Chapter 7
Database Integration—Multidatabase Systems

Up to this point, we considered distributed DBMSs that are designed in a top-down fashion. In particular, Chap. 2 focuses on techniques for partitioning and allocating a database, while Chap. 4 focuses on distributed query processing over such a database. These techniques and approaches are suitable for tightly integrated, homogeneous distributed DBMSs. In this chapter, we focus on distributed databases that are designed in a bottom-up fashion—we referred to these as multidatabase systems in Chap. 1. In this case, a number of databases already exist, and the design task involves integrating them into one database. The starting point of bottom-up design is the set of individual local conceptual schemas (LCSs). The process consists of integrating local databases with their (local) schemas into a global database and generating a global conceptual schema (GCS) (also called the *mediated schema*). Querying over a multidatabase system is more complicated in that applications and users can either query using the GCS (or views defined on it) or through the LCSs since each existing local database may already have applications running on it. Therefore, the techniques required for query processing require adjustments to the approach we discussed in Chap. 4 although many of those techniques carry over.

Database integration, and the related problem of querying multidatabases, is only one part of the more general *interoperability* problem, which includes nondatabase data sources and interoperability at the application level in addition to the database level. We separate this discussion into three pieces: in this chapter, we focus on the database integration and querying issues, we discuss the concerns related to web data integration and access in Chap. 12, and we discuss the more general issue of integrating data from arbitrary data sources in Chap. 10 under the title *data lakes*.

This chapter consists of two main sections. In Sect. 7.1, we discuss database integration—the bottom-up design process. In Sect. 7.2 we discuss approaches to querying these systems.

The original version of this chapter was revised. The correction to this chapter is available at https://doi.org/10.1007/978-3-030-26253-2_13

M. T. Özsu, P. Valduriez, *Principles of Distributed Database Systems*,
https://doi.org/10.1007/978-3-030-26253-2_7

7.1 Database Integration

Database integration can be either physical or logical. In the former, the source databases are integrated and the integrated database is *materialized*. These are known as *data warehouses*. The integration is aided by *extract–transform–load* (ETL) tools that enable extraction of data from sources, its transformation to match the GCS, and its loading (i.e., materialization). This process is depicted in Fig. 7.1. In logical integration, the global conceptual (or mediated) schema is entirely *virtual* and not materialized.

These two approaches are complementary and address differing needs. Data warehousing supports decision-support applications, which are commonly termed *Online Analytical Processing* (OLAP). Recall from Chap. 5 that OLAP applications analyze historical, summarized data coming from a number of operational databases through complex queries over potentially very large tables. Consequently, data warehouses gather data from a number of operational databases and materialize it. As updates happen on the operational databases, they are propagated to the data warehouse, which is known as *materialized view maintenance*.

By contrast, in logical data integration, the integration is only virtual and there is no materialized global database (see Fig. 1.13). The data resides in the operational databases and the GCS provides a virtual integration for querying over the multiple databases. In these systems, GCS may either be defined up-front and local databases (i.e., LCSs) mapped to it, or it may be defined bottom-up, by integrating parts of the LCSs of the local databases. Consequently, it is possible for the GCS not to capture

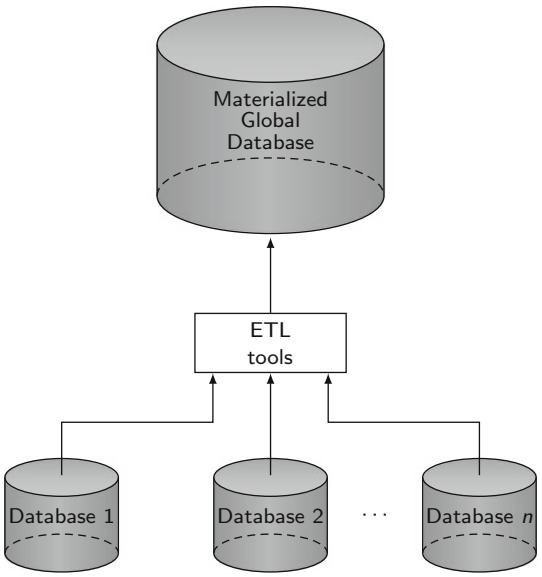

Fig. 7.1 Data warehouse approach

all of the information in each of the LCSs. User queries are posed over this global schema, which are then decomposed and shipped to the local operational databases for processing as is done in tightly integrated systems, with the main difference being the autonomy and potential heterogeneity of the local systems. These have important effects on query processing that we discuss in Sect. 7.2. Although there is ample work on transaction management in these systems, supporting global updates is quite difficult given the autonomy of the underlying operational DBMSs. Therefore, they are primarily read-only.

Logical data integration and the resulting systems are known by a variety of names; *data integration* and *information integration* are perhaps the most common terms used in literature although these generally refer to more than database integration and incorporate data from a variety of sources. In this chapter, we focus on the integration of autonomous and (possibly) heterogeneous databases; thus, we will use the term *database integration* or *multidatabase systems* (MDBSs).

7.1.1 Bottom-Up Design Methodology

Bottom-up design involves the process by which data from participating databases can be (physically or logically) integrated to form a single cohesive global database. As noted above, in some cases, the global conceptual (or mediated) schema is defined first, in which case the bottom-up design involves mapping LCSs to this schema. In other cases, the GCS is defined as an integration of parts of LCSs. In this case, the bottom-up design involves both the generation of the GCS and the mapping of individual LCSs to this GCS.

If the GCS is defined upfront, the relationship between the GCS and the LCSs can be of two fundamental types: local-as-view and global-as-view. In local-as-view (LAV) systems, the GCS definition exists, and each LCS is treated as a view definition over it. In global-as-view systems (GAV), on the other hand, the GCS is defined as a set of views over the LCSs. These views indicate how the elements of the GCS can be derived, when needed, from the elements of LCSs. One way to think of the difference between the two is in terms of the results that can be obtained from each system . In GAV, the query results are constrained to the set of objects that are defined in the GCS, although the local DBMSs may be considerably richer (Fig. 7.2a). In LAV, on the other hand, the results are constrained by the objects in the local DBMSs, while the GCS definition may be richer (Fig. 7.2b). Thus, in LAV systems, it may be necessary to deal with incomplete answers. A combination of these two approaches has also been proposed as global-local-as-view (GLAV) where the relationship between GCS and LCSs is specified using both LAV and GAV.

Bottom-up design occurs in two general steps (Fig. 7.3): *schema translation* (or simply *translation*) and *schema generation*. In the first step, the component database schemas are translated to a common intermediate canonical representation (InS_1, InS_2,. . ., InS_n). The use of a canonical representation facilitates the trans-

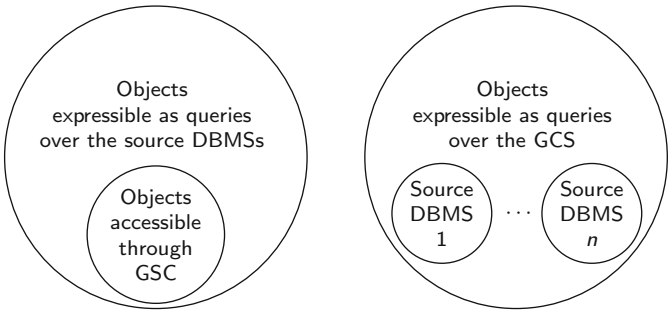

Fig. 7.2 GAV and LAV mappings (based on [Koch 2001])

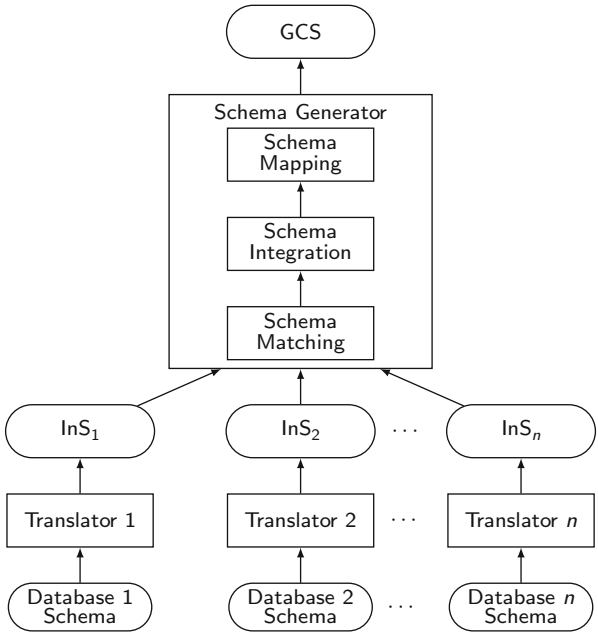

Fig. 7.3 Database integration process

lation process by reducing the number of translators that need to be written. The choice of the canonical model is important. As a principle, it should be one that is sufficiently expressive to incorporate the concepts available in all the databases that will later be integrated. Alternatives that have been used include the entity-relationship model, object-oriented model, or a graph that may be simplified to a tree or XML. In this chapter, we will simply use the relational model as our canonical data model despite its known deficiencies in representing rich semantic concepts. This choice does not affect in any fundamental way the discussion of the major

issues of data integration. In any case, we will not discuss the specifics of translating various data models to relational; this can be found in many database textbooks.

Clearly, the translation step is necessary only if the component databases are heterogeneous and local schemas are defined using different data models. There has been some work on the development of system federation, in which systems with similar data models are integrated together (e.g., relational systems are integrated into one conceptual schema and, perhaps, object databases are integrated to another schema) and these integrated schemas are "combined" at a later stage (e.g., AURORA project). In this case, the translation step is delayed, providing increased flexibility for applications to access underlying data sources in a manner that is suitable for their needs.

In the second step of bottom-up design, the intermediate schemas are used to generate a GCS. The schema generation process consists of the following steps:

1. Schema matching to determine the syntactic and semantic correspondences among the translated LCS elements or between individual LCS elements and the predefined GCS elements (Sect. 7.1.2).
2. Integration of the common schema elements into a global conceptual (mediated) schema if one has not yet been defined (Sect. 7.1.3).
3. Schema mapping that determines how to map the elements of each LCS to the other elements of the GCS (Sect. 7.1.4).

It is also possible that the schema mapping step be divided into two phases: mapping constraint generation and transformation generation. In the first phase, given correspondences between two schemas, a transformation function such as a query or view definition over the source schema is generated that would "populate" the target schema. In the second phase, an executable code is generated corresponding to this transformation function that would actually generate a target database consistent with these constraints. In some cases, the constraints are implicitly included in the correspondences, eliminating the need for the first phase.

Example 7.1 To facilitate our discussion of global schema design in multidatabase systems, we will use an example that is an extension of the engineering database we have been using throughout the book. To demonstrate both phases of the database integration process, we introduce some data model heterogeneity into our example.

Consider two organizations, each with their own database definitions. One is the (relational) database example that we introduced in Chap. 2. We repeat that definition in Fig. 7.4 for completeness. The second database also defines similar data, but is specified according to the entity-relationship (E-R) data model as depicted in Fig. 7.5.[1]

We assume that the reader is familiar with the entity-relationship data model. Therefore, we will not describe the formalism, except to make the following points regarding the semantics of Fig. 7.5. This database is similar to the relational

[1] In this chapter, we continue our notation of typesetting relation names in `typewriter` font, but we will use normal font for E-R model components to be able to easily differentiate them.

EMP(ENO, ENAME, TITLE)
PROJ(PNO, PNAME, BUDGET, LOC)
ASG(ENO, PNO, RESP, DUR)
PAY(TITLE, SAL)

Fig. 7.4 Relational engineering database representation

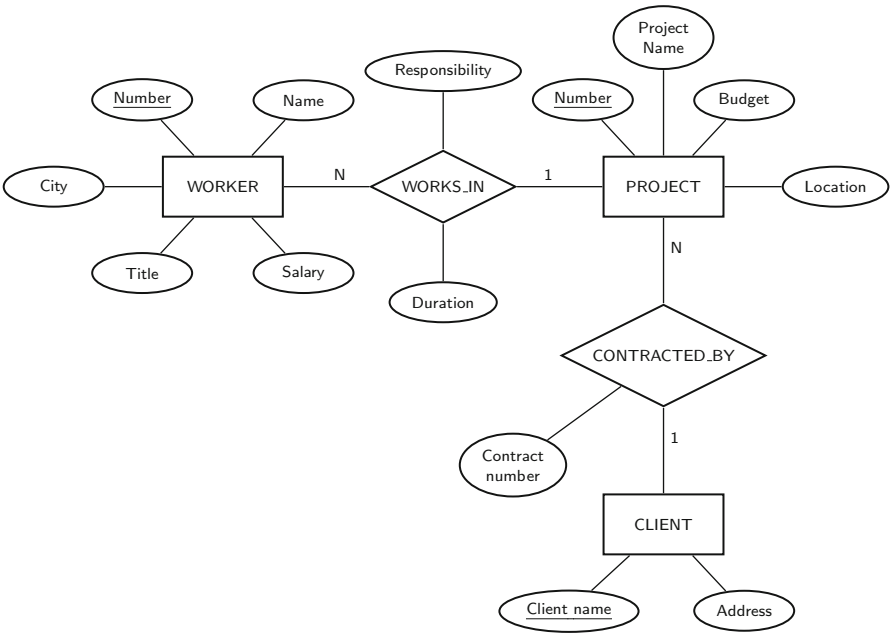

Fig. 7.5 Entity-relationship database

engineering database definition of Fig. 7.4, with one significant difference: it also maintains data about the clients for whom the projects are conducted. The rectangular boxes in Fig. 7.5 represent the entities modeled in the database, and the diamonds indicate a relationship between the entities to which they are connected. The relationship type is indicated around the diamonds. For example, the CONTRACTED-BY relation is a many-to-one from the PROJECT entity to the CLIENT entity (e.g., each project has a single client, but each client can have many projects). Similarly, the WORKS-IN relationship indicates a many-to-many relationship between the two connected relations. The attributes of entities and the relationships are shown as ellipses. ♦

Example 7.2 The mapping of the E-R model to the relational model is given in Fig. 7.6. Note that we have renamed some of the attributes in order to ensure name uniqueness. ♦

```
WORKER(WNUMBER, NAME, TITLE, SALARY,CITY)
PROJECT(PNUMBER, PNAME, BUDGET)
CLIENT(CNAME, ADDRESS)
WORKS_IN(WNUMBER, PNUMBER, RESPONSIBILITY, DURATION)
CONTRACTED_BY(PNUMBER, CNAME, CONTRACTNO)
```

Fig. 7.6 Relational mapping of E-R schema

7.1.2 Schema Matching

Given two schemas, schema matching determines for each concept in one schema what concept in the other matches it. As discussed earlier, if the GCS has already been defined, then one of these schemas is typically the GCS, and the task is to match each LCS to the GCS. Otherwise, matching is done over two LCSs. The matches that are determined in this phase are then used in schema mapping to produce a set of directed mappings, which, when applied to the source schema, would map its concepts to the target schema.

The matches that are defined or discovered during schema matching are specified as a set of rules where each rule (r) identifies a *correspondence* (c) between two elements, a *predicate* (p) that indicates when the correspondence may hold, and a *similarity value* (s) between the two elements identified in the correspondence. A correspondence may simply identify that two concepts are similar (which we will denote by $\approx$) or it may be a function that specifies that one concept may be derived by a computation over the other one (for example, if the budget value of one project is specified in US dollars, while the other one is specified in Euros, the correspondence may specify that one is obtained by multiplying the other one with the appropriate exchange rate). The predicate is a condition that qualifies the correspondence by specifying when it might hold. For example, in the budget example specified above, p may specify that the rule holds only if the location of one project is in US, while the other one is in the Euro zone. The similarity value for each rule can be specified or calculated. Similarity values are real values in the range [0,1]. Thus, a set of matches can be defined as $M = \{r\}$, where $r = \langle c, p, s \rangle$.

As indicated above, correspondences may either be discovered or specified. As much as it is desirable to automate this process, there are many complicating factors. The most important is schema heterogeneity, which refers to the differences in the way real-world phenomena are captured in different schemas. This is a critically important issue, and we devote a separate section to it (Sect. 7.1.2.1). Aside from schema heterogeneity, other issues that complicate the matching process are the following:

- *Insufficient schema and instance information:* Matching algorithms depend on the information that can be extracted from the schema and the existing data instances. In some cases there may be ambiguity due to the insufficient

information provided about these items. For example, using short names or ambiguous abbreviations for concepts, as we have done in our examples, can lead to incorrect matching.
- *Unavailability of schema documentation:* In most cases, the database schemas are not well documented or not documented at all. Quite often, the schema designer is no longer available to guide the process. The lack of these vital information sources adds to the difficulty of matching.
- *Subjectivity of matching:* Finally, it is important to recognize that matching schema elements can be highly subjective; two designers may not agree on a single "correct" mapping. This makes the evaluation of a given algorithm's accuracy significantly difficult.

Nevertheless, algorithmic approaches have been developed to the matching problem, which we discuss in this section. A number of issues affect the particular matching algorithm. The more important ones are the following:

- *Schema versus instance matching.* So far in this chapter, we have been focusing on schema integration; thus, our attention has naturally been on matching concepts of one schema to those of another. A large number of algorithms have been developed that work on schema elements. There are others, however, that have focused instead on the data instances or a combination of schema information and data instances. The argument is that considering data instances can help alleviate some of the semantic issues discussed above. For example, if an attribute name is ambiguous, as in "contact-info," then fetching its data may help identify its meaning; if its data instances have the phone number format, then obviously it is the phone number of the contact agent, while long strings may indicate that it is the contact agent name. Furthermore, there are a large number of attributes, such as postal codes, country names, email addresses, that can be defined easily through their data instances.

  Matching that relies solely on schema information may be more efficient, because it does not require a search over data instances to match the attributes. Furthermore, this approach is the only feasible one when few data instances are available in the matched databases, in which case learning may not be reliable. However, in some cases, e.g., peer-to-peer systems (see Chap. 9), there may not be a schema, in which case instance-based matching is the only appropriate approach.
- *Element-level vs. structure-level.* Some matching algorithms operate on individual schema elements, while others also consider the structural relationships between these elements. The basic concept of the element-level approach is that most of the schema semantics are captured by the elements' names. However, this may fail to find complex mappings that span multiple attributes. Match algorithms that also consider structure are based on the belief that, normally, the structures of matchable schemas tend to be similar.
- *Matching cardinality.* Matching algorithms exhibit various capabilities in terms of cardinality of mappings. The simplest approaches use 1:1 mapping, which means that each element in one schema is matched with exactly one element in

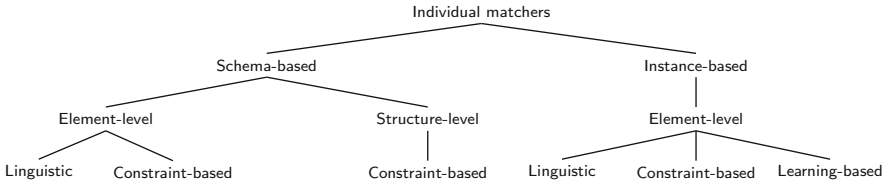

Fig. 7.7 Taxonomy of schema matching techniques

the other schema. The majority of proposed algorithms belong to this category, because problems are greatly simplified in this case. Of course there are many cases where this assumption is not valid. For example, an attribute named "Total price" could be mapped to the sum of two attributes in another schema named "Subtotal" and "Taxes." Such mappings require more complex matching algorithms that consider 1:M and N:M mappings.

These criteria, and others, can be used to come up with a taxonomy of matching approaches. According to this taxonomy (which we will follow in this chapter with some modifications), the first level of separation is between schema-based matchers versus instance-based matchers (Fig. 7.7). Schema-based matchers can be further classified as element-level and structure-level, while for instance-based approaches, only element-level techniques are meaningful. At the lowest level, the techniques are characterized as either linguistic or constraint-based. It is at this level that fundamental differences between matching algorithms are exhibited and we focus on these algorithms in the remainder, discussing linguistic approaches in Sect. 7.1.2.2, constraint-based approaches in Sect. 7.1.2.3, and learning-based techniques in Sect. 7.1.2.4. These are referred as *individual matcher* approaches, and their combinations are possible by developing either *hybrid matchers* or *composite matchers* (Sect. 7.1.2.5).

7.1.2.1 Schema Heterogeneity

Schema matching algorithms deal with both structural heterogeneity and semantic heterogeneity among the matched schemas. We discuss these in this section before presenting the different match algorithms.

Structural conflicts occur in four possible ways: as *type conflicts*, *dependency conflicts*, *key conflicts*, or *behavioral conflicts*. Type conflicts occur when the same object is represented by an attribute in one schema and an entity (relation) in another. Dependency conflicts occur when different relationship modes (e.g., one-to-one versus many-to-many) are used to represent the same thing in different schemas. Key conflicts occur when different candidate keys are available and different primary keys are selected in different schemas. Behavioral conflicts are implied by the modeling mechanism. For example, deleting the last item from one

database may cause the deletion of the containing entity (i.e., deletion of the last employee causes the dissolution of the department).

Example 7.3 We have two structural conflicts in the running example of this chapter. The first is a type conflict involving clients of projects. In the schema of Fig. 7.5, the client of a project is modeled as an entity. In the schema of Fig. 7.4, however, the client is included as an attribute of the PROJ entity.

The second structural conflict is a dependency conflict involving the WORKS_IN relationship in Fig. 7.5 and the ASG relation in Fig. 7.4. In the former, the relationship is many-to-one from the WORKER to the PROJECT, whereas in the latter, the relationship is many-to-many. ♦

Structural differences among schemas are important, but their identification and resolution is not sufficient. Schema matching has to take into account the (possibly different) semantics of the schema concepts. This is referred to as *semantic heterogeneity*, which is a fairly loaded term without a clear definition. It basically refers to the differences among the databases that relate to the meaning, interpretation, and intended use of data. There are attempts to formalize semantic heterogeneity and to establish its link to structural heterogeneity; we will take a more informal approach and discuss some of the semantic heterogeneity issues intuitively. The following are some of these problems that the match algorithms need to deal with.

- *Synonyms*, *homonyms*, *hypernyms*. Synonyms are multiple terms that all refer to the same concept. In our database example, PROJ relation and PROJECT entity refer to the same concept. Homonyms, on the other hand, occur when the same term is used to mean different things in different contexts. Again, in our example, BUDGET may refer to the gross budget in one database and it may refer to the net budget (after some overhead deduction) in another, making their simple comparison difficult. Hypernym is a term that is more generic than a similar word. Although there is no direct example of it in the databases we are considering, the concept of a Vehicle in one database is a hypernym for the concept of a Car in another (incidentally, in this case, Car is a *hyponym* of Vehicle). These problems can be addressed by the use of *domain ontologies* that define the organization of concepts and terms in a particular domain.
- *Different ontology:* Even if domain ontologies are used to deal with issues in one domain, it is quite often the case that schemas from different domains may need to be matched. In this case, one has to be careful of the meaning of terms across ontologies, as they can be highly domain dependent. For example, an attribute called LOAD may imply a measure of resistance in an electrical ontology, but in a mechanical ontology, it may represent a measure of weight.
- *Imprecise wording:* Schemas may contain ambiguous names. For example, the LOCATION (from E-R) and LOC (from relational) attributes in our example database may refer to the full address or just part of it. Similarly, an attribute named "contact-info" may imply that the attribute contains the name of the contact agent or his/her telephone number. These types of ambiguities are common.

7.1.2.2 Linguistic Matching Approaches

Linguistic matching approaches, as the name implies, use element names and other textual information (such as textual descriptions/annotations in schema definitions) to perform matches among elements. In many cases, they may use external sources, such as thesauri, to assist in the process.

Linguistic techniques can be applied in both schema-based approaches and instance-based ones. In the former case, similarities are established among schema elements, whereas in the latter, they are specified among elements of individual data instances. To focus our discussion, we will mostly consider schema-based linguistic matching approaches, briefly mentioning instance-based techniques. Consequently, we will use the notation $\langle$SC1.element-1 $\approx$ SC2.element-2, $p, s\rangle$ to represent that element-1 in schema SC1 corresponds to element-2 in schema SC2 if predicate p holds, with a similarity value of s. Matchers use these rules and similarity values to determine the similarity value of schema elements.

Linguistic matchers that operate at the schema element-level typically deal with the names of the schema elements and handle cases such as synonyms, homonyms, and hypernyms. In some cases, the schema definitions can have annotations (natural language comments) that may be exploited by the linguistic matchers. In the case of instance-based approaches, linguistic matchers focus on information retrieval techniques such as word frequencies, key terms, etc. In these cases, the matchers "deduce" similarities based on these information retrieval measures.

Schema linguistic matchers use a set of linguistic (also called terminological) rules that can be handcrafted or may be "discovered" using auxiliary data sources such as thesauri, e.g., WordNet. In the case of handcrafted rules, the designer needs to specify the predicate p and the similarity value s as well. For discovered rules, these may either be specified by an expert following the discovery, or they may be computed using one of the techniques we will discuss shortly.

The handcrafted linguistic rules may deal with issues such as capitalization, abbreviations, and concept relationships. In some systems, the handcrafted rules are specified for each schema individually (*intraschema rules*) by the designer, and *interschema rules* are then "discovered" by the matching algorithm. However, in most cases, the rule base contains both intra and interschema rules.

Example 7.4 In the relational database of Example 7.2, the set of rules may have been defined (quite intuitively) as follows where RelDB refers to the relational schema and ERDB refers to the translated E-R schema:

$\langle$uppercase names $\approx$ lower case names, $true$, 1.0$)\rangle$
$\langle$uppercase names $\approx$ capitalized names, $true$, 1.0$)\rangle$
$\langle$capitalized names $\approx$ lower case names, $true$, 1.0$)\rangle$
$\langle$RelDB.ASG $\approx$ ERDB.WORKS_IN, $true$, 0.8$\rangle$
...

The first three rules are generic ones specifying how to deal with capitalizations, while the fourth one specifies a similarity between the ASG of RelDB and the WORKS_IN of ERDB. Since these correspondences always hold, $p = true$. ♦

As indicated above, there are ways of determining the element name similarities automatically. For example, COMA uses the following techniques to determine similarity of two element names:

- The *affixes* which are the common prefixes and suffixes between the two element name strings are determined.
- The *n-grams* of the two element name strings are compared. An n-gram is a substring of length n and the similarity is higher if the two strings have more n-grams in common.
- The *edit distance* between two element name strings is computed. The edit distance (also called the Levenshtein metric) determines the number of character modifications (additions, deletions, insertions) that one has to perform on one string to convert it to the second string.
- The *soundex code* of the element names is computed. This gives the phonetic similarity between names based on their soundex codes. Soundex code of English words is obtained by hashing the word to a letter and three numbers. This hash value (roughly) corresponds to how the word would sound. The important aspect of this code in our context is that two words that sound similar will have close soundex codes.

Example 7.5 Consider matching the RESP and the RESPONSIBILITY attributes in the two example schemas we are considering. The rules defined in Example 7.4 take care of the capitalization differences, so we are left with matching RESP with RESPONSIBILITY. Let us consider how the similarity between the two strings can be computed using the edit distance and the n-gram approaches.

The number of editing changes that one needs to do to convert one of these strings to the other is 10 (either we add the characters "O," "N," "S," "I," "B," "I," "L," "I," "T," "Y," to string "RESP" or delete the same characters from string "RESPONSIBILITY"). Thus the ratio of the required changes is $10/14$, which defines the edit distance between these two strings; $1 - (10/14) = 4/14 = 0.29$ is then their similarity.

For n-gram computation, we need to first fix the value of n. For this example, let $n = 3$, so we are looking for 3-grams. The 3-grams of string "RESP" are "RES" and "ESP." Similarly, there are twelve 3-grams of "RESPONSIBILITY": "RES," "ESP," "SPO," "PON," "ONS," "NSI," "SIB," "IBI," "BIP," "ILI," "LIT," and "ITY." There are two matching 3-grams out of twelve, giving a 3-gram similarity of $2/12 = 0.17$. ♦

The examples we have covered in this section all fall into the category of 1:1 matches—we matched one element of a particular schema to an element of another schema. As discussed earlier, it is possible to have 1:N (e.g., Street address, City, and Country element values in one database can be extracted from a single Address element in another), N:1 (e.g., Total_price can be calculated from Subtotal and Taxes

elements), or N:M (e.g., Book_title, Rating information can be extracted via a join of two tables one of which holds book information and the other maintains reader reviews and ratings). 1:1, 1:N, and N:1 matchers are typically used in element-level matching, while schema-level matching can also use N:M matching, since, in the latter case the necessary schema information is available.

7.1.2.3 Constraint-Based Matching Approaches

Schema definitions almost always contain semantic information that constrain the values in the database. These are typically data type information, allowable ranges for data values, key constraints, etc. In the case of instance-based techniques, the existing ranges of the values can be extracted as well as some patterns that exist in the instance data. These can be used by matchers.

Consider data types that capture a large amount of semantic information. This information can be used to disambiguate concepts and also focus the match. For example, `RESP` and RESPONSIBILITY have relatively low similarity values according to calculations in Example 7.5. However, if they have the same data type definition, this may be used to increase their similarity value. Similarly, the data type comparison may differentiate between elements that have high lexical similarity. For example, `ENO` in Fig. 7.4 has the same edit distance and n-gram similarity values to the two NUMBER attributes in Fig. 7.5 (of course, we are referring to the *names* of these attributes). In this case, the data types may be of assistance—if the data type of both `ENO` and worker number (WORKER.NUMBER) is integer, while the data type of project number (PROJECT.NUMBER) is a string, the likelihood of `ENO` matching WORKER.NUMBER is significantly higher.

In structure-based approaches, the structural similarities in the two schemas can be exploited to determine the similarity of the schema elements. If two schema elements are structurally similar, this enhances our confidence that they indeed represent the same concept. For example, if two elements have very different names and we have not been able to establish their similarity through element matchers, but they have the same properties (e.g., same attributes) that have the same data types, then we can be more confident that these two elements may be representing the same concept.

The determination of structural similarity involves checking the similarity of the "neighborhoods" of the two concepts under consideration. Definition of the neighborhood is typically done using a graph representation of the schemas where each concept (relation, entity, attribute) is a vertex and there is a directed edge between two vertices if and only if the two concepts are related (e.g., there is an edge from a relation vertex to each of its attributes, or there is an edge from a foreign key attribute vertex to the primary key attribute vertex it is referencing). In this case, the neighborhood can be defined in terms of the vertices that can be reached within a certain path length of each concept, and the problem reduces to checking the similarity of the subgraphs in this neighborhood. Many of these algorithms consider the tree rooted at the concept that is being examined and compute the

similarity of the concepts represented by the root vertices in the two trees. The fundamental idea is that if the subgraphs (subtrees) are similar, this increases the similarity of the concepts represented by the "root" vertex in the two graphs. The similarity of the subgraphs is typically determined in a bottom-up process, starting at the leaves whose similarity is determined using element matching (e.g., name similarity to the level of synonyms or data type compatibility). The similarity of the two subtrees is recursively determined based on the similarity of the vertices in the subtree. The similarity of two subgraphs (subtrees) is then defined as the fraction of leaves in the two subtrees that are strongly linked. This is based on the assumption that leaf vertices carry more information and that the structural similarity of two nonleaf schema elements is determined by the similarity of the leaf vertices in their respective subtrees, even if their immediate children are not similar. These are heuristic rules and it is possible to define others.

Another interesting approach to considering neighborhood in directed graphs while computing similarity of vertices is *similarity flooding*. It starts from an initial graph where the vertex similarities are already determined by means of an element matcher, and propagates, iteratively, to determine the similarity of each vertex to its neighbors. Hence, whenever any two elements in two schemas are found to be similar, the similarity of their adjacent vertices increases. The iterative process stops when the vertex similarities stabilize. At each iteration, to reduce the amount of work, a subset of the vertices are selected as the "most plausible" matches, which are then considered in the subsequent iteration.

Both of these approaches are agnostic to the edge semantics. In some graph representations, there is additional semantics attached to these edges. For example, *containment edges* from a relation or entity vertex to its attributes may be distinguished from *referential edges* from a foreign key attribute vertex to the corresponding primary key attribute vertex. Some systems (e.g., DIKE) exploit these edge semantics.

7.1.2.4 Learning-Based Matching

A third alternative approach that has been proposed is to use machine learning techniques to determine schema matches. Learning-based approaches formulate the problem as one of classification where concepts from various schemas are classified into classes according to their similarity. The similarity is determined by checking the features of the data instances of the databases that correspond to these schemas. How to classify concepts according to their features is learned by studying the data instances in a training dataset.

The process is as follows (Fig. 7.8). A training set (τ) is prepared that consists of instances of example correspondences between the concepts of two databases D_i and D_j. This training set can be generated after manual identification of the schema correspondences between two databases followed by extraction of example training data instances or by the specification of a query expression that converts data from one database to another. The learner uses this training data to acquire probabilistic

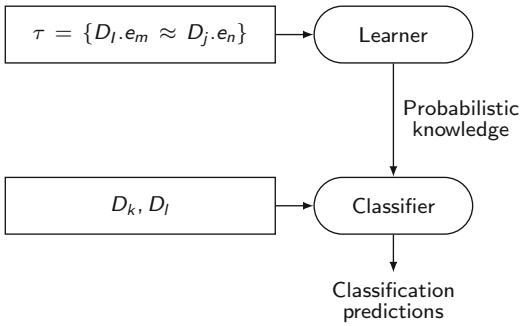

Fig. 7.8 Learning-based matching approach

information about the features of the datasets. The classifier, when given two other database instances (D_k and D_l), then uses this knowledge to go through the data instances in D_k and D_l and make predictions about classifying the elements of D_k and D_l.

This general approach applies to all of the proposed learning-based schema matching approaches. Where they differ is the type of learner that they use and how they adjust this learner's behavior for schema matching. Some have used neural networks (e.g., SEMINT), others have used Naïve Bayesian learner/classifier (Autoplex , LSD) and decision trees. We do not discuss the details of these learning techniques.

7.1.2.5 Combined Matching Approaches

The individual matching techniques that we have considered so far have their strong points and their weaknesses. Each may be more suitable for matching certain cases. Therefore, a "complete" matching algorithm or methodology usually needs to make use of more than one individual matcher.

There are two possible ways in which matchers can be combined: hybrid and composite. *Hybrid* algorithms combine multiple matchers within one algorithm. In other words, elements from two schemas can be compared using a number of element matchers (e.g., string matching as well as data type matching) and/or structural matchers within one algorithm to determine their overall similarity. Careful readers will have noted that in discussing the constraint-based matching algorithms that focused on structural matching, we followed a hybrid approach since they were based on an initial similarity determination of, for example, the leaf nodes using an element matcher, and these similarity values were then used in structural matching. *Composite* algorithms, on the other hand, apply each matcher to the elements of the two schemas (or two instances) individually, obtaining individual similarity scores, and then they apply a method for combining these similarity scores. More precisely, if $s_i(C_j^k, C_l^m)$ is the similarity score using matcher

i ($i = 1, \ldots, q$) over two concepts C_j from schema k and C_l from schema m, then the composite similarity of the two concepts is given by $s(C_j^k, C_l^m) = f(s_1, \ldots, s_q)$, where f is the function that is used to combine the similarity scores. This function can be as simple as average, max, or min, or it can be an adaptation of more complicated ranking aggregation functions that we will discuss further in Sect. 7.2. Composite approach has been proposed in the LSD and iMAP systems for handling 1:1 and N:M matches, respectively.

7.1.3 Schema Integration

Once schema matching is done, the correspondences between the various LCSs have been identified. The next step is to create the GCS, and this is referred to as *schema integration*. As indicated earlier, this step is only necessary if a GCS has not already been defined and matching was performed on individual LCSs. If the GCS was defined upfront, then the matching step would determine correspondences between it and each of the LCSs and there would be no need for the integration step. If the GCS is created as a result of the integration of LCSs based on correspondences identified during schema matching, then, as part of integration, it is important to identify the correspondences between the GCS and the LCSs. Although tools have been developed to aid in the integration process, human involvement is clearly essential.

Example 7.6 There are a number of possible integrations of the two example LCSs we have been discussing. Figure 7.9 shows one possible GCS that can be generated as a result of schema integration. We use this in the remainder of this chapter. ♦

Integration methodologies can be classified as binary or n-ary mechanisms based on the manner in which the local schemas are handled in the first phase (Fig. 7.10). Binary integration methodologies involve the manipulation of two schemas at a time. These can occur in a stepwise (ladder) fashion (Fig. 7.11a) where intermediate schemas are created for integration with subsequent schemas, or in a purely binary fashion (Fig. 7.11b), where each schema is integrated with one other, creating an intermediate schema for integration with other intermediate schemas.

EMP(E#, ENAME, TITLE, CITY)
PAY(TITLE, SAL)
PR(P#, PNAME, BUDGET, LOC)
CL(CNAME, ADDR, CT#, P#)
WORKS(E#, P#, RESP, DUR)

Fig. 7.9 Example integrated GCS (EMP is employee, PR is project, CL is client)

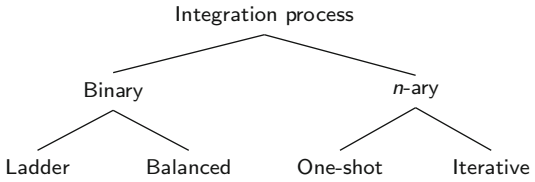

Fig. 7.10 Taxonomy of integration methodologies

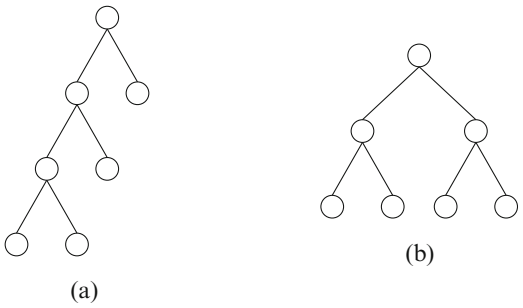

Fig. 7.11 Binary integration methods. (**a**) Stepwise. (**b**) Pure binary

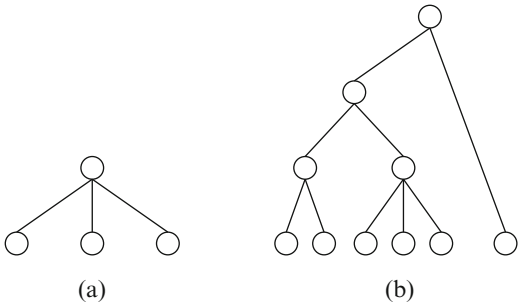

Fig. 7.12 *N*-ary integration methods. (**a**) One-pass. (**b**) Iterative

N-ary integration mechanisms integrate more than two schemas at each iteration. One-pass integration (Fig. 7.12a) occurs when all schemas are integrated at once, producing the global conceptual schema after one iteration. Benefits of this approach include the availability of complete information about all databases at integration time. There is no implied priority for the integration order of schemas, and the trade-offs, such as the best representation for data items or the most understandable structure, can be made between all schemas rather than between a few. Difficulties with this approach include increased complexity and difficulty of automation.

Iterative *n*-ary integration (Fig. 7.12b) offers more flexibility (typically, more information is available) and is more general (the number of schemas can be varied depending on the integrator's preferences). Binary approaches are a special case of

iterative n-ary. They decrease the potential integration complexity and lead towards automation techniques, since the number of schemas to be considered at each step is more manageable. Integration by an n-ary process enables the integrator to perform the operations on more than two schemas. For practical reasons, the majority of systems utilize binary methodology, but a number of researchers prefer the n-ary approach because complete information is available.

7.1.4 Schema Mapping

Once a GCS (or mediated schema) is defined, it is necessary to identify how the data from each of the local databases (source) can be mapped to GCS (target) while preserving semantic consistency (as defined by both the source and the target). Although schema matching has identified the correspondences between the LCSs and the GCS, it may not have identified explicitly how to obtain the global database from the local ones. This is what schema mapping is about.

In the case of data warehouses, schema mappings are used to explicitly extract data from the sources, and translate them to the data warehouse schema for populating it. In the case of data integration systems, these mappings are used in query processing phase by both the query processor and the wrappers (see Sect. 7.2).

There are two issues related to schema mapping that we will study: *mapping creation* and *mapping maintenance*. Mapping creation is the process of creating explicit queries that map data from a local database to the global one. Mapping maintenance is the detection and correction of mapping inconsistencies resulting from schema evolution. Source schemas may undergo structural or semantic changes that invalidate mappings. Mapping maintenance is concerned with the detection of broken mappings and the (automatic) rewriting of mappings such that semantic consistency with the new schema and semantic equivalence with the current mapping are achieved.

7.1.4.1 Mapping Creation

Mapping creation starts with a source LCS, the target GCS, and a set of schema matches M and produces a set of queries that, when executed, will create GCS data instances from the source data. In data warehouses, these queries are actually executed to create the data warehouse (global database), while in data integration systems, they are used in the reverse direction during query processing (Sect. 7.2).

Let us make this more concrete by referring to the canonical relational representation that we have adopted. The source LCS under consideration consists of a set of relations $Source = \{\mathtt{O}_1, \ldots, \mathtt{O}_m\}$, the GCS consists of a set of global (or target) relations $Target = \{\mathtt{T}_1, \ldots, \mathtt{T}_n\}$, and M consists of a set of schema match rules as defined in Sect. 7.1.2. We are looking for a way to generate, for each $\mathtt{T}_k$, a query

Q_k that is defined on a (possibly proper) subset of the relations in *Source* such that, when executed, it will generate data for $\mathtt{T}_k$ from the source relations.

This can be accomplished iteratively by considering each $\mathtt{T}_k$ in turn. It starts with $M_k \subseteq M$ (M_k is the set of rules that only apply to the attributes of $\mathtt{T}_k$) and divides it into subsets $\{M_k^1, \ldots, M_k^s\}$ such that each M_k^j specifies one possible way that values of $\mathtt{T}_k$ can be computed. Each M_k^j can be mapped to a query q_k^j that, when executed, would generate *some* of $\mathtt{T}_k$'s data. The union of all of these queries gives $Q_k (= \cup_j q_k^j)$ that we are looking for.

The algorithm proceeds in four steps that we discuss below. It does not consider the similarity values in the rules. It can be argued that the similarity values would be used in the final stages of the matching process to finalize correspondences, so that their use during mapping is unnecessary. Furthermore, by the time this phase of the integration process is reached, the concern is how to map source relation (LCS) data to target relation (GCS) data. Consequently, correspondences are not symmetric equivalences ($\approx$), but mappings ($\mapsto$): attribute(s) from (possibly multiple) source relations are mapped to an attribute of a target relation (i.e., $(\mathtt{O}_i.attribute_k, \mathtt{O}_j.attribute_l) \mapsto \mathtt{T}_w.attribute_z)$).

Example 7.7 To demonstrate the algorithm, we will use a different example database than what we have been working with, because it does not incorporate all the complexities that we wish to demonstrate. Instead, we will use the following abstract example.

Source relations (LCS):

$\mathtt{O}_1(\mathtt{A}_1, \mathtt{A}_2)$
$\mathtt{O}_2(\mathtt{B}_1, \mathtt{B}_2, \mathtt{B}_3)$
$\mathtt{O}_3(\mathtt{C}_1, \mathtt{C}_2, \mathtt{C}_3)$
$\mathtt{O}_4(\mathtt{D}_1, \mathtt{D}_2)$

Target relation (GCS):

$\mathtt{T}(\mathtt{W}_1, \mathtt{W}_2, \mathtt{W}_3, \mathtt{W}_4)$

We consider only one relation in GCS since the algorithm iterates over target relations one-at-a-time; this is sufficient to demonstrate the operation of the algorithm.

The foreign key relationships between the attributes are as follows:

Foreign key	Refers to
$\mathtt{A}_1$	$\mathtt{B}_1$
$\mathtt{A}_2$	$\mathtt{B}_1$
$\mathtt{C}_1$	$\mathtt{B}_1$

Assume that the following matches have been discovered for attributes of relation T (these make up $M_{\mathtt{T}}$). In the subsequent examples, we will not be concerned with the predicates, so they are not explicitly specified.

$$r_1 = \langle \mathtt{A}_1 \mapsto \mathtt{W}_1, p \rangle$$
$$r_2 = \langle \mathtt{A}_2 \mapsto \mathtt{W}_2, p \rangle$$
$$r_3 = \langle \mathtt{B}_2 \mapsto \mathtt{W}_4, p \rangle$$
$$r_4 = \langle \mathtt{B}_3 \mapsto \mathtt{W}_3, p \rangle$$
$$r_5 = \langle \mathtt{C}_1 \mapsto \mathtt{W}_1, p \rangle$$
$$r_6 = \langle \mathtt{C}_2 \mapsto \mathtt{W}_2, p \rangle$$
$$r_7 = \langle \mathtt{D}_1 \mapsto \mathtt{W}_4, p \rangle$$

♦

In the first step, M_k (corresponding to $\mathtt{T}_k$) is partitioned into its subsets $\{M_k^1, \ldots, M_k^n\}$ such that each M_k^j contains at most one match for each attribute of $\mathtt{T}_k$. These are called *potential candidate sets*, some of which may be *complete* in that they include a match for every attribute of $\mathtt{T}_k$, but others may not be. The reasons for considering incomplete sets are twofold. First, it may be the case that no match is found for one or more attributes of the target relation (i.e., none of the match sets is complete). Second, for large and complex database schemas, it may make sense to build the mapping iteratively so that the designer specifies the mappings incrementally.

Example 7.8 $M_{\mathtt{T}}$ is partitioned into fifty-three subsets (i.e., potential candidate sets). The first eight of these are complete, while the rest are not. We show some of these below. To make it easier to read, the complete rules are listed in the order of the target attributes to which they map (e.g., the third rule in $M_{\mathtt{T}}^1$ is r_4, because this rule maps to attribute $\mathtt{W}_3$):

$$M_{\mathtt{T}}^1 = \{r_1, r_2, r_4, r_3\} \quad M_{\mathtt{T}}^2 = \{r_1, r_2, r_4, r_7\}$$
$$M_{\mathtt{T}}^3 = \{r_1, r_6, r_4, r_3\} \quad M_{\mathtt{T}}^4 = \{r_1, r_6, r_4, r_7\}$$
$$M_{\mathtt{T}}^5 = \{r_5, r_2, r_4, r_3\} \quad M_{\mathtt{T}}^6 = \{r_5, r_2, r_4, r_7\}$$
$$M_{\mathtt{T}}^7 = \{r_5, r_6, r_4, r_3\} \quad M_{\mathtt{T}}^8 = \{r_5, r_6, r_4, r_7\}$$
$$M_{\mathtt{T}}^9 = \{r_1, r_2, r_3\} \qquad M_{\mathtt{T}}^{10} = \{r_1, r_2, r_4\}$$
$$M_{\mathtt{T}}^{11} = \{r_1, r_3, r_4\} \qquad M_{\mathtt{T}}^{12} = \{r_2, r_3, r_4\}$$
$$M_{\mathtt{T}}^{13} = \{r_1, r_3, r_6\} \qquad M_{\mathtt{T}}^{14} = \{r_3, r_4, r_6\}$$
$$\ldots \qquad \ldots$$
$$M_{\mathtt{T}}^{47} = \{r_1\} \qquad M_{\mathtt{T}}^{48} = \{r_2\}$$
$$M_{\mathtt{T}}^{49} = \{r_3\} \qquad M_{\mathtt{T}}^{50} = \{r_4\}$$
$$M_{\mathtt{T}}^{51} = \{r_5\} \qquad M_{\mathtt{T}}^{52} = \{r_6\}$$
$$M_{\mathtt{T}}^{53} = \{r_7\}$$

♦

In the second step, the algorithm analyzes each potential candidate set M_k^j to see if a "good" query can be produced for it. If all the matches in M_k^j map values from a single source relation to $\mathtt{T}_k$, then it is easy to generate a query corresponding to M_k^j. Of particular concern are matches that require access to multiple source relations. In this case the algorithm checks to see if there is a referential connection between these relations through foreign keys (i.e., whether there is a join path through the source relations). If there is not, then the potential candidate set is eliminated from further consideration. In case there are multiple join paths through foreign key relationships, the algorithm looks for those paths that will produce the most number of tuples (i.e., the estimated difference in size of the outer and inner joins is the smallest). If there are multiple such paths, then the database designer needs to be involved in selecting one (tools such as Clio, OntoBuilder, and others facilitate this process and provide mechanisms for designers to view and specify correspondences). The result of this step is a set $\overline{M_k} \subseteq M_k$ of *candidate sets*.

Example 7.9 In this example, there is no M_k^j where the values of all of T's attributes are mapped from a single source relation. Among those that involve multiple source relations, rules that involve $\mathtt{O}_1$, $\mathtt{O}_2$, and $\mathtt{O}_3$ can be mapped to "good" queries since there are foreign key relationships between them. However, the rules that involve $\mathtt{O}_4$ (i.e., those that include rule r_7) cannot be mapped to a "good" query since there is no join path from $\mathtt{O}_4$ to the other relations (i.e., any query would involve a cross product, which is expensive). Thus, these rules are eliminated from the potential candidate set. Considering only the complete sets, M_k^2, M_k^4, M_k^6, and M_k^8 are pruned from the set. In the end, the candidate set ($\overline{M_k}$) contains thirty-five rules (the readers are encouraged to verify this to better understand the algorithm). ♦

In the third step, the algorithm looks for a cover of the candidate sets $\overline{M_k}$. The cover $C_k \subseteq \overline{M_k}$ is a set of candidate sets such that each match in $\overline{M_k}$ appears in C_k at least once. The point of determining a cover is that it accounts for all of the matches and is, therefore, sufficient to generate the target relation $\mathtt{T}_k$. If there are multiple covers (a match can participate in multiple covers), then they are ranked in increasing number of the candidate sets in the cover. The fewer the number of candidate sets in the cover, the fewer are the number of queries that will be generated in the next step; this improves the efficiency of the mappings that are generated. If there are multiple covers with the same ranking, then they are further ranked in decreasing order of the total number of unique target attributes that are used in the candidate sets constituting the cover. The point of this ranking is that covers with higher number of attributes generate fewer null values in the result. At this stage, the designer may need to be consulted to choose from among the ranked covers.

Example 7.10 First note that we have six rules that define matches in $\overline{M_k}$ that we need to consider, since M_k^j that include rule r_7 have been eliminated. There are a large number of possible covers; let us start with those that involve M_k^1 to demonstrate the algorithm:

$$C_{\mathrm{T}}^1 = \{\{\underbrace{r_1, r_2, r_4, r_3}_{M_{\mathrm{T}}^1}\}, \{\underbrace{r_1, r_6, r_4, r_3}_{M_{\mathrm{T}}^3}\}, \underbrace{\{r_2\}}_{M_{\mathrm{T}}^{48}}\}$$

$$C_{\mathrm{T}}^2 = \{\{\underbrace{r_1, r_2, r_4, r_3}_{M_{\mathrm{T}}^1}\}, \{\underbrace{r_5, r_2, r_4, r_3}_{M_{\mathrm{T}}^5}\}, \underbrace{\{r_6\}}_{M_{\mathrm{T}}^{50}}\}$$

$$C_{\mathrm{T}}^3 = \{\{\underbrace{r_1, r_2, r_4, r_3}_{M_{\mathrm{T}}^1}\}, \{\underbrace{r_5, r_6, r_4, r_3}_{M_{\mathrm{T}}^7}\}\}$$

$$C_{\mathrm{T}}^4 = \{\{\underbrace{r_1, r_2, r_4, r_3}_{M_{\mathrm{T}}^1}\}, \{\underbrace{r_5, r_6, r_4}_{M_{\mathrm{T}}^{12}}\}\}$$

$$C_{\mathrm{T}}^5 = \{\{\underbrace{r_1, r_2, r_4, r_3}_{M_{\mathrm{T}}^1}\}, \{\underbrace{r_5, r_6, r_3}_{M_{\mathrm{T}}^{19}}\}\}$$

$$C_{\mathrm{T}}^6 = \{\{\underbrace{r_1, r_2, r_4, r_3}_{M_{\mathrm{T}}^1}\}, \{\underbrace{r_5, r_6}_{M_{\mathrm{T}}^{32}}\}\}$$

At this point we observe that the covers consist of either two or three candidate sets. Since the algorithm prefers those with fewer candidate sets, we only need to focus on those involving two sets. Furthermore, among these covers, we note that the number of target attributes in the candidate sets differ. Since the algorithm prefers covers with the largest number of target attributes in each candidate set, C_{T}^3 is the preferred cover.

Note that due to the two heuristics employed by the algorithm, the only covers we need to consider are those that involve M_{T}^1, M_{T}^3, M_{T}^5, and M_{T}^7. Similar covers can be defined involving M_{T}^3, M_{T}^5, and M_{T}^7; we leave that as an exercise. In the remainder, we will assume that the designer has chosen to use C_{T}^3 as the preferred cover. ♦

The final step of the algorithm builds a query q_k^j for each of the candidate sets in the cover selected in the previous step. The union of all of these queries (`UNION ALL`) results in the final mapping for relation $\mathtt{T}_k$ in the GCS.

Query q_k^j is built as follows:

- **`SELECT`** clause includes all correspondences (c) in each of the rules (r_k^i) in M_k^j.
- **`FROM`** clause includes all source relations mentioned in r_k^i and in the join paths determined in Step 2 of the algorithm.
- **`WHERE`** clause includes conjunct of all predicates (p) in r_k^i and all join predicates determined in Step 2 of the algorithm.
- If r_k^i contains an aggregate function either in c or in p
  - **`GROUP BY`** is used over attributes (or functions on attributes) in the **`SELECT`** clause that are not within the aggregate;

- If aggregate is in the correspondence c, it is added to **SELECT**, else (i.e., aggregate is in the predicate p) a **HAVING** clause is created with the aggregate.

Example 7.11 Since in Example 7.10 we have decided to use cover C_{T}^3 for the final mapping, we need to generate two queries: q_{T}^1 and q_{T}^7 corresponding to M_{T}^1 and M_{T}^7, respectively. For ease of presentation, we list the rules here again:

$$r_1 = \langle \mathrm{A}_1 \mapsto \mathrm{W}_1,\ p \rangle$$
$$r_2 = \langle \mathrm{A}_2 \mapsto \mathrm{W}_2,\ p \rangle$$
$$r_3 = \langle \mathrm{B}_2 \mapsto \mathrm{W}_4,\ p \rangle$$
$$r_4 = \langle \mathrm{B}_3 \mapsto \mathrm{W}_3,\ p \rangle$$
$$r_5 = \langle \mathrm{C}_1 \mapsto \mathrm{W}_1,\ p \rangle$$
$$r_6 = \langle \mathrm{C}_2 \mapsto \mathrm{W}_2,\ p \rangle$$

The two queries are as follows:

q_k^1 :
SELECT $\mathrm{A}_1, \mathrm{A}_2, \mathrm{B}_2, \mathrm{B}_3$
FROM $\mathrm{O}_1, \mathrm{O}_2$
WHERE p_1 **AND** $\mathrm{O}_1.\mathrm{A}_2 = \mathrm{O}_2.\mathrm{B}_1$

q_k^7 :
SELECT $\mathrm{B}_2, \mathrm{B}_3, \mathrm{C}_1, \mathrm{C}_2$
FROM $\mathrm{O}_2, \mathrm{O}_3$
WHERE p_3 **AND** p_4 **AND** p_5 **AND** p_6
AND $\mathrm{O}_3.c_1 = \mathrm{O}_2.\mathrm{B}_1$

Thus, the final query Q_k for target relation T becomes q_k^1 **UNION ALL** q_k^7. ♦

The output of this algorithm after it is iteratively applied to each target relation T_k is a set of queries $Q = \{Q_k\}$ that, when executed, produce data for the GCS relations. Thus, the algorithm produces GAV mappings between relational schemas—recall that GAV defines a GCS as a view over the LCSs and that is exactly what the set of mapping queries do. The algorithm takes into account the semantics of the source schema since it considers foreign key relationships in determining which queries to generate. However, it does not consider the semantics of the target, so that the tuples that are generated by the execution of the mapping queries are not guaranteed to satisfy target semantics. This is not a major issue in the case when the GCS is integrated from the LCSs; however, if the GCS is defined independent of the LCSs, then this is problematic.

It is possible to extend the algorithm to deal with target semantics as well as source semantics. This requires that interschema tuple-generating dependencies be considered. In other words, it is necessary to produce GLAV mappings. A GLAV

mapping, by definition, is not simply a query over the source relations; it is a relationship between a query over the source (i.e., LCS) relations and a query over the target (i.e., GCS) relations. Let us be more precise. Consider a schema match v that specifies a correspondence between attribute A of a source LCS relation R and attribute B of a target GCS relation T (in the notation we used in this section we have $v = \langle \mathtt{R.A} \approx \mathtt{T.B}, p, s \rangle$). Then the source query specifies how to retrieve R.A and the target query specifies how to obtain T.B. The GLAV mapping, then, is a relationship between these two queries.

This can be accomplished by starting with a source schema, a target schema, and M, and "discovering" mappings that satisfy both the source and the target schema semantics. This algorithm is also more powerful than the one we discussed in this section in that it can handle nested structures that are common in XML, object databases, and nested relational systems.

The first step in discovering all of the mappings based on schema match correspondences is *semantic translation*, which seeks to interpret schema matches in M in a way that is consistent with the semantics of both the source and target schemas as captured by the schema structure and the referential (foreign key) constraints. The result is a set of *logical mappings* each of which captures the design choices (semantics) made in both source and target schemas. Each logical mapping corresponds to one target schema relation. The second step is *data translation* that implements each logical mapping as a rule that can be translated into a query that would create an instance of the target element when executed.

Semantic translation takes as inputs the source $Source$ and target schemas $Target$, and M and performs the following two steps:

- It examines intraschema semantics within the $Source$ and $Target$ separately and produces for each a set of *logical relations* that are semantically consistent.
- It then interprets interschema correspondences M in the context of logical relations generated in Step 1 and produces a set of queries into Q that are semantically consistent with $Target$.

7.1.4.2 Mapping Maintenance

In dynamic environments where schemas evolve over time, schema mappings can be made invalid as the result of structural or constraint changes of the schemas. Thus, the detection of invalid/inconsistent schema mappings and the adaptation of such schema mappings to new schema structures/constraints are important.

In general, automatic detection of invalid/inconsistent schema mappings is desirable as the complexity of the schemas and the number of schema mappings used in database applications increase. Likewise, (semi-)automatic adaptation of mappings to schema changes is also a goal. It should be noted that automatic adaptation of schema mappings is not the same as automatic schema matching. Schema adaptation aims to resolve semantic correspondences using known changes in intraschema semantics, semantics in existing mappings, and detected semantic

inconsistencies (resulting from schema changes). Schema matching must take a much more "from scratch" approach at generating schema mappings and does not have the ability (or luxury) of incorporating such contextual knowledge.

Detecting Invalid Mappings

In general, detection of invalid mappings resulting from schema change can either happen proactively or reactively. In proactive detection environments, schema mappings are tested for inconsistencies as soon as schema changes are made by a user. The assumption (or requirement) is that the mapping maintenance system is completely aware of any and all schema changes, as soon as they are made. The ToMAS system, for example, expects users to make schema changes through its own schema editors, making the system immediately aware of any schema changes. Once schema changes have been detected, invalid mappings can be detected by doing a semantic translation of the existing mappings using the logical relations of the updated schema.

In reactive detection environments, the mapping maintenance system is unaware of when and what schema changes are made. To detect invalid schema mappings in this setting, mappings are tested at regular intervals by performing queries against the data sources and translating the resulting data using the existing mappings. Invalid mappings are then determined based on the results of these mapping tests.

An alternative method that has been proposed is to use machine learning techniques to detect invalid mappings (as in the Maveric system). What has been proposed is to build an ensemble of trained *sensors* (similar to multiple learners in schema matching) to detect invalid mappings. Examples of such sensors include value sensors for monitoring distribution characteristics of target instance values, trend sensors for monitoring the average rate of data modification, and layout and constraint sensors that monitor translated data against expected target schema syntax and semantics. A weighted combination of the findings of the individual sensors is then calculated where the weights are also learned. If the combined result indicates changes and follow-up tests suggest that this may indeed be the case, an alert is generated.

Adapting Invalid Mappings

Once invalid schema mappings are detected, they must be adapted to schema changes and made valid once again. Various high-level mapping adaptation approaches have been proposed. These can be broadly described as *fixed rule approaches* that define a remapping rule for every type of expected schema change, *map bridging approaches* that compare original schema S and the updated schema S', and generate new mapping from S to S' in addition to existing mappings, and *semantic rewriting approaches*, which exploit semantic information encoded in existing mappings, schemas, and semantic changes made to schemas to propose map

rewritings that produce semantically consistent target data. In most cases, multiple such rewritings are possible, requiring a ranking of the candidates for presentation to users who make the final decision (based on scenario- or business-level semantics not encoded in schemas or mappings).

Arguably, a complete remapping of schemas (i.e., from scratch, using schema matching techniques) is another alternative to mapping adaption. However, in most cases, map rewriting is cheaper than map regeneration as rewriting can exploit knowledge encoded in existing mappings to avoid computation of mappings that would be rejected by the user anyway (and to avoid redundant mappings).

7.1.5 Data Cleaning

Errors in source databases can always occur, requiring cleaning in order to correctly answer user queries. Data cleaning is a problem that arises in both data warehouses and data integration systems, but in different contexts. In data warehouses where data is actually extracted from local operational databases and materialized as a global database, cleaning is performed as the global database is created. In the case of data integration systems, data cleaning is a process that needs to be performed during query processing when data is returned from the source databases.

The errors that are subject to data cleaning can generally be broken down into either schema-level or instance-level concerns. Schema-level problems can arise in each individual LCS due to violations of explicit and implicit constraints. For example, values of attributes may be outside the range of their domains (e.g., 14th month or negative salary value), attribute values may violate implicit dependencies (e.g., the age attribute value may not correspond to the value that is computed as the difference between the current date and the birth date), uniqueness of attribute values may not hold, and referential integrity constraints may be violated. Furthermore, in the environment that we are considering in this chapter, the schema-level heterogeneities (both structural and semantic) among the LCSs that we discussed earlier can all be considered problems that need to be resolved. At the schema level, it is clear that the problems need to be identified at the schema match stage and fixed during schema integration.

Instance level errors are those that exist at the data level. For example, the values of some attributes may be missing although they were required, there could be misspellings and word transpositions (e.g., “M.D. Mary Smith” versus “Mary Smith, M.D.”) or differences in abbreviations (e.g., “J. Doe” in one source database, while “J.N. Doe” in another), embedded values (e.g., an aggregate address attribute that includes street name, value, province name, and postal code), values that were erroneously placed in other fields, duplicate values, and contradicting values (the salary value appearing as one value in one database and another value in another database). For instance-level cleaning, the issue is clearly one of generating the mappings such that the data is cleaned through the execution of the mapping functions (queries).

The popular approach to data cleaning has been to define a number of operators that operate either on schemas or on individual data. The operators can be composed into a data cleaning plan. Example schema operators add or drop columns from table, restructure a table by combining columns or splitting a column into two, or define more complicated schema transformation through a generic "map" operator that takes a single relation and produces one or more relations. Example data level operators include those that apply a function to every value of one attribute, merging values of two attributes into the value of a single attribute and its converse split operator, a matching operator that computes an approximate join between tuples of two relations, clustering operator that groups tuples of a relation into clusters, and a tuple merge operator that partitions the tuples of a relation into groups and collapses the tuples in each group into a single tuple through some aggregation over them, as well as basic operators to find duplicates and eliminate them. Many of the data level operators compare individual tuples of two relations (from the same or different schemas) and decide whether or not they represent the same fact. This is similar to what is done in schema matching, except that it is done at the individual data level and what is considered are not individual attribute values, but entire tuples. However, the same techniques we studied under schema matching (e.g., use of edit distance or soundex value) can be used in this context. There have been proposals for special techniques for handling this efficiently within the context of data cleaning such as fuzzy matching that computes a similarity function to determine whether the two tuples are identical or reasonably similar.

Given the large amount of data that needs to be handled, data level cleaning is expensive and efficiency is a significant issue. The physical implementation of each of the operators we discussed above is a considerable concern. Although cleaning can be done off-line as a batch process in the case of data warehouses, for data integration systems, cleaning may need to be done online as data is retrieved from the sources. The performance of data cleaning is, of course, more critical in the latter case.

7.2 Multidatabase Query Processing

We now turn our attention to querying and accessing an integrated database obtained through the techniques discussed in the previous section—this is known as the multidatabase querying problem. As previously noted, many of the distributed query processing and optimization techniques that we discussed in Chap. 4 carry over to multidatabase systems, but there are important differences. Recall from that chapter that we characterized distributed query processing in four steps: query decomposition, data localization, global optimization, and local optimization. The nature of multidatabase systems requires slightly different steps and different techniques. The component DBMSs may be autonomous and have different database languages and query processing capabilities. Thus, an MDBS layer (see Fig. 1.12) is necessary to communicate with component DBMSs in an effective way, and this requires

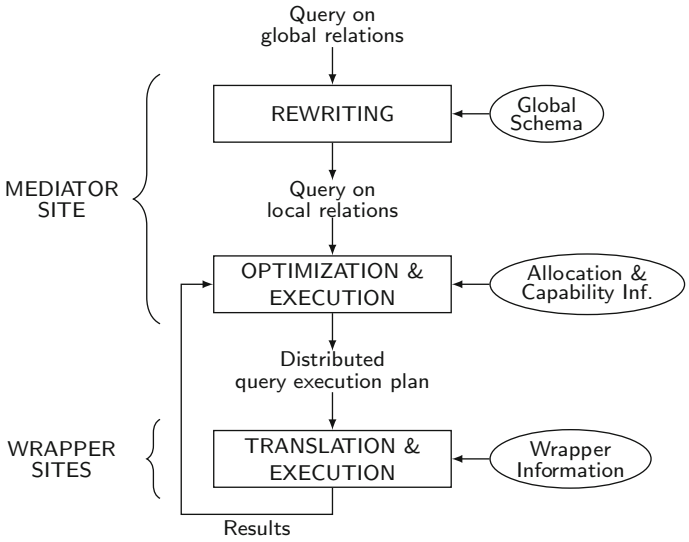

Fig. 7.13 Generic layering scheme for multidatabase query processing

additional query processing steps (Fig. 7.13). Furthermore, there may be many component DBMSs, each of which may exhibit different behavior, thereby posing new requirements for more adaptive query processing techniques.

7.2.1 Issues in Multidatabase Query Processing

Query processing in a multidatabase system is more complex than in a distributed DBMS for the following reasons:

1. The computing capabilities of the component DBMSs may be different, which prevents uniform treatment of queries across multiple DBMSs. For example, some DBMSs may be able to support complex SQL queries with join and aggregation, while some others cannot. Thus the multidatabase query processor should consider the various DBMS capabilities. The capabilities of each component is recorded in the directory along with data allocation information.
2. Similarly, the cost of processing queries may be different on different DBMSs, and the local optimization capability of each DBMS may be quite different. This increases the complexity of the cost functions that need to be evaluated.
3. The data models and languages of the component DBMSs may be quite different, for instance, relational, object-oriented, semi-structured, etc. This creates difficulties in translating multidatabase queries to component DBMS and in integrating heterogeneous results.

4. Since a multidatabase system enables access to very different DBMSs that may have different performance and behavior, distributed query processing techniques need to adapt to these variations.

The autonomy of the component DBMSs poses problems. DBMS autonomy can be defined along three main dimensions: communication, design, and execution. Communication autonomy means that a component DBMS communicates with others at its own discretion, and, in particular, it may terminate its services at any time. This requires query processing techniques that are tolerant to system unavailability. The question is how the system answers queries when a component system is either unavailable from the beginning or shuts down in the middle of query execution. Design autonomy may restrict the availability and accuracy of cost information that is needed for query optimization. The difficulty of determining local cost functions is an important issue. The execution autonomy of multidatabase systems makes it difficult to apply some of the query optimization strategies we discussed in previous chapters. For example, semijoin-based optimization of distributed joins may be difficult if the source and target relations reside in different component DBMSs, since, in this case, the semijoin execution of a join translates into three queries: one to retrieve the join attribute values of the target relation and to ship it to the source relation's DBMS, the second to perform the join at the source relation, and the third to perform the join at the target relation's DBMS. The problem arises because communication with component DBMSs occurs at a high level of the DBMS API.

In addition to these difficulties, the architecture of a distributed multidatabase system poses certain challenges. The architecture depicted in Fig. 1.12 points to an additional complexity. In distributed DBMSs, query processors have to deal only with data distribution across multiple sites. In a distributed multidatabase environment, on the other hand, data is distributed not only across sites but also across multiple databases, each managed by an autonomous DBMS. Thus, while there are two parties that cooperate in the processing of queries in a distributed DBMS (the control site and local sites), the number of parties increases to three in the case of a distributed MDBS: the MDBS layer at the control site (i.e., the mediator) receives the global query, the MDBS layers at the sites (i.e., the wrappers) participate in processing the query, and the component DBMSs ultimately optimize and execute the query.

7.2.2 Multidatabase Query Processing Architecture

Most of the work on multidatabase query processing has been done in the context of the mediator/wrapper architecture (see Fig. 1.13). In this architecture, each component database has an associated wrapper that exports information about the source schema, data, and query processing capabilities. A mediator centralizes the information provided by the wrappers in a unified view of all available data (stored

in a global data dictionary) and performs query processing using the wrappers to access the component DBMSs. The data model used by the mediator can be relational, object-oriented, or even semi-structured. In this chapter, for consistency with the previous chapters on distributed query processing, we continue to use the relational model, which is quite sufficient to explain the multidatabase query processing techniques.

The mediator/wrapper architecture has several advantages. First, the specialized components of the architecture allow the various concerns of different kinds of users to be handled separately. Second, mediators typically specialize in a related set of component databases with "similar" data, and thus export schemas and semantics related to a particular domain. The specialization of the components leads to a flexible and extensible distributed system. In particular, it allows seamless integration of different data stored in very different components, ranging from full-fledged relational DBMSs to simple files.

Assuming the mediator/wrapper architecture, we can now discuss the various layers involved in query processing in distributed multidatabase systems as shown in Fig. 7.13. As before, we assume the input is a query on global relations expressed in relational calculus. This query is posed on global (distributed) relations, meaning that data distribution and heterogeneity are hidden. Three main layers are involved in multidatabase query processing. This layering is similar to that of query processing in homogeneous distributed DBMSs (see Fig. 4.2). However, since there is no fragmentation, there is no need for the data localization layer.

The first two layers map the input query into an optimized distributed query execution plan (QEP). They perform the functions of query rewriting, query optimization, and some query execution. The first two layers are performed by the mediator and use metainformation stored in the global directory (global schema, allocation, and capability information). Query rewriting transforms the input query into a query on local relations, using the global schema. Recall that there are two main approaches for database integration: global-as-view (GAV) and local-as-view (LAV). Thus, the global schema provides the view definitions (i.e., mappings between the global relations and the local relations stored in the component databases) and the query is rewritten using the views.

Rewriting can be done at the relational calculus or algebra levels. In this chapter, we will use a generalized form of relational calculus called Datalog that is well-suited for such rewriting. Thus, there is an additional step of calculus to algebra translation that is similar to the decomposition step in homogeneous distributed DBMSs.

The second layer performs query optimization and (some) execution by considering the allocation of the local relations and the different query processing capabilities of the component DBMSs exported by the wrappers. The allocation and capability information used by this layer may also contain heterogeneous cost information. The distributed QEP produced by this layer groups within subqueries the operations that can be performed by the component DBMSs and wrappers. Similar to distributed DBMSs, query optimization can be static or dynamic. However, the lack of homogeneity in multidatabase systems (e.g., some component

DBMSs may have unexpectedly long delays in answering) makes dynamic query optimization more critical. In the case of dynamic optimization, there may be subsequent calls to this layer after execution by the subsequent layer as illustrated by the arrow showing results coming from the translation and execution layer. Finally, this layer integrates the results coming from the different wrappers to provide a unified answer to the user's query. This requires the capability of executing some operations on data coming from the wrappers. Since the wrappers may provide very limited execution capabilities, e.g., in the case of very simple component DBMSs, the mediator must provide the full execution capabilities to support the mediator interface.

The third layer performs *query translation and execution* using the wrappers. Then it returns the results to the mediator that can perform result integration from different wrappers and subsequent execution. Each wrapper maintains a *wrapper schema* that includes the local export schema and mapping information to facilitate the translation of the input subquery (a subset of the QEP) expressed in a common language into the language of the component DBMS. After the subquery is translated, it is executed by the component DBMS and the local result is translated back to the common format.

The wrapper information describes how mappings from/to participating local schemas and global schema can be performed. It enables conversions between components of the database in different ways. For example, if the global schema represents temperatures in Fahrenheit degrees, but a participating database uses Celsius degrees, the wrapper information must contain a conversion formula to provide the proper presentation to the global user and the local databases. If the conversion is across types and simple formulas cannot perform the translation, complete mapping tables could be used in the wrapper information stored in the directory.

7.2.3 Query Rewriting Using Views

Query rewriting reformulates the input query expressed on global relations into a query on local relations. It uses the global schema, which describes in terms of views the correspondences between the global relations and the local relations. Thus, the query must be rewritten using views. The techniques for query rewriting differ in major ways depending on the database integration approach that is used, i.e., GAV or LAV. In particular, the techniques for LAV (and its extension GLAV) are much more involved. Most of the work on query rewriting using views has been done using Datalog, which is a logic-based database language. Datalog is more concise than relational calculus and thus more convenient for describing complex query rewriting algorithms. In this section, we first introduce Datalog terminology. Then, we describe the main techniques and algorithms for query rewriting in the GAV and LAV approaches.

7.2.3.1 Datalog Terminology

Datalog can be viewed as an in-line version of domain relational calculus. Let us first define *conjunctive queries*, i.e., select-project-join queries, which are the basis for more complex queries. A conjunctive query in Datalog is expressed as a rule of the form:

$$Q(t) : -\mathtt{R}_1(t_1), \ldots, \mathtt{R}_n(t_n)$$

The atom $Q(t)$ is the *head* of the query and denotes the result relation. The atoms $\mathtt{R}_1(t_1), \ldots, \mathtt{R}_n(t_n)$ are the *subgoals* in the body of the query and denote database relations. Q and $\mathtt{R}_1, \ldots, \mathtt{R}_n$ are predicate names and correspond to relation names. $t, t_1, \ldots, t_n$ refer to the relation tuples and contain variables or constants. The variables are similar to domain variables in domain relational calculus. Thus, the use of the same variable name in multiple predicates expresses equijoin predicates. Constants correspond to equality predicates. More complex comparison predicates (e.g., using comparators such as $\neq$, $\leq$, and $<$) must be expressed as other subgoals. We consider queries that are *safe*, i.e., those where each variable in the head also appears in the body. Disjunctive queries can also be expressed in Datalog using unions, by having several conjunctive queries with the same head predicate.

Example 7.12 Let us consider GCS relations EMP and WORKS defined in Fig. 7.9. Consider the following SQL query:

```
SELECT E#, TITLE, P#
FROM   EMP NATURAL JOIN WORKS
WHERE  TITLE = "Programmer" OR DUR = 24
```

The corresponding query in Datalog can be expressed as:

$$\begin{aligned} Q(\mathtt{E\#}, \mathtt{TITLE}, \mathtt{P\#}) : -&\mathtt{EMP}(\mathtt{E\#}, \mathtt{ENAME}, \text{“Programmer”}, \mathtt{CITY}), \\ &\mathtt{WORKS}(\mathtt{E\#}, \mathtt{P\#}, \mathtt{RESP}, \mathtt{DUR}) \\ Q(\mathtt{E\#}, \mathtt{TITLE}, \mathtt{P\#}) : -&\mathtt{EMP}(\mathtt{E\#}, \mathtt{ENAME}, \mathtt{TITLE}, \mathtt{CITY}), \\ &\mathtt{WORKS}(\mathtt{E\#}, \mathtt{P\#}, \mathtt{RESP}, 24) \end{aligned}$$

♦

7.2.3.2 Rewriting in GAV

In the GAV approach, the global schema is expressed in terms of the data sources and each global relation is defined as a view over the local relations. This is similar to the global schema definition in tightly integrated distributed DBMS. In particular,

the local relations (i.e., relations in a component DBMS) can correspond to fragments. However, since the local databases preexist and are autonomous, it may happen that tuples in a global relation do not exist in local relations or that a tuple in a global relation appears in different local relations. Thus, the properties of completeness and disjointness of fragmentation cannot be guaranteed. The lack of completeness may yield incomplete answers to queries. The lack of disjointness may yield duplicate results that may still be useful information and may not need to be eliminated. Similar to queries, view definitions can use Datalog notation.

Example 7.13 Let us consider the global relations EMP and WORKS in Fig. 7.9, with a slight modification: the default responsibility of an employee in a project corresponds to its title, so that attribute TITLE is present in relation WORKS but absent in relation EMP. Let us consider the local relations EMP1 and EMP2 each with attributes E#, ENAME, TITLE, and CITY, and local relation WORKS1 with attributes E#, P#, and DUR. The global relations EMP and WORKS can be simply defined with the following Datalog rules:

$$\texttt{EMP(E\#, ENAME, CITY)} : -\texttt{EMP1(E\#, ENAME, TITLE, CITY)} \quad (d_1)$$

$$\texttt{EMP(E\#, ENAME, CITY)} : -\texttt{EMP2(E\#, ENAME, TITLE, CITY)} \quad (d_2)$$

$$\texttt{WORKS(E\#, P\#, TITLE, DUR)} : -\texttt{EMP1(E\#, ENAME, TITLE, CITY)}, \texttt{WORKS1(E\#, P\#, DUR)} \quad (d_3)$$

$$\texttt{WORKS(E\#, P\#, TITLE, DUR)} : -\texttt{EMP2(E\#, ENAME, TITLE, CITY))}, \texttt{WORKS1(E\#, P\#, DUR)} \quad (d_4)$$

♦

Rewriting a query expressed on the global schema into an equivalent query on the local relations is relatively simple and similar to data localization in tightly integrated distributed DBMS (see Sect. 4.2). The rewriting technique using views is called *unfolding*, and it replaces each global relation invoked in the query with its corresponding view. This is done by applying the view definition rules to the query and producing a union of conjunctive queries, one for each rule application. Since a global relation may be defined by several rules (see Example 7.13), unfolding can generate redundant queries that need to be eliminated.

Example 7.14 Let us consider the global schema in Example 7.13 and the following query q that asks for assignment information about the employees living in Paris:

$$Q(e, p) : -\texttt{EMP}(e, \texttt{ENAME}, \text{“Paris”}), \texttt{WORKS}(e, p, \texttt{TITLE}, \texttt{DUR}).$$

Unfolding q produces q' as follows:

$$Q'(e, p) : -\texttt{EMP1}(e, \texttt{ENAME}, \texttt{TITLE}, \text{“Paris”}), \texttt{WORKS1}(e, p, \texttt{DUR}). \quad (q_1)$$

$$Q'(e, p) : -\texttt{EMP2}(e, \texttt{ENAME}, \texttt{TITLE}, \text{“Paris”}), \texttt{WORKS1}(e, p, \texttt{DUR}). \quad (q_2)$$

Q' is the union of two conjunctive queries labeled as q_1 and q_2. q_1 is obtained by applying GAV rule d_3 or both rules d_1 and d_3. In the latter case, the query obtained is redundant with respect to that obtained with d_3 only. Similarly, q_2 is obtained by applying rule d_4 or both rules d_2 and d_4. ♦

Although the basic technique is simple, rewriting in GAV becomes difficult when local databases have limited access patterns. This is the case for databases accessed over the web where relations can be only accessed using certain binding patterns for their attributes. In this case, simply substituting the global relations with their views is not sufficient, and query rewriting requires the use of recursive Datalog queries.

7.2.3.3 Rewriting in LAV

In the LAV approach, the global schema is expressed independent of the local databases and each local relation is defined as a view over the global relations. This enables considerable flexibility for defining local relations.

Example 7.15 To facilitate comparison with GAV, we develop an example that is symmetric to Example 7.13 with EMP and WORKS defined in that example as global relations as. In the LAV approach, the local relations EMP1, EMP2, and WORKS1 can be defined with the following Datalog rules:

$$\texttt{EMP1}(\texttt{E\#}, \texttt{ENAME}, \texttt{TITLE}, \texttt{CITY}) : -\texttt{EMP}(\texttt{E\#}, \texttt{ENAME}, \texttt{CITY}), \texttt{WORKS}(\texttt{E\#}, \texttt{P\#}, \texttt{TITLE}, \texttt{DUR}) \quad (d_5)$$

$$\texttt{EMP2}(\texttt{E\#}, \texttt{ENAME}, \texttt{TITLE}, \texttt{CITY}) : -\texttt{EMP}(\texttt{E\#}, \texttt{ENAME}, \texttt{CITY}), \texttt{WORKS}(\texttt{E\#}, \texttt{P\#}, \texttt{TITLE}, \texttt{DUR}) \quad (d_6)$$

$$\texttt{WORKS1}(\texttt{E\#}, \texttt{P\#}, \texttt{DUR}) : -\texttt{WORKS}(\texttt{E\#}, \texttt{P\#}, \texttt{TITLE}, \texttt{DUR}) \quad (d_7)$$

♦

Rewriting a query expressed on the global schema into an equivalent query on the views describing the local relations is difficult for three reasons. First, unlike in the GAV approach, there is no direct correspondence between the terms used in the global schema, (e.g., EMP, ENAME) and those used in the views (e.g., EMP1, EMP2, ENAME). Finding the correspondences requires comparison with each view. Second, there may be many more views than global relations, thus making view comparison time consuming. Third, view definitions may contain complex predicates to reflect the specific contents of the local relations, e.g., view EMP3

containing only programmers. Thus, it is not always possible to find an equivalent rewriting of the query. In this case, the best that can be done is to find a *maximally contained* query, i.e., a query that produces the maximum subset of the answer. For instance, EMP3 could only return a subset of all employees, those who are programmers.

Rewriting queries using views has received much attention because of its relevance to both logical and physical data integration problems. In the context of physical integration (i.e., data warehousing), using materialized views may be much more efficient than accessing base relations. However, the problem of finding a rewriting using views is NP-complete in the number of views and the number of subgoals in the query. Thus, algorithms for rewriting a query using views essentially try to reduce the numbers of rewritings that need to be considered. Three main algorithms have been proposed for this purpose: the bucket algorithm, the inverse rule algorithm, and the MinCon algorithm. The bucket algorithm and the inverse rule algorithm have similar limitations that are addressed by the MinCon algorithm.

The bucket algorithm considers each predicate of the query independently to select only the views that are relevant to that predicate. Given a query Q, the algorithm proceeds in two steps. In the first step, it builds a bucket b for each subgoal q of Q that is not a comparison predicate and inserts in b the heads of the views that are relevant to answer q. To determine whether a view V should be in b, there must be a mapping that unifies q with one subgoal v in V.

For instance, consider query Q in Example 7.14 and the views in Example 7.15. The following mapping unifies the subgoal EMP(e, ENAME, "Paris") of Q with the subgoal EMP(E#, ENAME, CITY) in view EMP1:

$$e \rightarrow \text{E\#}, \text{“Paris”} \rightarrow \text{CITY}$$

In the second step, for each view V of the Cartesian product of the nonempty buckets (i.e., some subset of the buckets), the algorithm produces a conjunctive query and checks whether it is contained in Q. If it is, the conjunctive query is kept as it represents one way to answer part of Q from V. Thus, the rewritten query is a union of conjunctive queries.

Example 7.16 Let us consider query Q in Example 7.14 and the views in Example 7.15. In the first step, the bucket algorithm creates two buckets, one for each subgoal of Q. Let us denote by b_1 the bucket for the subgoal EMP(e, ENAME, "Paris") and by b_2 the bucket for the subgoal WORKS(e, p, TITLE, DUR). Since the algorithm inserts only the view heads in a bucket, there may be variables in a view head that are not in the unifying mapping. Such variables are simply primed. We obtain the following buckets:

$$\begin{aligned} b_1 = \{&\text{EMP1(E\#, ENAME, TITLE}', \text{CITY}), \\ &\text{EMP2(E\#, ENAME, TITLE}', \text{CITY})\} \\ b_2 = \{&\text{WORKS1(E\#, P\#, DUR}')\} \end{aligned}$$

In the second step, the algorithm combines the elements from the buckets, which produces a union of two conjunctive queries:

$$Q'(e, p) : -\texttt{EMP1}(e, \texttt{ENAME}, \texttt{TITLE}, \text{“Paris”}), \texttt{WORKS1}(e, p, \texttt{DUR}) \qquad (q_1)$$

$$Q'(e, p) : -\texttt{EMP2}(e, \texttt{ENAME}, \texttt{TITLE}, \text{“Paris”}), \texttt{WORKS1}(e, p, \texttt{DUR}) \qquad (q_2)$$

♦

The main advantage of the bucket algorithm is that, by considering the predicates in the query, it can significantly reduce the number of rewritings that need to be considered. However, considering the predicates in the query in isolation may yield the addition of a view in a bucket that is irrelevant when considering the join with other views. Furthermore, the second step of the algorithm may still generate a large number of rewritings as a result of the Cartesian product of the buckets.

Example 7.17 Let us consider query Q in Example 7.14 and the views in Example 7.15 with the addition of the following view that gives the projects for which there are employees who live in Paris.

$$\texttt{PROJ1(P\#)} : -\texttt{EMP1(E\#, ENAME, “Paris”)}, \texttt{WORKS(E\#, P\#, TITLE, DUR)} \qquad (d_8)$$

Now, the following mapping unifies the subgoal $\texttt{WORKS}(e, p, \texttt{TITLE}, \texttt{DUR})$ of Q with the subgoal $\texttt{WORKS(E\#, P\#, TITLE, DUR)}$ in view PROJ1:

$$p \rightarrow \texttt{PNAME}$$

Thus, in the first step of the bucket algorithm, PROJ1 is added to bucket b_2. However, PROJ1 cannot be useful in a rewriting of Q since the variable ENAME is not in the head of PROJ1 and thus makes it impossible to join PROJ1 on the variable e of Q. This can be discovered only in the second step when building the conjunctive queries. ♦

The MinCon algorithm addresses the limitations of the bucket algorithm (and the inverse rule algorithm) by considering the query globally and considering how each predicate in the query interacts with the views. It proceeds in two steps like the bucket algorithm. The first step starts by selecting the views that contain subgoals corresponding to subgoals of query Q. However, upon finding a mapping that unifies a subgoal q of Q with a subgoal v in view V, it considers the join predicates in Q and finds the minimum set of additional subgoals of Q that must be mapped to subgoals in V. This set of subgoals of Q is captured by a *MinCon description* (MCD) associated with V. The second step of the algorithm produces a rewritten query by combining the different MCDs. In this second step, unlike in the bucket algorithm, it is not necessary to check that the proposed rewritings are contained in the query

because the way the MCDs are created guarantees that the resulting rewritings will be contained in the original query.

Applied to Example 7.17, the algorithm would create 3 MCDs: two for the views EMP1 and EMP2 containing the subgoal EMP of Q and one for ASG1 containing the subgoal ASG. However, the algorithm cannot create an MCD for PROJ1 because it cannot apply the join predicate in Q. Thus, the algorithm would produce the rewritten query Q' of Example 7.16. Compared with the bucket algorithm, the second step of the MinCon algorithm is much more efficient since it performs fewer combinations of MCDs than buckets.

7.2.4 Query Optimization and Execution

The three main problems of query optimization in multidatabase systems are heterogeneous cost modeling, heterogeneous query optimization (to deal with different capabilities of component DBMSs), and adaptive query processing (to deal with strong variations in the environment—failures, unpredictable delays, etc.). In this section, we describe the techniques for the first two problems. In Sect. 4.6, we presented the techniques for adaptive query processing. These techniques can be used in multidatabase systems as well, provided that the wrappers are able to collect information regarding execution within the component DBMSs.

7.2.4.1 Heterogeneous Cost Modeling

Global cost function definition, and the associated problem of obtaining cost-related information from component DBMSs, is perhaps the most-studied of the three problems. A number of possible solutions have emerged, which we discuss below.

The first thing to note is that we are primarily interested in determining the cost of the lower levels of a query execution tree that correspond to the parts of the query executed at component DBMSs. If we assume that all local processing is "pushed down" in the tree, then we can modify the query plan such that the leaves of the tree correspond to subqueries that will be executed at individual component DBMSs. In this case, we are talking about the determination of the costs of these subqueries that are input to the first level (from the bottom) operators. Cost for higher levels of the query execution tree may be calculated recursively, based on the leaf node costs.

Three alternative approaches exist for determining the cost of executing queries at component DBMSs:

1. **Black-Box Approach.** This approach treats each component DBMS as a black box, running some test queries on it, and from these determines the necessary cost information.

2. **Customized Approach.** This approach uses previous knowledge about the component DBMSs, as well as their external characteristics, to subjectively determine the cost information.
3. **Dynamic Approach.** This approach monitors the runtime behavior of component DBMSs, and dynamically collects the cost information.

We discuss each approach, focusing on the proposals that have attracted the most attention.

Black-Box Approach

In the black-box approach, the cost functions are expressed logically (e.g., aggregate CPU and I/O costs, selectivity factors), rather than on the basis of physical characteristics (e.g., relation cardinalities, number of pages, number of distinct values for each column). Thus, the cost functions for component DBMSs are expressed as

$$\begin{aligned} Cost = {} & initialization\ cost + cost\ to\ find\ qualifying\ tuples \\ & + cost\ to\ process\ selected\ tuples \end{aligned}$$

The individual terms of this formula will differ for different operators. However, these differences are not difficult to specify a priori. The fundamental difficulty is the determination of the term coefficients in these formulas, which change with different component DBMSs. One way to deal with this is to construct a synthetic database (called a *calibrating database*), run queries against it in isolation, and measure the elapsed time to deduce the coefficients.

A problem with this approach is that the results obtained by using a synthetic database may not apply well to real DBMSs. An alternative is based on running probing queries on component DBMSs to determine cost information. Probing queries can, in fact, be used to gather a number of cost information factors. For example, probing queries can be issued to retrieve data from component DBMSs to construct and update the multidatabase catalog. Statistical probing queries can be issued that, for example, count the number of tuples of a relation. Finally, performance measuring probing queries can be issued to measure the elapsed time for determining cost function coefficients.

A special case of probing queries is sample queries. In this case, queries are classified according to a number of criteria, and sample queries from each class are issued and measured to derive component cost information. Query classification can be performed according to query characteristics (e.g., unary operation queries, two-way join queries), characteristics of the operand relations (e.g., cardinality, number of attributes, information on indexed attributes), and characteristics of the underlying component DBMSs (e.g., the access methods that are supported and the policies for choosing access methods).

Classification rules are defined to identify queries that execute similarly, and thus could share the same cost formula. For example, one may consider that two queries that have similar algebraic expressions (i.e., the same algebraic tree shape), but different operand relations, attributes, or constants, are executed the same way if their attributes have the same physical properties. Another example is to assume that join order of a query has no effect on execution since the underlying query optimizer applies reordering techniques to choose an efficient join ordering. Thus, two queries that join the same set of relations belong to the same class, whatever ordering is expressed by the user. Classification rules are combined to define query classes. The classification is performed either top-down by dividing a class into more specific ones or bottom-up by merging two classes into a larger one. In practice, an efficient classification is obtained by mixing both approaches. The global cost function consists of three components: initialization cost, cost of retrieving a tuple, and cost of processing a tuple. The difference is in the way the parameters of this function are determined. Instead of using a calibrating database, sample queries are executed and costs are measured. The global cost equation is treated as a regression equation, and the regression coefficients are calculated using the measured costs of sample queries. The regression coefficients are the cost function parameters. Eventually, the cost model quality is controlled through statistical tests (e.g., F-test): if the tests fail, the query classification is refined until quality is sufficient.

The above approaches require a preliminary step to instantiate the cost model (either by calibration or sampling). This may not be always appropriate, because it would slow down the system each time a new DBMS component is added. One way to address this problem is to progressively learn the cost model from queries. The assumption is that the mediator invokes the underlying component DBMSs by a function call. The cost of a call is composed of three values: the response time to access the first tuple, the whole result response time, and the result cardinality. This allows the query optimizer to minimize either the time to receive the first tuple or the time to process the whole query, depending on end-user requirements. Initially the query processor does not know any statistics about component DBMSs. Then it monitors ongoing queries: it collects processing time of every call and stores it for future estimation. To manage the large amount of collected statistics, the cost manager summarizes them, either without loss of precision or with less precision at the benefit of lower space use and faster cost estimation. Summarization consists of aggregating statistics: the average response time is computed for all the calls that match the same pattern, i.e., those with identical function name and zero or more identical argument values. The cost estimator module is implemented in a declarative language. This allows adding new cost formulas describing the behavior of a particular component DBMS. However, the burden of extending the mediator cost model remains with the mediator developer.

The major drawback of the black-box approach is that the cost model, although adjusted by calibration, is common for all component DBMSs and may not capture their individual specifics. Thus it might fail to estimate accurately the cost of a query executed at a component DBMS that exposes unforeseen behavior.

Customized Approach

The basis of this approach is that the query processors of the component DBMSs are too different to be represented by a unique cost model as used in the black-box approach. It also assumes that the ability to accurately estimate the cost of local subqueries will improve global query optimization. The approach provides a framework to integrate the component DBMSs' cost model into the mediator query optimizer. The solution is to extend the wrapper interface such that the mediator gets some specific cost information from each wrapper. The wrapper developer is free to provide a cost model, partially or entirely. Then, the challenge is to integrate this (potentially partial) cost description into the mediator query optimizer. There are two main solutions.

A first solution is to provide the logic within the wrapper to compute three cost estimates: the time to initiate the query process and receive the first result item (called $reset\_cost$), the time to get the next item (called $advance\_cost$), and the result cardinality. Thus, the total query cost is

$$Total\_access\_cost = reset\_cost + (cardinality - 1) * advance\_cost$$

This solution can be extended to estimate the cost of database procedure calls. In that case, the wrapper provides a cost formula that is a linear equation depending on the procedure parameters. This solution has been successfully implemented to model a wide range of heterogeneous components DBMSs, ranging from a relational DBMS to an image server. It shows that a little effort is sufficient to implement a rather simple cost model and this significantly improves distributed query processing over heterogeneous sources.

A second solution is to use a hierarchical generic cost model. As shown in Fig. 7.14, each node represents a cost rule that associates a query pattern with a cost function for various cost parameters.

The node hierarchy is divided into five levels depending on the genericity of the cost rules (in Fig. 7.14, the increasing width of the boxes shows the increased focus of the rules). At the top level, cost rules apply by default to any DBMS. At the underlying levels, the cost rules are increasingly focused on: specific DBMS, relation, predicate, or query. At the time of wrapper registration, the mediator receives wrapper metadata including cost information, and completes its built-in cost model by adding new nodes at the appropriate level of the hierarchy. This framework is sufficiently general to capture and integrate both general cost knowledge declared as rules given by wrapper developers and specific information derived from recorded past queries that were previously executed. Thus, through an inheritance hierarchy, the mediator cost-based optimizer can support a wide variety of data sources. The mediator benefits from specialized cost information about each component DBMS, to accurately estimate the cost of queries and choose a more efficient QEP.

Example 7.18 Consider the GCS relations `EMP` and `WORKS` (Fig. 7.9). `EMP` is stored at component DBMS db_1 and contains 1,000 tuples. `ASG` is stored at component DBMS db_2 and contains 10,000 tuples. We assume uniform distribution

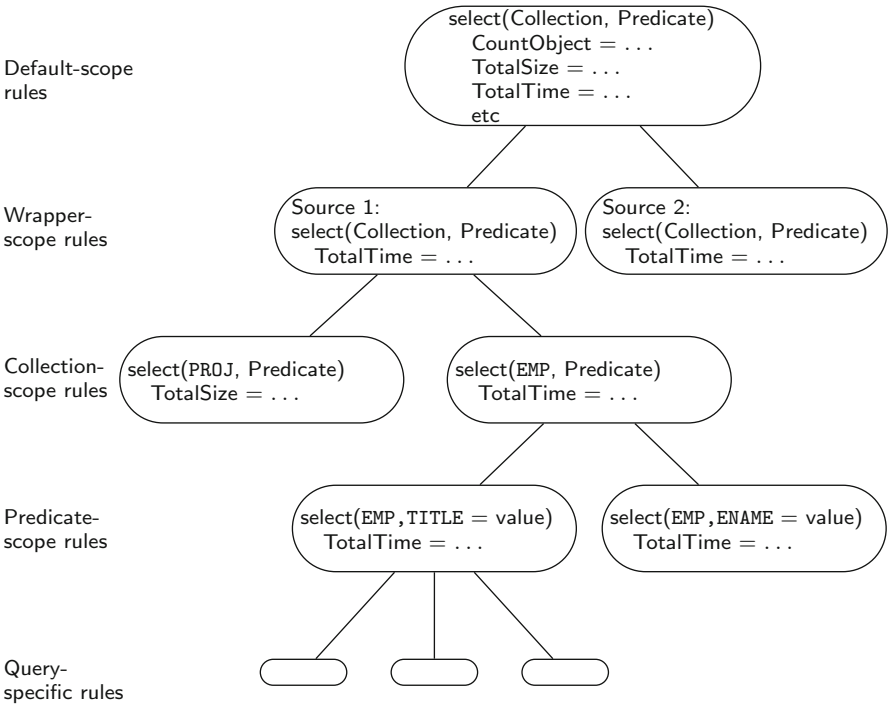

Fig. 7.14 Hierarchical cost formula tree

of attribute values. Half of the WORKS tuples have a duration greater than 6. We detail below some parts of the mediator generic cost model where R and S are two relations, A is the join attribute and we use superscripts to indicate the access method:

$cost(\mathtt{R}) = |\mathtt{R}|$

$cost(\sigma_{predicate}(\mathtt{R})) = cost(\mathtt{R})$ (access to R by sequential scan—by default)

$cost(\mathtt{R} \bowtie_{\mathtt{A}}^{ind} \mathtt{S}) = cost(\mathtt{R}) + |\mathtt{R}| * cost(\sigma_{\mathtt{A}=v}(\mathtt{S}))$ (using an index based (ind) join with the index on S.A)

$cost(\mathtt{R} \bowtie_{\mathtt{A}}^{nl} \mathtt{S}) = cost(\mathtt{R}) + |\mathtt{R}| * cost(\mathtt{S})$ (using a nested-loop (nl) join)

Consider the following global query Q:

```
SELECT *
FROM   EMP NATURAL JOIN WORKS
WHERE  WORKS.DUR>6
```

The cost-based query optimizer generates the following plans to process Q:

$$P_1 = \sigma_{\texttt{DUR}>6}(\texttt{EMP} \bowtie^{ind}_{\texttt{E\#}} \texttt{WORKS})$$

$$P_2 = \texttt{EMP} \bowtie^{nl}_{\texttt{E\#}} \sigma_{\texttt{DUR}>6}(\texttt{WORKS})$$

$$P_3 = \sigma_{\texttt{DUR}>6}(\texttt{WORKS}) \bowtie^{ind}_{\texttt{E\#}} \texttt{EMP}$$

$$P_4 = \sigma_{\texttt{DUR}>6}(\texttt{WORKS}) \bowtie^{nl}_{\texttt{E\#}} \texttt{EMP}$$

Based on the generic cost model, we compute their cost as:

$$\begin{aligned}
cost(P_1) &= cost(\sigma_{\texttt{DUR}>6}(\texttt{EMP} \bowtie^{ind}_{\texttt{E\#}} \texttt{WORKS}) \\
&= cost(\texttt{EMP} \bowtie^{ind}_{\texttt{E\#}} \texttt{WORKS}) \\
&= cost(\texttt{EMP}) + |\texttt{EMP}| * cost(\sigma_{\texttt{E\#}=v}(\texttt{WORKS})) \\
&= |\texttt{EMP}| + |\texttt{EMP}| * |\texttt{WORKS}| = 10,001,000 \\
cost(P_2) &= cost(\texttt{EMP}) + |\texttt{EMP}| * cost(\sigma_{\texttt{DUR}>6}(\texttt{WORKS})) \\
&= cost(\texttt{EMP}) + |\texttt{EMP}| * cost(\texttt{WORKS}) \\
&= |\texttt{EMP}\ | + |\texttt{EMP}| * |\texttt{WORKS}| = 10,001,000 \\
cost(P_3) &= cost(P_4) = |\texttt{WORKS}| + \frac{|\texttt{WORKS}|}{2} * |\texttt{EMP}| \\
&= 5,010,000
\end{aligned}$$

Thus, the optimizer discards plans P_1 and P_2 to keep either P_3 or P_4 for processing Q. Let us assume now that the mediator imports specific cost information about component DBMSs. db_1 exports the cost of accessing EMP tuples as:

$$cost(\sigma_{\texttt{A}=v}(\texttt{R})) = |\sigma_{\texttt{A}=v}(\texttt{R})|$$

db_2 exports the specific cost of selecting WORKS tuples that have a given E# as:

$$cost(\sigma_{\texttt{E\#}=v}(\texttt{WORKS})) = |\sigma_{\texttt{E\#}=v}(\texttt{WORKS})|$$

The mediator integrates these cost functions in its hierarchical cost model, and can now estimate more accurately the cost of the QEPs:

$$\begin{aligned}
cost(P_1) &= |\texttt{EMP}| + |\texttt{EMP}| * |\sigma_{\texttt{E\#}=v}(\texttt{WORKS})| \\
&= 1,000 + 1,000 * 10
\end{aligned}$$

$$= 11,000$$

$$cost(P_2) = |\texttt{EMP}| + |\texttt{EMP}| * |\sigma_{\texttt{DUR}>6}(\texttt{WORKS})|$$

$$= |\texttt{EMP}| + |\texttt{EMP}| * \frac{|\texttt{ASG}|}{2}$$

$$= 5,001,000$$

$$cost(P_3) = |\texttt{WORKS}| + \frac{|\texttt{WORKS}|}{2} * |\sigma_{\texttt{E\#}=v}(\texttt{EMP})|$$

$$= 10,000 + 5,000 * 1$$

$$= 15,000$$

$$cost(P_4) = |\texttt{WORKS}| + \frac{|\texttt{WORKS}|}{2} * |\texttt{EMP}|$$

$$= 10,000 + 5,000 * 1,000$$

$$= 5,010,000$$

The best QEP is now P_1 which was previously discarded because of lack of cost information about component DBMSs. In many situations P_1 is actually the best alternative to process Q_1. ♦

The two solutions just presented are well-suited to the mediator/wrapper architecture and offer a good trade-off between the overhead of providing specific cost information for diverse component DBMSs and the benefit of faster heterogeneous query processing.

Dynamic Approach

The above approaches assume that the execution environment is stable over time. However, in most cases, the execution environment factors are frequently changing. Three classes of environmental factors can be identified based on their dynamicity. The first class for frequently changing factors (every second to every minute) includes CPU load, I/O throughput, and available memory. The second class for slowly changing factors (every hour to every day) includes DBMS configuration parameters, physical data organization on disks, and database schema. The third class for almost stable factors (every month to every year) includes DBMS type, database location, and CPU speed. We focus on solutions that deal with the first two classes.

One way to deal with dynamic environments where network contention, data storage, or available memory changes over time is to extend the sampling method and consider user queries as new samples. Query response time is measured

to adjust the cost model parameters at runtime for subsequent queries. This avoids the overhead of processing sample queries periodically, but still requires heavy computation to solve the cost model equations and does not guarantee that cost model precision improves over time. A better solution, called qualitative, defines the system contention level as the combined effect of frequently changing factors on query cost. The system contention level is divided into several discrete categories: high, medium, low, or no system contention. This allows for defining a multicategory cost model that provides accurate cost estimates, while dynamic factors are varying. The cost model is initially calibrated using probing queries. The current system contention level is computed over time, based on the most significant system parameters. This approach assumes that query executions are short, so the environment factors remain rather constant during query execution. However, this solution does not apply to long running queries, since the environment factors may change rapidly during query execution.

To manage the case where the environment factor variation is predictable (e.g., the daily DBMS load variation is the same every day), the query cost is computed for successive date ranges. Then, the total cost is the sum of the costs for each range. Furthermore, it may be possible to learn the pattern of the available network bandwidth between the MDBS query processor and the component DBMS. This allows adjusting the query cost depending on the actual date.

7.2.4.2 Heterogeneous Query Optimization

In addition to heterogeneous cost modeling, multidatabase query optimization must deal with the issue of the heterogeneous computing capabilities of component DBMSs. For instance, one component DBMS may support only simple select operations, while another may support complex queries involving join and aggregate. Thus, depending on how the wrappers export such capabilities, query processing at the mediator level can be more or less complex. There are two main approaches to deal with this issue depending on the kind of interface between mediator and wrapper: query-based and operator-based.

1. **Query-based.** In this approach, the wrappers support the same query capability, e.g., a subset of SQL, which is translated to the capability of the component DBMS. This approach typically relies on a standard DBMS interface such as Open Database Connectivity (ODBC) and its extensions for the wrappers or SQL Management of External Data (SQL/MED). Thus, since the component DBMSs appear homogeneous to the mediator, query processing techniques designed for homogeneous distributed DBMS can be reused. However, if the component DBMSs have limited capabilities, the additional capabilities must be implemented in the wrappers, e.g., join queries may need to be handled at the wrapper, if the component DBMS does not support join.
2. **Operator-based.** In this approach, the wrappers export the capabilities of the component DBMSs through compositions of relational operators. Thus, there is

more flexibility in defining the level of functionality between the mediator and the wrapper. In particular, the different capabilities of the component DBMSs can be made available to the mediator. This makes wrapper construction easier at the expense of more complex query processing in the mediator. In particular, any functionality that may not be supported by component DBMSs (e.g., join) will need to be implemented at the mediator.

In the rest of this section, we present, in more detail, the approaches to query optimization in these systems.

Query-Based Approach

Since the component DBMSs appear homogeneous to the mediator, one approach is to use a distributed cost-based query optimization algorithm (see Chap. 4) with a heterogeneous cost model (see Sect. 7.2.4.1). However, extensions are needed to convert the distributed execution plan into subqueries to be executed by the component DBMSs and into subqueries to be executed by the mediator. The hybrid two-step optimization technique is useful in this case (see Sect. 4.5.3): in the first step, a static plan is produced by a centralized cost-based query optimizer; in the second step, at startup time, an execution plan is produced by carrying out site selection and allocating the subqueries to the sites. However, centralized optimizers restrict their search space by eliminating bushy join trees from consideration. Almost all the systems use left linear join orders. Consideration of only left linear join trees gives good results in centralized DBMSs for two reasons: it reduces the need to estimate statistics for at least one operand, and indexes can still be exploited for one of the operands. However, in multidatabase systems, these types of join execution plans are not necessarily the preferred ones as they do not allow any parallelism in join execution. As we discussed in earlier chapters, this is also a problem in homogeneous distributed DBMSs, but the issue is more serious in the case of multidatabase systems, because we wish to push as much processing as possible to the component DBMSs.

A way to resolve this problem is to somehow generate bushy join trees and consider them at the expense of left linear ones. One way to achieve this is to apply a cost-based query optimizer to first generate a left linear join tree, and then convert it to a bushy tree. In this case, the left linear join execution plan can be optimal with respect to total time, and the transformation improves the query response time without severely impacting the total time. A hybrid algorithm that concurrently performs a bottom-up and top-down sweep of the left linear join execution tree, transforming it, step-by-step, to a bushy one is possible. The algorithm maintains two pointers, called *upper anchor nodes* (UAN) on the tree. At the beginning, one of these, called the bottom UAN (UAN_B), is set to the grandparent of the leftmost root node (join with R_3 in Fig. 4.9), while the second one, called the top UAN (UAN_T), is set to the root (join with R_5). For each UAN the algorithm selects a *lower anchor node* (LAN). This is the node closest to the UAN and whose right

child subtree's response time is within a designer-specified range, relative to that of the UAN's right child subtree. Intuitively, the LAN is chosen such that its right child subtree's response time is **close** to the corresponding UAN's right child subtree's response time. As we will see shortly, this helps in keeping the transformed bushy tree balanced, which reduces the response time.

At each step, the algorithm picks one of the UAN/LAN pairs (strictly speaking, it picks the UAN and selects the appropriate LAN, as discussed above), and performs the following translation for the segment between that LAN and UAN pair:

1. The left child of UAN becomes the new UAN of the transformed segment.
2. The LAN remains unchanged, but its right child vertex is replaced with a new join node of two subtrees, which were the right child subtrees of the input UAN and LAN.

The UAN mode that will be considered in that particular iteration is chosen according to the following heuristic: choose UAN_B if the response time of its left child subtree is smaller than that of UAN_T's subtree; otherwise, choose UAN_T. If the response times are the same, choose the one with the more unbalanced child subtree.

At the end of each transformation step, the UAN_B and UAN_T are adjusted. The algorithm terminates when $UAN_B = UAN_T$, since this indicates that no further transformations are possible. The resulting join execution tree will be almost balanced, producing an execution plan whose response time is reduced due to parallel execution of the joins.

The algorithm described above starts with a left linear join execution tree that is generated by a centralized DBMS optimizer. These optimizers are able to generate very good plans, but the initial linear execution plan may not fully account for the peculiarities of the distributed multidatabase characteristics, such as data replication. A special global query optimization algorithm can take these into consideration. One proposed algorithm starts from an initial plan and checks for different parenthesizations of this linear join execution order to produce a parenthesized order that is optimal with respect to response time. The result is an (almost) balanced join execution tree. This approach is likely to produce better quality plans at the expense of longer optimization time.

Operator-Based Approach

Expressing the capabilities of the component DBMSs through relational operators allows tight integration of query processing between the mediator and the wrappers. In particular, the mediator/wrapper communication can be in terms of subplans. We illustrate the operator-based approach via the planning functions proposed in the Garlic project. In this approach, the capabilities of the component DBMSs are expressed by the wrappers as planning functions that can be directly called by a centralized query optimizer. It extends a rule-based optimizer with operators to create temporary relations and retrieve locally stored data. It also creates the

PushDown operator that pushes a portion of the work to the component DBMSs where it will be executed. The execution plans are represented, as usual, as operator trees, but the operator nodes are annotated with additional information that specifies the source(s) of the operand(s), whether the results are materialized, and so on. The Garlic operator trees are then translated into operators that can be directly executed by the execution engine.

Planning functions are considered by the optimizer as enumeration rules. They are called by the optimizer to construct subplans using two main functions: accessPlan to access a relation, and joinPlan to join two relations using the access plans. These functions precisely reflect the capabilities of the component DBMSs with a common formalism.

Example 7.19 We consider three component databases, each at a different site. Component database db_1 stores relation EMP(ENO, ENAME, CITY) and component database db_2 stores relation WORKS(ENO, PNAME, DUR). Component database db_3 stores only employee information with a single relation of schema EMPASG(ENAME, CITY, PNAME, DUR), whose primary key is (ENAME, PNAME). Component databases db_1 and db_2 have the same wrapper w_1, whereas db_3 has a different wrapper w_2.

Wrapper w_1 provides two planning functions typical of a relational DBMS. The accessPlan rule

accessPlan(R: relation, A: attribute list, P: select predicate) =
    scan(R, A, P, db(R))

produces a scan operator that accesses tuples of R from its component database db(R) (here we can have db(R) = db_1 or db(R) = db_2), applies select predicate P, and projects on the attribute list A. The joinPlan rule

joinPlan(R_1, R_2: relations, A: attribute list, P: join predicate) =
    join(R_1, R_2, A, P)
condition: $db(R_1) \neq db(R_2)$

produces a join operator that accesses tuples of relations R_1 and R_2 and applies join predicate P and projects on attribute list A. The condition expresses that R_1 and R_2 are stored in different component databases (i.e., db_1 and db_2). Thus, the join operator is implemented by the wrapper.

Wrapper w_2 also provides two planning functions. The accessPlan rule

accessPlan(R: relation, A: attribute list, P: select predicate) =
    fetch(CITY="c")
condition: (CITY="c") $\subseteq P$

produces a fetch operator that directly accesses (entire) employee tuples in component database db_3 whose CITY value is "c." The accessPlan rule

accessPlan(R: relation, A: attribute list, P: select predicate) =
    scan(R, A, P)

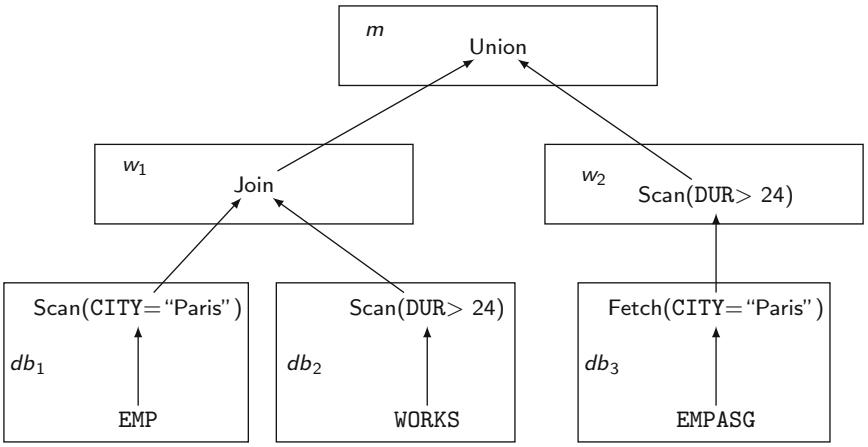

Fig. 7.15 Heterogeneous query execution plan

produces a scan operator that accesses tuples of relation R in the wrapper and applies select predicate P and attribute project list A. Thus, the scan operator is implemented by the wrapper, not the component DBMS.

Consider the following SQL query submitted to mediator m:

```
SELECT ENAME, PNAME, DUR
FROM   EMPASG
WHERE  CITY = "Paris" AND DUR > 24
```

Assuming the GAV approach, the global view EMPASG(ENAME, CITY, PNAME, DUR) can be defined as follows (for simplicity, we prefix each relation by its component database name):

$$\texttt{EMPASG} = (db_1.\texttt{EMP} \bowtie db_2.\texttt{WORKS}) \cup db_3.\texttt{EMPASG}$$

After query rewriting in GAV and query optimization, the operator-based approach could produce the QEP shown in Fig. 7.15. This plan shows that the operators that are not supported by the component DBMS are to be implemented by the wrappers or the mediator. ♦

Using planning functions for heterogeneous query optimization has several advantages in MDBSs. First, planning functions provide a flexible way to express precisely the capabilities of component data sources. In particular, they can be used to model nonrelational data sources such as web sites. Second, since these rules are declarative, so they make wrapper development easier. The only important development for wrappers is the implementation of specific operators, e.g., the scan operator of db_3 in Example 7.19. Finally, this approach can be easily incorporated in an existing, centralized query optimizer.

The operator-based approach has also been successfully used in DIMDBS, an MDBS designed to access multiple databases over the web. DISCO uses the

GAV approach and supports an object data model to represent both mediator and component database schemas and data types. This allows easy introduction of new component databases, easily handling potential type mismatches. The component DBMS capabilities are defined as a subset of an algebraic machine (with the usual operators such as scan, join, and union) that can be partially or entirely supported by the wrappers or the mediator. This gives much flexibility for the wrapper implementors in deciding where to support component DBMS capabilities (in the wrapper or in the mediator). Furthermore, compositions of operators, including specific datasets, can be specified to reflect component DBMS limitations. However, query processing is more complicated because of the use of an algebraic machine and compositions of operators. After query rewriting on the component schemas, there are three main steps:

1. **Search space generation.** The query is decomposed into a number of QEPs, which constitutes the search space for query optimization. The search space is generated using a traditional search strategy such as dynamic programming.
2. **QEP decomposition.** Each QEP is decomposed into a forest of *n wrapper QEPs* and a *composition QEP*. Each wrapper QEP is the largest part of the initial QEP that can be entirely executed by the wrapper. Operators that cannot be performed by a wrapper are moved up to the composition QEP. The composition QEP combines the results of the wrapper QEPs in the final answer, typically through unions and joins of the intermediate results produced by the wrappers.
3. **Cost evaluation.** The cost of each QEP is evaluated using a hierarchical cost model discussed in Sect. 7.2.4.1.

7.2.5 Query Translation and Execution

Query translation and execution is performed by the wrappers using the component DBMSs. A wrapper encapsulates the details of one or more component databases, each supported by the same DBMS (or file system). It also exports to the mediator the component DBMS capabilities and cost functions in a common interface. One of the major practical uses of wrappers has been to allow an SQL-based DBMS to access non-SQL databases.

The main function of a wrapper is conversion between the common interface and the DBMS-dependent interface. Figure 7.16 shows these different levels of interfaces between the mediator, the wrapper, and the component DBMSs. Note that, depending on the level of autonomy of the component DBMSs, these three components can be located differently. For instance, in the case of strong autonomy, the wrapper should be at the mediator site, possibly on the same server. Thus, communication between a wrapper and its component DBMS incurs network cost. However, in the case of a cooperative component database (e.g., within the same organization), the wrapper could be installed at the component DBMS site, much

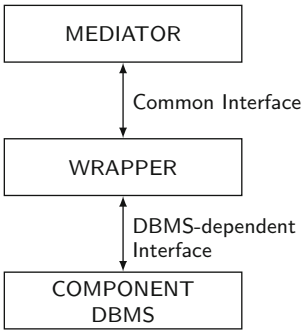

Fig. 7.16 Wrapper interfaces

like an ODBC driver. Thus, communication between the wrapper and the component DBMS is much more efficient.

The information necessary to perform conversion is stored in the wrapper schema that includes the local schema exported to the mediator in the common interface (e.g., relational) and the schema mappings to transform data between the local schema and the component database schema and vice versa. We discussed schema mappings in Sect. 7.1.4. Two kinds of conversion are needed. First, the wrapper must translate the input QEP generated by the mediator and expressed in a common interface into calls to the component DBMS using its DBMS-dependent interface. These calls yield query execution by the component DBMS that return results expressed in the DBMS-dependent interface. Second, the wrapper must translate the results to the common interface format so that they can be returned to the mediator for integration. In addition, the wrapper can execute operations that are not supported by the component DBMS (e.g., the scan operation by wrapper w_2 in Fig. 7.15).

As discussed in Sect. 7.2.4.2, the common interface to the wrappers can be query-based or operator-based. The problem of translation is similar in both approaches. To illustrate query translation in the following example, we use the query-based approach with the SQL/MED standard that allows a relational DBMS to access external data represented as foreign relations in the wrapper's local schema. This example illustrates how a very simple data source can be wrapped to be accessed through SQL.

Example 7.20 We consider relation EMP(ENO, ENAME, CITY) stored in a very simple component database, in server *ComponentDB*, built with Unix text files. Each EMP tuple can then be stored as a line in a file, e.g., with the attributes separated by ":". In SQL/MED, the definition of the local schema for this relation together with the mapping to a Unix file can be declared as a foreign relation with the following statement:

```
CREATE  FOREIGN TABLE EMP
        ENO INTEGER,
        ENAME VARCHAR(30),
```

```
        CITY VARCHAR(20)
SERVER  ComponentDB
OPTIONS (Filename '/usr/EngDB/emp.txt',
         Delimiter ':')
```

Then, the mediator can send SQL statements to the wrapper that supports access to this relation. For instance, the query

```
SELECT ENAME
FROM   EMP
```

can be translated by the wrapper using the following Unix shell command to extract the relevant attribute:

```
cut -d: -f2 /usr/EngDB/emp
```

Additional processing, e.g., for type conversion, can then be done using programming code. ♦

Wrappers are mostly used for read-only queries, which makes query translation and wrapper construction relatively easy. Wrapper construction typically relies on tools with reusable components to generate most of the wrapper code. Furthermore, DBMS vendors provide wrappers for transparently accessing their DBMS using standard interfaces. However, wrapper construction is much more difficult if updates to component databases are to be supported through wrappers (as opposed to directly updating the component databases through their DBMS). A major problem is due to the heterogeneity of integrity constraints between the common interface and the DBMS-dependent interface. As discussed in Chap. 3, integrity constraints are used to reject updates that violate database consistency. In modern DBMSs, integrity constraints are explicit and specified as rules that are part of the database schema. However, in older DBMSs or simpler data sources (e.g., files), integrity constraints are implicit and implemented by specific code in the applications. For instance, in Example 7.20, there could be applications with some embedded code that rejects insertions of new lines with an existing ENO in the EMP text file. This code corresponds to a unique key constraint on ENO in relation EMP but is not readily available to the wrapper. Thus, the main problem of updating through a wrapper is guaranteeing component database consistency by rejecting all updates that violate integrity constraints, whether they are explicit or implicit. A software engineering solution to this problem uses a tool with reverse engineering techniques to identify within application code the implicit integrity constraints that are then translated into validation code in the wrappers.

Another major problem is wrapper maintenance. Query translation relies heavily on the mappings between the component database schema and the local schema. If the component database schema is changed to reflect the evolution of the component database, then the mappings can become invalid. For instance, in Example 7.20, the administrator may switch the order of the fields in the EMP file. Using invalid mappings may prevent the wrapper from producing correct results. Since the

component databases are autonomous, detecting and correcting invalid mappings is important. The techniques to do so are those for mapping maintenance that we discussed in this chapter.

7.3 Conclusion

In this chapter, we discussed the bottom-up database design process, which we called database integration and how to execute queries over databases constructed in this manner. Database integration is the process of creating a GCS (or a mediated schema) and determining how each LCS maps to it. A fundamental separation is between data warehouses where the GCS is instantiated and materialized, and data integration systems where the GCS is merely a virtual view.

Although the topic of database integration has been studied extensively for a long time, almost all of the work has been fragmented. Individual projects focus either on schema matching, or data cleaning, or schema mapping. What is needed is an end-to-end methodology for database integration that is semiautomatic with sufficient hooks for expert involvement. One approach to such a methodology is the work of Bernstein and Melnik [2007], which provides the beginnings of a comprehensive "end-to-end" methodology.

A related concept that has received considerable discussion in literature is *data exchange*, which is defined as "the problem of taking data structured under a source schema and creating an instance of a target schema that reflects the source data as accurately as possible" [Fagin et al. 2005]. This is very similar to the physical integration (i.e., materialized) data integration, such as data warehouses, that we discussed in this chapter. A difference between data warehouses and the materialization approaches as addressed in data exchange environments is that data warehouse data typically belongs to one organization and can be structured according to a well-defined schema, while in data exchange environments data may come from different sources and contain heterogeneity.

Our focus in this chapter has been on integrating *databases*. Increasingly, however, the data that are used in distributed applications involve those that are not in a database. An interesting new topic of discussion among researchers is the integration of *structured* data that is stored in databases and *unstructured* data that is maintained in other systems (web servers, multimedia systems, digital libraries, etc.). We discuss these in Chap. 12 where we focus on the integration of data from different web repositories and introduce the recent concept of *data lakes*.

Another issue that we ignored in this chapter is data integration when a GCS does not exist or cannot be specified. The issue arises particularly in the peer-to-peer systems where the scale and the variety of data sources make it quite difficult (if not impossible) to design a GCS. We will discuss data integration in peer-to-peer systems in Chap. 9.

The second part of this chapter focused on query processing in multidatabase systems, which is significantly more complex than in tightly integrated and homogeneous distributed DBMSs. In addition to being distributed, component databases may be autonomous, have different database languages and query processing capabilities, and exhibit varying behavior. In particular, component databases may range from full-fledged SQL databases to very simple data sources (e.g., text files).

In this chapter, we addressed these issues by extending and modifying the distributed query processing architecture presented in Chap. 4. Assuming the popular mediator/wrapper architecture, we isolated the three main layers by which a query is successively rewritten (to bear on local relations) and optimized by the mediator, and then translated and executed by the wrappers and component DBMSs. We also discussed how to support OLAP queries in a multidatabase, an important requirement of decision-support applications. This requires an additional layer of translation from OLAP multidimensional queries to relational queries. This layered architecture for multidatabase query processing is general enough to capture very different variations. This has been useful to describe various query processing techniques, typically designed with different objectives and assumptions.

The main techniques for multidatabase query processing are query rewriting using multidatabase views, multidatabase query optimization and execution, and query translation and execution. The techniques for query rewriting using multidatabase views differ in major ways depending on whether the GAV or LAV integration approach is used. Query rewriting in GAV is similar to data localization in homogeneous distributed database systems. But the techniques for LAV (and its extension GLAV) are much more involved and it is often not possible to find an equivalent rewriting for a query, in which case a query that produces a maximum subset of the answer is necessary. The techniques for multidatabase query optimization include cost modeling and query optimization for component databases with different computing capabilities. These techniques extend traditional distributed query processing by focusing on heterogeneity. Besides heterogeneity, an important problem is to deal with the dynamic behavior of the component DBMSs. Adaptive query processing addresses this problem with a dynamic approach whereby the query optimizer communicates at runtime with the execution environment in order to react to unforeseen variations of runtime conditions. Finally, we discussed the techniques for translating queries for execution by the components DBMSs and for generating and managing wrappers.

The data model used by the mediator can be relational, object-oriented, or others. In this chapter, for simplicity, we assumed a mediator with a relational model that is sufficient to explain the multidatabase query processing techniques. However, when dealing with data sources on the Web, a richer mediator model such as object-oriented or semistructured (e.g., XML- or RDF-based) may be preferred. This requires significant extensions to query processing techniques.

7.4 Bibliographic Notes

A large volume of literature exists on the topic of this chapter. The work goes back to early 1980s and which is nicely surveyed by Batini et al. [1986]. Subsequent work is nicely covered by Elmagarmid et al. [1999] and Sheth and Larson [1990]. Another more recent good review of the field is by Jhingran et al. [2002].

The book by Doan et al. [2012] provides the broadest coverage of the subject. There are a number of overview papers on the topic. Bernstein and Melnik [2007] provide a very nice discussion of the integration methodology. It goes further by comparing the model management work with some of the data integration research. Halevy et al. [2006] review the data integration work in the 1990s, focusing on the Information Manifold system [Levy et al. 1996a], that uses a LAV approach. The paper provides a large bibliography and discusses the research areas that have been opened in the intervening years. Haas [2007] takes a comprehensive approach to the entire integration process and divides it into four phases: understanding that involves discovering relevant information (keys, constraints, data types, etc.), analyzing it to assess quality, and to determine statistical properties; standardization whereby the best way to represent the integrated information is determined; specification that involves the configuration of the integration process; and execution, which is the actual integration. The specification phase includes the techniques defined in this paper.

The LAV and GAV approaches are introduced and discussed by Lenzerini [2002], Koch [2001], and Calì and Calvanese [2002]. The GLAV approach is discussed in [Friedman et al. 1999] and [Halevy 2001]. A large number of systems have been developed that have tested the LAV versus GAV approaches. Many of these focus on querying over integrated systems. Examples of LAV approaches are described in the papers [Duschka and Genesereth 1997, Levy et al. 1996b, Manolescu et al. 2001], while examples of GAV are presented in papers [Adali et al. 1996a, Garcia-Molina et al. 1997, Haas et al. 1997b].

Topics of structural and semantic heterogeneity have occupied researchers for quite some time. While the literature on this topic is quite extensive, some of the interesting publications that discuss structural heterogeneity are [Dayal and Hwang 1984, Kim and Seo 1991, Breitbart et al. 1986, Krishnamurthy et al. 1991, Batini et al. 1986] (Batini et al. [1986] also discuss the structural conflicts introduced in this chapter) and those that focus on semantic heterogeneity are [Sheth and Kashyap 1992, Hull 1997, Ouksel and Sheth 1999, Kashyap and Sheth 1996, Bright et al. 1994, Ceri and Widom 1993, Vermeer 1997]. We should note that this list is seriously incomplete.

Various proposals for the canonical model for the GCS exist. The ones we discussed in this chapter and their sources are the ER model [Palopoli et al. 1998, Palopoli 2003, He and Ling 2006], object-oriented model [Castano and Antonellis 1999, Bergamaschi 2001], graph model (which is also used for determining structural similarity) [Palopoli et al. 1999, Milo and Zohar 1998, Melnik et al. 2002,

Do and Rahm 2002, Madhavan et al. 2001], tree model [Madhavan et al. 2001], and XML [Yang et al. 2003].

Doan and Halevy [2005] provide a very good overview of the various schema matching techniques, proposing a different, and simpler, classification of the techniques as rule-based, learning-based, and combined. More works in schema matching are surveyed by Rahm and Bernstein [2001], which gives a very nice comparison of various proposals. The interschema rules we discussed in this chapter are due to Palopoli et al. [1999]. The classical source for the ranking aggregation functions used in matching is [Fagin 2002].

A number of systems have been developed demonstrating the feasibility of various schema matching approaches. Among rule-based techniques, one can cite DIKE [Palopoli et al. 1998, Palopoli 2003, Palopoli et al. 2003], DIPE, which is an earlier version of this system [Palopoli et al. 1999], TranSCM [Milo and Zohar 1998], ARTEMIS [Bergamaschi 2001], similarity flooding [Melnik et al. 2002], CUPID [Madhavan et al. 2001], and COMA [Do and Rahm 2002]. For learning-based matching, Autoplex [Berlin and Motro 2001] implements a naïve Bayesian classifier, which is also the approach proposed by Doan et al. [2001, 2003a] and Naumann et al. [2002]. In the same class, decision trees are discussed in [Embley et al. 2001, 2002], and iMAP in [Dhamankar et al. 2004].

Roth and Schwartz [1997], Tomasic et al. [1997], and Thiran et al. [2006] focus on various aspects of wrapper technology. A software engineering solution to the problem of wrapper creation and maintenance, considering integrity control, is proposed in [Thiran et al. 2006].

Some sources for binary integration are [Batini et al. 1986, Pu 1988, Batini and Lenzirini 1984, Dayal and Hwang 1984, Melnik et al. 2002], while n-ary mechanisms are discussed in [Elmasri et al. 1987, Yao et al. 1982, He et al. 2004]. For some database integration tools the readers can consult [Sheth et al. 1988a], [Miller et al. 2001] that discuss Clio, and [Roitman and Gal 2006] that describes OntoBuilder.

Mapping creation algorithm in Sect. 7.1.4.1 is due to Miller et al. [2000], Yan et al. [2001], and [Popa et al. 2002]. Mapping maintenance is discussed by Velegrakis et al. [2004].

Data cleaning has gained significant interest in recent years as the integration efforts opened up to data sources more widely. The literature is rich on this topic and is well discussed in the book by Ilyas and Chu [2019]. In this context, the distinction between schema-level and instance-level cleaning is due to Rahm and Do [2000]. The data cleaning operators we discussed are column splitting [Raman and Hellerstein 2001], map operator [Galhardas et al. 2001], and fuzzy match [Chaudhuri et al. 2003].

Work on multidatabase query processing started in the early 1980s with the first multidatabase systems (e.g., [Brill et al. 1984, Dayal and Hwang 1984] and [Landers and Rosenberg 1982]). The objective then was to access different databases within an organization. In the 1990s, the increasing use of the Web for accessing all kinds of data sources triggered renewed interest and much more work in multidatabase query processing, following the popular mediator/wrapper architecture [Wiederhold

1992]. A brief overview of multidatabase query optimization issues can be found in [Meng et al. 1993]. Good discussions of multidatabase query processing can be found in [Lu et al. 1992, 1993], in Chapter 4 of [Yu and Meng 1998] and in [Kossmann 2000].

Query rewriting using views is discussed in [Levy et al. 1995] and surveyed in [Halevy 2001]. In [Levy et al. 1995], the general problem of finding a rewriting using views is shown to be NP-complete in the number of views and the number of subgoals in the query. The unfolding technique for rewriting a query expressed in Datalog in GAV was proposed in [Ullman 1997]. The main techniques for query rewriting using views in LAV are the bucket algorithm [Levy et al. 1996b], the inverse rule algorithm [Duschka and Genesereth 1997], and the MinCon algorithm [Pottinger and Levy 2000].

The three main approaches for heterogeneous cost modeling are discussed in [Zhu and Larson 1998]. The black-box approach is used in [Du et al. 1992, Zhu and Larson 1994]; the techniques in this group are probing queries: [Zhu and Larson 1996a], sample queries (which are a special case of probing) [Zhu and Larson 1998], and learning the cost over time as queries are posed and answered [Adali et al. 1996b]. The customized approach is developed in [Zhu and Larson 1996b, Roth et al. 1999, Naacke et al. 1999]; in particular cost computation can be done within the wrapper (as in Garlic) [Roth et al. 1999] or a hierarchical cost model can be developed (as in Disco) [Naacke et al. 1999]. The dynamic approach is used in [Zhu et al. 2000], [Zhu et al. 2003], and [Rahal et al. 2004] and also discussed by Lu et al. [1992]. Zhu [1995] discusses a dynamic approaching sampling and Zhu et al. [2000] present a qualitative approach.

The algorithm we described to illustrate the query-based approach to heterogeneous query optimization (Sect. 7.2.4.2) has been proposed in [Du et al. 1995] and also discussed in [Evrendilek et al. 1997]. To illustrate the operator-based approach, we described the popular solution with planning functions proposed in the Garlic project [Haas et al. 1997a]. The operator-based approach has been also used in DISCO, a multidatabase system to access component databases over the web [Tomasic et al. 1996, 1998].

The case for adaptive query processing is made by a number of researchers in a number of environments. Amsaleg et al. [1996] show why static plans cannot cope with unpredictability of data sources; the problem exists in continuous queries [Madden et al. 2002b], expensive predicates [Porto et al. 2003], and data skew [Shah et al. 2003]. The adaptive approach is surveyed in [Hellerstein et al. 2000, Gounaris et al. 2002]. The best-known dynamic approach is eddy (see Chap. 4), which is discussed in [Avnur and Hellerstein 2000]. Other important techniques for adaptive query processing are query scrambling [Amsaleg et al. 1996, Urhan et al. 1998], Ripple joins [Haas and Hellerstein 1999b], adaptive partitioning [Shah et al. 2003], and Cherry picking [Porto et al. 2003]. Major extensions to eddy are state modules [Raman et al. 2003] and distributed Eddies [Tian and DeWitt 2003].

In this chapter, we focused on the integration of structured data captured in databases. The more general problem of integrating both structured and unstructured data is discussed by Halevy et al. [2003] and Somani et al. [2002]. A different

generality direction is investigated by Bernstein and Melnik [2007], who propose a model management engine that "supports operations to match schemas, compose mappings, diff schemas, merge schemas, translate schemas into different data models, and generate data transformations from mappings."

In addition to the systems we noted above, in this chapter we referred to a number of other systems. These and their main sources are the following: SEMINT [Li and Clifton 2000, Li et al. 2000], ToMAS [Velegrakis et al. 2004], Maveric [McCann et al. 2005], and Aurora [Yan 1997, Yan et al. 1997].

Exercises

Problem 7.1 Distributed database systems and distributed multidatabase systems represent two different approaches to systems design. Find three real-life applications for which each of these approaches would be more appropriate. Discuss the features of these applications that make them more favorable for one approach or the other.

Problem 7.2 Some architectural models favor the definition of a global conceptual schema, whereas others do not. What do you think? Justify your selection with detailed technical arguments.

Problem 7.3 (*) Give an algorithm to convert a relational schema to an entity-relationship one.

Problem 7.4 (**) Consider the two databases given in Figs. 7.17 and 7.18 and described below. Design a global conceptual schema as a union of the two databases by first translating them into the E-R model.

Figure 7.17 describes a relational race database used by organizers of road races and Fig. 7.18 describes an entity-relationship database used by a shoe manufacturer. The semantics of each of these database schemas is discussed below. Figure 7.17 describes a relational road race database with the following semantics:

DIRECTOR is a relation that defines race directors who organize races; we assume that each race director has a unique name (to be used as the key), a phone number, and an address.

```
DIRECTOR(NAME, PHONE_NO, ADDRESS)
LICENSES(LIC_NO, CITY, DATE, ISSUES, COST, DEPT, CONTACT)
RACER(NAME, ADDRESS, MEM_NUM)
SPONSOR(SP_NAME, CONTACT)
RACE(R_NO, LIC_NO, DIR, MAL_WIN, FRM_WIN, SP_NAME)
```

Fig. 7.17 Road race database

LICENSES is required because all races require a governmental license, which is issued by a CONTACT in a department who is the ISSUER, possibly contained within another government department DEPT; each license has a unique LIC_NO (the key), which is issued for use in a specific CITY on a specific DATE with a certain COST.

RACER is a relation that describes people who participate in a race. Each person is identified by NAME, which is not sufficient to identify them uniquely, so a compound key formed with the ADDRESS is required. Finally, each racer may have a MEM_NUM to identify him or her as a member of the racing fraternity, but not all competitors have membership numbers.

SPONSOR indicates which sponsor is funding a given race. Typically, one sponsor funds a number of races through a specific person (CONTACT), and a number of races may have different sponsors.

RACE uniquely identifies a single race which has a license number (LIC_NO) and race number (R_NO) (to be used as a key, since a race may be planned without acquiring a license yet); each race has a winner in the male and female groups (MAL_WIN and FEM_WIN) and a race director (DIR).

Figure 7.18 illustrates an entity-relationship schema used by the sponsor's database system with the following semantics:

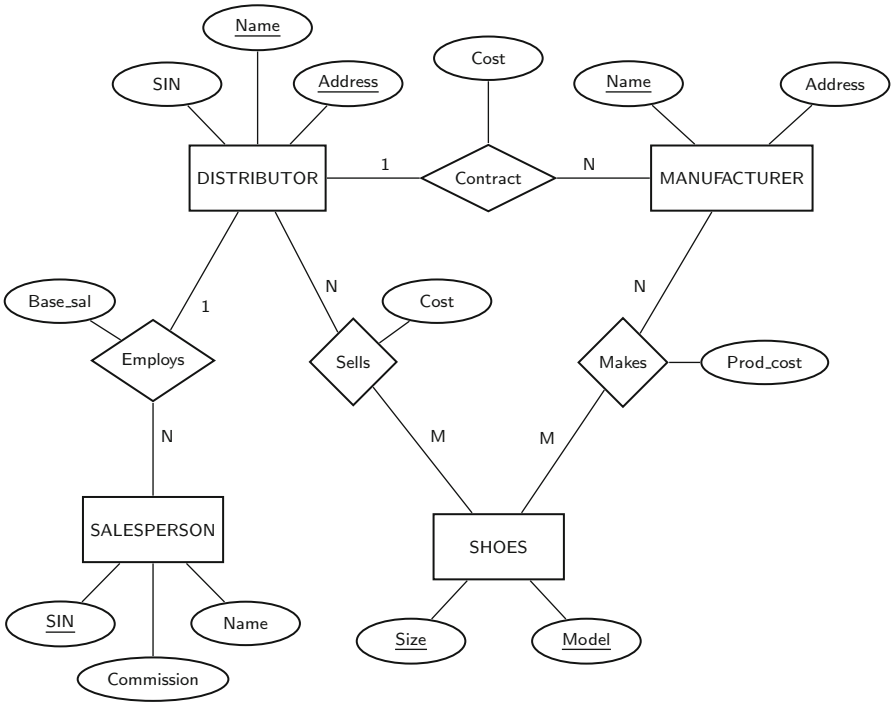

Fig. 7.18 Sponsor database

SHOES are produced by sponsors of a certain MODEL and SIZE, which forms the key to the entity.

MANUFACTURER is identified uniquely by NAME and resides at a certain ADDRESS.

DISTRIBUTOR is a person that has a NAME and ADDRESS (which are necessary to form the key) and a SIN number for tax purposes.

SALESPERSON is a person (entity) who has a NAME, earns a COMMISSION, and is uniquely identified by his or her SIN number (the key).

Makes is a relationship that has a certain fixed production cost (PROD_COST). It indicates that a number of different shoes are made by a manufacturer, and that different manufacturers produce the same shoe.

Sells is a relationship that indicates the wholesale COST to a distributor of shoes. It indicates that each distributor sells more than one type of shoe, and that each type of shoe is sold by more than one distributor.

Contract is a relationship whereby a distributor purchases, for a COST, exclusive rights to represent a manufacturer. Note that this does not preclude the distributor from selling different manufacturers' shoes.

Employs indicates that each distributor hires a number of salespeople to sell the shoes; each earns a BASE_SALARY.

Problem 7.5 (*) Consider three sources:

- Database 1 has one relation `Area(Id, Field)` providing areas of specialization of employees; the Id field identifies an employee.
- Database 2 has two relations, `Teach(Professor, Course)` and `In(Course, Field)`; `Teach` indicates the courses that each professor teaches and `In` specifies possible fields that a course can belong to.
- Database 3 has two relations, `Grant(Researcher, GrantNo)` for grants given to researchers, and `For(GrantNo, Field)` indicating which fields the grants are for.

The objective is to build a GCS with two relations: `Works(Id, Project)` stating that an employee works for a particular project, and `Area(Project, Field)` associating projects with one or more fields.

(a) Provide a LAV mapping between Database 1 and the GCS.

(b) Provide a GLAV mapping between the GCS and the local schemas.

(c) Suppose one extra relation, `Funds(GrantNo, Project)`, is added to Database 3. Provide a GAV mapping in this case.

Problem 7.6 Consider a GCS with the following relation: `Person(Name, Age, Gender)`. This relation is defined as a view over three LCSs as follows:

```
CREATE VIEW Person AS
SELECT Name, Age, "male" AS Gender
FROM   SoccerPlayer
UNION
SELECT Name, NULL AS Age, Gender
```

```
FROM   Actor
UNION
SELECT Name, Age, Gender
FROM   Politician
WHERE  Age > 30
```

For each of the following queries, discuss which of the three local schemas (`SoccerPlayer`, `Actor`, and `Politician`) contributes to the global query result.

(a) `SELECT Name FROM Person`
(b) `SELECT Name FROM Person WHERE Gender = "female"`
(c) `SELECT Name FROM Person WHERE Age > 25`
(d) `SELECT Name FROM Person WHERE Age < 25`
(e) `SELECT Name FROM Person WHERE Gender = "male" AND Age = 40`

Problem 7.7 A GCS with the relation `Country(Name, Continent, Population, HasCoast)` describes countries of the world. The attribute `HasCoast` indicates if the country has direct access to the sea. Three LCSs are connected to the global schema using the LAV approach as follows:

```
CREATE VIEW EuropeanCountry AS
SELECT Name, Continent, Population, HasCoast
FROM   Country
WHERE  Continent = "Europe"

CREATE VIEW BigCountry AS
SELECT Name, Continent, Population, HasCoast
FROM   Country
WHERE  Population >= 30000000

CREATE VIEW MidsizeOceanCountry AS
SELECT Name, Continent, Population, HasCoast
FROM   Country
WHERE  HasCoast = true AND Population > 10000000
```

(a) For each of the following queries, discuss the results with respect to their completeness, i.e., verify if the (combination of the) local sources cover all relevant results.

1. `SELECT Name FROM Country`
2. `SELECT Name FROM Country WHERE Population > 40`
3. `SELECT Name FROM Country WHERE Population > 20`

(b) For each of the following queries, discuss which of the three LCSs are necessary for the global query result.

1. `SELECT Name FROM Country`

2. `SELECT Name FROM Country WHERE Population > 30 AND Continent = "Europe"`
3. `SELECT Name FROM Country WHERE Population < 30`
4. `SELECT Name FROM Country WHERE Population > 30 AND HasCoast = true`

Problem 7.8 Consider the following two relations `PRODUCT` and `ARTICLE` that are specified in a simplified SQL notation. The perfect schema matching correspondences are denoted by arrows.

`PRODUCT`	⟶	`ARTICLE`
`Id`: int PRIMARY KEY	⟶	`Key`: varchar(255) PRIMARY KEY
`Name`: varchar(255)	⟶	`Title`: varchar(255)
`DeliveryPrice`: float	⟶	`Price`: real
`Description`: varchar(8000)	⟶	`Information`: varchar(5000)

(a) For each of the five correspondences, indicate which of the following match approaches will probably identify the correspondence:

1. Syntactic comparison of element names, e.g., using edit distance string similarity
2. Comparison of element names using a synonym lookup table
3. Comparison of data types
4. Analysis of instance data values

(b) Is it possible for the listed matching approaches to determine false correspondences for these match tasks? If so, give an example.

Problem 7.9 Consider two relations `S(a, b, c)` and `T(d, e, f)`. A match approach determines the following similarities between the elements of `S` and `T`:

	`T.d`	`T.e`	`T.f`
`S.a`	0.8	0.3	0.1
`S.b`	0.5	0.2	0.9
`S.c`	0.4	0.7	0.8

Based on the given matcher's result, derive an overall schema match result with the following characteristics:

- Each element participates in exactly one correspondence.
- There is no correspondence where both elements match an element of the opposite schema with a higher similarity than its corresponding counterpart.

Problem 7.10 (*) Figure 7.19 illustrates the schema of three different data sources:

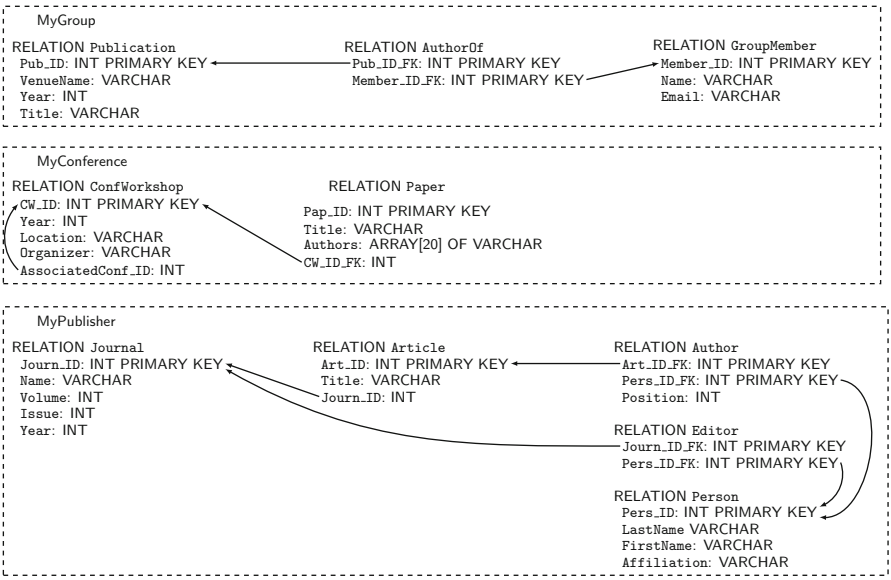

Fig. 7.19 Figure for Exercise 7.10

- MyGroup contains publications authored by members of a working group;
- MyConference contains publications of a conference series and associated workshops;
- MyPublisher contains articles that are published in journals.

The arrows show the foreign key-to-primary key relationships; note that we do not follow the proper SQL syntax of specifying foreign key relationships to save space—we resort to arrows.

The sources are defined as follows:
MyGroup

- `Publication`
  - `Pub_ID`: unique publication ID
  - `VenueName`: name of the journal, conference, or workshop
  - `VenueType`: "journal," "conference," or "workshop"
  - `Year`: year of publication
  - `Title`: publication's title
- `AuthorOf`
  - many-to-many relationship representing "group member is author of publication"
- `GroupMember`
  - `Member_ID`: unique member ID

- `Name`: name of the group member
- `Email`: email address of the group member

MyConference

- `ConfWorkshop`
  - `CW_ID`: unique ID for the conference/workshop
  - `Name`: name of the conference or workshop
  - `Year`: year when the event takes place
  - `Location`: event's location
  - `Organizer`: name of the organizing person
  - `AssociatedConf_ID_FK`: value is NULL if it is a conference, ID of the associated conference if the event is a workshop (this is assuming that workshops are organized in conjunction with a conference)
- `Paper`
  - `Pap_ID`: unique paper ID
  - `Title`: paper's title
  - `Author`: array of author names
  - `CW_ID_FK`: conference/workshop where the paper is published

MyPublisher

- `Journal`
  - `Journ_ID`: unique journal ID
  - `Name`: journal's name
  - `Year`: year when the event takes place
  - `Volume`: journal volume
  - `Issue`: journal issue
- `Article`
  - `Art_ID`: unique article ID
  - `Title`: title of the article
  - `Journ_ID_FK`: journal where the article is published
- `Person`
  - `Pers_ID`: unique person ID
  - `LastName`: last name of the person
  - `FirstName`: first name of the person
  - `Affiliation`: person's affiliation (e.g., the name of a university)
- `Author`
  - represents the many-to-many relationship for "person is author of article"
  - Position: author's position in the author list (e.g., first author has Position 1)
- `Editor`

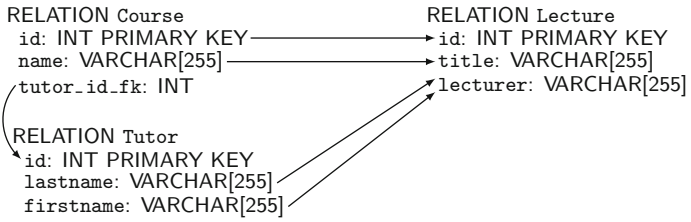

Fig. 7.20 Figure for Exercise 7.11

- represents the many-to-many relationship for "person is editor of journal issue"

(a) Identify all schema matching correspondences between the schema elements of the sources. Use the names and data types of the schema elements as well as the given description.

(b) Classify your correspondences along the following dimensions:

1. Type of schema elements (e.g., attribute–attribute or attribute–relation)
2. Cardinality (e.g., 1:1 or 1:N)

(c) Give a consolidated global schema that covers all information of the source schemas.

Problem 7.11 (*) Figure 7.20 illustrates (using a simplified SQL syntax) two sources $Source_1$ and $Source_2$. $Source_1$ has two relations, `Course` and `Tutor`, and $Source_2$ has only one relation, `Lecture`. The solid arrows denote schema matching correspondences. The dashed arrow represents a foreign key relationship between the two relations in $Source_1$.

The following are four schema mappings (represented as SQL queries) to transform $Source_1$'s data into $Source_2$.

```
1. SELECT C.id, C.name as Title, CONCAT(T.lastname,
          T.firstname) AS Lecturer
   FROM   Course AS C
   JOIN   Tutor AS T ON (C.tutor_id_fk = T.id)
2. SELECT C.id, C.name AS Title, NULL AS Lecturer
   FROM   Course AS C
   UNION
   SELECT T.id AS ID, NULL AS Title, T,
          lastname AS Lecturer
   FROM   Course AS C
   FULL OUTER JOIN Tutor AS T ON(C.tutor_id_fk=T.id)
3. SELECT C.id, C.name as Title, CONCAT(T.lastname,
          T.firstname) AS Lecturer
   FROM   Course AS C
   FULL OUTER JOIN Tutor AS T ON(C.tutor_id_fk=T.id)
```

Discuss each of these schema mappings with respect to the following questions:

(a) Is the mapping meaningful?
(b) Is the mapping complete (i.e., are all data instances of O_1 transformed)?
(c) Does the mapping potentially violate key constraints?

Problem 7.12 (*) Consider three data sources:

- Database 1 has one relation `AREA(ID, FIELD)` providing areas of specialization of employees where ID identifies an employee.
- Database 2 has two relations: `TEACH(PROFESSOR, COURSE)` and `IN(COURSE, FIELD)` specifying possible fields a course can belong to.
- Database 3 has two relations: `GRANT(RESEARCHER, GRANT#)` for grants given to researchers, and `FOR(GRANT#, FIELD)` indicating the fields that the grants are in.

Design a global schema with two relations: `WORKS(ID, PROJECT)` that records which projects employees work in, and `AREA(PROJECT, FIELD)` that associates projects with one or more fields for the following cases:

(a) There should be a LAV mapping between Database 1 and the global schema.
(b) There should be a GLAV mapping between the global schema and the local schemas.
(c) There should be a GAV mapping when one extra relation `FUNDS(GRANT#, PROJECT)` is added to Database 3.

Problem 7.13 (**) Logic (first-order logic, to be precise) has been suggested as a uniform formalism for schema translation and integration. Discuss how logic can be useful for this purpose.

Problem 7.14 (**) Can any type of global optimization be performed on global queries in a multidatabase system? Discuss and formally specify the conditions under which such optimization would be possible.

Problem 7.15 (**) Consider the global relations `EMP(ENAME, TITLE, CITY)` and `ASG(ENAME, PNAME, CITY, DUR)`. `CITY` in `ASG` is the location of the project of name `PNAME` (i.e., `PNAME` functionally determines `CITY`). Consider the local relations `EMP1(ENAME,TITLE, CITY)`, `EMP2(ENAME, TITLE, CITY)`, `PROJ1(PNAME, CITY)`, `PROJ2(PNAME, CITY)`, and `ASG1(ENAME, PNAME, DUR)`. Consider query Q which selects the names of the employees assigned to a project in Rio de Janeiro for more than 6 months and the duration of their assignment.

(a) Assuming the GAV approach, perform query rewriting.
(b) Assuming the LAV approach, perform query rewriting using the bucket algorithm.
(c) Same as (b) using the MinCon algorithm.

Problem 7.16 (*) Consider relations EMP and ASG of Example 7.18. We denote by |R| the number of pages to store R on disk. Consider the following statistics about the data:

$$|\texttt{EMP}| = 100$$
$$|\texttt{ASG}| = 2\,000$$
$$selectivity(\texttt{ASG.DUR} > 36) = 1\%$$

The mediator generic cost model is

$cost(\sigma_{\texttt{A}=v}(\texttt{R})) = |\texttt{R}|$
$cost(\sigma(X)) = cost(X)$, where X contains at least one operator.
$cost(\texttt{R} \ltimes^{ind}_{\texttt{A}} \texttt{S}) = cost(\texttt{R}) + |\texttt{R}| * cost(\sigma_{\texttt{A}=v}(\texttt{S}))$ using an indexed join algorithm.
$cost(\texttt{R} \ltimes^{nl}_{\texttt{A}} \texttt{S}) = cost(\texttt{R}) + |\texttt{R}| * cost(\texttt{S})$ using a nested loop join algorithm.

Consider the MDBS input query Q:

```
SELECT *
FROM   EMP NATURAL JOIN ASG
WHERE  ASG.DUR>36
```

Consider four plans to process Q:

$$P_1 = \texttt{EMP} \ltimes^{ind}_{\texttt{ENO}} \sigma_{\texttt{DUR}>36}(\texttt{ASG})$$
$$P_2 = \texttt{EMP} \ltimes^{nl}_{\texttt{ENO}} \sigma_{\texttt{DUR}>36}(\texttt{ASG})$$
$$P_3 = \sigma_{\texttt{DUR}>36}(\texttt{ASG}) \ltimes^{ind}_{\texttt{ENO}} \texttt{EMP}$$
$$P_4 = \sigma_{\texttt{DUR}>36}(\texttt{ASG}) \ltimes^{nl}_{\texttt{ENO}} \texttt{EMP}$$

(a) What is the cost of plans P_1 to P_4?
(b) Which plan has the minimal cost?

Problem 7.17 (*) Consider relations EMP and ASG of the previous exercise. Suppose now that the mediator cost model is completed with the following cost information issued from the component DBMSs.

The cost of accessing EMP tuples at db_1 is

$$cost(\sigma_{\texttt{A}=v}(\texttt{R})) = |\sigma_{\texttt{A}=v}(\texttt{R})|$$

The specific cost of selecting ASG tuples that have a given ENO at db_2 is

$$cost(\sigma_{\texttt{ENO}=v}(\texttt{ASG})) = |\sigma_{\texttt{ENO}=v}(\texttt{ASG})|$$

(a) What is the cost of plans P_1 to P_4?
(b) Which plan has the minimal cost?

Problem 7.18 (**) What are the respective advantages and limitations of the query-based and operator-based approaches to heterogeneous query optimization from the points of view of query expressiveness, query performance, development cost of wrappers, system (mediator and wrappers) maintenance, and evolution?

Problem 7.19 (**) Consider Example 7.19 by adding, at a new site, component database db_4 which stores relations `EMP(ENO, ENAME, CITY)` and `ASG(ENO, PNAME, DUR)`. db_4 exports through its wrapper w_3 join and scan capabilities. Let us assume that there can be employees in db_1 with corresponding assignments in db_4 and employees in db_4 with corresponding assignments in db_2.

(a) Define the planning functions of wrapper w_3.
(b) Give the new definition of global view `EMPASG(ENAME, CITY, PNAME, DUR)`.
(c) Give a QEP for the same query as in Example 7.19.