques that enable the schema matching system to reuse previous matches, that is, to learn from them to match new schemas. Section 5.9 then describes how to discover complex semantic matches, which require a more complex matching architecture, compared to the architecture of finding one-to-one matches that we described above.

5.3.2 Schema Mapping Systems

Once the matches have been produced, our task is to create the actual mappings. Here, the main challenge is to find how tuples from one source can be translated into tuples in the other. For example, the mapping should specify that in order to compute the price attribute in the **Items** table, we need to join **Products** with **Locations** on the attribute saleLocation = name and add the appropriate local tax. The challenge faced by this component is that there may be more than one possible way of joining the data. For example, we can join **Products** with **Movies** to obtain the origin of the director, and compute the price based on the taxes in the director's country of birth.

In Section 5.10, we describe how a mapping system can explore the possible ways of joining and taking unions of data and help the designer by proposing the most likely operations and by creating the actual transformations.

5.4 Matchers

The input to a matcher is a pair of schemas S and T. In addition, the matcher may consider any other surrounding information that may be available, such as data instances and text descriptions. The matcher outputs a similarity matrix that assigns to each element pair s of S and t of T a number between 0 and 1 predicting whether s corresponds to t.

There are a plethora of techniques for guessing matches between schema elements, each based on a different set of heuristics and clues. In this section we describe a few common basic matchers. We describe two classes of matchers: those that compare the names of schema elements and others that compare the data instances. It is important to keep in mind that for specific domains it is often possible to develop more specialized and effective matchers.

5.4.1 Name-Based Matchers

An obvious source of matching heuristics is based on comparing the names of the elements, in the hope that the names convey the true semantics of the elements. However, since names are seldom written in the exact same way, the challenge is to find effective similarity measures that reflect the similarity of element meanings. In principle, any of the techniques for string matching described in Chapter 4, such as edit distance, Jaccard measure, or the Soundex algorithm, can be used for name matching.

It is very common for element names to be composed of acronyms or short phrases. Hence, it is typically useful to normalize the element names before applying the similarity measures. We describe several common normalization procedures below, and note that some of them require domain-specific dictionaries.

- Split names according to certain delimiters, such as capitalization, numbers, or special symbols. For example, saleLocID would be split into sale, Loc, and ID, and agentAddress1 would be split to agent, Address, and 1.
- Expand known abbreviations or acronyms. For example, Loc may be expanded into Location, and cust may be expanded into customer.
- Expand a string with its synonyms. For example expand cost into price.
- Expand a string with its hypernyms. For example, product can be expanded into book, dvd, cd.
- Remove articles, propositions, and conjunctions, such as in, at, and.

Example 5.4

Consider again the two schemas DVD-VENDOR and AGGREGATOR, which are reproduced in Figure 5.3(a) for ease of exposition. To match the schemas, we can employ a name-based matcher that normalizes the names of the schema elements, then uses a set-based similarity measure such as the Jaccard measure or TF/IDF measure to compute the similarities between the names.

Figure 5.3(b) shows the similarity matrix produced by the name-based matcher. The figure shows the matrix in a compact fashion, by showing only schema element pairs with nonzero similarity scores and grouping these pairs by the first attribute. The first group, name $\approx$ ⟨name: 1,

DVD-VENDOR
Movies(id, title, year)
Products(mid, releaseDate, releaseCompany, basePrice, rating, saleLocID)
Locations(lid, name, taxRate)

AGGREGATOR
Items(name, releaseInfo, classification, price)

(a)

name-based matcher: name ≈ ⟨name: 1, title: 0.2⟩
releaseInfo ≈ ⟨releaseDate: 0.5, releaseCompany: 0.5⟩
price ≈ ⟨basePrice: 0.8⟩

(b)

data-based matcher: name ≈ ⟨name: 0.2, title: 0,8⟩
releaseInfo ≈ ⟨releaseDate: 0.7⟩
classification ≈ ⟨rating: 0.6⟩
price ≈ ⟨basePrice: 0.2⟩

(c)

average combiner: name ≈ ⟨name: 0.6, title: 0.5⟩
releaseInfo ≈ ⟨releaseDate: 0.6, releaseCompany: 0.25⟩
classification ≈ ⟨rating: 0.3⟩
price ≈ ⟨basePrice: 0.5⟩

(d)

FIGURE 5.3 (a) Two schemas (reproduced from Figure 5.1); (b)-(c) the similarity matrices produced by two matchers for the above two schemas; and (d) the combined similarity matrix.

title: 0.2⟩, states that attribute name in schema AGGREGATOR matches name in DVD-VENDOR with score 1 and matches title in DVD-VENDOR with score 0.2. The remaining two groups are similarly interpreted.

With all of the above techniques, it is important to keep in mind that element names in schemas do not always convey the entire semantics of the element. Commonly, the element names will be ambiguous because they already assume a certain context that would convey their meaning to a human looking at the schema. An example of an especially ambiguous element name is name. In one context, name can refer to a book or a movie, in another to a person or animal, and in yet another to a gene, chemical, or product. Further exacerbating the problem is the fact that schemas can be written in multiple languages or using different local or industry conventions. Of course, in many cases, the designers of the schema did not anticipate the need to integrate their data with others and therefore did not carefully think of their element names beyond their immediate needs.

5.4.2 Instance-Based Matchers

Data instances often convey a significant amount of meaning of schema elements. We now describe several common techniques for predicting correspondences based on analyzing data values. Keep in mind that not all schema matching scenarios have access to instance data.

Creating Recognizers

The first technique is to build *recognizers* that employ dictionaries or rules to recognize the data values of certain kinds of attributes. Consider, for example, attribute classification in schema AGGREGATOR. A recognizer for this attribute employs a small dictionary that lists all possible classification values (G, PG, PG-13, R, etc.). Given a new attribute, if most of its values appear in the dictionary, then the recognizer can conclude that the new attribute is likely to match classification. Other examples include using dictionaries to recognize U.S. states, cities, genes, and proteins and using relatively simple rules to recognize prices and zip codes.

In general, given two schemas S and T, a recognizer for an element s of S produces a similarity matrix. In this matrix, each cell (s, t_i), where t_i ranges over all elements of T, is assigned a similarity score. All other cells are assigned the special value "n/a," because the recognizer is not designed to make predictions for these cells.

Measuring the Overlap of Values

The second technique is based on measuring the overlap of values appearing in the two schema elements and applies to fields whose values are drawn from some finite domain, such as movie ratings, movie titles, or country names. The Jaccard measure is commonly used for this purpose (see Chapter 4).

Example 5.5

Continuing with Example 5.4, to match the two schemas DVD-VENDOR and AGGREGATOR, we can employ a data-based matcher that measures the overlap of values using the Jaccard measure. Figure 5.3(c) shows the similarity matrix produced by this matcher, again in a compact format.

Consider the first group in the output, name $\approx$ ⟨name: 0.2, title: 0.8⟩. Attribute name of schema AGGREGATOR (which refers to DVD titles) shares few values with name of DVD-VENDOR (which refers to the names of the sale locations), resulting in the low similarity score of 0.2. In contrast, name shares many values with title (both referring to DVD titles), resulting in the high similarity score of 0.8.

Using Classifiers

The third technique builds classifiers on one schema and uses them to classify the elements of the other schema. Classifier techniques commonly used are naive Bayes, decision trees, rule learning, and support vector machines. We will discuss classifiers in more detail in Section 5.8, but we give a brief example here. For each element s_i of schema S, we train a classifier C_i to recognize the instances of s_i. To do this, we need a training set of positive and negative examples. We build this training set by taking all data instances of s_i (that are available) to be positive examples and taking all data instances of other elements of schema S to be negative examples.

After we have trained classifier C_i on the training set, we are ready to use it to compute the similarity score between element s_i of schema S and each element t_j of schema T. To do this, we apply the classifier C_i to the data instances of t_j. For each data instance of t_j, the classifier C_i will produce a number between 0 and 1 that indicates how confident it is that the data instance is also a data instance of s_i. We can then aggregate these confidence scores to obtain a number that we return as the similarity between elements s_i and t_j. A simple way to perform the aggregation is to compute the average confidence score over all the data instances of t_j.

Example 5.6

If s_i is address, then positive examples may include “Madison WI” and “Mountain View CA,” and negative examples may include “(608) 695 9813” and “Lord of the Rings.” Now suppose that element t_j is location and that we have access to three data instances of this element: “Milwaukee WI,” “Palo Alto CA,” and “Philadelphia PA.” Then the classifier C_i may predict confidence scores 0.9, 0.7, and 0.5, respectively. In this case we may return the average confidence score of 0.7 as the similarity score between s_i = address and t_j = location.

In practice, the designer decides which schema should play the role of schema S (on which we build the classifiers) and which schema should play the role of schema T (the data instances to which we apply the classifiers). For example, if a schema is the mediated schema (such as AGGREGATOR), then it may be more appropriate to build the classifiers on that schema. This allows us to reuse the classifiers when we match the mediated schema with the schemas of new data sources.

In certain cases, such as when matching taxonomies of concepts, we may want to do it both ways: build classifiers on taxonomy S and use them to classify the data instances of taxonomy T, then build classifiers on taxonomy T and use them to classify the instances of S. The bibliographic notes provide pointers to such cases as well as to more techniques that examine data values to match schemas.

5.5 Combining Match Predictions

A combiner merges the similarity matrices output by the matchers into a single one. Simple combiners can take the average, minimum, or maximum of the scores. Specifically, if the matching system uses k matchers to predict the scores between the element s_i of schema S and the element t_j of schema T, then an average combiner will compute the score between these two elements as

$$combined(i,j) = \left[\sum_{m=1}^{k} matcherScore(m,i,j) \right] / k$$

where $matcherScore(m,i,j)$ is the score between s_i and t_j as produced by the mth matcher. The minimum combiner will compute the score between elements s_i and t_j to be the minimum of the scores produced by the matchers for the pair s_i and t_j, and the maximum combiner can be defined similarly.

Example 5.7

Continuing with Example 5.5, Figure 5.3(d) shows the similarity matrix produced by the average combiner, by merging the matrix of the name-based matcher in Figure 5.3(b) and the matrix of the data-based matcher in Figure 5.3(c).

The average combiner can be used when we do not have any reason to trust one matcher over the others. The maximum combiner is preferable when we trust a strong signal from the matchers. That is, if a matcher outputs a high value, then we are relatively confident that the two elements match. In that way, if the other matchers are neutral, then the strong signal would be reflected in the combination. The minimum combiner can be used when we want to be more conservative. That is, we require high similarity scores from all of the matchers in order to declare a match.

A more complex type of combiner uses elaborate hand-crafted scripts. For example, the script may specify that if s_i = address, then return the score of the data-based matcher that uses the naive Bayes classification technique; otherwise, return the average score of all matchers.

Another complex type of combiner, weighted-sum combiners, gives *weights* to each of the matchers, corresponding to their importance. While in some cases a domain expert may be able to provide such weights, in general it may be hard to specify them. Furthermore, the weights may differ depending on some characteristics of the elements being matched. In Section 5.8 we describe techniques that *learn* appropriate weights by inspecting the behavior of the schema matching system.

5.6 Enforcing Domain Integrity Constraints

During the process of schema matching the designer often has domain knowledge that can be naturally expressed as *domain integrity constraints*. The constraint enforcer can exploit these constraints to prune certain *match combinations*. Conceptually, the enforcer searches through the space of all match combinations produced by the combiner to find the one with the highest aggregated confidence score that satisfies the constraints. The following example illustrates the above idea.

Example 5.8

Recall from Example 5.7 that the average combiner produces the following similarity matrix:

$$\begin{aligned}
\mathsf{name} &\approx \langle \mathsf{name} : 0.6, \mathsf{title} : 0.5 \rangle \\
\mathsf{releaseInfo} &\approx \langle \mathsf{releaseDate} : 0.6, \mathsf{releaseCompany} : 0.25 \rangle \\
\mathsf{classification} &\approx \langle \mathsf{rating} : 0.3 \rangle \\
\mathsf{price} &\approx \langle \mathsf{basePrice} : 0.5 \rangle
\end{aligned}$$

In this matrix, the attributes name and releaseInfo each have two possible matches, resulting in a total of four match combinations, $M_1 - M_4$. The first combination M_1 is the set of matches $\{\mathsf{name} \approx \mathsf{name}, \mathsf{releaseInfo} \approx \mathsf{releaseDate}, \mathsf{classification} \approx \mathsf{rating}, \mathsf{price} \approx \mathsf{basePrice}\}$. The second combination M_2 is the same as M_1 except that it matches name with title, and so on.

For each match combination M_i, we can compute an aggregated score that captures the confidence of the combiner that M_i is correct. A straightforward way to compute this score is to multiply the scores of the matches in M_i, making the simplifying assumption that these scores have been produced independently of one another. Taking this approach, the score of the combination M_1 is $0.6 \cdot 0.6 \cdot 0.3 \cdot 0.5 = 0.054$, and the score of M_2 is $0.5 \cdot 0.6 \cdot 0.3 \cdot 0.5 = 0.045$, and so on.

Now suppose the designer knows that attribute name in schema AGGREGATOR refers to movie titles and that many movie titles contain at least four words. Then she can specify a constraint such as "if an attribute A matches name, then in any random sample of 100 data values of A, at least 10 values must contain four words or more."

The constraint enforcer searches for the match combination with the highest score that satisfies the above constraint. A simple way to do so is to check the combination with the highest score, then the one with the second highest score, and so on, until we find a combination that satisfies the constraint. In our example the combination with the highest score is M_1. M_1 specifies that attribute name of schema DVD-VENDOR matches name of AGGREGATOR. Since attribute name of DVD-VENDOR refers to the city names of the sale locations, the vast majority of its data values contain fewer than four words. Thus, the enforcer can quickly discover that M_1 does not satisfy the constraint.

The enforcer then considers M_2, the match combination with the second highest score, and finds that M_2 satisfies the constraint. So it selects M_2 as the desired match combination and produces the following similarity matrix as the output:

$$\begin{aligned}
\text{name} &\approx \langle \text{title} : 0.5 \rangle \\
\text{releaseInfo} &\approx \langle \text{releaseDate} : 0.6 \rangle \\
\text{classification} &\approx \langle \text{rating} : 0.3 \rangle \\
\text{price} &\approx \langle \text{basePrice} : 0.5 \rangle
\end{aligned}$$

We have described the conceptual working of the constraint enforcer. In practice, enforcing the constraints is significantly more complex. First, we have to handle a variety of constraints, some of which can only be satisfied to a certain degree. Second, the space of match combinations is often quite large, so we have to find ways to search it efficiently. We now elaborate on these challenges.

5.6.1 Domain Integrity Constraints

We distinguish between *hard constraints* and *soft constraints*. Hard constraints must be applied. The enforcer will not output any match combination that violates them. Soft constraints are of a more heuristic nature and may actually be violated in correct match combinations. Hence, the enforcer will try to minimize the number (and weight) of the soft constraints being violated, but may still output a match combination that violates one or more of them. Formally, we attach a *cost* to each constraint. For hard constraints, the cost is ∞, while for soft constraints, the cost can be any positive number.

Example 5.9

Figure 5.4 shows two schemas, BOOK-VENDOR and DISTRIBUTOR, and four common constraints. As the example shows, constraints typically rely on the structure of schemas (e.g., proximity of schema elements, whether an attribute is a key) and on special properties of attributes in a particular domain.

The constraint c_1 is a hard constraint. c_1 states that any attribute mapped to code in Items must be a key. The constraint c_2 is a soft constraint. It states that any attribute mapped to desc is likely to have an average length of at least 20 words, because descriptions tend to be long textual fields.

The constraint c_3 is a soft constraint. It captures the intuition that related attributes often appear in close proximity to one another in a schema; hence matches involving these attributes also often appear close to one another.

For example, suppose that A_1 = publisher $\approx B_1$ = brand and A_2 = location $\approx B_2$ = origin. Here B_2 is next to B_1, but A_2 is not next to A_1. Then the constraint states that there is no attribute $A*$ next to A_1 (e.g., pubCountry) that also closely matches B_2 (i.e., $|sim(A*, B_2) - sim(A_2, B_2)| \le \epsilon$ for

BOOK-VENDOR
Books(ISBN, publisher, pubCountry, title, review)
Inventory(ISBN, quantity, location)

DISTRIBUTOR
Items(code, name, brand, origin, desc)
InStore(code, availQuant)

	Constraint	**Cost**
c_1	If $A \approx$ Items.code, then A is a key.	∞
c_2	If $A \approx$ Items.desc, then any random sample of 100 data instances of A must have an average length of at least 20 words.	1.5
c_3	If $A_1 \approx B_1, A_2 \approx B_2$, B_2 is next to B_1 in the schema, but A_2 is not next to A_1, then there is no $A*$ next to A_1 such that $\lvert sim(A*, B_2) - sim(A_2, B_2) \rvert \leq \epsilon$ for a small prespecified ϵ.	2
c_4	If more than half of the attributes of Table U match those of Table V, then $U \approx V$.	1

FIGURE 5.4 Two schemas with integrity constraints over the matches between them.

a small prespecified ϵ). This is because if such $A*$ exists, we would rather match it, instead of A_2, to B_2, to keep the matches of A_1 and A_2 close to one another.

Finally, c_4 is also a soft constraint, which involves the matches at the table level.

Each constraint is specified only once by the designer during the matching process. The exact formalism in which constraints are represented is unimportant. The only requirement we impose is that given a constraint c and a match combination M, the constraint enforcer must be able to efficiently decide whether M violates c, given all the available data instances of the schemas.

It is important to note that just because the available data instances of the schemas conform to a constraint, that still does not mean that the constraint holds for other samples of data instances. For example, just because the current data values of an attribute A are distinct, that does not mean that A is a key. In many cases, however, the data instances that are available will be enough for the constraint enforcer to quickly detect a violation of such a constraint.

5.6.2 Searching the Space of Match Combinations

We now describe two algorithms for applying constraints to the similarity matrix output by the combiner. The first algorithm is an adaptation of A* search and is guaranteed to find the optimal solution but is computationally more expensive. The second algorithm, which applies only to constraints in which a schema element is affected by its neighbors (e.g., c_3 and c_4), is faster in practice but performs only local optimizations. We describe only the key ideas of the algorithms. The bibliographic notes provide pointers to the details.

Applying Constraints with A Search*

The A* algorithm takes as input the domain constraints $c_1, \ldots, c_p$ and the similarity matrix *combined*, produced by the combiner. The similarity matrix for our example is shown in Table 5.1. The algorithm searches through the possible match combinations and returns the one that has the lowest cost. The cost is measured by the likelihood of the match combination and the degree to which the combination violates the constraints.

Before we discuss the application of A* to constraint enforcement, we briefly review the main concepts of A*. The goal of A* is to search for a *goal state* within a set of states, beginning from an initial state. Each path through the search space is assigned a cost, and A* finds the goal state with the cheapest path from the initial state. A* performs a *best-first* search: start with the initial state, expand this state into a set of states, select the state with the smallest *estimated cost*, expand the selected state into a set of states, again select the state with the smallest estimated cost, and so on.

The estimated cost of a state n is computed as $f(n) = g(n) + h(n)$, where $g(n)$ is the cost of the path from the initial state to n, and $h(n)$ is a *lower bound* on the cost of the cheapest path from n to a goal state. Hence, the estimated cost $f(n)$ is a lower bound on the cost of the cheapest solution via n. A* terminates when it reaches a goal state, returning the path from the initial state to that goal state.

The A* algorithm is guaranteed to find a solution if one exists. When $h(n)$ is indeed a lower bound on the cost to reach a goal state, A* is also guaranteed to find the cheapest solution. The efficiency of A*, measured as the number of states that it needs to examine to reach the goal state, depends on the accuracy of the heuristic $h(n)$. The closer $h(n)$ is to

Table 5.1 The *Combined* Similarity Matrix for the Two Schemas in Figure 5.4

	Items	code	name	brand	origin	desc	**InStore**	code	availQuant
Books	0.5	0.2	0.5	0.1	0	0.4	0.1	0	0
ISBN	0.2	0.9	0	0	0	0	0	0.1	0
publisher	0.2	0.1	0.6	0.75	0.4	0.2	0	0.1	0
pubCountry	0.3	0.15	0.3	0.5	0.7	0.3	0.05	0.15	0
title	0.25	0.1	0.8	0.3	0.45	0.2	0.05	0.1	0
review	0.15	0	0.6	0.1	0.35	0.65	0	0.05	0
Inventory	0.35	0	0.25	0.05	0.1	0.1	0.5	0	0
ISBN	0.25	0.9	0	0	0	0.15	0.15	0.9	0
quantity	0	0.1	0	0	0	0	0	0.75	0.9
location	0.1	0.6	0.6	0.7	0.85	0.3	0	0.2	0

the actual lowest cost, the fewer states A* needs to examine. In the ideal case where $h(n)$ is the lowest cost, A* marches straight to the goal with the lowest cost.

We now describe applying A* to match schema S_1, whose attributes are $A_1, \ldots, A_n$, with schema S_2, whose attributes are $B_1, \ldots, B_m$. We note that the method below is not the only way to apply A* to constraint enforcement.

STATES

We define a state to be a tuple of size n, where the ith element either specifies a correspondence for A_i or is a wildcard *, representing that the correspondence for A_i is undetermined. Thus, a state can be interpreted as representing the *set* of match combinations that are consistent with the specifications. For example, suppose $n = 5$ and $m = 3$; then state $(B_2, *, B_1, B_3, B_2)$ represents three match combinations $(B_2, B_1, B_1, B_3, B_2)$, $(B_2, B_2, B_1, B_3, B_2)$, and $(B_2, B_3, B_1, B_3, B_2)$. We refer to a state as an *abstract state* if it contains wildcards, and a *concrete state* otherwise. A concrete state is a match combination.

INITIAL STATE

The initial state is defined to be $(*, *, \ldots, *)$, which represents all possible match combinations.

GOAL STATES

The goal states are the states that do not contain any wildcards, and hence completely specify a candidate match combination.

We now describe how the A* algorithm computes costs for goal states and then for abstract states.

EXPANDING STATES

To expand an abstract state, we choose a wildcard at position i and create the set of states in which the wildcard is replaced by the possible correspondences for the attribute in position i. The main decision to be made here is which of the multiple wildcards to choose. Clearly, the positions for which only one correspondence exists should be chosen first. After that, we may want to prefer the wildcards for which there is a correspondence that has a much higher score than the others.

COST OF GOAL STATES

The cost of a goal state combines our estimate of the likelihood of the combination and the degree to which it violates the domain constraints. Specifically, we evaluate a combination M as follows:

$$cost(M) = -LH(M) + cost(M, c_1) + cost(M, c_2) + \cdots + cost(M, c_p) \tag{5.1}$$

In the equation, $LH(M)$ represents the likelihood of M according to the similarity matrix, and $cost(M, c_i)$ represents the degree to which M violates the constraint c_i. In practice, we also typically assign to each component of the sum a weight, to represent the trade-offs among the cost components, but we omit this detail here.

The term $LH(M)$ is defined as $log\ conf(M)$, where $conf(M)$ is the confidence score of the match combination M. We compute the confidence as the product of the confidence scores of all the matches in M. Specifically, if $M = (B_{l_1}, \ldots, B_{l_n})$, where the B_{l_i} are the elements of schema S_2, then

$$conf(M) = combined(1, l_1) \times \cdots \times combined(n, l_n)$$

The formula for $conf(M)$ effectively assumes that the confidence scores of the matches in M are independent of one another. This assumption is clearly not true, because in many cases the appropriate match for one attribute depends on the matches of its neighbors. However, we make this assumption to reduce the cost of our search procedure. Note also that the definition of $LH(M)$ coupled with Equation 5.1 implies that we prefer the combination with the highest confidence score, all other things being equal.

Example 5.10

Consider the goal state corresponding to the following match combination (highlighted in boldface in Table 5.1).

Books ≈ Items	Inventory ≈ InStore
Books.ISBN ≈ Items.code	Inventory.ISBN ≈ InStore.code
Books.title ≈ Items.name	Inventory.quantity ≈ InStore.availQuant
Books.publisher ≈ Items.brand	
Books.pubCountry ≈ Items.origin	
Books.review ≈ Items.desc	

The cost of this state is the degree to which the combination violates the integrity constraints minus the likelihood of the combination. The likelihood of the combination is the log of the product of the boldface values in Table 5.1. This particular match combination satisfies all the integrity constraints in Figure 5.4 and therefore incurs no additional cost.

In contrast, consider the match combination in which **Books**.review ≈ **Items**.desc is replaced by **Books**.title ≈ **Items**.desc, and **Books**.pubCountry ≈ **Items**.origin is replaced by **Inventory**.location ≈ **Items**.origin. This combination would violate the two integrity constraints c_2 and c_3. Hence, the combination would incur an additional cost of 3.5.

COST OF ABSTRACT STATES

The cost of an abstract state s is the sum of the cost of the path from the initial state to s, denoted by $g(s)$, and an estimate of the cost of the path from s to a goal state, denoted by $h(s)$. The estimate $h(s)$ must be a lower bound on the cost of the cheapest path from s to a goal state.

The cost of the path from the initial state to s is computed in a similar fashion to the cost of a path to a goal state, except that we ignore the wildcards.

Example 5.11

Consider the abstract state with the following partial matches:

Books ≈ Items	Books.review ≈ Items.desc
Books.ISBN ≈ Items.code	Books.publisher ≈ Items.brand
Books.pubCountry ≈ Items.origin	Books.title ≈ Items.name

Given the similarity values in Table 5.1, the cost from the initial state to the above abstract state is $-log(0.5*0.9*0.7*0.65*0.75*0.8)$.

The cost $h(s)$ of the path from s to a goal state is estimated as the sum of two factors $h_1(s)$ and $h_2(S)$. The factor $h_1(s)$ is computed to be a lower bound on the cost of expanding all wildcards in s. For example, suppose that $s = (B_1 B_2 * * B_3)$; then the estimated cost $h_1(s)$ is

$$h_1(s) = -(log[\max_i combined(3,i)] + log[\max_i combined(4,i)])$$

Example 5.12

Continuing with Example 5.11, the wildcards are the assignment to **Inventory** and its attributes. The estimate is the product of the highest values in each of their corresponding rows, i.e., $-log(0.5*0.9*0.9*0.85)$.

The term $h_2(s)$ is an estimate on the degree to which goal states reachable from s violate the constraints. It is defined to be the sum $\Sigma_{i=1}^{p} cost(s,c_i)$, where $cost(s,c_i)$ is the estimate for the constraint c_i. To estimate the degree to which an abstract state violates a constraint, the algorithm assumes a best-case scenario—if it cannot show that *all* goal states reachable from s violate c_i, then it assumes that c_i is not violated. In Example 5.11, it is easy to show that at least one goal state reachable from the abstract state does not violate the constraints. Thus, the h_2 value of this abstract state is 0.

The specific method for determining whether the possible goal states violate c_i depends on the type of constraint of c_i. For example, consider $s = (B_1, B_2, *, *, B_3)$ and the hard constraint c = "at most one attribute matches B_2." Clearly there are goal states that are represented by s and satisfy c, such as goal state $(B_1, B_2, B_3, B_1, B_3)$, and therefore we say that s satisfies c. Consider $s' = (B_1, B_2, *, *, B_2)$. Since any goal state represented by s' violates the hard constraint c, we set $cost(s', c) = \infty$.

It is trivial to show that the cost $f(s) = g(s) + h(s)$ is a lower bound on the cost of any goal state that is in the set of concrete states represented by s, and therefore A* will find the cheapest goal state.

Applying Constraints with Local Propagation

The second algorithm we describe is based on propagating constraints locally from elements of the schema to their neighbors until we reach a fixed point. As before, we describe an algorithm that can be instantiated in a variety of ways.

Example 5.13

In Figure 5.4, constraints c_3 and c_4 involve computing correspondences based on properties of an element's neighbors. To apply them with the algorithm we describe now, we need to rephrase them to be stated from the perspective of a pair of nodes, one from S_1 and one from S_2.

Constraint c_3 can be stated as follows. If $sim(A_1, B_1) \leq 0.9$ and A_1 has a neighbor A_2 such that $sim(A_2, B_2) \geq 0.75$, and B_1 is a neighbor of B_2, then increase $sim(A_1, B_1)$ by α.

Constraint c_4 can be stated as follows. Let $R_1 \in S_1$ and $R_2 \in S_2$ be two tables, and suppose the number of neighbors they have is within a factor of 2. (The neighbors of a table node include all nodes that represent the attributes of the table.) Then, if $sim(R_1, R_2) \leq 0.9$ and at least half of the neighbors of R_1 are in the set $\{A_i \in attributes(R_1) \mid \exists B_j \in attributes(R_2) \text{ s.t. } sim(A_i, B_j) \geq 0.75\}$, increase the similarity $sim(R_1, R_2)$ by α.

INITIALIZATION

We begin by representing the schemas S_1 and S_2 as graphs. In the case of relational schemas, the graph representation is a tree (see Figure 5.5). The root of the tree is the schema node, its children are the relation names, and the leaves are their respective attributes. Other data models (e.g., XML, object-oriented) also have natural representations as graphs. Intuitively, edges in the graph represent neighborhood relations in the schema.

The algorithm computes a similarity matrix *sim* that is initialized to be the *combined* matrix. Recall that $combined(i, j)$ is the estimate that $A_i \in S_1$ corresponds to $B_j \in S_2$.

ITERATION

The algorithm proceeds by iteratively selecting a node s_1 in the graph of S_1 and updating the values in *sim* based on the similarities computed for its neighbors. A common method for tree traversal is bottom-up, starting from the leaves and going up to the root.

When we select a node $s_1 \in S_1$, we need to apply constraints that involve a node $s_2 \in S_2$. Since we do not want to compare every pair of nodes, the algorithm checks constraints only on pairs of elements that pass some filter. For example, the nodes need to have the same number of neighbors within a factor of 2. (Note that this filter is already built into the constraint c_4.) The algorithm applies the constraint, modifying the similarities if necessary, and proceeds to the next node.

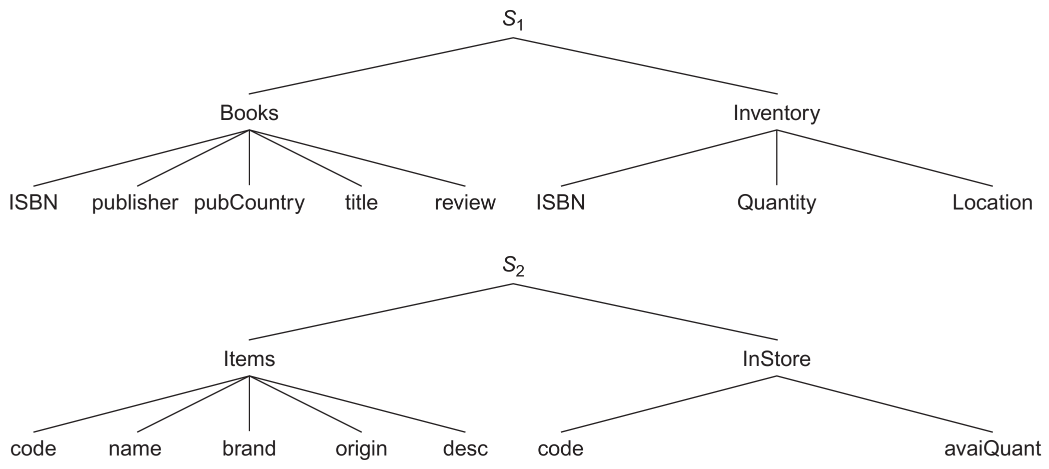

FIGURE 5.5 Tree representation of the schemas in Figure 5.4.

TERMINATION
The algorithm terminates either after a fixed number of iterations or when the changes to *sim* are smaller than a predefined threshold.

Example 5.14

In our example, let α be 20%. When we choose the node pubCountry in **Books**, we will consider the pair (pubCountry, origin) because it has the highest value in the row of pubCountry. Since pubCountry is a neighbor of publisher, and origin is a neighbor of brand, and the similarity score between publisher and brand is 0.75, constraint c_3 dictates that we increase the similarity between pubCountry and origin by 20% to 0.84. When we consider the nodes **Items** and **Books**, constraint c_4 will cause us to increase their similarity by 20% to 0.6.

The algorithm will terminate after a few iterations because the constraints c_3 and c_4 can only cause the similarities to increase, and both of them do not apply once the similarity is over 0.9.

5.7 Match Selector

The result of the previous components of the schema matcher is a similarity matrix for the schemas of S and T. The matrix combines the predictions of multiple matchers and the knowledge expressed as domain constraints. The last component of the matching system, the match selector, produces matches from the similarity matrix.

The simplest selection strategy is *thresholding:* all pairs of schema elements with similarity score equaling or exceeding a given threshold are returned as matches. For instance, in Example 5.8 the constraint enforcer produces the following matrix:

$$\begin{aligned}
\text{name} &\approx \langle \text{title} : 0.5 \rangle \\
\text{releaseInfo} &\approx \langle \text{releaseDate} : 0.6 \rangle \\
\text{classification} &\approx \langle \text{rating} : 0.3 \rangle \\
\text{price} &\approx \langle \text{basePrice} : 0.5 \rangle
\end{aligned}$$

Given the threshold 0.5, the match selector produces the following matches: name ≈ title, releaseInfo ≈ releaseDate, and price ≈ basePrice.

More sophisticated selection strategies produce the top few match combinations. This is so that when the user makes a particular choice, the other suggested matches can be adjusted accordingly. For example, suppose that the match selector produces the top-ranked match combination

$$\text{phone} \approx \text{shipPhone}, \ \text{addr} \approx \text{shipAddr}$$

and the second-ranked match combination

$$\text{phone} \approx \text{billPhone}, \ \text{addr} \approx \text{billAddr}$$

Then once the designer has selected phone ≈ billPhone as a correct match, the system may recommend addr ≈ billAddr, even though this match may have a lower score than addr ≈ shipAddr.

There are a variety of algorithms that can be used to select match combinations. A common algorithm formulates the match selection problem as an instance of finding a *stable marriage*. Specifically, imagine the elements of S to be men and the elements of T to be women. Let the value $sim(i,j)$ be the degree to which A_i and B_j desire each other (note that in the world of schema matching A and B desire each other equally, though we can consider asymmetric correspondences, too). Our goal is to find a stable match between the men and the women. A match would be unstable if there are two couples $A_i \approx B_j$ and $A_k \approx B_l$ such that A_i and B_l would clearly want to be matched with each other, that is, $sim(i,l) > sim(i,j)$ and $sim(i,l) > sim(k,l)$.

We can produce a match combination without unhappy couples as follows. Begin with `match={}`, and repeat the following. Let (i,j) be the highest value in sim such that A_i and B_j are not in `match`. Add $A_i \approx B_j$ to `match`.

In contrast to the stable marriage, we can consider a match combination that maximizes the sum of the correspondence predictions. For example, in Figure 5.6, matching A to C and B to D forms a stable marriage match. However, matching A to D and B to C maximizes the total confidences of the correspondences.

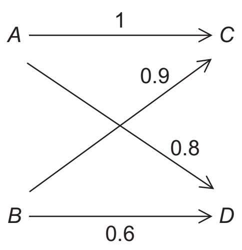

FIGURE 5.6 Matching A to C and B to D is a stable marriage match. However, matching A to D and B to C maximizes the total confidences of the correspondences.

5.8 Reusing Previous Matches

Schema matching tasks are often repetitive. For example, in the context of data integration, we are creating matches from data sources in the *same* domain to a single mediated schema. Similarly, in the enterprise context, we often have to update the semantic matches because one of the schemas has changed.

More generally, working in a particular domain, the same concepts tend to recur. As the designer works in the domain, he or she starts identifying how common domain concepts get expressed in schemas. As a result, the designer is able to create schema matches more quickly over time. For example, in the domain of real estate, one learns very quickly the typical concepts (e.g., house, lot, agent), the typical attributes of the concepts, and the properties of these attributes (e.g., the field that has numbers in the thousands is the price of the house, and the field that has longer text descriptions with words like "wonderful" and "spacious" is the description of the house).

Hence, an intriguing question is whether the schema matching system can *also* improve over time. Can a schema matching system learn from previous experience? In this section we describe how machine learning techniques can be applied to schema matching and enable the matching system to improve over time. We describe these techniques in the context of data integration, where the goal is to map a large number of data sources to a single mediated schema.

5.8.1 Learning to Match

Suppose we have n data sources, $S_1, \ldots, S_n$, and our goal is to map them to the mediated schema G. Recall that G is the schema used to formulate queries over the data integration system. We would like to *train* the system by manually providing it with semantic matches on a small number of data sources, say $S_1, \ldots, S_m$, where m is much smaller than n. The system should then *generalize* from these training examples and be able to *predict* matches for the sources $S_{m+1}, \ldots, S_n$.

We proceed by learning classifiers for the elements of the mediated schema. As explained in Section 5.4, a classifier for a concept C is an algorithm that identifies instances of C from those that are not. In this case, the classifier for an element e in the mediated schema will examine an element in a source schema and predict whether it matches e or not.

To create classifiers, we need to employ some machine learning algorithm. Like matchers, any machine learning algorithm also typically considers only one aspect of the schema and has its strengths and weaknesses. Hence, we combine multiple learning methods with a technique called *multi-strategy learning*. The training phase of multi-strategy learning works as follows:

- Employ a set of *learners*, $l_1, \ldots, l_k$. Each learner creates a classifier for each element e of the mediated schema G from the training examples of e. The training examples are derived using the semantic matches between G and the training data sources $S_1, \ldots, S_m$.
- Use a *meta-learner* to learn weights for the different learners. Specifically, for each element e of the mediated schema and each learner l, the meta-learner computes a weight $w_{e,l}$.

Example 5.15

Suppose the mediated schema G has three elements e_1, e_2, and e_3. Suppose further that we employ two instance-based learners: naive Bayes and decision tree (which we will describe shortly). Then the naive Bayes learner will create three classifiers, $C_{e_1,NB}$, $C_{e_2,NB}$, and $C_{e_3,NB}$, for the three elements e_1, e_2, and e_3, respectively. The goal of the classifier $C_{e_1,NB}$ is to decide whether a given data instance belongs to element e_1. The goals of classifiers $C_{e_2,NB}$ and $C_{e_3,NB}$ are similar.

To train the classifier $C_{e_1,NB}$, we have to assemble a set of positive and negative training examples. In principle, we can take all available data instances of element e_1 to be positive examples and all available instances of the other elements of the mediated schema to be negative examples. However, the mediated schema is virtual, so it has no data instances.

This is where the training sources $S_1, \ldots, S_m$ come in. Suppose when matching these sources to the mediated schema G, we have found that only two elements a and b of the sources match the element e_1 of the mediated schema. Then we can view the available data instances of a and b as the data instances of e_1, and hence as positive examples. We create negative examples in a similar fashion, using the instances of the elements (of the training data sources) that we know do not match e_1. We train the classifiers $C_{e_2,NB}$ and $C_{e_3,NB}$ in a similar fashion.

The decision tree learner creates three classifiers, $C_{e_1,DT}$, $C_{e_2,DT}$, and $C_{e_3,DT}$, and they can be trained analogously.

Finally, training the meta-learner means computing a weight for each pair of mediated-schema element and learner, for a total of six weights: $w_{e_1,NB}$, $w_{e_1,DT}$, $w_{e_2,NB}$, $w_{e_2,DT}$, $w_{e_3,NB}$, and $w_{e_3,DT}$.

Given the trained classifiers, the matching phase of multi-strategy learning works as follows. When presented with a schema S whose elements are $e'_1, \ldots, e'_n$:

- Apply the learners to $e'_1, \ldots, e'_n$. Denote by $p_{e,l}(e')$ the prediction of learner l on whether e' matches e.
- Combine the learners: $p_e(e') = \sum_{i=1}^{k} w_{e,l_i} * p_{e,l_i}(e')$.

The learners act as matchers, and the meta-learner acts as a combiner. Thus, the output of the meta-learner is a combined similarity matrix, which can be fed into a constraint enforcer, then a match selector, as described earlier.

Example 5.16

Continuing with Example 5.15, let S be a new data source with two elements e'_1 and e'_2. Then the naive Bayes learner will apply

- the classifier $C_{e_1,NB}$ to make predictions $p_{e_1,NB}(e'_1)$ and $p_{e_1,NB}(e'_2)$,
- the classifier $C_{e_2,NB}$ to make predictions $p_{e_2,NB}(e'_1)$ and $p_{e_2,NB}(e'_2)$, and
- the classifier $C_{e_3,NB}$ to make predictions $p_{e_3,NB}(e'_1)$ and $p_{e_3,NB}(e'_2)$.

The above six predictions form a similarity matrix, which can be viewed as the similarity matrix produced by the naive Bayes learner. A similar matrix is produced by the decision tree learner.

The meta-learner then combines the above two similarity matrices. Specifically, it computes the combined similarity score between e_1 and e'_1 to be the sum of the similarity scores, weighted by the learners:

$$p_{e_1}(e'_1) = w_{e_1,NB} * p_{e_1,NB}(e'_1) + w_{e_1,DT} * p_{e_1,DT}(e'_1)$$

The rest of the combined similarity scores are computed in a similar fashion.

In the next two sections we describe two common learners and how to train the meta-learner.

5.8.2 Learners

A variety of classification techniques have been employed as learners for schema matching. In what follows we describe two common learners: a rule-based learner and the naive Bayes learner.

Rule-Based Learner

A rule-based learner examines a set of training examples and computes a set of rules that can be applied to test instances. The rules can be represented as simple logical formulas or as decision trees. A rule-based learner works well in domains where there exists a set of rules, based on features of the training examples, that accurately characterize the instances of the class, such as identifying elements that adhere to certain formats. When the learned rules are applied to an instance, we return 1 if the instance satisfies the rules and 0 otherwise.

Consider the example of a rule learner for identifying phone numbers (for simplicity, we restrict our attention to American phone numbers). Example instances of strings that can be fed into the learners are “(608) 435-2322,” “849-7394,” and “5549902.” For each string, we first identify whether it is a positive or negative example of a phone number, and then extract values for a set of features we deem important. In our example, the features are “Does the string have 10 digits?”, “Does the string have 7 digits?”, “Is there a ‘(’ in position 1?”, “Is there a ‘)’ in position 5?”, and “Is there a ‘-’ in position 4?”

A common method to learn rules is to create a decision tree. Figure 5.7 shows an example decision tree for recognizing phone numbers. Informally, a decision tree is created as follows. We first search for a feature of the training data that most distinguishes between positive and negative examples. That feature becomes the root of the tree, and we divide the training instances into two sets, depending on whether they have the feature or not. We then continue recursively and build decision trees of the two subsets of instances. In Figure 5.7 we first distinguish between the instances that have 10 digits from those that do not. The next important feature for the 10-digit instances is the position of the ‘(’.

As described, the above decision tree encodes rules such as “if an instance i has 10 digits, a ‘(’ in position 1, and a ‘)’ in position 5, then i is a phone number,” and “if i has 7 digits, but no ‘-’ in position 4, then i is not a phone number.”

The Naive Bayes Learner

The naive Bayes learner examines the tokens of a given instance and assigns to the instance the most likely class, given the occurrences of tokens in the training data. Specifically, given an instance, the learner converts it into a *bag of tokens*. The tokens are generated by parsing and stemming the words and symbols in the instance. For example, “RE/MAX Greater Atlanta Affiliates of Roswell” becomes “re / max greater atlanta affili of roswell.”

Suppose that $c_1,\ldots,c_n$ are the elements of the mediated schema and that the learner is given a test instance $d = \{w_1,\ldots,w_k\}$ to classify, where the w_i are the tokens of the instance. The goal of the naive Bayes learner is to assign d to the element c_d that has

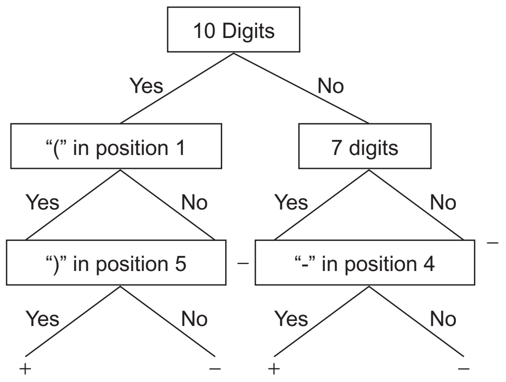

FIGURE 5.7 A decision tree for phone numbers.

the highest posterior probability given d. Formally, this probability is defined as $c_d = arg\ max_{c_i}\ P(c_i|d)$. To compute this probability, the learner can leverage Bayes' Rule, which states that $P(c_i|d) = P(d|c_i)P(c_i)/P(d)$. Using this rule, we have

$$\begin{aligned} c_d &= arg\ max_{c_i}\ [P(d|c_i)P(c_i)/P(d)] \\ &= arg\ max_{c_i}\ [P(d|c_i)P(c_i)] \end{aligned}$$

Hence, to make its prediction, the naive Bayes learner needs to estimate $P(d|c_i)$ and $P(c_i)$ from the training data. It approximates the probability $P(c_i)$ as the portion of training instances with label c_i. To compute $P(d|c_i)$, we assume that the tokens w_j appear in d *independently* of each other given c_i (this is where naive Bayes becomes naive). With this assumption, we have

$$P(d|c_i) = P(w_1|c_i)P(w_2|c_i)\cdots P(w_k|c_i)$$

We estimate $P(w_j|c_i)$ as $\frac{n(w_j,c_i)}{n(c_i)}$, where $n(c_i)$ is the total number of tokens in the training instances with label c_i, and $n(w_j,c_i)$ is the number of times token w_j appears in all training instances with label c_i.

Even though the independence assumption is typically not valid, the naive Bayes learner still performs surprisingly well in many domains. In particular, it works best when there are tokens that are strongly indicative of the correct label, by virtue of their frequencies. For example, it can effectively recognize house descriptions in real-estate listings, which frequently contain words such as "beautiful" and "fantastic"—words that seldom appear in other elements. The naive Bayes learner also works well when there are only weakly suggestive tokens, but many of them. In contrast, it fares poorly for short or numeric fields, such as color, zip code, or number of bathrooms.

5.8.3 Training the Meta-Learner

The meta-learner decides the weights to attach to each of the learners. The weights can be different for every mediated-schema element. To compute these weights, the meta-learner asks the learners for predictions on the training examples. Since the meta-learner knows the correct matches for the training examples, it is able to judge how well each learner performs with respect to each mediated-schema element. Based on this judgment, the meta-learner assigns to each combination of mediated-schema element e and base learner l a weight $w_{e,l}$ that indicates how much it *trusts* learner l's predictions regarding e.

Specifically, to learn the weights for a mediated-schema element e, the meta-learner first creates a set of training examples. Each training example is of the form $(d, p_1, \ldots, p_k, p*)$, where

- d is a data instance of a training source S_i, $1 \leq i \leq m$,
- each p_j is the prediction that the learner l_j made for the pair (d, e), and
- $p*$ is the correct prediction for the data instance d.

The meta-learner then applies a learning method, such as linear regression, to the above set of training examples, to learn the weights for e.

Example 5.17

Continuing with Examples 5.15–5.16, our goal is to learn the six weights $w_{e_1,NB}$, $w_{e_1,DT}$, $w_{e_2,NB}$, $w_{e_2,DT}$, $w_{e_3,NB}$, and $w_{e_3,DT}$.

We begin by learning the two weights $w_{e_1,NB}$ and $w_{e_1,DT}$ for the mediated-schema element e_1. To do so, we create a set of training examples. Each example is of the form $(d, p_{NB}, p_{DT}, p*)$, where (a) d is a data instance from a training source, (b) p_{NB} and p_{DT} are the confidence scores of the naive Bayes learner and the decision tree learner, respectively, that d matches e_1 (i.e., d is an instance of e_1), and (c) $p*$ is 1 if d indeed matches e_1 and 0 otherwise.

We then apply linear regression to compute the weights $w_{e_1,NB}$ and $w_{e_1,DT}$ that minimize the squared error $\sum_d (p* - [p_{NB} * w_{e_1,NB} + p_{DT} * w_{e_1,DT}])^2$, where the term after $\sum_d$ ranges over all training examples.

We compute the remaining four weights $w_{e_2,NB}$, $w_{e_2,DT}$, $w_{e_3,NB}$, and $w_{e_3,DT}$ in a similar fashion.

5.9 Many-to-Many Matches

Until now we have focused on schema matches where all the correspondences are one-to-one. In practice, there are many correspondences that involve multiple elements of one schema or both. For example, consider the simplified SOURCE and TARGET schemas in Figure 5.8. The price attribute in the **Items** table corresponds to basePrice * (1 + taxRate) in the **Books** table; the author attribute in the **Items** table corresponds to concat(authorFirstName, authorLastName) in the **Books** table. Hence, we need to consider similarity also among *compound elements* of the two schemas. A compound element is constructed by applying a function to multiple attributes.

In this section we discuss how to extend schema matching techniques to capture many-to-many correspondences. The main challenge is that there are too many compound elements, and therefore too many candidate correspondences, to test. Whereas in the case of one-to-one matches, at worst we need to test similarity of the cross product of elements from both schemas, the number of compound elements is potentially very large or even unbounded. For example, there are many ways to concatenate multiple string attributes in

SOURCE
Books(title, basePrice, taxRate, quantity, authorFirstName, authorLastName)

TARGET
Items(title, price, inventory, author, genre)

FIGURE 5.8 Two simple schemas between which several many-to-many matches exist.

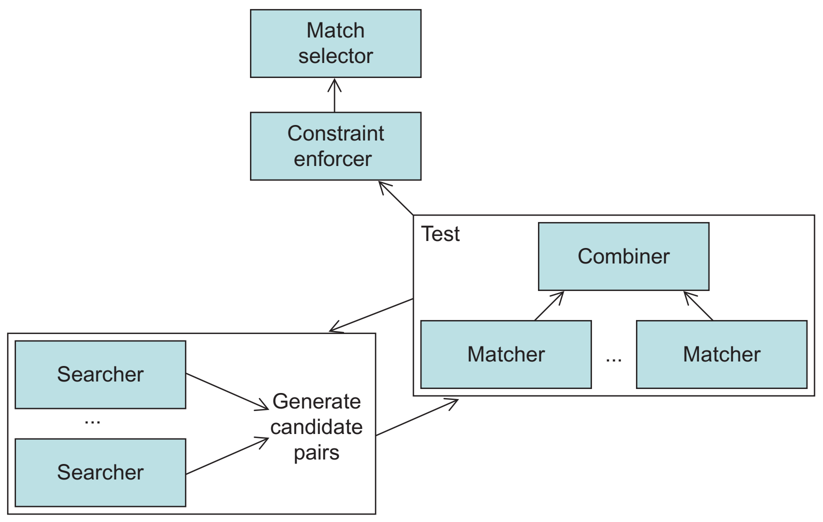

FIGURE 5.9 To discover correspondences among compound attributes, we employ several searchers to create candidate correspondences. We test them with the matchers and the combiner, as discussed earlier in the chapter. We terminate the searches when the quality of the predicted correspondences does not change significantly.

a schema, and there is an unbounded number of functions that involve multiple numerical elements of a schema.

We treat the problem of finding correspondences as a search problem (see Figure 5.9). We generate candidate element pairs, where one or both of them may be compound elements. We iteratively generate the pairs and test them as we test other candidate correspondences.

To generate candidate pairs, we employ several *specialized* searchers. Each specialized searcher focuses on a specific data type (e.g., text, string, numeric) and considers only compound elements appropriate for that type. Note that this strategy is very extensible. For example, it is easy to add another searcher that specializes in address fields and combinations thereof, or in particular ways of combining name-related fields.

A searcher has three components: a search strategy, a method for evaluating its candidate matches, and a termination condition. We explain each below.

SEARCH STRATEGY

The search strategy involves two parts. First, we define the set of candidate correspondences that a searcher will explore with a set of operators that the searcher can apply to build compound elements (e.g., concat, +, *). Second, the searcher needs a way to control its search, since the set of possible compound elements may be large or unbounded. A common technique is to use use *beam search*, which keeps the k best candidates at any point in the search.

EVALUATING CANDIDATE CORRESPONDENCES

Given a pair of compound elements, one from each schema, we evaluate their similarity using the techniques described previously in this chapter. As before, the result of the evaluation is a prediction on the correspondence between the pair. The type of searcher may dictate which matchers are more appropriate.

TERMINATION CONDITION

While in some cases the search will terminate because the set of possible compound elements is small, we need a method for terminating when the set is unbounded or too large. We terminate when we witness *diminishing returns*. Specifically, in the *i*th iteration of the beam search algorithm, we keep track of the highest score of any of the correspondences we have seen to that point, denoted by Max_i. When the difference between Max_{i+1} and Max_i is less than some prespecified threshold δ, we stop the search and return the top k correspondences.

Example 5.18

To discover many-to-many matches between the schemas SOURCE and TARGET in Figure 5.8, we employ two searchers. The first is a string searcher. The searcher considers attributes of type string, and tries to combine them by concatenation. The second is a numeric searcher that considers numerical fields and tries to combine them using arithmetic operations such as addition and multiplication, and some constants, such as 32, 5/9 (for Fahrenheit to Celsius), and 2.54 (for converting inches to centimeters).

When the algorithm looks for a match for the author attribute of Items, it will consider the elements title, authorFirstName, authorLastName, concat(title, authorFirstName), concat(authorFirstName, authorLastName), concat(authorLastName, authorFirstName), etc. When the algorithm looks for a match for the price attribute of Inventory, it tests many combinations including basePrice + taxRate, basePrice * taxRate, basePrice * (1 + taxRate), basePrice + quantity, and taxRate * quantity.

5.10 From Matches to Mappings

Recall from Section 5.1 that to create the descriptions of data sources, we often begin by creating semantic matches, then elaborate the matches into mappings. The reason we begin by creating the matches is that matches are often easier to elicit from designers, because they require the designer to reason about *individual* schema elements. As we have seen so far, there are also techniques that enable the system to guess matches.

In elaborating matches into mappings, the key challenge is to flesh out the matches and put all of them together into a coherent whole. This involves specifying the operations that need to be performed on the source or target data so they can be transformed from one to the other. In particular, creating the mapping requires aligning the tabular organization of the data in the source and the target by specifying joins and unions. It also involves specifying other operations on the data such as filtering columns, applying aggregates, and unnesting structures.

Given the complexity of creating mappings, the process needs to be supported by a very effective user interface. For example, the designer should be able to specify the mapping using a graphical user interface, and the system should generate the mapping expressions automatically, thus saving the designer significant effort. In addition, at every step the system should show the designer example data instances to verify that the transformation she is defining is correct and should allow her to correct errors when necessary.

This section describes how to explore the space of possible schema mappings. Given a set of matches, we describe an algorithm that searches through the possible schema mappings that are consistent with the matches. We show that we can define the appropriate search space based on some very natural principles of schema design. Furthermore, these principles also suggest methods for ranking the possible mappings when they are presented to a designer. We begin with an example that illustrates the intuition underlying these concepts.

Example 5.19

Figure 5.10 shows a subset of the schema match produced between a source university schema (on the left) and an accounting schema.

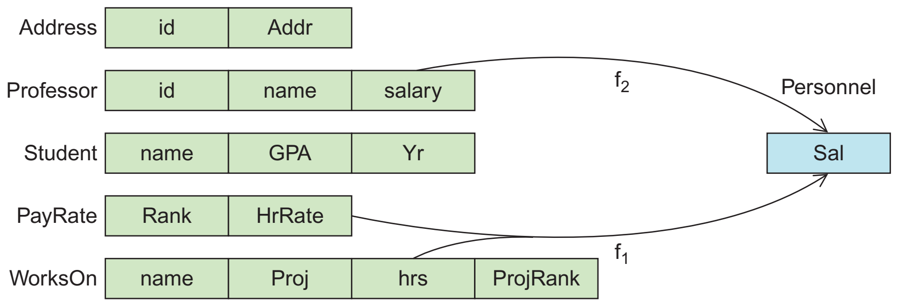

FIGURE 5.10 Combining correspondences from multiple tables. The correspondence f_1 states that the product of HrRate and hrs corresponds to Sal of **Personnel**, but the question is which tables to join to create a table that includes the two attributes. The correspondence f_2 states that salary also corresponds to Sal, and the question is whether to union professor salaries with employee salaries or to join salaries computed from the two correspondences.

The correspondence f_1 states that the product of HrRate from the **PayRate** relation and the Hrs attribute from the **WorksOn** relation corresponds to the Sal attribute in the target **Personnel** relation.

The schema mapping needs to specify how to join the relations in the source in order to map data into the target, and this choice is far from obvious. In the example, if the attribute ProjRank is a foreign key of the relation **PayRate**, then the natural mapping would be

```
SELECT P.HrRate * W.Hrs
FROM PayRate P, WorksOn
W WHERE P.Rank = W.ProjRank
```

However, suppose that ProjRank is not declared as a foreign key, and instead, the name attribute of **WorksOn** is declared as a foreign key of **Student** and the Yr attribute of **Student** is declared as a foreign key of **PayRate**. That is, the salary depends on the year of the student. In that case, the following join would be the natural one:

```
SELECT P.HrRate * W.Hrs
FROM PayRate P, WorksOn W, Student S
WHERE W.Name=S.Name AND S.Yr = P.Rank
```

If all three foreign keys are declared, then it is not clear which of the above joins would be the right one. In fact, it is also possible that the correct mapping does not do a join at all, and the mapping is a cross product between **PayRate** and **WorksOn**, but that seems less likely.

Suppose the correct mapping for the first correspondence, f_1, involves joining tables **PayRate**, **WorksOn**, and **Student**. Now consider the second correspondence, f_2, which states that the salary attribute of **Professor** maps to Sal in the target. One interpretation of f_2 is that the values produced from f_1 should be joined with those produced by f_2. However, that would mean that most of the values in the source database would not be mapped to the target. Instead, a more natural interpretation is that there are two ways of computing the salary for employees, one that applies to professors and another that applies to other employees. The mapping describing this interpretation is the following.

```
(SELECT P.HrRate * W.Hrs
 FROM PayRate P, WorksOn W, Student S
 WHERE W.Name=S.Name AND S.Yr = P.Rank)
UNION ALL
(SELECT Salary
 FROM Professor)
```

The above example illustrates two principles. First, while the space of possible mappings given a set of matches may be bewildering, there is some structure to it. In particular, the first choice we made considered how to join relations, while the second choice considered how to take the union of multiple relations. These two choices form the basis for the space of mappings we will explore in the algorithm that we will describe shortly.

The second principle illustrated by the example is that while the choices we made above seem to be of heuristic nature, they are actually grounded in solid principles from database design. For example, the fact that we preferred the simple join query when only one foreign key was declared is based on the intuition that we are reversing the data normalization performed by the designer of the source data when we map it to the target. The fact that we prefer the join mappings over the Cartesian product is based on the intuition that we do want to lose any relationships that exist between source data items. The fact that we prefer

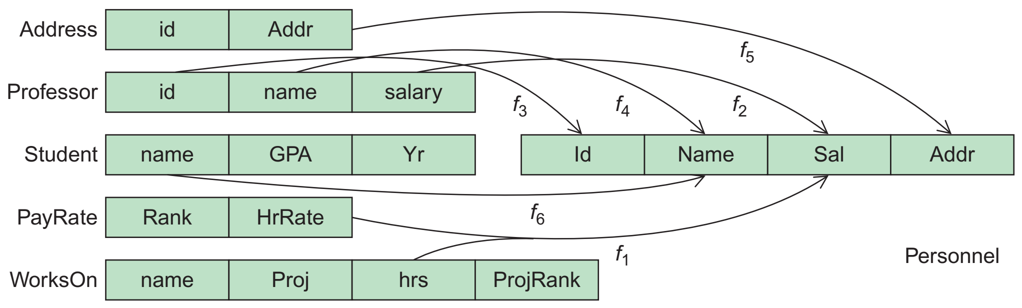

FIGURE 5.11 The full set of correspondences for Figure 5.10.

to union the data from **Professor** and **PayRate** is based on the intuition that every item from the source data should be represented *somewhere* in the target, unless it is explicitly filtered out. These principles will be the basis for ranking possible mappings in the algorithm.

Before we turn to the algorithm, we complete our example. Figure 5.11 shows our example with a few more correspondences:

f_3: $Professor(id) \approx Personnel(Id)$
f_4: $Professor(name) \approx Personnel(Name)$
f_5: $Address(Addr) \approx Personnel(Addr)$
f_6: $Student(name) \approx Personnel(Name)$

These correspondences naturally fall into two sets. The first set of f_2, f_3, f_4, and f_5 maps from **Professor** to **Personnel** and intuitively specifies how to create tuples for **Personnel** for professors. The second set of f_1 and f_6 specifies how to create tuples of **Personnel** for other employees. In the algorithm, each of these sets will be called a *candidate set*. The algorithm explores the possible joins within each candidate set, and then considers how to union the transformations corresponding to each candidate set.

The most intuitive mapping given these correspondences is the following:

```
(SELECT P.id, P.name, P.salary, A.Addr
 FROM Professor P, Address A
 WHERE A.id = P.id)
UNION ALL
(SELECT NULL AS Id, S.name, P.HrRate*W.hrs, NULL AS Addr
 FROM Student S, PayRate P, WorksOn W
 WHERE S.name = W.name AND S.Yr = P.Rank)
```

However, there are many other possible mappings, including the following one, which does not perform any joins at all and seems rather unintuitive:

```
(SELECT NULL AS Id,  NULL AS Name, NULL AS Sal, Addr
 FROM Address )
```

```
UNION ALL
(SELECT P.id, P.name, P.salary, NULL as Addr
 FROM Professor P)
UNION ALL
(SELECT NULL AS Id,  NULL AS Name, NULL AS Sal, NULL AS Addr
 FROM Student S)
...
```

Searching a Set of Possible Schema Mappings

We now describe an algorithm for searching the space of possible schema mappings given an input set of matches. For simplicity, we describe the algorithm for the case where the target consists of a single relation (as in our example). Extending the algorithm to the multi-relation case is left as a simple exercise for the reader.

The algorithm we describe is fundamentally an interactive one. The algorithm explores the space of possible mappings and proposes the most likely ones to the user. In the description of the algorithm we refer to several heuristics. These heuristics can be replaced by better ones if they are available. Though it is not reflected in our description, we assume that at every step the designer can provide feedback on the decisions made by the algorithm, therefore steering it in the right direction.

The input is specified as a set of correspondences: $M = \{f_i : (\bar{A}_i \approx B_i)\}$, where $\bar{A}_i$ is a set of attributes in the source, S_1, and B_i is an attribute of the target, S_2. We also allow the matches to specify filters on source attributes. The filters can specify a range restriction on an attribute or on an aggregate of an attribute (e.g., avg, sum, min). Algorithm 9, the FindMapping algorithm, operates in four phases.

In the first phase, we create all possible *candidate sets*, which are subsets of S_2 where each attribute of M is mentioned at most once. We denote the set of candidate sets by $\mathcal{V}$. Intuitively, a candidate set in $\mathcal{V}$ represents one way of computing the attributes of S_2. Note that sets in $\mathcal{V}$ do not need to cover all attributes of S_2, as some may be not be computed by the ultimate mapping. If a set does cover all attributes of S_2, then we call it a complete cover. Also note that the elements of $\mathcal{V}$ need not be disjoint—the same correspondence can be used in multiple ways to compute S_2.

Example 5.20

Suppose we had the correspondences

$$f_1 : S1.A \approx T.C, f_2 : S_2.A \approx T.D, f_3 : S2.B \approx T.C$$

Then the complete candidate sets are $\{\{f_1, f_2\}, \{f_2, f_3\}\}$. The singleton sets $\{f_1\}$, $\{f_2\}$, and $\{f_3\}$ are also candidate sets.

Algorithm 9. FindMapping: Searches through possible schema mappings.

Input: correspondences between schemas S_1 and S_2, $M = \{f_i : (\bar{A}_i \approx B_i)\}$ (when $\bar{A}_i$ has more than one attribute, it is of the form $g(\bar{A}_i)$ where g is the function that combines the attributes); *filter*$_i$: the set of filters associated with f_i. **Output:** mapping in the form of a query.

```
{Phase 1: Create candidate sets}
Let $\mathcal{V} := \{v \subseteq M \mid v$ does not mention an attribute of $S_2$ more than once$\}$
{Phase 2:}
Let $\mathcal{G} := \mathcal{V}$
for every $v \in \mathcal{V}$ do
  if $(\bar{A}_i \approx B_i) \in v$ and includes attributes from multiple relations in $S_1$ then
    use Heuristics 1 and 2 to find a single join path connecting the relations mentioned in $\bar{A}_i$
    if there is no join path then
      remove $v$ from $\mathcal{G}$
    end if
  end if
end for
{Phase 3:}
Let Covers := $\{\Gamma \mid \Gamma \subseteq \mathcal{G}, \Gamma$ mentions all $f_i \in M$ and no subset of $\Gamma$ mentions all $f_i \in M\}$
Let selectedCover := the cover $c \in$ Covers with the fewest candidate paths
if Covers has more than one cover then
  select one with Heuristic 3
end if
{Phase 4:}
for each $v \in$ selectedCover do
  create a query $Q_v$ of the following form:
      SELECT vars
      FROM $t_1, \ldots, t_k$
      WHERE $c^1, \ldots, c^j, p_1, \ldots, p_m$
  where
      vars are the attributes mentioned in the correspondences in $v$
      $t_1, \ldots, t_k$ are the relations of $S$ in the join paths found for $v$
      $c^1, \ldots, c^j$ are the join conditions for the join paths in $v$
      filter$_1, \ldots,$ filter$_m$ are the filters associated with the correspondences in $v$
end for
return the query
    $Q_1$ UNION ALL ... UNION ALL $Q_b$
    where $Q_1, \ldots, Q_b$ are the queries created above
```

In the second phase of the algorithm we consider the candidate sets in $\mathcal{V}$ and search for the best set of joins within each candidate set. Specifically, consider a candidate set $v \in \mathcal{V}$ and suppose $(\bar{A}_i \approx B_i) \in v$, and that $\bar{A}_i$ includes attributes from multiple relations in

S_1. We search for a join path connecting the relations mentioned in $\bar{A}_i$ using the following heuristic.

Heuristic 1 (**Finding a Join Path**). A join path can be

- a path through foreign keys,
- a path proposed by inspecting previous queries on S, or
- a path discovered by mining the data for joinable columns in S. □

The set of candidate sets in $\mathcal{V}$ for which we find join paths is denoted by $\mathcal{G}$. When there are multiple join paths (as in Example 5.19), we select among them using the following heuristic.

Heuristic 2 (**Selecting of Join Paths**). We prefer paths through foreign keys. If there are multiple such paths, we choose one that involves an attribute on which there is a filter in a correspondence, if such a path exists. To further rank paths, we favor the join path where the estimated difference between the outer join and inner join is the smallest. This last heuristic favors joins with the least number of dangling tuples. □

The third phase of the algorithm examines the candidate sets in $\mathcal{G}$ and tries to combine them by union so they cover all the correspondences in M. Specifically, we search for *covers* of the correspondences. A subset Γ of $\mathcal{G}$ is a cover if it includes all the correspondences in M and it is minimal, i.e., we cannot remove a candidate set from Γ and still obtain a cover.

In Example 5.20, $\mathcal{G} = \{\{f_1, f_2\}, \{f_2, f_3\}, \{f_1\}, \{f_2\}, \{f_3\}\}$. Possible covers include $\Gamma_1 = \{\{f_1\}, \{f_2, f_3\}\}$ and $\Gamma_2 = \{\{f_1, f_2\}, \{f_2, f_3\}$. When there are multiple possible covers, we select one using the following heuristic.

Heuristic 3 (**Selecting of the Cover**). If there are multiple possible covers, we choose the one with the smallest number of candidate sets with the intuition that a simpler mapping may be more appropriate. If there is more than one with the same number of candidate sets, we choose the one that includes more attributes of S_2 in order to cover more of that schema. □

The final phase of the algorithm creates the schema mapping expression. Here we describe the mapping as an SQL query. The algorithm first creates an SQL query from each candidate set in the selected cover, and then unions them.

Specifically, suppose v is a candidate set. We create an SQL query Q_v as follows. First, the attributes of S_2 in v are put into the `SELECT` clause. Second, each of the relations in the join paths found for v are put into the `FROM` clause, and their respective join predicates are put in the `WHERE` clause. In addition, any filters associated with the correspondences in v are also added to the `WHERE` clause. Finally, we output the query that takes the bag union of each of the Q_v's in the cover.

Bibliographic Notes

The areas of schema matching and schema mapping have been the topic of research for decades. The survey [487] summarizes the state of the art circa 2001 and covers some of the early work such as [68, 111, 434, 465, 472]. The survey [179] and special issues [180, 463] discuss work up to 2005, while the book [61] discusses work up to 2010 (see also `http://dbs.uni-leipzig.de/file/vldb2011.pdf` for a presentation that summarizes recent work). Other books on the topic include [214, 241].

The architecture presented in this chapter is based on the LSD system [181] and the COMA system [178] (see also [371, 511] for further details). The algorithms described often combine different components of the architecture we describe, but do not tease them apart. For example, the Similarity Flooding algorithm [427] used string-based similarity to give initial values to correspondences, and then used a local propagation algorithm to spread the estimates across the graph.

The work [140] exploits multiple heuristics to match schemas. The work [178, 181] introduced the idea of leveraging the combination of multiple matchers in a schema matching system. A similar idea was proposed in [212]. Applying machine learning techniques to schema matching was introduced in [384]. The LSD system [181] introduced the use of multiple base learners and combining them via learning. The system used a technique called stacking [549, 576] to combine the predictions of the base learners. Further explanation of the naive Bayes learner and an analysis for why it works well in practice can be found in [187]. More background on learning techniques and, in particular, rule-based learning can be found in [438]. The paper [570] presents a unified approach to schema matching and data matching, using a machine learning technique called conditional random field.

The LSD system introduced the idea of learning from past matches. The COMA system [178] considered a simple form of reusing past matches (without learning from them), and also considered composing multiple past matches. The paper [339] uses information-theoretic techniques to match schemas. The papers [206, 208] utilize query logs for hints about related schema elements (e.g., in join clauses).

Corpus-based schema matching [406] showed how to leverage a corpus of schemas and mappings in a particular domain to improve schema matching. The paper [295] looks at statistical properties of collections of Web forms in order to predict a mediated schema for a domain and mappings from the data sources to the mediated schema. The paper [581] discusses how to match a corpus of schemas via interactive clustering. The paper [510] discusses ranking of clustering alternatives based on probabilistic mappings.

Several schema matching systems considered how to apply domain constraints to prune and improve schema matches [181, 183, 407, 427]. The neighborhood-adaptation algorithm we described in Section 5.6 is adapted from [407], and the A* algorithm is from [181]. The paper [183] describes a relaxation labeling algorithm that exploits a broad range of domain constraints to match taxonomies.

User interaction in schema matching systems and incremental schema matching were discussed in [78, 181, 222, 240, 581]. The papers [420, 421] discuss a crowdsourcing approach to schema matching.

The iMap system [372] addressed the problem of searching for complex mappings. Other works that find complex matches with subsumption relationships and with conditionals include [92, 251]. Techniques to match large schemas are discussed in [25, 43, 289, 306]. Gal [240] discusses the complexity of finding the top-k schema matches.

Schema matching systems typically come with myriad knobs, and tuning these knobs is well-known to be difficult. Such tuning issues are discussed in [62, 201, 371, 511].

The paper [410] discusses how the task of building a deep Web crawler can use semantic matches "as is," without having to elaborate them into semantic mappings.

The Clio system was the first to look at the problem of generating schema mappings given schema matches [302, 432]. The algorithm we described in Section 5.10 and the running example there are taken from [432]. These references also describe how to extend the algorithm to mappings with aggregation and how to create illustrative examples of the instance data that can help the designer find the correct mapping. Subsequent work on the topic includes [21, 23, 24, 30, 220, 254].

Finally, the related issue of ontology matching and mapping has also received significant attention. See [183] for an early system that employs an architecture that is similar to the architecture proposed in this chapter to match ontologies. The book [214] discusses the state of the art up to 2007, and the Web site `http://ontologymatching.org` contain a wealth of current information on this active area.