Chapter 7

Creating Semantic Mappings

In Chapter 5 we described formalisms for specifying source descriptions and algorithms that use these descriptions to reformulate queries. This chapter describes how to *create* these source descriptions when faced with a particular data integration application.

In practice, quite a bit of the effort of setting up a data integration application involves creating the semantic mappings and maintaining them over time. The reason it is a hard and often error-prone task is that creating the mappings requires 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, if legacy data sources are involved. 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 schemas.

This chapter describes a set of techniques that helps a designer (or set of designers) create semantic mappings and source descriptions. 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 a correct mapping between them. Our goal is thus to create tools that significantly *reduce* the time it takes to create semantic mappings. Often, much of the work involved in creating schema mappings is tedious and repetitive. Therefore, a schema mapping tool, based on a set of heuristics, can enable a designer to speed through the tedious parts, and provide useful suggestions when guiding her through the semantically challenging parts.

DVD Vendor Schema:

Product	**Movie**	**Locations**
title	title	name
productionYear	year	taxRate
releaseDate	director	shippingCharge
basePrice	directorOrigin	
listPrice	mainActors	
rating	genre	
customerReviews	awards	
saleLocation		

Online Aggregator Schema:

Item	
title	director
year	price
classification	starring
genre	avgReviews

Figure 7.1: Example of two database schemas. The schema on the top belongs to a DVD vendor, while the schema on the bottom belongs to a shopping site that aggregates products from multiple vendors. Our goal is to semi-automatically create a schema mapping between the two schemas.

7.1 The Schema Mapping Challenge

We begin with an example that illustrates the challenges involved in automatically creating schema mappings. Our discussion of schema mapping is not specific to the virtual data integration architecture. We assume the input is a pair of schemas, and our goal is to create a mapping between them.

Figure 7.1 shows two example schemas. The schema on the top includes three tables and belongs to a vendor of DVDs. The schema models the product aspects of the DVD in one table, while the details of the movie itself are in another table. The third table models location-specific information that is needed to charge the correct overall price. The schema on the bottom belongs to a shopping aggregation site. Unlike the individual supplier, the aggregator is not interested in all the details of the product or how the price is computed, but only in attributes that are shown to its customers.

A schema mapping system needs to reconcile heterogeneity between disparate schemas. As we see in the example schemas, semantic heterogeneity arises in multiple

forms. First, 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 refer to the same underlying concept, as do the attributes mainActors and starring. 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 are different. The aggregator only needs a single table, while 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 awards that 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.

In order to correctly map between schemas, the schema mapping system needs to understand the intended semantics of the schema. However, the schema itself does not completely describe its full meaning. The schema is a set of symbols that represent some model that was in the mind of the designer, but does not fully capture it. For example, in the first schema, we have the attributes basePrice and listPrice, but the precise meaning of these attributes is not clear from their names (nor is it clear which price is the one charged to the customer). 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. Hence, to perform adequately, the schema mapping system must glean enough of the semantics of the schema from its formal specification and any other clues that may accompany it, such as text descriptions and the way the schema is used.

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 is used for the same purposes. But, as our example 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, movie goers, coffee consumers, etc. 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, and 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, exchanging data is critical for business processes, and there is strong incentive to agree on them. Examples include data that needs 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 at the schema level as well as 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 8.

7.2 Overview of the Schema Mapping Task

Broadly speaking, we identify two main steps in creating schema mappings. In the first step, often referred to as *schema matching*, we create *correspondences* between elements of the two schemas. For example, we would specify that the title attribute in the vendor schema corresponds to the title attribute in the aggregator schema, while rating in the vendor schema corresponds to classification in the aggregator schema. A correspondence between two attributes means that they convey the same information about the underlying entities, but it does not specify the exact transformation between data in one schema to data in the other.

In the second step, we create the schema mappings from the correspondences and fill in the missing details. For example, in this step we specify that in order to compute the value price in the aggregator schema, we need to join the **Product** table

with **Locations** table on the attribute saleLocation = name, and add the appropriate local tax given by the attribute taxRate.

The main reason that schema mapping is split into these two steps is that correspondences specify relationships between the two schemas at the level of individual schema elements, and are therefore easier to elicit from designers. In addition, there are applications in which schema matches suffice. For example, if the two schemas include a single table, then a schema match is already a mapping and no further elaboration is needed. We now outline the different components of each step.

7.2.1 Schema matching

The goal of a schema matcher is to produce a set of correspondences between two schemas. One solution is to provide the designer a convenient graphical user interface to specify correspondences manually, but our goal is to *automatically* create such correspondences that are then validated by a designer.

Suppose we are given two schemas S_1 and S_2. We use the term *schema elements*, or *elements*, to collectively refer to the table names and attribute names in the schema. In the most general form, a correspondence, $(\bar{A} \rightarrow \bar{B})$, states that a set of elements $\bar{A}$ in S_1 maps to a set of attributes $\bar{B}$ in S_2. The correspondence does not specify the exact details of the mapping, only that the corresponding elements refer to the same attribute or relation in the real world.

The most common correspondences in practice are 1-1 correspondences, in which $\bar{A}$ and $\bar{B}$ are singletons. When either $\bar{A}$ or $\bar{B}$ are more than singletons, the correspondence is called *many-to-one* or *many-to-many*. A *schema match* is a set of correspondences, and it is called a 1-1 schema match if all of its correspondences are 1-1.

Example 7.1: In our example, the schema match would include the following correspondences. In all of them, the target relation is **Item**. If a correspondence has only one attribute on the left or right hand side, we omit the set braces.

Product.title → title
Movie.year → year
Movie.genre → genre
Movie.director → director
Product.rating → classification
Movie.mainActors → starring
{**Product.**basePrice, **Locations**.taxRate} → price
Product.customerReviews → avgReviews

□

Since the process of creating schema matches is inherently heuristic, we often associate a confidence measure, a number between 0 and 1, with a correspondence. In fact, the schema matcher may output several top candidate matches. In addition, when appropriate we may associate a filter with a correspondence. For example, the correspondence {**Product.**basePrice, **Locations**.taxRate} → price may apply only to locations in the U.S.

The first observation from our example is that no *single* heuristic is guaranteed to yield accurate correspondences. One set of heuristics is based on examining the names of the schema elements. While exact equality between two element names (e.g., for title) is a strong indication of correspondence, it is not guaranteed to be correct. More often we need to consider similarities between element names (e.g., productionYear and year or customerReviews and avgReviews).

In addition to element names, we can glean important hints from the data values when they are available. For example, rating and classification attributes are likely to have the same set of values repeating (e.g., G, PG-13, R, NC-17). Other clues we may consider are the proximity of attributes to each other or how attributes are used in queries.

The variety of available clues motivates the architecture of a schema matcher, shown in Figure 7.2. The schema matcher employs several *base matchers* and then combines their predictions. We describe several base matchers in Section 7.3 and techniques for combining them in Section 7.4.

In addition to matching heuristics, domain knowledge plays an important role in pruning candidate schema matches. For example, knowing that a movie typically has a single director but multiple actors can help us guess that mainActors maps to starring and not to director. Hence, the third component of the schema matcher, described in Section 7.5, describes how to enforce constraints on the candidate schema matches. We note that domain knowledge can also play a key role in the second step of creating schema mappings, after the correspondences have been specified.

The first three phases of the schema matcher produce a *correspondence matrix*, which specifies a prediction for the correspondence between every pair of elements in S_1 and S_2. The match selector, described in Section 7.6, takes the similarity matrix as input and produces a schema match, or several candidate schema matches. The match selector needs to choose the best matching element in S_2 for each element in S_1, also keeping global constraints in mind.

As noted earlier, schema matching tasks are often repetitive. Section 7.7 describes a set of methods based on machine learning techniques that enable the schema matcher to learn from previous experience, and therefore reuse previous work. Section 7.8 describes how to discover many-to-many correspondences, which require slightly different treatment.

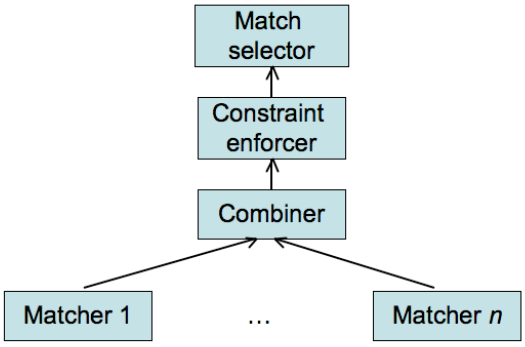

Figure 7.2: Components of a schema matcher. The matcher employs a collection of basic matchers, each of which predicts correspondences based on cues available in the schema and the data. The predictions of the basic matchers are then combined into a single similarity matrix. The matcher then applies domain knowledge and other constraints to prune the possible matches. Finally, the match selector produces the best match (or set of matches) from the similarity matrix.

7.2.2 From correspondences to mappings

Once the matches are 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 **Item** table, we need to join **Product** 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 **Product** with **Movie** to obtain the origin of the director, and compute the price based on the taxes in director's country of birth.

In Section 7.9, we describe how the 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.

7.3 Basic Matchers

The input to a base matcher is a pair of schemas S_1 and S_2, whose elements are $\bar{A}$ and $\bar{B}$, respectively. In addition, the base matcher may consider any other surrounding information that may be available, such as data instances and text descriptions. A basic matcher outputs a correspondence matrix that assigns to every pair of elements (A_i, B_j) a number between 0 and 1 predicting whether A_i corresponds to B_j.

There is a plethora of techniques for guessing matches between schema elements, each based on a different heuristic. In this section we describe a few common basic matchers that apply across many domains. We describe two classes of basic matchers: those that are based on comparing names of schema elements and others that are based on inspecting data instances. It is important to keep in mind that for specific domains it is often possible to develop more specialized and effective matchers. Furthermore, instance-based techniques will typically require looking at a significant amount of data and will therefore be slower than the schema-level techniques. In practice, they can be implemented efficiently and they improve precision significantly.

7.3.1 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 never written in the exact same way, the challenge is to find effective distance measures that reflect the distance of element meanings. In principle, any of the techniques for string matching described in Chapter 6, 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 to express their meanings. Hence, it is typically useful to normalize the element names before applying the distance measures. Normalization replaces a single token by several tokens that can be compared using the above techniques. Several possible normalization procedures are described below. Note that some of them require domain-specific dictionaries.

- split names according to certain delimiters, such as capitalization, numbers or special symbols. For example, AgentAddress1 would be split to Agent, Address and 1.
- expand known abbreviations or acronyms. For example cust may be expanded to customer

- expand a string with its synonyms. For example, add cost to price.
- expand a string with its hypernyms. For example, product can be expanded to book, dvd, cd.
- remove articles, propositions and conjunctions, such as in, at, and.

With all these 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 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.

7.3.2 Instance-based matchers

Data instances often convey the meaning of a schema element more than its name. We now describe techniques for predicting correspondences based on analyzing data values. Keep in mind that not all schema matching scenarios have access to instance data.

Inferring common types

The first technique is to develop a set of rules that infer common types from the format of the data values. This technique is effective for types such as phone numbers, prices, person names, location names (assuming the availability of a database of locations) and zip codes. In specific domains, we can apply additional recognizers, such as element names in Chemistry or gene names in Biology.

Value overlap

The second technique is based on measuring the overlap of values appearing in the two elements, and applies to fields whose values are drawn from some finite domain, such as movie ratings and country name. The Jaccard Coefficient is a commonly used measure, and is defined to be the fraction of the values for the two elements

that can be an instance for both of them. The Jaccard Coefficient can also be given a probabilistic interpretation: it is the conditional probability of a value being an instance of both elements, given that it is an instance of one of them.

Formally, if $Pr(e)$ is the probability of a random value being an instance of element e, and $D(e)$ is the set of values for the element e, then the Jaccard Coefficient is defined as follows:

$$jaccardSim(e_1, e_2) = Pr(e_1 \cap e_2 \mid e_1 \cup e_2) = \frac{||D(e_1) \cap D(e_2)||}{||D(e_1) \cup D(e_2)||} \tag{7.1}$$

The Jaccard Coefficient ranges from 0 to 1. It is 0 when the two elements have disjoint sets of values and 1 when the set of values exactly coincides.

Text-field analysis

The third technique applies to elements whose values are longer texts. Unlike categorical elements, their values vary quite drastically. For example, imagine trying to compare between text fields for house descriptions in a real-estate schema. In such cases, the probability of finding the exact string for both elements is very low, and therefore $D(e_1) \cap D(e_2)$ is likely empty. As a result the Jaccard Coefficient would not provide a useful distance measure.

Intuitively, we would like to compare the general topics these text fields are about, rather than the exact wording in the fields. We could do that by comparing the *characteristic* words in the texts of the two elements. For example, the distribution of words occurring in house descriptions (e.g., fantastic, remodeled, kitchen) will be different than the distribution of words in movie reviews (e.g., cinematography, acting, plot).

A common method to quantify the distribution of words is using *text classifiers*. We will discuss classifiers in more detail in Section 7.7, but we give a brief description here. A classifier for a concept C is an algorithm that identifies instances of C from those that are not. For example, a classifier may be able to recognize phone numbers or addresses from other strings. To make this prediction, the classifier creates an internal model of the concept based on *training examples*: positive examples that are known to be instances of C and negative examples that are known not to be instances of C. Given an example c , the classifier applies its model to decide whether c is an instance of C or not. Such a prediction is typically associated with a probability computed from the internal model. For a text classifier, all the examples are text strings. Popular examples of text classifiers are the Naive Bayes Classifier and Support Vector Machines.

To apply the above technique given two elements, e_1 and e_2, we create two text classifiers C_1 and C_2 as follows. The classifier C_1 will identify instances of e_1, and the classifier C_2 will identify instances of e_2. The positive examples for C_1 are values of e_1 and the positive examples for C_2 are values of e_2. In both cases, the negative examples are values taken from other elements.

After computing C_1 and C_2 we apply them to the values of the other elements. Specifically, let $D_2(e_1)$ be the set of values of e_1 that were deemed to be instances of C_2, and let $D_1(e_2)$ be the set of values of e_2 that were deemed to be instances of C_1. We estimate the similarity between e_1 and e_2 with the following formula, which also has a probabilistic interpretation similar to the Jaccard Coefficient:

$$textSim(e_1, e_2) = Pr(e_1 \cap e_2 \mid e_1 \cup e_2) = \frac{||D_2(e_1)|| + ||D_1(e_2)||}{||D(e_1)|| + ||D(e_2)|| - (||D_2(e_1)|| + ||D_1(e_2)||)}. \quad (7.2)$$

Note that if the values of e_1 and e_2 are very similar to each other, then Formula 7.2 will tend towards 1. Finally, we note that text-field analysis can also be applied to textual descriptions of schema elements when they are available.

7.4 Combining Match Predictions

The results of the basic matchers can be summarized in a *similarity cube*. Specifically, suppose the schema matcher used l basic matchers to predict correspondences between the elements $A_1, \ldots, A_n$ of S_1 and the elements $B_1, \ldots, B_m$ of S_2. The similarity cube assigns each triple (b, i, j) a number between 0 and 1 that describes base matcher b's prediction about the correspondence between A_i and B_j.

The goal of the combination step is to produce a *similarity matrix* that combines the predictions of the base matchers. That is, for every pair (i, j) we want a value between 0 and 1, $Combined(i, j)$, that describes a single prediction about the correspondence between A_i and B_j.

Two natural combinations to consider are Max and Average. That is, we can define

$$Combined(i, j) = \max_{b=1}^{l} Basic(b, i, j)$$

or

$$Combined(i, j) = \frac{1}{l} \sum_{b=1}^{l} Basic(b, i, j).$$

The Max combination function is preferable when we trust a strong signal from the basic matchers. That is, if a matcher outputs a high value, then we are relatively

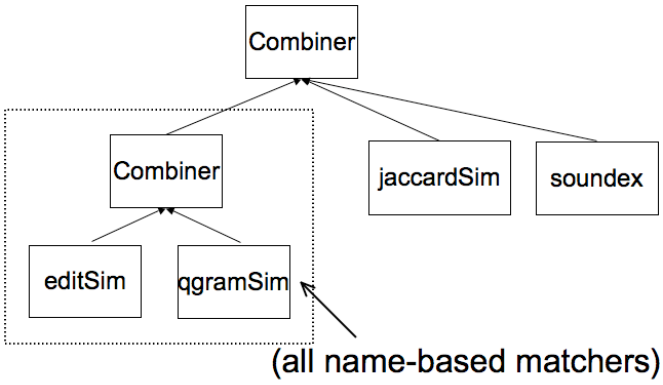

Figure 7.3: Multi-step combination of basic matchers. Here we first average the predictions of the name-based matchers and then combine their prediction with the Jaccard similarity and Soundex similarity.

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. On the other hand, if we are not always confident of strong signals, then the average is a natural combination function.

We can also consider multi-step combination functions. For example, as shown in Figure 7.3, we may want to first aggregate all the base matchers that consider the schema name, and only then combine them with other matchers. If a certain set of matchers leverages the same or similar cues from the schemas, then their predictions may not be independent of each other. In such a case, we may want to first aggregate them before combining their prediction with other basic matchers.

A generalization of the above method is to give *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. For example, for elements that have text values we may want to give $textSim$ a higher weight than $editSim$, but for numeric fields we may want $jaccardSim$ to have a higher value than both. In Section 7.7 we describe techniques that *learn* appropriate weights by inspecting the behavior of the schema matching system.

7.5 Applying Constraints and Domain Knowledge

Book	**Item**
ISBN	code
publisher	name
pubCountry	brand
title	origin
review	desc

Inventory	**InStore**
ISBN	code
quantity	availQuant
location	

	Constraint	**Cost**
T_1	if $A \rightarrow$ **Item.**name, then A is a key.	∞
T_2	if $sim(A_1, B_1) \approx sim(A_2, B_1)$, A_1 is next to A_3 and B_1 is next to B_2 and $sim(A_3, B_2) > 0.8$ then match A_1 to B_1.	2
T_3	Average(length(description)) ≈ 20.	1.5
T_4	if $\|\|\{A_i \in attributes(R_1) \mid \exists B_j \in attributes(R_2)$ s.t. $sim(A_i, B_j) > 0.8\}\| \geq \|\|attributes(R_1)\|\|/2$ then match table R_1 to table R_2.	1

Figure 7.4: A schema with integrity constraints.

The basic matchers we described in Section 7.3 employ techniques that apply across many domains. Often, we have domain-specific knowledge that can be helpful in the process of schema matching. It is typically more natural to express this knowledge as a set of *constraints* that enable pruning certain candidate matches, rather than as another basic matcher that produces additional predictions. In fact, there are also domain-independent heuristics that are more fit for pruning matches than to producing them.

In this section we describe the kinds of constraints that can be applied in schema matching and two algorithms for enforcing them on the similarity matrix. We distinguish between *hard constraints* and *soft constraints*. Hard constraints must be applied. The schema matcher will not output any match that violates them. Soft constraints are of more heuristic nature, and may actually be violated in some schemas. Hence, the schema matcher will try to minimize the number (and weight) of the soft constraints being violated, but may still output a match that violates one or more of them. Formally, we attach a *cost* to each constraint T, denoted by $cost(T)$. For hard constraints, the cost is ∞, while for soft constraints, the cost can be any positive number.

Example 7.2: Figure 7.4 shows an example pair of schemas 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 particular domains.

The constraint T_1 states that any attribute mapped to name in **Item** must be a key. T_1 is a domain-dependent hard constraint.

The constraint T_2 is a domain-independent soft constraint. The constraint shows how to leverage proximity of attributes in a schema to arbitrate between two possible matches. Specifically, if two attributes A_1 and A_2 are both very similar to B_1, but one of them (A_1) is close to attribute A_3 that is similar to B_2, and B_2 is close to B_1, then we prefer to match A_1 to B_1.

The constraint T_3 is a domain-dependent soft constraint. The constraint states that the average length of a description field is typically around 20 characters. It can be used to ensure that fields mapped to desc are also approximately of the same length.

Finally, T_4 is a domain-independent soft constraint. It states that if more than half of the attributes of table R_1 match those of table R_2, then R_1 corresponds to R_2. □

The exact formalism in which constraints are represented is unimportant. The only requirement we impose is that given a constraint T and a schema match M, we must be able to decide whether M violates T.

We now describe two algorithms for applying constraints to the similarity matrix. The first algorithm is an adaptation of A* search, and guarantees to find the optimal solution but is computationally more expensive. The second algorithm, that applies only to constraints in which a schema element is affected by its neighbors (e.g., T_2 and T_4) is faster in practice, but could get stuck in a local minimum.

7.5.1 Applying constraints with A* search

The A* algorithm takes as input the domain constraints $T_1, \ldots, T_l$ and the similarity matrix *Combined*, produced by the match combiner. The similarity matrix for our example is shown in Table 7.1. The algorithm searches through the possible schema matches and returns the one that has the lowest cost. The cost is measured by the (1) likelihood of the schema match and (2) the degree to which the match 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

attribute	**Item**	**code**	**name**	**brand**	**origin**	**desc**	**InStore**	**code**	**availQuant**
Book	**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

Table 7.1: The *Combined* similarity matrix for Example 7.4

state. A* performs a *best-first* search: starts with the initial state, and always expands the state with the smallest *estimated cost.*

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 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 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.

Intuitively, in our context, a state corresponds to a partial schema match, where some of the attributes of S_1 have correspondences and others do not. The initial state is an empty schema match, and a a goal state is a complete schema match. As before, the description of the algorithm assumes that we are trying 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 i-th element either specifies a correspondence for A_i, or is a wildcard *, representing that the correspondence for A_i is undetermined. Thus, a state *partially* specifies a schema match, and can be interpreted as representing the *set* of matches. For example, suppose $n = 5$ and $m = 3$, then state $(B_2, *, B_1, B_3, B_2)$ represents three states $(B_2, B_1, B_1, B_3, B_2)$,

$(B_2, B_2, B_1, B_3, B_2)$, and $(B_2, B_3, B_1, B_3, B_2)$. We refer to a state as an *abstract state* if it contains wildcards. There is an edge from state s_1 to state s_2 if s_2 is a refinement of s_1, i.e., s_2 replaces one of the wildcards in s_1 by a specific attribute of S_2.

Initial state: The initial state is defined to be $(**\ldots*)$, which represents all possible schema matches.

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

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

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

$$cost(M) = -LH(M) + cost(M, T_1) + cost(M, T_2) + \cdots + cost(M, T_v). \tag{7.3}$$

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

The term $LH(M)$ is defined as $log\ conf(M)$, where $conf(M)$ is the confidence score of candidate match M. We compute the confidence as the product of the appropriate cells in the similarity matrix. If $M = (B_{l_1}, \ldots, B_{l_n})$, then

$$conf(M) = Combined(1, l_1) * \cdots * Combined(n, l_n).$$

The formula for $conf(M)$ effectively assumes that the probabilities of the correspondences in M are independent of each other. This assumption is clearly not true, because in many cases the appropriate correspondence for one attribute depends on the correspondence 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 7.3 implies that we prefer the combination with the highest confidence score, all other things being equal.

Example 7.3: Consider the goal state corresponding to the following schema match (highlighted in boldface in Table 7.1).

Book → **Item**	**Inventory** → **InStore**
Book.ISBN → **Item.**code	publisher → brand
pubCountry → origin	title → name
review → desc	quantity → availQuant
	Inventory.ISBN → **InStore.**code

The cost of this state is the sum of the likelihood of the schema match and the degree to which the schema match violates the integrity constraints. The likelihood of the match is the negative log of the product of the boldface values in Table 7.1. This particular match satisfies all the integrity constraints and therefore incurs no additional cost.

In contrast, consider the match in which pubCountry → origin is replaced by location → origin, and review → desc is replaced by title → desc. This match would violate two integrity constraints, T_2 and T_3. Hence, this match 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 the 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 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 7.4: Consider the abstract state, s_1, with the following partial match.

Book → **Item**	
ISBN → code	publisher → brand
pubCountry → origin	title → name
review → desc	

Given the similarity values in Figure 7.1, the cost from the initial state to s_1 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 $h_2(S)$, as follows. First, we compute $h_1(s)$, which is lower bound on the cost of expanding each of the wildcards in s. For example, suppose that $s = (B_1 B_2 * * B_3)$, then the estimated cost $h_1(s)$ is

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

That is, $h_1(n)$ is a lower bound on the cost of expanding all wildcards of s.

Example 7.5: Continuing with Example 7.4, 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., $-log(0.6 * 0.7 * 0.9 * 0.7)$. □

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 $\Sigma_{i=1}^{k} cost(s, T_i)$, where $cost(s, T_i)$ is the estimate for the constraint T_i. To estimate the degree to which a partial schema match violates a constraint, the algorithm assumes a best-case scenario – if it cannot show that all goal states reachable from s violate T_i, then it assumes that T_i is not violated. In Example 7.4, the partial match is a prefix of one that does not violate the constraints, and therefore $h_2(s_1) = 0$.

The specific method for determining whether the possible goal states violate T_i depends on the type of constraint of T_i. For example, consider $s = (B_1 B_2 * * B_3)$ and the hard constraint $T =$ "at most one attribute matches B_2". Clearly there are goals states that are represented by s and satisfy T, such as goal state $(B_1 B_2 B_3 B_1 B_3)$, and therefore we say that s satisfies T. Consider $s' = (B_1 B_2 * * B_2)$. Since any goal state represented by s' violates the hard constraint T, we set $cost(s', T) = \infty$.

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.

7.5.2 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 7.6: In Figure 7.4, constraints T_2 and T_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 T_2 can be stated as follows. If $sim(A_1, B_1) \leq 0.9$ and A_1 has a neighbor A_2 such that $sim(A_2, B_2) > 0.8$, and B_1 is a neighbor of B_2, then increase $sim(A_1, B_1)$ by α.

Constraint T_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. Then, if $sim(R_1, R_2) \leq 0.9$ and at least half of the neighbors of R_1 are in the set $\{A_i \in attributes(R_1) \mid \exists B_j \in attributes(R_2)$ s.t. $sim(A_i, B_j) > 0.8\}$, then increase the similarity $sim(R_1, R_2)$ by α. □

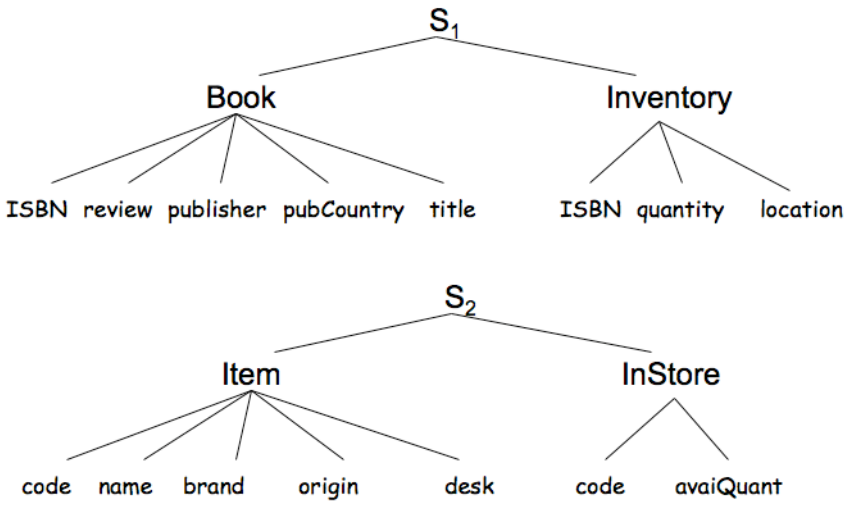

Figure 7.5: Tree representation of the schemas in Figure 7.4. The root of the tree is the name of the schema. Its children are the relation names and their children are attributes.

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 7.4). 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 T_4). The algorithm applies the constraint, modifying the similarities if necessary, and proceeds to the next node.

Termination: The algorithm terminates either after a fixed number of iterations or when the changes to *sim* are smaller than a pre-defined threshold.

Example 7.7: In our example, let α be 20%. When we choose the node pubCountry in **Book**, we will consider the pair (pubCountry, origin) because it has the highest value in the row of pubCountry. Constraint T_2 dictates that we increase their similarity by 20% to 0.84. When we consider the nodes **Item** and **Book**, constraint T_4 will cause us to increase their similarity by 20% to 0.6.

The algorithm will terminate after a few iterations because the constraints T_2 and T_4 can only cause the similarities to increase, and both of them do not apply once the similarity is over 0.9. □

Unlike the A* algorithm that outputs a schema match, the local propagation algorithm changes the similarity values in the matrix so the match selector is more likely to choose the appropriate match. We describe the match selector next.

7.6 Match Selector

The result of the previous components of the schema matcher is a similarity matrix for the schemas of S_1 and S_2. The matrix combines the predictions of multiple base matchers and the knowledge expressed as domain constraints. The match selector produces from this matrix a schema match or the top few matches.

If the matching system is interactive (a very typical case), then the system can proceed by presenting the user with the top k correspondences for each element of the schema S_1, and letting the user select the correct one. However, while such a simple approach works well in many contexts, it loses the relationship between possible choices. For example, suppose source S_1 has the attributes shipAddr, shipPhone, billAddr and billPhone, and source S_2 has the attributes addr and phone. We may be equally confident in the correspondences shipPhone → phone and billPhone → phone. However, once the designer indicated that shipAddr → addr holds, then shipPhone → phone becomes more likely than billPhone → phone. Hence, it is beneficial for the system to compute several possible schema matches, so when the user makes a particular choice, the other suggested correspondences can be adjusted accordingly.

There are a variety of algorithms that can be used to select schema matches. The key challenge is that we cannot select a correspondence for each element in S_1

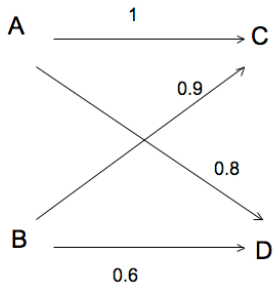

Figure 7.6: Matching A to C and B to D is a stable marriage match. However, matching A to D and B to C maximizes the total confidences of the correspondences.

in *isolation*, because the result may not be a schema match (e.g., we may match several elements of S_1 to the same element in S_2). We can formulate the match selection problem as an instance of finding a *stable marriage*. Specifically, imagine the elements of S_1 to be men and the elements of S_2 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 $A_i \rightarrow B_j$ and $A_k \rightarrow B_l$ such that $sim(i, l) > sim(i, j)$ and $sim(i, l) > sim(k, l)$. If these couples existed, then A_i and B_l would clearly want to be matched with each other.

We can produce a schema match 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 $A_i \rightarrow B_j$ to `match`.

In contrast to the stable marriage, we can consider a match that maximizes the sum of the correspondence predictions. For example, in Figure 7.6, 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.

Finally, we note that the match selector can also filter correspondences from a suggested schema match if its predicted likelihood is less some threshold. In particular, the A* algorithm described in Section 7.5 outputs a proposed schema match, and the only role of the match selector is to filter correspondences.

7.7 Reusing Previous Matches

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

More generally, working in a particular domain, the same concepts tend to re-occur. As the designer works in the domain, 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 that there is the concept of a house and an agent, the typical attributes each of them have 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 matcher learn from previous experience? In this section we describe how machine learning techniques can be applied to schema matching and enable the matcher 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.

7.7.1 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 schema 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 $s_{m+1}, \ldots, s_n$. Figure 7.7 shows the different components of this system.

We proceed by learning classifiers for elements of the mediated schema. As explained in Section 7.3, 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 basic matchers, any machine learning algorithm also typically considers only one aspect of the schema and has its strengths and weaknesses. Hence, we combine multiple

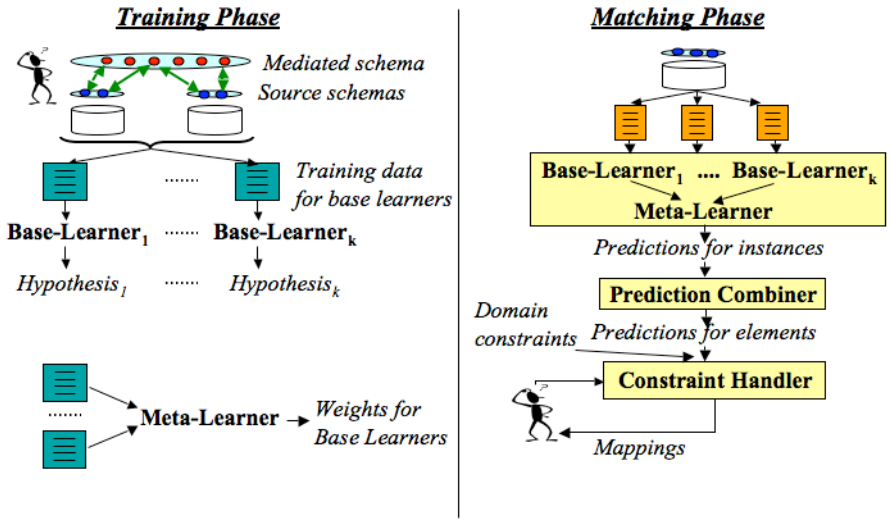

Figure 7.7: Learning from previous schema matches. Examples of schema matches are used as training examples to a set of base learners. Each of them learns a classifier for each element of the mediated schema. The meta-learner decides how to combine the predictions of the base learners. When a new schema is seen, we apply the classifiers and predict how schema elements map to the mediated schema. The learner can serve as another base matcher and its results are fed into the prediction combiner and the constraint enforcer.

learning methods with a technique called *multi-strategy learning*. The training phase of multi-strategy learning works as follows:

- Employ a set of *base learners*, $l_1, \ldots, l_k$. Each base learner creates a classifier for each element, e, of the mediated schema from its training examples.
- Use a *meta-learner* to learn weights for the different base learners. Specifically, for each element e of the mediated schema and base learner l, the meta-learner computes a weight $w_{e,l}$.

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

- Apply the base learners to $e'_1, \ldots, e'_n$. Denote by $p_{e,l}(e')$ the prediction of learner l on whether e' matches e.

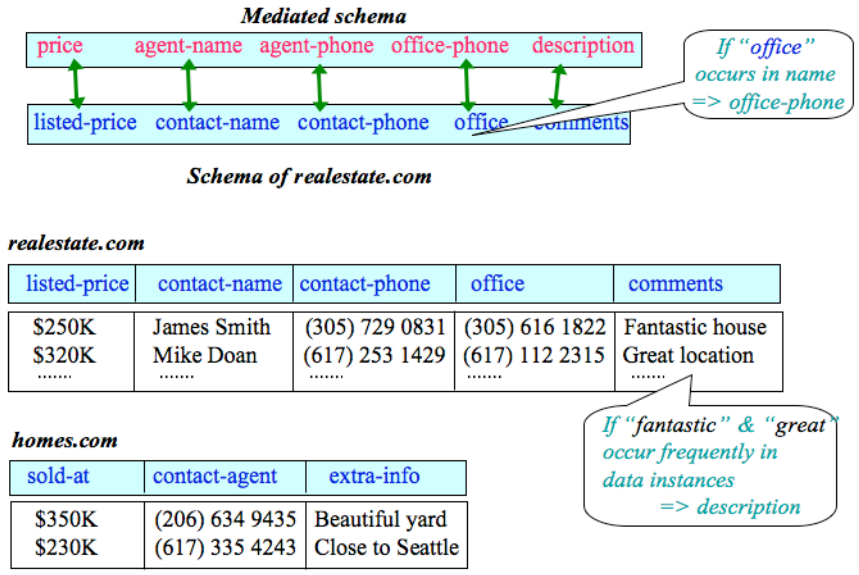

Figure 7.8: An example of learning schema matches. The mediated schema is shown on the top along with one training schema match. The callouts describe example rules that the learners can generate.

- Combine the learners:

$$p_e(e') = \sum_{j=1}^{k} w_{e,l_j} * p_{e,l_j}(e').$$

The values computed by the prediction of the meta-learner can be treated like any other base matcher in the overall schema matching system.

Example 7.8: Figure 7.8 illustrates a scenario of learning from previous matches in the context of the real-estate domain. The mediated schema is shown in the top of the figure and two sources are shown below (realestate.com and homes.com).

We train the system with two schema matches. For example, the figure shows the match from realestate.com to the mediated schema. The match from homes.com to the mediated schema is {sold-at → price, contact-agent → agent-phone, extra-info → description}.

In the next section we describe a rule-based learner that learns rules such as: *if the attribute name includes the token "office", then it likely matches the attribute* office-phone.

We also describe the Naive Bayes learner, that would learn rules such as: *if the words "fantastic", "beautiful" and "great" occur frequently in data instances, then the attribute most likely matches* description.

The meta-learner will learn rules such as: (1) *for fields that include long texts, give the Naive Bayes learner more weight than the rule-based learner* and (2) *for fields that have less than 20 characters, give the rule-based learner more weight than the Naive Bayes learner.* □

Section 7.7.2 describes two common base learners, and Section 7.7.3 describes how to train the meta-learner.

7.7.2 Base learners

We describe two base learners: a rule-based learner and the Naive Base 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 is set of rules, based on features of the training examples, that accurately characterize instances of the class, such as identifying elements that adhere to certain formats.

Consider the example of a rule learner for identifying phone numbers (for simplicity, we restrict our attention to American phone numbers). Table 7.2 shows example instances of strings that can be fed into the learner. 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

1. does the string have 10 digits?
2. does the string have 7 digits?
3. is there a '(' in position 1?
4. is there a ')' in position 5?
5. is there a '-' in position 4?

Note that Table 7.2 stores the data needed to compute the features, but for efficiency reasons its columns do not exactly correspond to the features. We also note that determining the important features can often be a non-obvious task.

Example	instance?	# of digits	position of (	position of)	position of -
(608)435-2322	yes	10	1	5	9
(60)445-284	no	9	1	4	8
849-7394	yes	7	-	-	4
(1343) 429-441	no	10	1	6	10
43 43 (12 1285)	no	10	5	12	-
5549902	no	7	-	-	-
(212) 433 8842	yes	10	1	5	-

Table 7.2: Positive and negative examples of phone numbers.

A common method to learn rules is to create a decision tree. For example, Figure 7.9 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 attribute 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 7.9 we first distinguish between the instances that have 10 digits from those that do not. The next important feature for those the 10-digit instances the position of the '('.

The decision tree in Figure 7.9 encodes rules such as

if i has 10 digits, a '(' in position 1 and ')' in position 5, **then** yes.

if i has 7 digits, but no '-' in position 4, **then** no.

The Naive Bayes learner

The Naive Bayes learner examines the tokens of a testing instance and assigns to the instance the most likely class, given the occurrences of tokens in the training data. The Naive Bayes learner is especially effective at recognizing text fields. The learner is trained by example instances that are values in a column of a data source. The learner can also be trained on names of attributes or descriptions of schema elements.

Given a test 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_j are tokens

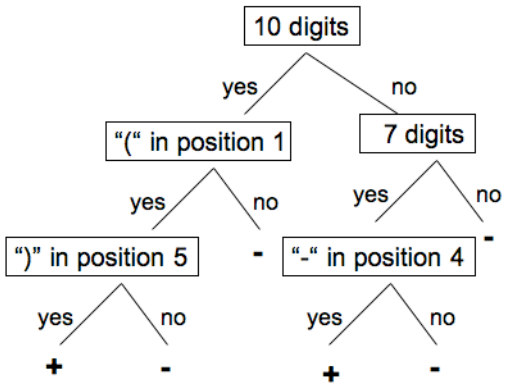

Figure 7.9: A decision tree for phone numbers. The decision tree encodes rules such as **if** i has 10 digits, a '(' in position 1 and ')' in position 5, **then** yes.

of the instance. The goal of the Naive Bayes Learner is to assign d to the element c_d that has 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 that states:

$$P(c_i|d) = P(d|c_i)P(c_i)/P(d).$$

Therefore,

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

Hence, to make its prediction, the Naive Bayes learner needs to estimate $P(d|c_i)$ and $P(c_i)$ from the training data, which it does as follows.

The probability $P(c_i)$ is approximated 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 $P(w_j|c_i)$ by

$$\frac{n(w_j, c_i)}{n(c_i)}$$

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, the Naive Bayes learner works best when there are tokens that are strongly indicative of the correct label, by virtue of their frequencies. For example, it works for house descriptions which frequently contain words such as "beautiful" and "fantastic" – words that seldom appear in other elements. It also works well when there are only weakly suggestive tokens, but many of them. In contrast, it does not work well for short or numeric fields, such as color, zip code, or number of bathrooms.

7.7.3 Training the meta-learner

The meta-learner decides the weights to attach to each of the base learners. The weights can be different for every mediated-schema element. The meta-learner learns the weights from the training examples. Specifically, the meta-learner begins by asking the base 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 base learner performs with respect to each mediated-schema element. Based on this judgment, the meta-learner assigns to each combination of mediated-schema element c_i and base learner L_j a weight $W_{L_j}^{c_i}$ that indicates how much it *trusts* learner L_j's predictions regarding c_i.

A training example describes the predictions of the base learners on an element e_s of a source schema s. The training example describes properties of e_s and the predictions of the learners. Formally, the the training data for the meta-learner are of the form $(f_1, \ldots, f_l, d_1, \ldots, d_n, D)$, where

- $f_1, \ldots, f_l$ are features of e_s. Example features include the average length of the field, the ratio of numeric characters to non-numeric characters in the field and number of different values in the field.
- d_i is the prediction that the base learner L_i made for the pair (e_s, c_i), and
- D is the correct prediction for the attribute s.

In principle, the meta-learner can use any classification algorithm to compute the appropriate weights. For example, a rule-based learner would learn a rule that gives the Naive Bayes learner more weight when the attribute has long texts, and gives the rule-based learner more weight when the attribute has mostly digits.

7.8 Many-to-Many Matches

Until now we 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 7.10. The price attribute in the **Item** table corresponds to basePrice * taxRate in the **Book** table; the author attribute in the **Item** table corresponds to concat(authorFirstName, authorLastName) in the **Book** 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.

Source
Book
- title
- basePrice
- taxRate
- quantity
- authorFirstName
- authorLastName

Target
Item
- title
- price
- inventory
- author
- genre

Figure 7.10: In this example schema the price attribute in the **Item** table corresponds to basePrice * taxRate in the **Book** table; the author attribute in the **Item** table corresponds to concat(authorFirstName, authorLastName) in the **Book** table.

In this section we discuss how to extend schema matching techniques to capture many-to-many correspondences. The main challenge lies in the fact that there may too many compound elements, and therefore too many candidate correspondences

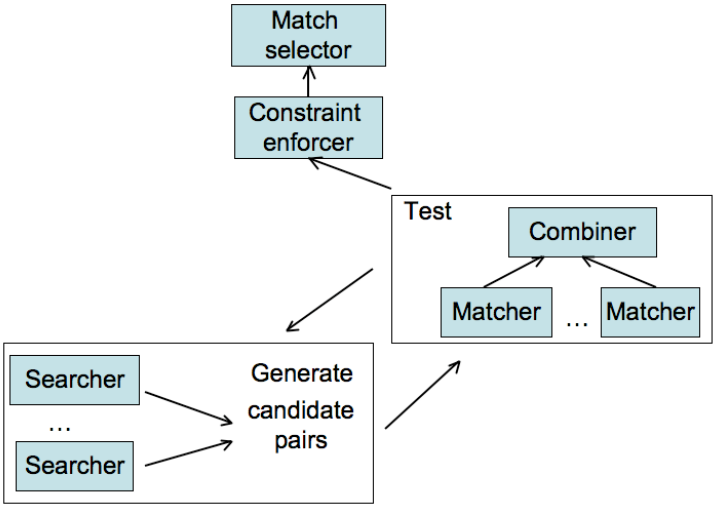

Figure 7.11: To discover correspondences among compound attributes, we employ several searchers to create candidate correspondences. We test them with the basic matchers and the combiner. We terminate the searches when the quality of the predicted correspondences does not change significantly.

to test. Whereas in the case of 1-1 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 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 7.11). We generate candidate element pairs, where one or two 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. We 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 basic matchers are more appropriate.

Termination condition: While in some cases the search will terminate because the number 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 ith 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 pre-specified threshold δ, we stop the search and return the top k correspondences.

Example 7.9: In our example, 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 addition, multiplication, percent increase (i.e., A + B%), and some constants, such as 32, 5/9 (for Fahrenheit to celsius), 2.54 (for converting inches to centimeters).

When the algorithm looks for a match for the author attribute of **Item**, 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 **Item**, it tests many combinations including basePrice + taxRate, basePrice + taxRate%, basePrice * taxRate, basePrice + quantity, and taxRate * quantity. □

7.9 From Matches to Mappings

Recall from Chapter 5 that a schema mapping between a source and target schema is an expression that can be used to map data instances from the source to the target, or to reformulate queries from the target onto the source. Schema matches only specify correspondences between the source and the target, and therefore only partially specify the mapping. The reason we single out the schema matching step in

schema mapping is that correspondences are 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 correspondences.

The remaining challenge in creating the schema mappings is to put all the correspondences together into a coherent whole. This involves specifying the operations that need to be performed on the source or target data so they can ben transformed from one to the other. In particular, creating the mapping requires aligning the 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 schema match, we describe an algorithm that searches through the possible schema mappings that are consistent with the match. 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 7.10: Figure 7.12 shows a subset of the schema match produced between a source university schema (on the left) and an accounting schema.

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
```

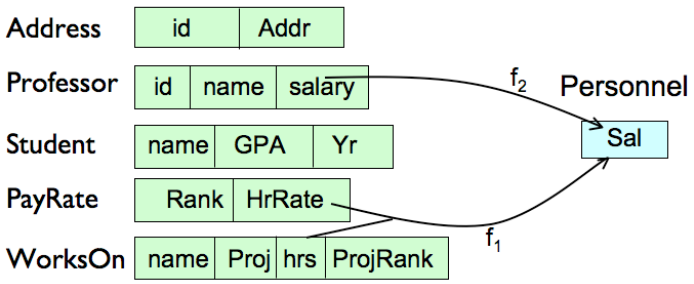

Figure 7.12: Combining correspondences from multiple tables. The correspondence f_1 states that the product of HrRate and hrs corresponds to sal of **Personnel**, but the question is which tables to join to create a table that includes the two attributes. The correspondence f_2 states that salary also corresponds to Sal, and the question is whether to union professor salaries with employee salaries or to join salaries computed from the two correspondences.

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. We may be able to resolve the quandary if the correspondence f_1 specified a filter, e.g., S.Yr > 2. In that case, it is likely that the second join is the right one.

Now consider the second correspondence, f_2, that states that the Salary attribute of **Professor** maps to Sal in the target. One interpretation of f_2 is that the values

produced from the 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 the another 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 schema match 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 we 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 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 7.13 shows our example with a few more correspondences:

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

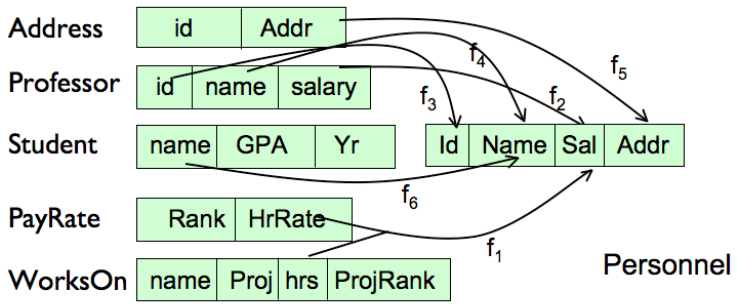

Figure 7.13: The full set of correspondences for Figure 7.12. The correspondences naturally divide into two sets: f_1 and f_6 specify how to compute target tuples for general personnel, while f_2, f_3, f_4 and f_5 describe how to compute target tuples for professors.

These correspondences naturally fall into two sets. The first set, including f_2, f_3, f_4 and f_5, map from **Professor** to **Personnel**, and intuitively specify how to create tuples for **Personnel** for professors. The second set, including the correspondences f_1 and f_6, specify 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 every 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.Sal, 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 that does not perform any joins at all, and seems rather unintuitive:

```
SELECT NULL as ID, NULL as Name, NULL as Sal, Addr
```

```
FROM   Address A
UNION ALL
SELECT P.Id, P.Name, P.Sal, 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 schema match. For simplicity, we describe the algorithm for the case where the target includes 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 match is specified as a set of correspondences: $M = \{f_i : (\bar{A}_i \rightarrow B_i)\}$, 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 schema match 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). The algorithm, shown in Figure 7.14, 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 7.11: Suppose we had the correspondences

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

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. □

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 $(\bar{A}_i \rightarrow B_i) \in v$, 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 join paths) A join path can either be:

- a path through foreign keys,
- a path proposed by inspecting previous queries on S, or
- 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 7.10), we select among them using the following heuristic.

Heuristic 2: (selecting 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 7.11, $\mathcal{G} = \{\{f_1, f_2\}, \{f_2, f_3\}, \{f_1\}, \{f_2\}, \{f_3\}\}$. 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 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.

7.10 Bibliographic Notes

The areas of schema matching and schema mapping have been the topic of research for over a decade. The survey [380] summarizes the state of the art circa 2001, and covers some of the early work such as [49, 84, 339, 361, 370].

The architecture presented in this chapter is based on the LSD System [145] and the COMA System [144]. The algorithms described often combine different components of the architecture we describe, but do not tease them apart. For example, the Similarity Flooding Algorithm [331] 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 idea of leveraging the combination of multiple matchers in a schema matching system was introduced in [144, 145]. The LSD System [145] introduced the use of multiple base learners and combining them via meta learning. The system used a technique called *stacking* [425, 444] to combine the predictions of the base learners. Stacking uses *cross-validation* to ensure that the weights learned for the base learners do not overfit the training sources, but instead generalize correctly to new ones. Further explanation of the Naive Bayes learner and an analysis for why it works well in practice can be found in [148]. More background on learning techniques, and in particular, rule-based learning, can be found in [341].

The COMA system considered a simple form of reusing past mappings (without learning from them), and also considered composing multiple past mappings. Corpus-based schema matching [313] showed how to leverage a corpus of schemas and mappings in a particular domain to improve schema matching. In [231], He and Chang looked 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 iMap System [287] addressed the problem of searching for many-to-many mappings, as we described in Section 7.8. Gal [190] discusses the complexity of finding the top-k schema matches.

Several schema matching systems considered how to apply domain constraints to prune and improve schema matches [145, 146, 312, 331]. The neighborhood-

adaptation algorithm we described in Section 7.5 is adapted from [312], and the A* algorithm is from [145].

The Clio System was the first to look at the problem of generating schema mappings given a schema match [337, 449]. The algorithm we described in Section 7.9 and the running example there are taken from [337]. 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.

Algorithm FindMapping
Input:
A set of correspondences between schemas S_1 and S_2: $M = \{f_i : (\bar{A}_i \rightarrow B_i)\}$.
When $\bar{A}_i$ has more than one attribute, we assume it is of the form $g(\bar{A}_i)$,
where g is the function that combines the attributes.
$filter_i$: is the set of filters associated with f_i.
Phase 1: /* Create candidate sets */
$\mathcal{V} = \{v \subseteq M | \, v$ does not mention an attribute of S_2 more than once$\}$.
Phase 2:
$\mathcal{G} = \mathcal{V}$
for every $v \in \mathcal{V}$
if $(\bar{A}_i \rightarrow B_i) \in v$ 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 $\mathcal{G}$
Phase 3:
$Covers = \{\Gamma \mid \Gamma \subseteq \mathcal{G}, \Gamma$ mentions all $f_i \in M$ and no subset of Γ mentions all $f_i \in M\}$
$selectedCover =$ the cover $c \in Covers$ with the fewest candidate paths
if $Covers$ has more than one cover **then** select one with Heuristic 3.
Phase 4:
for each $v \in selectedCover$ 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, and
$filter_1, \ldots, filter_m$ are the filters associated with the correspondences in v.
return the query
Q_1 `UNION ALL ... UNION ALL` Q_b,
where $Q_1, \ldots, Q_b$ are the queries created above.
end

Figure 7.14: Algorithm for searching through a space of possible schema mappings.