Chapter 2

OVERVIEW OF RELATIONAL DBMS

In this chapter we review fundamental relational database concepts. The aim of this chapter is to define the terminology and framework used in subsequent chapters, not to provide substantial background on database systems. Nevertheless, we try to be complete. The reasons for choosing the relational model of data as the underlying formalism are numerous: the mathematical foundation of the relational model makes it a good candidate for theoretical treatment; most of the problems discussed in future chapters are easier to formulate; the relational DBMS market has matured and is now sizable; and finally, most distributed database systems are also relational.

The relational model can be characterized by at least three powerful features [Codd, 1982]:

1. Its data structures are simple. They are relations that are two-dimensional tables whose elements are data items. This allows a high degree of independence from the physical data representation (e.g., files and indices).

2. The relational model provides a solid foundation to data consistency. Database design is aided by the normalization process that eliminates data anomalies. Also, consistent states of a database can be uniformly defined and maintained through integrity rules.

3. The relational model allows the set-oriented manipulation of relations. This feature has led to the development of powerful nonprocedural languages based either on set theory (relational algebra) or on logic (relational calculus).

The outline of this chapter is as follows. In Section 2.1 we define fundamental relational database concepts such as a relation, a tuple, a key, and so on. In Section 2.2 the concept of normalization is introduced and different normal forms are discussed. This is followed by a short discussion of integrity rules in Section 2.3. Section 2.4 contains the details of relational data languages (relational algebra and relational calculus), and finally, Section 2.5 presents a short discussion of relational DBMSs. The latter section also serves as a preparation to the architecture discussions of Chapter 4.

2.1 RELATIONAL DATABASE CONCEPTS

A *database* is a structured collection of data related to some real-life phenomena that we are trying to model. A *relational database* is one where the database structure is in the form of tables. Formally, a relation R defined over n sets $D_1, D_2, ..., D_n$ (not necessarily distinct) is a set of *n-tuples* (or simply *tuples*) $< d_1, d_2, ..., d_n >$ such that $d_1 \in D_1, d_2 \in D_2, ..., d_n \in D_n$.

Example 2.1

As an example we will use a database that models a manufacturing company. The entities to be modeled are the *employees* (EMP) and *projects* (PROJ). For each employee, we would like to keep track of the employee number (ENO), name (ENAME), title in the company (TITLE), salary (SAL), identification number of the project(s) the employee is working on (PNO), responsibility within the project (RESP), and duration of the assignment (DUR) in months. Similarly, for each project we would like to store the project number (PNO), the project name (JNAME), and the project budget (BUDGET).

The *relation schemes* for this database can be defined as follows:

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

In relation scheme EMP, there are seven *attributes*: ENO, ENAME, TITLE, SAL, PNO, RESP, DUR. The values of ENO come from the *domain* of all valid employee numbers, say D_1, the values of ENAME come from the domain of all valid names, say D_2, and so on. Note that each attribute of each relation does not have to come from a distinct domain. Various attributes within a relation or from a number of relations may be defined over the same domain.

The *key* of a relation scheme is the minimum nonempty subset of its attributes such that the values of the attributes comprising the key uniquely identify each tuple of the relation. The attributes that make up key are called *prime* attributes. The superset of a key is usually called a *superkey*. Thus in our example the key of PROJ is PNO, and that of EMP is the set (ENO, PNO). Each relation has at least one key. Sometimes, there may be more than one possibility for the key. In such cases, each alternative is considered a *candidate key*, and one of the candidate keys is chosen as the *primary key*. The number of attributes of a relation defines its *degree*, whereas the number of tuples of the relation defines its *cardinality*.

EMP

ENO	ENAME	TITLE	SAL	PNO	RESP	DUR

PROJ

PNO	PNAME	BUDGET

Figure 2.1. Sample Database Scheme

In tabular form, the example database consists of two tables, as shown in Figure 2.1. The columns of the tables correspond to the attributes of the relations; if there were any information entered as the rows, they would correspond to the tuples. The empty table, showing the structure of the table, corresponds to the relation scheme; when the table is filled with rows, it correspdnds to a *relation instance*. Since the information within a table varies over time, many instances can be generated from one relation scheme. Note that from now on, the term *relation* refers to a relation instance. In Figure 2.2 we depict instances of the relations that are defined in Figure 2.1.

An attribute value may be undefined. This lack of definition may have various interpretations, the most common being unknown or not applicable. This special value of the attribute is generally referred to as the *null value*. The representation of a null value must be different from any other domain value, and special care should be given to differentiate it from zero. For example, value "0" for attribute DUR is *known information* (e.g., in the case of a newly hired employee), while value "null" for DUR means unknown in tuple E4 of Figure 2.2. Supporting null values is an important feature necessary to deal with *maybe* queries [Codd, 1979].

2.2 NORMALIZATION

"Normalization is a step-by-step reversible process of replacing a given collection of relations by successive collections in which relations have a progressively simpler and more regular structure" [Tsichritzis and Lochovsky, 1977]. The aim of normalization is to eliminate various anomalies (or undesirable aspects) of a relation in order to obtain "better" relations. The following four problems might exist in a relation scheme:

1. *Repetition anomaly.* Certain information may be repeated unnecessarily. Consider, for example, the EMP relation in Figure 2.2. The name, title, and salary of an employee are repeated for each project on which this person serves. This is obviously a waste of storage and is contrary to the spirit of databases.

2. *Update anomaly.* As a consequence of the repetition of data, performing updates may be troublesome. For example, if the salary of an employee changes, multiple tuples have to be updated to reflect this change.

3. *Insertion anomaly.* It may not be possible to add new information to the database. For example, when a new employee joins the company, we cannot add personal information (name, title, salary) to the EMP relation unless an appointment to a project is made. This is because the key of EMP includes the attribute PNO, and null values cannot be part of the key.

4. *Deletion anomaly.* This is the converse of the insertion anomaly. If an employee works on only one project, and that project is terminated, it is not possible to delete the project information from the EMP relation. To do so would result in deleting the only tuple about the employee, thereby resulting in the loss of personal information we might want to retain.

EMP

ENO	ENAME	TITLE	SALE	PNO	RESP	DUR
E1	J. Doe	Elect. Eng.	40000	P1	Manager	12
E2	M. Smith	Analyst	34000	P1	Analyst	24
E2	M. Smith	Analyst	34000	P2	Analyst	6
E3	A. Lee	Mech. Eng.	27000	P3	Consultant	10
E3	A. Lee	Mech. Eng.	27000	P4	Engineer	48
E4	J. Miller	Programmer	24000	P2	Programmer	18
E5	B. Casey	Syst. Anal.	34000	P2	Manager	24
E6	L. Chu	Elect. Eng.	40000	P4	Manager	48
E7	R. DAvis	Mech. Eng.	27000	P3	Engineer	36
E8	J. Jones	Syst. Anal.	34000	P3	Manager	40

PROJ

PNO	PNAME	BUDGET
P1	Instrumentation	150000
P2	Database Develop.	135000
P3	CAD/CAM	250000
P4	Maintenance	310000

Figure 2.2. Sample Database Instance

Normalization transforms arbitrary relation schemes into ones without these problems. The most popular approach to normalizing a relational database scheme is the *decomposition* approach, where one starts with a single relation, called the universal relation, which contains all attributes (and probably anomalies) and iteratively reduces it. At each iteration, a relation is split into two or more relations of a higher *normal form*. A relation is said to be in a normal form if it satisfies the conditions associated with that normal form. Codd initially defined the *first*, *second*, and *third* normal forms (1NF, 2NF, and 3NF, respectively). Boyce and Codd [Codd, 1974] later defined a modified version of the third normal form, commonly known as the *Boyce–Codd normal form* (BCNF). This was followed by the definition of the *fourth* (4NF) [Fagin, 1977] and *fifth* normal forms (5NF) [Fagin, 1979].

There is a hierarchical relationship among these normal forms. Every normalized relation is in 1NF; some of the relations in 1NF are also in 2NF, some of which are in 3NF, and so on. The higher normal forms have better properties than others with respect to the four anomalies discussed above.

One of the requirements of a normalization process is that the decomposition be lossless. This means that the replacement of a relation by several others should not result in loss of information. If it is possible to join the decomposed relations to obtain the original relation, the process is said to be a *lossless decomposition*.

The join operation is defined formally in Section 2.4.1. Intuitively, it is an operation that takes two relations and concatenates each tuple of the second relation with those tuples of the first relation that satisfy a specified condition. The

condition is defined over the attributes of the two relations. For example, it might be specified that the value of an attribute of the first relation should be equal to the value of an attribute of the second relation.

Another requirement of the normalization process is dependency preservation. A decomposition is said to be dependency preserving if the union of the dependencies in the decomposed relations is equivalent to the closure (with respect to a set of inference rules) of the dependencies of the original relation.

2.2.1 Dependency Structures

The normal forms are based on certain dependency structures. BCNF and lower normal forms are based on *functional dependencies* (FDs), 4NF is based on *multivalued dependencies*, and 5NF is based on *projection-join dependencies*. In this section we define what we mean by dependence.

Let R be a relation defined over the set of attributes $A = \{A_1, A_2, \ldots, A_n\}$ and let $X \subset A$, $Y \subset A$. If for each value of X in R, there is only one associated Y value, we say that "*X functionally determines Y*" or that "*Y is functionally dependent* on X." Notationally, this is shown as $X \rightarrow Y$. The key of a relation functionally determines the nonkey attributes of the same relation.

Example 2.2

For example, in the PROJ relation of Figure 2.2, the valid FD is

$$\text{PNO} \rightarrow (\text{PNAME, BUDGET})$$

In the EMP relation we have

$$(\text{ENO, PNO}) \rightarrow (\text{ENAME, TITLE, SAL, RESP, DUR})$$

This last FD is not the only FD in EMP, however. If each employee is given unique employee numbers, we have

$$\text{ENO} \rightarrow (\text{ENAME, TITLE, SAL})$$

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

It is also reasonable to state that the salary for a given position is fixed, which gives rise to the FD

$$\text{TITLE} \rightarrow \text{SAL}$$

Notice that some of the attributes on the right-hand side of the second FD are also dependent on a subset of the set of attributes at the left-hand side of the same FD. Such attributes (ENAME, TITLE, SAL) are said to be *partially functionally dependent* on (ENO, PNO), whereas the others (RESP, DUR) are said to be *fully functionally dependent*.

Let R be a relation defined over the set of attributes $A = \{A_1, A_2, \ldots, A_n\}$, and let $X \subset A$, $Y \subset A$, $Z \subset A$. If for each value of Z in R, there is only one value for

the (X, Y) pair, and the value of Z depends only on the value of X, we say that "X *multidetermines* Z" or that "Z is *multidependent* on X." This type of dependency is called *multivalued dependency* (MVD) and is denoted as $X \rightarrow\rightarrow Z$.

Intuitively, a MVD represents a situation where the value of one attribute (or a set of attributes) determines a *set of values* of another attribute (or set of attributes). Note that every FD is also an MVD, but the reverse is not necessarily true.

Example 2.3

Going back to our example, let us assume that we want to maintain information on the set of employees and on the set of projects that the company is involved in, as well as the branch offices where this project may be carried out. This can be done by defining the relation.

SKILL (ENO, PNO, PLACE)

SKILL

ENO	PNO	PLACE
E1	P1	Toronto
E1	P1	New York
E1	P1	London
E1	P1	Toronto
E1	P2	New York
E1	P2	London
E2	P1	Toronto
E2	P1	New York
E2	P1	London
E2	P2	Toronto
E2	P2	New York
E2	P2	London

Figure 2.3. Example of MVD

Let us assume (probably unrealistically) that (1) each employee can work on each project, that (2) each employee is willing to work at any of the branch offices, and that (3) each project can be carried out at any of the branch offices. A sample relation instance satisfying these conditions is illustrated in Figure 2.3.

Notice that there are no FDs in the SKILL relation; the relation consists solely of key attributes. The only dependencies are the two MVDs

$$\text{ENO} \rightarrow\rightarrow \text{PNO}$$

$$\text{ENO} \rightarrow\rightarrow \text{PLACE}$$

Let R be a relation denned over the set of attributes $A = \{A_1, A_2, \ldots, A_n\}$, and $X \subset A$, $Y \subset A$, $Z \subset A$. Then, if R is equal to the join of X, Y, and Z, (X, Y, Z) constitutes a *projection-join dependency* for R. Again, we have not yet defined the join operation formally, but the intuitive discussion that we gave before is sufficient.

There is a set of inference rules based on a set of axioms—known as *Armstrong's axioms* [Armstrong, 1974]—that permit the algebraic manipulation of dependencies. They enable the discovery of the minimal cover of a set of FDs that is the minimal set of FDs from which all others can be generated. Given the minimal cover of FDs and a set of attributes, an algorithm can be developed to generate a relational scheme in higher normal forms.

2.2.2 Normal Forms

The first normal form (1NF) states simply that the attributes of the relation contain atomic values only. In other words, the tables should be flat with no repeating groups. The relations EMP and PROJ in Figure 2.2 satisfy this condition, so they both are in 1NF.

Relations in 1NF still suffer from the anomalies discussed earlier. To eliminate some of these anomalies, they should be decomposed into relations in higher normal forms. We are not particularly interested in the second normal form. In fact, it is only of historical importance, since there are algorithms that take a 1NF relation and directly normalize it to third normal form (3NF) or higher.

A relation R is in 3NF if for each FD $X \rightarrow Y$ where Y is not in X, either X is a superkey of R or Y is a prime attribute. There are algorithms that provide a lossless and dependency-preserving decomposition of a 1NF relation into a 3NF relation.

Example 2.4

The PROJ relation in the example we are considering is in 3NF, but EMP is not because of FD:

$$\text{TITLE} \rightarrow \text{SAL}$$

This violates 3NF, because TITLE is not a superkey and SAL is not prime. The problem with EMP is the following. If we want to insert the fact that a given position (title) earns a specific salary, we cannot do so unless there is at least one employee holding that title. (Similar arguments can be made for the update and deletion anomalies.) Thus we have to decompose EMP into the following two relations:

EMP(ENO, ENAME, TITLE, PNO, RESP, DUR)
PAY(TITLE, SAL)

A careful reader will notice that even though PAY is in 3NF, EMP is not due to the FD:

$$\text{ENO} \rightarrow (\text{ENAME}, \text{TITLE})$$

ENO is not a superkey and ENAME and TITLE are not prime attributes. Therefore, EMP needs to be further decomposed into

EMP(ENO, ENAME, TITLE)
ASG(ENO, PNO, RESP, DUR)

both of which are in 3NF.

Boyce–Codd normal form (BCNF) is a stronger form of 3NF. The definitions are identical except for the last part. For a relation to be in BCNF, for every FD $X \rightarrow Y$, X has to be a superkey. Notice that the clause "or Y is a prime attribute" is deleted from the definition. The final form of relation EMP, as well as the relations PAY, PROJ, and ASG, are in BCNF.

It is possible to decompose a 1NF relation directly into a set of relations in BCNF. These algorithms are guaranteed to generate lossless decompositions; however, they cannot be guaranteed to preserve dependencies.

A relation R is in fourth normal form (4NF) if for each MVD of the type $X \rightarrow\rightarrow Y$ in R, X also functionally determines all the attributes of R. Thus, if a relation is in BCNF, and all MVDs are also FDs, the relation is in 4NF. The point is that a 4NF relation either does not contain a real MVD (i.e., every MVD is actually an FD) or there is exactly one MVD represented in the attributes and nothing else.

Example 2.5

Note that the relations EMP, PAY, PROJ, and ASG are in 4NF since there is no MVD defined on them. However, the SKILL relation discussed previously is not in 4NF. To satisfy the requirements, it needs to be decomposed into two relations:

EP(ENO, PNO)
EL(ENO, PLACE)

The careful reader will notice that in all of the previous normal forms, decomposition was into two relations. Fifth normal form (5NF) deals with those situations where n-way decompositions ($n > 2$) may be necessary.

A relation R is in 5NF (also called *projection-join normal form—PJNF*) if every join dependency defined for the relation is implied by the candidate keys of R. For a join dependency to be implied by a candidate key of a relation, the subset (or projections) X, Y, and Z (see definition of join dependency) should be made according to a candidate key.

Example 2.6

For relation EMP we can define the join dependency

*((ENO, ENAME), (ENO, TITLE))

which is implied by the candidate key ENO (which also happens to be the primary key). It is quite easy to verify that the relations EMP, PAY, PROJ, and ASG are in 5NF. Thus the relation schemes that we end up with after the decompositions are as follows:

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
P1	Instrumentation	150000
P2	Database Develop.	135000
P3	CAD/CAM	250000
P4	Maintenance	310000

PAY

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

Figure 2.4. Normalized Relations

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

The normalized instances of these relations are shown in Figure 2.4.

All NFs presented above are lossless. An important result [Fagin, 1977] is that a 5NF relation cannot be further decomposed without loss of information.

2.3 INTEGRITY RULES

Integrity rules are constraints that define consistent states of the database. They are usually expressed as assertions. Integrity constraints can be *structural* or *behavioral*. Structural constraints are inherent to the data model in the sense that they capture information on data relationships that cannot be modeled directly.

Behavioral constraints permit the capturing of the semantics of the applications. The dependencies discussed in the preceding section are behavioral constraints. Maintaining integrity constraints is generally expensive in terms of system resources. Ideally, they should be verified at each database update since updates can lead to inconsistent database states. The problem of maintaining distributed integrity constraints is covered in Chapter 6.

According to [Codd, 1982], the two minimal structural constraints of the relational model are the *entity rule* and the *referential integrity rule.* By definition, any relation has a primary key. The entity rule dictates that each attribute of the key is nonnull. In Example 2.1, attribute PNO of relation PROJ and attributes (ENO, PNO) of relation EMP cannot have null values. This constraint is necessary to enforce the fact that keys are unique.

Referential integrity [Date, 1983] is useful for capturing relationships between objects that the relational model cannot represent. We make use of referential integrity in Chapter 5 during our discussion of distributed database design. Other data models, such as the hierarchical model [Tsichritzis and Lochovsky, 1976] or the entity/relationship model [Chen, 1976], can capture this type of information directly. Referential integrity involves two relations and imposes the constraint that a group of attributes in one relation is the key of another relation. In Example 2.1 there can be a referential integrity constraint between relations PROJ and EMP on attribute PNO. This rule prescribes that each employee belongs to at least one existing project. In other words, the set of PNO values in relation EMP is included in relation PROJ. Thus there cannot be employees that belong to projects not in relation PROJ.

2.4 RELATIONAL DATA LANGUAGES

Data manipulation languages developed for the relational model (commonly called *query languages*) fall into two fundamental groups: *relational algebra*-based languages and *relational calculus*-based languages. The difference between them is based on how the user query is formulated. The relational algebra is procedural in that the user is expected to specify, using certain high-level operators, how the result is to be obtained. The relational calculus, on the other hand, is nonprocedural; the user only specifies the relationships that should hold in the result. Both of these languages were originally proposed by Codd [Codd, 1970], who also proved that they were equivalent in terms of expressive power [Codd, 1972].

Relational algebra is used more than relational calculus in the study of distributed database issues, because it is of lower level and corresponds more directly to the programs exchanged on a network. However, for the sake of completeness, we discuss them both here. Essentially, relational calculus can be translated into relational algebra.

2.4.1 Relational Algebra

Relational algebra consists of a set of operators that operate on relations. It is derived from set theory (relations corresponding to sets). Each operator takes one

or two relations as operands and produces a result relation, which, in turn, may be an operand to another operator. These operations permit the querying and updating of a relational database.

Algebra operations

There are five fundamental relational algebra operators and five others that can be defined in terms of these. The fundamental operators are *selection, projection, union, set difference,* and *Cartesian product.* The first two of these operators are unary operators, and the last three are binary operators. The additional operators that can be defined in terms of these fundamental operators are *intersection, θ-join, natural join, semijoin* and *quotient.* In practice, relational algebra is extended with operators for grouping or sorting the results, and for performing arithmetic and aggregate functions. Other operators, such as *outer join* and *transitive closure,* are sometimes used as well to provide additional functionality. However, they are not discussed here.

The operands of some of the binary relations should be *union compatible.* Two relations R and S are union compatible if and only if they are of the same degree and the ith attribute of each is defined over the same domain. The second part of the definition holds, obviously, only when the attributes of a relation are identified by their relative positions within the relation and not by their names. If relative ordering of attributes is not important, it is necessary to replace the second part of the definition by the phrase "the corresponding attributes of the two relations should be defined over the same domain." The correspondence is defined rather loosely here.

Selection. Selection produces a horizontal subset of a given relation. The subset consists of all the tuples that satisfy a formula (condition). The selection from a relation R is

$$\sigma_F(R)$$

where R is the relation and F is a formula.

Since we refer to formulas repeatedly in this chapter, let us define precisely what we mean at this point. We define a formula within the context of first-order predicate calculus ([Stoll, 1963] and [Enderton, 1972]) (since we use that formalism later), and follow the notation of [Gallaire et al., 1984]. First-order predicate calculus is based on a *symbol alphabet* that consists of (1) variables, constants, functions, and predicate symbols; (2) parentheses; (3) the logical connectors $\wedge$ (and), $\vee$ (or), $\neg$ (not), $\rightarrow$ (implication), and $\leftrightarrow$ (equivalence); and (4) quantifiers $\forall$ (for all) and $\exists$ (there exists). A *term* is either a constant or a variable. Recursively, if f is an $n$ary function and $t_1, \ldots, t_n$ are terms, $f(t_1, \ldots, t_n)$ is also a term. An *atomic formula* is of the form $P(t_1, \ldots, t_n)$, where P is an $n$ary predicate symbol and the t_i's are terms. A *well-formed formula* (*wff*) can be defined recursively as follows: If w_i and w_j are wffs, then (w_i), $\neg(w_i)$, $(w_i) \wedge (w_j)$, $(w_i) \vee (w_j)$, $(w_i) \rightarrow (w_j)$, and $(w_i) \leftrightarrow (w_j)$ are all wffs. Variables in a wff may be *free* or they may be *bound* by one of the two quantifiers.

The formula in the selection operation is called a *selection predicate* and is an atomic formula whose terms are of the form $A\theta c$, where A is an attribute of R and θ is one of the arithmetic comparison operators $<$, $>$, $=$, $\neq$, $\leq$, and $\geq$. The terms can be connected by the logical connectors $\wedge$, $\vee$, and $\neg$. Furthermore, the selection predicate does not contain any quantifiers.

Example 2.7

Consider the relation EMP shown in Figure 2.4. The result of selecting those tuples for electrical engineers is shown in Figure 2.5.

$\sigma_{\text{TITLE="Elect. Eng."}}(\text{EMP})$

ENO	ENAME	TITLE
E1	J. Doe	Elect. Eng.
E6	L. Chu	Elect. Eng.

Figure 2.5. Result of Selection

Projection. Projection produces a vertical subset of a relation. The result relation contains only those attributes of the original relation over which projection is performed. Thus the degree of the result is less than or equal to the degree of the original relation.

The projection of relation R over attributes A and B is denoted as

$$\Pi_{A,B}(R)$$

Note that the result of a projection might contain tuples which are identical. In that case the duplicate tuples may be deleted from the result relation. It is possible to specify projection with or without duplicate elimination.

Example 2.8

The projection of relation PROJ shown in Figure 2.4 over attributes PNO and BUDGET is depicted in Figure 2.6.

$\Pi_{\text{PNO,BUDGET}}(\text{PROJ})$

PNO	BUDGET
P1	150000
P2	135000
P3	250000
P4	310000

Figure 2.6. Result of Projection

Union. The union of two relations R and S (denoted as $R \cup S$) is the set of all tuples that are in R, or in S, or in both. We should note that R and S should be union compatible. As in the case of projection, the duplicate tuples are normally eliminated. Union may be used to insert new tuples into an existing relation, where these tuples form one of the operand relations.

Set Difference. The set difference of two relations R and S ($R - S$) is the set of all tuples that are in R but not in S. In this case, not only should R and S be union compatible, but the operation is also asymmetric (i.e., $R - S \neq S - R$). This operation allows the deletion of tuples from a relation. Together with the union operation, we can perform modification of tuples by deletion followed by insertion.

Cartesian Product. The Cartesian product of two relations R of degree k_1 and S of degree k_2 is the set of $(k_1 + k_2)$-tuples, where each result tuple is a concatenation of one tuple of R with one tuple of S, for all tuples of R and S. The Cartesian product of R and S is denoted as $R \times S$.

It is possible that the two relations might have attributes with the same name. In this case the attribute names are prefixed with the relation name so as to maintain the uniqueness of the attribute names within a relation.

Example 2.9

Consider relations EMP and PAY in Figure 2.4. EMP x PAY is shown in Figure 2.7. Note that the attribute TITLE, which is common to both relations, appears twice, prefixed with the relation name.

Intersection. Intersection of two relations R and S ($R \cap S$) consists of the set of all tuples that are in both R and S. In terms of the basic operators, it can be specified as follows:

$$R \cap S = R - (R - S)$$

θ-Join. Join is a derivative of Cartesian product. There are various forms of join, the most general of which is the θ-join, commonly called the join. The θ-join of two relations R and S is denoted as

$$R \bowtie_F S$$

where F is a formula specifying the *join predicate*. A join predicate is specified similar to a selection predicate, except that the terms are of the form $R.A\theta S.B$, where A and B are attributes of R and S, respectively.

The join of two relations is equivalent to performing a selection, using the join predicate as the selection formula, over the Cartesian product of the two operand relations. Thus

$$R \bowtie_F S = \sigma_F (R \times S)$$

EMP × PAY

ENO	ENAME	E.TITLE	PAY.TITLE	SAL
E1	J. Doe	Elect. Eng.	Elect. Eng.	40000
E1	J. Doe	Elect. Eng.	Syst. Anal.	34000
E1	J. Doe	Elect. Eng.	Mech. Eng.	27000
E1	J. Doe	Elect. Eng.	Programmer	24000
E2	M. Smith	Syst. Anal.	Elect. Eng.	40000
E2	M. Smith	Syst. Anal.	Syst. Anal.	34000
E2	M. Smith	Syst. Anal.	Mech. Eng.	27000
E2	M. Smith	Syst. Anal.	Programmer	24000
E3	A. Lee	Mech. Eng.	Elect. Eng.	40000
E3	A. Lee	Mech. Eng.	Syst. Anal.	34000
E3	A. Lee	Mech. Eng.	Mech. Eng.	27000
E3	A. Lee	Mech. Eng.	Programmer	24000
E8	J. Jones	Syst. Anal.	Elect. Eng.	40000
E8	J. Jones	Syst. Anal.	Syst. Anal.	34000
E8	J. Jones	Syst. Anal.	Mech. Eng.	27000
E8	J. Jones	Syst. Anal.	Programmer	24000

Figure 2.7. Partial Result of Cartesian Product

In the equivalence above, we should note that if F involves attributes of the two relations that are common to both of them, a projection is necessary to make sure that those attributes do not appear twice in the result.

Example 2.10

Figure 2.8 shows the θ-join of relations EMP and PAY in Figure 2.4 over the join predicate EMP.TITLE=PAY.TITLE. The same result could have been obtained as

$$EMP \bowtie_{EMP.TITLE=PAY.TITLE} PAY =$$
$$\Pi_{ENO, ENAME, TITLE, SAL}(\sigma_{EMP.TITLE = PAY.TITLE}(EMP \times PAY))$$

This example demonstrates a special case of θ-join which is called the *equi-join*. This is a case where the formula F only contains equality (=) as the arithmetic operator. It should be noted, however, that an equi-join does not have to be specified over a common attribute as the example above might suggest.

Natural Join. A natural join is an equi-join of two relations over a specified attribute, more specifically, over attributes with the same domain. There is a difference, however, in that usually the attributes over which the natural join is performed appear only once in the result. A natural join is denoted as the join without the formula

$$R \bowtie_A S$$

$EMP \bowtie_{EMP.TITLE=PAY.TITLE} PAY$

ENO	ENAME	TITLE	SAL
E1	J. Doe	Elect. Eng.	40000
E2	M. Smith	Analyst	34000
E3	A. Lee	Mech. Eng.	27000
E4	J. Miller	Programmer	24000
E5	B. Casey	Syst. Anal.	34000
E6	L. Chu	Elect. Eng.	40000
E7	R. Davis	Mech. Eng.	27000
E8	J. Jones	Syst. Anal.	34000

Figure 2.8. The Result of Join

where, A is the attribute common to both R and S. We should note here that the natural join attribute may have different names in the two relations; what is required is that they come from the same domain. In this case the join is denoted as

$$R_A \bowtie_B S$$

where B is the corresponding join attribute of S.

Example 2.11

The join of EMP and PAY in Example 2.10 is actually a natural join.

Semijoin. The semijoin of relation R, defined over the set of attributes A, by relation S, defined over the set of attributes B, is the subset of the tuples of R that participate in the join of R with S. It is denoted as $R \ltimes_F S$ (where F is a predicate as defined before) and can be obtained as follows:

$$\begin{aligned} R \ltimes_F S &= \Pi_A(R \bowtie_F S) = \Pi_A(R) \bowtie_F \Pi_{A \cap B}(S) \\ &= R \bowtie_F \Pi_{A \cap B}(S) \end{aligned}$$

The advantage of semijoin is that it decreases the number of tuples that need to be handled to form the join. In centralized database systems, this is important because it usually results in a decreased number of secondary storage accesses by making better use of the memory. It is even more important in distributed databases since it usually reduces the amount of data that needs to be transmitted between sites in order to evaluate a query. We talk about this in more detail in Chapters 5 and 9. At this point note that the operation is asymmetric (i.e., $R \ltimes_F S \neq S \ltimes_F R$).

Example 2.12

To demonstrate the difference between join and semijoin, let us consider the semijoin of EMP with PAY over the predicate EMP.TITLE = PAY.TITLE, that is,

$$\text{EMP} \ltimes_{\text{EMP.TITLE} = \text{PAY.TITLE}} \text{PAY}$$

The result of the operation is shown in Figure 2.9, We would like to encourage the reader to compare Figures 2.8 and 2.9 to see the difference between the join and the semijoin operations. Note that the resultant relation does not have the PAY attribute and is therefore smaller.

$\text{EMP} \ltimes_{\text{EMP.TITLE-PAY.TITLE}} \text{PAY}$

ENO	ENAME	TITLE
E1	J. Doe	Elect. Eng.
E2	M. Smith	Analyst
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.

Figure 2.9. The Result of Semijoin

(Division) Quotient. The division of relation R of degree r with relation S of degree s (where $r > s$ and $s \neq 0$) is the set of $(r - s)$-tuples t such that for all s-tuples u in S, the tuple tu is in R. The division operation is denoted as $R \div S$ and can be specified in terms of the fundamental operators as follows:

$$R \div S = \Pi_{\bar{A}}(R) - \Pi_{\bar{A}}((\Pi_{\bar{A}}(R) \times S) - R)$$

where $\bar{A}$ is the set of attributes of R that are not in S [i.e., the $(r - s)$-tuples].

Example 2.13

Assume that we have a modified version of the ASG relation (call it ASG′) depicted in Figure 2.10a and defined as follows:

$$\text{ASG}' = \Pi_{\text{ENO,PNO}}(\text{ASG}) \bowtie_{\text{PNO}} \text{PROJ}$$

If one wants to find the employee numbers of those employees who are assigned to all the projects that have a budget greater than $200,000, it is necessary to divide ASG′ with a restricted version of PROJ, called PROJ′

(see Figure 2.10b). The result of division (G′ ÷ J′) is shown in Figure 2.10c. The keyword in the query above is "*all*." This rules out the possibility of doing a selection on ASG′ to find the necessary tuples, since that would only give those which correspond to employees working on *some* project with a budget greater than $200,000, not those who work on all projects. Note that the result contains only the tuple <E3> since the tuples <E3, P3, CAD/CAM, 250000> and <E3, P4, Maintenance, 310000> both exist in ASG′. On the other hand, for example, <E7> is not in the result, since even though the tuple <E7, P3, CAD/CAM, 250000> is in ASG′, the tuple <ET, P4, Maintenance, 310000> is not.

Relational algebra programs.

Since all operations take relations as input and produce relations as outputs, we can nest operations using a parenthesized notation and represent relational algebra programs. The parentheses indicate the order of execution. The following are a few examples that demonstrate the issue.

Example 2.14

Consider the relations of Figure 2.4. The retrieval query

"Find the names of employees working on the CAD/CAM project"

ASG′

ENO	PNO	PNAME	BUDGET
E1	P1	Instrumentation	150000
E2	P1	Instrumentation	150000
E2	P2	Database Develop.	135000
E3	P3	CAD/CAM	250000
E3	P4	Maintenance	310000
E4	P2	Database Develop.	135000
E5	P2	Database Develop.	135000
E6	P4	Maintenance	310000
E7	P3	CAD/CAM	250000
E8	P3	CAD/CAM	250000

(a)

PROJ′

ENO	PNAME	BUDGET
P3	CAD/CAM	250000
P4	Maintenance	310000

(b)

ASG′ ÷ PROJ′)

ENO
E3

(c)

Figure 2.10. The Result of Division

can be answered by the relational algebra program

$$\Pi_{ENAME}(((\sigma_{PNAME = \text{"CAD/CAM"}} PROJ) \ltimes_{PNO} ASG) \ltimes_{ENO} EMP)$$

The order of execution is: the selection on PROJ, followed by the join with ASG, followed by the join with EMP, and finally the project on ENAME. An equivalent program where the size of the intermediate relations is smaller is

$$\Pi_{ENAME}(EMP \ltimes_{ENO} (\Pi_{ENO}(ASG \ltimes_{PNO} (\sigma_{PNAME = \text{"CAD/CAM"}} PROJ))))$$

Example 2.15

The update query

"Replace the salary of programmers by $25,000"

can be computed by

$$(PAY - (\sigma TITLE = \text{"Programmer"} PAY)) \cup (<\text{Programmer}, 25000>)$$

2.4.2 Relational Calculus

In relational calculus-based languages, instead of specifying *how* to obtain the result, one specifies *what* the result is by stating the relationship that is supposed to hold for the result. Relational calculus languages fall into two groups: *tuple relational calculus* and *domain relational calculus*. The difference between the two is in terms of the primitive variable used in specifying the queries. We briefly review these two types of languages.

Relational calculus languages have a solid theoretical foundation since they are based on first-order predicate logic as we discussed before. Semantics is given to formulas by interpreting them as assertions on the database. A relational database can be viewed as a collection of tuples or a collection of domains. Tuple relational calculus interprets a variable in a formula as a tuple of a relation, whereas domain relational calculus interprets a variable as the value of a domain.

Tuple relational calculus.

The primitive variable used in tuple relational calculus is a *tuple variable* which specifies a tuple of a relation. In other words, it ranges over the tuples of a relation. Tuple calculus is the original relational calculus developed by Codd [Codd, 1970]. In tuple relational calculus queries are specified as

$$\{t | F(t)\}$$

where t is a tuple variable and F is a well-formed formula. The atomic formulas are of two forms:

1. *Tuple-variable membership expressions.* If t is a tuple variable ranging over the tuples of relation R (predicate symbol), the expression "tuple t belongs to relation R " is an atomic formula, which is usually specified as $R.t$ or $R(t)$.

2. *Conditions.* These can be defined as follows:

   (a) s [A]θt [B], where s and t are tuple variables and A and B are components of s and t, respectively. θ is one of the arithmetic comparison operators $<$, $>$, $=$, $\neq$, $\leq$, and $\geq$. This condition specifies that component A of s stands in relation θ to the B component of t: for example, s[SAL] $>$ t[SAL].

   (b) $s[A]\theta c$, where s, A, and θ are as defined above and c is a constant. For example, s[ENAME] = "Smith".

Note that A is defined as a component of the tuple variable s. Since the range of s is a relation instance, say S, it is obvious that component A of s corresponds to attribute A of relation S. The same thing is obviously true for B.

There are many languages that are based on relational tuple calculus, the most popular ones being SQL[1] [Date, 1987] and QUEL [Stonebraker et al., 1976]. SQL is now an international standard (actually, the only one) with standard versions released in 1986 (known as SQL1), 1989 (modifications to SQL1) and 1992 (known as SQL2). A new version, including object-oriented language extensions, is in the works. This version, known as SQL3, will be released in parts starting in 1998.

SQL provides a uniform approach to data manipulation (retrieval, update), data definition (schema manipulation), and control (authorization, integrity, etc.). We limit ourselves to the expression, in SQL, of the queries in Examples 2.14 and 2.15.

Example 2.16

The query from Example 2.14,

"Find the names of employees working on the CAD/CAM project" can be expressed as follows:

```
SELECT   EMP.ENAME
FROM     EMP,ASG,PROJ
WHERE    EMP.ENO  =  ASG.ENO
AND      ASG.PNO  =  PROJ.PNO
AND      PROJ.PNAME  =  "CAD/CAM"
```

[1]Sometimes SQL is cited as lying somewhere between relation algebra and relation calculus. Its originators called it a "mapping language." However, it follows the tuple calculus definition quite closely; hence we classify it as such.

Note that a retrieval query generates a new relation similar to the relational algebra operations.

Example 2.17

The update query of Example 2.15,

"Replace the salary of programmers by $25,000"

is expressed as

```
UPDATE   PAY
SET      SAL = 25000
WHERE    PAY.TITLE = "Programmer"
```

Domain relational calculus.

The domain relational calculus was first proposed by [Lacroix and Pirotte, 1977]. The fundamental difference between a tuple relational language and a domain relational language is the use of a *domain variable* in the latter. A domain variable ranges over the values in a domain and specifies a component of a tuple. In other words, the range of a domain variable consists of the domains over which the relation is denned. The wffs are formulated accordingly. The queries are specified in the following form:

$$x_1, x_2, \ldots, x_n | F(x_1, x_2, \ldots, x_n)$$

where F is a wff in which $x_1, \ldots, x_n$ are the free variables.

EMP	ENO	ENAME	TITLE
	E2	P.	

ASG	ENO	PNO	RESP	DUR
	E2	P3		

PROJ	PNO	PNAME	BUDGET
	P3	CAD/CAM	

Figure 2.11. Retrieval Query in QBE

The success of domain relational calculus languages is due mainly to QBE [Zloof, 1977], which is a visual application of domain calculus. QBE, designed only for interactive use from a visual terminal, is user-friendly. The basic concept is an

example: the user formulates queries by providing a possible example of the answer. Typing relation names triggers the printing, on screen, of their schemes. Then, by supplying keywords into the columns (domains), the user specifies the query. For instance, the attributes of the project relation are given by P, which stands for "Print."

By default, all queries are retrieval. An update query requires the specification of U under the name of the updated relation or in the updated column. The retrieval query corresponding to Example 2.16 is given in Figure 2.11 and the update query of Example 2.17 is given in Figure 2.12. To distinguish examples from constants, examples are underlined.

PAY	TITLE	SAG
	Programmer	U.25000

Figure 2.12. Update Query in QBE

2.4.3 Interface with Programming Languages

Relational data languages are insufficient to write complex application programs. Therefore, interfacing a relational data language to a programming language in which the application is written is usually necessary. We can distinguish between two main approaches in providing this service: the *tightly coupled* and *loosely coupled* approaches. The tightly coupled approach consists of extending some programming language with data manipulation commands. The loosely coupled approach consists of integrating the data language with a programming language using database calls.

Tightly coupled approach.

In this approach the programming language and the database language are merged in a single language. A typical example of this approach is Pascal/R [Schmidt, 1977], in which the language Pascal is extended with a new variable type *relation* that contains several (possibly many) instances of tuples (implemented via Pascal records) and with commands to manipulate relations (e.g., for each tuple in relation).

Another typical example of this approach is that of *fourth-generation languages* (4GL) [Martin, 1985]. They are well known because of their commercial success. Fourth-generation languages are high-level languages that combine relational algebra operators with programming constructs. The possibility of using temporary variables and powerful programming constructs (e.g., loops) makes them "database-oriented programming languages" in which application development is facilitated.

Example 2.18

To illustrate this type of language, let us consider the following simple application program (based on relation ASG in Figure 2.4):

> "For the tuples in relation ASG where DUR > 40, perform a complex subprogram on attribute DUR, and produce a report that uses ENO and the result of the subprogram"

Assuming a generic 4GL, this program can be written as

```
for each ASG where DUR > 40
do
    perform subprogram X on DUR giving RESULT
    produce Report on ENO, RESULT
end
```

where RES is the results of the subprogram and Report is a predefined query to a report generator (often part of a 4GL).

Loosely coupled approach.

In this approach the programming language is a high-level language such as COBOL or PL/I which does not know about the database concepts. The programming language is simply extended with special commands that call the database system. These commands are database language constructs simply preceded by a key character (e.g., $) to distinguish them from the constructs of the programming language.

The database languages used with these types of programming languages are usually based on the tuple relational calculus and are set-oriented. The programming language, on the other hand, is procedural. The conversion from the set-oriented mode to the procedural mode required by the programming language is generally simple; it is based on the use of a cursor on a set. A cursor can be explicit, as in embedded SQL, or implicit, as in EQUEL (Embedded QUEL).

Two execution modes of such extended languages are possible: interpretation of database calls at run time or precompilation of database calls. The latter mode is more efficient.

Example 2.19

We illustrate this approach by considering the same application program as in Example 2.18, but this time writing it in SQL embedded in PL/I. as shown in Figure 2.13. The $ indicates a database command.

```
.
.
.
$DCL VARDUR INT;
$DCL VARENO INT;
```

```
.
.
.
$LET C BE { cursor definition }
    SELECT   ASG.ENO: VARENO, ASG.DUR: VARDUR
    FROM     ASG
    WHERE    ASG.DUR>40
.
.
.
$OPEN C
DO WHILE "not end C"
BEGIN
    $FETCH C;
    perform subprogram X on DUR giving RESULT;
    produce Report on ENO, RESULT
END;
```

Figure 2.13. Example PL/I Program

2.5 RELATIONAL DBMS

A relational DBMS is a software component supporting the relational model and a relational language. A DBMS is a reentrant program shared by multiple alive entities, called *transactions*, that run database programs. When running on a general purpose computer, a DBMS is interfaced with two other components: the communication subsystem and the operating system. The generic architecture of a (centralized) DBMS is depicted in Figure 2.14. The communication subsystem permits interfacing the DBMS with other subsystems in order to communicate with applications. For example, the terminal monitor needs to communicate with the DBMS to run interactive transactions. The operating system provides the interface between the DBMS and computer resources (processor, memory, disk drives, etc.).

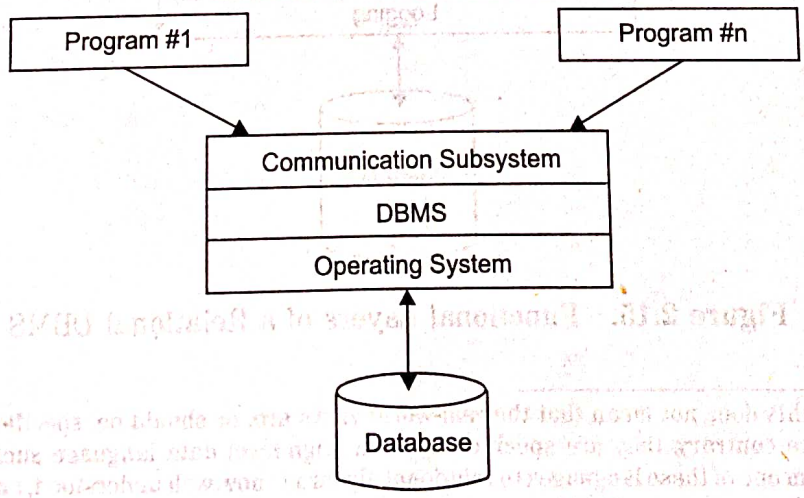

Figure 2.14. Generic Architecture of a Centralized DBMS

The functions performed by a relational DBMS can be layered as in Figure 2.15, where the arrows indicate the direction of the data and the control flow. Taking a top-down approach, the layers are the interface, control, compilation, execution, data access, and consistency management.

The *interface layer* manages the interface to the applications. There can be several interfaces such as SQL embedded in COBOL and QBE (query by example). Database application programs are executed against external *views* of the database. For an application, a view is useful in representing its particular perception of the database (shared by many applications). A relational view is a virtual relation derived from base relations by applying relational algebra operations.[2]

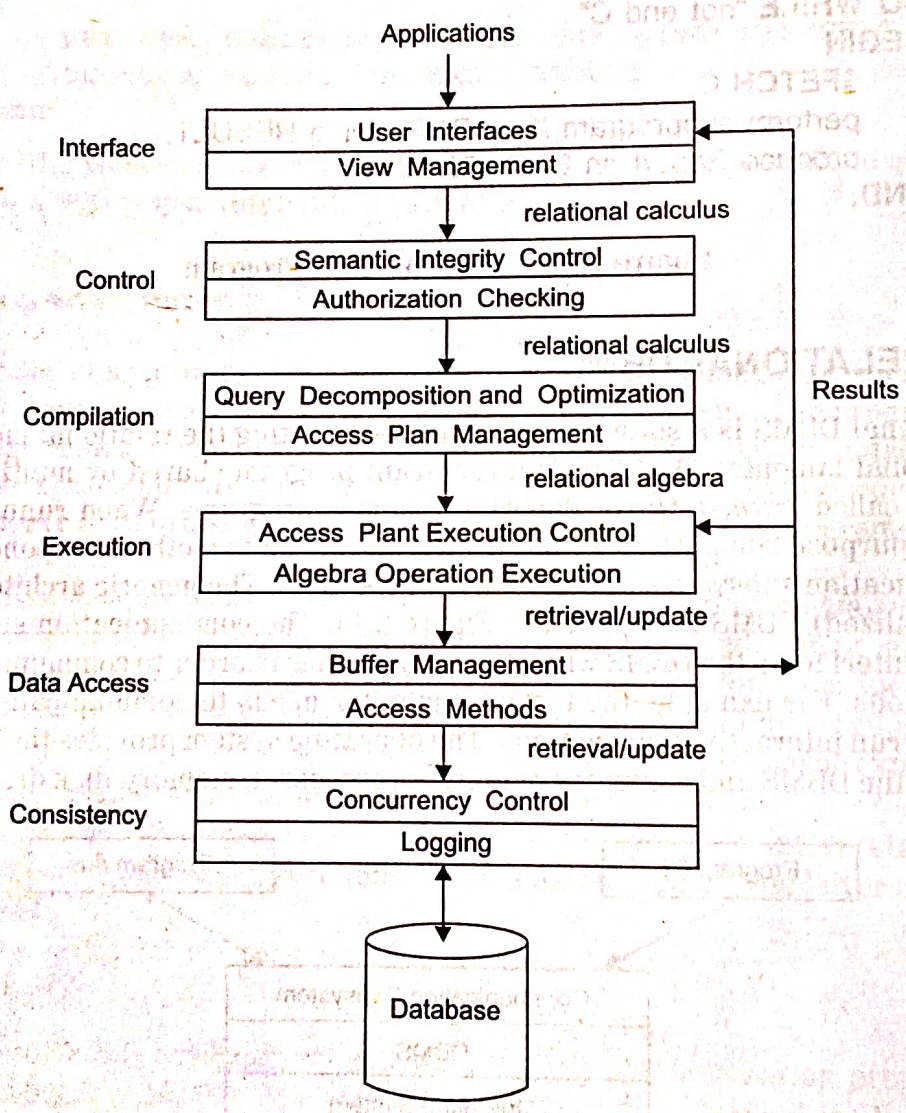

Figure 2.15. Functional Layers of a Relational DBMS

[2]Note that this does not mean that the real-world views are, or should be, specified in relational algebra. On the contrary, they are specified by some high-level data language such as SQL. The translation from one of these languages to relational algebra is now well understood, and the effects of the view definition can be specified in terms of relational algebra operations.

View management consists of translating the user query from external data to conceptual data (base relations). If the user query is expressed in relational calculus, the query applied to conceptual data is still in the same form.

The *control layer* controls the query by adding semantic integrity predicates and authorization predicates. Semantic integrity constraints and authorizations are specified in relational calculus, as discussed in Chapter 6. The output of this layer is an enriched query in relational calculus.

The *query processing* layer maps the query into an optimized sequence of lower-level operations. This layer is concerned with performance. It decomposes the query into a tree of relational algebra operations and tries to find the "optimal" ordering of the operations. The result is stored in an access plan. The output of this layer is a query expressed in relational algebra (or in lower-level code).

REVIEW QUESTIONS

2.1 Explain relational database concepts with examples.
2.2 What do you mean by normalization?
2.3 Explain dependency structures with examples.
2.4 What do you mean by normal forms? Give examples.
2.5 Explain algebra operations with examples.
2.6 What do you mean by relational calculus? Give examples.
2.7 Explain with examples: tightly coupled and loosely coupled approach.
2.8 What do you mean by relational DBMS?