update queries. This decision affects almost all of the distributed DBMS algorithms and control functions.

A non-replicated database (commonly called a *partitioned* database) contains fragments that are allocated to sites, and there is only one copy of any fragment on the network. In case of replication, either the database exists in its entirety at each site (*fully replicated* database), or fragments are distributed to the sites in such a way that copies of a fragment may reside in multiple sites (*partially replicated* database). In the latter the number of copies of a fragment may be an input to the allocation algorithm or a decision variable whose value is determined by the algorithm. Figure 3.6 compares these three replication alternatives with respect to various distributed DBMS functions. We will discuss replication at length in Chapter 13.

| | Full replication | Partial replication | Partitioning |
|---|---|---|---|
| QUERY PROCESSING | Easy | $\leftarrow$ Same | difficulty $\rightarrow$ |
| DIRECTORY MANAGEMENT | Easy or nonexistent | $\leftarrow$ Same | difficulty $\rightarrow$ |
| CONCURRENCY CONTROL | Moderate | Difficult | Easy |
| RELIABILITY | Very high | High | Low |
| REALITY | Possible application | Realistic | Possible application |

**Fig. 3.6** Comparison of Replication Alternatives

### 3.2.6 Information Requirements

One aspect of distribution design is that too many factors contribute to an optimal design. The logical organization of the database, the location of the applications, the access characteristics of the applications to the database, and the properties of the computer systems at each site all have an influence on distribution decisions. This makes it very complicated to formulate the distribution problem.

The information needed for distribution design can be divided into four categories: database information, application information, communication network information, and computer system information. The latter two categories are completely quantitative in nature and are used in allocation models rather than in fragmentation algorithms. We do not consider them in detail here. Instead, the detailed information requirements of the fragmentation and allocation algorithms are discussed in their respective sections.

## 3.3 Fragmentation

In this section we present the various fragmentation strategies and algorithms. As mentioned previously, there are two fundamental fragmentation strategies: horizontal and vertical. Furthermore, there is a possibility of nesting fragments in a hybrid fashion.

### *3.3.1 Horizontal Fragmentation*

As we explained earlier, horizontal fragmentation partitions a relation along its tuples. Thus each fragment has a subset of the tuples of the relation. There are two versions of horizontal partitioning: primary and derived. *Primary horizontal fragmentation* of a relation is performed using predicates that are defined on that relation. *Derived horizontal fragmentation*, on the other hand, is the partitioning of a relation that results from predicates being defined on another relation.

Later in this section we consider an algorithm for performing both of these fragmentations. However, first we investigate the information needed to carry out horizontal fragmentation activity.

#### 3.3.1.1 Information Requirements of Horizontal Fragmentation

**Database Information.**

The database information concerns the global conceptual schema. In this context it is important to note how the database relations are connected to one another, especially with joins. In the relational model, these relationships are also depicted as relations. However, in other data models, such as the entity-relationship (E–R) model [Chen, 1976], these relationships between database objects are depicted explicitly. Ceri et al. [1983] also model the relationship explicitly, within the relational framework, for purposes of the distribution design. In the latter notation, directed *links* are drawn between relations that are related to each other by an equijoin operation.

*Example 3.3.* Figure 3.7 shows the expression of links among the database relations given in Figure 2.3. Note that the direction of the link shows a one-to-many relationship. For example, for each title there are multiple employees with that title; thus there is a link between the PAY and EMP relations. Along the same lines, the many-to-many relationship between the EMP and PROJ relations is expressed with two links to the ASG relation. ♦

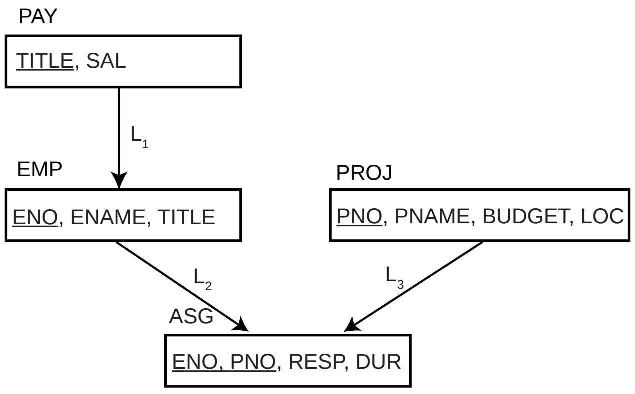


**Fig. 3.7** Expression of Relationships Among Relations Using Links

The links between database objects (i.e., relations in our case) should be quite familiar to those who have dealt with network models of data. In the relational model, they are introduced as join graphs, which we discuss in detail in subsequent chapters on query processing. We introduce them here because they help to simplify the presentation of the distribution models we discuss later.

The relation at the tail of a link is called the *owner* of the link and the relation at the head is called the *member* [Ceri et al., 1983]. More commonly used terms, within the relational framework, are *source* relation for owner and *target* relation for member. Let us define two functions: *owner* and *member*, both of which provide mappings from the set of links to the set of relations. Therefore, given a link, they return the member or owner relations of the link, respectively.

*Example 3.4.* Given link $L_1$ of Figure 3.7, the *owner and member* functions have the following values:

$$owner(L_1) = \text{PAY}$$
$$member(L_1) = \text{EMP}$$

♦

The quantitative information required about the database is the cardinality of each relation $R$, denoted $card(R)$.

## Application Information.

As indicated previously in relation to Figure 3.2, both qualitative and quantitative information is required about applications. The qualitative information guides the fragmentation activity, whereas the quantitative information is incorporated primarily into the allocation models.

The fundamental qualitative information consists of the predicates used in user queries. If it is not possible to analyze all of the user applications to determine these predicates, one should at least investigate the most "important" ones. It has been suggested that as a rule of thumb, the most active 20% of user queries account for 80% of the total data accesses [Wiederhold, 1982]. This "80/20 rule" may be used as a guideline in carrying out this analysis.

At this point we are interested in determining *simple predicates*. Given a relation $R(A_1, A_2, \ldots, A_n)$, where $A_i$ is an attribute defined over domain $D_i$, a simple predicate $p_j$ defined on $R$ has the form

$$p_j : A_i\ \theta\ Value$$

where $\theta \in \{=, <, \neq, \leq, >, \geq\}$ and *Value* is chosen from the domain of $A_i$ ($Value \in D_i$). We use $Pr_i$ to denote the set of all simple predicates defined on a relation $R_i$. The members of $Pr_i$ are denoted by $p_{ij}$.

*Example 3.5.* Given the relation instance PROJ of Figure 3.3,

PNAME = "Maintenance"

is a simple predicate, as well as

BUDGET $\leq$ 200000

♦

Even though simple predicates are quite elegant to deal with, user queries quite often include more complicated predicates, which are Boolean combinations of simple predicates. One combination that we are particularly interested in, called a *minterm predicate*, is the conjunction of simple predicates. Since it is always possible to transform a Boolean expression into conjunctive normal form, the use of minterm predicates in the design algorithms does not cause any loss of generality.

Given a set $Pr_i = \{p_{i1}, p_{i2}, \ldots, p_{im}\}$ of simple predicates for relation $R_i$, the set of minterm predicates $M_i = \{m_{i1}, m_{i2}, \ldots, m_{iz}\}$ is defined as

$$M_i = \{m_{ij} | m_{ij} = \bigwedge_{p_{ik} \in Pr_i} p^*_{ik}\},\ 1 \leq k \leq m,\ 1 \leq j \leq z$$

where $p^*_{ik} = p_{ik}$ or $p^*_{ik} = \neg p_{ik}$. So each simple predicate can occur in a minterm predicate either in its natural form or its negated form.

It is important to note that the negation of a predicate is meaningful for equality predicates of the form $Attribute = Value$. For inequality predicates, the negation should be treated as the complement. For example, the negation of the simple predicate $Attribute \leq Value$ is $Attribute > Value$. Besides theoretical problems of complementation in infinite sets, there is also the practical problem that the complement may be difficult to define. For example, if two simple predicates are defined of the form $Lower\_bound \leq Attribute\_1$, and $Attribute\_1 \leq Upper\_bound$, their complements are $\neg(Lower\_bound \leq Attribute\_1)$ and $\neg(Attribute\_1 \leq Upper\_bound)$. However, the original two simple predicates can be written as $Lower\_bound \leq Attribute\_1 \leq Upper\_bound$ with a complement $\neg(Lower\_bound \leq Attribute\_1 \leq Upper\_bound)$

that may not be easy to define. Therefore, the research in this area typically considers only simple equality predicates [Ceri et al., 1982b; Ceri and Pelagatti, 1984].

*Example 3.6.* Consider relation PAY of Figure 3.3. The following are some of the possible simple predicates that can be defined on PAY.

$p_1$: TITLE = "Elect. Eng."
$p_2$: TITLE = "Syst. Anal."
$p_3$: TITLE = "Mech. Eng."
$p_4$: TITLE = "Programmer"
$p_5$: SAL $\leq$ 30000

The following are *some* of the minterm predicates that can be defined based on these simple predicates.

$m_1$: TITLE = "Elect. Eng." $\wedge$ SAL $\leq$ 30000
$m_2$: TITLE = "Elect. Eng." $\wedge$ SAL $>$ 30000
$m_3$: $\neg$(TITLE = "Elect. Eng.") $\wedge$ SAL $\leq$ 30000
$m_4$: $\neg$(TITLE = "Elect. Eng.") $\wedge$ SAL $>$ 30000
$m_5$: TITLE = "Programmer" $\wedge$ SAL $\leq$ 30000
$m_6$: TITLE = "Programmer" $\wedge$ SAL $>$ 30000

♦

There are a few points to mention here. First, these are not all the minterm predicates that can be defined; we are presenting only a representative sample. Second, some of these may be meaningless given the semantics of relation PAY; we are not addressing that issue here. Third, these are simplified versions of the minterms. The minterm definition requires each predicate to be in a minterm in either its natural or its negated form. Thus, $m_1$, for example, should be written as

$m_1$: TITLE = "Elect. Eng." $\wedge$ TITLE $\neq$ "Syst. Anal." $\wedge$ TITLE $\neq$ "Mech. Eng." $\wedge$ TITLE $\neq$ "Programmer" $\wedge$ SAL $\leq$ 30000

However, clearly this is not necessary, and we use the simplified form. Finally, note that there are logically equivalent expressions to these minterms; for example, $m_3$ can also be rewritten as

$m_3$: TITLE $\neq$ "Elect. Eng." $\wedge$ SAL $\leq$ 30000

In terms of quantitative information about user applications, we need to have two sets of data.

1. *Minterm selectivity*: number of tuples of the relation that would be accessed by a user query specified according to a given minterm predicate. For example, the selectivity of $m_1$ of Example 3.6 is 0 since there are no tuples in PAY that satisfy the minterm predicate. The selectivity of $m_2$, on the other hand, is 0.25

since one of the four tuples in PAY satisfy $m_2$. We denote the selectivity of a minterm $m_i$ as $sel(m_i)$.

2. *Access frequency*: frequency with which user applications access data. If $Q = \{q_1, q_2, \ldots, q_q\}$ is a set of user queries, $acc(q_i)$ indicates the access frequency of query $q_i$ in a given period.

Note that minterm access frequencies can be determined from the query frequencies. We refer to the access frequency of a minterm $m_i$ as $acc(m_i)$.

#### 3.3.1.2 Primary Horizontal Fragmentation

Before we present a formal algorithm for horizontal fragmentation, we intuitively discuss the process for primary (and derived) horizontal fragmentation. A *primary horizontal fragmentation* is defined by a selection operation on the owner relations of a database schema. Therefore, given relation $R$, its horizontal fragments are given by

$$R_i = \sigma_{F_i}(R),\ 1 \leq i \leq w$$

where $F_i$ is the selection formula used to obtain fragment $R_i$ (also called the *fragmentation predicate*). Note that if $F_i$ is in conjunctive normal form, it is a minterm predicate ($m_i$). The algorithm we discuss will, in fact, insist that $F_i$ be a minterm predicate.

*Example 3.7.* The decomposition of relation PROJ into horizontal fragments $\text{PROJ}_1$ and $\text{PROJ}_2$ in Example 3.1 is defined as follows[1]:

$$\text{PROJ}_1 = \sigma_{\text{BUDGET} \leq 200000}\ (\text{PROJ})$$
$$\text{PROJ}_2 = \sigma_{\text{BUDGET} > 200000}\ (\text{PROJ})$$

♦

Example 3.7 demonstrates one of the problems of horizontal partitioning. If the domain of the attributes participating in the selection formulas are continuous and infinite, as in Example 3.7, it is quite difficult to define the set of formulas $F = \{F_1, F_2, \ldots, F_n\}$ that would fragment the relation properly. One possible course of action is to define ranges as we have done in Example 3.7. However, there is always the problem of handling the two endpoints. For example, if a new tuple with a BUDGET value of, say, \$600,000 were to be inserted into PROJ, one would have had to review the fragmentation to decide if the new tuple is to go into $\text{PROJ}_2$ or if the fragments need to be revised and a new fragment needs to be defined as

[1] We assume that the non-negativity of the BUDGET values is a feature of the relation that is enforced by an integrity constraint. Otherwise, a simple predicate of the form $0 \leq$ BUDGET also needs to be included in $Pr$. We assume this to be true in all our examples and discussions in this chapter.

$$\text{PROJ}_2 = \sigma_{200000<\text{BUDGET} \leq 400000}\ (\text{PROJ})$$
$$\text{PROJ}_3 = \sigma_{\text{BUDGET} > 400000}\ (\text{PROJ})$$

*Example 3.8.* Consider relation PROJ of Figure 3.3. We can define the following horizontal fragments based on the project location. The resulting fragments are shown in Figure 3.8.

$$\text{PROJ}_1 = \sigma_{\text{LOC=“Montreal”}}\ (\text{PROJ})$$
$$\text{PROJ}_2 = \sigma_{\text{LOC=“New York”}}\ (\text{PROJ})$$
$$\text{PROJ}_3 = \sigma_{\text{LOC=“Paris”}}\ (\text{PROJ})$$

♦

$\text{PROJ}_1$

| PNO | PNAME | BUDGET | LOC |
|---|---|---|---|
| P1 | Instrumentation | 150000 | Montreal |

$\text{PROJ}_2$

| PNO | PNAME | BUDGET | LOC |
|---|---|---|---|
| P2 | Database Develop. | 135000 | New York |
| P3 | CAD/CAM | 250000 | New York |

$\text{PROJ}_3$

| PNO | PNAME | BUDGET | LOC |
|---|---|---|---|
| P4 | Maintenance | 310000 | Paris |

**Fig. 3.8** Primary Horizontal Fragmentation of Relation PROJ

Now we can define a horizontal fragment more carefully. A horizontal fragment $R_i$ of relation $R$ consists of all the tuples of $R$ that satisfy a minterm predicate $m_i$. Hence, given a set of minterm predicates $M$, there are as many horizontal fragments of relation $R$ as there are minterm predicates. This set of horizontal fragments is also commonly referred to as the set of *minterm fragments*.

From the foregoing discussion it is obvious that the definition of the horizontal fragments depends on minterm predicates. Therefore, the first step of any fragmentation algorithm is to determine a set of simple predicates that will form the minterm predicates.

An important aspect of simple predicates is their *completeness*; another is their *minimality*. A set of simple predicates $Pr$ is said to be *complete* if and only if there is an equal probability of access by every application to any tuple belonging to any minterm fragment that is defined according to $Pr$[2].

*Example 3.9.* Consider the fragmentation of relation PROJ given in Example 3.8. If the only application that accesses PROJ wants to access the tuples according to the location, the set is complete since each tuple of each fragment $\text{PROJ}_i$ (Example 3.8) has the same probability of being accessed. If, however, there is a second application which accesses only those project tuples where the budget is less than or equal to \$200,000, then $Pr$ is not complete. Some of the tuples within each $\text{PROJ}_i$ have a higher probability of being accessed due to this second application. To make the set of predicates complete, we need to add (BUDGET $\leq$ 200000, BUDGET $>$ 200000) to $Pr$:

$$\begin{aligned} Pr = \{&\text{LOC=“Montreal”}, \text{LOC=“New York”}, \text{LOC=“Paris”},\\ &\text{BUDGET} \leq 200000, \text{BUDGET} > 200000\} \end{aligned}$$

♦

The reason completeness is a desirable property is because fragments obtained according to a complete set of predicates are logically uniform since they all satisfy the minterm predicate. They are also statistically homogeneous in the way applications access them. These characteristics ensure that the resulting fragmentation results in a balanced load (with respect to the given workload) across all the fragments. Therefore, we will use a complete set of predicates as the basis of primary horizontal fragmentation.

It is possible to define completeness more formally so that a complete set of predicates can be obtained automatically. However, this would require the designer to specify the access probabilities for *each* tuple of a relation for *each* application under consideration. This is considerably more work than appealing to the common sense and experience of the designer to come up with a complete set. Shortly, we will present an algorithmic way of obtaining this set.

The second desirable property of the set of predicates, according to which minterm predicates and, in turn, fragments are to be defined, is minimality, which is very intuitive. It simply states that if a predicate influences how fragmentation is performed (i.e., causes a fragment $f$ to be further fragmented into, say, $f_i$ and $f_j$), there should be at least one application that accesses $f_i$ and $f_j$ differently. In other words, the simple predicate should be *relevant* in determining a fragmentation. If all the predicates of a set $Pr$ are relevant, $Pr$ is *minimal*.

A formal definition of relevance can be given as follows [Ceri et al., 1982b]. Let $m_i$ and $m_j$ be two minterm predicates that are identical in their definition, except that $m_i$ contains the simple predicate $p_i$ in its natural form while $m_j$ contains $\neg p_i$. Also, let $f_i$ and $f_j$ be two fragments defined according to $m_i$ and $m_j$, respectively. Then $p_i$ is *relevant* if and only if

[2] It is clear that the definition of completeness of a set of simple predicates is different from the completeness rule of fragmentation given in Section 3.2.4.

$$\frac{acc(m_i)}{card(f_i)} \neq \frac{acc(m_j)}{card(f_j)}$$

*Example 3.10.* The set *Pr* defined in Example 3.9 is complete and minimal. If, however, we were to add the predicate

PNAME = “Instrumentation”

to *Pr*, the resulting set would not be minimal since the new predicate is not relevant with respect to *Pr* – there is no application that would access the resulting fragments any differently. ♦

We can now present an iterative algorithm that would generate a complete and minimal set of predicates $Pr'$ given a set of simple predicates *Pr*. This algorithm, called COM_MIN, is given in Algorithm 3.1. To avoid lengthy wording, we have adopted the following notation:

*Rule 1*: each fragment is accessed differently by at least one application.’

$f_i$ *of* $Pr'$: fragment $f_i$ defined according to a minterm predicate defined over the predicates of $Pr'$.

**Algorithm 3.1**: COM_MIN Algorithm

```
Input: R: relation; Pr: set of simple predicates
Output: Pr': set of simple predicates
Declare: F: set of minterm fragments
begin
    find $p_i \in Pr$ such that $p_i$ partitions $R$ according to Rule 1 ;
    $Pr' \leftarrow p_i$ ;
    $Pr \leftarrow Pr - p_i$ ;
    $F \leftarrow f_i$                {$f_i$ is the minterm fragment according to $p_i$} ;
    repeat
        find a $p_j \in Pr$ such that $p_j$ partitions some $f_k$ of $Pr'$ according to Rule 1
        ;
        $Pr' \leftarrow Pr' \cup p_j$ ;
        $Pr \leftarrow Pr - p_j$ ;
        $F \leftarrow F \cup f_j$ ;
        if $\exists p_k \in Pr'$ which is not relevant then
            $Pr' \leftarrow Pr' - p_k$ ;
            $F \leftarrow F - f_k$ ;
    until $Pr'$ is complete ;
end
```

The algorithm begins by finding a predicate that is relevant and that partitions the input relation. The **repeat-until** loop iteratively adds predicates to this set, ensuring minimality at each step. Therefore, at the end the set $Pr'$ is both minimal and complete.

The second step in the primary horizontal design process is to derive the set of minterm predicates that can be defined on the predicates in set $Pr'$. These minterm predicates determine the fragments that are used as candidates in the allocation step. Determination of individual minterm predicates is trivial; the difficulty is that the set of minterm predicates may be quite large (in fact, exponential on the number of simple predicates). We look at ways of reducing the number of minterm predicates that need to be considered in fragmentation.

This reduction can be achieved by eliminating some of the minterm fragments that may be meaningless. This elimination is performed by identifying those minterms that might be contradictory to a set of implications $I$. For example, if $Pr' = \{p_1, p_2\}$, where

$p_1 : att = value\_1$
$p_2 : att = value\_2$

and the domain of $att$ is $\{value\_1, value\_2\}$, it is obvious that $I$ contains two implications:

$i_1 : (att = value\_1) \Rightarrow \neg(att = value\_2)$
$i_2 : \neg(att = value_1) \Rightarrow (att = value\_2)$

The following four minterm predicates are defined according to $Pr'$:

$m_1 : (att = value\_1) \wedge (att = value\_2)$
$m_2 : (att = value\_1) \wedge \neg(att = value\_2)$
$m_3 : \neg(att = value\_1) \wedge (att = value\_2)$
$m_4 : \neg(att = value\_1) \wedge \neg(att = value\_2)$

In this case the minterm predicates $m_1$ and $m_4$ are contradictory to the implications $I$ and can therefore be eliminated from $M$.

The algorithm for primary horizontal fragmentation is given in Algorithm 3.2. The input to the algorithm PHORIZONTAL is a relation $R$ that is subject to primary horizontal fragmentation, and $Pr$, which is the set of simple predicates that have been determined according to applications defined on relation $R$.

*Example 3.11.* We now consider the design of the database scheme given in Figure 3.7. The first thing to note is that there are two relations that are the subject of primary horizontal fragmentation: PAY and PROJ.

Suppose that there is only one application that accesses PAY, which checks the salary information and determines a raise accordingly. Assume that employee records are managed in two places, one handling the records of those with salaries less than

**Algorithm 3.2**: PHORIZONTAL Algorithm

```
Input: $R$: relation; $Pr$: set of simple predicates
Output: $M$: set of minterm fragments
begin
    $Pr' \leftarrow$COM_MIN($R, Pr$) ;
    determine the set $M$ of minterm predicates ;
    determine the set $I$ of implications among $p_i \in Pr'$ ;
    foreach $m_i \in M$ do
        if $m_i$ is contradictory according to $I$ then
            $M \leftarrow M - m_i$
end
```

or equal to \$30,000, and the other handling the records of those who earn more than \$30,000. Therefore, the query is issued at two sites.

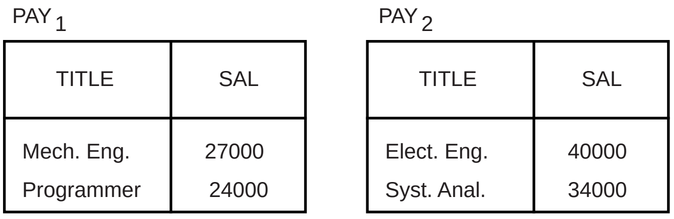
The simple predicates that would be used to partition relation PAY are

$p_1$: SAL $\leq$ 30000
$p_2$: SAL $>$ 30000

thus giving the initial set of simple predicates $Pr = \{p_1, p_2\}$. Applying the COM_MIN algorithm with $i = 1$ as initial value results in $Pr' = \{p_1\}$. This is complete and minimal since $p_2$ would not partition $f_1$ (which is the minterm fragment formed with respect to $p_1$) according to Rule 1. We can form the following minterm predicates as members of $M$:

$m_1$: (SAL $<$ 30000)
$m_2$: $\neg$(SAL $\leq$ 30000) $=$ SAL $>$ 30000

Therefore, we define two fragments $F_s = \{S_1, S_2\}$ according to $M$ (Figure 3.9).



PAY$_1$

| TITLE | SAL |
|---|---|
| Mech. Eng. | 27000 |
| Programmer | 24000 |

PAY$_2$

| TITLE | SAL |
|---|---|
| Elect. Eng. | 40000 |
| Syst. Anal. | 34000 |

**Fig. 3.9** Horizontal Fragmentation of Relation PAY

Let us next consider relation PROJ. Assume that there are two applications. The first is issued at three sites and finds the names and budgets of projects given their location. In SQL notation, the query is

```
SELECT PNAME, BUDGET
FROM   PROJ
WHERE  LOC=Value
```

For this application, the simple predicates that would be used are the following:

$p_1$: LOC = “Montreal”
$p_2$: LOC = “New York”
$p_3$: LOC = “Paris”

The second application is issued at two sites and has to do with the management of the projects. Those projects that have a budget of less than or equal to $200,000 are managed at one site, whereas those with larger budgets are managed at a second site. Thus, the simple predicates that should be used to fragment according to the second application are

$p_4$: BUDGET $\leq$ 200000
$p_5$: BUDGET $>$ 200000

If the COM_MIN algorithm is followed, the set $Pr' = \{p_1, p_2, p_4\}$ is obviously complete and minimal. Actually COM_MIN would add any two of $p_1, p_2, p_3$ to $Pr'$; in this example we have selected to include $p_1, p_2$.

Based on $Pr'$, the following six minterm predicates that form $M$ can be defined:

$m_1$: (LOC = “Montreal”) $\wedge$ (BUDGET $\leq$ 200000)
$m_2$: (LOC = “Montreal”) $\wedge$ (BUDGET $>$ 200000)
$m_3$: (LOC = “New York”) $\wedge$ (BUDGET $\leq$ 200000)
$m_4$: (LOC = “New York”) $\wedge$ (BUDGET $>$ 200000)
$m_5$: (LOC = “Paris”) $\wedge$ (BUDGET $\leq$ 200000)
$m_6$: (LOC = “Paris”) $\wedge$ (BUDGET $>$ 200000)

As noted in Example 3.6, these are not the only minterm predicates that can be generated. It is, for example, possible to specify predicates of the form

$$p_1 \wedge p_2 \wedge p_3 \wedge p_4 \wedge p_5$$

However, the obvious implications

$$i_1 : p_1 \Rightarrow \neg p_2 \wedge \neg p_3$$
$$i_2 : p_2 \Rightarrow \neg p_1 \wedge \neg p_3$$
$$i_3 : p_3 \Rightarrow \neg p_1 \wedge \neg p_2$$
$$i_4 : p_4 \Rightarrow \neg p_5$$
$$i_5 : p_5 \Rightarrow \neg p_4$$
$$i_6 : \neg p_4 \Rightarrow p_5$$
$$i_7 : \neg p_5 \Rightarrow p_4$$

eliminate these minterm predicates and we are left with $m_1$ to $m_6$.

Looking at the database instance in Figure 3.3, one may be tempted to claim that the following implications hold:

$i_8$: LOC = "Montreal" $\Rightarrow \neg$ (BUDGET $>$ 200000)
$i_9$: LOC = "Paris" $\Rightarrow \neg$ (BUDGET $\leq$ 200000)
$i_{10}$: $\neg$ (LOC = "Montreal") $\Rightarrow$ BUDGET $\leq$ 200000
$i_{11}$: $\neg$ (LOC = "Paris") $\Rightarrow$ BUDGET $>$ 200000

However, remember that implications should be defined according to the semantics of the database, not according to the current values. There is nothing in the database semantics that suggest that the implications $i_8$ through $i_{11}$ hold. Some of the fragments defined according to $M = \{m_1, \ldots, m_6\}$ may be empty, but they are, nevertheless, fragments.

The result of the primary horizontal fragmentation of PROJ is to form six fragments $F_{PROJ} = \{$PROJ$_1$, PROJ$_2$, PROJ$_3$, PROJ$_4$, PROJ$_5$, PROJ$_6\}$ of relation PROJ according to the minterm predicates $M$ (Figure 3.10). Since fragments PROJ$_2$, and PROJ$_5$ are empty, they are not depicted in Figure 3.10. ♦

PROJ$_1$

| PNO | PNAME | BUDGET | LOC |
|---|---|---|---|
| P1 | Instrumentation | 150000 | Montreal |

PROJ$_3$

| PNO | PNAME | BUDGET | LOC |
|---|---|---|---|
| P2 | Database Develop. | 135000 | New York |

PROJ$_4$

| PNO | PNAME | BUDGET | LOC |
|---|---|---|---|
| P3 | CAD/CAM | 250000 | New York |

PROJ$_6$

| PNO | PNAME | BUDGET | LOC |
|---|---|---|---|
| P4 | Maintenance | 310000 | Paris |

**Fig. 3.10** Horizontal Partitioning of Relation PROJ

### 3.3.1.3 Derived Horizontal Fragmentation

A derived horizontal fragmentation is defined on a member relation of a link according to a selection operation specified on its owner. It is important to remember two points. First, the link between the owner and the member relations is defined as an equi-join. Second, an equi-join can be implemented by means of semijoins. This second point is especially important for our purposes, since we want to partition a member relation according to the fragmentation of its owner, but we also want the resulting fragment to be defined *only* on the attributes of the member relation.

Accordingly, given a link $L$ where $owner(L) = S$ and $member(L) = R$, the derived horizontal fragments of $R$ are defined as

$$R_i = R \ltimes S_i, 1 \leq i \leq w$$

where $w$ is the maximum number of fragments that will be defined on $R$, and $S_i = \sigma_{F_i}(S)$, where $F_i$ is the formula according to which the primary horizontal fragment $S_i$ is defined.

*Example 3.12.* Consider link $L_1$ in Figure 3.7, where $owner(L_1)$ = PAY and $member(L_1)$ = EMP. Then we can group engineers into two groups according to their salary: those making less than or equal to $30,000, and those making more than $30,000. The two fragments $\text{EMP}_1$ and $\text{EMP}_2$ are defined as follows:

$$\text{EMP}_1 = \text{EMP} \ltimes \text{PAY}_1$$
$$\text{EMP}_2 = \text{EMP} \ltimes \text{PAY}_2$$

where

$$\text{PAY}_1 = \sigma_{\text{SAL} \leq 30000}(\text{PAY})$$
$$\text{PAY}_2 = \sigma_{\text{SAL} > 30000}(\text{PAY})$$

The result of this fragmentation is depicted in Figure 3.11. ♦

$\text{EMP}_1$

| ENO | ENAME | TITLE |
|---|---|---|
| E3 | A. Lee | Mech. Eng. |
| E4 | J. Miller | Programmer |
| E7 | R. Davis | Mech. Eng. |

$\text{EMP}_2$

| ENO | ENAME | TITLE |
|---|---|---|
| E1 | J. Doe | Elect. Eng. |
| E2 | M. Smith | Syst. Anal. |
| E5 | B. Casey | Syst. Anal. |
| E6 | L. Chu | Elect. Eng. |
| E8 | J. Jones | Syst. Anal. |

**Fig. 3.11** Derived Horizontal Fragmentation of Relation EMP

To carry out a derived horizontal fragmentation, three inputs are needed: the set of partitions of the owner relation (e.g., $\text{PAY}_1$ and $\text{PAY}_2$ in Example 3.12), the member relation, and the set of semijoin predicates between the owner and the member (e.g., EMP.TITLE = PAY.TITLE in Example 3.12). The fragmentation algorithm, then, is quite trivial, so we will not present it in any detail.

There is one potential complication that deserves some attention. In a database schema, it is common that there are more than two links into a relation $R$ (e.g., in Figure 3.7, ASG has two incoming links). In this case there is more than one possible derived horizontal fragmentation of $R$. The choice of candidate fragmentation is based on two criteria:

1. The fragmentation with better join characteristics
2. The fragmentation used in more applications

Let us discuss the second criterion first. This is quite straightforward if we take into consideration the frequency with which applications access some data. If possible, one should try to facilitate the accesses of the "heavy" users so that their total impact on system performance is minimized.

Applying the first criterion, however, is not that straightforward. Consider, for example, the fragmentation we discussed in Example 3.1. The effect (and the objective) of this fragmentation is that the join of the EMP and PAY relations to answer the query is assisted (1) by performing it on smaller relations (i.e., fragments), and (2) by potentially performing joins in parallel.

The first point is obvious. The fragments of EMP are smaller than EMP itself. Therefore, it will be faster to join any fragment of PAY with any fragment of EMP than to work with the relations themselves. The second point, however, is more important and is at the heart of distributed databases. If, besides executing a number of queries at different sites, we can parallelize execution of one join query, the response time or throughput of the system can be expected to improve. In the case of joins, this is possible under certain circumstances. Consider, for example, the join graph (i.e., the links) between the fragments of EMP and PAY derived in Example 3.10 (Figure 3.12). There is only one link coming in or going out of a fragment. Such a join graph is called a *simple* graph. The advantage of a design where the join relationship between fragments is simple is that the member and owner of a link can be allocated to one site and the joins between different pairs of fragments can proceed independently and in parallel.

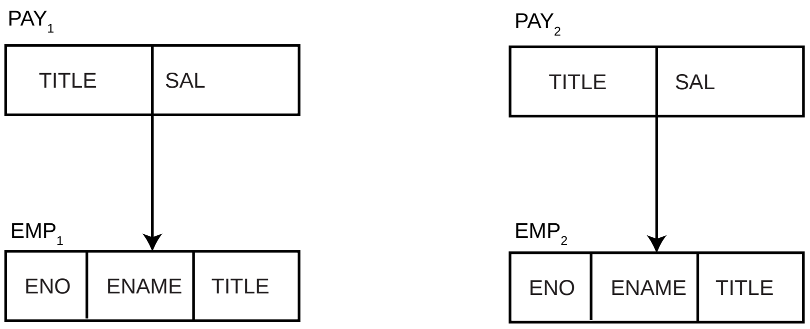


**Fig. 3.12** Join Graph Between Fragments

Unfortunately, obtaining simple join graphs may not always be possible. In that case, the next desirable alternative is to have a design that results in a *partitioned* join graph. A partitioned graph consists of two or more subgraphs with no links between them. Fragments so obtained may not be distributed for parallel execution as easily as those obtained via simple join graphs, but the allocation is still possible.

*Example 3.13.* Let us continue with the distribution design of the database we started in Example 3.11. We already decided on the fragmentation of relation EMP according to the fragmentation of PAY (Example 3.12). Let us now consider ASG. Assume that there are the following two applications:

1. The first application finds the names of engineers who work at certain places. It runs on all three sites and accesses the information about the engineers who work on local projects with higher probability than those of projects at other locations.
2. At each administrative site where employee records are managed, users would like to access the responsibilities on the projects that these employees work on and learn how long they will work on those projects.

The first application results in a fragmentation of ASG according to the (non-empty) fragments $PROJ_1$, $PROJ_3$, $PROJ_4$ and $PROJ_6$ of PROJ obtained in Example 3.11. Remember that

$PROJ_1$: $\sigma_{LOC=\text{"Montreal"} \wedge BUDGET \leq 200000}$ (PROJ)
$PROJ_3$: $\sigma_{LOC=\text{"New York"} \wedge BUDGET \leq 200000}$ (PROJ)
$PROJ_4$: $\sigma_{LOC=\text{"New York"} \wedge BUDGET > 200000}$ (PROJ)
$PROJ_6$: $\sigma_{LOC=\text{"Paris"} \wedge BUDGET > 200000}$ (PROJ)

Therefore, the derived fragmentation of ASG according to $\{PROJ_1, PROJ_2, PROJ_3\}$ is defined as follows:

$ASG_1 = ASG \ltimes PROJ_1$
$ASG_2 = ASG \ltimes PROJ_3$
$ASG_3 = ASG \ltimes PROJ_4$
$ASG_4 = ASG \ltimes PROJ_6$

These fragment instances are shown in Figure 3.13.

The second query can be specified in SQL as

```
SELECT RESP, DUR
FROM   ASG, EMPi
WHERE  ASG.ENO = EMPi.ENO
```

where $i = 1$ or $i = 2$, depending on the site where the query is issued. The derived fragmentation of ASG according to the fragmentation of EMP is defined below and depicted in Figure 3.14.

$ASG_1 = ASG \ltimes EMP_1$
$ASG_2 = ASG \ltimes EMP_2$

♦

$ASG_1$

| ENO | PNO | RESP | DUR |
|---|---|---|---|
| E1 | P1 | Manager | 12 |
| E2 | P1 | Analyst | 24 |

$ASG_2$

| ENO | PNO | RESP | DUR |
|---|---|---|---|
| E2 | P2 | Analyst | 6 |
| E4 | P2 | Programmer | 18 |
| E5 | P2 | Manager | 24 |

$ASG_3$

| ENO | PNO | RESP | DUR |
|---|---|---|---|
| E3 | P3 | Consultant | 10 |
| E7 | P3 | Engineer | 36 |
| E8 | P3 | Manager | 40 |

$ASG_4$

| ENO | PNO | RESP | DUR |
|---|---|---|---|
| E3 | P4 | Engineer | 48 |
| E6 | P4 | Manager | 48 |

**Fig. 3.13** Derived Fragmentation of ASG with respect to PROJ

$ASG_1$

| ENO | PNO | RESP | DUR |
|---|---|---|---|
| E3 | P3 | Consultant | 10 |
| E3 | P4 | Engineer | 48 |
| E4 | P2 | Programmer | 18 |
| E7 | P3 | Engineer | 36 |

$ASG_2$

| ENO | PNO | RESP | DUR |
|---|---|---|---|
| E1 | P1 | Manager | 12 |
| E2 | P1 | Analyst | 24 |
| E2 | P2 | Analyst | 6 |
| E5 | P2 | Manager | 24 |
| E6 | P4 | Manager | 48 |
| E8 | P3 | Manager | 40 |

**Fig. 3.14** Derived Fragmentation of ASG with respect to EMP

This example demonstrates two things:

1. Derived fragmentation may follow a chain where one relation is fragmented as a result of another one's design and it, in turn, causes the fragmentation of another relation (e.g., the chain PAY$\rightarrow$EMP$\rightarrow$ASG).
2. Typically, there will be more than one candidate fragmentation for a relation (e.g., relation ASG). The final choice of the fragmentation scheme may be a decision problem addressed during allocation.

#### 3.3.1.4 Checking for Correctness

We should now check the fragmentation algorithms discussed so far with respect to the three correctness criteria presented in Section 3.2.4.

**Completeness.**

The completeness of a primary horizontal fragmentation is based on the selection predicates used. As long as the selection predicates are complete, the resulting fragmentation is guaranteed to be complete as well. Since the basis of the fragmentation algorithm is a set of *complete* and *minimal* predicates, $Pr'$, completeness is guaranteed as long as no mistakes are made in defining $Pr'$.

The completeness of a derived horizontal fragmentation is somewhat more difficult to define. The difficulty is due to the fact that the predicate determining the fragmentation involves two relations. Let us first define the completeness rule formally and then look at an example.

Let $R$ be the member relation of a link whose owner is relation $S$, where $R$ and $S$ are fragmented as $F_R = \{R_1, R_2, \ldots, R_w\}$ and $F_S = \{S_1, S_2, \ldots, S_w\}$, respectively. Furthermore, let $A$ be the join attribute between $R$ and $S$. Then for each tuple $t$ of $R_i$, there should be a tuple $t'$ of $S_i$ such that $t[A] = t'[A]$.

For example, there should be no ASG tuple which has a project number that is not also contained in PROJ. Similarly, there should be no EMP tuples with TITLE values where the same TITLE value does not appear in PAY as well. This rule is known as *referential integrity* and ensures that the tuples of any fragment of the member relation are also in the owner relation.

**Reconstruction.**

Reconstruction of a global relation from its fragments is performed by the union operator in both the primary and the derived horizontal fragmentation. Thus, for a relation $R$ with fragmentation $F_R = \{R_1, R_2, \ldots, R_w\}$,

$$R = \bigcup R_i, \quad \forall R_i \in F_R$$

**Disjointness.**

It is easier to establish disjointness of fragmentation for primary than for derived horizontal fragmentation. In the former case, disjointness is guaranteed as long as the minterm predicates determining the fragmentation are mutually exclusive.

In derived fragmentation, however, there is a semijoin involved that adds considerable complexity. Disjointness can be guaranteed if the join graph is simple. Otherwise, it is necessary to investigate actual tuple values. In general, we do not want a tuple of a member relation to join with two or more tuples of the owner relation when these tuples are in different fragments of the owner. This may not be very easy to establish, and illustrates why derived fragmentation schemes that generate a simple join graph are always desirable.

*Example 3.14.* In fragmenting relation PAY (Example 3.11), the minterm predicates $M = \{m_1, m_2\}$ were

$m_1$: SAL $\leq$ 30000
$m_2$: SAL $>$ 30000

Since $m_1$ and $m_2$ are mutually exclusive, the fragmentation of PAY is disjoint. For relation EMP, however, we require that

**1.** Each engineer has a single title.

**2.** Each title have a single salary value associated with it.

Since these two rules follow from the semantics of the database, the fragmentation of EMP with respect to PAY is also disjoint. ♦

## 3.3.2 Vertical Fragmentation

Remember that a vertical fragmentation of a relation $R$ produces fragments $R_1, R_2, \ldots, R_r$, each of which contains a subset of $R$'s attributes as well as the primary key of $R$. The objective of vertical fragmentation is to partition a relation into a set of smaller relations so that many of the user applications will run on only one fragment. In this context, an "optimal" fragmentation is one that produces a fragmentation scheme which minimizes the execution time of user applications that run on these fragments.

Vertical fragmentation has been investigated within the context of centralized database systems as well as distributed ones. Its motivation within the centralized context is as a design tool, which allows the user queries to deal with smaller relations, thus causing a smaller number of page accesses [Navathe et al., 1984]. It has also been suggested that the most "active" subrelations can be identified and placed in a faster memory subsystem in those cases where memory hierarchies are supported [Eisner and Severance, 1976].

Vertical partitioning is inherently more complicated than horizontal partitioning. This is due to the total number of alternatives that are available. For example, in horizontal partitioning, if the total number of simple predicates in $Pr$ is $n$, there are $2^n$ possible minterm predicates that can be defined on it. In addition, we know that some of these will contradict the existing implications, further reducing the candidate fragments that need to be considered. In the case of vertical partitioning, however, if a relation has $m$ non-primary key attributes, the number of possible fragments is equal to $B(m)$, which is the $m$th Bell number [Niamir, 1978]. For large values of

$m, B(m) \approx m^m$; for example, for $m$=10, $B(m) \approx$ 115,000, for $m$=15, $B(m) \approx 10^9$, for $m$=30, $B(m) = 10^{23}$ [Hammer and Niamir, 1979; Navathe et al., 1984].

These values indicate that it is futile to attempt to obtain optimal solutions to the vertical partitioning problem; one has to resort to heuristics. Two types of heuristic approaches exist for the vertical fragmentation of global relations:

1. *Grouping:* starts by assigning each attribute to one fragment, and at each step, joins some of the fragments until some criteria is satisfied. Grouping was first suggested for centralized databases [Hammer and Niamir, 1979], and was used later for distributed databases [Sacca and Wiederhold, 1985].
2. *Splitting:* starts with a relation and decides on beneficial partitionings based on the access behavior of applications to the attributes. The technique was also first discussed for centralized database design [Hoffer and Severance, 1975]. It was then extended to the distributed environment [Navathe et al., 1984].

In what follows we discuss only the splitting technique, since it fits more naturally within the top-down design methodology, since the "optimal" solution is probably closer to the full relation than to a set of fragments each of which consists of a single attribute [Navathe et al., 1984]. Furthermore, splitting generates non-overlapping fragments whereas grouping typically results in overlapping fragments. We prefer non-overlapping fragments for disjointness. Of course, non-overlapping refers only to non-primary key attributes.

Before we proceed, let us clarify an issue that we only mentioned in Example 3.2, namely, the replication of the global relation's key in the fragments. This is a characteristic of vertical fragmentation that allows the reconstruction of the global relation. Therefore, splitting is considered only for those attributes that do not participate in the primary key.

There is a strong advantage to replicating the key attributes despite the obvious problems it causes. This advantage has to do with semantic integrity enforcement, to be discussed in Chapter 5. Note that the dependencies briefly discussed in Section 2.1 is, in fact, a constraint that has to hold among the attribute values of the respective relations at all times. Remember also that most of these dependencies involve the key attributes of a relation. If we now design the database so that the key attributes are part of one fragment that is allocated to one site, and the implied attributes are part of another fragment that is allocated to a second site, every update request that causes an integrity check will necessitate communication among sites. Replication of the key attributes at each fragment reduces the chances of this occurring but does not eliminate it completely, since such communication may be necessary due to integrity constraints that do not involve the primary key, as well as due to concurrency control.

One alternative to the replication of the key attributes is the use of *tuple identifiers* (TIDs), which are system-assigned unique values to the tuples of a relation. Since TIDs are maintained by the system, the fragments are disjoint at a logical level.

#### 3.3.2.1 Information Requirements of Vertical Fragmentation

The major information required for vertical fragmentation is related to applications. The following discussion, therefore, is exclusively focused on what needs to be determined about applications that will run against the distributed database. Since vertical partitioning places in one fragment those attributes usually accessed together, there is a need for some measure that would define more precisely the notion of "togetherness." This measure is the *affinity* of attributes, which indicates how closely related the attributes are. Unfortunately, it is not realistic to expect the designer or the users to be able to easily specify these values. We now present one way by which they can be obtained from more primitive data.

The major information requirement related to applications is their access frequencies. Let $Q = \{q_1, q_2, \ldots, q_q\}$ be the set of user queries (applications) that access relation $R(A_1, A_2, \ldots, A_n)$. Then, for each query $q_i$ and each attribute $A_j$, we associate an *attribute usage value*, denoted as $use(q_i, A_j)$, and defined as follows:

$$use(q_i, A_j) = \begin{cases} 1 \text{ if attribute } A_j \text{ is referenced by query } q_i \\ 0 \text{ otherwise} \end{cases}$$

The $use(q_i, \bullet)$ vectors for each application are easy to define if the designer knows the applications that will run on the database. Again, remember that the 80-20 rule discussed earlier should be helpful in this task.

*Example 3.15.* Consider relation PROJ of Figure 3.3. Assume that the following applications are defined to run on this relation. In each case we also give the SQL specification.

$q_1$: Find the budget of a project, given its identification number.

```
SELECT BUDGET
FROM   PROJ
WHERE  PNO=Value
```

$q_2$: Find the names and budgets of all projects.

```
SELECT  PNAME, BUDGET
FROM    PROJ
```

$q_3$: Find the names of projects located at a given city.

```
SELECT PNAME
FROM   PROJ
WHERE  LOC=Value
```

$q_4$: Find the total project budgets for each city.

```
SELECT SUM(BUDGET)
FROM   PROJ
WHERE  LOC=Value
```

According to these four applications, the attribute usage values can be defined. As a notational convenience, we let $A_1$ = PNO, $A_2$ = PNAME, $A_3$ = BUDGET, and $A_4$ = LOC. The usage values are defined in matrix form (Figure 3.15), where entry $(i, j)$ denotes $use(q_i, A_j)$. ♦

$$\begin{array}{c|cccc|} & A_1 & A_2 & A_3 & A_4 \\ \hline q_1 & 1 & 0 & 1 & 0 \\ q_2 & 0 & 1 & 1 & 0 \\ q_3 & 0 & 1 & 0 & 1 \\ q_4 & 0 & 0 & 1 & 1 \\ \hline \end{array}$$

**Fig. 3.15** Example Attribute Usage Matrix

Attribute usage values are not sufficiently general to form the basis of attribute splitting and fragmentation. This is because these values do not represent the weight of application frequencies. The frequency measure can be included in the definition of the attribute affinity measure $aff(A_i,A_j)$, which measures the bond between two attributes of a relation according to how they are accessed by applications.

The attribute affinity measure between two attributes $A_i$ and $A_j$ of a relation $R(A_1,A_2, \ldots,A_n)$ with respect to the set of applications $Q = \{q_1,q_2,\ldots,q_q\}$ is defined as

$$aff(A_i,A_j) = \sum_{k|use(q_k,A_i)=1 \wedge use(q_k,A_j)=1} \sum_{\forall S_l} ref_l(q_k)acc_l(q_k)$$

where $ref_l(q_k)$ is the number of accesses to attributes $(A_i,A_j)$ for each execution of application $q_k$ at site $S_l$ and $acc_l(q_k)$ is the application access frequency measure previously defined and modified to include frequencies at different sites.

The result of this computation is an $n \times n$ matrix, each element of which is one of the measures defined above. We call this matrix the *attribute affinity matrix* (*AA*).

*Example 3.16.* Let us continue with the case that we examined in Example 3.15. For simplicity, let us assume that $ref_l(q_k)$ = 1 for all $q_k$ and $S_l$. If the application frequencies are

$$\begin{array}{lll} acc_1(q_1) = 15 & acc_2(q_1) = 20 & acc_3(q_1) = 10 \\ acc_1(q_2) = 5 & acc_2(q_2) = 0 & acc_3(q_2) = 0 \\ acc_1(q_3) = 25 & acc_2(q_3) = 25 & acc_3(q_3) = 25 \\ acc_1(q_4) = 3 & acc_2(q_4) = 0 & acc_3(q_4) = 0 \end{array}$$

then the affinity measure between attributes $A_1$ and $A_3$ can be measured as

$$aff(A_1,A_3) = \sum_{k=1}^{1} \sum_{l=1}^{3} acc_l(q_k) = acc_1(q_1) + acc_2(q_1) + acc_3(q_1) = 45$$

since the only application that accesses both of the attributes is $q_1$. The complete attribute affinity matrix is shown in Figure 3.16. Note that the diagonal values are not computed since they are meaningless. ♦

$$\begin{array}{c|cccc|} & A_1 & A_2 & A_3 & A_4 \\ A_1 & - & 0 & 45 & 0 \\ A_2 & 0 & - & 5 & 75 \\ A_3 & 45 & 5 & - & 3 \\ A_4 & 0 & 75 & 3 & - \end{array}$$

**Fig. 3.16** Attribute Affinity Matrix

The attribute affinity matrix will be used in the rest of this chapter to guide the fragmentation effort. The process involves first clustering together the attributes with high affinity for each other, and then splitting the relation accordingly.

#### 3.3.2.2 Clustering Algorithm

The fundamental task in designing a vertical fragmentation algorithm is to find some means of grouping the attributes of a relation based on the attribute affinity values in *AA*. It has been suggested that the bond energy algorithm (BEA) [McCormick et al., 1972] should be used for this purpose ([Hoffer and Severance, 1975] and [Navathe et al., 1984]). It is considered appropriate for the following reasons [Hoffer and Severance, 1975]:

1. It is designed specifically to determine groups of similar items as opposed to, say, a linear ordering of the items (i.e., it clusters the attributes with larger affinity values together, and the ones with smaller values together).
2. The final groupings are insensitive to the order in which items are presented to the algorithm.
3. The computation time of the algorithm is reasonable: $O(n^2)$, where $n$ is the number of attributes.
4. Secondary interrelationships between clustered attribute groups are identifiable.

The bond energy algorithm takes as input the attribute affinity matrix, permutes its rows and columns, and generates a *clustered affinity matrix* (*CA*). The permutation is done in such a way as to *maximize* the following *global affinity measure* (*AM*):

$$AM = \sum_{i=1}^{n}\sum_{j=1}^{n} aff(A_i,A_j)[aff(A_i,A_{j-1}) + aff(A_i,A_{j+1}) \\ + aff(A_{i-1},A_j) + aff(A_{i+1},A_j)]$$

where

$$aff(A_0,A_j) = aff(A_i,A_0) = aff(A_{n+1},A_j) = aff(A_i,A_{n+1}) = 0$$

The last set of conditions takes care of the cases where an attribute is being placed in *CA* to the left of the leftmost attribute or to the right of the rightmost attribute during column permutations, and prior to the topmost row and following the last row during row permutations. In these cases, we take 0 to be the *aff* values between the attribute being considered for placement and its left or right (top or bottom) neighbors, which do not exist in *CA*.

The maximization function considers the nearest neighbors only, thereby resulting in the grouping of large values with large ones, and small values with small ones. Also, the attribute affinity matrix (*AA*) is symmetric, which reduces the objective function of the formulation above to

$$AM = \sum_{i=1}^{n}\sum_{j=1}^{n} aff(A_i,A_j)[aff(A_i,A_{j-1}) + aff(A_i,A_{j+1})]$$

The details of the bond energy algorithm are given in Algorithm 3.3. Generation of the clustered affinity matrix (*CA*) is done in three steps:

1. *Initialization.* Place and fix one of the columns of *AA* arbitrarily into *CA*. Column 1 was chosen in the algorithm.
2. *Iteration.* Pick each of the remaining $n-i$ columns (where $i$ is the number of columns already placed in *CA*) and try to place them in the remaining $i+1$ positions in the *CA* matrix. Choose the placement that makes the greatest contribution to the global affinity measure described above. Continue this step until no more columns remain to be placed.
3. *Row ordering.* Once the column ordering is determined, the placement of the rows should also be changed so that their relative positions match the relative positions of the columns.[3]

[3] From now on, we may refer to elements of the *AA* and *CA* matrices as $AA(i,j)$ and $CA(i,j)$, respectively. This is done for notational convenience only. The mapping to the affinity measures is $AA(i,j) = aff(A_i,A_j)$ and $CA(i,j) = aff($attribute placed at column $i$ in *CA*, attribute placed at column $j$ in *CA*$)$. Even though *AA* and *CA* matrices are identical except for the ordering of attributes, since the algorithm orders all the *CA* columns before it orders the rows, the affinity measure of *CA* is specified with respect to columns. Note that the endpoint condition for the calculation of the affinity measure (*AM*) can be specified, using this notation, as $CA(0,j) = CA(i,0) = CA(n+1,j) = CA(i,n+1) = 0$.

**Algorithm 3.3**: BEA Algorithm

```
Input: $AA$: attribute affinity matrix
Output: $CA$: clustered affinity matrix
begin
    {initialize; remember that $AA$ is an $n \times n$ matrix}
    $CA(\bullet, 1) \leftarrow AA(\bullet, 1)$ ;
    $CA(\bullet, 2) \leftarrow AA(\bullet, 2)$ ;
    $index \leftarrow 3$ ;
    while $index \le n$ do          {choose the "best" location for attribute $AA_{index}$}
        for $i$ from 1 to $index - 1$ by 1 do calculate $cont(A_{i-1}, A_{index}, A_i)$ ;
        calculate $cont(A_{index-1}, A_{index}, A_{index+1})$ ;          {boundary condition}
        $loc \leftarrow$ placement given by maximum $cont$ value ;
        for $j$ from $index$ to $loc$ by $-1$ do
            $CA(\bullet, j) \leftarrow CA(\bullet, j-1)$          {shuffle the two matrices}
        $CA(\bullet, loc) \leftarrow AA(\bullet, index)$ ;
        $index \leftarrow index + 1$
    order the rows according to the relative ordering of columns
end
```

For the second step of the algorithm to work, we need to define what is meant by the contribution of an attribute to the affinity measure. This contribution can be derived as follows. Recall that the global affinity measure *AM* was previously defined as

$$AM = \sum_{i=1}^{n} \sum_{j=1}^{n} aff(A_i, A_j)[aff(A_i, A_{j-1}) + aff(A_i, A_{j+1})]$$

which can be rewritten as

$$\begin{aligned} AM &= \sum_{i=1}^{n} \sum_{j=1}^{n} [aff(A_i, A_j)aff(A_i, A_{j-1}) + aff(A_i, A_j)aff(A_i, A_{j+1})] \\ &= \sum_{j=1}^{n} \left[ \sum_{i=1}^{n} aff(A_i, A_j)aff(A_i, A_{j-1}) + \sum_{i=1}^{n} aff(A_i, A_j)aff(A_i, A_{j+1}) \right] \end{aligned}$$

Let us define the *bond* between two attributes $A_x$ and $A_y$ as

$$bond(A_x, A_y) = \sum_{z=1}^{n} aff(A_z, A_x)aff(A_z, A_y)$$

Then *AM* can be written as

$$AM = \sum_{j=1}^{n} [bond(A_j, A_{j-1}) + bond(A_j, A_{j+1})]$$

Now consider the following $n$ attributes

$$\underbrace{A_1\ A_2\ \ldots\ A_{i-1}}_{AM'}\ A_i\ A_j\ \underbrace{A_{j+1}\ \ldots\ A_n}_{AM''}$$

The global affinity measure for these attributes can be written as

$$\begin{aligned}
AM_{old} &= AM' + AM'' \\
&\quad + bond(A_{i-1},A_i) + bond(A_i,A_j) + bond(A_j,A_i) + bond(A_j,A_{j+1}) \\
&= \sum_{l=1}^{i}[bond(A_l,A_{l-1}) + bond(A_l,A_{l+1})] \\
&\quad + \sum_{l=i+2}^{n}[bond(A_l,A_{l-1}) + bond(A_l,A_{l+1})] \\
&\quad + 2bond(A_i,A_j)
\end{aligned}$$

Now consider placing a new attribute $A_k$ between attributes $A_i$ and $A_j$ in the clustered affinity matrix. The new global affinity measure can be similarly written as

$$\begin{aligned}
AM_{new} &= AM' + AM'' + bond(A_i,A_k) + bond(A_k,A_i) \\
&\quad + bond(A_k,A_j) + bond(A_j,A_k) \\
&= AM' + AM'' + 2bond(A_i,A_k) + 2bond(A_k,A_j)
\end{aligned}$$

Thus, the net *contribution*[4] to the global affinity measure of placing attribute $A_k$ between $A_i$ and $A_j$ is

$$\begin{aligned}
cont(A_i,A_k,A_j) &= AM_{new} - AM_{old} \\
&= 2bond(A_i,A_k) + 2bond(A_k,A_j) - 2bond(A_i,A_j)
\end{aligned}$$

*Example 3.17.* Let us consider the *AA* matrix given in Figure 3.16 and study the contribution of moving attribute $A_4$ between attributes $A_1$ and $A_2$, given by the formula

$$cont(A_1,A_4,A_2) = 2bond(A_1,A_4) + 2bond(A_4,A_2) - 2bond(A_1,A_2)$$

Computing each term, we get

$$\begin{aligned}
bond(A_1,A_4) &= 45*0+0*75+45*3+0*78 = 135 \\
bond(A_4,A_2) &= 11865 \\
bond(A_1,A_2) &= 225
\end{aligned}$$

Therefore,

[4] In literature [Hoffer and Severance, 1975] this measure is specified as $bond(A_i,A_k) + bond(A_k,A_j) - 2bond(A_i,A_j)$. However, this is a pessimistic measure which does not follow from the definition of *AM*.

$$cont(A_1,A_4,A_2) = 2*135+2*11865-2*225 = 23550$$

♦

Note that the calculation of the bond between two attributes requires the multiplication of the respective elements of the two columns representing these attributes and taking the row-wise sum.

The algorithm and our discussion so far have both concentrated on the columns of the attribute affinity matrix. We can also make the same arguments and redesign the algorithm to operate on the rows. Since the $AA$ matrix is symmetric, both of these approaches will generate the same result.

Another point about Algorithm 3.3 is that to improve its efficiency, the second column is also fixed and placed next to the first one during the initialization step. This is acceptable since, according to the algorithm, $A_2$ can be placed either to the left of $A_1$ or to its right. The bond between the two, however, is independent of their positions relative to one another.

Finally, we should indicate the problem of computing *cont* at the endpoints. If an attribute $A_i$ is being considered for placement to the left of the leftmost attribute, one of the bond equations to be calculated is between a non-existent left element and $A_k$ [i.e., $bond(A_0,A_k)$]. Thus we need to refer to the conditions imposed on the definition of the global affinity measure $AM$, where $CA(0,k) = 0$. The other extreme is if $A_j$ is the rightmost attribute that is already placed in the $CA$ matrix and we are checking for the contribution of placing attribute $A_k$ to the right of $A_j$. In this case the $bond(k,k+1)$ needs to be calculated. However, since no attribute is yet placed in column $k+1$ of $CA$, the affinity measure is not defined. Therefore, according to the endpoint conditions, this *bond* value is also 0.

*Example 3.18.* We consider the clustering of the PROJ relation attributes and use the attribute affinity matrix $AA$ of Figure 3.16.

According to the initialization step, we copy columns 1 and 2 of the $AA$ matrix to the $CA$ matrix (Figure 3.17a) and start with column 3 (i.e., attribute $A_3$). There are three alternative places where column 3 can be placed: to the left of column 1, resulting in the ordering (3-1-2), in between columns 1 and 2, giving (1-3-2), and to the right of 2, resulting in (1-2-3). Note that to compute the contribution of the last ordering we have to compute $cont(A_2,A_3,A_4)$ rather than $cont(A_1,A_2,A_3)$. Furthermore, in this context $A_4$ refers to the fourth index position in the $CA$ matrix, which is empty (Figure 3.17b), not to the attribute column $A_4$ of the $AA$ matrix. Let us calculate the contribution to the global affinity measure of each alternative.

Ordering (0-3-1):

$$cont(A_0,A_3,A_1) = 2bond(A_0,A_3) + 2bond(A_3,A_1) - 2bond(A_0,A_1)$$

We know that

$$bond(A_0,A_1) = bond(A_0,A_3) = 0$$
$$bond(A_3,A_1) = 45*45+5*0+53*45+3*0 = 4410$$

$$\begin{array}{c|cccc|}
 & A_1 & A_2 & & \\
\hline
A_1 & 45 & 0 & & \\
A_2 & 0 & 80 & & \\
A_3 & 45 & 5 & & \\
A_4 & 0 & 75 & & \\
\hline
\end{array}$$

(a)

$$\begin{array}{c|cccc|}
 & A_1 & A_3 & A_2 & \\
\hline
A_1 & 45 & 45 & 0 & \\
A_2 & 0 & 5 & 80 & \\
A_3 & 45 & 53 & 5 & \\
A_4 & 0 & 3 & 75 & \\
\hline
\end{array}$$

(b)

$$\begin{array}{c|cccc|}
 & A_1 & A_3 & A_2 & A_4 \\
\hline
A_1 & 45 & 45 & 0 & 0 \\
A_2 & 0 & 5 & 80 & 75 \\
A_3 & 45 & 53 & 5 & 3 \\
A_4 & 0 & 3 & 75 & 78 \\
\hline
\end{array}$$

(c)

$$\begin{array}{c|cccc|}
 & A_1 & A_3 & A_2 & A_4 \\
\hline
A_1 & 45 & 45 & 0 & 0 \\
A_3 & 45 & 53 & 5 & 3 \\
A_2 & 0 & 5 & 80 & 75 \\
A_4 & 0 & 3 & 75 & 78 \\
\hline
\end{array}$$

(d)

**Fig. 3.17** Calculation of the Clustered Affinity (CA) Matrix

Thus

$$cont(A_0, A_3, A_1) = 8820$$

Ordering (1-3-2):

$$\begin{aligned}
cont(A_1, A_3, A_2) &= 2bond(A_1, A_3) + 2bond(A_3, A_2) - 2bond(A_1, A_2)\\
bond(A_1, A_3) &= bond(A_3, A_1) = 4410\\
bond(A_3, A_2) &= 890\\
bond(A_1, A_2) &= 225
\end{aligned}$$

Thus

$$cont(A_1, A_3, A_2) = 10150$$

Ordering (2-3-4):

$$\begin{aligned}
cont(A_2, A_3, A_4) &= 2bond(A_2, A_3) + 2bond(A_3, A_4) - 2bond(A_2, A_4)\\
bond(A_2, A_3) &= 890\\
bond(A_3, A_4) &= 0\\
bond(A_2, A_4) &= 0
\end{aligned}$$

Thus

$$cont(A_2,A_3,A_4) = 1780$$

Since the contribution of the ordering (1-3-2) is the largest, we select to place $A_3$ to the right of $A_1$ (Figure 3.17b). Similar calculations for $A_4$ indicate that it should be placed to the right of $A_2$ (Figure 3.17c).

Finally, the rows are organized in the same order as the columns and the result is shown in Figure 3.17d. ♦

In Figure 3.17d we see the creation of two clusters: one is in the upper left corner and contains the smaller affinity values and the other is in the lower right corner and contains the larger affinity values. This clustering indicates how the attributes of relation PROJ should be split. However, in general the border for this split may not be this clear-cut. When the *CA* matrix is big, usually more than two clusters are formed and there are more than one candidate partitionings. Thus, there is a need to approach this problem more systematically.

#### 3.3.2.3 Partitioning Algorithm

The objective of the splitting activity is to find sets of attributes that are accessed solely, or for the most part, by distinct sets of applications. For example, if it is possible to identify two attributes, $A_1$ and $A_2$, which are accessed only by application $q_1$, and attributes $A_3$ and $A_4$, which are accessed by, say, two applications $q_2$ and $q_3$, it would be quite straightforward to decide on the fragments. The task lies in finding an algorithmic method of identifying these groups.

Consider the clustered attribute matrix of Figure 3.18. If a point along the diagonal is fixed, two sets of attributes are identified. One set $\{A_1,A_2,\ldots,A_i\}$ is at the upper left-hand corner and the second set $\{A_{i+1},\ldots,A_n\}$ is to the right and to the bottom of this point. We call the former set *top* and the latter set *bottom* and denote the attribute sets as *TA* and *BA*, respectively.

We now turn to the set of applications $Q = \{q_1,q_2,\ldots,q_q\}$ and define the set of applications that access only *TA*, only *BA*, or both. These sets are defined as follows:

$$\begin{aligned} AQ(q_i) &= \{A_j | use(q_i,A_j) = 1\} \\ TQ &= \{q_i | AQ(q_i) \subseteq TA\} \\ BQ &= \{q_i | AQ(q_i) \subseteq BA\} \\ OQ &= Q - \{TQ \cup BQ\} \end{aligned}$$

The first of these equations defines the set of attributes accessed by application $q_i$; $TQ$ and $BQ$ are the sets of applications that only access *TA* or *BA*, respectively, and $OQ$ is the set of applications that access both.

There is an optimization problem here. If there are $n$ attributes of a relation, there are $n-1$ possible positions where the dividing point can be placed along the diagonal

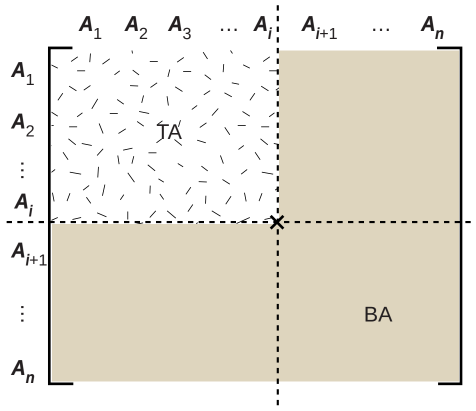


**Fig. 3.18** Locating a Splitting Point

of the clustered attribute matrix for that relation. The best position for division is one which produces the sets $TQ$ and $BQ$ such that the total accesses to *only one* fragment are maximized while the total accesses to *both* fragments are minimized. We therefore define the following cost equations:

$$CQ = \sum_{q_i \in Q} \sum_{\forall S_j} ref_j(q_i) acc_j(q_i)$$

$$CTQ = \sum_{q_i \in TQ} \sum_{\forall S_j} ref_j(q_i) acc_j(q_i)$$

$$CBQ = \sum_{q_i \in BQ} \sum_{\forall S_j} ref_j(q_i) acc_j(q_i)$$

$$COQ = \sum_{q_i \in OQ} \sum_{\forall S_j} ref_j(q_i) acc_j(q_i)$$

Each of the equations above counts the total number of accesses to attributes by applications in their respective classes. Based on these measures, the optimization problem is defined as finding the point $x$ $(1 \leq x \leq n)$ such that the expression

$$z = CTQ * CBQ - COQ^2$$

is maximized [Navathe et al., 1984]. The important feature of this expression is that it defines two fragments such that the values of $CTQ$ and $CBQ$ are as nearly equal as possible. This enables the balancing of processing loads when the fragments are distributed to various sites. It is clear that the partitioning algorithm has linear complexity in terms of the number of attributes of the relation, that is, $O(n)$.

There are two complications that need to be addressed. The first is with respect to the splitting. The procedure splits the set of attributes two-way. For larger sets of attributes, it is quite likely that $m$-way partitioning may be necessary.

Designing an $m$-way partitioning is possible but computationally expensive. Along the diagonal of the $CA$ matrix, it is necessary to try $1, 2, \ldots, m-1$ split points, and for each of these, it is necessary to check which place maximizes $z$. Thus, the complexity of such an algorithm is $O(2^m)$. Of course, the definition of $z$ has to be modified for those cases where there are multiple split points. The alternative solution is to recursively apply the binary partitioning algorithm to each of the fragments obtained during the previous iteration. One would compute $TQ$, $BQ$, and $OQ$, as well as the associated access measures for each of the fragments, and partition them further.

The second complication relates to the location of the block of attributes that should form one fragment. Our discussion so far assumed that the split point is unique and single and divides the $CA$ matrix into an upper left-hand partition and a second partition formed by the rest of the attributes. The partition, however, may also be formed in the middle of the matrix. In this case, we need to modify the algorithm slightly. The leftmost column of the $CA$ matrix is shifted to become the rightmost column and the topmost row is shifted to the bottom. The shift operation is followed by checking the $n-1$ diagonal positions to find the maximum $z$. The idea behind shifting is to move the block of attributes that should form a cluster to the topmost left corner of the matrix, where it can easily be identified. With the addition of the shift operation, the complexity of the partitioning algorithm increases by a factor of $n$ and becomes $O(n^2)$.

Assuming that a shift procedure, called SHIFT, has already been implemented, the partitioning algorithm is given in Algorithm 3.4. The input of the PARTITION is the clustered affinity matrix $CA$, the relation $R$ to be fragmented, and the attribute usage and access frequency matrices. The output is a set of fragments $F_R = \{R_1, R_2\}$, where $R_i \subseteq \{A_1, A_2 \ldots, A_n\}$ and $R_1 \cap R_2$ = the key attributes of relation $R$. Note that for $n$-way partitioning, this routine should either be invoked iteratively, or implemented as a recursive procedure.

*Example 3.19.* When the PARTITION algorithm is applied to the $CA$ matrix obtained for relation PROJ (Example 3.18), the result is the definition of fragments $F_{PROJ} = \{\text{PROJ}_1, \text{PROJ}_2\}$, where $\text{PROJ}_1 = \{A_1, A_3\}$ and $\text{PROJ}_2 = \{A_1, A_2, A_4\}$. Thus

$$\begin{aligned}\text{PROJ}_1 &= \{\text{PNO, BUDGET}\}\\ \text{PROJ}_2 &= \{\text{PNO, PNAME, LOC}\}\end{aligned}$$

Note that in this exercise we performed the fragmentation over the entire set of attributes rather than only on the non-key ones. The reason for this is the simplicity of the example. For that reason, we included PNO, which is the key of PROJ in $\text{PROJ}_2$ as well as in $\text{PROJ}_1$. ♦

### 3.3.2.4 Checking for Correctness

We follow arguments similar to those of horizontal partitioning to prove that the PARTITION algorithm yields a correct vertical fragmentation.

**Algorithm 3.4**: PARTITION Algorithm

```
Input: $CA$: clustered affinity matrix; $R$: relation; $ref$: attribute usage matrix;
       $acc$: access frequency matrix
Output: $F$: set of fragments
begin
    {determine the $z$ value for the first column}
    {the subscripts in the cost equations indicate the split point}
    calculate $CTQ_{n-1}$ ;
    calculate $CBQ_{n-1}$ ;
    calculate $COQ_{n-1}$ ;
    $best \leftarrow CTQ_{n-1} * CBQ_{n-1} - (COQ_{n-1})^2$ ;
    repeat
        {determine the best partitioning}
        for $i$ from $n-2$ to $1$ by $-1$ do
            calculate $CTQ_i$ ;
            calculate $CBQ_i$ ;
            calculate $COQ_i$ ;
            $z \leftarrow CTQ * CBQ_i - COQ_i^2$ ;
            if $z > best$ then $best \leftarrow z$        {record the split point within shift}
        call SHIFT($CA$)
    until no more SHIFT is possible ;
    reconstruct the matrix according to the shift position ;
    $R_1 \leftarrow \Pi_{TA}(R) \cup K$ ;        {$K$ is the set of primary key attributes of $R$}
    $R_2 \leftarrow \Pi_{BA}(R) \cup K$ ;
    $F \leftarrow \{R_1, R_2\}$
end
```

**Completeness.**

Completeness is guaranteed by the PARTITION algorithm since each attribute of the global relation is assigned to one of the fragments. As long as the set of attributes $A$ over which the relation $R$ is defined consists of

$$A = \bigcup R_i$$

completeness of vertical fragmentation is ensured.

**Reconstruction.**

We have already mentioned that the reconstruction of the original global relation is made possible by the join operation. Thus, for a relation $R$ with vertical fragmentation $F_R = \{R_1, R_2, \ldots, R_r\}$ and key attribute(s) $K$,

$$R = \bowtie_K R_i, \forall R_i \in F_R$$

Therefore, as long as each $R_i$ is complete, the join operation will properly reconstruct $R$. Another important point is that either each $R_i$ should contain the key attribute(s) of $R$, or it should contain the system assigned tuple IDs (TIDs).

**Disjointness.**

As we indicated before, the disjointness of fragments is not as important in vertical fragmentation as it is in horizontal fragmentation. There are two cases here:

1. TIDs are used, in which case the fragments are disjoint since the TIDs that are replicated in each fragment are system assigned and managed entities, totally invisible to the users.
2. The key attributes are replicated in each fragment, in which case one cannot claim that they are disjoint in the strict sense of the term. However, it is important to realize that this duplication of the key attributes is known and managed by the system and does not have the same implications as tuple duplication in horizontally partitioned fragments. In other words, as long as the fragments are disjoint except for the key attributes, we can be satisfied and call them disjoint.

## 3.3.3 *Hybrid Fragmentation*

In most cases a simple horizontal or vertical fragmentation of a database schema will not be sufficient to satisfy the requirements of user applications. In this case a vertical fragmentation may be followed by a horizontal one, or vice versa, producing a tree-structured partitioning (Figure 3.19). Since the two types of partitioning strategies are applied one after the other, this alternative is called *hybrid* fragmentation. It has also been named *mixed* fragmentation or *nested* fragmentation.

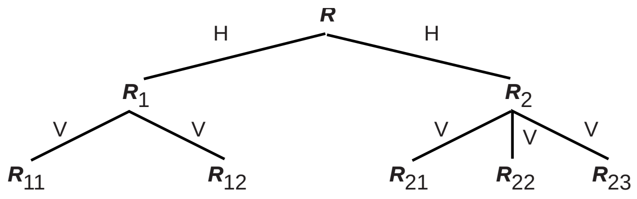


**Fig. 3.19** Hybrid Fragmentation

A good example for the necessity of hybrid fragmentation is relation PROJ, which we have been working with. In Example 3.11 we partitioned it into six horizontal fragments based on two applications. In Example 3.19 we partitioned the same relation vertically into two. What we have, therefore, is a set of horizontal fragments, each of which is further partitioned into two vertical fragments.

The number of levels of nesting can be large, but it is certainly finite. In the case of horizontal fragmentation, one has to stop when each fragment consists of only one tuple, whereas the termination point for vertical fragmentation is one attribute per fragment. These limits are quite academic, however, since the levels of nesting in most practical applications do not exceed 2. This is due to the fact that normalized global relations already have small degrees and one cannot perform too many vertical fragmentations before the cost of joins becomes very high.

We will not discuss in detail the correctness rules and conditions for hybrid fragmentation, since they follow naturally from those for vertical and horizontal fragmentations. For example, to reconstruct the original global relation in case of hybrid fragmentation, one starts at the leaves of the partitioning tree and moves upward by performing joins and unions (Figure 3.20). The fragmentation is complete if the intermediate and leaf fragments are complete. Similarly, disjointness is guaranteed if intermediate and leaf fragments are disjoint.

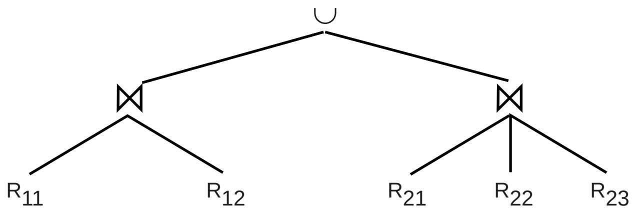


**Fig. 3.20** Reconstruction of Hybrid Fragmentation

## 3.4 Allocation

The allocation of resources across the nodes of a computer network is an old problem that has been studied extensively. Most of this work, however, does not address the problem of distributed database design, but rather that of placing individual files on a computer network. We will examine the differences between the two shortly. We first need to define the allocation problem more precisely.

### *3.4.1 Allocation Problem*

Assume that there are a set of fragments $F = \{F_1, F_2, \ldots, F_n\}$ and a distributed system consisting of sites $S = \{S_1, S_2, \ldots, S_m\}$ on which a set of applications $Q = \{q_1, q_2, \ldots, q_q\}$ is running. The allocation problem involves finding the "optimal" distribution of $F$ to $S$.

The optimality can be defined with respect to two measures [Dowdy and Foster, 1982]:

1. *Minimal cost.* The cost function consists of the cost of storing each $F_i$ at a site $S_j$, the cost of querying $F_i$ at site $S_j$, the cost of updating $F_i$ at all sites where it is stored, and the cost of data communication. The allocation problem, then, attempts to find an allocation scheme that minimizes a combined cost function.
2. *Performance.* The allocation strategy is designed to maintain a performance metric. Two well-known ones are to minimize the response time and to maximize the system throughput at each site.

Most of the models that have been proposed to date make this distinction of optimality. However, if one really examines the problem in depth, it is apparent that the "optimality" measure should include both the performance and the cost factors. In other words, one should be looking for an allocation scheme that, for example, answers user queries in minimal time while keeping the cost of processing minimal. A similar statement can be made for throughput maximization. One can then ask why such models have not been developed. The answer is quite simple: complexity.

Let us consider a *very* simple formulation of the problem. Let $F$ and $S$ be defined as before. For the time being, we consider only a single fragment, $F_k$. We make a number of assumptions and definitions that will enable us to model the allocation problem.

1. Assume that $Q$ can be modified so that it is possible to identify the update and the retrieval-only queries, and to define the following for a *single* fragment $F_k$:

   $$T = \{t_1, t_2, \ldots, t_m\}$$

   where $t_i$ is the read-only traffic generated at site $S_i$ for $F_k$, and

   $$U = \{u_1, u_2, \ldots, u_m\}$$

   where $u_i$ is the update traffic generated at site $S_i$ for $F_k$.
2. Assume that the communication cost between any two pair of sites $S_i$ and $S_j$ is fixed for a unit of transmission. Furthermore, assume that it is different for updates and retrievals in order that the following can be defined:

$$C(T) = \{c_{12}, c_{13}, \ldots, c_{1m}, \ldots, c_{m-1,m}\}$$
$$C'(U) = \{c'_{12}, c'_{13}, \ldots, c'_{1m}, \ldots, c'_{m-1,m}\}$$

where $c_{ij}$ is the unit communication cost for retrieval requests between sites $S_i$ and $S_j$, and $c'_{ij}$ is the unit communication cost for update requests between sites $S_i$ and $S_j$.

**3.** Let the cost of storing the fragment at site $S_i$ be $d_i$. Thus we can define $D = \{d_1, d_2, \ldots, d_m\}$ for the storage cost of fragment $F_k$ at all the sites.

**4.** Assume that there are no capacity constraints for either the sites or the communication links.

Then the allocation problem can be specified as a cost-minimization problem where we are trying to find the set $I \subseteq S$ that specifies where the copies of the fragment will be stored. In the following, $x_j$ denotes the decision variable for the placement such that

$$x_j = \begin{cases} 1 \text{ if fragment } F_k \text{ is assigned to site } S_j \\ 0 \text{ otherwise} \end{cases}$$

The precise specification is as follows:

$$\min \left[ \sum_{i=1}^{m} \left( \sum_{j|S_j \in I} x_j u_j c'_{ij} + t_j \min_{j|S_j \in I} c_{ij} \right) + \sum_{j|S_j \in I} x_j d_j \right]$$

subject to

$$x_j = 0 \; or \; 1$$

The second term of the objective function calculates the total cost of storing all the duplicate copies of the fragment. The first term, on the other hand, corresponds to the cost of transmitting the updates to all the sites that hold the replicas of the fragment, and to the cost of executing the retrieval-only requests at the site, which will result in minimal data transmission cost.

This is a very simplistic formulation that is not suitable for distributed database design. But even if it were, there is another problem. This formulation, which comes from Casey [1972], has been proven to be NP-complete [Eswaran, 1974]. Various different formulations of the problem have been proven to be just as hard over the years (e.g., [Sacca and Wiederhold, 1985] and [Lam and Yu, 1980]). The implication is, of course, that for large problems (i.e., large number of fragments and sites), obtaining optimal solutions is probably not computationally feasible. Considerable research has therefore been devoted to finding good heuristics that may provide suboptimal solutions.

There are a number of reasons why simplistic formulations such as the one we have discussed are not suitable for distributed database design. These are inherent in all the early file allocation models for computer networks.

1. One cannot treat fragments as individual files that can be allocated one at a time, in isolation. The placement of one fragment usually has an impact on the placement decisions about the other fragments which are accessed together since the access costs to the remaining fragments may change (e.g., due to distributed join). Therefore, the relationship between fragments should be taken into account.
2. The access to data by applications is modeled very simply. A user request is issued at one site and all the data to answer it is transferred to that site. In distributed database systems, access to data is more complicated than this simple "remote file access" model suggests. Therefore, the relationship between the allocation and query processing should be properly modeled.
3. These models do not take into consideration the cost of integrity enforcement, yet locating two fragments involved in the same integrity constraint at two different sites can be costly.
4. Similarly, the cost of enforcing concurrency control mechanisms should be considered [Rothnie and Goodman, 1977].

In summary, let us remember the interrelationship between the distributed database problems as depicted in Figure 1.7. Since the allocation is so central, its relationship with algorithms that are implemented for other problem areas needs to be represented in the allocation model. However, this is exactly what makes it quite difficult to solve these models. To separate the traditional problem of file allocation from the fragment allocation in distributed database design, we refer to the former as the *file allocation problem* (FAP) and to the latter as the *database allocation problem* (DAP).

There are no general heuristic models that take as input a set of fragments and produce a near-optimal allocation subject to the types of constraints discussed here. The models developed to date make a number of simplifying assumptions and are applicable to certain specific formulations. Therefore, instead of presenting one or more of these allocation algorithms, we present a relatively general model and then discuss a number of possible heuristics that might be employed to solve it.

### 3.4.2 Information Requirements

It is at the allocation stage that we need the quantitative data about the database, the applications that run on it, the communication network, the processing capabilities, and storage limitations of each site on the network. We will discuss each of these in detail.

#### 3.4.2.1 Database Information

To perform horizontal fragmentation, we defined the selectivity of minterms. We now need to extend that definition to fragments, and define the selectivity of a fragment $F_j$ with respect to query $q_i$. This is the number of tuples of $F_j$ that need to be accessed in order to process $q_i$. This value will be denoted as $sel_i(F_j)$.

Another piece of necessary information on the database fragments is their size. The size of a fragment $F_j$ is given by

$$size(F_j) = card(F_j) * length(F_j)$$

where $length(F_j)$ is the length (in bytes) of a tuple of fragment $F_j$.

#### 3.4.2.2 Application Information

Most of the application-related information is already compiled during the fragmentation activity, but a few more are required by the allocation model. The two important measures are the number of read accesses that a query $q_i$ makes to a fragment $F_j$ during its execution (denoted as $RR_{ij}$), and its counterpart for the update accesses ($UR_{ij}$). These may, for example, count the number of block accesses required by the query.

We also need to define two matrices $UM$ and $RM$, with elements $u_{ij}$ and $r_{ij}$, respectively, which are specified as follows:

$$u_{ij} = \begin{cases} 1 \text{ if query } q_i \text{ updates fragment } F_j \\ 0 \text{ otherwise} \end{cases}$$

$$r_{ij} = \begin{cases} 1 \text{ if query } q_i \text{ retrieves from fragment } F_j \\ 0 \text{ otherwise} \end{cases}$$

A vector $O$ of values $o(i)$ is also defined, where $o(i)$ specifies the originating site of query $q_i$. Finally, to define the response-time constraint, the maximum allowable response time of each application should be specified.

#### 3.4.2.3 Site Information

For each computer site, we need to know its storage and processing capacity. Obviously, these values can be computed by means of elaborate functions or by simple estimates. The unit cost of storing data at site $S_k$ will be denoted as $USC_k$. There is also a need to specify a cost measure $LPC_k$ as the cost of processing one unit of work at site $S_k$. The work unit should be identical to that of the $RR$ and $UR$ measures.

#### 3.4.2.4 Network Information

In our model we assume the existence of a simple network where the cost of communication is defined in terms of one frame of data. Thus $g_{ij}$ denotes the communication cost per frame between sites $S_i$ and $S_j$. To enable the calculation of the number of messages, we use $fsize$ as the size (in bytes) of one frame. There is no question that there are more elaborate network models which take into consideration the channel capacities, distances between sites, protocol overhead, and so on. However, the derivation of those equations is beyond the scope of this chapter.

## *3.4.3 Allocation Model*

We discuss an allocation model that attempts to minimize the total cost of processing and storage while trying to meet certain response time restrictions. The model we use has the following form:

min(Total Cost)

subject to

response-time constraint
storage constraint
processing constraint

In the remainder of this section we expand the components of this model based on the information requirements discussed in Section 3.4.2. The decision variable is $x_{ij}$, which is defined as

$$x_{ij} = \begin{cases} 1 \text{ if the fragment } F_i \text{ is stored at site } S_j \\ 0 \text{ otherwise} \end{cases}$$

#### 3.4.3.1 Total Cost

The total cost function has two components: query processing and storage. Thus it can be expressed as

$$TOC = \sum_{\forall q_i \in Q} QPC_i + \sum_{\forall S_k \in S} \sum_{\forall F_j \in F} STC_{jk}$$

where $QPC_i$ is the query processing cost of application $q_i$, and $STC_{jk}$ is the cost of storing fragment $F_j$ at site $S_k$.

Let us consider the storage cost first. It is simply given by

$$STC_{jk} = USC_k * size(F_j) * x_{jk}$$

and the two summations find the total storage costs at all the sites for all the fragments.

The query processing cost is more difficult to specify. Most models of the file allocation problem (FAP) separate it into two components: the retrieval-only processing cost, and the update processing cost. We choose a different approach in our model of the database allocation problem (DAP) and specify it as consisting of the processing cost ($PC$) and the transmission cost ($TC$). Thus the query processing cost ($QPC$) for application $q_i$ is

$$QPC_i = PC_i + TC_i$$

According to the guidelines presented in Section 3.4.1, the processing component, $PC$, consists of three cost factors, the access cost ($AC$), the integrity enforcement cost ($IE$), and the concurrency control cost ($CC$):

$$PC_i = AC_i + IE_i + CC_i$$

The detailed specification of each of these cost factors depends on the algorithms used to accomplish these tasks. However, to demonstrate the point, we specify $AC$ in some detail:

$$AC_i = \sum_{\forall S_k \in S} \sum_{\forall F_j \in F} (u_{ij} * UR_{ij} + r_{ij} * RR_{ij}) * x_{jk} * LPC_k$$

The first two terms in the above formula calculate the number of accesses of user query $q_i$ to fragment $F_j$. Note that $(UR_{ij} + RR_{ij})$ gives the total number of update and retrieval accesses. We assume that the local costs of processing them are identical. The summation gives the total number of accesses for all the fragments referenced by $q_i$. Multiplication by $LPC_k$ gives the cost of this access at site $S_k$. We again use $x_{jk}$ to select only those cost values for the sites where fragments are stored.

A very important issue needs to be pointed out here. The access cost function assumes that processing a query involves decomposing it into a set of subqueries, each of which works on a fragment stored at the site, followed by transmitting the results back to the site where the query has originated. As we discussed earlier, this is a very simplistic view which does not take into consideration the complexities of database processing. For example, the cost function does not take into account the cost of performing joins (if necessary), which may be executed in a number of ways, studied in Chapter 8. In a model that is more realistic than the generic model we are considering, these issues should not be omitted.

The integrity enforcement cost factor can be specified much like the processing component, except that the unit local processing cost would probably change to reflect the true cost of integrity enforcement. Since the integrity checking and concurrency control methods are discussed later in the book, we do not need to study these cost components further here. The reader should refer back to this section after reading Chapters 5 and 11 to be convinced that the cost functions can indeed be derived.

The transmission cost function can be formulated along the lines of the access cost function. However, the data transmission overhead for update and that for retrieval requests are quite different. In update queries it is necessary to inform all the sites where replicas exist, while in retrieval queries, it is sufficient to access only one of the copies. In addition, at the end of an update request, there is no data transmission back to the originating site other than a confirmation message, whereas the retrieval-only queries may result in significant data transmission.

The update component of the transmission function is

$$TCU_i = \sum_{\forall S_k \in S} \sum_{\forall F_j \in F} u_{ij} * x_{jk} * g_{o(i),k} + \sum_{\forall S_k \in S} \sum_{\forall F_j \in F} u_{ij} * x_{jk} * g_{k,o(i)}$$

The first term is for sending the update message from the originating site $o(i)$ of $q_i$ to all the fragment replicas that need to be updated. The second term is for the confirmation.

The retrieval cost can be specified as

$$TCR_i = \sum_{\forall F_j \in F} \min_{S_k \in S} \left( r_{ij} * x_{jk} * g_{o(i),k} + r_{ij} * x_{jk} * \frac{sel_i(F_j) * length(F_j)}{fsize} * g_{k,o(i)} \right)$$

The first term in $TCR$ represents the cost of transmitting the retrieval request to those sites which have copies of fragments that need to be accessed. The second term accounts for the transmission of the results from these sites to the originating site. The equation states that among all the sites with copies of the same fragment, only the site that yields the minimum total transmission cost should be selected for the execution of the operation.

Now the transmission cost function for query $q_i$ can be specified as

$$TC_i = TCU_i + TCR_i$$

which fully specifies the total cost function.

### 3.4.3.2 Constraints

The constraint functions can be specified in similar detail. However, instead of describing these functions in depth, we will simply indicate what they should look like. The response-time constraint should be specified as

$$\text{execution time of } q_i \leq \text{ maximum response time of } q_i, \forall q_i \in Q$$

Preferably, the cost measure in the objective function should be specified in terms of time, as it makes the specification of the execution-time constraint relatively straightforward.

The storage constraint is

$$\sum_{\forall F_j \in F} STC_{jk} \leq \text{ storage capacity at site } S_k, \forall S_k \in S$$

whereas the processing constraint is

$$\sum_{\forall q_i \in Q} \text{processing load of } q_i \text{ at site } S_k \leq \text{ processing capacity of } S_k, \forall S_k \in S$$

This completes our development of the allocation model. Even though we have not developed it entirely, the precision in some of the terms indicates how one goes about formulating such a problem. In addition to this aspect, we have indicated the important issues that need to be addressed in allocation models.

### *3.4.4 Solution Methods*

In the preceding section we developed a generic allocation model which is considerably more complex than the FAP model presented in Section 3.4.1. Since the FAP model is NP-complete, one would expect the solution of this formulation of the database allocation problem (DAP) to be NP-complete as well. Even though we will not prove this conjecture, it is indeed true. Thus one has to look for heuristic methods that yield suboptimal solutions. The test of "goodness" in this case is, obviously, how close the results of the heuristic algorithm are to the optimal allocation.

A number of different heuristics have been applied to the solution of FAP and DAP models. It was observed early on that there is a correspondence between FAP and the plant location problem that has been studied in operations research. In fact, the isomorphism of the simple FAP and the single commodity warehouse location problem has been shown [Ramamoorthy and Wah, 1983]. Thus heuristics developed by operations researchers have commonly been adopted to solve the FAP and DAP problems. Examples are the knapsack problem solution [Ceri et al., 1982a], branch-and-bound techniques [Fisher and Hochbaum, 1980], and network flow algorithms [Chang and Liu, 1982].

There have been other attempts to reduce the complexity of the problem. One strategy has been to assume that all the candidate partitionings have been determined together with their associated costs and benefits in terms of query processing. The problem, then, is modeled so as to choose the optimal partitioning and placement for each relation [Ceri et al., 1983]. Another simplification frequently employed is to ignore replication at first and find an optimal non-replicated solution. Replication is handled at the second step by applying a greedy algorithm which starts with the non-replicated solution as the initial feasible solution, and tries to improve upon it ([Ceri et al., 1983] and [Ceri and Pernici, 1985]). For these heuristics, however, there is not enough data to determine how close the results are to the optimal.

## 3.5 Data Directory

The distributed database schema needs to be stored and maintained by the system. This information is necessary during distributed query optimization, as we will discuss later. The schema information is stored in a *data dictionary/directory*, also called a *catalog* or simply a directory. A directory is a meta-database that stores a number of information.

Within the context of the centralized ANSI/SPARC architecture discussed in Section 1.7.1, directory is the system component that permits mapping between different data organizational views. It should at least contain schema and mapping definitions. It may also contain usage statistics, access control information, and the like. It is clearly seen that the data dictionary/directory serves as the central component in both processing different schemas and in providing mappings among them.

In the case of a distributed database, as depicted in Figure 1.14 and discussed earlier in this chapter, schema definition is done at the global level (i.e., the global conceptual schema – GCS) as well as at the local sites (i.e., local conceptual schemas – LCSs). Consequently, there are two types of directories: a *global directory/dictionary* (GD/D)[5] that describes the database schema as the end users see it, and that permits the required global mappings between external schemas and the GCS, and the *local directory/dictionary* (LD/D), that describes the local mappings and describes the schema at each site. Thus, the local database management components are integrated by means of global DBMS functions.

As stated above, the directory is itself a database that contains *metadata* about the actual data stored in the database. Therefore, the techniques we discussed in this chapter with respect to distributed database design also apply to directory management. Briefly, a directory may be either *global* to the entire database or *local* to each site. In other words, there might be a single directory containing information about all the data in the database, or a number of directories, each containing the information stored at one site. In the latter case, we might either build hierarchies of directories to facilitate searches, or implement a distributed search strategy that involves considerable communication among the sites holding the directories.

The second issue has to do with location. In the case of a global directory, it may be maintained *centrally* at one site, or in a *distributed* fashion by distributing it over a number of sites. Keeping the directory at one site might increase the load at that site, thereby causing a bottleneck as well as increasing message traffic around that site. Distributing it over a number of sites, on the other hand, increases the complexity of managing directories. In the case of multi-DBMSs, the choice is dependent on whether or not the system is distributed. If it is, the directory is always distributed; otherwise of course, it is maintained centrally.

The final issue is replication. There may be a *single* copy of the directory or *multiple* copies. Multiple copies would provide more reliability, since the probability of reaching one copy of the directory would be higher. Furthermore, the delays

[5] In the remainder, we will simply refer to this as the *global directory*.

in accessing the directory would be lower, due to less contention and the relative proximity of the directory copies. On the other hand, keeping the directory up to date would be considerably more difficult, since multiple copies would need to be updated. Therefore, the choice should depend on the environment in which the system operates and should be made by balancing such factors as the response-time requirements, the size of the directory, the machine capacities at the sites, the reliability requirements, and the volatility of the directory (i.e., the amount of change experienced by the database, which would cause a change to the directory).

## 3.6 Conclusion

In this chapter, we presented the techniques that can be used for distributed database design with special emphasis on the fragmentation and allocation issues. There are a number of lines of research that have been followed in distributed database design. For example, Chang has independently developed a theory of fragmentation [Chang and Cheng, 1980], and allocation [Chang and Liu, 1982]. However, for its maturity of development, we have chosen to develop this chapter along the track developed by Ceri, Pelagatti, Navathe, and Wiederhold. Our references to the literature by these authors reflect this quite clearly.

There is a considerable body of literature on the allocation problem, focusing mostly on the simpler file allocation issue. We still do not have sufficiently general models that take into consideration all the aspects of data distribution. The model presented in Section 3.4 highlights the types of issues that need to be taken into account. Within this context, it might be worthwhile to take a somewhat different approach to the solution of the distributed allocation problem. One might develop a set of heuristic rules that might accompany the mathematical formulation and reduce the solution space, thus making the solution feasible.

We have discussed, in detail, the algorithms that one can use to fragment a relational schema in various ways. These algorithms have been developed quite independently and there is no underlying design methodology that combines the horizontal and vertical partitioning techniques. If one starts with a global relation, there are algorithms to decompose it horizontally as well as algorithms to decompose it vertically into a set of fragment relations. However, there are no algorithms that fragment a global relation into a set of fragment relations some of which are decomposed horizontally and others vertically. It is commonly pointed out that most real-life fragmentations would be mixed, i.e., would involve both horizontal and vertical partitioning of a relation, but the methodology research to accomplish this is lacking. What is needed is a distribution design methodology which encompasses the horizontal and vertical fragmentation algorithms and uses them as part of a more general strategy. Such a methodology should take a global relation together with a set of design criteria and come up with a set of fragments some of which are obtained via horizontal and others obtained via vertical fragmentation.

The second part of distribution design, namely allocation, is typically treated independently of fragmentation. The process is, therefore, linear when the output of fragmentation is input to allocation. At first sight, the isolation of the fragmentation and the allocation steps appears to simplify the formulation of the problem by reducing the decision space. However, closer examination reveals that isolating the two steps actually contributes to the complexity of the allocation models. Both steps have similar inputs, differing only in that fragmentation works on global relations whereas allocation considers fragment relations. They both require information about the user applications (e.g., how often they access data, what the relationships of individual data objects to one another are, etc.), but ignore how each other makes use of these inputs. The end result is that the fragmentation algorithms decide how to partition a relation based partially on how applications access it, but the allocation models ignore the part that this input plays in fragmentation. Therefore, the allocation models have to include all over again detailed specification of the relationship among the fragment relations and how user applications access them. What would be more promising is to formulate a methodology that more properly reflects the interdependence of the fragmentation and the allocation decisions. This requires extensions to existing distribution design strategies. We recognize that integrated methodologies such as the one we propose here may be considerably complex. However, there may be synergistic effects of combining these two steps enabling the development of quite acceptable heuristic solution methods. There are a few studies that follow such an integrated methodology (e.g., [Muro et al., 1983, 1985; Yoshida et al., 1985]). These methodologies build a simulation model of the distributed DBMS, taking as input a specific database design, and measure its effectiveness. Development of tools based on such methodologies, which aid the human designer rather than attempt to replace him, is probably the more appropriate approach to the design problem.

Another aspect of the work described in this chapter is that it assumes a static environment where design is conducted only once and this design can persist. Reality, of course, is quite different. Both physical (e.g., network characteristics, available storage at various sites) and logical (e.g., migration of applications from one site to another, access pattern modifications) changes occur necessitating redesign of the database. This problem has been studied to some extent. In a dynamic environment, the process becomes one of design-redesign-materialization of the redesign. The design step follows techniques that have been described in this chapter. Redesign can either be limited in that only parts of the database are affected, or total, requiring a complete redistribution [Wilson and Navathe, 1986]. Materialization refers to the reorganization of the distributed database to reflect the changes required by the redesign step. Limited redesign, in particular, the materialization issue is studied in [Rivera-Vega et al., 1990; Varadarajan et al., 1989]. Complete redesign and materialization issues have been studied in [Karlapalem et al., 1996b; Karlapalem and Navathe, 1994; Kazerouni and Karlapalem, 1997]. In particular, Kazerouni and Karlapalem [1997] describes a stepwise redesign methodology which involves a split phase where fragments are further subdivided based on the changed application requirements until no further subdivision is profitable based on a cost function. At this point, the merging phase starts where fragments that are accessed together by a set of applications are merged into one fragment.

## 3.7 Bibliographic Notes

Most of the known results about fragmentation have been covered in this chapter. Work on fragmentation in distributed databases initially concentrated on horizontal fragmentation. Most of the literature on this has been cited in the appropriate section. The topic of vertical fragmentation for distribution design has been addressed in several papers ([Navathe et al., 1984] and [Sacca and Wiederhold, 1985]. The original work on vertical fragmentation goes back to Hoffer's dissertation [Hoffer, 1975; Hoffer and Severance, 1975] and to Hammer and Niamir's work ([Niamir, 1978] and [Hammer and Niamir, 1979]).

It is not possible to be as exhaustive when discussing allocation as we have been for fragmentation, given there is no limit to the literature on the subject. The investigation of FAP on wide area networks goes back to Chu's work [Chu, 1969, 1973]. Most of the early work on FAP has been covered in the excellent survey by Dowdy and Foster [1982]. Some theoretical results about FAP are reported by Grapa and Belford [1977] and Kollias and Hatzopoulos [1981].

The DAP work dates back to the mid-1970s to the works of Eswaran [1974] and others. In their earlier work, Levin and Morgan [1975] concentrated on data allocation, but later they considered program and data allocation together [Morgan and Levin, 1977]. The DAP has been studied in many specialized settings as well. Work has been done to determine the placement of computers and data in a wide area network design [Gavish and Pirkul, 1986]. Channel capacities have been examined along with data placement [Mahmoud and Riordon, 1976] and data allocation on supercomputer systems [Irani and Khabbaz, 1982] as well as on a cluster of processors [Sacca and Wiederhold, 1985]. An interesting work is the one by Apers, where the relations are optimally placed on the nodes of a virtual network, and then the best matching between the virtual network nodes and the physical network are found [Apers, 1981].

Some of the allocation work has also touched upon physical design. The assignment of files to various levels of a memory hierarchy has been studied by Foster and Browne [1976] and by Navathe et al. [1984]. These are outside the scope of this chapter, as are those that deal with general resource and task allocation in distributed systems (e.g., [Bucci and Golinelli, 1977], [Ceri and Pelagatti, 1982], and [Haessig and Jenny, 1980]).

We should finally point out that some effort was spent to develop a general methodology for distributed database design along the lines that we presented (Figure 3.2). Ours is similar to the DATAID-D methodology [Ceri and Navathe, 1983; Ceri et al., 1987]. Other attempts to develop a methodology are due to Fisher et al. [1980], Dawson [1980]; Hevner and Schneider [1980] and Mohan [1979].

## Exercises

**Problem 3.1 (*).** Given relation EMP as in Figure 3.3, let $p_1$: TITLE $<$ "Programmer" and $p_2$: TITLE $>$ "Programmer" be two simple predicates. Assume that character strings have an order among them, based on the alphabetical order.

**(a)** Perform a horizontal fragmentation of relation EMP with respect to $\{p_1, p_2\}$.
**(b)** Explain why the resulting fragmentation (EMP$_1$, EMP$_2$) does not fulfill the correctness rules of fragmentation.
**(c)** Modify the predicates $p_1$ and $p_2$ so that they partition EMP obeying the correctness rules of fragmentaion. To do this, modify the predicates, compose all minterm predicates and deduce the corresponding implications, and then perform a horizontal fragmentation of EMP based on these minterm predicates. Finally, show that the result has completeness, reconstruction and disjointness properties.

**Problem 3.2 (*).** Consider relation ASG in Figure 3.3. Suppose there are two applications that access ASG. The first is issued at five sites and attempts to find the duration of assignment of employees given their numbers. Assume that managers, consultants, engineers, and programmers are located at four different sites. The second application is issued at two sites where the employees with an assignment duration of less than 20 months are managed at one site, whereas those with longer duration are managed at a second site. Derive the primary horizontal fragmentation of ASG using the foregoing information.

**Problem 3.3.** Consider relations EMP and PAY in Figure 3.3. EMP and PAY are horizontally fragmented as follows:

$$\begin{aligned}
\text{EMP}_1 &= \sigma_{\text{TITLE=``Elect.Eng.''}}(\text{EMP})\\
\text{EMP}_2 &= \sigma_{\text{TITLE=``Syst.Anal.''}}(\text{EMP})\\
\text{EMP}_3 &= \sigma_{\text{TITLE=``Mech.Eng.''}}(\text{EMP})\\
\text{EMP}_4 &= \sigma_{\text{TITLE=``Programmer''}}(\text{EMP})
\end{aligned}$$

$$\begin{aligned}
\text{PAY}_1 &= \sigma_{\text{SAL}\geq 30000}(\text{PAY})\\
\text{PAY}_2 &= \sigma_{\text{SAL}<30000}(\text{PAY})
\end{aligned}$$

Draw the join graph of EMP $\ltimes_{\text{TITLE}}$ PAY. Is the graph simple or partitioned? If it is partitioned, modify the fragmentation of either EMP or PAY so that the join graph of EMP$\ltimes_{\text{TITLE}}$ PAY is simple.

**Problem 3.4.** Give an example of a *CA* matrix where the split point is not unique and the partition is in the middle of the matrix. Show the number of shift operations required to obtain a single, unique split point.

**Problem 3.5 (**).** Given relation PAY as in Figure 3.3, let $p_1$: SAL $<$ 30000 and $p_2$: SAL $\geq$ 30000 be two simple predicates. Perform a horizontal fragmentation of PAY with respect to these predicates to obtain PAY$_1$, and PAY$_2$. Using the fragmentation of PAY, perform further derived horizontal fragmentation for EMP. Show completeness, reconstruction, and disjointness of the fragmentation of EMP.

**Problem 3.6 (**).** Let $Q = \{q_1, \ldots, q_5\}$ be a set of queries, $A = \{A_1, \ldots, A_5\}$ be a set of attributes, and $S = \{S_1, S_2, S_3\}$ be a set of sites. The matrix of Figure 3.21a describes the attribute usage values and the matrix of Figure 3.21b gives the application access frequencies. Assume that $ref_i(q_k) = 1$ for all $q_k$ and $S_i$ and that $A_1$ is the key attribute. Use the bond energy and vertical partitioning algorithms to obtain a vertical fragmentation of the set of attributes in $A$.

$$\begin{array}{c|ccccc|}
 & A_1 & A_2 & A_3 & A_4 & A_5 \\
\hline
q_1 & 0 & 1 & 1 & 0 & 1 \\
q_2 & 1 & 1 & 1 & 0 & 1 \\
q_3 & 1 & 0 & 0 & 1 & 1 \\
q_4 & 0 & 0 & 1 & 0 & 0 \\
q_5 & 1 & 1 & 1 & 0 & 0 \\
\end{array}
\qquad
\begin{array}{c|ccc|}
 & S_1 & S_2 & S_3 \\
\hline
q_1 & 10 & 20 & 0 \\
q_2 & 5 & 0 & 10 \\
q_3 & 0 & 35 & 5 \\
q_4 & 0 & 10 & 0 \\
q_5 & 0 & 15 & 0 \\
\end{array}$$

(a) (b)

**Fig. 3.21** Attribute Usage Values and Application Access Frequencies in Exercise 3.6

**Problem 3.7 (**).** Write an algorithm for derived horizontal fragmentation.

**Problem 3.8 (**).** Assume the following view definition

```
CREATE VIEW   EMPVIEW(ENO, ENAME, PNO, RESP)
AS     SELECT EMP.ENO, EMP.ENAME, ASG.PNO,
              ASG.RESP
       FROM   EMP, ASG
       WHERE  EMP.ENO=ASG.ENO
       AND    DUR=24
```

is accessed by application $q_1$, located at sites 1 and 2, with frequencies 10 and 20, respectively. Let us further assume that there is another query $q_2$ defined as

```
SELECT ENO, DUR
FROM   ASG
```

which is run at sites 2 and 3 with frequencies 20 and 10, respectively. Based on the above information, construct the $use(q_i, A_j)$ matrix for the attributes of both relations EMP and ASG. Also construct the affinity matrix containing all attributes of EMP and ASG. Finally, transform the affinity matrix so that it could be used to split the relation into two vertical fragments using heuristics or BEA.

**Problem 3.9 (**).** Formally define the three correctness criteria for derived horizontal fragmentation.

**Problem 3.10 (*).** Given a relation $R(K,A,B,C)$ (where $K$ is the key) and the following query

```
SELECT *
FROM   R
WHERE  R.A = 10 AND R.B=15
```

**(a)** What will be the outcome of running PHF on this query?
**(b)** Does the COM_MIN algorithm produce in this case a complete and minimal predicate set? Justify your answer.

**Problem 3.11 (*).** Show that the bond energy algorithm generates the same results using either row or column operation.

**Problem 3.12 (**).** Modify algorithm PARTITION to allow $n$-way partitioning, and compute the complexity of the resulting algorithm.

**Problem 3.13 (**).** Formally define the three correctness criteria for hybrid fragmentation.

**Problem 3.14.** Discuss how the order in which the two basic fragmentation schemas are applied in hybrid fragmentation affects the final fragmentation.

**Problem 3.15 (**).** Describe how the following can be properly modeled in the database allocation problem.

**(a)** Relationships among fragments
**(b)** Query processing
**(c)** Integrity enforcement
**(d)** Concurrency control mechanisms

**Problem 3.16 (**).** Consider the various heuristic algorithms for the database allocation problem.

**(a)** What are some of the reasonable criteria for comparing these heuristics? Discuss.
**(b)** Compare the heuristic algorithms with respect to these criteria.

**Problem 3.17 (*).** Pick one of the heuristic algorithms used to solve the DAP, and write a program for it.

**Problem 3.18 (**).** Assume the environment of Exercise 3.8. Also assume that 60% of the accesses of query $q_1$ are updates to PNO and RESP of view EMPVIEW and that ASG.DUR is not updated through EMPVIEW. In addition, assume that the data transfer rate between site 1 and site 2 is half of that between site 2 and site 3. Based on the above information, find a reasonable fragmentation of ASG and EMP and an optimal replication and placement for the fragments, assuming that storage costs do not matter here, but copies are kept consistent.

Hint: Consider horizontal fragmentation for ASG based on DUR=24 predicate and the corresponding derived horizontal fragmentation for EMP. Also look at the affinity matrix obtained in Example 3.8 for EMP and ASG together, and consider whether it would make sense to perform a vertical fragmentation for ASG.

# Chapter 4
# Database Integration

In the previous chapter, we discussed top-down distributed database design, which is suitable for tightly integrated, homogeneous distributed DBMSs. In this chapter, we focus on bottom-up design that is appropriate in multidatabase systems. 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 individual local conceptual schemas. The process consists of integrating local databases with their (local) schemas into a global database with its global conceptual schema (GCS) (also called the *mediated schema*).

Database integration, and the related problem of querying multidatabases (see Chapter 9), is only one part of the more general *interoperability* problem. In recent years, new distributed applications have started to pose new requirements regarding the data source(s) they access. In parallel, the management of "legacy systems" and reuse of the data they generate have gained importance. The result has been a renewed consideration of the broader question of information system interoperability, including non-database sources and interoperability at the application level in addition to the database level.

Database integration can be either physical or logical [Jhingran et al., 2002]. 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, their transformation to match the GCS, and their loading (i.e., materialization). *Enterprise Application Integration* (EAI), which allows data exchange between applications, perform similar transformation functions, although data are not entirely materialized. This process is depicted in Figure 4.1. In logical integration, the global conceptual (or mediated) schema is entirely *virtual* and not materialized. This is also known as *Enterprise Information Integration* (EII)[1].

These two approaches are complementary and address differing needs. Data warehousing [Inmon, 1992; Jarke et al., 2003] supports decision support applications,

[1] It has been (rightly) argued that the second "I" should stand for Interoperability rather than Integration (see J. Pollock's contribution in [Halevy et al., 2005]).

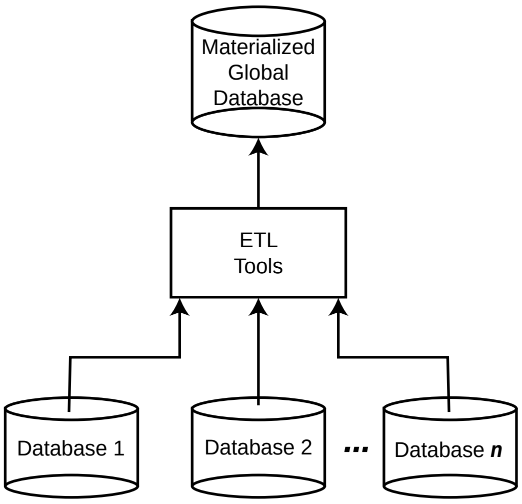


**Fig. 4.1** Data Warehouse Approach

which are commonly termed *On-line Analytical Processing* (OLAP) [Codd, 1995] to better reflect their different requirements relative to the On-Line Transaction Processing (OLTP) applications. OLTP applications, such as airline reservation or banking systems, are high-throughput transaction-oriented. They need extensive data control and availability, high multiuser throughput and predictable, fast response times. In contrast, OLAP applications, such as trend analysis or forecasting, need to analyze historical, summarized data coming from a number of operational databases. They use complex queries over potentially very large tables. Because of their strategic nature, response time is important. The users are managers or analysts. Performing OLAP queries directly over distributed operational databases raises two problems. First, it hurts the OLTP applications' performance by competing for local resources. Second, the overall response time of the OLAP queries can be very poor because large quantities of data must be transferred over the network. Furthermore, most OLAP applications do not need the most current versions of the data, and thus do not need direct access to most up-to-date operational data. Consequently, data warehouses gather data from a number of operational databases and materialize them. As updates happen on the operational databases, they are propagated to the data warehouse (also referred to as *materialized view maintenance* [Gupta and Mumick, 1999b]).

By contrast, in logical data integration, the integration is only virtual and there is no materialized global database (see Figure 1.18). The data resides in the operational databases and the GCS provides a virtual integration for querying over them similar to the case described in the previous chapter. The difference is that the GCS may not be the union of the local conceptual schamas (LCSs). It is possible for the GCS not to capture all of the information in each of the LCSs. Furthermore, in some cases, the GCS may be defined bottom-up, by "integrating" parts of the LCSs of the local operational databases rather than being defined up-front (more on this shortly). 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. The main differences are the autonomy and potential heterogeneity of the local systems. These have important effects on query processing that we discuss in Chapter 9. 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. The generality of these terms point to the fact that the underlying data sources do not have to be databases. In this chapter we focus our attention on the integration of autonomous and (possibly) heterogeneous databases; thus we will use the term *database integration* (which also helps to distinguish these systems from data warehouses).

## 4.1 Bottom-Up Design Methodology

Bottom-up design involves the process by which information from participating databases can be (physically or logically) integrated to form a single cohesive multi-database. There are two alternative approaches. 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. This is the case in data warehouses, but the practice is not restricted to these and other data integration methodologies may follow the same strategy. 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 up-front, the relationship between the GCS and the local conceptual schemas (LCS) can be of two fundamental types [Lenzerini, 2002]: 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 [Koch, 2001]. 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 (Figure 4.2a). In LAV, on the other hand, the results are constrained by the objects in the local DBMSs, while the GCS definition may be richer (Figure 4.2b). 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) [Friedman et al., 1999] where the relationship between GCS and LCSs is specified using both LAV and GAV.

Bottom-up design occurs in two general steps (Figure 4.3): *schema translation* (or simply *translation*) and *schema generation*. In the first step, the component

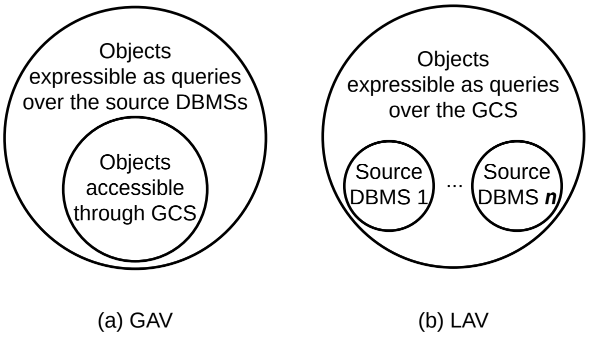

Objects expressible as queries over the source DBMSs
Objects accessible through GCS
(a) GAV
Objects expressible as queries over the GCS
Source DBMS 1
...
Source DBMS *n*
(b) LAV

**Fig. 4.2** GAV and LAV Mappings (Based on [Koch, 2001])

database schemas are translated to a common intermediate canonical representation ($InS_1$, $InS_2, \ldots,$ $InS_n$). The use of a canonical representation facilitates the translation 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 [Palopoli et al., 1998, 2003b; He and Ling, 2006], object-oriented model [Castano and Antonellis, 1999; Bergamaschi et al., 2001], or a graph [Palopoli et al., 1999; Milo and Zohar, 1998; Melnik et al., 2002; Do and Rahm, 2002] that may be simplified to a tree [Madhavan et al., 2001]. The graph (tree) models have become more popular as XML data sources have proliferated, since it is fairly straightforward to map XML to graphs, although there are efforts to target XML directly [Yang et al., 2003]. In this chapter, we will simply use the relational model as our canonical data model, because we have been using it throughout the book, and the graph models used in literature are quite diverse with no common graph representation. The choice of the relational model as the canonical data representation 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 [Yan, 1997; Yan et al., 1997]). 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. In some methodologies, *local external* (or *export*) *schemas* are considered for integration rather than full database schemas, to reflect the fact that

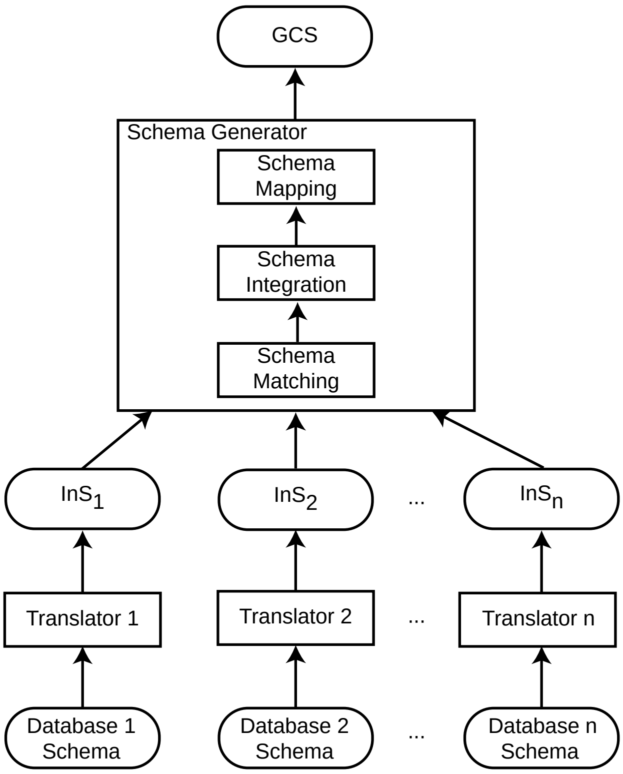


**Fig. 4.3** Database Integration Process

local systems may only be willing to contribute some of their data to the multidatabase [Sheth and Larson, 1990].

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 pre-defined GCS elements (Section 4.2).
2. Integration of the common schema elements into a global conceptual (mediated) schema if one has not yet been defined (Section 4.3).
3. Schema mapping that determines how to map the elements of each LCS to the other elements of the GCS (Section 4.4).

It is also possible that the schema mapping step may be divided into two phases [Bernstein and Melnik, 2007]: 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 4.1.* 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 have developed in Chapter 2. We repeat that definition in Figure 4.4 for completeness. The underscored attributes are the keys of the associated relations. We have made one modification in the PROJ relation by including attributes LOC and CNAME. LOC is the location of the project, whereas CNAME is the name of the client for whom the project is carried out. The second database also defined similar data, but is specified according to the entity-relationship (E-R) data model [Chen, 1976] as depicted in Figure 4.5.

EMP($\underline{\text{ENO}}$, ENAME, TITLE)

PROJ($\underline{\text{PNO}}$, PNAME, BUDGET, LOC, CNAME)

ASG($\underline{\text{ENO, PNO}}$, RESP, DUR)

PAY($\underline{\text{TITLE}}$, SAL)

**Fig. 4.4** Relational Engineering Database Representation

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 Figure 4.5. This database is similar to the relational engineering database definition of Figure 4.4, with one significant difference: it also maintains data about the clients for whom the projects are conducted. The rectangular boxes in Figure 4.5 represent the entities modeled in the database, and the diamonds indicate a relationship between the entities to which they are connected. The type of relationship 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 elliptical circles. ♦

*Example 4.2.* The mapping of the E-R model to the relational model is given in Figure 4.6. Note that we have renamed some of the attributes in order to ensure name uniqueness. ♦

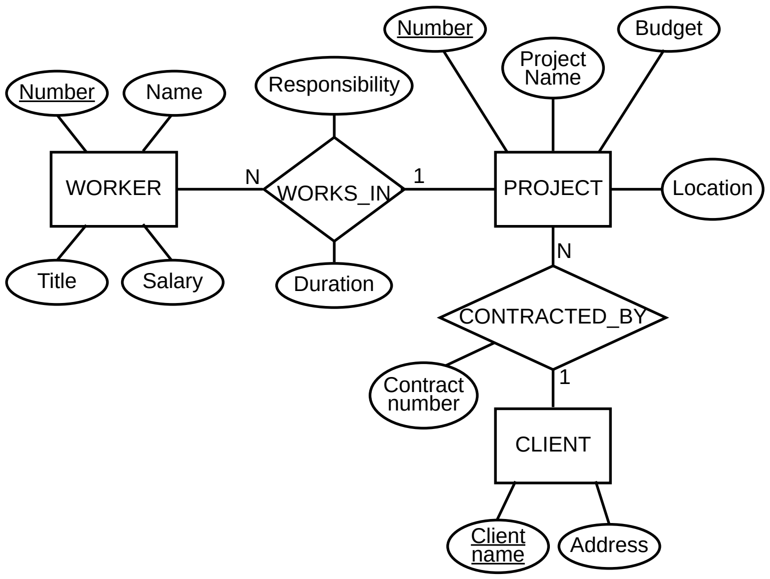


**Fig. 4.5** Entity-Relationship Database

WORKER(<u>WNUMBER</u>, NAME, TITLE, SALARY)

PROJECT(<u>PNUMBER</u>, PNAME, BUDGET)

CLIENT(<u>CNAME</u>, ADDRESS)

WORKS_IN(<u>WNUMBER, PNUMBER</u>, RESPONSIBILITY, DURATION)

CONTRACTED_BY(<u>PNUMBER, CNAME</u>, CONTRACTNO)

**Fig. 4.6** Relational Mapping of E-R Schema

## 4.2 Schema Matching

Schema matching determines which concepts of one schema match those of another. 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 on 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 ($c$) 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 ($p$) 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 ($s$) 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 $\mathcal{M} = \{r\}$ 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, as we discuss below, 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 (Section 4.2.1). 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 is some ambiguity of the terms 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, we need to note (and admit) 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.

Despite these difficulties, serious progress has been made in recent years in developing algorithmic approaches to the matching problem. In this section, we discuss a number of these algorithms and the various approaches.

A number of issues affect the particular matching algorithm [Rahm and Bernstein, 2001]. 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 objects." 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 data 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 peer-to-peer systems (see Chapter 16), 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 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 [Rahm and Bernstein, 2001]. 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 (Figure 4.7). 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 Section 4.2.2, constraint-based approaches in Section 4.2.3, and learning-based techniques in Section 4.2.4. Rahm and Bernstein [2001] refer to all of these as *individual matcher* approaches, and their combinations are possible by developing either *hybrid matchers* or *composite matchers* (Section 4.2.5).

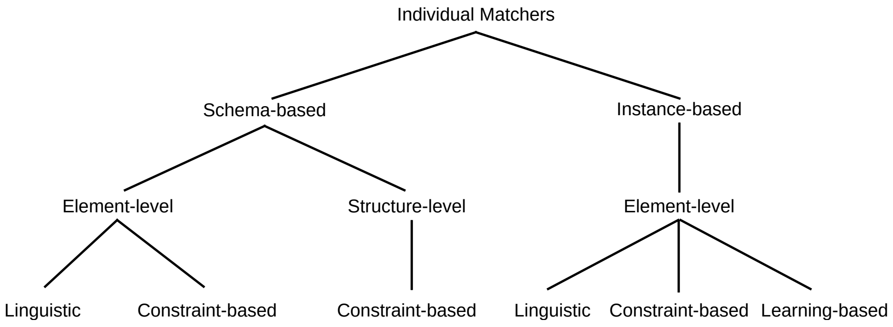


**Fig. 4.7** Taxonomy of Schema Matching Techniques

### *4.2.1 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* [Batini et al., 1986]. Type conflicts occur when the same object is represented by an attribute in one schema and by 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 4.3.* We have two structural conflicts in the example we are considering. The first is a type conflict involving clients of projects. In the schema of Figure 4.5, the client of a project is modeled as an entity. In the schema of Figure 4.4, 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 Figure 4.5 and the ASG relation in Figure 4.4. 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 [Vermeer, 1997]. There are attempts to formalize semantic heterogeneity and to establish its link to structural heterogeneity [Kashyap and Sheth, 1996; Sheth and Kashyap, 1992]; 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 and PROJECT 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 dependent on the domain they are used in. 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 and LOC attributes in our example database may refer to the full address or just the city name. 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.

## 4.2.2 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 $\langle \mathsf{SC1.element\text{-}1} \approx \mathsf{SC2.element\text{-}2}, p, s \rangle$ 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 hand-crafted or may be "discovered" using auxiliary data sources such as thesauri, e.g., WordNet [Miller, 1995] (http://wordnet.princeton.edu/). In the case of hand-crafted 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 hand-crafted linguistic rules may deal with capitalization, abbreviations, concept relationships, etc. In some systems, the hand-crafted rules are specified for each schema individually (*intraschema rules*) by the designer, and *interschema rules* are then "discovered" by the matching algorithm [Palopoli et al., 1999]. However, in most cases, the rule base contains both intra and interschema rules.

*Example 4.4.* In the relational database of Example 4.2, 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:

$\langle \textsf{uppercase names} \approx \textsf{lower case names}, true, 1.0)\rangle$
$\langle \textsf{uppercase names} \approx \textsf{capitalized names}, true, 1.0)\rangle$
$\langle \textsf{capitalized names} \approx \textsf{lower case names}, true, 1.0)\rangle$
$\langle \textsf{RelDB.ASG} \approx \textsf{ERDB.WORKS\_IN}, true, 0.8\rangle$
...

The first three rules are generic ones specifying how to deal with capitalizations, while the fourth one specifies a similarity between the ASG element of RelDB and the WORKS_IN element 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 [Do and Rahm, 2002] 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 Lewenstein 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 are 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 4.5.* Consider matching the RESP and the RESPONSIBILITY attributes in the two example schemas we are considering. The rules defined in Example 4.4 take care of the capitalization differences, so we are left with matching RESP with RESPONSIBILITY. Let us consider how the similarity between these 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 RESP or delete the same characters from 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 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). Rahm and Bernstein [2001] suggest that 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.

### 4.2.3 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 computations in Example 4.5. 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 Figure 4.4 has the same edit distance and *n*-gram similarity values to the two NUMBER attributes in Figure 4.5 (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) are 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 in determining 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 [Madhavan et al., 2001; Do and Rahm, 2002] where each concept (relation, entity, attribute) is a node and there is a directed edge between two nodes if and only if the two concepts are related (e.g., there is an edge from a relation node to each of its attributes, or there is an edge from a foreign key attribute node to the primary key attribute node it is referencing). In this case, the neighborhood can be defined in terms of the nodes 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.

The traversing of the graph can be done in a number of ways; for example CUPID [Madhavan et al., 2001] converts the graphs to trees and then looks at similarities of subtrees rooted at the two nodes in consideration, while COMA [Do and Rahm, 2002] considers the paths from the root to these element nodes. The fundamental point of these algorithms is that if the subgraphs are similar, this increases the similarity of the roots of these subtrees. The similarity of the subgraphs are determined in a bottom-up process, starting at the leaves whose similarity are 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 nodes in the subtree. A number of formulae may be used to for this recursive computation. CUPID, for example, looks at the similarity of two leaf nodes and if it is higher than a threshold value, then those two leaf nodes are said to be *strongly linked*. The similarity of two subgraphs is then defined as the fraction of leaves in the two subtrees that are strongly linked. This is based on the assumption that leafs carry more information and that the structural similarity of two non-leaf schema elements is determined by the similarity of the leaf nodes 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 nodes is *similarity flooding* [Melnik et al., 2002]. It starts from an initial graph where the node similarities are already determined by means of an element matcher, and propagates, iteratively, to determine the similarity of each node to its neighbors. Hence, whenever any two elements in two schemas are found to be similar, the similarity of their adjacent nodes increases. The iterative process stops when the node similarities stabilize. At each iteration, to reduce the amount of work, a subset of the nodes are selected as the "most plausible" matches, which are then considered in the subsequent iteration.

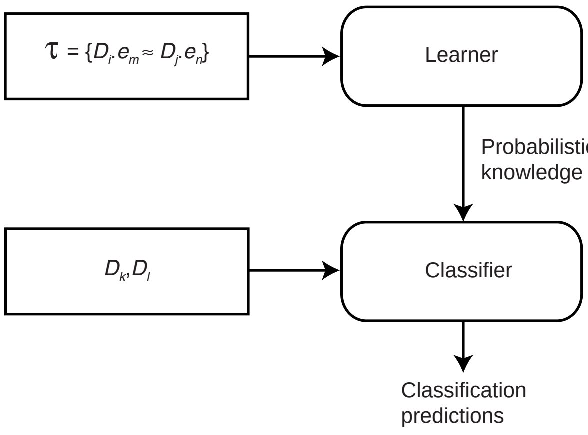
Both of these approaches are agnostic to the edge semantics. In some graph representations, there is additional semantics attached to these edges. For example, *containment edges* from a relation or entity node to its attributes may be distinguished from *referential edges* from a foreign key attribute node to the corresponding primary key attribute node. Some systems exploit these edge semantics (e.g., DIKE [Palopoli et al., 1998, 2003a]).

### *4.2.4 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 data set.

The process is as follows (Figure 4.8). A training set ($\tau$) 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 [Doan et al., 2003a], or by the specification of a query expression that converts data from one database to another [Berlin and Motro, 2001]. The learner uses this training data to acquire probabilistic information about the features of the data sets. 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 [Li and Clifton, 2000; Li et al., 2000]), others have used Naïve Bayesian learner/classifier (Autoplex [Berlin and Motro, 2001], LSD [Doan



$\tau = \{D_i.e_m \approx D_j.e_n\}$
Learner
Probabilistic knowledge
$D_k, D_l$
Classifier
Classification predictions

**Fig. 4.8** Learning-based Matching Approach

et al., 2001, 2003a] and [Naumann et al., 2002]), and decision trees [Embley et al., 2001, 2002]. Discussing the details of these learning techniques are beyond our scope.

### *4.2.5 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 [Rahm and Bernstein, 2001]: 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 $i$ ($i = 1, ..., q$) 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 [Fagin, 2002] that we will discuss further in Chapter 9. Composite approach has been proposed in the LSD [Doan et al., 2001, 2003a] and iMAP [Dhamankar et al., 2004] systems for handling 1:1 and N:M matches, respectively.

## 4.3 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 GSC was defined up-front, 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 (e.g., [Sheth et al., 1988a]) have been developed to aid in the integration process, human involvement is clearly essential.

*Example 4.6.* There are a number of possible integrations of the two example LCSs we have been discussing. Figure 4.9 shows one possible GCS that can be generated as a result of schema integration. ♦

Employee(ENUMBER, ENAME, TITLE)

Pay(TITLE, SALARY)

Project(PNUMBER, PNAME, BIDGET, LOCATION)

Client(CNAME, ADDRESS, CONTRACTNO, PNUMBER)

Works(ENUMBER, PNUMBER, RESP, DURATION)

**Fig. 4.9** Example Integrated GCS

Integration methodologies can be classified as binary or *n*ary mechanisms [Batini et al., 1986] based on the manner in which the local schemas are handled in the first phase (Figure 4.10). Binary integration methodologies involve the manipulation of two schemas at a time. These can occur in a stepwise (ladder) fashion (Figure 4.11a) where intermediate schemas are created for integration with subsequent schemas [Pu, 1988], or in a purely binary fashion (Figure 4.11b), where each schema is integrated with one other, creating an intermediate schema for integration with other intermediate schemas ([Batini and Lenzirini, 1984] and [Dayal and Hwang, 1984]).

Other binary integration approaches do not make this distinction [Melnik et al., 2002].

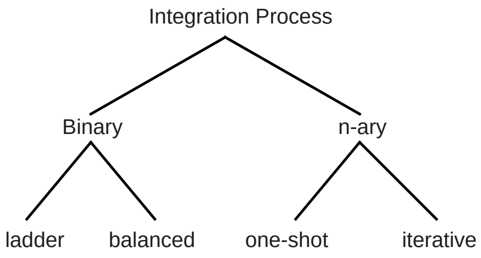


**Fig. 4.10** Taxonomy of Integration Methodologies

*N*ary integration mechanisms integrate more than two schemas at each iteration. One-pass integration (Figure 4.12a) 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.

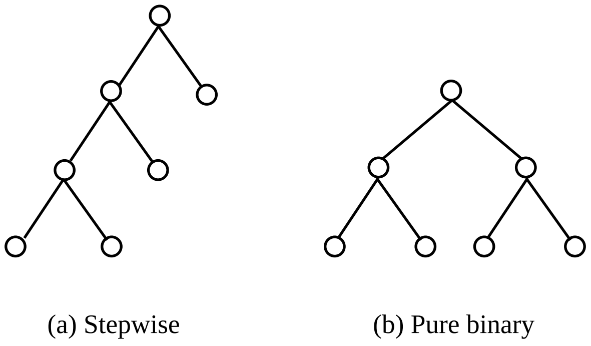


**Fig. 4.11** Binary Integration Methods

Iterative *n*ary integration (Figure 4.12b) 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 toward 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

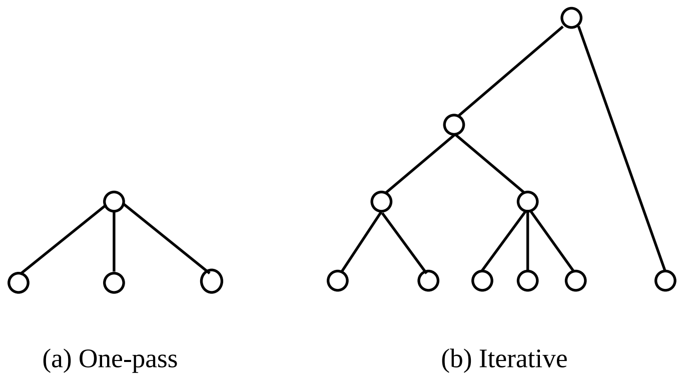


**Fig. 4.12** *N*ary Integration Methods

systems utilize binary methodology, but a number of researchers prefer the *n*ary approach because complete information is available ([Elmasri et al., 1987; Yao et al., 1982b; He et al., 2004]).

## 4.4 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 Chapter 9).

There are two issues related to schema mapping that we will be studying: *mapping creation*, and *mapping maintenance*. Mapping creation is the process of creating explicit queries that map data from a local database to the global data. 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.

## 4.4.1 Mapping Creation

Mapping creation starts with a source LCS, the target GCS, and a set of schema matches $\mathcal{M}$ 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, these queries are used in the reverse direction during query processing (Chapter 9).

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 $\mathcal{S} = \{S_1, \ldots, S_m\}$, the GCS consists of a set of global (or target) relations $\mathcal{T} = \{T_1, \ldots, T_n\}$, and $\mathcal{M}$ consists of a set of schema match rules as defined in Section 4.2. We are looking for a way to generate, for each $T_k$, a query $Q_k$ that is defined on a (possibly proper) subset of the relations in $\mathcal{S}$ such that, when executed, will generate data for $T_k$ from the source relations.

An algorithm due to Miller et al. [2000] accomplishes this iteratively by considering each $T_k$ in turn. It starts with $M_k \subseteq \mathcal{M}$ ($M_k$ is the set of rules that only apply to the attributes of $T_k$) 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 $T_k$ can be computed. Each $M_k^j$ can be mapped to a query $q_k^j$ that, when executed, would generate *some* of $T_k$'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., $(S_i.attribute_k, S_j.attribute_l) \mapsto T_w.attribute_z$)).

*Example 4.7.* 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):

$S_1(A_1, A_2)$
$S_2(B_1, B_2, B_3)$
$S_3(C_1, C_2, C_3)$
$S_4(D_1, D_2)$

Target relation (GCS)

$T(W_1, W_2, W_3, W_4)$

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

The foreign key relationships between the attributes are as follows:

| Foreign key | Refers to |
|---|---|
| $A_1$ | $B_1$ |
| $A_2$ | $B_1$ |
| $C_1$ | $B_1$ |

The following matches have been discovered for attributes of relation $T$ (these make up $M_T$). In the subsequent examples, we will not be concerned with the predicates, so they are not explicitly specified.

$$\begin{aligned}
r_1 &= \langle A_1 \mapsto W_1, p\rangle\\
r_2 &= \langle A_2 \mapsto W_2, p\rangle\\
r_3 &= \langle B_2 \mapsto W_4, p\rangle\\
r_4 &= \langle B_3 \mapsto W_3, p\rangle\\
r_5 &= \langle C_1 \mapsto W_1, p\rangle\\
r_6 &= \langle C_2 \mapsto W_2, p\rangle\\
r_7 &= \langle D_1 \mapsto W_4, p\rangle
\end{aligned}$$

♦

In the first step, $M_k$ (corresponding to $T_k$) 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 $T_k$. These are called *potential candidate sets*, some of which may be *complete* in that they include a match for every attribute of $T_k$, 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 are 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 4.8.* $M_T$ is partitioned into the following fifty-three subsets (i.e., potential candidate sets). The first eight of these are complete, while the rest are not. 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 $M_T^1$ is $r_4$, because this rule maps to attribute $W_3$):

$$\begin{aligned}
M_T^1 &= \{r_1, r_2, r_4, r_3\} & M_T^2 &= \{r_1, r_2, r_4, r_7\}\\
M_T^3 &= \{r_1, r_6, r_4, r_3\} & M_T^4 &= \{r_1, r_6, r_4, r_7\}\\
M_T^5 &= \{r_5, r_2, r_4, r_3\} & M_T^6 &= \{r_5, r_2, r_4, r_7\}\\
M_T^7 &= \{r_5, r_6, r_4, r_3\} & M_T^8 &= \{r_5, r_6, r_4, r_7\}\\
M_T^9 &= \{r_1, r_2, r_3\} & M_T^{10} &= \{r_1, r_2, r_4\}\\
M_T^{11} &= \{r_1, r_3, r_4\} & M_T^{12} &= \{r_2, r_3, r_4\}
\end{aligned}$$

$$M_T^{13} = \{r_1, r_3, r_6\} \qquad M_T^{14} = \{r_3, r_4, r_6\}$$
$$\ldots \qquad \ldots$$
$$M_T^{47} = \{r_1\} \qquad M_T^{48} = \{r_2\}$$
$$M_T^{49} = \{r_3\} \qquad M_T^{50} = \{r_4\}$$
$$M_T^{51} = \{r_5\} \qquad M_T^{52} = \{r_6\}$$
$$M_T^{53} = \{r_7\}$$

♦

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 $T_k$, 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 isn’t, 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 [Miller et al., 2001], OntoBuilder [Roitman and Gal, 2006] and others facilitate this process and provide mechanisms for designers to view and specify correspondences [Yan et al., 2001]). The result of this step is a set $\overline{M_k} \subseteq M_k$ of *candidate sets*.

*Example 4.9.* 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 $S_1, S_2$ and $S_3$ can be mapped to “good” queries since there are foreign key relationships between them. However, the rules that involve $S_4$ (i.e., those that include rule $r_7$) cannot be mapped to a “good” query since there is no join path from $S_4$ 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 $\mathcal{C}_k \subseteq \overline{M_k}$ is a set of candidate sets such that each match in $\overline{M_k}$ appears in $\mathcal{C}_k$ 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 $T_k$. 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 4.10.* 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:

$$\mathcal{C}_T^1 = \{\underbrace{\{r_1,r_2,r_4,r_3\}}_{M_T^1},\underbrace{\{r_1,r_6,r_4,r_3\}}_{M_T^3},\underbrace{\{r_2\}}_{M_T^{48}}\}$$

$$\mathcal{C}_T^2 = \{\underbrace{\{r_1,r_2,r_4,r_3\}}_{M_T^1},\underbrace{\{r_5,r_2,r_4,r_3\}}_{M_T^5},\underbrace{\{r_6\}}_{M_T^{50}}\}$$

$$\mathcal{C}_T^3 = \{\underbrace{\{r_1,r_2,r_4,r_3\}}_{M_T^1},\underbrace{\{r_5,r_6,r_4,r_3\}}_{M_T^7}\}$$

$$\mathcal{C}_T^4 = \{\underbrace{\{r_1,r_2,r_4,r_3\}}_{M_T^1},\underbrace{\{r_5,r_6,r_4\}}_{M_T^{12}}\}$$

$$\mathcal{C}_T^5 = \{\underbrace{\{r_1,r_2,r_4,r_3\}}_{M_T^1},\underbrace{\{r_5,r_6,r_3\}}_{M_T^{19}}\}$$

$$\mathcal{C}_T^6 = \{\underbrace{\{r_1,r_2,r_4,r_3\}}_{M_T^1},\underbrace{\{r_5,r_6\}}_{M_T^{32}}\}$$

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, $\mathcal{C}_T^3$ is the preferred cover in this case.

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 $\mathcal{C}_T^3$ 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 $T_k$ 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 4.11.* Since in Example 4.10 we have decided to use cover $\mathcal{C}_T^3$ 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:

$$
\begin{aligned}
r_1 &= \langle A_1 \mapsto W_1, p\rangle \\
r_2 &= \langle A_2 \mapsto W_2, p\rangle \\
r_3 &= \langle B_2 \mapsto W_4, p\rangle \\
r_4 &= \langle B_3 \mapsto W_3, p\rangle \\
r_5 &= \langle C_1 \mapsto W_1, p\rangle \\
r_6 &= \langle C_2 \mapsto W_2, p\rangle
\end{aligned}
$$

The two queries are as follows:

$$
\begin{aligned}
q_k^1 :\ & \texttt{SELECT}\ A_1, A_2, B_2, B_3 \\
& \texttt{FROM}\quad S_1, S_2 \\
& \texttt{WHERE}\quad p_1\ \texttt{AND}\ p_2\ \texttt{AND}\ p_3\ \texttt{AND}\ p_4 \\
& \qquad\quad \texttt{AND}\ S_1.A_1 = S_2.B_1\ \texttt{AND}\ S_1.A_2 = S_2.B_1
\end{aligned}
$$

$$
\begin{aligned}
q_k^7 :\ & \texttt{SELECT}\ B_2, B_3, C_1, C_2 \\
& \texttt{FROM}\quad S_2, S_3 \\
& \texttt{WHERE}\quad p_3\ \texttt{AND}\ p_4\ \texttt{AND}\ p_5\ \texttt{AND}\ p_6 \\
& \qquad\quad \texttt{AND}\ S_3.c_1 = S_2.B_1
\end{aligned}
$$

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 $\mathcal{Q} = \{Q_k\}$ 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 inter-schema 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 $S$ and attribute $B$ of a target GCS relation $T$ (in the notation we used in this section we have $v = \langle S.A \approx T.B, p, s \rangle$). Then the source query specifies how to retrieve $S.A$ and the target query specifies how to obtain $T.B$. The GLAV mapping, then, is a relationship between these two queries.

An algorithm to accomplish this [Popa et al., 2002] also starts, as above, with a source schema, a target schema, and $\mathcal{M}$, and "discovers" mappings that satisfy both the source and the target schema semantics. The 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 $\mathcal{M}$ 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 $\mathcal{S}$ and target schemas $\mathcal{T}$, and $\mathcal{M}$ and performs the following two steps:

- It examines intra-schema semantics within the $\mathcal{S}$ and $\mathcal{T}$ separately and produces for each a set of *logical relations* that are semantically consistent.
- It then interprets inter-schema correspondences $\mathcal{M}$ in the context of logical relations generated in Step 1 and produces a set of queries into $\mathcal{Q}$ that are semantically consistent with $\mathcal{T}$.

### *4.4.2 Mapping Maintenance*

In dynamic environments where schemas evolve over time, schema mappings can be made invalid as the result of structural or constraint changes made to the schemas.

Thus, the detection of invalid/inconsistent schema mappings and the adaptation of such schema mappings to new schema structures/constraints becomes 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, increases. 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 intra-schema 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.

### 4.4.2.1 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 [Velegrakis et al., 2004], 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 [McCann et al., 2005]). 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.

#### 4.4.2.2 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 [Velegrakis et al., 2004]. These can be broadly described as *fixed rule approaches* that define a re-mapping 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).

## 4.5 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 are 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 are 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 [Rahm and Do, 2000]. 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 are 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 [Raman and Hellerstein, 2001], or define more complicated schema transformation through a generic “map” operator [Galhardas et al., 2001] that takes a single relation and produces one ore 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 [Raman and Hellerstein, 2001], 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 [Galhardas et al., 2001], as well as basic operators to find duplicates and eliminate them (this has long been known as the purge/merge problem [Hernández and Stolfo, 1998]). 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 (e.g., [Chaudhuri et al., 2003]).

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 needs to be done online as data are retrieved from the sources. The performance of data cleaning is, of course, more critical in the latter case. In fact, the performance and scalability concerns in the latter systems have resulted in proposals where data cleaning is forfeited in favor of querying that is tolerant to conflicts [Yan and Özsu, 1999].

## 4.6 Conclusion

In this chapter we discussed the bottom-up database design process, which we called database integration. This 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 on schema matching, or data cleaning, or schema mapping. There is a serious lack of research that considers end-to-end methodology for database integration. The lack of a methodology is made more serious by the fact that each of these research activities work on different assumptions related to data models, types of heterogeneities and so on. A notable exception is the work of Bernstein and Melnik [2007], which provides the beginnings of a comprehensive "end-to-end" methodology. This is probably the most important topic that requires attention.

A related concept that has received considerable discussion in literature is *data exchange*. This 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 [Doan et al., 2010]. However, for most of the discussions of this chapter, this is not a major concern.

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) [Halevy et al., 2003; Somani et al., 2002]. In next generation systems, ability to handle both types of data will be increasingly important.

Another issue that we ignored in this chapter is interoperability when a GCS does not exist or cannot be specified. As we discussed in Chapter 1, there have been early objections to interoperable access to multiple data sources through a GCS, arguing instead that the languages should provide facilities to access multiple heterogeneous sources without requiring a GCS. The issue becomes critical in the modern 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 Chapter 16.

## 4.7 Bibliographic Notes

A large volume of literature exists on the topic of this chapter. The work goes back to early 1980's 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].

There is an upcoming book on this topic that provides the broadest coverage of the subject [Doan et al., 2010]. There are a number of recent overview papers on the topic. Bernstein and Melnik [2007] provides 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] reviews the data integration work in the 1990's, focusing on the Information Manifold system [Levy et al., 1996c], 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, an 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. Doan and Halevy [2005] is another very good overview of the various schema matching techniques. They propose a different, and simpler, classification of the techniques as rule-based, learning-based, and combined.

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, so we will discuss them in Chapter 9. Examples of LAV approaches are described in the papers [Duschka and Genesereth, 1997; Levy et al., 1996a; 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 and those that focus on semantic heterogeneity are [Dayal and Hwang, 1984; Kim and Seo, 1991; Breitbart et al., 1986; Krishnamurthy et al., 1991] [Hull, 1997; Ouksel and Sheth, 1999; Kashyap and Sheth, 1996; Bright et al., 1994; Ceri and Widom, 1993]. We should note that this list is seriously incomplete.

More recent works in schema matching are surveyed by Rahm and Bernstein [2001] and Doan and Halevy [2005]. In particular, Rahm and Bernstein [2001] gives a very nice comparison of various proposals.

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, 2003b,a], DIPE, which is an earlier version of this system [Palopoli et al., 1999], TranSCM [Milo and Zohar, 1998], ARTEMIS [Bergamaschi et al., 2001], similarity flooding [Melnik et al., 2002], CUPID [Madhavan et al., 2001], and COMA [Do and Rahm, 2002].

## Exercises

**Problem 4.1.** 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 4.2.** 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 4.3 (*).** Give an algorithm to convert a relational schema to an entity-relationship one.

**Problem 4.4 (**).** Consider the two databases given in Figures 4.13 and 4.14 and described below. Design a global conceptual schema as a union of the two databases by first translating them into the E-R model.

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. 4.13** Road Race Database

Figure 4.13 describes a relational race database used by organizers of road races and Figure 4.14 describes an entity-relationship database used by a shoe manufacturer. The semantics of each of these database schemas is discussed below. Figure 4.13 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.

**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.

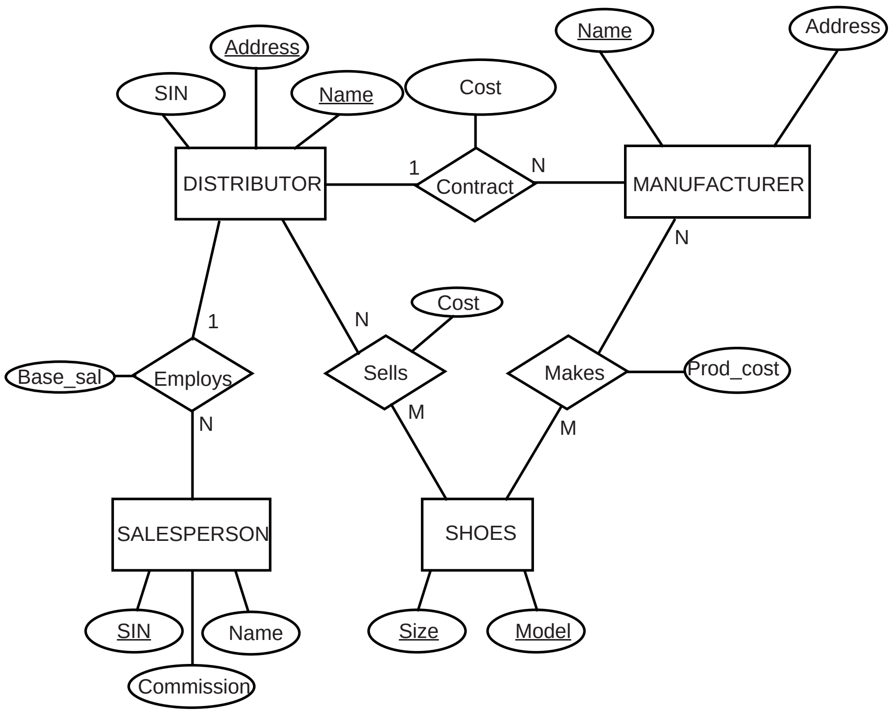


**Fig. 4.14** Sponsor Database

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

**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 4.5 (*).** 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 that specifies possible fields that a course can blong 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 4.6.** 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) contribute 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 4.7.** 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 4.8.** 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 | $\longrightarrow$ | ARTICLE |
| Id: int PRIMARY KEY | $\longrightarrow$ | Key: varchar(255) PRIMARY KEY |
| Name: varchar(255) | $\longrightarrow$ | Title: varchar(255) |
| DeliveryPrice: float | $\longrightarrow$ | Price: real |
| Description: varchar(8000) | $\longrightarrow$ | 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 4.9.** 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 4.10 (*).** Figure 4.15 illustrates the schema of three different data sources:

- 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.
The sources are defined as follows:
MyGroup

- Publication

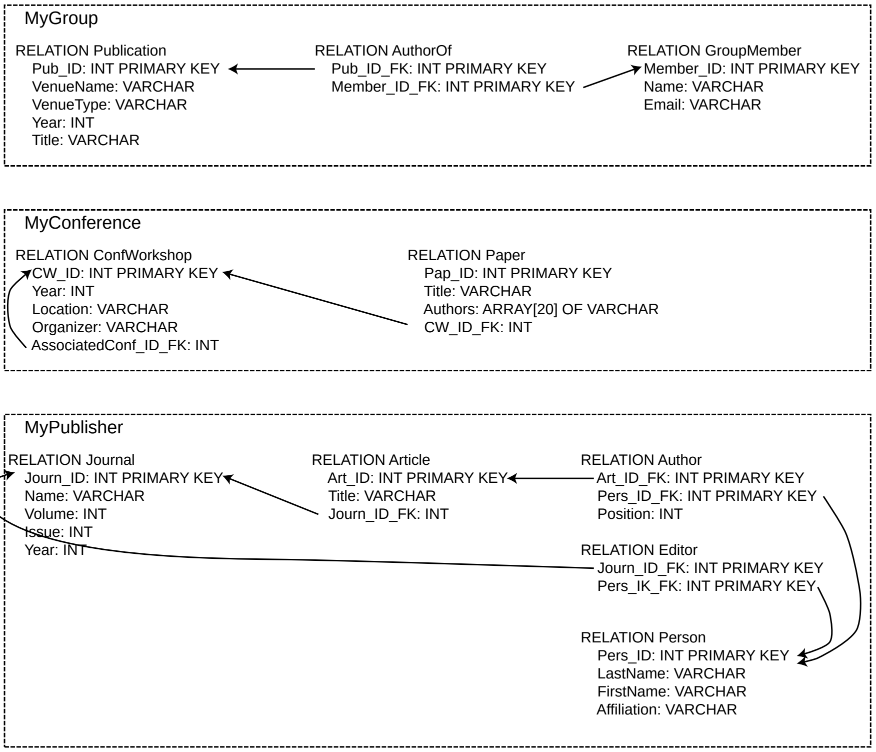


**Fig. 4.15** Figure for Exercise 10

  - 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
    - 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 4.11 (*).** Figure 4.16 illustrates (using a simplified SQL syntax) two sources $S_1$ and $S_2$. $S_1$ has two relations, Course and Tutor, and $S_2$ 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 $S_1$.

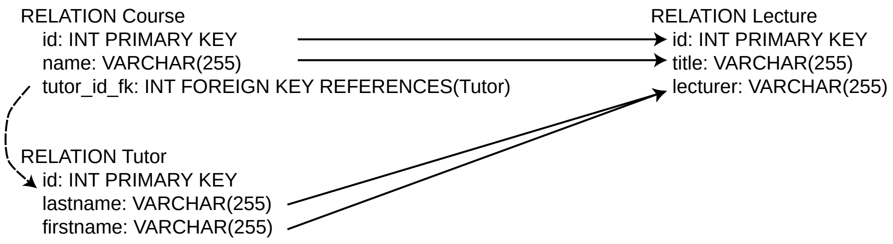


**Fig. 4.16** Figure for Exercise 11

The following are four schema mappings (represented as SQL queries) to transform $S_1$'s data into $S_2$.

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 $S_1$ transformed)?
**(c)** Does the mapping potentially violate key constraints?

**Problem 4.12 (*).** 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 4.13 (**).** 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.

# Chapter 5
# Data and Access Control

An important requirement of a centralized or a distributed DBMS is the ability to support semantic data control, i.e., data and access control using high-level semantics. Semantic data control typically includes view management, security control, and semantic integrity control. Informally, these functions must ensure that *authorized* users perform *correct* operations on the database, contributing to the maintenance of database integrity. The functions necessary for maintaining the physical integrity of the database in the presence of concurrent accesses and failures are studied separately in Chapters 10 through 12 in the context of transaction management. In the relational framework, semantic data control can be achieved in a uniform fashion. Views, security constraints, and semantic integrity constraints can be defined as rules that the system automatically enforces. The violation of some rules by a user program (a set of database operations) generally implies the rejection of the effects of that program (e.g., undoing its updates) or propagating some effects (e.g., updating related data) to preserve the database integrity.

The definition of the rules for controlling data manipulation is part of the administration of the database, a function generally performed by a database administrator (DBA). This person is also in charge of applying the organizational policies. Well-known solutions for semantic data control have been proposed for centralized DBMSs. In this chapter we briefly review the centralized solution to semantic data control, and present the special problems encountered in a distributed environment and solutions to these problems. The cost of enforcing semantic data control, which is high in terms of resource utilization in a centralized DBMS, can be prohibitive in a distributed environment.

Since the rules for semantic data control must be stored in a catalog, the management of a distributed directory (also called a catalog) is also relevant in this chapter. We discussed directories in Section 3.5. Remember that the directory of a distributed DBMS is itself a distributed database. There are several ways to store semantic data control definitions, according to the way the directory is managed. Directory information can be stored differently according to its type; in other words, some information might be fully replicated whereas other information might be distributed. For example, information that is useful at compile time, such as security control information, could be replicated. In this chapter we emphasize the impact of directory management on the performance of semantic data control mechanisms.

This chapter is organized as follows. View management is the subject of Section 5.1. Security control is presented in Section 5.2. Finally, semantic integrity control is treated in Section 5.3. For each section we first outline the solution in a centralized DBMS and then give the distributed solution, which is often an extension of the centralized one, although more difficult.

## 5.1 View Management

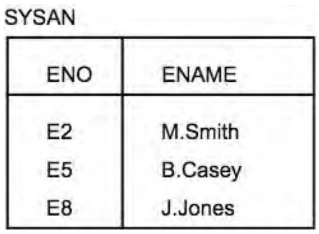
One of the main advantages of the relational model is that it provides full logical data independence. As introduced in Chapter 1, external schemas enable user groups to have their particular *view* of the database. In a relational system, a view is a *virtual relation*, defined as the result of a query on *base relations* (or real relations), but not materialized like a base relation, which is stored in the database. A view is a dynamic window in the sense that it reflects all updates to the database. An external schema can be defined as a set of views and/or base relations. Besides their use in external schemas, views are useful for ensuring data security in a simple way. By selecting a subset of the database, views *hide* some data. If users may only access the database through views, they cannot see or manipulate the hidden data, which are therefore secure.

In the remainder of this section we look at view management in centralized and distributed systems as well as the problems of updating views. Note that in a distributed DBMS, a view can be derived from distributed relations, and the access to a view requires the execution of the distributed query corresponding to the view definition. An important issue in a distributed DBMS is to make view materialization efficient. We will see how the concept of materialized views helps in solving this problem, among others, but requires efficient techniques for materialized view maintenance.

### *5.1.1 Views in Centralized DBMSs*

Most relational DBMSs use a view mechanism where a view is a relation derived from base relations as the result of a relational query (this was first proposed within the INGRES [Stonebraker, 1975] and System R [Chamberlin et al., 1975] projects). It is defined by associating the name of the view with the retrieval query that specifies it.

*Example 5.1.* The view of system analysts (SYSAN) derived from relation EMP (ENO,ENAME,TITLE), can be defined by the following SQL query:



SYSAN

| ENO | ENAME |
|---|---|
| E2 | M.Smith |
| E5 | B.Casey |
| E8 | J.Jones |

**Fig. 5.1** Relation Corresponding to the View SYSAN

```
CREATE VIEW   SYSAN(ENO, ENAME)
AS      SELECT ENO, ENAME
        FROM   EMP
        WHERE  TITLE = "Syst. Anal."
```

♦

The single effect of this statement is the storage of the view definition in the catalog. No other information needs to be recorded. Therefore, the result of the query defining the view (i.e., a relation having the attributes ENO and ENAME for the system analysts as shown in Figure 5.1) is *not* produced. However, the view SYSAN can be manipulated as a base relation.

*Example 5.2.* The query

> "Find the names of all the system analysts with their project number and responsibility(ies)"

involving the view SYSAN and relation ASG(ENO,PNO,RESP,DUR) can be expressed as

```
SELECT ENAME, PNO, RESP
FROM   SYSAN, ASG
WHERE  SYSAN.ENO = ASG.ENO
```

♦

Mapping a query expressed on views into a query expressed on base relations can be done by *query modification* [Stonebraker, 1975]. With this technique the variables are changed to range on base relations and the query qualification is merged (ANDed) with the view qualification.

*Example 5.3.* The preceding query can be modified to

```
SELECT ENAME, PNO, RESP
FROM   EMP, ASG
WHERE  EMP.ENO = ASG.ENO
AND    TITLE = "Syst. Anal."
```

The result of this query is illustrated in Figure 5.2. ♦

The modified query is expressed on base relations and can therefore be processed by the query processor. It is important to note that view processing can be done at compile time. The view mechanism can also be used for refining the access controls to include subsets of objects. To specify any user from whom one wants to hide data, the keyword USER generally refers to the logged-on user identifier.

| ENAME | PNO | RESP |
|---|---|---|
| M.Smith | P1 | Analyst |
| M.Smith | P2 | Analyst |
| B.Casey | P3 | Manager |
| J.Jones | P4 | Manager |

**Fig. 5.2** Result of Query involving View SYSAN

*Example 5.4.* The view ESAME restricts the access by any user to those employees having the same title:

```
CREATE VIEW   ESAME
AS     SELECT *
       FROM   EMP E1, EMP E2
       WHERE  E1.TITLE = E2.TITLE
       AND    E1.ENO = USER
```

In the view definition above, * stands for “all attributes” and the two tuple variables (E1 and E2) ranging over relation EMP are required to express the join of one tuple of EMP (the one corresponding to the logged-on user) with all tuples of EMP based on the same title. For example, the following query issued by the user J. Doe,

```
SELECT *
FROM   ESAME
```

returns the relation of Figure 5.3. Note that the user J. Doe also appears in the result. If the user who creates ESAME is an electrical engineer, as in this case, the view represents the set of all electrical engineers. ♦

| ENO | ENAME | TITLE |
|---|---|---|
| E1 | J. Doe | Elect. Eng |
| E2 | L. Chu | Elect. Eng |

**Fig. 5.3** Result of Query on View ESAME

Views can be defined using arbitrarily complex relational queries involving selection, projection, join, aggregate functions, and so on. All views can be interrogated as base relations, but not all views can be manipulated as such. Updates through views can be handled automatically only if they can be propagated correctly to the base relations. We can classify views as being updatable and not updatable. A view is updatable only if the updates to the view can be propagated to the base relations without ambiguity. The view SYSAN above is updatable; the insertion, for example, of a new system analyst ⟨201, Smith⟩ will be mapped into the insertion of a new employee ⟨201, Smith, Syst. Anal.⟩. If attributes other than TITLE were hidden by the view, they would be assigned *null values*.

*Example 5.5.* The following view, however, is not updatable:

```
CREATE VIEW   EG(ENAME, RESP)
AS     SELECT DISTINCT ENAME, RESP
       FROM   EMP, ASG
       WHERE  EMP.ENO = ASG.ENO
```

The deletion, for example, of the tuple ⟨Smith, Analyst⟩ cannot be propagated, since it is ambiguous. Deletions of Smith in relation EMP or analyst in relation ASG are both meaningful, but the system does not know which is correct. ♦

Current systems are very restrictive about supporting updates through views. Views can be updated only if they are derived from a single relation by selection and projection. This precludes views defined by joins, aggregates, and so on. However, it is theoretically possible to automatically support updates of a larger class of views [Bancilhon and Spyratos, 1981; Dayal and Bernstein, 1978; Keller, 1982]. It is interesting to note that views derived by join are updatable if they include the keys of the base relations.

### 5.1.2 Views in Distributed DBMSs

The definition of a view is similar in a distributed DBMS and in centralized systems. However, a view in a distributed system may be derived from fragmented relations stored at different sites. When a view is defined, its name and its retrieval query are stored in the catalog.

Since views may be used as base relations by application programs, their definition should be stored in the directory in the same way as the base relation descriptions. Depending on the degree of site autonomy offered by the system [Williams et al., 1982], view definitions can be centralized at one site, partially duplicated, or fully duplicated. In any case, the information associating a view name to its definition site should be duplicated. If the view definition is not present at the site where the query is issued, remote access to the view definition site is necessary.

The mapping of a query expressed on views into a query expressed on base relations (which can potentially be fragmented) can also be done in the same way as in centralized systems, that is, through query modification. With this technique, the qualification defining the view is found in the distributed database catalog and then merged with the query to provide a query on base relations. Such a modified query is a *distributed query*, which can be processed by the distributed query processor (see Chapter 6). The query processor maps the distributed query into a query on physical fragments.

In Chapter 3 we presented alternative ways of fragmenting base relations. The definition of fragmentation is, in fact, very similar to the definition of particular views. It is possible to manage views and fragments using a unified mechanism [Adiba, 1981]. This is based on the observation that views in a distributed DBMS can be defined with rules similar to fragment definition rules. Furthermore, replicated data can be handled in the same way. The value of such a unified mechanism is to facilitate distributed database administration. The objects manipulated by the database administrator can be seen as a hierarchy where the leaves are the fragments from which relations and views can be derived. Therefore, the DBA may increase locality of reference by making views in one-to-one correspondence with fragments. For example, it is possible to implement the view SYSAN illustrated in Example 5.1 by a fragment at a given site, provided that most users accessing the view SYSAN are at the same site.

Evaluating views derived from distributed relations may be costly. In a given organization it is likely that many users access the same view which must be recomputed for each user. We saw in Section 5.1.1 that view derivation is done by merging the view qualification with the query qualification. An alternative solution is to avoid view derivation by maintaining actual versions of the views, called *materialized views*. A *materialized view* stores the tuples of a view in a database relation, like the other database tuples, possibly with indices. Thus, access to a materialized view is much faster than deriving the view, in particular, in a distributed DBMS where base relations can be remote. Introduced in the early 1980s [Adiba and Lindsay, 1980], materialized views have since gained much interest in the context of data warehousing to speed up On Line Analytical Processing (OLAP) applications [Gupta and Mumick, 1999c]. Materialized views in data warehouses typically involve aggregate (such as SUM and COUNT) and grouping (GROUP BY) operators because they provide compact database summaries. Today, all major database products support materialized views.

*Example 5.6.* The following view over relation PROJ(PNO,PNAME,BUDGET,LOC) gives, for each location, the number of projects and the total budget.

```
CREATE VIEW     PL(LOC, NBPROJ, TBUDGET)
AS     SELECT   LOC, COUNT(*),SUM(BUDGET)
       FROM     PROJ
       GROUP BY LOC
```

♦

### *5.1.3 Maintenance of Materialized Views*

A materialized view is a copy of some base data and thus must be kept consistent with that base data which may be updated. *View maintenance* is the process of updating (or refreshing) a materialized view to reflect the changes made to the base data. The issues related to view materialization are somewhat similar to those of database replication which we will address in Chapter 13. However, a major difference is that materialized view expressions, in particular, for data warehousing, are typically more complex than replica definitions and may include join, group by and aggregate operators. Another major difference is that database replication is concerned with more general replication configurations, e.g., with multiple copies of the same base data at multiple sites.

A view maintenance policy allows a DBA to specify *when* and *how* a view should be refreshed. The first question (when to refresh) is related to consistency (between the view and the base data) and efficiency. A view can be refreshed in two modes: *immediate* or *deferred*. With the immediate mode, a view is refreshed immediately as part as the transaction that updates base data used by the view. If the view and the base data are managed by different DBMSs, possibly at different sites, this requires the use of a distributed transaction, for instance, using the two-phase commit (2PC) protocol (see Chapter 12). The main advantages of immediate refreshment are that the view is always consistent with the base data and that read-only queries can be fast. However, this is at the expense of increased transaction time to update both the base data and the views within the same transactions. Furthermore, using distributed transactions may be difficult.

In practice, the deferred mode is preferred because the view is refreshed in separate (refresh) transactions, thus without performance penalty on the transactions that update the base data. The refresh transactions can be triggered at different times: *lazily*, just before a query is evaluated on the view; *periodically*, at predefined times, e.g., every day; or *forcedly*, after a predefined number of updates to the base data. Lazy refreshment enables queries to see the latest consistent state of the base data but at the expense of increased query time to include the refreshment of the view. Periodic and forced refreshment allow queries to see views whose state is not consistent with the latest state of the base data. The views managed with these strategies are also called *snapshots* [Adiba, 1981; Blakeley et al., 1986].

The second question (how to refresh a view) is an important efficiency issue. The simplest way to refresh a view is to recompute it from scratch using the base data. In some cases, this may be the most efficient strategy, e.g., if a large subset of the base data has been changed. However, there are many cases where only a small subset of view needs to be changed. In these cases, a better strategy is to compute the view *incrementally*, by computing only the changes to the view. Incremental view maintenance relies on the concept of differential relation. Let $u$ be an update of relation $R$. $R^+$ and $R^-$ are *differential relations* of $R$ by $u$, where $R^+$ contains the tuples inserted by $u$ into $R$, and $R^-$ contains the tuples of $R$ deleted by $u$. If $u$ is an insertion, $R^-$ is empty. If $u$ is a deletion, $R^+$ is empty. Finally, if $u$ is a modification, relation $R$ can be obtained by computing $(R - R^-) \cup R^+$. Similarly, a materialized view $V$ can be refreshed by computing $(V - V^-) \cup V^+$. Computing the changes to the view, i.e., $V^+$ and $V^-$, may require using the base relations in addition to differential relations.

*Example 5.7.* Consider the view EG of Example 5.5 which uses relations EMP and ASG as base data and assume its state is derived from that of Example 3.1, so that EG has 9 tuples (see Figure 5.4). Let EMP$^+$ consist of one tuple ⟨E9, B. Martin, Programmer⟩ to be inserted in EMP, and ASG$^+$ consist of two tuples ⟨E4, P3, Programmer, 12⟩ and ⟨E9, P3, Programmer, 12⟩ to be inserted in ASG. The changes to the view EG can be computed as:

```
EG+ =   (SELECT  ENAME, RESP
        FROM     EMP, ASG+
        WHERE    EMP.ENO = ASG+.ENO)
                 UNION
        (SELECT  ENAME, RESP
        FROM     EMP+, ASG
        WHERE    EMP+.ENO = ASG.ENO)
                 UNION
        (SELECT  ENAME, RESP
        FROM     EMP+, ASG+
        WHERE    EMP+.ENO = ASG+.ENO)
```

which yields tuples ⟨B. Martin, Programmer⟩ and ⟨J. Miller, Programmer⟩. Note that integrity constraints would be useful here to avoid useless work (see Section 5.3.2). Assuming that relations EMP and ASG are related by a referential constraint that says that ENO in ASG must exist in EMP, the second SELECT statement is useless as it produces an empty relation. ♦

EG

| ENAME | RESP |
|---|---|
| J. Doe | Manager |
| M. Smith | Analyst |
| A. Lee | Consultant |
| A. Lee | Engineer |
| J. Miller | Programmer |
| B. Casey | Manager |
| L. Chu | Manager |
| R. Davis | Engineer |
| J.Jones | Manager |

**Fig. 5.4** State of View EG

Efficient techniques have been devised to perform incremental view maintenance using both the materialized views and the base relations. The techniques essentially differ in their views' expressiveness, their use of integrity constraints, and the way they handle insertion and deletion. Gupta and Mumick [1999a] classify these techniques along the view expressiveness dimension as non-recursive views, views involving outerjoins, and recursive views. For non-recursive views, i.e., select-project-join (SPJ) views that may have duplicate elimination, union and aggregation, an elegant solution is the counting algorithm [Gupta et al., 1993]. One problem stems from the fact that individual tuples in the view may be derived from several tuples in the base relations, thus making deletion in the view difficult. The basic idea of the counting algorithm is to maintain a count of the number of derivations for each tuple in the view, and to increment (resp. decrement) tuple counts based on insertions (resp. deletions); a tuple in the view of which count is zero can then be deleted.

*Example 5.8.* Consider the view EG in Figure 5.4. Each tuple in EG has one derivation (i.e., a count of 1) except tuple ⟨M. Smith, Analyst⟩ which has two (i.e., a count of 2). Assume now that tuples ⟨E2, P1, Analyst, 24⟩ and ⟨E3, P3, Consultant, 10⟩ are deleted from ASG. Then only tuple ⟨A. Lee, Consultant⟩ needs to be deleted from EG. ♦

We now present the basic counting algorithm for refreshing a view $V$ defined over two relations $R$ and $S$ as a query $q(R,S)$. Assuming that each tuple in $V$ has an associated derivation count, the algorithm has three main steps (see Algorithm 5.1). First, it applies the view differentiation technique to formulate the differential views $V^+$ and $V^-$ as queries over the view, the base relations, and the differential relations. Second, it computes $V^+$ and $V^-$ and their tuple counts. Third, it applies the changes $V^+$ and $V^-$ in $V$ by adding positive counts and subtracting negative counts, and deleting tuples with a count of zero.

**Algorithm 5.1**: COUNTING Algorithm

```
Input: $V$: view defined as $q(R,S)$; $R$, $S$: relations; $R^+$, $R^-$: changes to $R$
begin
    $V^+ = q^+(V, R^+, R, S)$;
    $V^- = q^-(V, R^-, R, S)$ ;
    compute $V^+$ with positive counts for inserted tuples;
    compute $V^-$ with negative counts for deleted tuples;
    compute $(V - V^-) \cup V^+$ by adding positive counts and substracting
    negative counts deleting each tuple in $V$ with count = 0;
end
```

The counting algorithm is optimal since it computes exactly the view tuples that are inserted or deleted. However, it requires access to the base relations. This implies that the base relations be maintained (possibly as replicas) at the sites of the materialized view. To avoid accessing the base relations so the view can be stored at a different site, the view should be maintainable using only the view and the differential relations. Such views are called *self-maintainable* [Gupta et al., 1996].