M. Tamer Özsu
Patrick Valduriez

Principles of Distributed Database Systems

Third Edition

Springer

M. Tamer Özsu • Patrick Valduriez

Principles of Distributed Database Systems

Third Edition

Springer

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

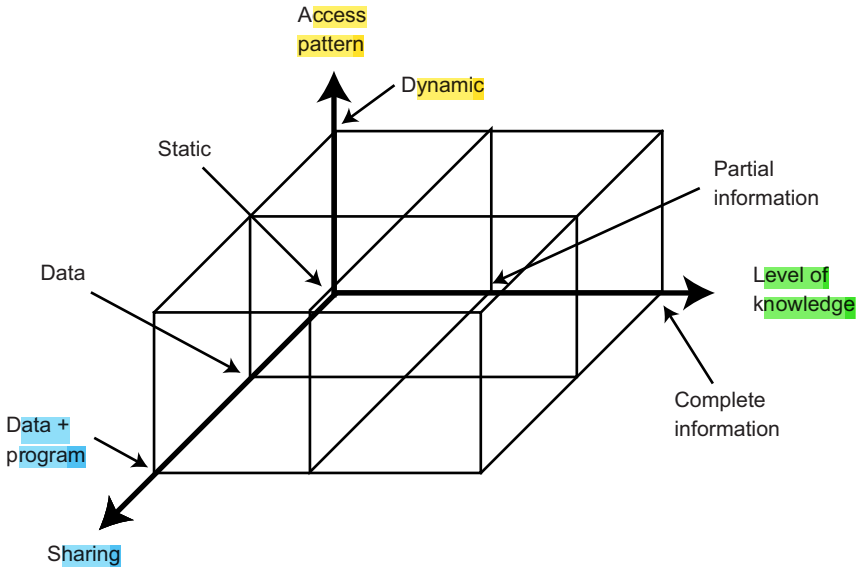

Fig. 3.1 Framework of Distribution

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.

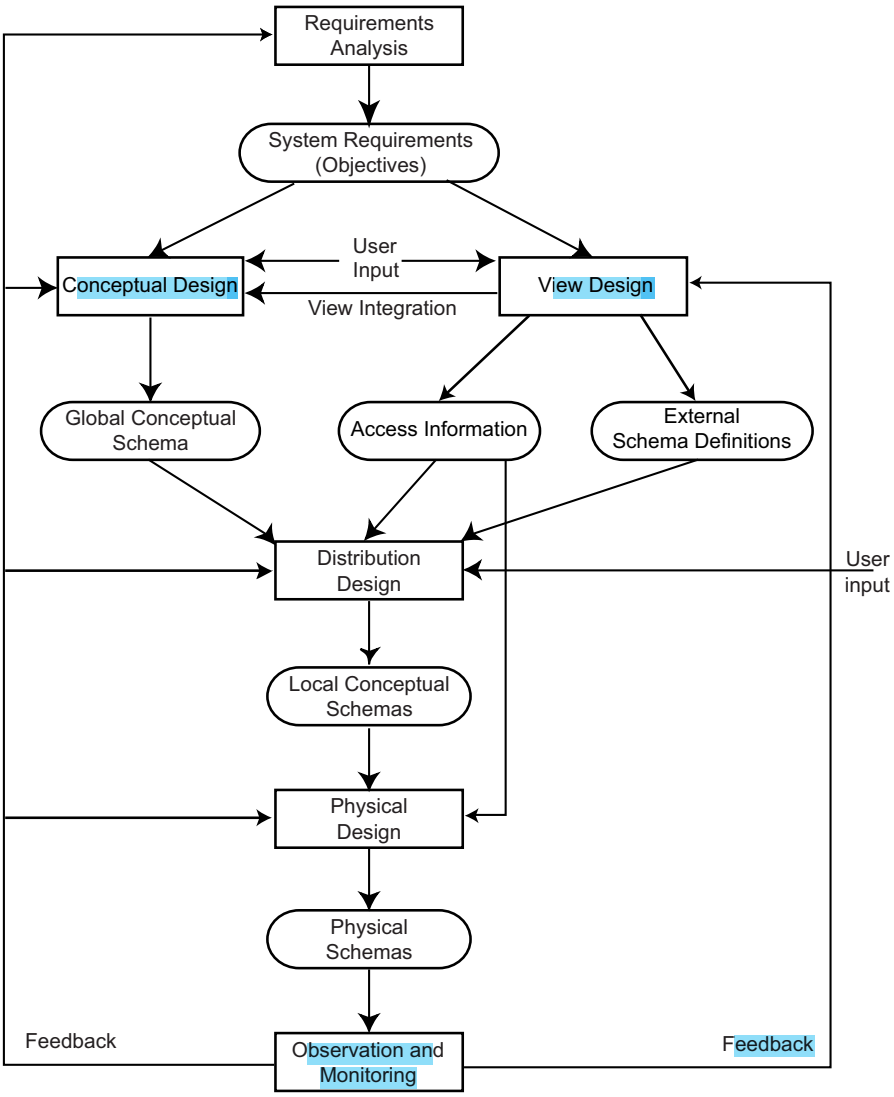

Fig. 3.2 Top-Down Design Process

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 ∇ such that

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

   The operator ∇ 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. ♦

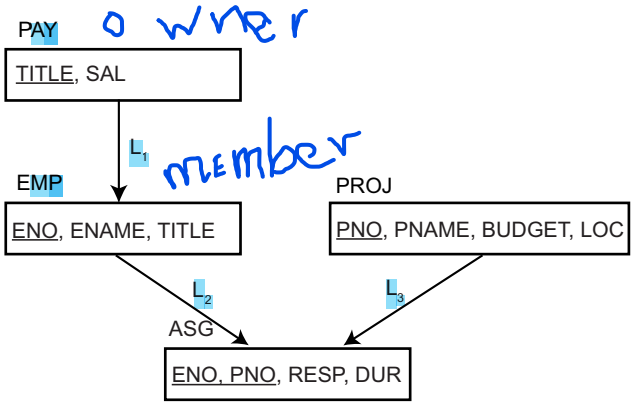

Fig. 3.7 Expression of Relationships Among Relations Using Links

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:

$$\begin{aligned} owner(L_1) &= \text{PAY} \\ member(L_1) &= \text{EMP} \end{aligned}$$

♦

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.

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

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

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

♦

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:

$$Pr = \{\text{LOC="Montreal"}, \text{LOC="New York"}, \text{LOC="Paris"}, \text{BUDGET} \leq 200000, \text{BUDGET} > 200000\}$$

♦

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' ←COM_MIN(R, Pr) ;
    determine the set M of minterm predicates ;
    determine the set I of implications among p_i ∈ Pr' ;
    foreach m_i ∈ M do
        if m_i is contradictory according to I then
            M ← 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 ≤ 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 ≤ 200000)
m_2: (LOC = "Montreal") $\wedge$ (BUDGET > 200000)
m_3: (LOC = "New York") $\wedge$ (BUDGET ≤ 200000)
m_4: (LOC = "New York") $\wedge$ (BUDGET > 200000)
m_5: (LOC = "Paris") $\wedge$ (BUDGET ≤ 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 $F_{PROJ} = \{\text{PROJ}_1, \text{PROJ}_2, \text{PROJ}_3, \text{PROJ}_4, \text{PROJ}_5, \text{PROJ}_6\}$ 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