Chapter 3
Distributed Database Design

The design of a distributed computer system involves making decisions on the placement of *data* and *programs* across the sites of a computer network, as well as possibly designing the network itself. In the case of distributed DBMSs, the distribution of applications involves two things: the distribution of the distributed DBMS software and the distribution of the application programs that run on it. Different architectural models discussed in Chapter 1 address the issue of application distribution. In this chapter we concentrate on distribution of data.

It has been suggested that the organization of distributed systems can be investigated along three orthogonal dimensions [Levin and Morgan, 1975] (Figure 3.1):

1. Level of sharing
2. Behavior of access patterns
3. Level of knowledge on access pattern behavior

In terms of the level of sharing, there are three possibilities. First, there is *no sharing*: each application and its data execute at one site, and there is no communication with any other program or access to any data file at other sites. This characterizes the very early days of networking and is probably not very common today. We then find the level of *data sharing*; all the programs are replicated at all the sites, but data files are not. Accordingly, user requests are handled at the site where they originate and the necessary data files are moved around the network. Finally, in *data-plus-program sharing*, both data and programs may be shared, meaning that a program at a given site can request a service from another program at a second site, which, in turn, may have to access a data file located at a third site.

Levin and Morgan draw a distinction between data sharing and data-plus-program sharing to illustrate the differences between homogeneous and heterogeneous distributed computer systems. They indicate, correctly, that in a heterogeneous environment it is usually very difficult, and sometimes impossible, to execute a given program on different hardware under a different operating system. It might, however, be possible to move data around relatively easily.

M.T. Özsu and P. Valduriez, *Principles of Distributed Database Systems: Third Edition*,
DOI 10.1007/978-1-4419-8834-8_3, © Springer Science+Business Media, LLC 2011

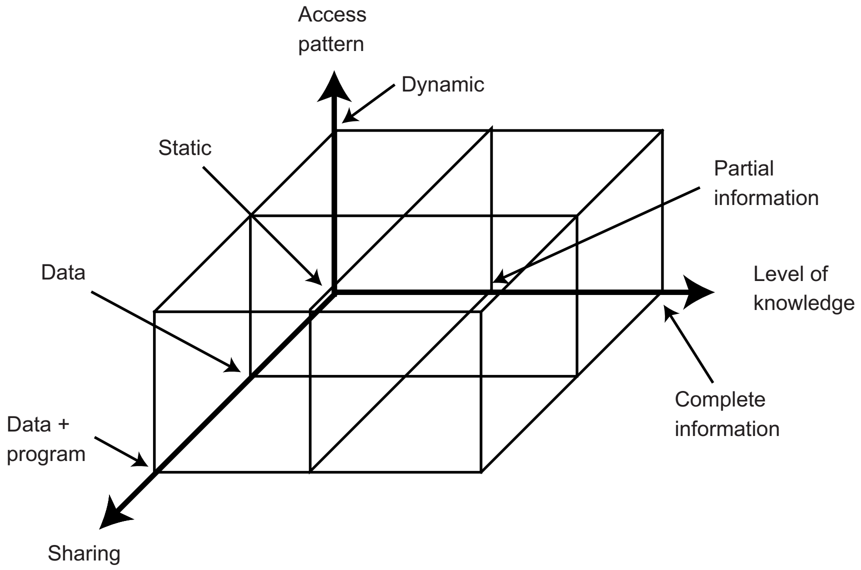


Fig. 3.1 Framework of Distribution

Along the second dimension of access pattern behavior, it is possible to identify two alternatives. The access patterns of user requests may be *static*, so that they do not change over time, or *dynamic*. It is obviously considerably easier to plan for and manage the static environments than would be the case for dynamic distributed systems. Unfortunately, it is difficult to find many real-life distributed applications that would be classified as static. The significant question, then, is not whether a system is static or dynamic, but how dynamic it is. Incidentally, it is along this dimension that the relationship between the distributed database design and query processing is established (refer to Figure 1.7).

The third dimension of classification is the level of knowledge about the access pattern behavior. One possibility, of course, is that the designers do not have any information about how users will access the database. This is a theoretical possibility, but it is very difficult, if not impossible, to design a distributed DBMS that can effectively cope with this situation. The more practical alternatives are that the designers have *complete information*, where the access patterns can reasonably be predicted and do not deviate significantly from these predictions, or *partial information*, where there are deviations from the predictions.

The distributed database design problem should be considered within this general framework. In all the cases discussed, except in the no-sharing alternative, new problems are introduced in the distributed environment which are not relevant in a centralized setting. In this chapter it is our objective to focus on these unique problems.

Two major strategies that have been identified for designing distributed databases are the *top-down approach* and the *bottom-up approach* [Ceri et al., 1987]. As the names indicate, they constitute very different approaches to the design process. Top-down approach is more suitable for tightly integrated, homogeneous distributed DBMSs, while bottom-up design is more suited to multidatabases (see the classification in Chapter 1). In this chapter, we focus on top-down design and defer bottom-up to the next chapter.

3.1 Top-Down Design Process

A framework for top-down design process is shown in Figure 3.2. The activity begins with a requirements analysis that defines the environment of the system and "elicits both the data and processing needs of all potential database users" [Yao et al., 1982a]. The requirements study also specifies where the final system is expected to stand with respect to the objectives of a distributed DBMS as identified in Section 1.4. These objectives are defined with respect to performance, reliability and availability, economics, and expandability (flexibility).

The requirements document is input to two parallel activities: view design and conceptual design. The *view design* activity deals with defining the interfaces for end users. The *conceptual design*, on the other hand, is the process by which the enterprise is examined to determine entity types and relationships among these entities. One can possibly divide this process into two related activity groups [Davenport, 1981]: ~~entity analysis and functional analysis. *Entity analysis* is concerned with determining the entities, their attributes, and the relationships among them. *Functional analysis*, on the other hand, is concerned with determining the fundamental functions with which the modeled enterprise is involved~~. The results of these two steps need to be cross-referenced to get a better understanding of which functions deal with which entities.

There is a relationship between the conceptual design and the view design. In one sense, the conceptual design can be interpreted as being an integration of user views. Even though this *view integration* activity is very important, the conceptual model should support not only the existing applications, but also future applications. View integration should be used to ensure that entity and relationship requirements for all the views are covered in the conceptual schema.

In conceptual design and view design activities the user needs to specify the data entities and must determine the applications that will run on the database as well as statistical information about these applications. Statistical information includes the specification of the frequency of user applications, the volume of various information, and the like. Note that from the conceptual design step comes the definition of global conceptual schema discussed in Section 1.7. We have not yet considered the implications of the distributed environment; in fact, up to this point, the process is identical to that in a centralized database design.

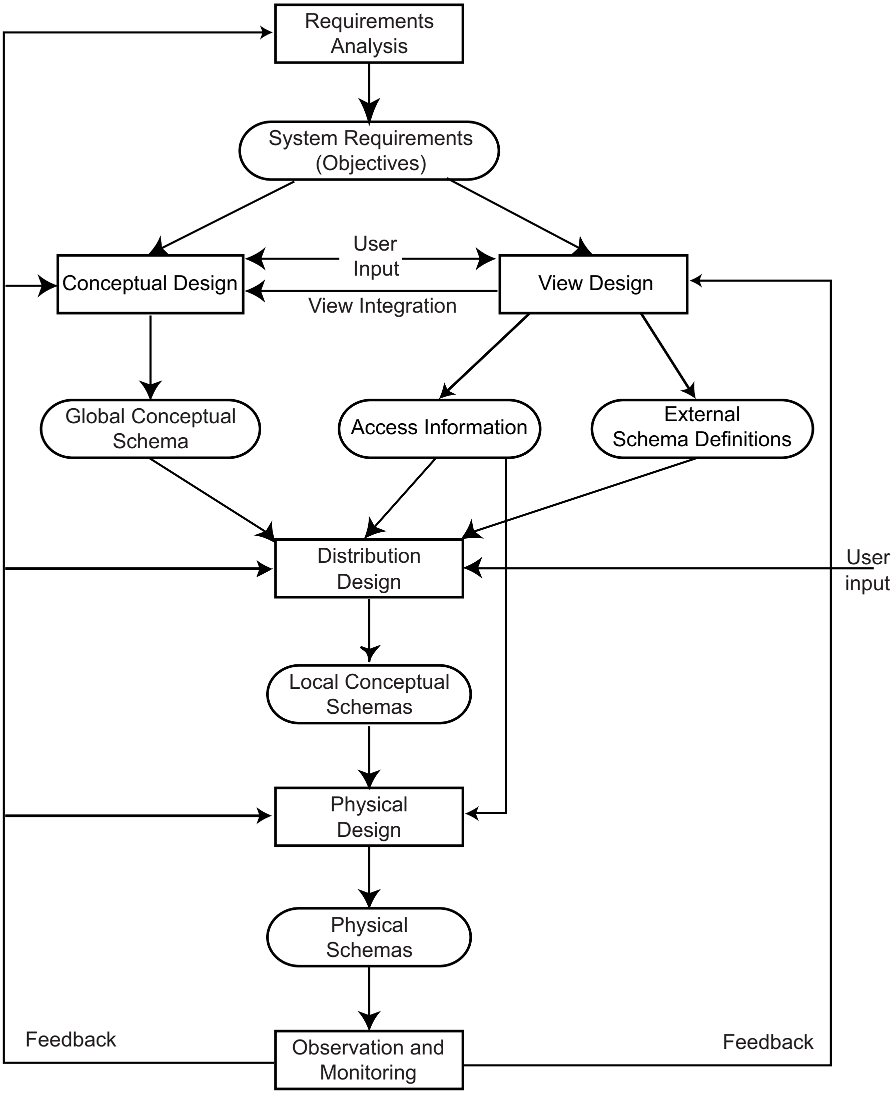


Fig. 3.2 Top-Down Design Process

The global conceptual schema (GCS) and access pattern information collected as a result of view design are inputs to the *distribution design* step. The objective at this stage, which is the focus of this chapter, is to design the local conceptual schemas (LCSs) by distributing the entities over the sites of the distributed system. It is possible, of course, to treat each entity as a unit of distribution. Given that we use the relational model as the basis of discussion in this book, the entities correspond to relations.

~~Rather than distributing relations, it is quite common to divide them into subrelations, called *fragments*, which are then distributed. Thus, the distribution design activity consists of two steps: *fragmentation* and *allocation*. The reason for separating the distribution design into two steps is to better deal with the complexity~~ of the problem. However, this raises other concerns as we discuss at the end of the chapter.

The last step in the design process is the physical design, which maps the local conceptual schemas to the physical storage devices available at the corresponding sites. The inputs to this process are the local conceptual schema and the access pattern information about the fragments in them.

It is well known that design and development activity of any kind is an ongoing process requiring constant monitoring and periodic adjustment and tuning. We have therefore included observation and monitoring as a major activity in this process. Note that one does not monitor only the behavior of the database implementation but also the suitability of user views. The result is some form of feedback, which may result in backing up to one of the earlier steps in the design.

3.2 Distribution Design Issues

In the preceding section we indicated that the relations in a database schema are usually decomposed into smaller fragments, but we did not offer any justification or details for this process. The objective of this section is to fill in these details.

The following set of interrelated questions covers the entire issue. We will therefore seek to answer them in the remainder of this section.

1. Why fragment at all?
2. How should we fragment?
3. How much should we fragment?
4. Is there any way to test the correctness of decomposition?
5. How should we allocate?
6. What is the necessary information for fragmentation and allocation?

3.2.1 Reasons for Fragmentation

From a data distribution viewpoint, there is really no reason to fragment data. After all, in distributed file systems, the distribution is performed on the basis of entire files. In fact, the very early work dealt specifically with the allocation of files to nodes on a computer network. We consider earlier models in Section 3.4.

With respect to fragmentation, the important issue is the appropriate unit of distribution. A relation is not a suitable unit, for a number of reasons. First, application views are usually subsets of relations. Therefore, the locality of accesses of applications is defined not on entire relations but on their subsets. Hence it is only natural to consider subsets of relations as distribution units.

Second, if the applications that have views defined on a given relation reside at different sites, two alternatives can be followed, with the entire relation being the unit of distribution. Either the relation is not replicated and is stored at only one site, or it is replicated at all or some of the sites where the applications reside. The former results in an unnecessarily high volume of remote data accesses. The latter, on the other hand, has unnecessary replication, which causes problems in executing updates (to be discussed later) and may not be desirable if storage is limited.

Finally, the decomposition of a relation into fragments, each being treated as a unit, permits a number of transactions to execute concurrently. In addition, the fragmentation of relations typically results in the parallel execution of a single query by dividing it into a set of subqueries that operate on fragments. Thus fragmentation typically increases the level of concurrency and therefore the system throughput. This form of concurrency, which we refer to as *intraquery concurrency*, is dealt with mainly in Chapters 7 and 8, under query processing.

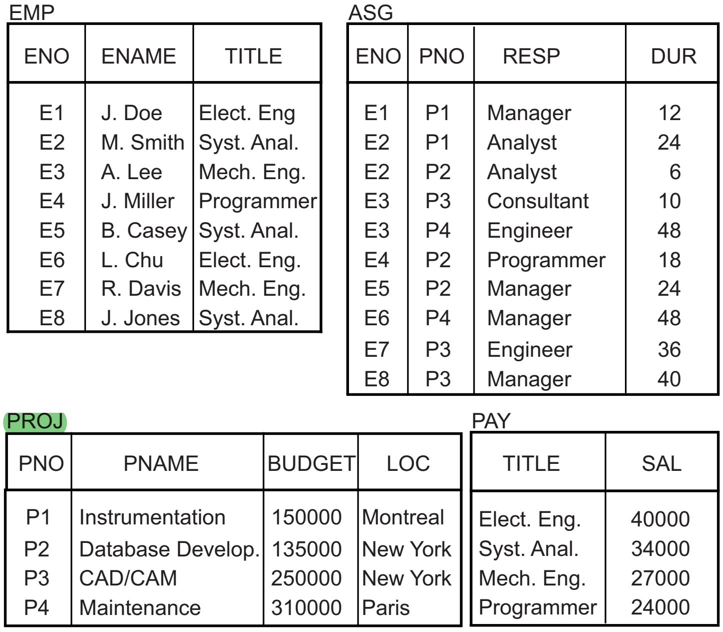
Fragmentation raises difficulties as well. If the applications have conflicting requirements that prevent decomposition of the relation into mutually exclusive fragments, those applications whose views are defined on more than one fragment may suffer performance degradation. It might, for example, be necessary to retrieve data from two fragments and then take their join, which is costly. Minimizing distributed joins is a fundamental fragmentation issue.

The second problem is related to semantic data control, specifically to integrity checking. As a result of fragmentation, attributes participating in a dependency may be decomposed into different fragments that might be allocated to different sites. In this case, even the simpler task of checking for dependencies would result in chasing after data in a number of sites. In Chapter 5 we return to the issue of semantic data control.

3.2.2 *Fragmentation Alternatives*

Relation instances are essentially tables, so the issue is one of finding alternative ways of dividing a table into smaller ones. There are clearly two alternatives for this: dividing it *horizontally* or dividing it *vertically*.

Example 3.1. In this chapter we use a modified version of the relational database scheme developed in Section 2.1. We have added to the PROJ relation a new attribute (LOC) that indicates the place of each project. Figure 3.3 depicts the database instance we will use. Figure 3.4 shows the PROJ relation of Figure 3.3 divided horizontally into two relations. Subrelation PROJ_1 contains information about projects whose



EMP

ENO	ENAME	TITLE
E1	J. Doe	Elect. Eng
E2	M. Smith	Syst. Anal.
E3	A. Lee	Mech. Eng.
E4	J. Miller	Programmer
E5	B. Casey	Syst. Anal.
E6	L. Chu	Elect. Eng.
E7	R. Davis	Mech. Eng.
E8	J. Jones	Syst. Anal.

ASG

ENO	PNO	RESP	DUR
E1	P1	Manager	12
E2	P1	Analyst	24
E2	P2	Analyst	6
E3	P3	Consultant	10
E3	P4	Engineer	48
E4	P2	Programmer	18
E5	P2	Manager	24
E6	P4	Manager	48
E7	P3	Engineer	36
E8	P3	Manager	40

PROJ

PNO	PNAME	BUDGET	LOC
P1	Instrumentation	150000	Montreal
P2	Database Develop.	135000	New York
P3	CAD/CAM	250000	New York
P4	Maintenance	310000	Paris

PAY

TITLE	SAL
Elect. Eng.	40000
Syst. Anal.	34000
Mech. Eng.	27000
Programmer	24000

Fig. 3.3 Modified Example Database

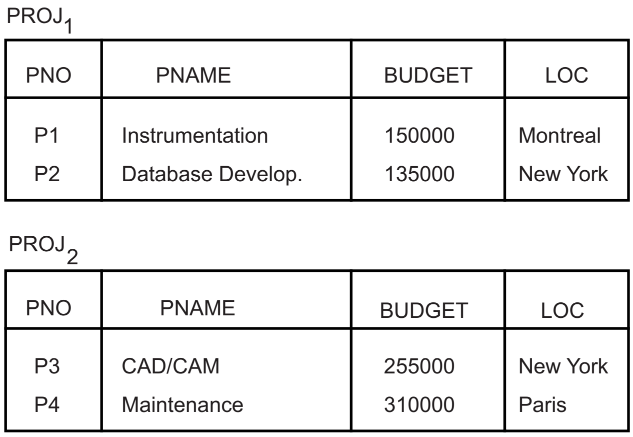
budgets are less than \$200,000, whereas PROJ_2 stores information about projects with larger budgets. ♦

Example 3.2. Figure 3.5 shows the PROJ relation of Figure 3.3 partitioned vertically into two subrelations, PROJ_1 and PROJ_2. PROJ_1 contains only the information about project budgets, whereas PROJ_2 contains project names and locations. It is important to notice that the primary key to the relation (PNO) is included in both fragments. ♦

The fragmentation may, of course, be nested. If the nestings are of different types, one gets *hybrid fragmentation*. Even though we do not treat hybrid fragmentation as a primitive fragmentation strategy, many real-life partitionings may be hybrid.

3.2.3 Degree of Fragmentation

The extent to which the database should be fragmented is an important decision that affects the performance of query execution. In fact, the issues in Section 3.2.1 concerning the reasons for fragmentation constitute a subset of the answers to the question we are addressing here. The degree of fragmentation goes from one extreme, that is, not to fragment at all, to the other extreme, to fragment to the level of



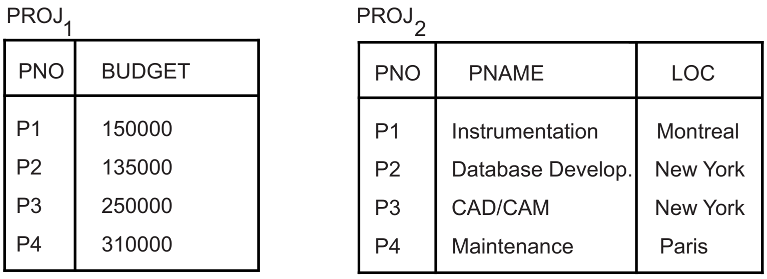
PROJ$_1$

PNO	PNAME	BUDGET	LOC
P1	Instrumentation	150000	Montreal
P2	Database Develop.	135000	New York

PROJ$_2$

PNO	PNAME	BUDGET	LOC
P3	CAD/CAM	255000	New York
P4	Maintenance	310000	Paris

Fig. 3.4 Example of Horizontal Partitioning



PROJ$_1$

PNO	BUDGET
P1	150000
P2	135000
P3	250000
P4	310000

PROJ$_2$

PNO	PNAME	LOC
P1	Instrumentation	Montreal
P2	Database Develop.	New York
P3	CAD/CAM	New York
P4	Maintenance	Paris

Fig. 3.5 Example of Vertical Partitioning

individual tuples (in the case of horizontal fragmentation) or to the level of individual attributes (in the case of vertical fragmentation).

We have already addressed the adverse effects of very large and very small units of fragmentation. What we need, then, is to find a suitable level of fragmentation that is a compromise between the two extremes. Such a level can only be defined with respect to the applications that will run on the database. The issue is, how? In general, the applications need to be characterized with respect to a number of parameters. According to the values of these parameters, individual fragments can be identified. In Section 3.3 we describe how this characterization can be carried out for alternative fragmentations.

3.2.4 Correctness Rules of Fragmentation

We will enforce the following three rules during fragmentation, which, together, ensure that the database does not undergo semantic change during fragmentation.

1. *Completeness.* If a relation instance R is decomposed into fragments $F_R = \{R_1, R_2, \ldots, R_n\}$, each data item that can be found in R can also be found in one or more of R_i's. This property, which is identical to the *lossless decomposition* property of normalization (Section 2.1), is also important in fragmentation since it ensures that the data in a global relation are mapped into fragments without any loss [Grant, 1984]. Note that in the case of horizontal fragmentation, the "item" typically refers to a tuple, while in the case of vertical fragmentation, it refers to an attribute.
2. *Reconstruction.* If a relation R is decomposed into fragments $F_R = \{R_1, R_2, \ldots, R_n\}$, it should be possible to define a relational operator $\triangledown$ such that

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

   The operator $\triangledown$ will be different for different forms of fragmentation; it is important, however, that it can be identified. The reconstructability of the relation from its fragments ensures that constraints defined on the data in the form of dependencies are preserved.
3. *Disjointness.* If a relation R is horizontally decomposed into fragments $F_R = \{R_1, R_2, \ldots, R_n\}$ and data item d_i is in R_j, it is not in any other fragment R_k $(k \neq j)$. This criterion ensures that the horizontal fragments are disjoint. If relation R is vertically decomposed, its primary key attributes are typically repeated in all its fragments (for reconstruction). Therefore, in case of vertical partitioning, disjointness is defined only on the non-primary key attributes of a relation.

3.2.5 Allocation Alternatives

Assuming that the database is fragmented properly, one has to decide on the allocation of the fragments to various sites on the network. When data are allocated, it may either be replicated or maintained as a single copy. The reasons for replication are reliability and efficiency of read-only queries. If there are multiple copies of a data item, there is a good chance that some copy of the data will be accessible somewhere even when system failures occur. Furthermore, read-only queries that access the same data items can be executed in parallel since copies exist on multiple sites. On the other hand, the execution of update queries cause trouble since the system has to ensure that all the copies of the data are updated properly. Hence the decision regarding replication is a trade-off that depends on the ratio of the read-only queries to the 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	Same difficulty	
DIRECTORY MANAGEMENT	Easy or nonexistent	Same difficulty	
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. ♦

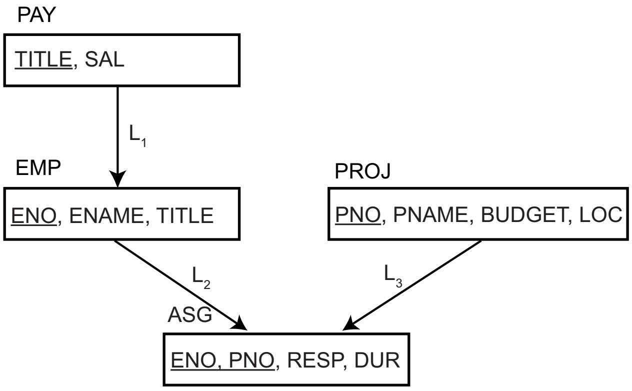


Fig. 3.7 Expression of Relationships Among Relations Using Links

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 PROJ_1 and 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 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.

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

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.

$$\begin{aligned} \text{PROJ}_1 &= \sigma_{\text{LOC=``Montreal''}}(\text{PROJ}) \\ \text{PROJ}_2 &= \sigma_{\text{LOC=``New York''}}(\text{PROJ}) \\ \text{PROJ}_3 &= \sigma_{\text{LOC=``Paris''}}(\text{PROJ}) \end{aligned}$$

♦

PROJ_1

PNO	PNAME	BUDGET	LOC
P1	Instrumentation	150000	Montreal

PROJ_2

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

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 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 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:

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

♦

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.

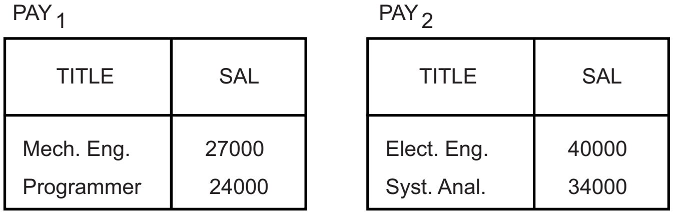
The simple predicates that would be used to partition relation PAY are

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.

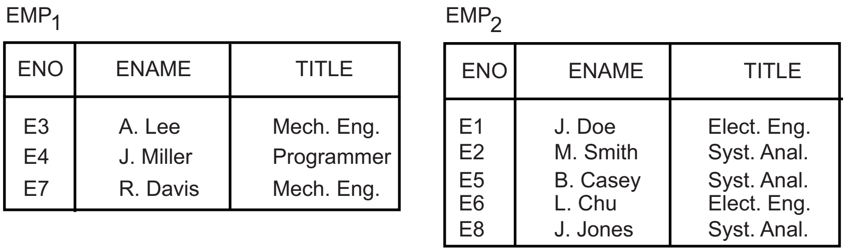
Example 3.12. Consider link L_1 in Figure 3.7, where $owner(L_1)$ = PAY and $member(L_1)$ = EMP. Then we can group engineers into two groups according to their salary: those making less than or equal to $30,000, and those making more than $30,000. The two fragments EMP_1 and EMP_2 are defined as follows:

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

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. ♦



EMP_1

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

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., PAY_1 and 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.

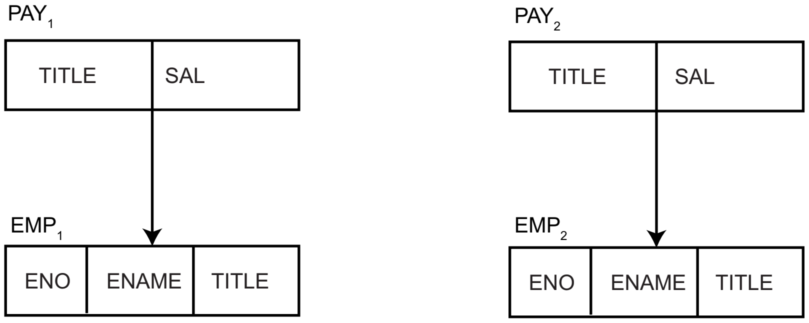


Fig. 3.12 Join Graph Between Fragments

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

Therefore, the derived fragmentation of ASG according to $\{\text{PROJ}_1, \text{PROJ}_2, \text{PROJ}_3\}$ is defined as follows:

$\text{ASG}_1 = \text{ASG} \ltimes \text{PROJ}_1$
$\text{ASG}_2 = \text{ASG} \ltimes \text{PROJ}_3$
$\text{ASG}_3 = \text{ASG} \ltimes \text{PROJ}_4$
$\text{ASG}_4 = \text{ASG} \ltimes \text{PROJ}_6$

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.

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

♦

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→EMP→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 mth 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 $index \leq n$ 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

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

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

(a)

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

(b)

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

(c)

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

(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

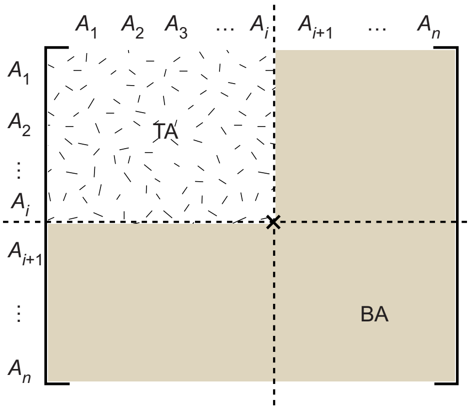


Fig. 3.18 Locating a Splitting Point

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} = $\{$PROJ$_1$,PROJ$_2\}$, where PROJ$_1$ = $\{A_1, A_3\}$ and PROJ$_2$ = $\{A_1, A_2, A_4\}$. Thus

PROJ$_1$ = $\{$PNO, BUDGET$\}$
PROJ$_2$ = $\{$PNO, PNAME, LOC$\}$

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 PROJ$_2$ as well as in 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.

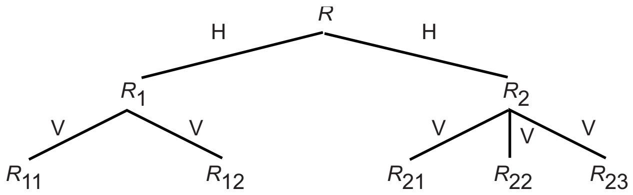


Fig. 3.19 Hybrid Fragmentation

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.

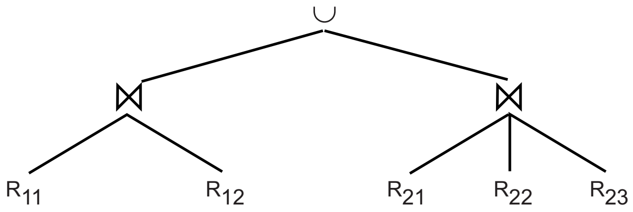


Fig. 3.20 Reconstruction of Hybrid Fragmentation

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

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

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.

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

(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.