Integration of the extracted data to produce a consistent and coherent database

Abstract

we studied data extraction from Web pages. The extracted data is put in tables. For an application, it is, however, often not sufficient to extract data from only a single site. Instead, data from a large number of sites are gathered in order to provide value-added services. In such cases, extraction is only part of the story. The other part is the integration of the extracted data to produce a consistent and coherent database because different sites typically use different data formats. Intuitively, integration means to match columns in different data tables that contain the same type of information (e.g., product names) and to match values that are semantically identical but represented differently in different Web sites (e.g., "Coke" and "Coca Cola"). Unfortunately, limited integration research has been done so far in this specific context. Much of the Web information integration research has been focused on the integration of **Web query interfaces**. This chapter will have several sections on their integration. However, many ideas developed are also applicable to the integration of the extracted data because the problems are similar.

Web query interfaces are used to formulate queries to retrieve needed data from **Web databases** (called the **deep Web**). Figure 10.1 shows two query interfaces from two travel sites, expedia.com and vacation.com. The user who wants to buy an air ticket typically tries many sites to find the cheapest ticket. Given a large number of alternative sites, he/she has to access each individually in order to find the best price, which is tedious. To reduce the manual effort, we can construct a **global query interface** that allows uniform access to disparate relevant sources. The user can then fill in his/her requirements in this single global interface and all the underlying sources (or databases) will be automatically filled and searched. The retrieved results from multiple sources also need to be integrated. Both integration problems, i.e., integration of query interfaces and integration of returned results, are very challenging due to the heterogeneity of Web sites.

Clearly, integration is not peculiar only to the Web. It was, in fact, first studied in the context of relational databases and data warehouse. Hence, this chapter first introduces most integration related concepts using traditional data models (e.g., relational) and then shows how the concepts are tailored to Web applications and how Web specific problems are handled.

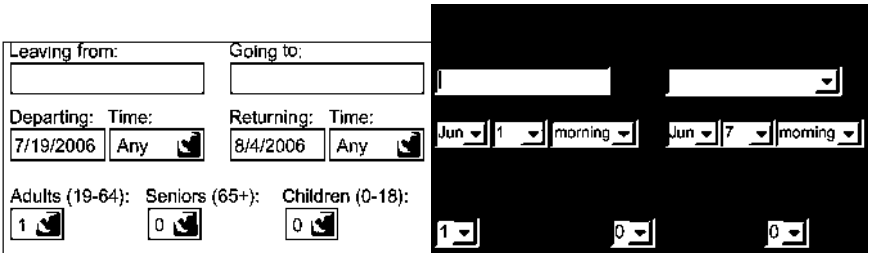

Fig. 10.1. Two examples of Web query interfaces

10.1 Introduction to Schema Matching

Information/data integration has been studied in the database community since the early 1980s [40, 146, 455]. The fundamental problem is **schema matching**, which takes two (or more) database schemas to produce a mapping between **elements** (or **attributes**) of the two (or more) schemas that correspond semantically to each other. The objective is to merge the schemas into a single global schema. This problem arises in building a global database that comprises several distinct but related databases. One application scenario in a company is that each department has its database about customers and products that are related to the operations of the department. Each database is typically designed independently and possibly by different people to optimize database operations required by the functions of the department. This results in different database schemas in different departments. However, to consolidate the data about customers or company operations across the organization in order to have a more complete understanding of its customers and to better serve them, integration of databases is needed. The integration problem is clearly also important on the Web as we discussed above, where the task is to integrate data from multiple sites.

There is a large body of literature on the topic. Most techniques have been proposed to achieve semi-automatic matching in specific domains (see the surveys in [146, 265, 455, 491]). Unfortunately, the criteria and methods used in match operations are almost all based on domain heuristics which are not easily formulated mathematically. Thus, to build a schema matching system, we need to produce mapping heuristics which reflect our understanding of what the user considers to be a good match.

Schema matching is challenging for many reasons. First of all, schemas of identical concepts may have structural and naming differences. Schemas may model similar but not identical contents, and may use different data models. They may also use similar words for different meanings.

Although it may be possible for some specific applications, in general, it is not possible to fully automate all matches between two schemas because some semantic information that determines the matches between two schemas may not be formally specified or even documented. Thus, any automatic algorithm can only generate candidate matches that the user needs to verify, i.e., accept, reject or change. Furthermore, the user should also be allowed to specify matches for elements that the system is not able to find satisfactory match candidates. Let us see a simple example.

Example 1: Consider two schemas, S_1 and S_2, representing two customer relations, Cust and Customer.

S_1	S_2
Cust	Customer
CNo	CustID
CompName	Company
FirstName	Contact
LastName	Phone

We can represent the mapping with a similarity relation, $\cong$, over the power sets of S_1 and S_2, where each pair in $\cong$ represents one element of the mapping. For our example schemas, we may obtain

Cust.CNo $\cong$ Customer.CustID
Cust.CompName $\cong$ Customer.Company
{Cust.FirstName, Cust.LastName} $\cong$ Customer.Contact ■

There are various types of matching based on the input information [455].

1. **Schema-level only matching:** In this type of matching, only the schema information (e.g. names and data types) is considered. No data instance is available.
2. **Domain and instance-level only matching:** In this type of match, only instance data and possibly the domain of each attribute are provided. No schema is available. Such cases occur quite frequently on the Web, where we need to match corresponding columns of the hidden schemas.
3. **Integrated matching of schema, domain and instance data:** In this type of match, both schemas and instance data (possibly domain information) are available. The match algorithm can exploit clues from all of them to perform matching.

There are existing approaches to all above types of matching. We will focus on the first two types. The third type usually combines the results of techniques from the first two, which we discuss in Sect. 10.5. Before going to the details, we first discuss some pre-processing tasks that usually need to be done before matching.

10.2 Pre-Processing for Schema Matching

For pre-processing, issues such as concatenated words, abbreviations, and acronyms are dealt with. That is, they need to be normalized before being used in matching [227, 358, 559].

Prep 1 (**Tokenization**): This process breaks an item, which can be a schema element (attribute) or attribute value, into atomic words. Such items are usually concatenated words. Delimiters (such as "-", "_", etc.) and case changes of letters are used to suggest the breakdown. For example, we can break "fromCity" into "from City", and "first-name" into "first name". A domain dictionary of words is typically maintained to help the breakdown. Note that if "from", "city", "first" and "name" are not in the dictionary, they will be added to the dictionary. Existing dictionary words are also utilized to suggest the breakdown. For example, "deptcity" will be split into "dept" and "city" if "city" is a word. The dictionary may be constructed automatically, which consists of all the individual words appeared in the given input used in matching, e.g., schemas, instance data and domains. The dictionary is updated as the processing progresses. However, the tokenization step has to be done with care. For example, we have "Baths" and "Bathrooms" if we split "Bath" with "Room" it could be a mistake because "Rooms" could have a very different meaning (the number of rooms in the house). To be sure, we need to ensure that "Bathroom" is not an English word, for which an online English dictionary may be employed.

Prep 2 (**Expansion**): It expands abbreviations and acronyms to their full words, e.g., from "dept" to "departure". The expansion is usually done based on the auxiliary information provided by the user or collected from other sources. Constraints may be imposed to ensure that the expansion is likely to be correct. For example, we may require that the word to be expanded is not in the English dictionary, with at least three letters, and having the same first letter as the expanding word. For example, "CompName" is first converted to (Comp, Name) in tokenization, and then "Comp" is expanded to "Company".

Prep 3 (**Stopword removal and stemming**): These are information retrieval pre-processing methods (see Chap. 6). They can be performed to attribute names and domain values. A domain specific stopword list may also be constructed manually. This step is useful especially in linguistic based matching methods discussed below.

Prep 4 (**Standardization of words**): Irregular words are standardized to a single form (e.g., using WordNet [175]), "colour"→ "color", "Children" → "Child".

10.3 Schema-Level Matching

A schema level matching algorithm relies on information about schema elements, such as name, description, data type and relationship types (such as **part-of**, **is-a**, etc.), constraints and schema structures. Before introducing some matching methods using such information, let us introduce the notion of **match cardinality**, which describes the number of elements in one schema that match the number of elements in the other schema.

In general, given two schemas, S_1 and S_2, within a single match in the match relation one or more elements of S_1 can match one or more elements of S_2. We thus have 1:1, 1:*m*, *m*:1 and *m*:*n* matches. 1:1 match means that one element of S_1 corresponds to one element of S_2, and 1:*m* means that one element of S_1 corresponds to a set of *m* ($m > 1$) elements of S_2.

Example 2: Consider the following schemas:

S_1	S_2
Cust	Customer
CustomID	CustID
Name	FirstName
Phone	LastName

We can find the following 1:1 and 1:*m* matches:

1:1	CustomID	CustID
1:*m*	Name	FirstName, LastName

■

m:1 match is similar to 1:*m* match; *m*:*n* match is considerably more complex. An example of an *m*:*n* match is to match Cartesian coordinates with polar coordinates. There is little work on such complex matches. Most existing approaches are for 1:1 and 1:*m* matches.

We now describe some general matching approaches that employ various types of information available in schemas. There are two main types of information in schemas, natural language words and constraints. Thus, there are two main types of approaches to matching.

10.3.1 Linguistic Approaches

They are used to derive match candidates based on the names, comments or descriptions of schema elements [107, 133, 144, 145, 227, 358, 559].

Name Match

N1 – Equality of names: The same name in different schemas often has the same semantics.

N2 – Synonyms: The names of two elements from different schemas are **synonyms**, e.g., Customer ≅ Client. This requires the use of thesaurus and/or dictionaries such as WordNet. In many cases, domain dependent or enterprise specific thesaurus and dictionaries are required.

N3 – Equality of hypernyms: *A* is a **hypernym** of *B* if *B* is **a kind of** *A*. If *X* and *Y* have the same hypernym, they are likely to match. For example, "Car" *is-a* "vehicle" and "automobile" *is-a* "vehicle". Thus, we have Car ≅ vehicle, automobile ≅ vehicle, and Car ≅ automobile.

N4 – Common substrings: Edit distance and similar pronunciation may be used. For example, CustomerID ≅ CustID, and ShipTo ≅ Ship2.

N5 – Cosine similarity: Some names are natural language words or phrases (after pre-processing). Then, text similarity measures are useful. **Cosine similarity** is a popular similarity measure used in information retrieval (see Chap. 6). This method is also very useful for Web query interface integration since the labels of the schema elements are natural language words or phrases (see the query interfaces in Fig. 10.1)

N6 – User provided name matches: The user may provide a domain dependent match dictionary (or table), a thesaurus, and/or an ontology.

Description Match

In many databases, there are comments to schema elements, e.g.,

S_1: CNo // customer unique number
S_2: CustID // id number of a customer

These comments can be compared based on the cosine similarity as well.

D1 – Use the cosine similarity to compare comments after stemming and stopword removal.

10.3.2 Constraint Based Approaches

Constraints such as data types, value ranges, uniqueness, relationship types and cardinalities, etc., can be exploited in determining candidate matches [327, 358, 382, 424].

C1: An equivalence or compatibility table for data types and keys that specifies compatibility constraints for two schema elements to match can be provided, e.g., string ≅ varchar, and (primary key) ≅ unique.

Example 3: Consider the following two schemas:

S_1	S_2
Cust	Customer
CNo: int, primary key	CustID: int, unique
CompName: varchar (60)	Company: string
CTname: varchar (15)	Contact: string
StartDate: date	Date: date

Constraints can suggest that "CNo" matches "CustID", and "StartDate" may match "Date". "CompName" in S_1 may match "Company" in S_2 or "Contact" in S_2. Likewise, "CTname" in S_1 may match "Company" or "Contact" in S_2. In both cases, the types match. Although in these two cases, we are unable to find a unique match, the approach helps limit the number of match candidates and may be combined with other matchers (e.g., name and instance matchers). For structured schemas, hierarchical relationships such as **is-a** and **part-of** relationships may be utilized to help match. ■

In the context of the Web, the constraint information above is often not explicitly available because Web databases are for general public who are unlikely to know what an int, string or varchar is. Thus, these types are never shown in Web pages. However, some information may be inferred from the domain or instance information, which we discuss next.

10.4 Domain and Instance-Level Matching

In this type of matching, value characteristics are exploited to match schema elements [53, 145, 327, 531, 558]. For example, the two attribute names may match according to the linguistic similarity, but they may have different domain value characteristics. Then, they may not be the same but **homonyms**. For example, Location in a real estate sell may mean the address, but could also mean some specific locations, e.g., lakefront property, hillside property, etc.

In many applications, data instances are available, which is often the case in the Web database context. In some applications, although the instance information is not available, the domain information of each attribute may be obtained. This is the case for Web query interfaces. Some attributes in the query interface contain a list of possible values (the domain) for the user to choose from. No type information is explicitly given, but it can often be inferred. We note that the set of value instances of an attribute can be treated in the similar way as a domain. Thus, we will only deal with domains below.

Let us look at two types of domains or types of values: simple domains and composite domains. The domain similarity of two attributes, A and B, is the similarity of their domains: $dom(A)$ and $dom(B)$.

Definition (Simple Domain): A **simple domain** is a domain in which each value has only a single component, i.e., the value cannot be decomposed.

A simple domain can be of any type, e.g., year, time, money, area, month, integer, real, string, etc.

Data Type: If there is no type specification at the schema level, we identify the data type from the domain values. Even if there is a type specification at the schema level for each attribute, we can still refine the type to find more characteristic patterns. For example, the ISBN number of a book may be specified as a string type in a given schema. However, due to its fixed format, it is easy to generate a characteristic pattern from a set of ISBN numbers, e.g., a regular expression. Other examples include phone numbers, post codes, money, etc. Such specialized patterns are more useful in matching compatible attribute types.

We describe two approaches for type identification: semi-automatic [559, 563] and automatic [145, 327] approaches.

Semi-automatic approach: This is done via pattern matching. The pattern for each type may be expressed as a regular expression, which is defined by a human expert. For example, the regular expression for the time type can be defined as "[0–9]{2}:[0–9]{2}" or "dd:dd" (d for digit from 0-9) which recognizes time of the form "03:15". One can use such regular expressions to recognize integer, real, string, month, weekday, date, time, datetime (combination of date and time), etc. To identify the data type, we can simply apply all the regular expression patterns to determine the type.

In some cases, the values themselves may contain some information on the type. For example, values that contain "$" or "US$" indicate the monetary type. For all values that we cannot infer their types, we can assume their domains are of string type with an infinite cardinality.

Automated approach: Machine learning techniques, e.g., grammar induction, may be used to learn the underlying grammar/pattern of the values of an attribute, and then use the grammar to match attribute values of the other schemas. This method is particularly useful for value of fixed format, e.g., zip codes, phone numbers, zip codes, ISBNs, date entries, or money-related entries, if their regular expressions are not specified by the user.

The following methods may be used in matching:

DI 1 – Data types are used as constraints. The method C1 above is applicable here. If the data/domain types of two attributes are not compatible, they should not be matched. We can use a table specifying the degree of compatibility between a set of predefined generic data types, to which data types of schema elements are mapped in order to determine their similarity.

DI 2 – For numerical data, value ranges, averages and variances can be computed to access the level of similarity.

DI 3 – For categorical data, we can extract and compare the set of values in the two domains to check whether the two attributes from different schemas share some common values. For example, if an attribute from S_1 contains many "Microsoft" entries and an attribute in S_2 also contains some "Microsoft"'s, then we can propose them as a match candidate.

DI 4 – For alphanumeric data, string-lengths and alphabetic/non-alphabetic ratios are also helpful.

DI 5 – For textual data, information retrieval methods such as the cosine measure may be used to compare the similarity of all data values in the two attributes.

DI 6 – **Schema element name as value** is another match indicator, which characterizes the cases where matches relate some data instances of a schema with a set of elements (attributes) in another schema. For example, in the airfare domain one schema uses "Economy" and "Business" as instances (values) of the attribute "Ticket Class", while in another interface, "Economy" and "Business" are attributes with the Boolean domain (i.e., "Yes" and "No"). This kind of match can be detected if the words used in one schema as attribute names are among the values of attributes in another schema [133, 563].

Definition (Composite Domain and Attribute): A **composite domain** d of arity k is a set of ordered k-tuples, where the ith component of each tuple is a value from the ith sub-domain of d, denoted as d_i. Each d_i is a simple domain. The arity of domain d is denoted as $arity(d)$ (= k). An **attribute** is **composite** if its domain is composite.

A composite domain is usually indicated by its values that contained delimiters of various forms. The delimiters can be punctuation marks (such as ",", "-", "/", "_", etc) and white spaces and some special words such as "to". To detect a composite domain, we can use these delimiters to split a composite domain into simple sub-domains. In order to ensure correctness, we may also want to require that a majority of (composite) values can be consistently split into the same number of components. For example, the date can be expressed as a composite domain with MM/DD/YY.

DI 7 – The similarity of a simple domain and a composite domain is determined by comparing the simple domain with each sub-domain of the composite domain. The similarity of composite domains is established by comparing their component sub-domains.

We note that splitting a composite domain can be quite difficult in the Web context. For example, without sufficient auxiliary information (e.g., information from other sites) it is not easy to split the following: "Dell desktop PC 1.5GHz 1GB RAM 30GB disk space"

10.5 Combining Similarities

Let us call a program that assesses the similarity of a pair of elements from two different schemas based on a particular match criterion a **matcher**. It is typically the case that the more indicators we have the better results we can achieve, because different matchers have their own advantages and also shortcomings. Combining schema-level and instance-level approach will produce better results than each type of approaches alone. This combination can be done in various ways.

Given the set of similarity values, $sim_1(u, v)$, $sim_2(u, v)$, …, $sim_n(u, v)$, of a set of n matchers that compared two schema elements u (from S_1) and v (from S_2), one of the following strategies can be used to combine their similarity values.

1. **Max:** This strategy returns the maximal similarity value of any matcher. It is thus optimistic. Let the combined similarity be *CSim*. Then

$$CSim(u, v) = \max\{sim_1(u, v), sim_2(u, v), \ldots, sim_n(u, v)\} \quad (1)$$

2. **Weighted Sum:** This strategy computes a weighted sum of similarity values of the individual matchers. It needs relative weights which correspond to the expected importance of the matchers:

$$CSim(u, v) = \lambda_1 * sim_1(u, v) + \lambda_2 sim_2(u, v) + \ldots + \lambda_n * sim_n(u, v), \quad (2)$$

where λ_i is a weight coefficient, and usually determined empirically.

3. **Weighted Average:** This strategy computes a weighted average of similarity values of the individual matchers. It also needs relative weights that correspond to the expected importance of the matchers.

$$CSim(u,v) = \frac{\lambda_1 Sim_1(u,v) + \lambda_2 Sim_2(u,v) + \ldots + \lambda_n Sim_n(u,v)}{n}, \quad (3)$$

where λ_i is a weight coefficient and is determined experimentally.

4. **Machine Learning:** This approach uses a classification algorithm, e.g., a decision tree, a naïve Bayesian classifier, or SVM, to determine whether two schema elements match each other. In this case, the user needs to label a set of training examples, which is described by a set of attributes and a class. The attributes can be the similarities. Each training example thus represents the similarity values of a pair of schema elements. The class of the example is either Yes or No, which indicates whether the two elements match or not as decided by the user.

There are many other possible approaches. In practice, which method to use involves a significant amount of experimentation and parameter tuning. Note that the combination can also be done in stages for different types of matches. For example, we can combine the instance based similarities first using one method, e.g., **Max**, and then combine schema based similarities using another method, e.g., **Weighted Average**. After that, the final combined similarity computation may use **Weighted Sum**.

10.6 1:*m* Match

The approaches presented above are for 1:1 matches. For 1:*m* match, other techniques are needed [133, 563, 559]. There are mainly two types of 1:*m* matches.

Part-of Type: Each relevant schema element on the many side is a part of the element on the one side. For example, in one schema, we may have an attribute called "Address", while in another schema, we may have three attributes, "Street", "City" and "State". In this case, "Street", "City" or "State" is a part of "Address". That is, the combination of "Street", "City" or "State" forms "Address". Thus, it is a 1:*m* match.

Is-a Type: Each relevant schema element on the many side is a specialization of the schema element on the one side. The content of the attribute on the one side is the union or sum of the contents of the attributes on the many side. For example, "HomePhone" and "CellPhone" in S_2 are specializations of "Phone" in S_1. Another example is the (number of) "Passengers" in Fig. 10.3 (page 397), and the (number of) "Adults", the (number of) "Seniors", and the (number of) "Children" in Fig. 10.1 in the airline ticket domain.

Identifying Part-of 1:*m* Matches: For each attribute A in interface S_1, we first check if it is a composite attribute as described above. If A is a composite attribute, we find a subset of schema elements in S_2 that has a 1:1 correspondence with the sub-attributes of A. For a real application, we may need additional conditions to make the decision (see Sect. 10.8.1).

Identify Is-a 1:*m* Matches: In the case of part-of 1:*m* mappings, the domains of the sub-attributes are typically different. In contrast, the identification of is-a 1:*m* mappings of attributes requires that the domain of each corresponding sub-attribute be similar to that of the general attribute. Name matching of schema elements is useful here. For example, in the case of "Phone" in S_1 and "HomePhone" and "CellPhone" in S_2, the name similarity can help decide 1:*m* mapping. However, this strategy alone is usually not sufficient, e.g., "Passengers" in S_1 and "Adults", "Seniors" and "Children" in S_2 have no name similarity. Additional information is needed. We will show an example in Sect. 10.8.1.

Using the auxiliary information provided by the user is also a possibility. It is not unreasonable to ask the user to provide some information about the domain. For example, a domain ontology that includes a set of concepts and their relationships such as the following (Fig. 10.2) will be of great help:

Part-of("street", "address")	Is-a("home phone", "phone")
Part-of("city", "address")	Is-a("cell phone", "phone")
Part-of("state", "address")	Is-a("office phone", "phone")
Part-of("country", "address")	Is-a("day phone", "phone")

Fig. 10.2. Part-of(X, Y) – X is a part of Y, and Is-a(X, Y) – X is a Y.

10.7 Some Other Issues

10.7.1 Reuse of Previous Match Results

We have mentioned in several places that auxiliary information in addition to the input schemas and data instances, such as dictionaries, thesauri, and user-provided ontology information are very useful in schema matching. The past matching results can also be stored and reused for future matches [356, 455]. Reuse is important because many schemas are very similar to each other and to previously matched schemas. Given a new schema S to be matched with a set of existing schemas E, we may not need to match S with every existing schema in E. There are two slightly different scenarios:

1. Matching of a large number of schemas: If we have a large number of schemas to match, we may not need to perform all pair-wise matches, which have $n(n+1)/2$ of them with n being the number of input schemas. Since most schemas are very similar, the $n(n+1)/2$ number of matches are not necessary.

2. Incremental schema matching: In this scenario, given a set of schemas that has already been matched, when a new schema S needs to be matched with existing matched schemas E, we may not want to use S to match every schema in E using pair-wise matching. This is the same situation as the first case above. If the original match algorithm is not based on pair-wise match, we may not want to run the original algorithm on all the schemas to just match this single schema with them.

For both cases, we want to use old matches to facilitate the discovery of new matches. The key idea is to exploit the **transitive property** of similarity relationship. For example, "Cname" in S_1 matches "CustName" in S_2 as they are both customer names. If "CTname" in the new schema S matches "Cname" in S_1, we may conclude that "CTname" matches "CustName" in S_2. The transitive property has also been used to deal with some difficult matching cases. For example, it may be difficult to map a schema element A directly to a schema element B, but easy to map both A and B to the schema element C in another schema. This helps us decide that A corresponds to B [144, 559, 563].

In the incremental case, we can also use a clustering-based method. For example, if we already have a large number of matches, we can group them into clusters and find a centroid to represent each cluster, in term of schema names and domains. When a new schema needs to be matched, the schema is compared with the centroid rather than with each individual schema in the cluster.

10.7.2 Matching a Large Number of Schemas

The techniques discussed so far are mainly for pair-wise matching of schemas. However, in many cases, we may have a large number of schemas. This is the case for many Web applications because there are many Web databases in any domain or application. With a large number of schemas, new techniques can be applied. We do not need to depend solely on pair-wise matches. Instead, we can use statistical approaches such as data mining to find patterns, correlations and clusters to match the schemas. In the next section, we will see two examples in which clustering and correlation methods are applied.

10.7.3 Schema Match Results

In pair-wise matching, for each element v in S_2, the set of matching elements in S_1 can be decided by one of the following methods [144].

1. **Top *N* candidates:** The top N elements of S_1 that have the highest similarities are chosen as match candidates. In most cases, $N = 1$ is the natural choice for 1:1 correspondences. Generally, $N > 1$ is useful in interactive mode, i.e., the user can select among several match candidates.
2. **MaxDelta:** The S_1 element with the maximal similarity is determined as match candidate plus all S_1 elements with a similarity differing at most by a tolerance value t, which can be specified either as an absolute or relative value. The idea is to return multiple match candidates when there are several S_1 elements with almost the same similarity values.
3. **Threshold:** All S_1 elements with the final combined similarity values exceeding a given threshold t are selected.

10.7.4 User Interactions

Due to the difficulty of schema matching, extensive user interaction is often needed in building an accurate matching system for both parameter tuning and resolving uncertainties

Building the Match System: There are typically many parameters and thresholds in an integration system, e.g., similarity values, weight coefficients, and decision thresholds, which are usually domain-specific or even attribute specific. Before the system is used to match other schemas, interactive experiments are needed to tune the parameters by trial-and-errors.

After Matching: Although the parameters are fixed in the system building, their values may not be perfect. Matching mistakes and failures will still occur: (1) some matched attributes may be wrong (false positive); (2) some true matches may not be found (false negative). User interactions are needed to correct the situations and to confirm the correct matches.

10.8 Integration of Web Query Interfaces

The preceding discussions are generic to database integration and Web data integration. In this and the next sections, we focus on integration in the Web context. The Web consists of the **surface Web** and the **deep Web**. The surface Web can be browsed using any Web browser, while the deep Web consists of databases that can only be accessed through parameterized query interfaces. With the rapid expansion of the Web, there are now a huge number of deep web data sources. In almost any domain, one can find a large number of them, which are hosted by e-commerce sites. Each of such sources usually has a keyword based search engine or a query

interface that allows the user to fill in some information in order to retrieve the needed data. We have seen two query interfaces in Fig. 10.1 for finding airline tickets. We want to integrate multiple interfaces in order to provide the user a **global query interface** [153, 227] so that he/she does not need to manually query each individual source to obtain more complete information. Only the **global interface** needs to be filled with the required information. The individual interfaces are filled and searched automatically.

We focus on query interface integration mainly because there is extensive research in this area, although the returned instance data integration is also of great importance and perhaps even more important due to the fact that the number of sites that provide such structured data is huge and most of them do not have query interfaces but only keyword search or can only be browsed by users (see Chap. 9).

Since query interfaces are different from traditional database schemas, we first define a schema model.

Schema Model of Query Interfaces: In each domain, there is a set of concepts $C = \{c_1, c_2, \ldots, c_n\}$ that represents the essential information of the domain. These concepts are used in query interfaces to enable the user to restrict the search for some specific instances or objects of the domain. A particular query interface uses a subset of the concepts $S \subseteq C$. A concept i in S may be represented in the interface with a set of attributes (or fields) $f_{i1}, f_{i2}, ..., f_{ik}$. In most cases, each concept is only represented with a single attribute. Each attribute is labeled with a word or phrase, called the **label** of the attribute, which is visible to the user. Each attribute may also have a set of possible values that the user can use in search, which is its **domain**.

All the attributes with their labels in a query interface are called the **schema** of the query interface [227, 608]. Each attribute also has a **name** in the HTML code. The name is attached to a TEXTBOX (which takes the user input). However, this name is not visible to the user. It is attached to the input value of the attribute and returned to the server as the attribute of the input value. The name is often an acronym that is less useful than the label for schema matching. For practical schema integration, we are not concerned with the set of concepts but only the label and name of each attribute and its domain.

Most ideas for schema matching in traditional databases are applicable to Web query interfaces as the schema of a query interface is similar to a schema in databases. However, there are also some important differences [67, 92].

1. **Limited use of acronyms and abbreviations:** Data displayed in Web pages are for the general public to view and must be easy to understand. Hence, the use of acronyms and abbreviations is limited to those very

obvious ones. Enterprise-specific acronyms and abbreviations seldom appear. In the case of a company database, abbreviations are frequently used, which are often hard to understand by human users and difficult to analyze by automated systems. To a certain extent, this feature makes information integration on the Web easier.

2. **Limited vocabulary:** In the same domain, there are usually a limited number of essential attributes that describe each object in the domain. For example, in the book domain, we have the title, author, publisher, ISBN number, etc. For each attribute, there is usually limited ways to express the attribute. The chosen label (describing a data attribute, e.g., "departure city") needs to be short, and easily understood by the general public. Therefore, there are not many ways to express the same attributes. Limited vocabulary also makes statistical approaches possible.
3. **A large number of similar databases:** There are often a large number of sites that offer the same services or sell the same products, which result in a large number of query interfaces and make it possible to use statistical methods. This is not the case in a company because the number of related databases is small. Integration of databases from multiple companies seldom happens.
4. **Additional structure:** The attributes of a Web interface are usually organized in some meaningful ways. For example, related attributes are grouped and put together physically (e.g., "first name" and "last name" are usually next to each other), and there may also be a hierarchical organization of attributes. Such structures also help integration as we will see later. In the case of databases, attributes usually have no structure.

Due to these differences, schema matching of query interfaces can exploit new methods. For example, data mining techniques can be employed as we will see in the next few sub-sections. Traditional schema matching approaches in the database context are usually based on pair-wise matching.

Similar to schema integration, query interface integration also requires mapping of corresponding attributes of all the query interfaces.

Example 4: For the two query interfaces in Fig. 10.3, the attribute correspondences are:

Interface 1 (S_1)	Interface 2 (S_2)
Leaving from	From
Going to	To
Departure date	Departure date
Return date	Return date
Passengers:	Number of tickets
Time	
Preferred cabin	

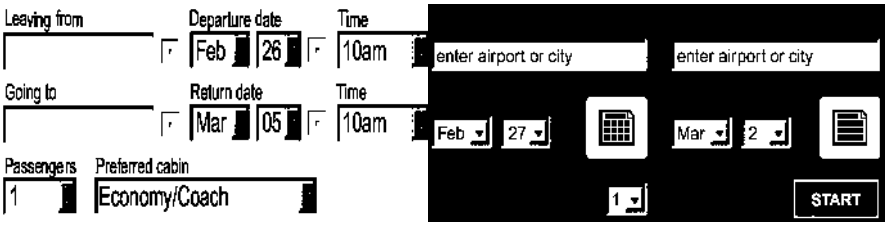

Fig. 10.3. Two query interfaces from the domain of airline ticket reservation

The last two attributes from Interface 1 do not have matching attributes in Interface 2. ■

The problem of generating the mapping is basically the problem of identifying synonyms in the application domain. However, it is important to note that the synonyms here are domain dependent. A general-purpose semantic lexicon such as WordNet or any thesaurus is not sufficient for the identification of most domain-specific synonyms. For example, it is difficult to infer from WordNet or any thesaurus that "Passengers" is synonymous to "Number of tickets" in the context of airline ticket reservation. Domain-specific lexicons are not generally available as they are expensive to build. In this section, we discuss three query interface matching techniques. We also describe a method for building a global interface.

10.8.1 A Clustering Based Approach

This technique is a simplified version of the work in [559]. Given a large set of schemas from query interfaces in the same application domain, this technique utilizes a data mining method, **clustering**, to find attribute matches of all interfaces. Three types of information are employed, namely, attribute labels, attribute names and value domains. Let the set of interface schemas be $\{S_1, S_2, \ldots, S_n\}$. The technique works in five steps:

1. Pre-processing the data. It uses the methods given in Sect. 10.2.
2. Computing all pair-wise attribute similarities of u ($\in S_i$) and v ($\in S_j$), $i \neq j$. This produces a similarity matrix.
3. Identify initial 1:m matches.
4. Cluster schema elements based on the similarity matrix. This step discovers 1:1 matches.
5. Generate the final 1:m matches of attributes.

We now discuss each step in turn except the first step.

Computing all Pair-Wise Attribute Similarities: Let u be an attribute of S_i and v be an attribute of S_j ($i \neq j$). This step computes all **linguistic similarities** (denoted by $LingSim(u, v)$) and **domain similarities** (denoted $DomSim(u, v)$). The aggregated similarity (denoted by $AS(u, v)$) is:

$$AS(u,v) = \lambda_{ls} * LingSim(u,v) + \lambda_{ds} * DomSim(u,v), \tag{4}$$

where λ_{ls} and λ_{ds} are weight coefficients reflecting the relative importance of each component similarity.

The linguistic similarity is based on both attribute labels and attribute names, which give two similarity values, $lSim(u, v)$ and $nSim(u, v)$, representing label and name similarities respectively. Both similarities are computed using the cosine measure as discussed in N5 of Sect. 10.3.1. The two similarities are then combined through a linear combination method similar to Equation (4) above.

Domain similarity of two simple domains d_v and d_u is computed based on the data type similarity (denoted by $typeSim(d_v, d_u)$ and values similarity (denoted by $valueSim(d_v, d_u)$). The final $DomSim$ is again a linear combination of the two values. For the type similarity computation, if the types of domains d_v and d_u are the same, $typeSim(d_v, d_u) = 1$ and 0 otherwise. If $typeSim(d_v, d_u) = 0$, then $valueSim(d_v, d_u) = 0$.

For two domains d_v and d_u of the same type, the algorithm further evaluates their value similarity. Let us consider two character string domains. Let the set of values in d_v be $\{t_1, t_2, \ldots, t_n\}$ and the set of values in d_u be $\{q_1, q_2, \ldots, q_k\}$. $valueSim(d_v, d_u)$ is computed as follows:

1. Calculate all pair-wise value (i.e., (t_i, q_j)) similarities using the cosine measure with one value from each domain.
2. Choose the pair with the maximum similarity among all pairs and delete the corresponding two values from d_v and d_u. For a pair to be considered, its similarity must be greater than a threshold value τ.
3. Repeat step 2 on all remaining values in the domains until no pair of values has a similarity greater than τ.

Let the pairs of values chosen be P. $valueSim(d_v, d_u)$ is then computed using the **Dice function** [136]:

$$valueSim(d_v, d_u) = \frac{2|P|}{|d_v| + |d_u|}. \tag{5}$$

For two numeric domains, their value similarity is the proportion of the overlapping range of the domains. For an attribute whose domain is unknown, it is assumed that its domain is dissimilar to the domain of any other attribute, be it finite or infinite.

Identify a Preliminary Set of 1:*m* Mappings: To identify 1:*m* mappings, the technique exploits the hierarchical organization of the interfaces. The hierarchical organization is determined using the layout and the proximity of attributes as they are likely to be physically close to each other.

Part-of type: To identify the initial set of aggregate 1:*m* mappings of attributes, it first finds all composite attributes in all interfaces as discussed in Sect. 10.4. For each composite attribute e in S, in every interface other than S, denoted by X, it looks for a set of attributes $f = \{f_1, f_2, \ldots f_r\}$ $(r > 1)$ with the same parent p, such that the following conditions hold:

1. f_i's are siblings, i.e., they share the same parent p. The sibling information is derived from the physical proximity in the interface.
2. The label of the parent p of f_i's is highly similar to the label of e.
3. The domains of f_i's have a 1-to-1 mapping with a subset of the subdomains of e based on the high domain similarities.

If there exists such a f in interface X, a 1:*m* mapping of the part-of type is identified between e and attributes in f.

Is-a type: The identification of is-a 1:*m* attribute mappings requires that the domain of each corresponding sub-attribute on the *m* side be similar to that of the general attribute on the one side. More precisely, for each non-composite attribute h in an interface, we look for a set of attributes $f = \{f_1, f_2, \ldots f_r\}$ $(r > 1)$ in another interface X, that meets the following conditions:

1. f_i's are siblings of the same parent p, and p does not have any children other than f_i's.
2. The label of the parent p is highly similar to the label of h.
3. The domain of each f_i is highly similar to the domain of h.

If the conditions are met, a 1:*m* mapping of the is-a type is identified between h and attributes in f.

Cluster the Schema Elements based on the Similarity Matrix: Step 2 produces a similarity matrix M. Let the total number of simple domains in the set of all given query interfaces S be w. We then have a $w \times w$ symmetric similarity matrix. $M[i, j]$ is the aggregated similarity of two attributes i and j. For attributes in the same interface, $M[i, j]$ is infinite, which indicate that they should not be put together into a cluster.

The clustering algorithm used is the hierarchical agglomerative clustering algorithm. The stopping criterion is a similarity threshold. That is, when there is no pair of clusters has the similarity greater than the threshold, the algorithm stops. Each output cluster contains a set of 1:1 attribute mappings from different interfaces.

Obtain Additional 1:*m* Mapping: The preliminary set of 1:*m* correspondences may not have found all such mappings. The clustering results may suggest additional 1:*m* mappings. The **transitivity** property can be used here. For example, assume that a composite attribute *e* maps to two attributes f_1 and f_2 in another interface in step 3 and the clustering results suggest that f_1 and f_2 map to h_1 and h_2 in yet another interface. Then, *e* also matches h_1 and h_2.

10.8.2 A Correlation Based Approach

This technique also makes use of a large number of interfaces. It is based on the technique in [229]. For pre-processing, the methods discussed in Sect. 10.2 are applied. The approach is based on co-occurrences of schema attributes and the following observations:

1. In an interface, some attributes may be grouped together to form a bigger concept. For example, "first name" and "last name" compose the name of a person. This is called the **grouping relationship**, denoted by a set, e.g., {first name, last name}. Attributes in such a group often co-occur in schemas, i.e., they are **positively correlated**.
2. An attribute group rarely co-occurs in schemas with their synonym attribute groups. For example, "first name" and "last name" rarely co-occur with name in the same query interface. Thus, {first name, last name} and {name} are **negatively correlated**. They represent 2:1 match. Note that a group may contain only one attribute.

Based on the two observations, a correlation-based method to schema matching is in order. Negatively correlated groups represent **synonym groups** or **matching groups**.

Given a set of input schemas $S = \{S_1, S_2, \ldots, S_n\}$ in the same application domain, where each schema S_i is a transaction of attributes, we want to find all the matches $M = \{m_1, \ldots, m_v\}$. Each m_j is a complex matching $g_{j1} = g_{j2} = \ldots = g_{jw}$, where each g_{jk} is an positively correlated attribute group and $g_{jk} \subseteq \bigcup_{i=1}^{n} S_i$. Each m_j represents the synonym relationship of attribute groups $g_{j1}, \ldots, g_{jw}$. The approach for finding *M* consists of three steps:

1. **Group discovery:** This step mines co-occurring or positively correlated attribute groups. It is done by first finding the set of all 2-attribute groups (i.e., each group contains only two attributes), denoted by L_2, that are positively correlated according to the input schema set *S* (one data scan is needed). A 2-attribute group {a, b} is considered positively correlated if c_p(a, b) is greater than a threshold value τ_p, where c_p is a positive correlation measure. The algorithm then extends 2-attribute

groups to 3-attribute groups L_3. A 3-attribute group g is considered positively correlated if every 2-attribute subset of g is in L_2. In general, a k-attribute group g is in L_k if every (k–1)-attribute sub-group of g is in L_{k-1}. This is similar to candidate generation in the Apriori algorithm for association rule mining (see Chap. 2). However, the method here does not scan the data after all 2-attribute groups have been generated.

Example 5: Let L_2 = {{a, b}, {b, c}, {a, c}, {c, d}, {d, f}}, which contains all 2-attribute groups that are discovered from the data. {a, b, c} is in L_3, but {a, c, d} is not because {a, d} is not in L_2. ■

2. **Match discovery:** This step mines negatively correlated groups including those singleton groups. Each discovered positively correlated group is first added into those transactions in S that contain some attributes of the group. That is, for a schema S_i and a group g, if $S_i \cap g \neq \emptyset$, then $S_i = S_i \cup \{g\}$. The final augmented transaction set S is then used to mine negatively correlated groups; which are potential **matching groups**. The procedure for finding all negatively correlated groups is exactly the same as the above procedure for finding positively correlated groups. The only difference is that a different measure is used to determine negative correlations, which will be discussed shortly. A 2-attribute group {a, b} is considered negatively correlated if c_n(a, b) is greater than a threshold value τ_n, where c_n is a negative correlation measure.
3. **Matching selection:** The discovered negative correlations may contain conflicts due to the idiosyncrasy of the data. Some correlations may also subsume others. For instance, in the book domain, the mining result may contain both {author} = {first name, last name}, denoted by m_1 and {subject} = {first name, last name}, denoted by m_2. Clearly, m_1 is correct, but m_2 is not. Since {subject} = {author} is not discovered, which should be due to transitivity of synonyms, m_1 and m_2 cannot be both correct. This causes a **conflict**. A match m_j semantically **subsumes** a match m_k, denoted by $m_j \succeq m_k$, if all the semantic relationships in m_k are contained in m_j. For instance, {arrival city} = {destination} = {to} $\succeq$ {arrival city} = {destination} because the synonym relationship in the second match is subsumed by the first one. Also, {author} = {first name, last name} $\succeq$ {author} = {first name} because the second match is part of the first.

We now present a method to choose the most confident and consistent matches and to remove possibly false ones. Between conflicting matches, we want to select the most negatively correlated one because it is more likely to be a group of genuine synonyms. Thus, a score function is needed, which is defined as the maximum negative correlation values of all 2-attribute groups in the match:

$$score(m_j, c_n) = max\ c_n(g_{j_r}, g_{j_t}),\ g_{j_r}, g_{j_t} \in m_j, j_r \neq j_t. \quad (6)$$

Combining the score function and semantic subsumption, the matches are ranked based on the following rules:

1. If $score(m_j, c_n) > score(m_k, c_n)$, m_j is ranked higher than m_k.
2. If $score(m_j, c_n) = score(m_k, c_n)$ and $m_j \succeq m_k$, m_j is ranked higher than m_k.
3. Otherwise, m_j and m_k are ranked arbitrarily.

Figure 10.4 gives the MatchingSelection() function. After the highest ranked match m_t in an iteration is selected, the inconsistent parts in the remaining matches are removed (lines 6–10). The final output is the selected n-ary complex matches with no conflict. Note that ranking is redone in each iteration instead of sorting all the matches in the beginning, because after removing some conflicting parts, the ranking may change.

```
Function MatchingSelection(M, c_n)
1  R ← ∅                          // R stores the selected n-ary complex matches
2  while M ≠ ∅ do
4     Let m_t be the highest ranked match in M  //select the top ranked match
5     R ← R ∪ {m_t}
6     for each m_j ∈ M do
7        m_j ← m_j – m_t;          // remove the conflicting part
8        if |m_j | < 2 then
9           M ← M – {m_j} // delete m_j if it contains no matching
10    endfor
11 endwhile
12 return R
```

Fig. 10.4. The MatchingSelection function

Correlation Measures: There are many existing correlation tests in statistic, e.g., χ^2 test and lift, etc. However, it was found that these methods were not suitable for this application. Hence, a new negative correlation measure $corr_n$ for two attributes A_p and A_q was proposed, which is called the **H-measure**. Let us use a contingency table (Fig. 10.5) to define it. f_{ij} in the figure is the co-occurrence frequency count of the corresponding cell:

	A_p	$\neg A_p$	
A_q	f_{11}	f_{10}	f_{1+}
$\neg A_q$	f_{01}	f_{00}	f_{0+}
	f_{+1}	f_{+0}	f_{++}

Fig. 10.5. Contingency table for test of correlation

$$corr_n(A_p, A_q) = H(A_p, A_q) = \frac{f_{01} f_{10}}{f_{+1} f_{1+}}. \quad (7)$$

The positive correlation measure $corr_p$ is defined as (τ_d is a threshold):

$$corr_p(A_p, A_q) = \begin{cases} 1 - H(A_p, A_q) & \frac{f_{11}}{f_{++}} < \tau_d \\ 0 & \text{otherwise.} \end{cases} \quad (8)$$

10.8.3 An Instance Based Approach

This method is based on the technique given in [531]. It matches query interfaces and also the query results. It assumes that:

1. a global schema (GS) for the application domain is given, which represents the key attributes of the domain, and
2. a number of sample data instances under the domain global schema are also available.

This technique only finds 1:1 attribute matches. We use IS to denote the query interface schema and RS the returned result schema. Let us use an example to introduce the key observation exploited in this technique. Figure 10.6 shows an example of an online bookstore. The part labeled **Data Attributes** is the global schema with six attributes {Title, Author, Publisher, ISBN, Publication Date, Format}. The part labeled **Interface** is the query interface with five input elements/attributes. When the keyword query "Harry Potter" is submitted through the Title attribute in the interface, a result page is returned which contains the answer to the query (labeled **Result Page**), which shows three book instances.

Three types of semantic correspondence represented by different lines (dotted, dashed and solid) are also shown in Fig. 10.6. They are respectively, the correspondence between attributes of the global schema and those of the query interface, the correspondence between the attributes of the global schema and those of the instance values in the result page, and the correspondence between attributes in the query interface and those of the instance values in the result pages.

Observation: When a proper query is submitted to the right element of the query interface, the query words are very likely to reappear in the corresponding attribute of the returned results. However, if an improper query is submitted to the Web database there are often few or no returned results.

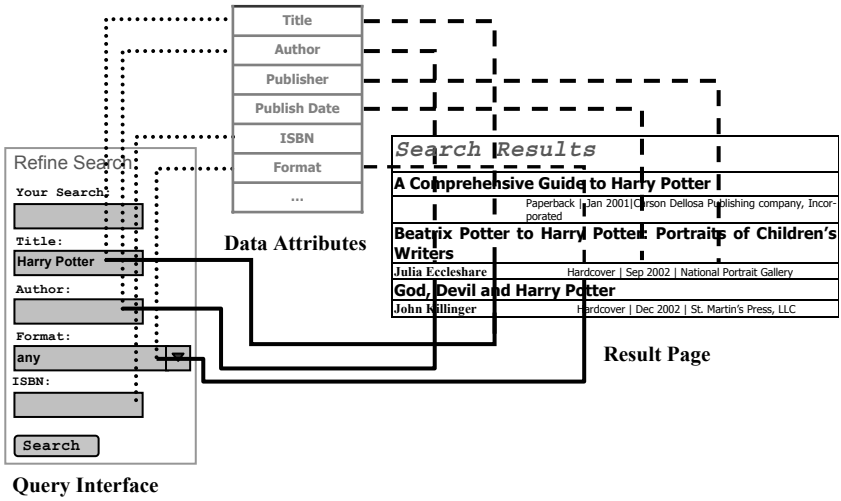

Fig. 10.6. An example of a Web database with its query interface and a result page

In the example shown in Fig. 10.6, the site retrieves only three matches for the query "Harry Potter" when submitted through the "Author" attribute, while it retrieves 228 matches for the same query when submitted to the Title attribute. If "Harry Potter" is submitted to the "ISBN" attribute, there is no returned result. Intuitively, the number of times that query words reappear in the returned results gives us a good indication what attributes match in the interface schema, the global schema, and the result schema.

To obtain the number of reappearing occurrences, each value from the given instances can be submitted to each interface element while keeping default values for the other elements. For each TEXTBOX element in the query interface, all attribute values from the given instances are tried exhaustively. For each SELECT element, its domain values are limited to a set of fixed options. Then, an option similar to a value in the given instances is found and submitted. Here, "similar" means that the attribute value and the option value have at least one common word. Note that this approach assumes that a data extraction system is available to produce a table from a returned result page (see Chap. 9). Each column has a hidden attribute (i.e., of the result schema).

By counting the number of times that the query words re-occur in each column of the result table, a 3-dimensional **occurrence matrix** (OM) can be constructed. The three dimensions are: global schema (GS) attributes, query interface schema (IS) attributes and result schema (RS) attributes. Each cell $OM[i, j, k]$ contains the sum of the occurrence counts obtained from kth attribute of RS of all the sample query words from the ith attribute of GS when the query words are submitted to the jth attribute of IS.

Intra-Site Schema Matching: We now briefly describe how to match attributes in IS and GS, IS and RS, and GS and RS based on the projected matrices of OM, i.e., $OM_{IG(M\times N)}$, $OM_{IR(M\times L)}$, and $OM_{GR(N\times L)}$, where N is the number of attributes in the global schema, M is the number of elements in the interface schema, and L is the number of columns in the result table. An example $OM_{IG(5\times 4)}$ matrix is shown in Fig. 10.7 with the correct matching highlighted, GS = {Title_{GS}, Author_{GS}, Publisher_{GS}, ISBN_{GS}} and IS = {Author_{IS}, Title_{IS}, Publisher_{IS}, Keyword_{IS}, ISBN_{IS}}.

We observe from Fig. 10.7 that the highest occurrence count may not represent a correct match. For example, the cell for Author_{IS} and Publisher_{GS} (534) has the highest value in the matrix but Author_{IS} and Publisher_{GS} do not correspond to each other. In general, for a cell m_{ij}, its value in comparison with those of other cells in its row i and its column j is more important than its absolute count.

	Title_{GS}	Author_{GS}	Publisher_{GS}	ISBN_{GS}
Author_{IS}	93	498	534	0
Title_{IS}	451	345	501	0
Publisher_{IS}	62	184	468	2
Keyword_{IS}	120	248	143	275
ISBN_{IS}	0	0	0	258

Fig. 10.7. An example of a $OM_{IG(M\times N)}$ matrix with all matches highlighted

The algorithm in [531] uses the **mutual information measure** (MI) to determine correct matches. The mutual information, which measures the mutual dependence of two variables, is defined as follows:

$$MI(x,y) = \Pr(x,y)\log_2 \frac{\Pr(x,y)}{\Pr(x)\Pr(y)}. \quad (9)$$

In our context, x and y are attributes from IS and GS respectively. The probabilities, $\Pr(x, y)$, $\Pr(x)$ and $\Pr(y)$, can be easily computed using the $OM_{IG(M\times N)}$ matrix.

The algorithm simply computes the mutual information of every pair of attributes based on the counts in the matrix such as the one in Fig. 10.7. A corresponding mutual information matrix (called **MI matrix**) is then constructed (not shown here). To find 1-1 matches of the two schemas, the algorithm chooses each cell in the MI matrix whose value is the largest among all the values in the same row and the same column. The corresponding attributes of the cell forms a final match.

The paper also has a similar method for finding matches from multiple Web databases, which is called **inter-site schema matching**.

10.9 Constructing a Unified Global Query Interface

Once a set of query interfaces in the same domain is matched, we can automatically construct a *well-designed* **global query interface** that contains all the (or the most significant) distinct attributes of all source interfaces. To build a "good" global interface, three requirements are identified in [154].

1. **Structural appropriateness:** As noted earlier, elements of query interfaces are usually organized in groups (logical units) of related attributes so that semantically related attributes are placed in close vicinity. For example, "Adults", "Seniors", and "Children" of the interfaces shown in Fig. 10.1 are placed together. In addition, multiple related groups of attributes are organized as super-groups (e.g., "Where and when do you want to go?" in Fig. 10.1). This leads to a hierarchical structure for interfaces (see Fig. 10.8), where a leaf in the tree corresponds to an attribute in the interface, an internal node corresponds to a (super)group of attributes and the order among the sibling nodes within the tree resembles the order of attributes in the interface (from left to right). The global query interface should reflect this hierarchical structure of the domain.
2. **Lexical appropriateness:** Labels of elements should be chosen so as to convey the meaning of each individual element and to underline the hierarchical organization of attributes (e.g., the three attributes together with the parent attribute "Number of Passengers" in Fig. 10.1).
3. **Instance appropriateness:** The domain values for each attribute in the global interface must contain the values of the source interfaces.

We will give a high level description of the algorithms in [153, 154] that build the global interface by merging given interfaces based on the above three requirements. The input to the algorithms consists of (1) a set of query interfaces and (2) a global mapping of corresponding attributes in the query interfaces. It is assumed that mapping is organized in clusters as discussed in Sect. 10.8.1. Each cluster contains all the matched attributes from different interfaces. We note that the domain model discovery idea in [227] can be seen as another approach to building global interfaces.

10.9.1 Structural Appropriateness and the Merge Algorithm

Structural appropriateness means to satisfy grouping constraints and ancestor-descendant relationship constraints of the attributes in individual interfaces. These constraints guide the merging algorithm to produce the global interface, which has one attribute for each cluster.

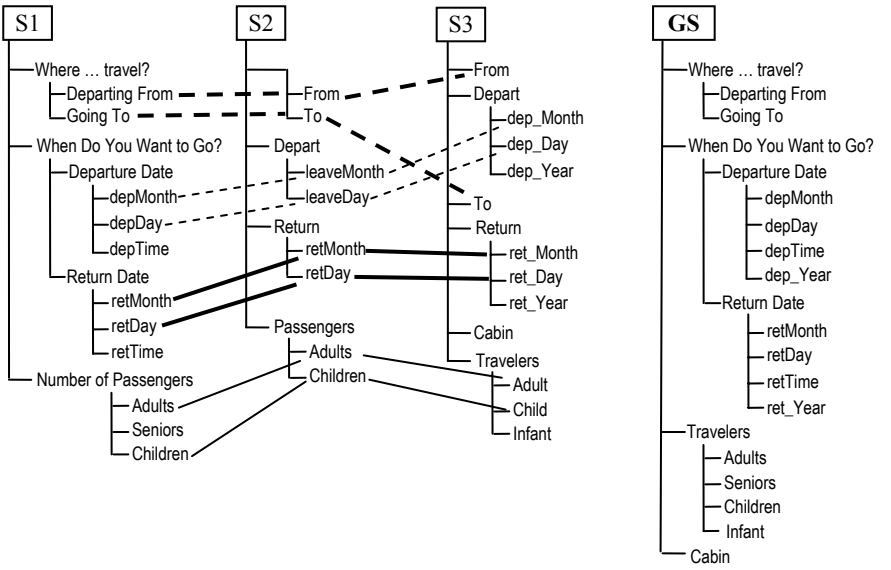

Fig. 10.8. Three input query interfaces (S1, S2, and S3) and the derived global query interface (GS).

Grouping Constraints: Recall that semantically related attributes within an interface are usually grouped together. Grouping constraints require that these attributes should also appear together in the global interface.

As the global interface has an attribute for each cluster, the problem is to partition all the clusters into semantically meaningful subsets (or groups), which are employed to organize attributes in the global interface. For instance, for the example in Fig. 10.8, the following sets of clusters are produced, {c_deptCity, c_destCity}, {c_deptYear, c_deptTime, c_deptDay, c_depMonth}, and {c_Senior, c_Adult, c_Child, c_Infant}, where c_X is a cluster representing X (e.g., c_deptCity and c_destCity are clusters representing departure cities and destination cities, respectively).

The partition is determined by considering each *maximal set* of adjacent sibling leaves in the schema tree of each source interface whose parent is not the root. The leaves whose parent is the root are not considered because no reliable information can be derived. These structural constraints are collected from all source interfaces in order to infer the way that attributes are organized in the global interface. All those sets (or groups) of clusters whose intersection is not empty are merged to form the final groups, which are sequences of attribute clusters that preserve adjacency constraints in all interfaces. For example, {c_Adult, c_Senior, c_Child}, {c_Adult, c_Child}, {c_Adult, c_Child, c_Infant} are merged to produce the final group, [c_Senior, c_Adult, c_Child, c_Infant], which preserves all

adjacency constraints. Such a sequence does not always exist. In such a case, a sequence that accommodates most adjacency constraints is sought.

Ancestor-Descendant Relationships: In hierarchical modeling of data the same information can be represented in various ways. For instance, the relationship between "Authors" and "Books" can be captured as either "Authors" having "Books", which makes "Books" a descendant of "Authors", or "Books" having "Authors", which makes "Books" an ancestor of "Authors". This, however, was not found to be a problem [153]. No such conflicting cases were found from a study of 300 query interfaces in eight application domains.

Merge Algorithm: The merge algorithm merges two interfaces at a time to produce the final global interface schema. One of them is the current global interface G. At the beginning, the schema tree with the most levels is chosen as the initial global schema G. Then each other interface is sequentially merged with G. During each merge, G is refined and expanded. The algorithm works in a bottom-up fashion. The merging between leaves is produced based on the clusters. The mapping between internal nodes is based on mappings of their children, which may be either leaf nodes or internal nodes. To meaningfully insert leaves without a match in the correct position, the algorithm relies on groups computed above to infer each leaf position. In our example, we start by merging S_1 and S_3. S_1 is the initial global interface G. Within each group, it is easy to see the position of "Infant", "ret_Year" and "dep_Year" (see Fig. 10.8 on the right). "Cabin" is inserted at the end since leaf children of the root are discarded before merging and then added as children of the root of the integrated schema tree. Additional details can be found in [153].

10.9.2 Lexical Appropriateness

After the interfaces are merged, the attributes in the integrated interface need to be labeled so that (1) the labels of the attributes within a group are consistent and (2) the labels of the internal nodes are consistent with respect to themselves and to the leaf nodes [154].

It can be observed in the query interface of Fig. 10.1 that between the labels of the attributes grouped together there are certain commonalities. For instance, "Adults", "Seniors" and "Children" are all plurals, whereas "Leaving" and "Returning" are gerunds. Ideally, the groups within the global interface should have the same uniformity property. Since the attributes may be from different interfaces, a group of attributes within the unified interface might not correspond to any group in a single interface,

which makes it hard to assign consistent labels. To deal with the problem, a strategy called **intersect-and-union** is used, which finds groups with non-empty intersection from different interfaces and then unions them.

Example 6: Consider the example of the three interfaces in Fig. 10.8 with their passenger related groups organized as the table below. It is easy to see a systematic way of building a consistent solution.

Cluster/Interface	c_Adult	c_Senior	c_Child	c_Infant
S_1	Adults	Seniors	Children	
S_2	Adults		Children	
S_3	Adult		Child	Infant

Notice that by combining the labels given by S_1 and S_2 a consistent naming assignment, namely, "Seniors", "Adults" and "Children", can be achieved because the two sets share labels (i.e., "Adults" and "Children") that are consistent with the labels in both sets. This strategy can be iteratively applied until a label is assigned to each attribute in the group.

To deal with minor variations, more relaxed rules for combining attribute labels can be used, e.g., requiring that the set of tokens of the labels to be equal after removal of stopwords (e.g., "Number of Adults" has the same set of tokens as "Adults Number", i.e. {Number, Adults}) and stemming. If a consistent solution for the entire group cannot be found, consistent solutions for subsets of attributes are constructed.

The assignment of consistent labels to the internal nodes uses a set of rules [154] that tries to select a label for each node in such a way that it is generic enough to semantically cover the set of its descendant leaf nodes. For example, the label "Travelers" is obtained in the integrated interface in Fig. 10.8 as follows. First, we know that "Passengers" is more generic than "Number of Passengers" and thus semantically covers both {Seniors, Adults, Children} and {Adults, Children}. Then, "Travelers" is found to be a hypernym of "Passengers" (using WordNet) and thus semantically covers the union of {Seniors, Adults, Children} and {Adults, Children, Infant} which is the desired set {Seniors, Adults, Children, Infant}

10.9.3 Instance Appropriateness

Finally, we discuss how to determine the domain for each attribute in the global schema (interface). A domain has two aspects: the type and the set of values. To determine the domain type of a global attribute, compatibility rules are needed [230]. For instance, if all attributes in a cluster have a finite (infinite) domain then the global attribute will have a finite (infinite) domain. If in the cluster there are both finite and infinite domains, then the

domain of the global attribute will be **hybrid** (i.e., users can either select from a list of pre-compiled values or fill in a new value). As a case in point, the "Adults" attribute on the global interface derived from the two interface in Fig. 10.1 will have a finite domain, whereas the attribute "Going to" will have a hybrid domain.

The set of domain values of a global attribute is given by the union of the domains of the attributes in the cluster. Computing the union is not always easy. For example, the values of the domains may have different *scale/unit* (e.g., the price may be in US$ or in Euro). Moreover, the same value may be specified in various ways (e.g., "Chicago O'Hare" vs. "ORD"). Currently, the problem is dealt with using user-provided auxiliary thesauruses [230].

Bibliographic Notes

Schema integration has been studied in the database community since the early 1980s. The main contributions are described in the surveys by Batini et al [40], Doan and Halevy [146], Kalfoglou and Schorlemmer [265], Larson et al. [306], Kashyap and Sheth [269], Rahm and Bernstein [455], Sheth and Larson [488], and Shvaiko and Euzenat [491]. The database integration aspects of this chapter are mainly based on the survey paper by Rahm and Bernstein [455]. Many ideas are also taken from Clifton et al. [105], Cohen [107], Do and Rahm [144], Dhamankar et al. [133], Embley et al. [162], Madhavan et al. [358], Xu and Embley [563], and Yan et al. [566]. Web data integration is considerably more recent. Various ideas on Web information integration in the early part of the chapter are taken from papers by He and Chang [227, 229], and Wu et al. [559].

On Web query interface integration, which perhaps received the most attention in the research community, several methods have been studied in the chapter, which are based on the works of Dragut et al. [153, 154], He and Chang [227, 229], He et al. [230], Wang et al. [531], and Wu et al. [559]. Before matching can be performed, the Web interfaces have to be found and extracted first. This extraction task was investigated by Zhang et al. [609] and He et al. [231].

Another area of research is the ontology, taxonomy or catalog integration. Ontologies (taxonomies or catalogs) are tree structured schemas. They are similar to query interfaces as most interfaces have some hierarchical structures. More focused works on ontology integration include those by Agrawal and Srikant [13], Doan et al. [147], Gal et al. [190], Zhang and Lee [602]. Wache et al. gave a survey of the area in [527].