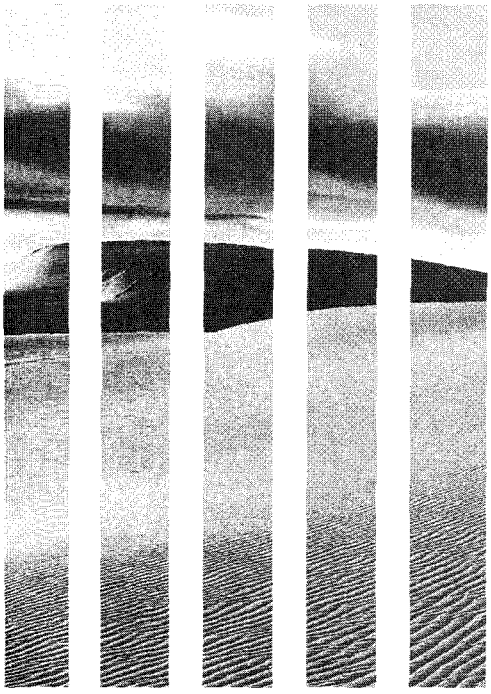


2

RELATIONAL MODEL: CONCEPTS, CONSTRAINTS, LANGUAGES, DESIGN, AND PROGRAMMING

5

The Relational Data Model and Relational Database Constraints

This chapter opens Part II of the book on relational databases. The relational model was first introduced by Ted Codd of IBM Research in 1970 in a classic paper (Codd 1970), and attracted immediate attention due to its simplicity and mathematical foundation. The model uses the concept of a *mathematical relation*—which looks somewhat like a table of values—as its basic building block, and has its theoretical basis in set theory and first-order predicate logic. In this chapter we discuss the basic characteristics of the model and its constraints.

The first commercial implementations of the relational model became available in the early 1980s, such as the Oracle DBMS and the SQL/DS system on the MVS operating system by IBM. Since then, the model has been implemented in a large number of commercial systems. Current popular relational DBMSs (RDBMSs) include DB2 and Informix Dynamic Server (from IBM), Oracle and Rdb (from Oracle), and SQL Server and Access (from Microsoft).

Because of the importance of the relational model, we have devoted all of Part II of this textbook to this model and the languages associated with it. Chapter 6 covers the operations of the relational algebra and introduces the relational calculus notation for two types of calculi—tuple calculus and domain calculus. Chapter 7 relates the relational model data structures to the constructs of the ER and EER models, and presents algorithms for designing a relational database schema by mapping a conceptual schema in the ER or EER model (see Chapters 3 and 4) into a relational representation. These mappings are incorporated into many database design and CASE[1] tools. In Chapter 8, we describe the

1. CASE stands for computer-aided software engineering.

SQL query language, which is the *standard* for commercial relational DBMSs. Chapter 9 discusses the programming techniques used to access database systems, and presents additional topics concerning the SQL language—constraints, views, and the notion of connecting to relational databases via ODBC and JDBC standard protocols. Chapters 10 and 11 in Part III of the book present another aspect of the relational model, namely the formal constraints of functional and multivalued dependencies; these dependencies are used to develop a relational database design theory based on the concept known as *normalization*.

Data models that preceded the relational model include the hierarchical and network models. They were proposed in the 1960s and were implemented in early DBMSs during the 1970s and 1980s. Because of their historical importance and the large existing user base for these DBMSs, we have included a summary of the highlights of these models in appendices, which are available on the Web site for the book. These models and systems will be with us for many years and are now referred to as *legacy database systems*.

In this chapter, we concentrate on describing the basic principles of the relational model of data. We begin by defining the modeling concepts and notation of the relational model in Section 5.1. Section 5.2 is devoted to a discussion of relational constraints that are now considered an important part of the relational model and are automatically enforced in most relational DBMSs. Section 5.3 defines the update operations of the relational model and discusses how violations of integrity constraints are handled.

5.1 Relational Model Concepts

The relational model represents the database as a collection of *relations*. Informally, each relation resembles a table of values or, to some extent, a "flat" file of records. For example, the database of files that was shown in Figure 1.2 is similar to the relational model representation. However, there are important differences between relations and files, as we shall soon see.

When a relation is thought of as a **table** of values, each row in the table represents a collection of related data values. We introduced entity types and relationship types as concepts for modeling real-world data in Chapter 3. In the relational model, each row in the table represents a fact that typically corresponds to a real-world entity or relationship. The table name and column names are used to help in interpreting the meaning of the values in each row. For example, the first table of Figure 1.2 is called STUDENT because each row represents facts about a particular student entity. The column names—Name, StudentNumber, Class, and Major—specify how to interpret the data values in each row, based on the column each value is in. All values in a column are of the same data type.

In the formal relational model terminology, a row is called a *tuple*, a column header is called an *attribute*, and the table is called a *relation*. The data type describing the types of values that can appear in each column is represented by a *domain* of possible values. We now define these terms—*domain*, *tuple*, *attribute*, and *relation*—more precisely.

5.1.1 Domains, Attributes, Tuples, and Relations

A **domain** D is a set of atomic values. By **atomic** we mean that each value in the domain is indivisible as far as the relational model is concerned. A common method of specifying a domain is to specify a data type from which the data values forming the domain are drawn. It is also useful to specify a name for the domain, to help in interpreting its values. Some examples of domains follow:

- USA_phone_numbers: The set of ten-digit phone numbers valid in the United States.
- Local_phone_numbers: The set of seven-digit phone numbers valid within a particular area code in the United States.
- Social_security_numbers: The set of valid nine-digit social security numbers.
- Names: The set of character strings that represent names of persons.
- Grade_point_averages: Possible values of computed grade point averages; each must be a real (floating-point) number between 0 and 4.
- Employee_ages: Possible ages of employees of a company; each must be a value between 15 and 80 years old.
- Academic_department_names: The set of academic department names in a university, such as Computer Science, Economics, and Physics.
- Academic_department_codes: The set of academic department codes, such as CS, ECON, and PHYS.

The preceding are called *logical* definitions of domains. A **data type** or **format** is also specified for each domain. For example, the data type for the domain USA_phone_numbers can be declared as a character string of the form (*ddd*)*ddd-dddd*, where each *d* is a numeric (decimal) digit and the first three digits form a valid telephone area code. The data type for Employee_ages is an integer number between 15 and 80. For Academic_department_names, the data type is the set of all character strings that represent valid department names. A domain is thus given a name, data type, and format. Additional information for interpreting the values of a domain can also be given; for example, a numeric domain such as Person_weights should have the units of measurement, such as pounds or kilograms.

A **relation schema**[2] R, denoted by $R(A_1, A_2, \ldots, A_n)$, is made up of a relation name R and a list of attributes $A_1, A_2, \ldots, A_n$. Each **attribute** A_i is the name of a role played by some domain D in the relation schema R. D is called the **domain** of A_i and is denoted by **dom(A_i)**. A relation schema is used to *describe* a relation; R is called the **name** of this relation. The **degree** (or **arity**) of a relation is the number of attributes n of its relation schema.

2. A relation schema is sometimes called a **relation scheme**.

An example of a relation schema for a relation of degree seven, which describes university students, is the following:

STUDENT(Name, SSN, HomePhone, Address, OfficePhone, Age, GPA)

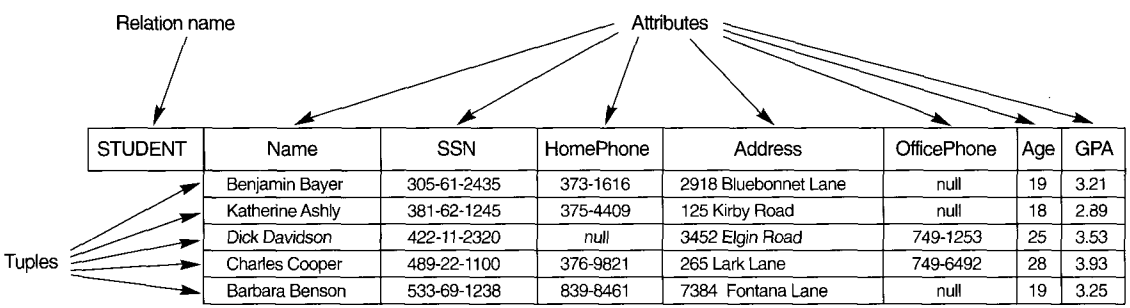
Using the data type of each attribute, the definition is sometimes written as:

STUDENT(Name: string, SSN: string, HomePhone: string, Address: string, OfficePhone: string, Age: integer, GPA: real)

For this relation schema, STUDENT is the name of the relation, which has seven attributes. In the above definition, we showed assignment of generic types such as string or integer to the attributes. More precisely, we can specify the following previously defined domains for some of the attributes of the STUDENT relation: dom(Name) = Names; dom(SSN) = Social_security_numbers; dom(HomePhone) = Local_phone_numbers,[3] dom(OfficePhone) = Local_phone_numbers, and dom(GPA) = Grade_point_averages. It is also possible to refer to attributes of a relation schema by their position within the relation; thus, the second attribute of the STUDENT relation is SSN, whereas the fourth attribute is Address.

A **relation** (or **relation state**)[4] r of the relation schema $R(A_1, A_2, \ldots, A_n)$, also denoted by $r(R)$, is a set of n-tuples $r = \{t_1, t_2, \ldots, t_m\}$. Each ***n*-tuple** t is an ordered list of n values $t = <v_1, v_2, \ldots, v_n>$, where each value v_i, $1 \leq i \leq n$, is an element of dom(A_i) or is a special **null** value. The i^{th} value in tuple t, which corresponds to the attribute A_i, is referred to as $t[A_i]$ (or $t[i]$ if we use the positional notation). The terms **relation intension** for the schema R and **relation extension** for a relation state $r(R)$ are also commonly used.

Figure 5.1 shows an example of a STUDENT relation, which corresponds to the STUDENT schema just specified. Each tuple in the relation represents a particular student entity. We



STUDENT	Name	SSN	HomePhone	Address	OfficePhone	Age	GPA
	Benjamin Bayer	305-61-2435	373-1616	2918 Bluebonnet Lane	null	19	3.21
	Katherine Ashly	381-62-1245	375-4409	125 Kirby Road	null	18	2.89
	Dick Davidson	422-11-2320	null	3452 Elgin Road	749-1253	25	3.53
	Charles Cooper	489-22-1100	376-9821	265 Lark Lane	749-6492	28	3.93
	Barbara Benson	533-69-1238	839-8461	7384 Fontana Lane	null	19	3.25

FIGURE 5.1 The attributes and tuples of a relation STUDENT.

3. With the large increase in phone numbers caused by the proliferation of mobile phones, some metropolitan areas now have multiple area codes, so that seven-digit local dialing has been discontinued. In this case, we would use USA_phone_numbers as the domain.

4. This has also been called a **relation instance.** We will not use this term because *instance* is also used to refer to a single tuple or row.

display the relation as a table, where each tuple is shown as a *row* and each attribute corresponds to a *column header* indicating a role or interpretation of the values in that column. *Null values* represent attributes whose values are unknown or do not exist for some individual STUDENT tuple.

The earlier definition of a relation can be *restated* more formally as follows. A relation (or relation state) $r(R)$ is a **mathematical relation** of degree n on the domains $\text{dom}(A_1)$, $\text{dom}(A_2)$, . . . , $\text{dom}(A_n)$, which is a **subset** of the **Cartesian product** of the domains that define R:

$$r(R) \subseteq (\text{dom}(A_1) \times \text{dom}(A_2) \times \ldots \times \text{dom}(A_n))$$

The Cartesian product specifies all possible combinations of values from the underlying domains. Hence, if we denote the total number of values, or **cardinality,** in a domain D by $|D|$ (assuming that all domains are finite), the total number of tuples in the Cartesian product is

$$|\text{dom}(A_1)| \times |\text{dom}(A_2)| \times \ldots \times |\text{dom}(A_n)|$$

Of all these possible combinations, a relation state at a given time—the **current relation state**—reflects only the valid tuples that represent a particular state of the real world. In general, as the state of the real world changes, so does the relation, by being transformed into another relation state. However, the schema R is relatively static and does *not* change except very infrequently—for example, as a result of adding an attribute to represent new information that was not originally stored in the relation.

It is possible for several attributes to *have the same domain*. The attributes indicate different **roles,** or interpretations, for the domain. For example, in the STUDENT relation, the same domain Local_phone_numbers plays the role of HomePhone, referring to the "home phone of a student," and the role of OfficePhone, referring to the "office phone of the student."

5.1.2 Characteristics of Relations

The earlier definition of relations implies certain characteristics that make a relation different from a file or a table. We now discuss some of these characteristics.

Ordering of Tuples in a Relation. A relation is defined as a *set* of tuples. Mathematically, elements of a set have *no order* among them; hence, tuples in a relation do not have any particular order. However, in a file, records are physically stored on disk (or in memory), so there always is an order among the records. This ordering indicates first, second, *i*th, and last records in the file. Similarly, when we display a relation as a table, the rows are displayed in a certain order.

Tuple ordering is not part of a relation definition, because a relation attempts to represent facts at a logical or abstract level. Many logical orders can be specified on a relation. For example, tuples in the STUDENT relation in Figure 5.1 could be logically ordered by values of Name, or by SSN, or by Age, or by some other attribute. The definition of a relation does not specify any order: There is *no preference* for one logical

ordering over another. Hence, the relation displayed in Figure 5.2 is considered *identical* to the one shown in Figure 5.1. When a relation is implemented as a file or displayed as a table, a particular ordering may be specified on the records of the file or the rows of the table.

Ordering of Values within a Tuple, and an Alternative Definition of a Relation. According to the preceding definition of a relation, an n-tuple is an *ordered list* of n values, so the ordering of values in a tuple—and hence of attributes in a relation schema—is important. However, at a logical level, the order of attributes and their values is *not* that important as long as the correspondence between attributes and values is maintained.

An **alternative definition** of a relation can be given, making the ordering of values in a tuple *unnecessary*. In this definition, a relation schema $R = \{A_1, A_2, \ldots, A_n\}$ is a *set* of attributes, and a relation state $r(R)$ is a finite set of mappings $r = \{t_1, t_2, \ldots, t_m\}$, where each tuple t_i is a **mapping** from R to D, and D is the union of the attribute domains; that is, $D = \text{dom}(A_1) \cup \text{dom}(A_2) \cup \ldots \cup \text{dom}(A_n)$. In this definition, $t[A_i]$ must be in $\text{dom}(A_i)$ for $1 \leq i \leq n$ for each mapping t in r. Each mapping t_i is called a tuple.

According to this definition of tuple as a mapping, a **tuple** can be considered as a **set** of (<attribute>, <value>) pairs, where each pair gives the value of the mapping from an attribute A_i to a value v_i from $\text{dom}(A_i)$. The ordering of attributes is *not* important, because the attribute name appears with its value. By this definition, the two tuples shown in Figure 5.3 are identical. This makes sense at an abstract or logical level, since there really is no reason to prefer having one attribute value appear before another in a tuple.

When a relation is implemented as a file, the attributes are physically ordered as fields within a record. We will generally use the **first definition** of relation, where the attributes and the values within tuples *are ordered*, because it simplifies much of the notation. However, the alternative definition given here is more general.[5]

Values and Nulls in the Tuples. Each value in a tuple is an **atomic** value; that is, it is not divisible into components within the framework of the basic relational model. Hence, composite and multivalued attributes (see Chapter 3) are not allowed. This

STUDENT	Name	SSN	HomePhone	Address	OfficePhone	Age	GPA
	Dick Davidson	422-11-2320	null	3452 Elgin Road	749-1253	25	3.53
	Barbara Benson	533-69-1238	839-8461	7384 Fontana Lane	null	19	3.25
	Charles Cooper	489-22-1100	376-9821	265 Lark Lane	749-6492	28	3.93
	Katherine Ashly	381-62-1245	375-4409	125 Kirby Road	null	18	2.89
	Benjamin Bayer	305-61-2435	373-1616	2918 Bluebonnet Lane	null	19	3.21

FIGURE 5.2 The relation STUDENT from Figure 5.1 with a different order of tuples.

5. As we shall see, the alternative definition of relation is useful when we discuss query processing in Chapters 15 and 16.

t = < (Name, Dick Davidson),(SSN, 422-11-2320),(HomePhone, null),(Address, 3452 Elgin Road),
(OfficePhone, 749-1253),(Age, 25),(GPA, 3.53)>

t = < (Address, 3452 Elgin Road),(Name, Dick Davidson),(SSN, 422-11-2320),(Age, 25),
(OfficePhone,749-1253),(GPA, 3.53),(HomePhone, null)>

FIGURE 5.3 Two identical tuples when the order of attributes and values is not part of relation definition.

model is sometimes called the **flat relational model.** Much of the theory behind the relational model was developed with this assumption in mind, which is called the **first normal form** assumption.[6] Hence, multivalued attributes must be represented by separate relations, and composite attributes are represented only by their simple component attributes in the basic relational model.[7]

An important concept is that of ***nulls***, which are used to represent the values of attributes that may be unknown or may not apply to a tuple. A special value, called **null,** is used for these cases. For example, in Figure 5.1, some student tuples have null for their office phones because they do not have an office (that is, office phone *does not apply* to these students). Another student has a null for home phone, presumably because either he does not have a home phone or he has one but we do not know it (value is *unknown*). In general, we can have *several meanings* for null values, such as "value unknown," "value exists but is not available," or "attribute does not apply to this tuple." An example of the last type of null will occur if we add an attribute Visa_status to the STUDENT relation that applies only to tuples that represent foreign students. It is possible to devise different codes for different meanings of null values. Incorporating different types of null values into the relational model operations (see Chapter 6) has proven difficult and is outside the scope of our presentation.

Interpretation (Meaning) of a Relation. The relation schema can be interpreted as a declaration or a type of **assertion.** For example, the schema of the STUDENT relation of Figure 5.1 asserts that, in general, a student entity has a Name, SSN, HomePhone, Address, OfficePhone, Age, and GPA. Each tuple in the relation can then be interpreted as a **fact** or a particular instance of the assertion. For example, the first tuple in Figure 5.1 asserts the fact that there is a student whose name is Benjamin Bayer, SSN is 305-61-2435, Age is 19, and so on.

Notice that some relations may represent facts about *entities*, whereas other relations may represent facts about *relationships*. For example, a relation schema MAJORS (StudentSSN, DepartmentCode) asserts that students major in academic departments. A tuple in this

6. We discuss this assumption in more detail in Chapter 10.

7. Extensions of the relational model remove these restrictions. For example, object-relational systems allow complex-structured attributes, as do the **non-first normal form** or **nested** relational models, as we shall see in Chapter 22.

relation relates a student to his or her major department. Hence, the relational model represents facts about both entities and relationships *uniformly* as relations. This sometimes compromises understandability because one has to guess whether a relation represents an entity type or a relationship type. The mapping procedures in Chapter 7 show how different constructs of the ER and EER models get converted to relations.

An alternative interpretation of a relation schema is as a **predicate;** in this case, the values in each tuple are interpreted as values that *satisfy* the predicate. This interpretation is quite useful in the context of logic programming languages, such as Prolog, because it allows the relational model to be used within these languages (see Section 24.4).

5.1.3 Relational Model Notation

We will use the following notation in our presentation:

- A relation schema R of degree n is denoted by $R(A_1, A_2, \ldots, A_n)$.
- An n-tuple t in a relation $r(R)$ is denoted by $t = <v_1, v_2, \ldots, v_n>$, where v_i is the value corresponding to attribute A_i. The following notation refers to **component values** of tuples:
  - Both $t[A_i]$ and $t.A_i$ (and sometimes $t[i]$) refer to the value v_i in t for attribute A_i.
  - Both $t[A_u, A_w, \ldots, A_z]$ and $t.(A_u, A_w, \ldots, A_z)$, where $A_u, A_w, \ldots, A_z$ is a list of attributes from R, refer to the subtuple of values $<v_u, v_w, \ldots, v_z>$ from t corresponding to the attributes specified in the list.
- The letters Q, R, S denote relation names.
- The letters q, r, s denote relation states.
- The letters t, u, v denote tuples.
- In general, the name of a relation schema such as STUDENT *also indicates* the current set of tuples in that relation—the *current relation state*—whereas STUDENT(Name, SSN, . . .) refers *only* to the relation schema.
- An attribute A can be qualified with the relation name R to which it belongs by using the *dot notation* $R.A$—for example, STUDENT.Name or STUDENT.Age. This is because the same name may be used for two attributes in different relations. However, all attribute names *in a particular relation* must be distinct.

As an example, consider the tuple t = <'Barbara Benson', '533-69-1238', '839-8461', '7384 Fontana Lane', null, 19, 3.25> from the STUDENT relation in Figure 5.1; we have t[Name] = <'Barbara Benson'>, and t[SSN, GPA, Age] = <'533-69-1238', 3.25, 19>.

5.2 Relational Model Constraints and Relational Database Schemas

So far, we have discussed the characteristics of single relations. In a relational database, there will typically be many relations, and the tuples in those relations are usually related

in various ways. The state of the whole database will correspond to the states of all its relations at a particular point in time. There are generally many restrictions or **constraints** on the actual values in a database state. These constraints are derived from the rules in the miniworld that the database represents, as we discussed in Section 1.6.8.

In this section, we discuss the various restrictions on data that can be specified on a relational database in the form of constraints. Constraints on databases can generally be divided into three main categories:

1. Constraints that are inherent in the data model. We call these **inherent model-based constraints.**
2. Constraints that can be directly expressed in the schemas of the data model, typically by specifying them in the DDL (data definition language, see Section 2.3.1). We call these **schema-based constraints.**
3. Constraints that *cannot* be directly expressed in the schemas of the data model, and hence must be expressed and enforced by the application programs. We call these **application-based constraints.**

The characteristics of relations that we discussed in Section 5.1.2 are the inherent constraints of the relational model and belong to the first category; for example, the constraint that a relation cannot have duplicate tuples is an inherent constraint. The constraints we discuss in this section are of the second category, namely, constraints that can be expressed in the schema of the relational model via the DDL. Constraints in the third category are more general and are difficult to express and enforce within the data model, so they are usually checked within application programs.

Another important category of constraints is *data dependencies*, which include *functional dependencies* and *multivalued dependencies*. They are used mainly for testing the "goodness" of the design of a relational database and are utilized in a process called *normalization*, which is discussed in Chapters 10 and 11.

We now discuss the main types of constraints that can be expressed in the relational model—the schema-based constraints from the second category. These include domain constraints, key constraints, constraints on nulls, entity integrity constraints, and referential integrity constraints.

5.2.1 Domain Constraints

Domain constraints specify that within each tuple, the value of each attribute A must be an atomic value from the domain dom(A). We have already discussed the ways in which domains can be specified in Section 5.1.1. The data types associated with domains typically include standard numeric data types for integers (such as short integer, integer, and long integer) and real numbers (float and double-precision float). Characters, booleans, fixed-length strings, and variable-length strings are also available, as are date, time, timestamp, and, in some cases, money data types. Other possible domains may be described by a subrange of values from a data type or as an enumerated data type in which all possible values are explicitly listed. Rather than describe these in detail here, we discuss the data types offered by the SQL-99 relational standard in Section 8.1.

5.2.2 Key Constraints and Constraints on Null Values

A *relation* is defined as a *set of tuples*. By definition, all elements of a set are distinct; hence, all tuples in a relation must also be distinct. This means that no two tuples can have the same combination of values for *all* their attributes. Usually, there are other **subsets of attributes** of a relation schema R with the property that no two tuples in any relation state r of R should have the same combination of values for these attributes. Suppose that we denote one such subset of attributes by SK; then for any two *distinct* tuples t_1 and t_2 in a relation state r of R, we have the constraint that

$$t_1[SK] \neq t_2[SK]$$

Any such set of attributes SK is called a **superkey** of the relation schema R. A superkey SK specifies a *uniqueness constraint* that no two distinct tuples in any state r of R can have the same value for SK. Every relation has at least one default superkey—the set of all its attributes. A superkey can have redundant attributes, however, so a more useful concept is that of a *key*, which has no redundancy. A **key** K of a relation schema R is a superkey of R with the additional property that removing any attribute A from K leaves a set of attributes K' that is not a superkey of R any more. Hence, a key satisfies two constraints:

1. Two distinct tuples in any state of the relation cannot have identical values for (all) the attributes in the key.
2. It is a *minimal superkey*—that is, a superkey from which we cannot remove any attributes and still have the uniqueness constraint in condition 1 hold.

The first condition applies to both keys and superkeys. The second condition is required only for keys. For example, consider the STUDENT relation of Figure 5.1. The attribute set {SSN} is a key of STUDENT because no two student tuples can have the same value for SSN.[8] Any set of attributes that includes SSN—for example, {SSN, Name, Age}—is a superkey. However, the superkey {SSN, Name, Age} is not a key of STUDENT, because removing Name or Age or both from the set still leaves us with a superkey. In general, any superkey formed from a single attribute is also a key. A key with multiple attributes must require *all* its attributes to have the uniqueness property hold.

The value of a key attribute can be used to identify uniquely each tuple in the relation. For example, the SSN value 305-61-2435 identifies uniquely the tuple corresponding to Benjamin Bayer in the STUDENT relation. Notice that a set of attributes constituting a key is a property of the relation schema; it is a constraint that should hold on *every* valid relation state of the schema. A key is determined from the meaning of the attributes, and the property is *time-invariant*: It must continue to hold when we insert new tuples in the relation. For example, we cannot and should not designate the Name attribute of the STUDENT relation in Figure 5.1 as a key, because it is possible that two students with identical names will exist at some point in a valid state.[9]

8. Note that SSN is also a superkey.

9. Names are sometimes used as keys, but then some artifact—such as appending an ordinal number—must be used to distinguish between identical names.

CAR	LicenseNumber	EngineSerialNumber	Make	Model	Year
	Texas ABC-739	A69352	Ford	Mustang	96
	Florida TVP-347	B43696	Oldsmobile	Cutlass	99
	New York MPO-22	X83554	Oldsmobile	Delta	95
	California 432-TFY	C43742	Mercedes	190-D	93
	California RSK-629	Y82935	Toyota	Camry	98
	Texas RSK-629	U028365	Jaguar	XJS	98

FIGURE 5.4 The CAR relation, with two candidate keys: LicenseNumber and EngineSerialNumber.

In general, a relation schema may have more than one key. In this case, each of the keys is called a **candidate key.** For example, the CAR relation in Figure 5.4 has two candidate keys: LicenseNumber and EngineSerialNumber. It is common to designate one of the candidate keys as the **primary key** of the relation. This is the candidate key whose values are used to *identify* tuples in the relation. We use the convention that the attributes that form the primary key of a relation schema are underlined, as shown in Figure 5.4. Notice that when a relation schema has several candidate keys, the choice of one to become the primary key is arbitrary; however, it is usually better to choose a primary key with a single attribute or a small number of attributes.

Another constraint on attributes specifies whether null values are or are not permitted. For example, if every STUDENT tuple must have a valid, nonnull value for the Name attribute, then Name of STUDENT is constrained to be NOT NULL.

5.2.3 Relational Databases and Relational Database Schemas

The definitions and constraints we have discussed so far apply to single relations and their attributes. A relational database usually contains many relations, with tuples in relations that are related in various ways. In this section we define a relational database and a relational database schema. A **relational database schema** S is a set of relation schemas $S = \{R_1, R_2, \ldots, R_m\}$ and a set of **integrity constraints** IC. A **relational database state**[10] DB of S is a set of relation states $DB = \{r_1, r_2, \ldots, r_m\}$ such that each r_i is a state of R_i and such that the r_i relation states satisfy the integrity constraints specified in IC. Figure 5.5 shows a relational database schema that we call COMPANY = {EMPLOYEE, DEPARTMENT, DEPT_LOCATIONS, PROJECT, WORKS_ON, DEPENDENT}. The underlined attributes represent primary keys. Figure 5.6 shows a relational database state corresponding to the COMPANY schema. We will use this schema and database state in this chapter and in Chapters 6 through 9 for developing example queries in different relational languages. When we refer to a relational database,

10. A relational database *state* is sometimes called a relational database *instance*. However, as we mentioned earlier, we will not use the term *instance* since it also applies to single tuples.

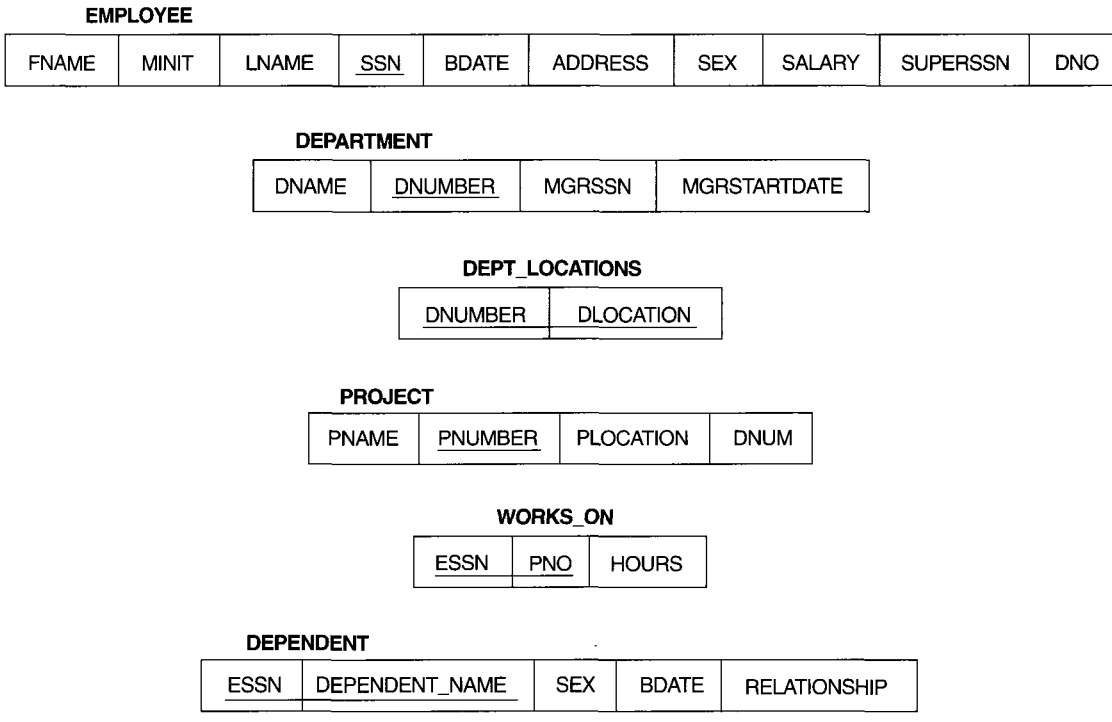


FIGURE 5.5 Schema diagram for the COMPANY relational database schema.

we implicitly include both its schema and its current state. A database state that does not obey all the integrity constraints is called an **invalid state,** and a state that satisfies all the constraints in IC is called a **valid state.**

In Figure 5.5, the DNUMBER attribute in both DEPARTMENT and DEPT_LOCATIONS stands for the same real-world concept—the number given to a department. That same concept is called DNO in EMPLOYEE and DNUM in PROJECT. Attributes that represent the same real-world concept may or may not have identical names in different relations. Alternatively, attributes that represent different concepts may have the same name in different relations. For example, we could have used the attribute name NAME for both PNAME of PROJECT and DNAME of DEPARTMENT; in this case, we would have two attributes that share the same name but represent different real-world concepts—project names and department names.

In some early versions of the relational model, an assumption was made that the same real-world concept, when represented by an attribute, would have *identical* attribute names in all relations. This creates problems when the same real-world concept is used in different roles (meanings) in the same relation. For example, the concept of social security number appears twice in the EMPLOYEE relation of Figure 5.5: once in the role of the employee's social security number, and once in the role of the supervisor's social security number. We gave them distinct attribute names—SSN and SUPERSSN, respectively—in order to distinguish their meaning.

EMPLOYEE	FNAME	MINIT	LNAME	SSN	BDATE	ADDRESS	SEX	SALARY	SUPERSSN	DNO
	John	B	Smith	123456789	1965-01-09	731 Fondren, Houston, TX	M	30000	333445555	5
	Franklin	T	Wong	333445555	1955-12-08	638 Voss, Houston, TX	M	40000	888665555	5
	Alicia	J	Zelaya	999887777	1968-01-19	3321 Castle, Spring, TX	F	25000	987654321	4
	Jennifer	S	Wallace	987654321	1941-06-20	291 Berry, Bellaire, TX	F	43000	888665555	4
	Ramesh	K	Narayan	666884444	1962-09-15	975 Fire Oak, Humble, TX	M	38000	333445555	5
	Joyce	A	English	453453453	1972-07-31	5631 Rice, Houston, TX	F	25000	333445555	5
	Ahmad	V	Jabbar	987987987	1969-03-29	980 Dallas, Houston, TX	M	25000	987654321	4
	James	E	Borg	888665555	1937-11-10	450 Stone, Houston, TX	M	55000	null	1

DEPT_LOCATIONS	DNUMBER	DLOCATION
	1	Houston
	4	Stafford
	5	Bellaire
	5	Sugarland
		Houston

DEPARTMENT	DNAME	DNUMBER	MGRSSN	MGRSTARTDATE
	Research	5	333445555	1988-05-22
	Administration	4	987654321	1995-01-01
	Headquarters	1	888665555	1981-06-19

WORKS_ON	ESSN	PNO	HOURS
	123456789	1	32.5
	123456789	2	7.5
	666884444	3	40.0
	453453453	1	20.0
	453453453	2	20.0
	333445555	2	10.0
	333445555	3	10.0
	333445555	10	10.0
	333445555	20	10.0
	999887777	30	30.0
	999887777	10	10.0
	987987987	10	35.0
	987987987	30	5.0
	987654321	30	20.0
	987654321	20	15.0
	888665555	20	null

PROJECT	PNAME	PNUMBER	PLOCATION	DNUM
	ProductX	1	Bellaire	5
	ProductY	2	Sugarland	5
	ProductZ	3	Houston	5
	Computerization	10	Stafford	4
	Reorganization	20	Houston	1
	Newbenefits	30	Stafford	4

DEPENDENT	ESSN	DEPENDENT_NAME	SEX	BDATE	RELATIONSHIP
	333445555	Alice	F	1986-04-05	DAUGHTER
	333445555	Theodore	M	1983-10-25	SON
	333445555	Joy	F	1958-05-03	SPOUSE
	987654321	Abner	M	1942-02-28	SPOUSE
	123456789	Michael	M	1988-01-04	SON
	123456789	Alice	F	1988-12-30	DAUGHTER
	123456789	Elizabeth	F	1967-05-05	SPOUSE

FIGURE 5.6 One possible database state for the COMPANY relational database schema.

Each relational DBMS must have a data definition language (DDL) for defining a relational database schema. Current relational DBMSs are mostly using SQL for this purpose. We present the SQL DDL in Sections 8.1 through 8.3.

Integrity constraints are specified on a database schema and are expected to hold on every valid database state of that schema. In addition to domain, key, and NOT NULL

constraints, two other types of constraints are considered part of the relational model: entity integrity and referential integrity.

5.2.4 Entity Integrity, Referential Integrity, and Foreign Keys

The **entity integrity constraint** states that no primary key value can be null. This is because the primary key value is used to identify individual tuples in a relation. Having null values for the primary key implies that we cannot identify some tuples. For example, if two or more tuples had null for their primary keys, we might not be able to distinguish them if we tried to reference them from other relations.

Key constraints and entity integrity constraints are specified on individual relations. The **referential integrity constraint** is specified between two relations and is used to maintain the consistency among tuples in the two relations. Informally, the referential integrity constraint states that a tuple in one relation that refers to another relation must refer to an *existing tuple* in that relation. For example, in Figure 5.6, the attribute DNO of EMPLOYEE gives the department number for which each employee works; hence, its value in every EMPLOYEE tuple must match the DNUMBER value of some tuple in the DEPARTMENT relation.

To define referential integrity more formally, we first define the concept of a *foreign key*. The conditions for a foreign key, given below, specify a referential integrity constraint between the two relation schemas R_1 and R_2. A set of attributes FK in relation schema R_1 is a **foreign key** of R_1 that **references** relation R_2 if it satisfies the following two rules:

1. The attributes in FK have the same domain(s) as the primary key attributes PK of R_2; the attributes FK are said to **reference** or **refer to** the relation R_2.
2. A value of FK in a tuple t_1 of the current state $r_1(R_1)$ either occurs as a value of PK for some tuple t_2 in the current state $r_2(R_2)$ *or is null.* In the former case, we have $t_1[FK] = t_2[PK]$, and we say that the tuple t_1 **references** or **refers to** the tuple t_2.

In this definition, R_1 is called the **referencing relation** and R_2 is the **referenced relation.** If these two conditions hold, a **referential integrity constraint** from R_1 to R_2 is said to hold. In a database of many relations, there are usually many referential integrity constraints.

To specify these constraints, we must first have a clear understanding of the meaning or role that each set of attributes plays in the various relation schemas of the database. Referential integrity constraints typically arise from the *relationships among the entities* represented by the relation schemas. For example, consider the database shown in Figure 5.6. In the EMPLOYEE relation, the attribute DNO refers to the department for which an employee works; hence, we designate DNO to be a foreign key of EMPLOYEE referring to the DEPARTMENT relation. This means that a value of DNO in any tuple t_1 of the EMPLOYEE relation must match a value of the primary key of DEPARTMENT—the DNUMBER attribute—in some tuple t_2 of the DEPARTMENT relation, or the value of DNO *can be null* if the employee does not belong

to a department. In Figure 5.6 the tuple for employee 'John Smith' references the tuple for the 'Research' department, indicating that 'John Smith' works for this department.

Notice that a foreign key can *refer to its own relation*. For example, the attribute SUPERSSN in EMPLOYEE refers to the supervisor of an employee; this is another employee, represented by a tuple in the EMPLOYEE relation. Hence, SUPERSSN is a foreign key that references the EMPLOYEE relation itself. In Figure 5.6 the tuple for employee 'John Smith' references the tuple for employee 'Franklin Wong,' indicating that 'Franklin Wong' is the supervisor of 'John Smith.'

We can *diagrammatically display referential integrity constraints* by drawing a directed arc from each foreign key to the relation it references. For clarity, the arrowhead may point to the primary key of the referenced relation. Figure 5.7 shows the schema in Figure 5.5 with the referential integrity constraints displayed in this manner.

All integrity constraints should be specified on the relational database schema if we want to enforce these constraints on the database states. Hence, the DDL includes provisions for specifying the various types of constraints so that the DBMS can automatically enforce them. Most relational DBMSs support key and entity integrity

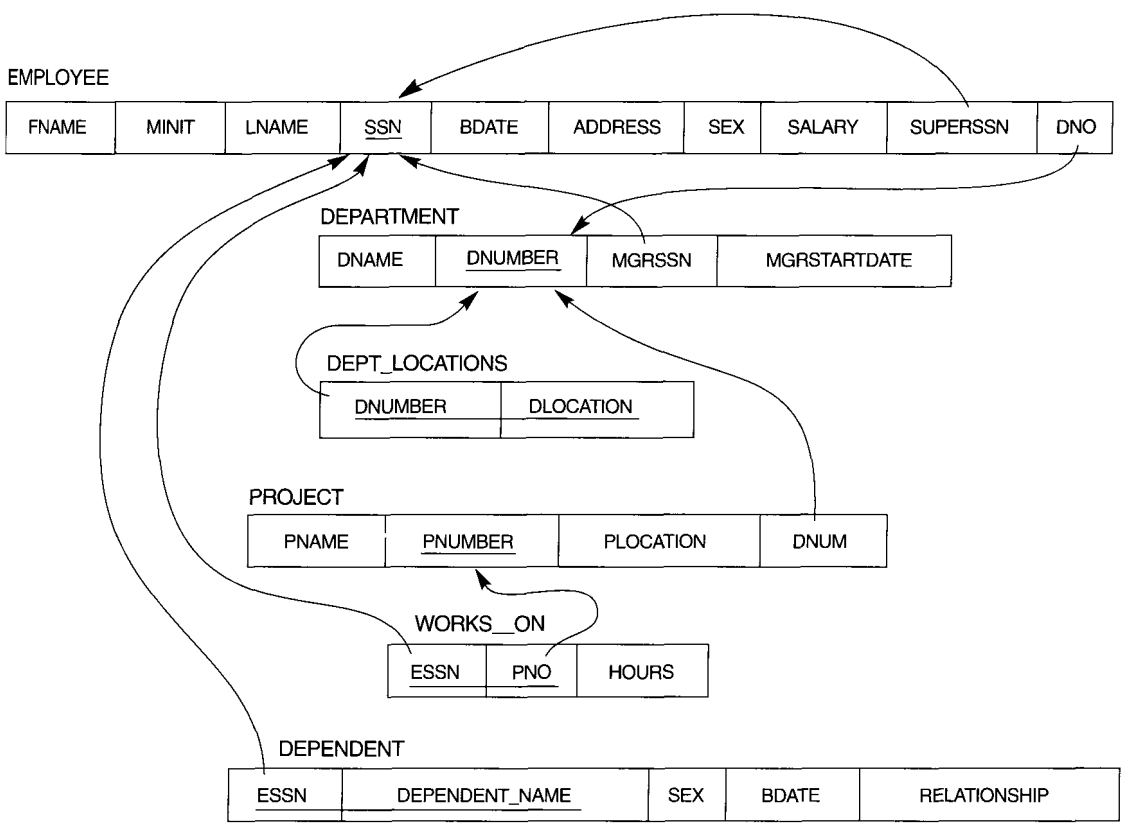


FIGURE 5.7 Referential integrity constraints displayed on the COMPANY relational database schema.

constraints, and make provisions to support referential integrity. These constraints are specified as a part of data definition.

5.2.5 Other Types of Constraints

The preceding integrity constraints do not include a large class of general constraints, sometimes called *semantic integrity constraints*, that may have to be specified and enforced on a relational database. Examples of such constraints are "the salary of an employee should not exceed the salary of the employee's supervisor" and "the maximum number of hours an employee can work on all projects per week is 56." Such constraints can be specified and enforced within the application programs that update the database, or by using a general-purpose **constraint specification language.** Mechanisms called **triggers** and **assertions** can be used. In SQL-99, a CREATE ASSERTION statement is used for this purpose (see Chapters 8 and 9). It is more common to check for these types of constraints within the application programs than to use constraint specification languages, because the latter are difficult and complex to use correctly, as we discuss in Section 24.1.

Another type of constraint is the *functional dependency* constraint, which establishes a functional relationship among two sets of attributes X and Y. This constraint specifies that the value of X determines the value of Y in all states of a relation; it is denoted as a functional dependency $X \rightarrow Y$. We use functional dependencies and other types of dependencies in Chapters 10 and 11 as tools to analyze the quality of relational designs and to "normalize" relations to improve their quality.

The types of constraints we discussed so far may be called **state constraints,** because they define the constraints that a *valid state* of the database must satisfy. Another type of constraint, called **transition constraints,** can be defined to deal with state changes in the database.[11] An example of a transition constraint is: "the salary of an employee can only increase." Such constraints are typically enforced by the application programs or specified using active rules and triggers, as we discuss in Section 24.1.

5.3 Update Operations and Dealing with Constraint Violations

The operations of the relational model can be categorized into *retrievals* and *updates*. The relational algebra operations, which can be used to specify **retrievals**, are discussed in detail in Chapter 6. A relational algebra expression forms a new relation after applying a number of algebraic operators to an existing set of relations; its main use is for querying a database. The user formulates a query that specifies the data of interest, and a new relation is formed by applying relational operators to retrieve this data. That relation

11. State constraints are sometimes called *static constraints*, and transition constraints are sometimes called *dynamic constraints*.

becomes the answer to the user's query. Chapter 6 also introduces the language called relational calculus, which is used to declaratively define the new relation without giving a specific order of operations.

In this section, we concentrate on the database **modification** or **update** operations. There are three basic update operations on relations: insert, delete, and modify. **Insert** is used to insert a new tuple or tuples in a relation, **Delete** is used to delete tuples, and **Update** (or **Modify**) is used to change the values of some attributes in existing tuples. Whenever these operations are applied, the integrity constraints specified on the relational database schema should not be violated. In this section we discuss the types of constraints that may be violated by each update operation and the types of actions that may be taken if an update does cause a violation. We use the database shown in Figure 5.6 for examples and discuss only key constraints, entity integrity constraints, and the referential integrity constraints shown in Figure 5.7. For each type of update, we give some example operations and discuss any constraints that each operation may violate.

5.3.1 The Insert Operation

The **Insert** operation provides a list of attribute values for a new tuple t that is to be inserted into a relation R. Insert can violate any of the four types of constraints discussed in the previous section. Domain constraints can be violated if an attribute value is given that does not appear in the corresponding domain. Key constraints can be violated if a key value in the new tuple t already exists in another tuple in the relation $r(R)$. Entity integrity can be violated if the primary key of the new tuple t is null. Referential integrity can be violated if the value of any foreign key in t refers to a tuple that does not exist in the referenced relation. Here are some examples to illustrate this discussion.

1. Insert <'Cecilia', 'F', 'Kolonsky', null, '1960-04-05', '6357 Windy Lane, Katy, TX', F, 28000, null, 4> into EMPLOYEE.
   - This insertion violates the entity integrity constraint (null for the primary key SSN), so it is rejected.
2. Insert <'Alicia', 'J', 'Zelaya', '999887777', '1960-04-05', '6357 Windy Lane, Katy, TX', F, 28000, '987654321', 4> into EMPLOYEE.
   - This insertion violates the key constraint because another tuple with the same SSN value already exists in the EMPLOYEE relation, and so it is rejected.
3. Insert <'Cecilia', 'F', 'Kolonsky', '677678989', '1960-04-05', '6357 Windswept, Katy, TX', F, 28000, '987654321', 7> into EMPLOYEE.
   - This insertion violates the referential integrity constraint specified on DNO because no DEPARTMENT tuple exists with DNUMBER = 7.
4. Insert <'Cecilia', 'F', 'Kolonsky', '677678989', '1960-04-05', '6357 Windy Lane, Katy, TX', F, 28000, null, 4> into EMPLOYEE.
   - This insertion satisfies all constraints, so it is acceptable.

If an insertion violates one or more constraints, the default option is to *reject the insertion*. In this case, it would be useful if the DBMS could explain to the user why the insertion was rejected. Another option is to attempt to *correct the reason for rejecting the insertion*, but this is typically not used for violations caused by Insert; rather, it is used more often in correcting violations for Delete and Update. In operation 1 above, the DBMS could ask the user to provide a value for SSN and could accept the insertion if a valid SSN value were provided. In operation 3, the DBMS could either ask the user to change the value of DNO to some valid value (or set it to null), or it could ask the user to insert a DEPARTMENT tuple with DNUMBER = 7 and could accept the original insertion only after such an operation was accepted. Notice that in the latter case the insertion violation can **cascade** back to the EMPLOYEE relation if the user attempts to insert a tuple for department 7 with a value for MGRSSN that does not exist in the EMPLOYEE relation.

5.3.2 The Delete Operation

The **Delete** operation can violate only referential integrity, if the tuple being deleted is referenced by the foreign keys from other tuples in the database. To specify deletion, a condition on the attributes of the relation selects the tuple (or tuples) to be deleted. Here are some examples.

1. Delete the WORKS_ON tuple with ESSN = '999887777' and PNO = 10.
   - This deletion is acceptable.
2. Delete the EMPLOYEE tuple with SSN = '999887777'.
   - This deletion is not acceptable, because tuples in WORKS_ON refer to this tuple. Hence, if the tuple is deleted, referential integrity violations will result.
3. Delete the EMPLOYEE tuple with SSN = '333445555'.
   - This deletion will result in even worse referential integrity violations, because the tuple involved is referenced by tuples from the EMPLOYEE, DEPARTMENT, WORKS_ON, and DEPENDENT relations.

Several options are available if a deletion operation causes a violation. The first option is to *reject the deletion*. The second option is to *attempt to cascade (or propagate) the deletion* by deleting tuples that reference the tuple that is being deleted. For example, in operation 2, the DBMS could automatically delete the offending tuples from WORKS_ON with ESSN = '999887777'. A third option is to *modify the referencing attribute values* that cause the violation; each such value is either set to null or changed to reference another valid tuple. Notice that if a referencing attribute that causes a violation is *part of the primary key*, it *cannot* be set to null; otherwise, it would violate entity integrity.

Combinations of these three options are also possible. For example, to avoid having operation 3 cause a violation, the DBMS may automatically delete all tuples from WORKS_ON and DEPENDENT with ESSN = '333445555'. Tuples in EMPLOYEE with SUPERSSN = '333445555' and the tuple in DEPARTMENT with MGRSSN = '333445555' can have their SUPERSSN and MGRSSN values changed to other valid values or to null. Although it may make sense to delete

automatically the WORKS_ON and DEPENDENT tuples that refer to an EMPLOYEE tuple, it may not make sense to delete other EMPLOYEE tuples or a DEPARTMENT tuple.

In general, when a referential integrity constraint is specified in the DDL, the DBMS will allow the user to *specify which of the options* applies in case of a violation of the constraint. We discuss how to specify these options in the SQL-99 DDL in Chapter 8.

5.3.3 The Update Operation

The **Update** (or **Modify**) operation is used to change the values of one or more attributes in a tuple (or tuples) of some relation *R*. It is necessary to specify a condition on the attributes of the relation to select the tuple (or tuples) to be modified. Here are some examples.

1. Update the SALARY of the EMPLOYEE tuple with SSN = '999887777' to 28000.
   - Acceptable.
2. Update the DNO of the EMPLOYEE tuple with SSN = '999887777' to 1.
   - Acceptable.
3. Update the DNO of the EMPLOYEE tuple with SSN = '999887777' to 7.
   - Unacceptable, because it violates referential integrity.
4. Update the SSN of the EMPLOYEE tuple with SSN = '999887777' to '987654321'.
   - Unacceptable, because it violates primary key and referential integrity constraints.

Updating an attribute that is neither a primary key nor a foreign key usually causes no problems; the DBMS need only check to confirm that the new value is of the correct data type and domain. Modifying a primary key value is similar to deleting one tuple and inserting another in its place, because we use the primary key to identify tuples. Hence, the issues discussed earlier in both Sections 5.3.1 (Insert) and 5.3.2 (Delete) come into play. If a foreign key attribute is modified, the DBMS must make sure that the new value refers to an existing tuple in the referenced relation (or is null). Similar options exist to deal with referential integrity violations caused by Update as those options discussed for the Delete operation. In fact, when a referential integrity constraint is specified in the DDL, the DBMS will allow the user to choose separate options to deal with a violation caused by Delete and a violation caused by Update (see Section 8.2).

5.4 SUMMARY

In this chapter we presented the modeling concepts, data structures, and constraints provided by the relational model of data. We started by introducing the concepts of domains, attributes, and tuples. We then defined a relation schema as a list of attributes that describe the structure of a relation. A relation, or relation state, is a set of tuples that conforms to the schema.

Several characteristics differentiate relations from ordinary tables or files. The first is that tuples in a relation are not ordered. The second involves the ordering of attributes in a relation schema and the corresponding ordering of values within a tuple. We gave an alternative definition of relation that does not require these two orderings, but we continued to use the first definition, which requires attributes and tuple values to be ordered, for convenience. We then discussed values in tuples and introduced null values to represent missing or unknown information.

We then classified database constraints into inherent model-based constraints, schema-based constraints and application-based constraints. We then discussed the schema constraints pertaining to the relational model, starting with domain constraints, then key constraints, including the concepts of superkey, candidate key, and primary key, and the NOT NULL constraint on attributes. We then defined relational databases and relational database schemas. Additional relational constraints include the entity integrity constraint, which prohibits primary key attributes from being null. The interrelation referential integrity constraint was then described, which is used to maintain consistency of references among tuples from different relations.

The modification operations on the relational model are Insert, Delete, and Update. Each operation may violate certain types of constraints. These operations were discussed in Section 5.3. Whenever an operation is applied, the database state after the operation is executed must be checked to ensure that no constraints have been violated.

Review Questions

5.1. Define the following terms: *domain*, *attribute*, *n-tuple*, *relation schema*, *relation state*, *degree of a relation*, *relational database schema*, *relational database state*.

5.2. Why are tuples in a relation not ordered?

5.3. Why are duplicate tuples not allowed in a relation?

5.4. What is the difference between a key and a superkey?

5.5. Why do we designate one of the candidate keys of a relation to be the primary key?

5.6. Discuss the characteristics of relations that make them different from ordinary tables and files.

5.7. Discuss the various reasons that lead to the occurrence of null values in relations.

5.8. Discuss the entity integrity and referential integrity constraints. Why is each considered important?

5.9. Define *foreign key*. What is this concept used for?

Exercises

5.10. Suppose that each of the following update operations is applied directly to the database state shown in Figure 5.6. Discuss *all* integrity constraints violated by each operation, if any, and the different ways of enforcing these constraints.

a. Insert <'Robert', 'F', 'Scott', '943775543', '1952-06-21', '2365 Newcastle Rd, Bellaire, TX', M, 58000, '888665555', 1> into EMPLOYEE.

b. Insert <'ProductA', 4, 'Bellaire', 2> into PROJECT.

c. Insert <'Production', 4, '943775543', '1998-10-01'> into DEPARTMENT.
d. Insert <'677678989', null, '40.0'> into WORKS_ON.
e. Insert <'453453453', 'John', M, '1970-12-12', 'SPOUSE'> into DEPENDENT.
f. Delete the WORKS_ON tuples with ESSN = '333445555'.
g. Delete the EMPLOYEE tuple with SSN = '987654321'.
h. Delete the PROJECT tuple with PNAME = 'ProductX'.
i. Modify the MGRSSN and MGRSTARTDATE of the DEPARTMENT tuple with DNUMBER = 5 to '123456789' and '1999-10-01', respectively.
j. Modify the SUPERSSN attribute of the EMPLOYEE tuple with SSN = '999887777' to '943775543'.
k. Modify the HOURS attribute of the WORKS_ON tuple with ESSN = '999887777' and PNO = 10 to '5.0'.

5.11. Consider the AIRLINE relational database schema shown in Figure 5.8, which describes a database for airline flight information. Each FLIGHT is identified by a flight NUMBER, and consists of one or more FLIGHT_LEGS with LEG_NUMBERS 1, 2, 3, and so on. Each leg has scheduled arrival and departure times and airports and has many LEG_INSTANCES–one for each DATE on which the flight travels. FARES are kept for each flight. For each leg instance, SEAT_RESERVATIONS are kept, as are the AIRPLANE used on the leg and the actual arrival and departure times and airports. An AIRPLANE is identified by an AIRPLANE_ID and is of a particular AIRPLANE_TYPE. CAN_LAND relates AIRPLANE_TYPES to the AIRPORTS in which they can land. An AIRPORT is identified by an AIRPORT_CODE. Consider an update for the AIRLINE database to enter a reservation on a particular flight or flight leg on a given date.
a. Give the operations for this update.
b. What types of constraints would you expect to check?
c. Which of these constraints are key, entity integrity, and referential integrity constraints, and which are not?
d. Specify all the referential integrity constraints that hold on the schema shown in Figure 5.8.

5.12. Consider the relation CLASS(Course#, Univ_Section#, InstructorName, Semester, BuildingCode, Room#, TimePeriod, Weekdays, CreditHours). This represents classes taught in a university, with unique Univ_Section#. Identify what you think should be various candidate keys, and write in your own words the constraints under which each candidate key would be valid.

5.13. Consider the following six relations for an order-processing database application in a company:

CUSTOMER(Cust#, Cname, City)

ORDER(Order#, Odate, Cust#, Ord_Amt)

ORDER_ITEM(Order#, Item#, Qty)

ITEM(Item#, Unit_price)

SHIPMENT(Order#, Warehouse#, Ship_date)

WAREHOUSE(Warehouse#, City)

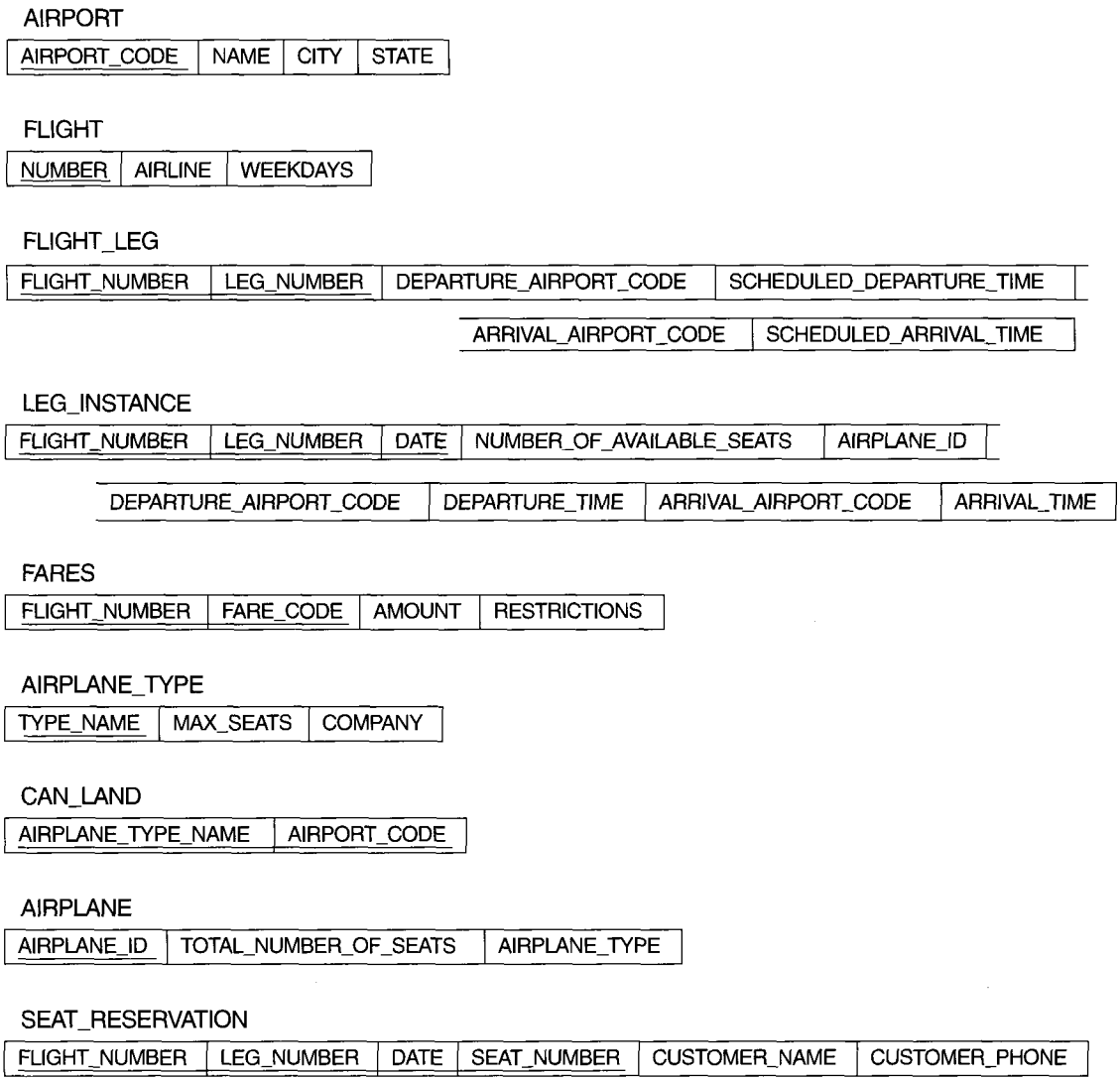


FIGURE 5.8 The AIRLINE relational database schema.

Here, Ord_Amt refers to total dollar amount of an order; Odate is the date the order was placed; Ship_date is the date an order is shipped from the warehouse. Assume that an order can be shipped from several warehouses. Specify the foreign keys for this schema, stating any assumptions you make.

5.14. Consider the following relations for a database that keeps track of business trips of salespersons in a sales office:

SALESPERSON(SSN, Name, Start_Year, Dept_No)

TRIP(SSN, From_City, To_City, Departure_Date, Return_Date, Trip_ID)

EXPENSE(Trip_ID, Account#, Amount)

Specify the foreign keys for this schema, stating any assumptions you make.

5.15. Consider the following relations for a database that keeps track of student enrollment in courses and the books adopted for each course:

STUDENT(SSN, Name, Major, Bdate)

COURSE(Course#, Cname, Dept)

ENROLL(SSN, Course#, Quarter, Grade)

BOOK_ADOPTION(Course#, Quarter, Book_ISBN)

TEXT(Book_ISBN, Book_Title, Publisher, Author)

Specify the foreign keys for this schema, stating any assumptions you make.

5.16. Consider the following relations for a database that keeps track of auto sales in a car dealership (Option refers to some optional equipment installed on an auto):

CAR(Serial-No, Model, Manufacturer, Price)

OPTIONS(Serial-No, Option-Name, Price)

SALES(Salesperson-id, Serial-No, Date, Sale-price)

SALESPERSON(Salesperson-id, Name, Phone)

First, specify the foreign keys for this schema, stating any assumptions you make. Next, populate the relations with a few example tuples, and then give an example of an insertion in the SALES and SALESPERSON relations that *violates* the referential integrity constraints and of another insertion that does not.

Selected Bibliography

The relational model was introduced by Codd (1970) in a classic paper. Codd also introduced relational algebra and laid the theoretical foundations for the relational model in a series of papers (Codd 1971, 1972, 1972a, 1974); he was later given the Turing award, the highest honor of the ACM, for his work on the relational model. In a later paper, Codd (1979) discussed extending the relational model to incorporate more meta-data and semantics about the relations; he also proposed a three-valued logic to deal with uncertainty in relations and incorporating NULLs in the relational algebra. The resulting model is known as RM/T. Childs (1968) had earlier used set theory to model databases. Later, Codd (1990) published a book examining over 300 features of the relational data model and database systems.

Since Codd's pioneering work, much research has been conducted on various aspects of the relational model. Todd (1976) describes an experimental DBMS called PRTV that directly implements the relational algebra operations. Schmidt and Swenson (1975) introduces additional semantics into the relational model by classifying different types of relations. Chen's (1976) entity-relationship model, which is discussed in Chapter 3, is a means to communicate the real-world semantics of a relational database at the conceptual level. Wiederhold and Elmasri (1979) introduces various types of connections

between relations to enhance its constraints. Extensions of the relational model are discussed in Chapter 24. Additional bibliographic notes for other aspects of the relational model and its languages, systems, extensions, and theory are given in Chapters 6 to 11, 15, 16, 17, and 22 to 25.

6

The Relational Algebra and Relational Calculus

In this chapter we discuss the two formal languages for the relational model: the relational algebra and the relational calculus. As we discussed in Chapter 2, a data model must include a set of operations to manipulate the database, in addition to the data model's concepts for defining database structure and constraints. The basic set of operations for the relational model is the **relational algebra.** These operations enable a user to specify basic retrieval requests. The result of a retrieval is a new relation, which may have been formed from one or more relations. The algebra operations thus produce new relations, which can be further manipulated using operations of the same algebra. A sequence of relational algebra operations forms a **relational algebra expression,** whose result will also be a relation that represents the result of a database query (or retrieval request).

The relational algebra is very important for several reasons. First, it provides a formal foundation for relational model operations. Second, and perhaps more important, it is used as a basis for implementing and optimizing queries in relational database management systems (RDBMSs), as we discuss in Part IV of the book. Third, some of its concepts are incorporated into the SQL standard query language for RDBMSs.

Whereas the algebra defines a set of operations for the relational model, the **relational calculus** provides a higher-level declarative notation for specifying relational queries. A relational calculus expression creates a new relation, which is specified in terms of variables that range over rows of the stored database relations (in tuple calculus) or over columns of the stored relations (in domain calculus). In a calculus expression, there is *no order of operations* to specify how to retrieve the query result—a calculus

expression specifies only what information the result should contain. This is the main distinguishing feature between relational algebra and relational calculus. The relational calculus is important because it has a firm basis in mathematical logic and because the SQL (standard query language) for RDBMSs has some of its foundations in the tuple relational calculus.[1]

The relational algebra is often considered to be an integral part of the relational data model, and its operations can be divided into two groups. One group includes set operations from mathematical set theory; these are applicable because each relation is defined to be a set of tuples in the formal relational model. Set operations include UNION, INTERSECTION, SET DIFFERENCE, and CARTESIAN PRODUCT. The other group consists of operations developed specifically for relational databases—these include SELECT, PROJECT, and JOIN, among others. We first describe the SELECT and PROJECT operations in Section 6.1, because they are **unary operations** that operate on single relations. Then we discuss set operations in Section 6.2. In Section 6.3, we discuss JOIN and other complex **binary operations,** which operate on two tables. The COMPANY relational database shown in Figure 5.6 is used for our examples.

Some common database requests cannot be performed with the original relational algebra operations, so additional operations were created to express these requests. These include **aggregate functions,** which are operations that can *summarize* data from the tables, as well as additional types of JOIN and UNION operations. These operations were added to the original relational algebra because of their importance to many database applications, and are described in Section 6.4. We give examples of specifying queries that use relational operations in Section 6.5. Some of these queries are used in subsequent chapters to illustrate various languages.

In Sections 6.6 and 6.7 we describe the other main formal language for relational databases, the **relational calculus.** There are two variations of relational calculus. The tuple relational calculus is described in Section 6.6, and the domain relational calculus is described in Section 6.7. Some of the SQL constructs discussed in Chapter 8 are based on the tuple relational calculus. The relational calculus is a formal language, based on the branch of mathematical logic called predicate calculus.[2] In tuple relational calculus, variables range over tuples, whereas in domain relational calculus, variables range over the domains (values) of attributes. In Appendix D we give an overview of the QBE (Query-By-Example) language, which is a graphical user-friendly relational language based on domain relational calculus. Section 6.8 summarizes the chapter.

For the reader who is interested in a less detailed introduction to formal relational languages, Sections 6.4, 6.6, and 6.7 may be skipped.

1. SQL is based on tuple relational calculus, but also incorporates some of the operations from the relational algebra and its extensions, as we shall see in Chapters 8 and 9.
2. In this chapter no familiarity with first-order predicate calculus—which deals with quantified variables and values—is assumed.

6.1 UNARY RELATIONAL OPERATIONS: SELECT AND PROJECT

6.1.1 The SELECT Operation

The SELECT operation is used to select a *subset* of the tuples from a relation that satisfy a **selection condition.** One can consider the SELECT operation to be a *filter* that keeps only those tuples that satisfy a qualifying condition. The SELECT operation can also be visualized as a *horizontal partition* of the relation into two sets of tuples—those tuples that satisfy the condition and are selected, and those tuples that do not satisfy the condition and are discarded. For example, to select the EMPLOYEE tuples whose department is 4, or those whose salary is greater than $30,000, we can individually specify each of these two conditions with a SELECT operation as follows:

$\sigma_{DNO=4}(EMPLOYEE)$

$\sigma_{SALARY>30000}(EMPLOYEE)$

In general, the SELECT operation is denoted by

$$\sigma_{<\text{selection condition}>}(R)$$

where the symbol σ (sigma) is used to denote the SELECT operator, and the selection condition is a Boolean expression specified on the attributes of relation R. Notice that R is generally a *relational algebra expression* whose result is a relation—the simplest such expression is just the name of a database relation. The relation resulting from the SELECT operation has the *same attributes* as R.

The Boolean expression specified in <selection condition> is made up of a number of **clauses** of the form

<attribute name> <comparison op> <constant value>,

or

<attribute name> <comparison op> <attribute name>

where <attribute name> is the name of an attribute of R, <comparison op> is normally one of the operators {=, <, ≤, >, ≥, ≠}, and <constant value> is a constant value from the attribute domain. Clauses can be arbitrarily connected by the Boolean operators AND, OR, and NOT to form a general selection condition. For example, to select the tuples for all employees who either work in department 4 and make over $25,000 per year, or work in department 5 and make over $30,000, we can specify the following SELECT operation:

$$\sigma_{(DNO=4 \text{ AND } SALARY>25000) \text{ OR } (DNO=5 \text{ AND } SALARY>30000)}(EMPLOYEE)$$

The result is shown in Figure 6.1a.

Notice that the comparison operators in the set {=, <, ≤, >, ≥, ≠} apply to attributes whose domains are *ordered values*, such as numeric or date domains. Domains of strings of characters are considered ordered based on the collating sequence of the characters. If the domain of an attribute is a set of *unordered values*, then only the comparison operators in the set {=, ≠} can be used. An example of an unordered domain is the domain Color = {red,

(a)

FNAME	MINIT	LNAME	SSN	BDATE	ADDRESS	SEX	SALARY	SUPERSSN	DNO
Franklin	T	Wong	333445555	1955-12-08	638 Voss,Houston,TX	M	40000	888665555	5
Jennifer		Wallace	987654321	1941-06-20	291 Berry,Bellaire,TX	F	43000	888665555	4
Ramesh		Narayan	666884444	1962-09-15	975 FireOak,Humble,TX	M	38000	333445555	5

(b)

LNAME	FNAME	SALARY
Smith	John	30000
Wong	Franklin	40000
Zelaya	Alicia	25000
Wallace	Jennifer	43000
Narayan	Ramesh	38000
English	Joyce	25000
Jabbar	Ahmad	25000
Borg	James	55000

(c)

SEX	SALARY
M	30000
M	40000
F	25000
F	43000
M	38000
M	25000
M	55000

FIGURE 6.1 Results of SELECT and PROJECT operations. (a) $\sigma_{(DNO=4 \text{ AND } SALARY>25000) \text{ OR } (DNO=5 \text{ AND } SALARY>30000)}(EMPLOYEE)$. (b) $\pi_{LNAME, FNAME, SALARY}(EMPLOYEE)$. (c) $\pi_{SEX, SALARY}(EMPLOYEE)$.

blue, green, white, yellow, . . .} where no order is specified among the various colors. Some domains allow additional types of comparison operators; for example, a domain of character strings may allow the comparison operator SUBSTRING_OF.

In general, the result of a SELECT operation can be determined as follows. The <selection condition> is applied independently to each tuple t in R. This is done by substituting each occurrence of an attribute A_i in the selection condition with its value in the tuple $t[A_i]$. If the condition evaluates to TRUE, then tuple t is **selected.** All the selected tuples appear in the result of the SELECT operation. The Boolean conditions AND, OR, and NOT have their normal interpretation, as follows:

- (cond1 AND cond2) is TRUE if both (cond1) and (cond2) are TRUE; otherwise, it is FALSE.
- (cond1 OR cond2) is TRUE if either (cond1) or (cond2) or both are TRUE; otherwise, it is FALSE.
- (NOT cond) is TRUE if cond is FALSE; otherwise, it is FALSE.

The SELECT operator is unary; that is, it is applied to a single relation. Moreover, the selection operation is applied to *each tuple individually*; hence, selection conditions cannot involve more than one tuple. The **degree** of the relation resulting from a SELECT operation—its number of attributes—is the same as the degree of R. The number of tuples in the resulting relation is always *less than or equal to* the number of tuples in R. That is, $|\sigma_c (R)| \leq |R|$ for any condition C. The fraction of tuples selected by a selection condition is referred to as the **selectivity** of the condition.

Notice that the SELECT operation is **commutative;** that is,

$$\sigma_{<cond1>}(\sigma_{<cond2>}(R)) = \sigma_{<cond2>}(\sigma_{<cond1>}(R))$$

Hence, a sequence of SELECTs can be applied in any order. In addition, we can always combine a **cascade** of SELECT operations into a single SELECT operation with a conjunctive (AND) condition; that is:

$$\sigma_{<cond1>}(\sigma_{<cond2>}(\ldots(\sigma_{<condn>}(R))\ldots)) = \sigma_{<cond1> \text{ AND } <cond2> \text{ AND } \ldots \text{ AND } <condn>}(R)$$

6.1.2 The PROJECT Operation

If we think of a relation as a table, the SELECT operation selects some of the *rows* from the table while discarding other rows. The **PROJECT** operation, on the other hand, selects certain *columns* from the table and discards the other columns. If we are interested in only certain attributes of a relation, we use the PROJECT operation to *project* the relation over these attributes only. The result of the PROJECT operation can hence be visualized as a *vertical partition* of the relation into two relations: one has the needed columns (attributes) and contains the result of the operation, and the other contains the discarded columns. For example, to list each employee's first and last name and salary, we can use the PROJECT operation as follows:

$$\pi_{\text{LNAME, FNAME, SALARY}}(\text{EMPLOYEE})$$

The resulting relation is shown in Figure 6.1(b). The general form of the PROJECT operation is

$$\pi_{<attribute\ list>}(R)$$

where π (pi) is the symbol used to represent the PROJECT operation, and <attribute list> is the desired list of attributes from the attributes of relation R. Again, notice that R is, in general, a *relational algebra expression* whose result is a relation, which in the simplest case is just the name of a database relation. The result of the PROJECT operation has only the attributes specified in <attribute list> *in the same order as they appear in the list*. Hence, its **degree** is equal to the number of attributes in <attribute list>.

If the attribute list includes only nonkey attributes of R, duplicate tuples are likely to occur. The PROJECT operation *removes any duplicate tuples*, so the result of the PROJECT operation is a set of tuples, and hence a valid relation.[3] This is known as **duplicate elimination.** For example, consider the following PROJECT operation:

$$\pi_{\text{SEX, SALARY}}(\text{EMPLOYEE})$$

The result is shown in Figure 6.1c. Notice that the tuple <F, 25000> appears only once in Figure 6.1c, even though this combination of values appears twice in the EMPLOYEE relation.

The number of tuples in a relation resulting from a PROJECT operation is always less than or equal to the number of tuples in R. If the projection list is a superkey of R—that

3. If duplicates are not eliminated, the result would be a **multiset** or **bag** of tuples rather than a set. Although this is not allowed in the formal relation model, it is permitted in practice. We shall see in Chapter 8 that SQL allows the user to specify whether duplicates should be eliminated or not.

is, it includes some key of R—the resulting relation has the *same number* of tuples as R. Moreover,

$$\pi_{<list1>}(\pi_{<list2>}(R)) = \pi_{<list1>}(R)$$

as long as <list2> contains the attributes in <list1>; otherwise, the left-hand side is an incorrect expression. It is also noteworthy that commutativity *does not* hold on PROJECT.

6.1.3 Sequences of Operations and the RENAME Operation

The relations shown in Figure 6.1 do not have any names. In general, we may want to apply several relational algebra operations one after the other. Either we can write the operations as a single **relational algebra expression** by nesting the operations, or we can apply one operation at a time and create intermediate result relations. In the latter case, we must give names to the relations that hold the intermediate results. For example, to retrieve the first name, last name, and salary of all employees who work in department number 5, we must apply a SELECT and a PROJECT operation. We can write a single relational algebra expression as follows:

$$\pi_{FNAME,\ LNAME,\ SALARY}(\sigma_{DNO=5}(EMPLOYEE))$$

Figure 6.2a shows the result of this relational algebra expression. Alternatively, we can explicitly show the sequence of operations, giving a name to each intermediate relation:

$$DEP5\_EMPS \leftarrow \sigma_{DNO=5}(EMPLOYEE)$$

$$RESULT \leftarrow \pi_{FNAME,\ LNAME,\ SALARY}(DEP5\_EMPS)$$

(a)

FNAME	LNAME	SALARY
John	Smith	30000
Franklin	Wong	40000
Ramesh	Narayan	38000
Joyce	English	25000

(b)

TEMP	FNAME	MINIT	LNAME	SSN	BDATE	ADDRESS	SEX	SALARY	SUPERSSN	DNO
	John	B	Smith	123456789	1965-01-09	731 Fondren,Houston,TX	M	30000	333445555	5
	Franklin	T	Wong	333445555	1955-12-08	638 Voss,Houston,TX	M	40000	888665555	5
	Ramesh	K	Narayan	666884444	1962-09-15	975 Fire Oak,Humble,TX	M	38000	333445555	5
	Joyce	A	English	453453453	1972-07-31	5631 Rice,Houston,TX	F	25000	333445555	5

R	FIRSTNAME	LASTNAME	SALARY
	John	Smith	30000
	Franklin	Wong	40000
	Ramesh	Narayan	38000
	Joyce	English	25000

FIGURE 6.2 Results of a sequence of operations. (a) $\pi_{FNAME,\ LNAME,\ SALARY}(\sigma_{DNO=5}(EMPLOYEE))$. (b) Using intermediate relations and renaming of attributes.

It is often simpler to break down a complex sequence of operations by specifying intermediate result relations than to write a single relational algebra expression. We can also use this technique to **rename** the attributes in the intermediate and result relations. This can be useful in connection with more complex operations such as UNION and JOIN, as we shall see. To rename the attributes in a relation, we simply list the new attribute names in parentheses, as in the following example:

$$\text{TEMP} \leftarrow \sigma_{\text{DNO}=5}(\text{EMPLOYEE})$$

$$\text{R}(\text{FIRSTNAME}, \text{LASTNAME}, \text{SALARY}) \leftarrow \pi_{\text{FNAME}, \text{LNAME}, \text{SALARY}}(\text{TEMP})$$

These two operations are illustrated in Figure 6.2b.

If no renaming is applied, the names of the attributes in the resulting relation of a SELECT operation are the same as those in the original relation and in the same order. For a PROJECT operation with no renaming, the resulting relation has the same attribute names as those in the projection list and in the same order in which they appear in the list.

We can also define a formal **RENAME** operation—which can rename either the relation name or the attribute names, or both—in a manner similar to the way we defined SELECT and PROJECT. The general RENAME operation when applied to a relation R of degree n is denoted by any of the following three forms

$$\rho_{S(B_1, B_2, \ldots, B_n)}(R) \text{ or } \rho_S(R) \text{ or } \rho_{(B_1, B_2, \ldots, B_n)}(R)$$

where the symbol ρ (rho) is used to denote the RENAME operator, S is the new relation name, and $B_1, B_2, \ldots, B_n$ are the new attribute names. The first expression renames both the relation and its attributes, the second renames the relation only, and the third renames the attributes only. If the attributes of R are $(A_1, A_2, \ldots, A_n)$ in that order, then each A_i is renamed as B_i.

6.2 Relational Algebra Operations from Set Theory

6.2.1 The UNION, INTERSECTION, and MINUS Operations

The next group of relational algebra operations are the standard mathematical operations on sets. For example, to retrieve the social security numbers of all employees who either work in department 5 or directly supervise an employee who works in department 5, we can use the UNION operation as follows:

$$\text{DEP5\_EMPS} \leftarrow \sigma_{\text{DNO}=5}(\text{EMPLOYEE})$$

$$\text{RESULT1} \leftarrow \pi_{\text{SSN}}(\text{DEP5\_EMPS})$$

$$\text{RESULT2}(\text{SSN}) \leftarrow \pi_{\text{SUPERSSN}}(\text{DEP5\_EMPS})$$

$$\text{RESULT} \leftarrow \text{RESULT1} \cup \text{RESULT2}$$

The relation RESULT1 has the social security numbers of all employees who work in department 5, whereas RESULT2 has the social security numbers of all employees who

directly supervise an employee who works in department 5. The UNION operation produces the tuples that are in either RESULT1 or RESULT2 or both (see Figure 6.3). Thus, the SSN value 333445555 appears only once in the result.

Several set theoretic operations are used to merge the elements of two sets in various ways, including **UNION**, **INTERSECTION**, and **SET DIFFERENCE** (also called **MINUS**). These are binary operations; that is, each is applied to two sets (of tuples). When these operations are adapted to relational databases, the two relations on which any of these three operations are applied must have the same **type of tuples;** this condition has been called *union compatibility*. Two relations $R(A_1, A_2, \ldots, A_n)$ and $S(B_1, B_2, \ldots, B_n)$ are said to be **union compatible** if they have the same degree n and if $\text{dom}(A_i) = \text{dom}(B_i)$ for $1 \leq i \leq n$. This means that the two relations have the same number of attributes, and each corresponding pair of attributes has the same domain.

We can define the three operations UNION, INTERSECTION, and SET DIFFERENCE on two union-compatible relations R and S as follows:

- **union**: The result of this operation, denoted by $R \cup S$, is a relation that includes all tuples that are either in R or in S or in both R and S. Duplicate tuples are eliminated.
- **intersection**: The result of this operation, denoted by $R \cap S$, is a relation that includes all tuples that are in both R and S.
- **set difference** (or **MINUS**): The result of this operation, denoted by $R - S$, is a relation that includes all tuples that are in R but not in S.

We will adopt the convention that the resulting relation has the same attribute names as the *first* relation R. It is always possible to rename the attributes in the result using the rename operator.

Figure 6.4 illustrates the three operations. The relations STUDENT and INSTRUCTOR in Figure 6.4a are union compatible, and their tuples represent the names of students and instructors, respectively. The result of the UNION operation in Figure 6.4b shows the names of all students and instructors. Note that duplicate tuples appear only once in the result. The result of the INTERSECTION operation (Figure 6.4c) includes only those who are both students and instructors.

Notice that both UNION and INTERSECTION are *commutative operations*; that is,

$$R \cup S = S \cup R, \quad \text{and} \quad R \cap S = S \cap R$$

RESULT1	SSN
	123456789
	333445555
	666884444
	453453453

RESULT2	SSN
	333445555
	888665555

RESULT	SSN
	123456789
	333445555
	666884444
	453453453
	888665555

FIGURE 6.3 Result of the UNION operation RESULT ← RESULT1 ∪ RESULT2.

(a)

STUDENT	FN	LN
	Susan	Yao
	Ramesh	Shah
	Johnny	Kohler
	Barbara	Jones
	Amy	Ford
	Jimmy	Wang
	Ernest	Gilbert

INSTRUCTOR	FNAME	LNAME
	John	Smith
	Ricardo	Browne
	Susan	Yao
	Francis	Johnson
	Ramesh	Shah

(b)

FN	LN
Susan	Yao
Ramesh	Shah
Johnny	Kohler
Barbara	Jones
Amy	Ford
Jimmy	Wang
Ernest	Gilbert
John	Smith
Ricardo	Browne
Francis	Johnson

(c)

FN	LN
Susan	Yao
Ramesh	Shah

(d)

FN	LN
Johnny	Kohler
Barbara	Jones
Amy	Ford
Jimmy	Wang
Ernest	Gilbert

(e)

FNAME	LNAME
John	Smith
Ricardo	Browne
Francis	Johnson

FIGURE 6.4 The set operations UNION, INTERSECTION, and MINUS. (a) Two union-compatible relations. (b) STUDENT $\cup$ INSTRUCTOR. (c) STUDENT $\cap$ INSTRUCTOR. (d) STUDENT $-$ INSTRUCTOR. (e) INSTRUCTOR $-$ STUDENT.

Both UNION and INTERSECTION can be treated as n-ary operations applicable to any number of relations because both are *associative operations*; that is,

$$R \cup (S \cup T) = (R \cup S) \cup T, \quad \text{and} \quad (R \cap S) \cap T = R \cap (S \cap T)$$

The MINUS operation is *not commutative*; that is, in general,

$$R - S \neq S - R$$

Figure 6.4d shows the names of students who are not instructors, and Figure 6.4e shows the names of instructors who are not students.

6.2.2 The CARTESIAN PRODUCT (or CROSS PRODUCT) Operation

Next we discuss the CARTESIAN PRODUCT operation—also known as CROSS PRODUCT or CROSS JOIN—which is denoted by ×. This is also a binary set operation, but the relations on which it is applied do *not* have to be union compatible. This operation is used to combine tuples from two relations in a combinatorial fashion. In general, the result of $R(A_1, A_2, \ldots, A_n) \times S(B_1, B_2, \ldots, B_m)$ is a relation Q with degree $n + m$ attributes $Q(A_1, A_2, \ldots, A_n, B_1, B_2, \ldots, B_m)$, in that order. The resulting relation Q has one tuple for each combination of tuples—one from R and one from S. Hence, if R has n_R tuples (denoted as $|R| = n_R$), and S has n_S tuples, then $R \times S$ will have $n_R * n_S$ tuples.

The operation applied by itself is generally meaningless. It is useful when followed by a selection that matches values of attributes coming from the component relations. For example, suppose that we want to retrieve a list of names of each female employee's dependents. We can do this as follows:

FEMALE_EMPS ← $\sigma_{SEX='F'}$(EMPLOYEE)

EMPNAMES ← $\pi_{FNAME, LNAME, SSN}$(FEMALE_EMPS)

EMP_DEPENDENTS ← EMPNAMES × DEPENDENT

ACTUAL_DEPENDENTS ← $\sigma_{SSN=ESSN}$(EMP_DEPENDENTS)

RESULT ← $\pi_{FNAME, LNAME, DEPENDENT\_NAME}$(ACTUAL_DEPENDENTS)

The resulting relations from this sequence of operations are shown in Figure 6.5. The EMP_DEPENDENTS relation is the result of applying the CARTESIAN PRODUCT operation to EMPNAMES from Figure 6.5 with DEPENDENT from Figure 5.6. In EMP_DEPENDENTS, every tuple from EMPNAMES is combined with every tuple from DEPENDENT, giving a result that is not very meaningful. We want to combine a female employee tuple only with her particular dependents—namely, the DEPENDENT tuples whose ESSN values match the SSN value of the EMPLOYEE tuple. The ACTUAL_DEPENDENTS relation accomplishes this. The EMP_DEPENDENTS relation is a good example of the case where relational algebra can be correctly applied to yield results that make no sense at all. It is therefore the responsibility of the user to make sure to apply only meaningful operations to relations.

The CARTESIAN PRODUCT creates tuples with the combined attributes of two relations. We can then SELECT only related tuples from the two relations by specifying an appropriate selection condition, as we did in the preceding example. Because this sequence of CARTESIAN PRODUCT followed by SELECT is used quite commonly to identify and select related tuples from two relations, a special operation, called JOIN, was created to specify this sequence as a single operation. We discuss the JOIN operation next.

6.3 BINARY RELATIONAL OPERATIONS: JOIN AND DIVISION

6.3.1 The JOIN Operation

The JOIN operation, denoted by ⋈, is used to combine *related tuples* from two relations into single tuples. This operation is very important for any relational database with more

FEMALE_EMPS

FNAME	MINIT	LNAME	SSN	BDATE	ADDRESS	SEX	SALARY	SUPERSSN	DNO
Alicia	J	Zelaya	999887777	1968-07-19	3321 Castle,Spring,TX	F	25000	987654321	4
Jennifer	S	Wallace	987654321	1941-06-20	291 Berry,Bellaire,TX	F	43000	888665555	4
Joyce	A	English	453453453	1972-07-31	5631 Rice,Houston,TX	F	25000	333445555	5

EMPNAMES

FNAME	LNAME	SSN
Alicia	Zelaya	999887777
Jennifer	Wallace	987654321
Joyce	English	453453453

EMP_DEPENDENTS

FNAME	LNAME	SSN	ESSN	DEPENDENT_NAME	SEX	BDATE	• • •
Alicia	Zelaya	999887777	333445555	Alice	F	1986-04-05	• • •
Alicia	Zelaya	999887777	333445555	Theodore	M	1983-10-25	• • •
Alicia	Zelaya	999887777	333445555	Joy	F	1958-05-03	• • •
Alicia	Zelaya	999887777	987654321	Abner	M	1942-02-28	• • •
Alicia	Zelaya	999887777	123456789	Michael	M	1988-01-04	• • •
Alicia	Zelaya	999887777	123456789	Alice	F	1988-12-30	• • •
Alicia	Zelaya	999887777	123456789	Elizabeth	F	1967-05-05	• • •
Jennifer	Wallace	987654321	333445555	Alice	F	1986-04-05	• • •
Jennifer	Wallace	987654321	333445555	Theodore	M	1983-10-25	• • •
Jennifer	Wallace	987654321	333445555	Joy	F	1958-05-03	• • •
Jennifer	Wallace	987654321	987654321	Abner	M	1942-02-28	• • •
Jennifer	Wallace	987654321	123456789	Michael	M	1988-01-04	• • •
Jennifer	Wallace	987654321	123456789	Alice	F	1988-12-30	• • •
Jennifer	Wallace	987654321	123456789	Elizabeth	F	1967-05-05	• • •
Joyce	English	453453453	333445555	Alice	F	1986-04-05	• • •
Joyce	English	453453453	333445555	Theodore	M	1983-10-25	• • •
Joyce	English	453453453	333445555	Joy	F	1958-05-03	• • •
Joyce	English	453453453	987654321	Abner	M	1942-02-28	• • •
Joyce	English	453453453	123456789	Michael	M	1988-01-04	• • •
Joyce	English	453453453	123456789	Alice	F	1988-12-30	• • •
Joyce	English	453453453	123456789	Elizabeth	F	1967-05-05	• • •

ACTUAL_DEPENDENTS

FNAME	LNAME	SSN	ESSN	DEPENDENT_NAME	SEX	BDATE	• • •
Jennifer	Wallace	987654321	987654321	Abner	M	1942-02-28	• • •

RESULT

FNAME	LNAME	DEPENDENT_NAME
Jennifer	Wallace	Abner

FIGURE 6.5 The CARTESIAN PRODUCT (CROSS PRODUCT) operation.

than a single relation, because it allows us to process relationships among relations. To illustrate JOIN, suppose that we want to retrieve the name of the manager of each department. To get the manager's name, we need to combine each department tuple with the employee tuple whose SSN value matches the MGRSSN value in the department tuple. We do

this by using the JOIN operation, and then projecting the result over the necessary attributes, as follows:

$$\text{DEPT\_MGR} \leftarrow \text{DEPARTMENT} \bowtie_{\text{MGRSSN=SSN}} \text{EMPLOYEE}$$

$$\text{RESULT} \leftarrow \pi_{\text{DNAME, LNAME, FNAME}}(\text{DEPT\_MGR})$$

The first operation is illustrated in Figure 6.6. Note that MGRSSN is a foreign key and that the referential integrity constraint plays a role in having matching tuples in the referenced relation EMPLOYEE.

The JOIN operation can be stated in terms of a CARTESIAN PRODUCT followed by a SELECT operation. However, JOIN is very important because it is used very frequently when specifying database queries. Consider the example we gave earlier to illustrate CARTESIAN PRODUCT, which included the following sequence of operations:

$$\text{EMP\_DEPENDENTS} \leftarrow \text{EMPNAMES} \times \text{DEPENDENT}$$

$$\text{ACTUAL\_DEPENDENTS} \leftarrow \sigma_{\text{SSN=ESSN}}(\text{EMP\_DEPENDENTS})$$

These two operations can be replaced with a single JOIN operation as follows:

$$\text{ACTUAL\_DEPENDENTS} \leftarrow \text{EMPNAMES} \bowtie_{\text{SSN=ESSN}} \text{DEPENDENT}$$

The general form of a JOIN operation on two relations[4] $R(A_1, A_2, \ldots, A_n)$ and $S(B_1, B_2, \ldots, B_m)$ is

$$R \bowtie_{\text{<join condition>}} S$$

The result of the JOIN is a relation Q with $n + m$ attributes $Q(A_1, A_2, \ldots, A_n, B_1, B_2, \ldots, B_m)$ in that order; Q has one tuple for each combination of tuples—one from R and one from S—*whenever the combination satisfies the join condition*. This is the main difference between CARTESIAN PRODUCT and JOIN. In JOIN, only combinations of tuples *satisfying the join condition* appear in the result, whereas in the CARTESIAN PRODUCT *all* combinations of tuples are included in the result. The join condition is specified on attributes from the two relations R and S and is evaluated for each combination of tuples. Each tuple combination for which the join condition evaluates to TRUE is included in the resulting relation Q *as a single combined tuple*.

A general join condition is of the form

<condition> AND <condition> AND . . . AND <condition>

DEPT_MGR	DNAME	DNUMBER	MGRSSN	• • •	FNAME	MINIT	LNAME	SSN	• • •
	Research	5	333445555	• • •	Franklin	T	Wong	333445555	• • •
	Administration	4	987654321	• • •	Jennifer	S	Wallace	987654321	• • •
	Headquarters	1	888665555	• • •	James	E	Borg	888665555	• • •

FIGURE 6.6 Result of the JOIN operation $\text{DEPT\_MGR} \leftarrow \text{DEPARTMENT} \bowtie_{\text{MGRSSN=SSN}} \text{EMPLOYEE}$.

4. Again, notice that R and S can be any relations that result from general *relational algebra expressions*.

where each condition is of the form $A_i \theta B_j$, A_i is an attribute of R, B_j is an attribute of S, A_i and B_j have the same domain, and θ (theta) is one of the comparison operators $\{=, <, \leq, >, \geq, \neq\}$. A JOIN operation with such a general join condition is called a **THETA JOIN**. Tuples whose join attributes are null *do not* appear in the result. In that sense, the JOIN operation does *not* necessarily preserve all of the information in the participating relations.

6.3.2 The EQUIJOIN and NATURAL JOIN Variations of JOIN

The most common use of JOIN involves join conditions with equality comparisons only. Such a JOIN, where the only comparison operator used is =, is called an **EQUIJOIN**. Both examples we have considered were EQUIJOINs. Notice that in the result of an EQUIJOIN we always have one or more pairs of attributes that have *identical values* in every tuple. For example, in Figure 6.6, the values of the attributes MGRSSN and SSN are identical in every tuple of DEPT_MGR because of the equality join condition specified on these two attributes. Because one of each pair of attributes with identical values is superfluous, a new operation called **NATURAL JOIN**—denoted by *—was created to get rid of the second (superfluous) attribute in an EQUIJOIN condition.[5] The standard definition of NATURAL JOIN requires that the two join attributes (or each pair of join attributes) have the same name in both relations. If this is not the case, a renaming operation is applied first.

In the following example, we first rename the DNUMBER attribute of DEPARTMENT to DNUM—so that it has the same name as the DNUM attribute in PROJECT—and then apply NATURAL JOIN:

$$\text{PROJ\_DEPT} \leftarrow \text{PROJECT} * \rho_{(\text{DNAME,DNUM,MGRSSN,MGRSTARTDATE})}(\text{DEPARTMENT})$$

The same query can be done in two steps by creating an intermediate table DEPT as follows:

$$\text{DEPT} \leftarrow \rho_{(\text{DNAME,DNUM,MGRSSN,MGRSTARTDATE})}(\text{DEPARTMENT})$$
$$\text{PROJ\_DEPT} \leftarrow \text{PROJECT} * \text{DEPT}$$

The attribute DNUM is called the **join attribute.** The resulting relation is illustrated in Figure 6.7a. In the PROJ_DEPT relation, each tuple combines a PROJECT tuple with the DEPARTMENT tuple for the department that controls the project, but *only one join attribute* is kept.

If the attributes on which the natural join is specified already *have the same names in both relations*, renaming is unnecessary. For example, to apply a natural join on the DNUMBER attributes of DEPARTMENT and DEPT_LOCATIONS, it is sufficient to write

$$\text{DEPT\_LOCS} \leftarrow \text{DEPARTMENT} * \text{DEPT\_LOCATIONS}$$

The resulting relation is shown in Figure 6.7b, which combines each department with its locations and has one tuple for each location. In general, NATURAL JOIN is performed by equating *all* attribute pairs that have the same name in the two relations. There can be a list of join attributes from each relation, and each corresponding pair must have the same name.

5. NATURAL JOIN is basically an EQUIJOIN followed by removal of the superfluous attributes.

(a)

PROJ_DEPT	PNAME	PNUMBER	PLOCATION	DNUM	DNAME	MGRSSN	MGRSTARTDATE
	ProductX	1	Bellaire	5	Research	333445555	1988-05-22
	ProductY	2	Sugarland	5	Research	333445555	1988-05-22
	ProductZ	3	Houston	5	Research	333445555	1988-05-22
	Computerization	10	Stafford	4	Administration	987654321	1995-01-01
	Reorganization	20	Houston	1	Headquarters	888665555	1981-06-19
	Newbenefits	30	Stafford	4	Administration	987654321	1995-01-01

(b)

DEPT_LOCS	DNAME	DNUMBER	MGRSSN	MGRSTARTDATE	LOCATION
	Headquarters	1	888665555	1981-06-19	Houston
	Administration	4	987654321	1995-01-01	Stafford
	Research	5	333445555	1988-05-22	Bellaire
	Research	5	333445555	1988-05-22	Sugarland
	Research	5	333445555	1988-05-22	Houston

FIGURE 6.7 Results of two NATURAL JOIN operations. (a) PROJ_DEPT ← PROJECT * DEPT. (b) DEPT_LOCS ← DEPARTMENT * DEPT_LOCATIONS.

A more general *but nonstandard* definition for NATURAL JOIN is

$$Q \leftarrow R *_{(<\text{list1}>),(<\text{list2}>)} S$$

In this case, <list1> specifies a list of i attributes from R, and <list2> specifies a list of i attributes from S. The lists are used to form equality comparison conditions between pairs of corresponding attributes, and the conditions are then ANDed together. Only the list corresponding to attributes of the first relation R—<list1>—is kept in the result Q.

Notice that if no combination of tuples satisfies the join condition, the result of a JOIN is an empty relation with zero tuples. In general, if R has n_R tuples and S has n_S tuples, the result of a JOIN operation $R \bowtie_{<\text{join condition}>} S$ will have between zero and $n_R * n_S$ tuples. The expected size of the join result divided by the maximum size $n_R * n_S$ leads to a ratio called **join selectivity,** which is a property of each join condition. If there is no join condition, all combinations of tuples qualify and the JOIN degenerates into a CARTESIAN PRODUCT, also called CROSS PRODUCT or CROSS JOIN.

As we can see, the JOIN operation is used to combine data from multiple relations so that related information can be presented in a single table. These operations are also known as **inner joins,** to distinguish them from a different variation of join called *outer joins* (see Section 6.4.3). Note that sometimes a join may be specified between a relation and itself, as we shall illustrate in Section 6.4.2. The NATURAL JOIN or EQUIJOIN operation can also be specified among multiple tables, leading to an *n-way join*. For example, consider the following three-way join:

$$((\text{PROJECT} \bowtie_{\text{DNUM=DNUMBER}} \text{DEPARTMENT}) \bowtie_{\text{MGRSSN=SSN}} \text{EMPLOYEE})$$

This links each project to its controlling department, and then relates the department to its manager employee. The net result is a consolidated relation in which each tuple contains this project-department-manager information.

6.3.3 A Complete Set of Relational Algebra Operations

It has been shown that the set of relational algebra operations $\{\sigma, \pi, \cup, -, x\}$ is a **complete** set; that is, any of the other original relational algebra operations can be expressed as a *sequence of operations from this set*. For example, the INTERSECTION operation can be expressed by using UNION and MINUS as follows:

$$R \cap S \equiv (R \cup S) - ((R - S) \cup (S - R))$$

Although, strictly speaking, INTERSECTION is not required, it is inconvenient to specify this complex expression every time we wish to specify an intersection. As another example, a JOIN operation can be specified as a CARTESIAN PRODUCT followed by a SELECT operation, as we discussed:

$$R \times_{<\text{condition}>} S \equiv \sigma_{<\text{condition}>} (R \times S)$$

Similarly, a NATURAL JOIN can be specified as a CARTESIAN PRODUCT preceded by RENAME and followed by SELECT and PROJECT operations. Hence, the various JOIN operations are also *not strictly necessary* for the expressive power of the relational algebra. However, they are important to consider as separate operations because they are convenient to use and are very commonly applied in database applications. Other operations have been included in the relational algebra for convenience rather than necessity. We discuss one of these—the DIVISION operation—in the next section.

6.3.4 The DIVISION Operation

The DIVISION operation, denoted by $\div$, is useful for a special kind of query that sometimes occurs in database applications. An example is "Retrieve the names of employees who work on *all* the projects that 'John Smith' works on." To express this query using the DIVISION operation, proceed as follows. First, retrieve the list of project numbers that 'John Smith' works on in the intermediate relation SMITH_PNOS:

$$\text{SMITH} \leftarrow \sigma_{\text{FNAME='JOHN' AND LNAME='SMITH'}}(\text{EMPLOYEE})$$

$$\text{SMITH\_PNOS} \leftarrow \pi_{\text{PNO}}(\text{WORKS\_ON} \bowtie_{\text{ESSN=SSN}} \text{SMITH})$$

Next, create a relation that includes a tuple <PNO, ESSN> whenever the employee whose social security number is ESSN works on the project whose number is PNO in the intermediate relation SSN_PNOS:

$$\text{SSN\_PNOS} \leftarrow \pi_{\text{ESSN,PNO}}(\text{WORKS\_ON})$$

Finally, apply the DIVISION operation to the two relations, which gives the desired employees' social security numbers:

$$\text{SSNS(SSN)} \leftarrow \text{SSN\_PNOS} \div \text{SMITH\_PNOS}$$

$$\text{RESULT} \leftarrow \pi_{\text{FNAME, LNAME}}(\text{SSNS} * \text{EMPLOYEE})$$

The previous operations are shown in Figure 6.8a.

(a)

SSN_PNOS	ESSN	PNO
	123456789	1
	123456789	2
	666884444	3
	453453453	1
	453453453	2
	333445555	2
	333445555	3
	333445555	10
	333445555	20
	999887777	30
	999887777	10
	987987987	10
	987987987	30
	987654321	30
	987654321	20
	888665555	20

SMITH_PNOS	PNO
	1
	2

SSNS	SSN
	123456789
	453453453

(b)

R	A	B
	a1	b1
	a2	b1
	a3	b1
	a4	b1
	a1	b2
	a3	b2
	a2	b3
	a3	b3
	a4	b3
	a1	b4
	a2	b4
	a3	b4

S	A
	a1
	a2
	a3

T	B
	b1
	b4

FIGURE 6.8 The DIVISION operation. (a) Dividing SSN_PNOS by SMITH_PNOS. (b) $T \leftarrow R \div S$.

In general, the DIVISION operation is applied to two relations $R(Z) \div S(X)$, where $X \subseteq Z$. Let $Y = Z - X$ (and hence $Z = X \cup Y$); that is, let Y be the set of attributes of R that are not attributes of S. The result of DIVISION is a relation $T(Y)$ that includes a tuple t if tuples t_R appear in R with $t_R[Y] = t$, and with $t_R[X] = t_S$ for every tuple t_S in S. This means that, for a tuple t to appear in the result T of the DIVISION, the values in t must appear in R in combination with every tuple in S. Note that in the formulation of the DIVISION operation, the tuples in the denominator relation restrict the numerator relation by selecting those tuples in the result that match all values present in the denominator. It is not necessary to know what those values are.

Figure 6.8b illustrates a DIVISION operation where $X = \{A\}$, $Y = \{B\}$, and $Z = \{A, B\}$. Notice that the tuples (values) b_1 and b_4 appear in R in combination with all three tuples in S; that is why they appear in the resulting relation T. All other values of B in R do not appear with all the tuples in S and are not selected: b_2 does not appear with a_2, and b_3 does not appear with a_1.

The DIVISION operation can be expressed as a sequence of π, $\times$, and $-$ operations as follows:

$$T1 \leftarrow \pi Y(R)$$

$$T2 \leftarrow \pi Y((S \times T_1) - R)$$

$$T \leftarrow T_1 - T_2$$

The DIVISION operation is defined for convenience for dealing with queries that involve "universal quantification" (see Section 6.6.6) or the *all* condition. Most RDBMS implementations with SQL as the primary query language do not directly implement division. SQL has a roundabout way of dealing with the type of query illustrated above (see Section 8.5.4). Table 6.1 lists the various basic relational algebra operations we have discussed.

6.4 Additional Relational Operations

Some common database requests—which are needed in commercial query languages for RDBMSs—cannot be performed with the original relational algebra operations described in Sections 6.1 through 6.3. In this section we define additional operations to express these requests. These operations enhance the expressive power of the original relational algebra.

6.4.1 Aggregate Functions and Grouping

The first type of request that cannot be expressed in the basic relational algebra is to specify mathematical **aggregate functions** on collections of values from the database. Examples of such functions include retrieving the average or total salary of all employees or the total number of employee tuples. These functions are used in simple statistical queries that summarize information from the database tuples. Common functions applied to collections of numeric values include SUM, AVERAGE, MAXIMUM, and MINIMUM. The COUNT function is used for counting tuples or values.

TABLE 6.1 OPERATIONS OF RELATIONAL ALGEBRA

Operation	Purpose	Notation
SELECT	Selects all tuples that satisfy the selection condition from a relation R.	$\sigma_{\text{<SELECTION CONDITION>}}(R)$
PROJECT	Produces a new relation with only some of the attributes of R, and removes duplicate tuples.	$\pi_{\text{<ATTRIBUTE LIST>}}(R)$
THETA JOIN	Produces all combinations of tuples from R_1 and R_2 that satisfy the join condition.	$R_1 \bowtie_{\text{<JOIN CONDITION>}} R_2$
EQUIJOIN	Produces all the combinations of tuples from R_1 and R_2 that satisfy a join condition with only equality comparisons.	$R_1 \bowtie_{\text{<JOIN CONDITION>}} R_2$, OR $R_1 \bowtie_{(\text{<JOIN ATTRIBUTES 1>}),\ (\text{<JOIN ATTRIBUTES 2>})} R_2$
NATURAL JOIN	Same as EQUIJOIN except that the join attributes of R_2 are not included in the resulting relation; if the join attributes have the same names, they do not have to be specified at all.	$R_1 *_{\text{<JOIN CONDITION>}} R_2$, OR $R_1 *_{(\text{<JOIN ATTRIBUTES 1>}),\ (\text{<JOIN ATTRIBUTES 2>})} R_2$ OR $R_1 * R_2$
UNION	Produces a relation that includes all the tuples in R_1 or R_2 or both R_1 and R_2; R_1 and R_2 must be union compatible.	$R_1 \cup R_2$
INTERSECTION	Produces a relation that includes all the tuples in both R_1 and R_2; R_1 and R_2 must be union compatible.	$R_1 \cap R_2$
DIFFERENCE	Produces a relation that includes all the tuples in R_1 that are not in R_2; R_1 and R_2 must be union compatible.	$R_1 - R_2$
CARTESIAN PRODUCT	Produces a relation that has the attributes of R_1 and R_2 and includes as tuples all possible combinations of tuples from R_1 and R_2.	$R_1 \times R_2$
DIVISION	Produces a relation $R(X)$ that includes all tuples $t[X]$ in $R_1(Z)$ that appear in R_1 in combination with every tuple from $R_2(Y)$, where $Z = X \cup Y$.	$R_1(Z) \div R_2(Y)$

Another common type of request involves grouping the tuples in a relation by the value of some of their attributes and then applying an aggregate function independently to each group. An example would be to group employee tuples by DNO, so that each group includes the tuples for employees working in the same department. We can then list each DNO value along with, say, the average salary of employees within the department, or the number of employees who work in the department.

We can define an AGGREGATE FUNCTION operation, using the symbol $\mathfrak{F}$ (pronounced "script F"),[6] to specify these types of requests as follows:

$$\text{<grouping attributes>}\ \mathfrak{F}\ \text{<function list>}\ (R)$$

6. There is no single agreed-upon notation for specifying aggregate functions. In some cases a "script A" is used.

where <grouping attributes> is a list of attributes of the relation specified in R, and <function list> is a list of (<function> <attribute>) pairs. In each such pair, <function> is one of the allowed functions—such as SUM, AVERAGE, MAXIMUM, MINIMUM, COUNT—and <attribute> is an attribute of the relation specified by R. The resulting relation has the grouping attributes plus one attribute for each element in the function list. For example, to retrieve each department number, the number of employees in the department, and their average salary, while renaming the resulting attributes as indicated below, we write:

$$\rho_{R(\text{DNO, NO\_OF\_EMPLOYEES, AVERAGE\_SAL})}(\text{DNO}\ \mathfrak{F}\ \text{COUNT}_{\text{SSN}}, \text{AVERAGE}_{\text{SALARY}}(\text{EMPLOYEE}))$$

The result of this operation is shown in Figure 6.9a.

In the above example, we specified a list of attribute names—between parentheses in the RENAME operation—for the resulting relation R. If no renaming is applied, then the attributes of the resulting relation that correspond to the function list will each be the concatenation of the function name with the attribute name in the form <function>_<attribute>.[7] For example, Figure 6.9b shows the result of the following operation:

$$\text{DNO}\ \mathfrak{F}\ \text{COUNT}_{\text{SSN}}, \text{AVERAGE}_{\text{SALARY}}(\text{EMPLOYEE})$$

If no grouping attributes are specified, the functions are applied to *all the tuples* in the relation, so the resulting relation has a *single tuple only*. For example, Figure 6.9c shows the result of the following operation:

$$\mathfrak{F}\ \text{COUNT}_{\text{SSN}}, \text{AVERAGE}_{\text{SALARY}}(\text{EMPLOYEE})$$

(a)

R	DNO	NO_OF_EMPLOYEES	AVERAGE_SAL
	5	4	33250
	4	3	31000
	1	1	55000

(b)

DNO	COUNT_SSN	AVERAGE_SALARY
5	4	33250
4	3	31000
1	1	55000

(c)

COUNT_SSN	AVERAGE_SALARY
8	35125

FIGURE 6.9 The AGGREGATE FUNCTION operation. (a) $\rho_{R(\text{DNO, NO\_OF\_EMPLOYEES, AVERAGE\_SAL})}(\text{DNO}\ \mathfrak{F}\ \text{COUNT}_{\text{SSN}}, \text{AVERAGE}_{\text{SALARY}}(\text{EMPLOYEE}))$. (b) $\text{DNO}\ \mathfrak{F}\ \text{COUNT}_{\text{SSN}}, \text{AVERAGE}_{\text{SALARY}}(\text{EMPLOYEE})$. (c) $\mathfrak{F}\ \text{COUNT}_{\text{SSN}}, \text{AVERAGE}_{\text{SALARY}}(\text{EMPLOYEE})$.

7. Note that this is an arbitrary notation we are suggesting. There is no standard notation.

It is important to note that, in general, duplicates are *not eliminated* when an aggregate function is applied; this way, the normal interpretation of functions such as SUM and AVERAGE is computed.[8] It is worth emphasizing that the result of applying an aggregate function is a relation, not a scalar number—even if it has a single value. This makes the relational algebra a closed system.

6.4.2 Recursive Closure Operations

Another type of operation that, in general, cannot be specified in the basic original relational algebra is **recursive closure.** This operation is applied to a **recursive relationship** between tuples of the same type, such as the relationship between an employee and a supervisor. This relationship is described by the foreign key SUPERSSN of the EMPLOYEE relation in Figures 5.5 and 5.6, and it relates each employee tuple (in the role of supervisee) to another employee tuple (in the role of supervisor). An example of a recursive operation is to retrieve all supervisees of an employee e at all levels—that is, all employees e' directly supervised by e, all employees e'' directly supervised by each employee e'; all employees e''' directly supervised by each employee e''; and so on.

Although it is straightforward in the relational algebra to specify all employees supervised by e *at a specific level,* it is difficult to specify all supervisees at *all* levels. For example, to specify the SSNs of all employees e' directly supervised—*at level one*—by the employee e whose name is 'James Borg' (see Figure 5.6), we can apply the following operation:

$$\text{BORG\_SSN} \leftarrow \pi_{\text{SSN}}(\sigma_{\text{FNAME='JAMES' AND LNAME='BORG'}}(\text{EMPLOYEE}))$$

$$\text{SUPERVISION(SSN1,SSN2)} \leftarrow \pi_{\text{SSN, SUPERSSN}}(\text{EMPLOYEE})$$

$$\text{RESULT1(SSN)} \leftarrow \pi_{\text{SSN1}}(\text{SUPERVISION} \bowtie_{\text{SSN2=SSN}} \text{BORG\_SSN})$$

To retrieve all employees supervised by Borg at level 2—that is, all employees e'' supervised by some employee e' who is directly supervised by Borg—we can apply another JOIN to the result of the first query, as follows:

$$\text{RESULT2(SSN)} \leftarrow \pi_{\text{SSN1}}(\text{SUPERVISION} \bowtie_{\text{SSN2=SSN}} \text{RESULT1})$$

To get both sets of employees supervised at levels 1 and 2 by 'James Borg,' we can apply the UNION operation to the two results, as follows:

$$\text{RESULT} \leftarrow \text{RESULT2} \cup \text{RESULT1}$$

The results of these queries are illustrated in Figure 6.10. Although it is possible to retrieve employees at each level and then take their UNION, we cannot, in general, specify a query such as "retrieve the supervisees of 'James Borg' at all levels" without utilizing a looping mechanism.[9] An operation called the *transitive closure* of relations has been proposed to compute the recursive relationship as far as the recursion proceeds.

8. In SQL, the option of eliminating duplicates before applying the aggregate function is available by including the keyword DISTINCT (see Section 8.4.4).
9. The SQL3 standard includes syntax for recursive closure.

(Borg's SSN is 888665555)

	(SSN)	(SUPERSSN)
SUPERVISION	SSN1	SSN2
	123456789	333445555
	333445555	888665555
	999887777	987654321
	987654321	888665555
	666884444	333445555
	453453453	333445555
	987987987	987654321
	888665555	null

RESULT 1	SSN
	333445555
	987654321

(Supervised by Borg)

RESULT 2	SSN
	123456789
	999887777
	666884444
	453453453
	987987987

(Supervised by Borg's subordinates)

RESULT	SSN
	123456789
	999887777
	666884444
	453453453
	987987987
	333445555
	987654321

(RESULT1 ∪ RESULT2)

FIGURE 6.10 A two-level recursive query.

6.4.3 OUTER JOIN Operations

We now discuss some extensions to the JOIN operation that are necessary to specify certain types of queries. The JOIN operations described earlier match tuples that satisfy the join condition. For example, for a NATURAL JOIN operation $R * S$, only tuples from R that have matching tuples in S—and vice versa—appear in the result. Hence, tuples without a *matching* (or *related*) tuple are eliminated from the JOIN result. Tuples with null values in the join attributes are also eliminated. This amounts to loss of information, if the result of JOIN is supposed to be used to generate a report based on all the information in the component relations.

A set of operations, called **outer joins,** can be used when we want to keep all the tuples in R, or all those in S, or all those in both relations in the result of the JOIN, regardless of whether or not they have matching tuples in the other relation. This satisfies the need of queries in which tuples from two tables are to be combined by matching corresponding rows, but without losing any tuples for lack of matching values. The join operations we described earlier in Section 6.3, where only matching tuples are kept in the result, are called **inner joins.**

For example, suppose that we want a list of all employee names and also the name of the departments they manage *if they happen to manage a department*; if they do not manage any, we can so indicate with a null value. We can apply an operation **LEFT OUTER JOIN**, denoted by ⟕, to retrieve the result as follows:

TEMP ← (EMPLOYEE ⟕ SSN=MGRSSN DEPARTMENT)

RESULT ← π FNAME, MINIT, LNAME, DNAME (TEMP)

The LEFT OUTER JOIN operation keeps every tuple in the *first*, or *left*, relation R in R ⟕ S; if no matching tuple is found in S, then the attributes of S in the join result are filled or "padded" with null values. The result of these operations is shown in Figure 6.11.

A similar operation, **RIGHT OUTER JOIN**, denoted by ⟖, keeps every tuple in the *second*, or right, relation S in the result of R ⟖ S. A third operation, **FULL OUTER JOIN**, denoted by ⟗ keeps all tuples in both the left and the right relations when no matching tuples are found, padding them with null values as needed. The three outer join operations are part of the SQL2 standard (see Chapter 8).

6.4.4 The OUTER UNION Operation

The **OUTER UNION** operation was developed to take the union of tuples from two relations if the relations are *not union compatible*. This operation will take the UNION of tuples in two relations $R(X, Y)$ and $S(X, Z)$ that are **partially compatible,** meaning that only some of their attributes, say X, are union compatible. The attributes that are union compatible are represented only once in the result, and those attributes that are not union compatible from either relation are also kept in the result relation $T(X, Y, Z)$.

Two tuples t_1 in R and t_2 in S are said to **match** if $t_1[X]=t_2[X]$, and are considered to represent the same entity or relationship instance. These will be combined (unioned) into a single tuple in T. Tuples in either relation that have no matching tuple in the other relation are padded with null values. For example, an OUTER UNION can be applied to two relations whose schemas are STUDENT(Name, SSN, Department, Advisor) and INSTRUCTOR(Name, SSN, Department, Rank). Tuples from the two relations are matched based on having the same

RESULT	FNAME	MINIT	LNAME	DNAME
	John	B	Smith	null
	Franklin	T	Wong	Research
	Alicia	J	Zelaya	null
	Jennifer	S	Wallace	Administration
	Ramesh	K	Narayan	null
	Joyce	A	English	null
	Ahmad	V	Jabbar	null
	James	E	Borg	Headquarters

FIGURE 6.11 The result of a LEFT OUTER JOIN operation.

combination of values of the shared attributes—Name, SSN, Department. The result relation, STUDENT_OR_INSTRUCTOR, will have the following attributes:

STUDENT_OR_INSTRUCTOR(Name, SSN, Department, Advisor, Rank)

All the tuples from both relations are included in the result, but tuples with the same (Name, SSN, Department) combination will appear only once in the result. Tuples appearing only in STUDENT will have a null for the Rank attribute, whereas tuples appearing only in INSTRUCTOR will have a null for the Advisor attribute. A tuple that exists in both relations, such as a student who is also an instructor, will have values for all its attributes.[10]

Notice that the same person may still appear twice in the result. For example, we could have a graduate student in the Mathematics department who is an instructor in the Computer Science department. Although the two tuples representing that person in STUDENT and INSTRUCTOR will have the same (Name, SSN) values, they will not agree on the Department value, and so will not be matched. This is because Department has two separate meanings in STUDENT (the department where the person studies) and INSTRUCTOR (the department where the person is employed as an instructor). If we wanted to union persons based on the same (Name, SSN) combination only, we should rename the Department attribute in each table to reflect that they have different meanings, and designate them as not being part of the union-compatible attributes.

Another capability that exists in most commercial languages (but not in the basic relational algebra) is that of specifying operations on values after they are extracted from the database. For example, arithmetic operations such as +, −, and * can be applied to numeric values that appear in the result of a query.

6.5 Examples of Queries in Relational Algebra

We now give additional examples to illustrate the use of the relational algebra operations. All examples refer to the database of Figure 5.6. In general, the same query can be stated in numerous ways using the various operations. We will state each query in one way and leave it to the reader to come up with equivalent formulations.

QUERY 1

Retrieve the name and address of all employees who work for the 'Research' department.

$$\text{RESEARCH\_DEPT} \leftarrow \sigma_{\text{DNAME='RESEARCH'}}(\text{DEPARTMENT})$$

$$\text{RESEARCH\_EMPS} \leftarrow (\text{RESEARCH\_DEPT} \bowtie_{\text{DNUMBER=DNOEMPLOYEE}})$$

$$\text{RESULT} \leftarrow \pi_{\text{FNAME, LNAME, ADDRESS}}(\text{RESEARCH\_EMPS})$$

10. Notice that OUTER UNION is equivalent to a FULL OUTER JOIN if the join attributes are *all* the common attributes of the two relations.

This query could be specified in other ways; for example, the order of the JOIN and SELECT operations could be reversed, or the JOIN could be replaced by a NATURAL JOIN after renaming one of the join attributes.

QUERY 2

For every project located in 'Stafford', list the project number, the controlling department number, and the department manager's last name, address, and birth date.

$$\text{STAFFORD\_PROJS} \leftarrow \sigma_{\text{PLOCATION='STAFFORD'}}(\text{PROJECT})$$

$$\text{CONTR\_DEPT} \leftarrow (\text{STAFFORD\_PROJS} \bowtie_{\text{DNUM=DNUMBER}} \text{DEPARTMENT})$$

$$\text{PROJ\_DEPT\_MGR} \leftarrow (\text{CONTR\_DEPT} \bowtie_{\text{MGRSSN=SSN}} \text{EMPLOYEE})$$

$$\text{RESULT} \leftarrow \pi_{\text{PNUMBER, DNUM, LNAME, ADDRESS, BDATE}}(\text{PROJ\_DEPT\_MGR})$$

QUERY 3

Find the names of employees who work on *all* the projects controlled by department number 5.

$$\text{DEPT5\_PROJS}(\text{PNO}) \leftarrow \pi_{\text{PNUMBER}}(\sigma_{\text{DNUM=5}}(\text{PROJECT}))$$

$$\text{EMP\_PROJ}(\text{SSN, PNO}) \leftarrow \pi_{\text{ESSN, PNO}}(\text{WORKS\_ON})$$

$$\text{RESULT\_EMP\_SSNS} \leftarrow \text{EMP\_PROJ} \div \text{DEPT5\_PROJS}$$

$$\text{RESULT} \leftarrow \pi_{\text{LNAME, FNAME}}(\text{RESULT\_EMP\_SSNS} * \text{EMPLOYEE})$$

QUERY 4

Make a list of project numbers for projects that involve an employee whose last name is 'Smith', either as a worker or as a manager of the department that controls the project.

$$\text{SMITHS}(\text{ESSN}) \leftarrow \pi_{\text{SSN}}(\sigma_{\text{LNAME='SMITH'}}(\text{EMPLOYEE}))$$

$$\text{SMITH\_WORKER\_PROJ} \leftarrow \pi_{\text{PNO}}(\text{WORKS\_ON} * \text{SMITHS})$$

$$\text{MGRS} \leftarrow \pi_{\text{LNAME, DNUMBER}}(\text{EMPLOYEE} \overset{\text{SSN=MGRSSN}}{\times} \text{DEPARTMENT})$$

$$\text{SMITH\_MANAGED\_DEPTS}(\text{DNUM}) \leftarrow \pi_{\text{DNUMBER}}(\sigma_{\text{LNAME='SMITH'}}(\text{MGRS}))$$

$$\text{SMITH\_MGR\_PROJS}(\text{PNO}) \leftarrow \pi_{\text{PNUMBER}}(\text{SMITH\_MANAGED\_DEPTS} * \text{PROJECT})$$

$$\text{RESULT} \leftarrow (\text{SMITH\_WORKER\_PROJS} \cup \text{SMITH\_MGR\_PROJS})$$

QUERY 5

List the names of all employees with two or more dependents.

Strictly speaking, this query cannot be done in the *basic (original) relational algebra*. We have to use the AGGREGATE FUNCTION operation with the COUNT aggregate function. We assume that dependents of the *same* employee have *distinct* DEPENDENT_NAME values.

$$T1(\text{SSN, NO\_OF\_DEPTS}) \leftarrow {}_{\text{ESSN}}\mathfrak{F}_{\text{COUNT DEPENDENT\_NAME}}(\text{DEPENDENT})$$

$$T2 \leftarrow \sigma_{\text{NO\_OF\_DEPS} \geq 2}(T1)$$

$$\text{RESULT} \leftarrow \pi_{\text{LNAME, FNAME}}(T2 * \text{EMPLOYEE})$$

QUERY 6

Retrieve the names of employees who have no dependents.

This is an example of the type of query that uses the MINUS (SET DIFFERENCE) operation.

ALL_EMPS ← π_{SSN}(EMPLOYEE)

EMPS_WITH_DEPS(SSN) ← π_{ESSN}(DEPENDENT)

EMPS_WITHOUT_DEPS ← (ALL_EMPS − EMPS_WITH_DEPS)

RESULT ← $\pi_{LNAME, FNAME}$(EMPS_WITHOUT_DEPS * EMPLOYEE)

QUERY 7

List the names of managers who have at least one dependent.

MGRS(SSN) ← π_{MGRSSN}(DEPARTMENT)

EMPS_WITH_DEPS(SSN) ← π_{ESSN}(DEPENDENT)

MGRS_WITH_DEPS ← (MGRS ∩ EMPS_WITH_DEPS)

RESULT ← $\pi_{LNAME, FNAME}$(MGRS_WITH_DEPS * EMPLOYEE)

As we mentioned earlier, the same query can in general be specified in many different ways. For example, the operations can often be applied in various orders. In addition, some operations can be used to replace others; for example, the INTERSECTION operation in Query 7 can be replaced by a NATURAL JOIN. As an exercise, try to do each of the above example queries using different operations.[11] In Chapter 8 and in Sections 6.6 and 6.7, we show how these queries are written in other relational languages.

6.6 The Tuple Relational Calculus

In this and the next section, we introduce another formal query language for the relational model called **relational calculus.** In relational calculus, we write one **declarative** expression to specify a retrieval request, and hence there is no description of how to evaluate a query. A calculus expression specifies *what* is to be retrieved rather than *how* to retrieve it. Therefore, the relational calculus is considered to be a **nonprocedural** language. This differs from relational algebra, where we must write a *sequence of operations* to specify a retrieval request; hence, it can be considered as a **procedural** way of stating a query. It is possible to nest algebra operations to form a single expression; however, a certain order among the operations is always explicitly specified in a relational algebra expression. This order also influences the strategy for evaluating the query. A calculus expression may be written in different ways, but the way it is written has no bearing on how a query should be evaluated.

11. When queries are optimized (see Chapter 15), the system will choose a particular sequence of operations that corresponds to an execution strategy that can be executed efficiently.

It has been shown that any retrieval that can be specified in the basic relational algebra can also be specified in relational calculus, and vice versa; in other words, the **expressive power** of the two languages is *identical*. This led to the definition of the concept of a relationally complete language. A relational query language L is considered **relationally complete** if we can express in L any query that can be expressed in relational calculus. Relational completeness has become an important basis for comparing the expressive power of high-level query languages. However, as we saw in Section 6.4, certain frequently required queries in database applications cannot be expressed in basic relational algebra or calculus. Most relational query languages are relationally complete but have *more expressive power* than relational algebra or relational calculus because of additional operations such as aggregate functions, grouping, and ordering.

In this section and the next, all our examples again refer to the database shown in Figures 5.6 and 5.7. We will use the same queries that were used in Section 6.5. Sections 6.6.5 and 6.6.6 discuss dealing with universal quantifiers and may be skipped by students interested in a general introduction to tuple calculus.

6.6.1 Tuple Variables and Range Relations

The tuple relational calculus is based on specifying a number of **tuple variables.** Each tuple variable usually *ranges over* a particular database relation, meaning that the variable may take as its value any individual tuple from that relation. A simple tuple relational calculus query is of the form

$\{t \mid \text{COND}(t)\}$

where t is a tuple variable and COND(t) is a conditional expression involving t. The result of such a query is the set of all tuples t that satisfy COND(t). For example, to find all employees whose salary is above $50,000, we can write the following tuple calculus expression:

$\{t \mid \text{EMPLOYEE}(t)$ **and** $t.\text{SALARY}>50000\}$

The condition EMPLOYEE(t) specifies that the **range relation** of tuple variable t is EMPLOYEE. Each EMPLOYEE tuple t that satisfies the condition t.SALARY>50000 will be retrieved. Notice that t.SALARY references attribute SALARY of tuple variable t; this notation resembles how attribute names are qualified with relation names or aliases in SQL, as we shall see in Chapter 8. In the notation of Chapter 5, t.SALARY is the same as writing t[SALARY].

The above query retrieves all attribute values for each selected EMPLOYEE tuple t. To retrieve only *some* of the attributes—say, the first and last names—we write

$\{t.\text{FNAME}, t.\text{LANME} \mid \text{EMPLOYEE}(t)$ **AND** $t.\text{SALARY}>50000\}$

Informally, we need to specify the following information in a tuple calculus expression:

- For each tuple variable t, the **range relation** R of t. This value is specified by a condition of the form $R(t)$.

- A condition to select particular combinations of tuples. As tuple variables range over their respective range relations, the condition is evaluated for every possible combination of tuples to identify the **selected combinations** for which the condition evaluates to TRUE.
- A set of attributes to be retrieved, the **requested attributes.** The values of these attributes are retrieved for each selected combination of tuples.

Before we discuss the formal syntax of tuple relational calculus, consider another query.

QUERY 0

Retrieve the birth date and address of the employee (or employees) whose name is 'John B. Smith'.

Q0: {*t*.BDATE, *t*.ADDRESS | EMPLOYEE(*t*) **AND** *t*.FNAME='John' **AND** *t*.MINIT='B' **AND** *t*.LNAME='Smith'}

In tuple relational calculus, we first specify the requested attributes t.BDATE and t.ADDRESS for each selected tuple *t*. Then we specify the condition for selecting a tuple following the bar (|)—namely, that *t* be a tuple of the EMPLOYEE relation whose FNAME, MINIT, and LNAME attribute values are 'John', 'B', and 'Smith', respectively.

6.6.2 Expressions and Formulas in Tuple Relational Calculus

A general **expression** of the tuple relational calculus is of the form

$$\{t_1.A_j, t_2.A_k, \ldots, t_n.A_m \mid \text{COND}(t_1, t_2, \ldots, t_n, t_{n+1}, t_{n+2}, \ldots, t_{n+m})\}$$

where $t_1, t_2, \ldots, t_n, t_{n+1}, \ldots, t_{n+m}$ are tuple variables, each A_i is an attribute of the relation on which t_i ranges, and COND is a **condition** or **formula**[12] of the tuple relational calculus. A formula is made up of predicate calculus **atoms,** which can be one of the following:

1. An atom of the form $R(t_i)$, where R is a relation name and t_i is a tuple variable. This atom identifies the range of the tuple variable t_i as the relation whose name is R.
2. An atom of the form $t_i.A$ **op** $t_j.B$, where **op** is one of the comparison operators in the set $\{=, <, \leq, >, \geq, \neq\}$, t_i and t_j are tuple variables, A is an attribute of the relation on which t_i ranges, and B is an attribute of the relation on which t_j ranges.
3. An atom of the form $t_i.A$ **op** c or c **op** $t_j.B$, where **op** is one of the comparison operators in the set $\{=, <, \leq, >, \geq, \neq\}$, t_i and t_j are tuple variables, A is an attribute of the relation on which t_i ranges, B is an attribute of the relation on which t_j ranges, and c is a constant value.

12. Also called a **well-formed formula,** or **wff,** in mathematical logic.

Each of the preceding atoms evaluates to either TRUE or FALSE for a specific combination of tuples; this is called the **truth value** of an atom. In general, a tuple variable t ranges over all possible tuples "in the universe." For atoms of the form $R(t)$, if t is assigned to a tuple that is a *member of the specified relation* R, the atom is TRUE; otherwise, it is FALSE. In atoms of types 2 and 3, if the tuple variables are assigned to tuples such that the values of the specified attributes of the tuples satisfy the condition, then the atom is TRUE.

A **formula** (condition) is made up of one or more atoms connected via the logical operators **AND, OR,** and **NOT** and is defined recursively as follows:

1. Every atom is a formula.
2. If F_1 and F_2 are formulas, then so are (F_1 **AND** F_2), (F_1 **OR** F_2), **NOT**(F_1), and **NOT** (F_2). The truth values of these formulas are derived from their component formulas F_1 and F_2 as follows:
   a. (F_1 **AND** F_2) is TRUE if both F_1 and F_2 are TRUE; otherwise, it is FALSE.
   b. (F_1 **OR** F_2) is FALSE if both F_1 and F_2 are FALSE; otherwise, it is TRUE.
   c. **NOT**(F_1) is TRUE if F_1 is FALSE; it is FALSE if F_1 is TRUE.
   d. **NOT**(F_2) is TRUE if F_2 is FALSE; it is FALSE if F_2 is TRUE.

6.6.3 The Existential and Universal Quantifiers

In addition, two special symbols called **quantifiers** can appear in formulas; these are the **universal quantifier** ($\forall$) and the **existential quantifier** ($\exists$). Truth values for formulas with quantifiers are described in rules 3 and 4 below; first, however, we need to define the concepts of free and bound tuple variables in a formula. Informally, a tuple variable t is bound if it is quantified, meaning that it appears in an ($\exists$ t) or ($\forall$ t) clause; otherwise, it is free. Formally, we define a tuple variable in a formula as **free** or **bound** according to the following rules:

- An occurrence of a tuple variable in a formula F that *is an atom* is free in F.
- An occurrence of a tuple variable t is free or bound in a formula made up of logical connectives—(F_1 **ANd** F_2), (F_1 **OR** F_2), **NOT**(F_1), and **NOT**(F_2)—depending on whether it is free or bound in F_1 or F_2 (if it occurs in either). Notice that in a formula of the form F = (F_1 **AND** F_2) or F = (F_1 **OR** F_2), a tuple variable may be free in F_1 and bound in F_2, or vice versa; in this case, one occurrence of the tuple variable is bound and the other is free in F.
- All *free* occurrences of a tuple variable t in F are **bound** in a formula F' of the form F' = ($\exists$ t)(F) or F' = ($\forall$ t)(F). The tuple variable is bound to the quantifier specified in F'. For example, consider the following formulas:

  F_1 : D.DNAME='RESEARCH'

  F_2 : ($\exists$T)(D.DNUMBER=T.DNO)

  F_3 : ($\forall$D)(D.MGRSSN='333445555')

The tuple variable d is free in both F_1 and F_2, whereas it is bound to the ($\forall$) quantifier in F_3. Variable t is bound to the ($\exists$) quantifier in F_2.

We can now give rules 3 and 4 for the definition of a formula we started earlier:

3. If F is a formula, then so is ($\exists$ t)(F), where t is a tuple variable. The formula ($\exists$ t)(F) is TRUE if the formula F evaluates to TRUE for *some* (at least one) tuple assigned to free occurrences of t in F; otherwise, ($\exists$ t)(F) is FALSE.
4. If F is a formula, then so is ($\forall$ t)(F), where t is a tuple variable. The formula ($\forall$ t)(F) is TRUE if the formula F evaluates to TRUE for *every tuple* (in the universe) assigned to free occurrences of t in F; otherwise, ($\forall$ t)(F) is FALSE.

The ($\exists$) quantifier is called an existential quantifier because a formula ($\exists$ t)(F) is TRUE if "there exists" some tuple that makes F TRUE. For the universal quantifier, ($\forall$ t)(F) is TRUE if every possible tuple that can be assigned to free occurrences of t in F is substituted for t, and F is TRUE for *every such substitution*. It is called the universal or "for all" quantifier because every tuple in "the universe of " tuples must make F TRUE to make the quantified formula TRUE.

6.6.4 Example Queries Using the Existential Quantifier

We will use some of the same queries from Section 6.5 to give a flavor of how the same queries are specified in relational algebra and in relational calculus. Notice that some queries are easier to specify in the relational algebra than in the relational calculus, and vice versa.

QUERY 1

Retrieve the name and address of all employees who work for the 'Research' department.

Q1: {t.FNAME, t.LNAME, t.ADDRESS | EMPLOYEE(t) **AND** ($\exists d$)
(DEPARTMENT(d) **AND** d.DNAME='Research' **AND** d.DNUMBER=t.DNO) }

The *only free tuple variables* in a relational calculus expression should be those that appear to the left of the bar (|). In Q1, t is the only free variable; it is then *bound successively* to each tuple. If a tuple *satisfies the conditions* specified in Q1, the attributes FNAME, LNAME, and ADDRESS are retrieved for each such tuple. The conditions EMPLOYEE(t) and DEPARTMENT(d) specify the range relations for t and d. The condition d.DNAME = 'Research' is a **selection condition** and corresponds to a SELECT operation in the relational algebra, whereas the condition d.DNUMBER = t.DNO is a **join condition** and serves a similar purpose to the JOIN operation (see Section 6.3).

QUERY 2

For every project located in 'Stafford', list the project number, the controlling department number, and the department manager's last name, birth date, and address.

Q2: {p.PNUMBER, p.DNUM, m.LNAME, m.BDATE, m.ADDRESS | PROJECT(p) **AND**
EMPLOYEE(m) **AND** p.PLOCATION='Stafford' **AND**
(($\exists d$)(DEPARTMENT(d) **AND** p.DNUM=d.DNUMBER **AND** d.MGRSSN=m.SSN)) }

In Q2 there are two free tuple variables, p and m. Tuple variable d is bound to the existential quantifier. The query condition is evaluated for every combination of tuples assigned to p and m; and out of all possible combinations of tuples to which p and m are bound, only the combinations that satisfy the condition are selected.

Several tuple variables in a query can range over the same relation. For example, to specify the query Q8—for each employee, retrieve the employee's first and last name and the first and last name of his or her immediate supervisor—we specify two tuple variables e and s that both range over the EMPLOYEE relation:

Q8: {e.FNAME, e.LNAME, s.FNAME, s.LNAME | EMPLOYEE(e) AND EMPLOYEE(s) AND e.SUPERSSN=s.SSN}

QUERY 3′

Find the name of each employee who works on *some* project controlled by department number 5. This is a variation of query 3 in which "all" is changed to "some." In this case we need two join conditions and two existential quantifiers.

Q3′: {e.LNAME, e.FNAME | EMPLOYEE(e) AND (($\exists$ x)($\exists$ w) (PROJECT(x) AND WORKS_ON(w) AND x.DNUM=5 AND w.ESSN=e.SSN AND x.PNUMBER=w.PNO)) }

QUERY 4

Make a list of project numbers for projects that involve an employee whose last name is 'Smith', either as a worker or as manager of the controlling department for the project.

Q4: {p.PNUMBER | PROJECT(p) AND ((($\exists$ e)($\exists$ w)(EMPLOYEE(e) AND WORKS_ON(w) AND w.PNO=p.PNUMBER AND e.LNAME='Smith' AND e.SSN=w.ESSN))

or

(($\exists$ m)($\exists$ d)(EMPLOYEE(m) AND DEPARTMENT(d) AND p.DNUM=d.DNUMBER AND d.MGRSSN=m.SSN AND m.LNAME='Smith'))) }

Compare this with the relational algebra version of this query in Section 6.5. The UNION operation in relational algebra can usually be substituted with an OR connective in relational calculus. In the next section we discuss the relationship between the universal and existential quantifiers and show how one can be transformed into the other.

6.6.5 Transforming the Universal and Existential Quantifiers

We now introduce some well-known transformations from mathematical logic that relate the universal and existential quantifiers. It is possible to transform a universal quantifier into an existential quantifier, and vice versa, to get an equivalent expression. One general transformation can be described informally as follows: Transform one type of quantifier

into the other with negation (preceded by NOT); AND and OR replace one another; a negated formula becomes unnegated; and an unnegated formula becomes negated. Some special cases of this transformation can be stated as follows, where the ≡ symbol stands for **equivalent to:**

(∀ x) ($P(x)$) ≡ NOT (∃ x) (NOT ($P(x)$))
(∃ x) ($P(x)$) ≡ NOT (∀ x) (NOT ($P(x)$))
(∀ x) ($P(x)$ AND $Q(x)$) ≡ NOT (∃ x) (NOT ($P(x)$) OR NOT ($Q(x)$))
(∀ x) ($P(x)$ OR $Q(x)$) ≡ NOT (∃ x) (NOT ($P(x)$) AND NOT ($Q(x)$))
(∃ x) ($P(x)$) OR $Q(x)$) ≡ NOT (∀ x) (NOT ($P(x)$) AND NOT ($Q(x)$))
(∃ x) ($P(x)$ AND $Q(x)$) ≡ NOT (∀ x) (NOT ($P(x)$) OR NOT ($Q(x)$))

Notice also that the following is TRUE, where the ⇒ symbol stands for **implies:**

(∀ x) ($P(x)$) ⇒ (∃ x) ($P(x)$)
NOT (∃ x) ($P(x)$) ⇒ NOT (∀ x) ($P(x)$)

6.6.6 Using the Universal Quantifier

Whenever we use a universal quantifier, it is quite judicious to follow a few rules to ensure that our expression makes sense. We discuss these rules with respect to Query 3.

QUERY 3

Find the names of employees who work on *all* the projects controlled by department number 5. One way of specifying this query is by using the universal quantifier as shown.

Q3: {e.LNAME, e.FNAME | EMPLOYEE(e) AND ((∀ x)(NOT(PROJECT(x)) OR NOT (x.DNUM=5)
OR ((∃ w)(WORKS_ON(w) AND w.ESSN=e.SSN AND x.PNUMBER=w.PNO)))) }

We can break up Q3 into its basic components as follows:

Q3: {e.LNAME, e.FNAME | EMPLOYEE(e) AND F' }
F' = ((∀ x)(NOT(PROJECT(x)) OR F_1))
F_1 = NOT(x.DNUM=5) OR F_2
F_2 = ((∃ w)(WORKS_ON(w) AND w.ESSN = e.SSN AND x.PNUMBER=w.PNO))

We want to make sure that a selected employee e works on *all the projects* controlled by department 5, but the definition of universal quantifier says that to make the quantified formula TRUE, the inner formula must be TRUE *for all tuples in the universe.* The trick is to exclude from the universal quantification all tuples that we are not interested in by making the condition TRUE *for all such tuples*. This is necessary because a universally quantified tuple variable, such as x in Q3, must evaluate to TRUE *for every possible tuple* assigned to it to make the quantified formula TRUE. The first tuples to

exclude (by making them evaluate automatically to TRUE) are those that are not in the relation R of interest. In Q3, using the expression **NOT**(PROJECT(x)) inside the universally quantified formula evaluates to TRUE all tuples x that are not in the PROJECT relation. Then we exclude the tuples we are not interested in from R itself. In Q3, using the expression **NOT**(x.DNUM=5) evaluates to TRUE all tuples x that are in the PROJECT relation but are not controlled by department 5. Finally, we specify a condition F_2 that must hold on all the remaining tuples in R. Hence, we can explain Q3 as follows:

1. For the formula $F' = (\forall x)(F)$ to be TRUE, we must have the formula F be TRUE *for all tuples in the universe that can be assigned to* x. However, in Q3 we are only interested in F being TRUE for all tuples of the PROJECT relation that are controlled by department 5. Hence, the formula F is of the form (**NOT**(PROJECT(x)) **OR** F_1). The '**NOT**(PROJECT(x)) **OR** . . .' condition is TRUE for all tuples *not in the* PROJECT *relation* and has the effect of eliminating these tuples from consideration in the truth value of F_1. For every tuple in the PROJECT relation, F_1 must be TRUE if F' is to be TRUE.
2. Using the same line of reasoning, we do not want to consider tuples in the PROJECT relation that are not controlled by department number 5, since we are only interested in PROJECT tuples whose DNUM = 5. We can therefore write:

   IF (x.DNUM=5) **THEN** F_2

   which is equivalent to

   (**NOT** (x.DNUM=5) **OR** F_2)
3. Formula F_1, hence, is of the form **NOT**(x.DNUM=5) **OR** F_2. In the context of Q3, this means that, for a tuple x in the PROJECT relation, either its DNUM≠5 or it must satisfy F_2.
4. Finally, F_2 gives the condition that we want to hold for a selected EMPLOYEE tuple: that the employee works on *every* PROJECT *tuple that has not been excluded yet*. Such employee tuples are selected by the query.

In English, Q3 gives the following condition for selecting an EMPLOYEE tuple e: For every tuple x in the PROJECT relation with x.DNUM = 5, there must exist a tuple w in WORKS_ON such that w.ESSN = e.SSN and w.PNO = x.PNUMBER. This is equivalent to saying that EMPLOYEE e works on every PROJECT x in DEPARTMENT number 5. (Whew!)

Using the general transformation from universal to existential quantifiers given in Section 6.6.5, we can rephrase the query in Q3 as shown in Q3A:

Q3A: {e.LNAME, e.FNAME | EMPLOYEE(e) **AND** (**NOT** (∃ x) (PROJECT(x) **AND** (x.DNUM=5) **AND**

(**NOT** (∃ w)(WORKS_ON(w) **AND** w.ESSN=e.SSN **AND** x.PNUMBER=w.PNO))))}

We now give some additional examples of queries that use quantifiers.

QUERY 6

Find the names of employees who have no dependents.

Q6: {*e*.FNAME, *e*.LNAME | EMPLOYEE(*e*) AND (NOT (∃*d*)(DEPENDENT(*d*) AND *e*.SSN=*d*.ESSN))}

Using the general transformation rule, we can rephrase Q6 as follows:

Q6A: {*e*.FNAME, *e*.LNAME | EMPLOYEE(*e*) AND ((∀*d*) (NOT(DEPENDENT(*d*)) OR NOT(*e*.SSN=*d*.ESSN)))}

QUERY 7

List the names of managers who have at least one dependent.

Q7: {*e*.FNAME, *e*.LNAME | EMPLOYEE(*e*) AND ((∃ *d*) (∃ *p*) (DEPARTMENT(*d*) AND DEPENDENT(*p*) AND *e*.SSN=*d*.MGRSSN AND *p*.ESSN=*e*.SSN))}

This query is handled by interpreting "managers who have at least one dependent" as "managers for whom there exists some dependent."

6.6.7 Safe Expressions

Whenever we use universal quantifiers, existential quantifiers, or negation of predicates in a calculus expression, we must make sure that the resulting expression makes sense. A **safe expression** in relational calculus is one that is guaranteed to yield a *finite number of tuples* as its result; otherwise, the expression is called **unsafe.** For example, the expression

{*t* | NOT (EMPLOYEE(*t*))}

is *unsafe* because it yields all tuples in the universe that are *not* EMPLOYEE tuples, which are infinitely numerous. If we follow the rules for Q3 discussed earlier, we will get a safe expression when using universal quantifiers. We can define safe expressions more precisely by introducing the concept of the *domain of a tuple relational calculus expression:* This is the set of all values that either appear as constant values in the expression or exist in any tuple in the relations referenced in the expression. The domain of {*t* | NOT(EMPLOYEE(*t*))} is the set of all attribute values appearing in some tuple of the EMPLOYEE relation (for any attribute). The domain of the expression Q3A would include all values appearing in EMPLOYEE, PROJECT, and WORKS_ON (unioned with the value 5 appearing in the query itself).

An expression is said to be **safe** if all values in its result are from the domain of the expression. Notice that the result of {*t* | NOT(EMPLOYEE(*t*))} is unsafe, since it will, in general, include tuples (and hence values) from outside the EMPLOYEE relation; such values are not in the domain of the expression. All of our other examples are safe expressions.

6.7 The Domain Relational Calculus

There is another type of relational calculus called the domain relational calculus, or simply, **domain calculus.** While SQL (see Chapter 8), a language based on tuple relational calculus, was being developed by IBM Research at San Jose, California, another language

called QBE (Query-By-Example) that is related to domain calculus was being developed almost concurrently at IBM Research at Yorktown Heights, New York. The formal specification of the domain calculus was proposed after the development of the QBE system.

Domain calculus differs from tuple calculus in the *type of variables* used in formulas: Rather than having variables range over tuples, the variables range over single values from domains of attributes. To form a relation of degree n for a query result, we must have n of these **domain variables**—one for each attribute. An expression of the domain calculus is of the form

$$\{x_1, x_2, \ldots, x_n \mid \text{COND}(x_1, x_2, \ldots, x_n, x_{n+1}, x_{n+2}, \ldots, x_{n+m})\}$$

where $x_1, x_2, \ldots, x_n, x_{n+1}, x_{n+2}, \ldots, x_{n+m}$ are domain variables that range over domains (of attributes), and COND is a **condition** or **formula** of the domain relational calculus.

A formula is made up of **atoms.** The atoms of a formula are slightly different from those for the tuple calculus and can be one of the following:

1. An atom of the form $R(x_1, x_2, \ldots, x_j)$, where R is the name of a relation of degree j and each x_i, $1 \leq i \leq j$, is a domain variable. This atom states that a list of values of $<x_1, x_2, \ldots, x_j>$ must be a tuple in the relation whose name is R, where x_i is the value of the ith attribute value of the tuple. To make a domain calculus expression more concise, we can *drop the commas* in a list of variables; thus, we can write

   $\{x_1, x_2, \ldots, x_n \mid R(x_1\ x_2\ x_3)$ **AND** $\ldots\}$

   instead of

   $\{x_1, x_2, \ldots, x_n \mid R(x_1, x_2, x_3)$ **AND** $\ldots\}$

2. An atom of the form x_i **op** x_j, where **op** is one of the comparison operators in the set $\{=, <, \leq, >, \geq, \neq\}$, and x_i and x_j are domain variables.
3. An atom of the form x_i **op** c or c **op** x_j, where **op** is one of the comparison operators in the set $\{=, <, \leq, >, \geq, \neq\}$, x_i and x_j are domain variables, and c is a constant value.

As in tuple calculus, atoms evaluate to either TRUE or FALSE for a specific set of values, called the **truth values** of the atoms. In case 1, if the domain variables are assigned values corresponding to a tuple of the specified relation R, then the atom is TRUE. In cases 2 and 3, if the domain variables are assigned values that satisfy the condition, then the atom is TRUE.

In a similar way to the tuple relational calculus, formulas are made up of atoms, variables, and quantifiers, so we will not repeat the specifications for formulas here. Some examples of queries specified in the domain calculus follow. We will use lowercase letters $l, m, n, \ldots, x, y, z$ for domain variables.

QUERY 0

Retrieve the birthdate and address of the employee whose name is 'John B. Smith'.

Q0: $\{uv \mid (\exists q)\ (\exists r)\ (\exists s)\ (\exists t)\ (\exists w)\ (\exists x)\ (\exists y)\ (\exists z)$
(EMPLOYEE$(qrstuvwxyz)$ **AND** q='JOHN' **AND** r='B' **AND** s='SMITH')$\}$

We need ten variables for the EMPLOYEE relation, one to range over the domain of each attribute in order. Of the ten variables $q, r, s, \ldots, z$, only u and v are free. We first specify the *requested attributes*, BDATE and ADDRESS, by the free domain variables u for BDATE and v for ADDRESS. Then we specify the condition for selecting a tuple following the bar (|)—namely, that the sequence of values assigned to the variables $qrstuvwxyz$ be a tuple of the EMPLOYEE relation and that the values for q (FNAME), r (MINIT), and s (LNAME) be 'John', 'B', and 'Smith', respectively. For convenience, we will quantify only those variables *actually appearing in a condition* (these would be q, r, and s in Q0) in the rest of our examples.[13]

An alternative shorthand notation, used in QBE, for writing this query is to assign the constants 'John', 'B', and 'Smith' directly as shown in Q0A. Here, all variables not appearing to the left of the bar are implicitly existentially quantified:[14]

Q0A: {uv | EMPLOYEE('John','B','Smith',t,u,v,w,x,y,z) }

QUERY 1

Retrieve the name and address of all employees who work for the 'Research' department.

Q1: {qsv | (∃ z) (∃ l) (∃ m) (EMPLOYEE($qrstuvwxyz$) AND DEPARTMENT($lmno$) AND l='RESEARCH' AND m=z)}

A condition relating two domain variables that range over attributes from two relations, such as $m = z$ in Q1, is a **join condition;** whereas a condition that relates a domain variable to a constant, such as l = 'Research', is a **selection condition.**

QUERY 2

For every project located in 'Stafford', list the project number, the controlling department number, and the department manager's last name, birth date, and address.

Q2: {$iksuv$ | (∃ j) (∃ m)(∃ n) (∃ t)(PROJECT($hijk$) AND EMPLOYEE($qrstuvwxyz$) AND DEPARTMENT($lmno$) AND k=m AND n=t AND j='STAFFORD')}

QUERY 6

Find the names of employees who have no dependents.

Q6: {qs | (∃ t) (EMPLOYEE($qrstuvwxyz$) AND (NOT(∃ l) (DEPENDENT($lmnop$) AND t=l)))}

Query 6 can be restated using universal quantifiers instead of the existential quantifiers, as shown in Q6A:

Q6A: {qs | (∃ t) (EMPLOYEE($qrstuvwxyz$) AND ((∀ l) (NOT(DEPENDENT($lmnop$)) OR NOT(t=l))))}

13. Note that the notation of quantifying only the domain variables actually used in conditions and of showing a predicate such as EMPLOYEE($qrstuvwxyz$) without separating domain variables with commas is an abbreviated notation used for convenience; it is not the correct formal notation.
14. Again, this is not formally accurate notation.

QUERY 7

List the names of managers who have at least one dependent.

Q7: {*sq* | (∃ *t*) (∃ *j*) (∃ *l*)(EMPLOYEE(*qrstuvwxyz*) AND DEPARTMENT(*hijk*) AND DEPENDENT(*lmnop*) AND *t*=*j* AND *l*=*t*)}

As we mentioned earlier, it can be shown that any query that can be expressed in the relational algebra can also be expressed in the domain or tuple relational calculus. Also, any *safe expression* in the domain or tuple relational calculus can be expressed in the relational algebra.

The Query-By-Example (QBE) language was based on the domain relational calculus, although this was realized later, after the domain calculus was formalized. QBE was one of the first graphical query languages with minimum syntax developed for database systems. It was developed at IBM Research and is available as an IBM commercial product as part of the QMF (Query Management Facility) interface option to DB2. It has been mimicked by several other commercial products. Because of its important place in the field of relational languages, we have included an overview of QBE in Appendix D.

6.8 SUMMARY

In this chapter we presented two formal languages for the relational model of data. They are used to manipulate relations and produce new relations as answers to queries. We discussed the relational algebra and its operations, which are used to specify a sequence of operations to specify a query. Then we introduced two types of relational calculi called tuple calculus and domain calculus; they are declarative in that they specify the result of a query without specifying how to produce the query result.

In Sections 6.1 through 6.3, we introduced the basic relational algebra operations and illustrated the types of queries for which each is used. The unary relational operators SELECT and PROJECT, as well as the RENAME operation, were discussed first. Then we discussed binary set theoretic operations requiring that relations on which they are applied be union compatible; these include UNION, INTERSECTION, and SET DIFFERENCE. The CARTESIAN PRODUCT operation is a set operation that can be used to combine tuples from two relations, producing all possible combinations. It is rarely used in practice; however, we showed how CARTESIAN PRODUCT followed by SELECT can be used to define matching tuples from two relations and leads to the JOIN operation. Different JOIN operations called THETA JOIN, EQUIJOIN, and NATURAL JOIN were introduced.

We then discussed some important types of queries that *cannot* be stated with the basic relational algebra operations but are important for practical situations. We introduced the AGGREGATE FUNCTION operation to deal with aggregate types of requests. We discussed recursive queries, for which there is no direct support in the algebra but which can be approached in a step-by-step approach, as we demonstrated. We then presented the OUTER JOIN and OUTER UNION operations, which extend JOIN and UNION and allow all information in source relations to be preserved in the result.

The last two sections described the basic concepts behind relational calculus, which is based on the branch of mathematical logic called predicate calculus. There are two types of relational calculi: (1) the tuple relational calculus, which uses tuple variables that range over tuples (rows) of relations, and (2) the domain relational calculus, which uses domain variables that range over domains (columns of relations). In relational calculus, a query is specified in a single declarative statement, without specifying any order or method for retrieving the query result. Hence, relational calculus is often considered to be a higher-level language than the relational algebra because a relational calculus expression states *what* we want to retrieve regardless of *how* the query may be executed.

We discussed the syntax of relational calculus queries using both tuple and domain variables. We also discussed the existential quantifier ($\exists$) and the universal quantifier ($\forall$). We saw that relational calculus variables are bound by these quantifiers. We described in detail how queries with universal quantification are written, and we discussed the problem of specifying safe queries whose results are finite. We also discussed rules for transforming universal into existential quantifiers, and vice versa. It is the quantifiers that give expressive power to the relational calculus, making it equivalent to relational algebra. There is no analog to grouping and aggregation functions in basic relational calculus, although some extensions have been suggested.

Review Questions

6.1. List the operations of relational algebra and the purpose of each.

6.2. What is union compatibility? Why do the UNION, INTERSECTION, and DIFFERENCE operations require that the relations on which they are applied be union compatible?

6.3. Discuss some types of queries for which renaming of attributes is necessary in order to specify the query unambiguously.

6.4. Discuss the various types of *inner join* operations. Why is theta join required?

6.5. What role does the concept of *foreign key* play when specifying the most common types of meaningful join operations?

6.6. What is the FUNCTION operation? What is it used for?

6.7. How are the OUTER JOIN operations different from the INNER JOIN operations? How is the OUTER UNION operation different from UNION?

6.8. In what sense does relational calculus differ from relational algebra, and in what sense are they similar?

6.9. How does tuple relational calculus differ from domain relational calculus?

6.10. Discuss the meanings of the existential quantifier ($\exists$) and the universal quantifier ($\forall$).

6.11. Define the following terms with respect to the tuple calculus: *tuple variable*, *range relation*, *atom*, *formula*, and *expression*.

6.12. Define the following terms with respect to the domain calculus: *domain variable*, *range relation*, *atom*, *formula*, and *expression*.

6.13. What is meant by a *safe expression* in relational calculus?

6.14. When is a query language called relationally complete?

Exercises

6.15. Show the result of each of the example queries in Section 6.5 as it would apply to the database state of Figure 5.6.

6.16. Specify the following queries on the database schema shown in Figure 5.5, using the relational operators discussed in this chapter. Also show the result of each query as it would apply to the database state of Figure 5.6.

a. Retrieve the names of all employees in department 5 who work more than 10 hours per week on the 'ProductX' project.
b. List the names of all employees who have a dependent with the same first name as themselves.
c. Find the names of all employees who are directly supervised by 'Franklin Wong'.
d. For each project, list the project name and the total hours per week (by all employees) spent on that project.
e. Retrieve the names of all employees who work on every project.
f. Retrieve the names of all employees who do not work on any project.
g. For each department, retrieve the department name and the average salary of all employees working in that department.
h. Retrieve the average salary of all female employees.
i. Find the names and addresses of all employees who work on at least one project located in Houston but whose department has no location in Houston.
j. List the last names of all department managers who have no dependents.

6.17. Consider the AIRLINE relational database schema shown in Figure 5.8, which was described in Exercise 5.11. Specify the following queries in relational algebra:

a. For each flight, list the flight number, the departure airport for the first leg of the flight, and the arrival airport for the last leg of the flight.
b. List the flight numbers and weekdays of all flights or flight legs that depart from Houston Intercontinental Airport (airport code 'IAH') and arrive in Los Angeles International Airport (airport code 'LAX').
c. List the flight number, departure airport code, scheduled departure time, arrival airport code, scheduled arrival time, and weekdays of all flights or flight legs that depart from some airport in the city of Houston and arrive at some airport in the city of Los Angeles.
d. List all fare information for flight number 'CO197'.
e. Retrieve the number of available seats for flight number 'CO197' on '1999-10-09'.

6.18. Consider the LIBRARY relational database schema shown in Figure 6.12, which is used to keep track of books, borrowers, and book loans. Referential integrity constraints are shown as directed arcs in Figure 6.12, as in the notation of Figure 5.7. Write down relational expressions for the following queries:

a. How many copies of the book titled *The Lost Tribe* are owned by the library branch whose name is 'Sharpstown'?
b. How many copies of the book titled *The Lost Tribe* are owned by each library branch?
c. Retrieve the names of all borrowers who do not have any books checked out.

d. For each book that is loaned out from the 'Sharpstown' branch and whose DueDate is today, retrieve the book title, the borrower's name, and the borrower's address.
e. For each library branch, retrieve the branch name and the total number of books loaned out from that branch.
f. Retrieve the names, addresses, and number of books checked out for all borrowers who have more than five books checked out.
g. For each book authored (or coauthored) by 'Stephen King,' retrieve the title and the number of copies owned by the library branch whose name is 'Central.'

6.19. Specify the following queries in relational algebra on the database schema given in Exercise 5.13:
a. List the Order# and Ship_date for all orders shipped from Warehouse number 'W2'.
b. List the Warehouse information from which the Customer named 'Jose Lopez' was supplied his orders. Produce a listing: Order#, Warehouse#.
c. Produce a listing CUSTNAME, #OFORDERS, AVG_ORDER_AMT, where the middle column is the total number of orders by the customer and the last column is the average order amount for that customer.
d. List the orders that were not shipped within 30 days of ordering.
e. List the Order# for orders that were shipped from *all* warehouses that the company has in New York.

6.20. Specify the following queries in relational algebra on the database schema given in Exercise 5.14:
a. Give the details (all attributes of TRIP relation) for trips that exceeded $2000 in expenses.

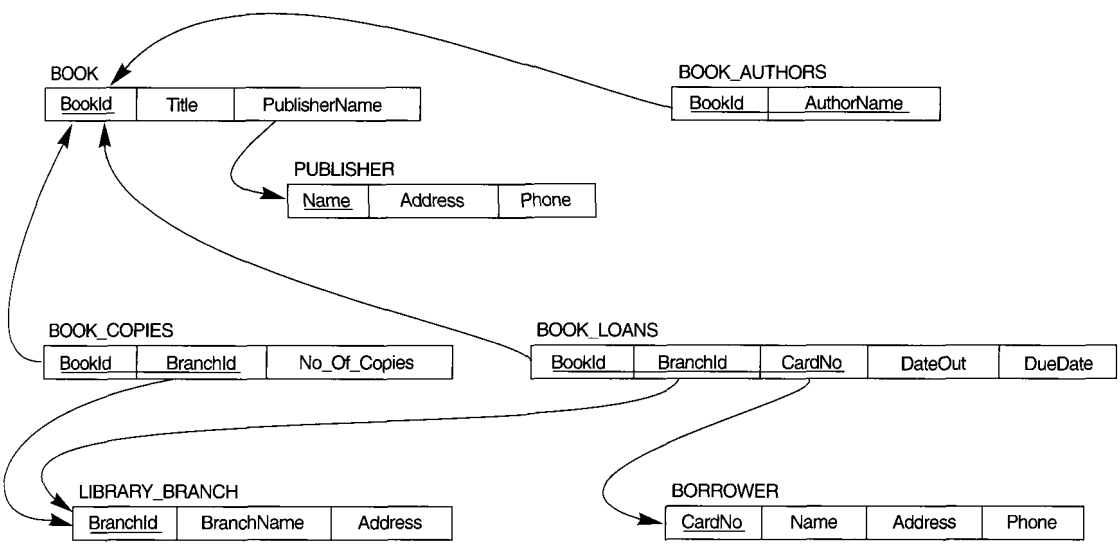


FIGURE 6.12 A relational database schema for a LIBRARY database.

b. Print the SSN of salesman who took trips to 'Honolulu'.
c. Print the total trip expenses incurred by the salesman with SSN = '234-56-7890'.

6.21. Specify the following queries in relational algebra on the database schema given in Exercise 5.15:
a. List the number of courses taken by all students named 'John Smith' in Winter 1999 (i.e., Quarter = 'W99').
b. Produce a list of textbooks (include Course#, Book_ISBN, Book_ Title) for courses offered by the 'CS' department that have used more than two books.
c. List any department that has *all* its adopted books published by 'AWL Publishing'.

6.22. Consider the two tables T1 and T2 shown in Figure 6.13. Show the results of the following operations:
a. $T1 \ltimes_{T1.P = T2.A} T2$
b. $T1 \ltimes_{T1.Q = T2.B} T2$
c. $T1 \rtimes_{T1.P = T2.A} T2$
d. $T1 \rtimes_{T1.Q = T2.B} T2$
e. $T1 \cup T2$
f. $T1 \ltimes_{(T1.P = T2.A \text{ AND } T1.R = T2.C)} T2$

6.23. Specify the following queries in relational algebra on the database schema of Exercise 5.16:
a. For the salesperson named 'Jane Doe', list the following information for all the cars she sold: Serial#, Manufacturer, Sale-price.
b. List the Serial# and Model of cars that have no options.
c. Consider the NATURAL JOIN operation between SALESPERSON and SALES. What is the meaning of a left OUTER JOIN for these tables (do not change the order of relations). Explain with an example.
d. Write a query in relational algebra involving selection and one set operation and say in words what the query does.

6.24. Specify queries a, b, c, e, f, i, and j of Exercise 6.16 in both tuple and domain relational calculus.

6.25. Specify queries a, b, c, and d of Exercise 6.17 in both tuple and domain relational calculus.

6.26. Specify queries c, d, f, and g of Exercise 6.18 in both tuple and domain relational calculus.

Table T1

P	Q	R
10	a	5
15	b	8
25	a	6

Table T2

A	B	C
10	b	6
25	c	3
10	b	5

FIGURE 6.13 A database state for the relations T1 and T2.

6.27. In a tuple relational calculus query with n tuple variables, what would be the typical minimum number of join conditions? Why? What is the effect of having a smaller number of join conditions?

6.28. Rewrite the domain relational calculus queries that followed Q0 in Section 6.7 in the style of the abbreviated notation of Q0A, where the objective is to minimize the number of domain variables by writing constants in place of variables wherever possible.

6.29. Consider this query: Retrieve the SSNs of employees who work on at least those projects on which the employee with SSN = 123456789 works. This may be stated as (FORALL x) (IF P THEN Q), where

- x is a tuple variable that ranges over the PROJECT relation.
- $P \equiv$ employee with SSN = 123456789 works on project x.
- $Q \equiv$ employee e works on project x.

Express the query in tuple relational calculus, using the rules

- $(\forall x)(P(x)) \equiv \text{NOT}(\exists x)(\text{NOT}(P(x)))$.
- (IF P THEN Q) $\equiv$ (NOT(P) OR Q).

6.30. Show how you may specify the following relational algebra operations in both tuple and domain relational calculus.

a. $\sigma_{A=C}(R(A, B, C))$
b. $\pi_{<A, B>}(R(A, B, C))$
c. $R(A, B, C) * S(C, D, E)$
d. $R(A, B, C) \cup S(A, B, C)$
e. $R(A, B, C) \cap S(A, B, C)$
f. $R(A, B, C) - S(A, B, C)$
g. $R(A, B, C) \times S(D, E, F)$
h. $R(A, B) \div S(A)$

6.31. Suggest extensions to the relational calculus so that it may express the following types of operations that were discussed in Section 6.4: (a) aggregate functions and grouping; (b) OUTER JOIN operations; (c) recursive closure queries.

Selected Bibliography

Codd (1970) defined the basic relational algebra. Date (1983a) discusses outer joins. Work on extending relational operations is discussed by Carlis (1986) and Ozsoyoglu et al. (1985). Cammarata et al. (1989) extends the relational model integrity constraints and joins.

Codd (1971) introduced the language Alpha, which is based on concepts of tuple relational calculus. Alpha also includes the notion of aggregate functions, which goes beyond relational calculus. The original formal definition of relational calculus was given by Codd (1972), which also provided an algorithm that transforms any tuple relational calculus expression to relational algebra. The QUEL (Stonebraker et al. 1976) is based on tuple relational calculus, with implicit existential quantifiers but no universal quantifiers, and was implemented in the Ingres system as a commercially available language. Codd defined relational completeness of a query language to mean at least as powerful as

relational calculus. Ullman (1988) describes a formal proof of the equivalence of relational algebra with the safe expressions of tuple and domain relational calculus. Abiteboul et al. (1995) and Atzeni and deAntonellis (1993) give a detailed treatment of formal relational languages.

Although ideas of domain relational calculus were initially proposed in the QBE language (Zloof 1975), the concept was formally defined by Lacroix and Pirotte (1977). The experimental version of the Query-By-Example system is described in Zloof (1977). The ILL (Lacroix and Pirotte 1977a) is based on domain relational calculus. Whang et al. (1990) extends QBE with universal quantifiers. Visual query languages, of which QBE is an example, are being proposed as a means of querying databases; conferences such as the Visual Database Systems Workshop (e.g., Arisawa and Catarci (2000) or Zhou and Pu (2002) have a number of proposals for such languages.

7

Relational Database Design by ER- and EER-to-Relational Mapping

We now focus on how to **design a relational database schema** based on a conceptual schema design. This corresponds to the logical database design or data model mapping step discussed in Section 3.1 (see Figure 3.1). We present the procedures to create a relational schema from an entity-relationship (ER) or an enhanced ER (EER) schema. Our discussion relates the constructs of the ER and EER models, presented in Chapters 3 and 4, to the constructs of the relational model, presented in Chapters 5 and 6. Many CASE (computer-aided software engineering) tools are based on the ER or EER models, or other similar models, as we have discussed in Chapters 3 and 4. These computerized tools are used interactively by database designers to develop an ER or EER schema for a database application. Many tools use ER or EER diagrams or variations to develop the schema graphically, and then automatically convert it into a relational database schema in the DDL of a specific relational DBMS by employing algorithms similar to the ones presented in this chapter.

We outline a seven-step algorithm in Section 7.1 to convert the basic ER model constructs—entity types (strong and weak), binary relationships (with various structural constraints), *n*-ary relationships, and attributes (simple, composite, and multivalued)—into relations. Then, in Section 7.2, we continue the mapping algorithm by describing how to map EER model constructs—specialization/generalization and union types (categories)—into relations.

7.1 Relational Database Design Using ER-to-Relational Mapping

7.1.1 ER-to-Relational Mapping Algorithm

We now describe the steps of an algorithm for ER-to-relational mapping. We will use the COMPANY database example to illustrate the mapping procedure. The COMPANY ER schema is shown again in Figure 7.1, and the corresponding COMPANY relational database schema is shown in Figure 7.2 to illustrate the mapping steps.

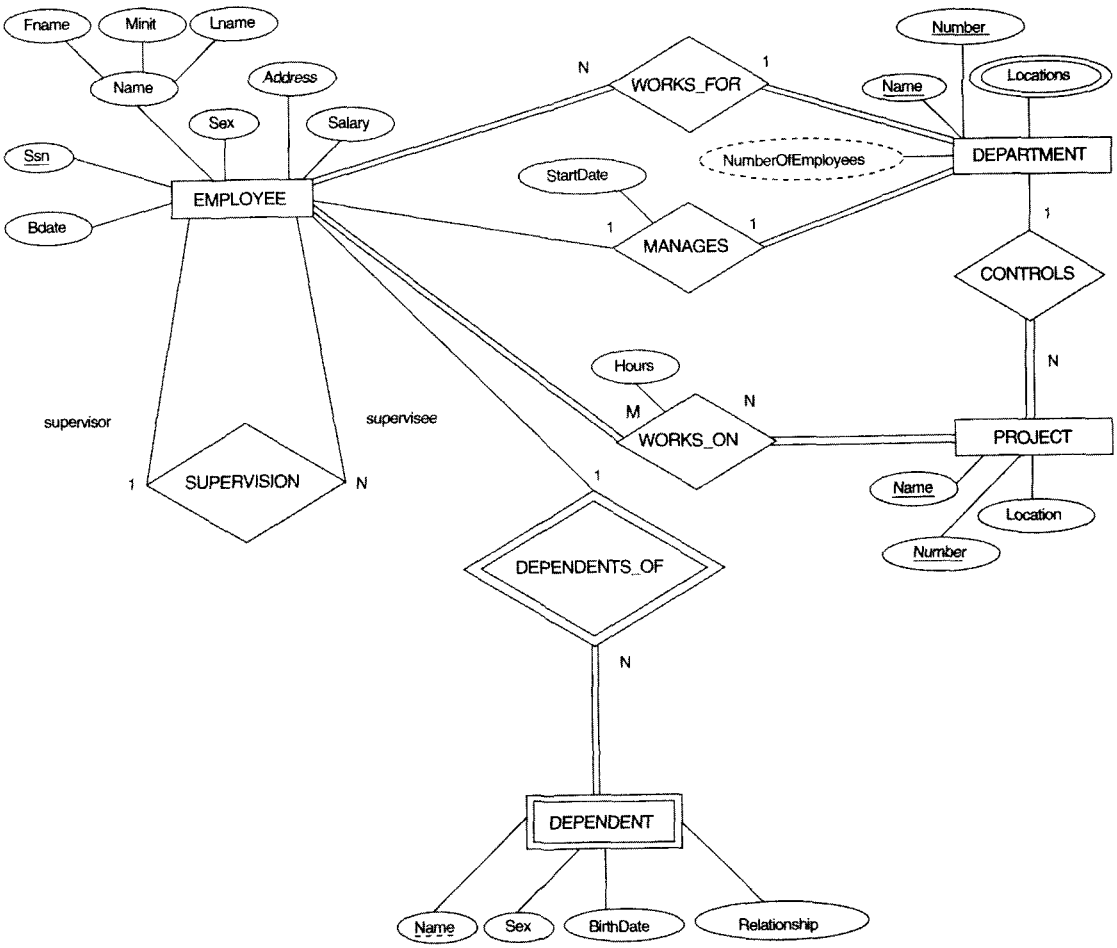


FIGURE 7.1 The ER conceptual schema diagram for the COMPANY database.

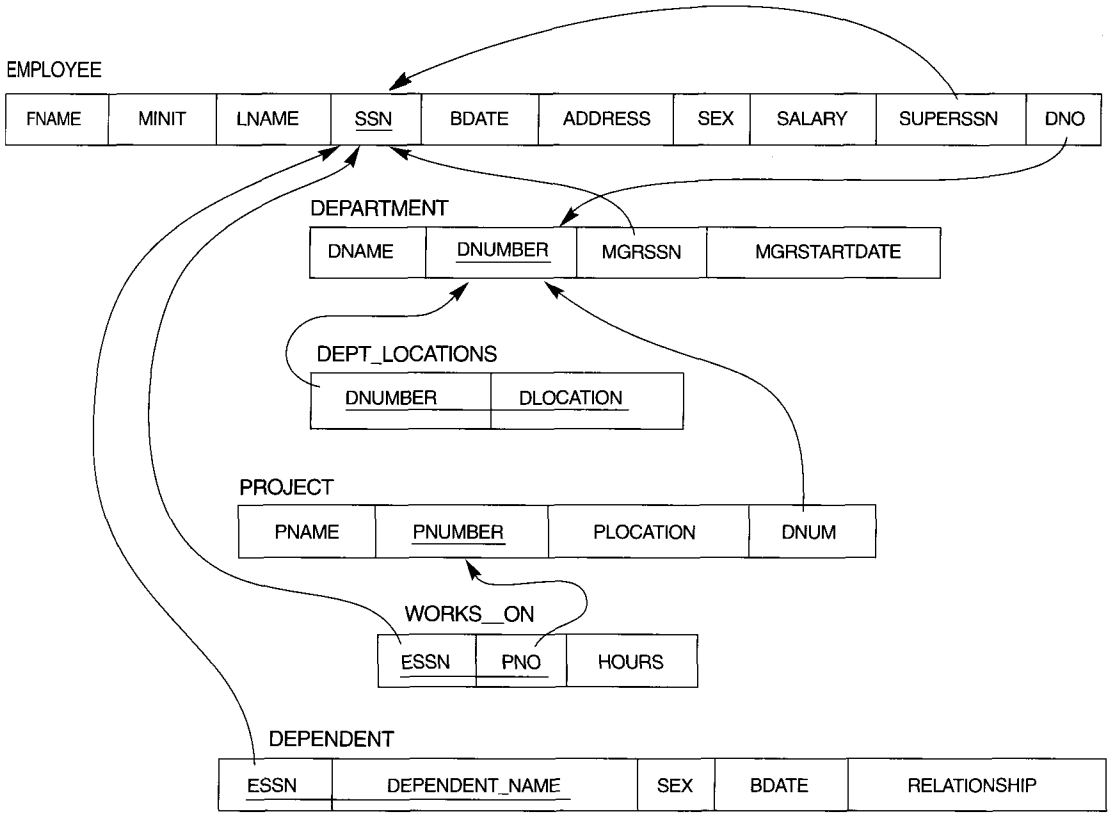


FIGURE 7.2 Result of mapping the COMPANY ER schema into a relational database schema.

Step 1: Mapping of Regular Entity Types. For each regular (strong) entity type E in the ER schema, create a relation R that includes all the simple attributes of E. Include only the simple component attributes of a composite attribute. Choose one of the key attributes of E as primary key for R. If the chosen key of E is composite, the set of simple attributes that form it will together form the primary key of R.

If multiple keys were identified for E during the conceptual design, the information describing the attributes that form each additional key is kept in order to specify secondary (unique) keys of relation R. Knowledge about keys is also kept for indexing purposes and other types of analyses.

In our example, we create the relations EMPLOYEE, DEPARTMENT, and PROJECT in Figure 7.2 to correspond to the regular entity types EMPLOYEE, DEPARTMENT, and PROJECT from Figure 7.1. The foreign key and relationship attributes, if any, are not included yet; they will be added during subsequent steps. These include the attributes SUPERSSN and DNO of EMPLOYEE, MGRSSN and MGRSTARTDATE of DEPARTMENT, and DNUM of PROJECT. In our example, we choose SSN, DNUMBER, and PNUMBER as primary keys for the relations EMPLOYEE, DEPARTMENT, and PROJECT,

respectively. Knowledge that DNAME of DEPARTMENT and PNAME of PROJECT are secondary keys is kept for possible use later in the design.

The relations that are created from the mapping of entity types are sometimes called **entity relations** because each tuple (row) represents an entity instance.

Step 2: Mapping of Weak Entity Types. For each weak entity type W in the ER schema with owner entity type E, create a relation R and include all simple attributes (or simple components of composite attributes) of W as attributes of R. In addition, include as foreign key attributes of R the primary key attribute(s) of the relation(s) that correspond to the owner entity type(s); this takes care of the identifying relationship type of W. The primary key of R is the combination of the primary key(s) of the owner(s) and the partial key of the weak entity type W, if any.

If there is a weak entity type E_2 whose owner is also a weak entity type E_1, then E_1 should be mapped before E_2 to determine its primary key first.

In our example, we create the relation DEPENDENT in this step to correspond to the weak entity type DEPENDENT. We include the primary key SSN of the EMPLOYEE relation—which corresponds to the owner entity type—as a foreign key attribute of DEPENDENT; we renamed it ESSN, although this is not necessary. The primary key of the DEPENDENT relation is the combination {ESSN, DEPENDENT_NAME} because DEPENDENT_NAME is the partial key of DEPENDENT.

It is common to choose the propagate (CASCADE) option for the referential triggered action (see Section 8.2) on the foreign key in the relation corresponding to the weak entity type, since a weak entity has an existence dependency on its owner entity. This can be used for both ON UPDATE and ON DELETE.

Step 3: Mapping of Binary 1:1 Relationship Types. For each binary 1:1 relationship type R in the ER schema, identify the relations S and T that correspond to the entity types participating in R. There are three possible approaches: (1) the foreign key approach, (2) the merged relationship approach, and (3) the cross-reference or relationship relation approach. Approach 1 is the most useful and should be followed unless special conditions exist, as we discuss below.

1. *Foreign key approach:* Choose one of the relations—S, say—and include as a foreign key in S the primary key of T. It is better to choose an entity type with *total participation* in R in the role of S. Include all the simple attributes (or simple components of composite attributes) of the 1:1 relationship type R as attributes of S.

   In our example, we map the 1:1 relationship type MANAGES from Figure 7.1 by choosing the participating entity type DEPARTMENT to serve in the role of S, because its participation in the MANAGES relationship type is total (every department has a manager). We include the primary key of the EMPLOYEE relation as foreign key in the DEPARTMENT relation and rename it MGRSSN. We also include the simple attribute STARTDATE of the MANAGES relationship type in the DEPARTMENT relation and rename it MGRSTARTDATE.

   Note that it is possible to include the primary key of S as a foreign key in T instead. In our example, this amounts to having a foreign key attribute, say DEPARTMENT_MANAGED in the EMPLOYEE relation, but it will have a null value for

employee tuples who do not manage a department. If only 10 percent of employees manage a department, then 90 percent of the foreign keys would be null in this case. Another possibility is to have foreign keys in both relations S and T redundantly, but this incurs a penalty for consistency maintenance.

2. *Merged relation option*: An alternative mapping of a 1:1 relationship type is possible by merging the two entity types and the relationship into a single relation. This may be appropriate when *both participations are total*.
3. *Cross-reference or relationship relation option*: The third alternative is to set up a third relation R for the purpose of cross-referencing the primary keys of the two relations S and T representing the entity types. As we shall see, this approach is required for binary M:N relationships. The relation R is called a **relationship relation**, (or sometimes a **lookup table**), because each tuple in R represents a relationship instance that relates one tuple from S with one tuple of T.

Step 4: Mapping of Binary 1:N Relationship Types. For each regular binary 1:N relationship type R, identify the relation S that represents the participating entity type at the *N-side* of the relationship type. Include as foreign key in S the primary key of the relation T that represents the other entity type participating in R; this is done because each entity instance on the N-side is related to at most one entity instance on the 1-side of the relationship type. Include any simple attributes (or simple components of composite attributes) of the 1:N relationship type as attributes of S.

In our example, we now map the 1:N relationship types WORKS_FOR, CONTROLS, and SUPERVISION from Figure 7.1. For WORKS_FOR we include the primary key DNUMBER of the DEPARTMENT relation as foreign key in the EMPLOYEE relation and call it DNO. For SUPERVISION we include the primary key of the EMPLOYEE relation as foreign key in the EMPLOYEE relation itself—because the relationship is recursive—and call it SUPERSSN. The CONTROLS relationship is mapped to the foreign key attribute DNUM of PROJECT, which references the primary key DNUMBER of the DEPARTMENT relation.

An alternative approach we can use here is again the relationship relation (cross-reference) option as in the case of binary 1:1 relationships. We create a separate relation R whose attributes are the keys of S and T, and whose primary key is the same as the key of S. This option can be used if few tuples in S participate in the relationship to avoid excessive null values in the foreign key.

Step 5: Mapping of Binary M:N Relationship Types. For each binary M:N relationship type R, create a new relation S to represent R. Include as foreign key attributes in S the primary keys of the relations that represent the participating entity types; their combination will form the primary key of S. Also include any simple attributes of the M:N relationship type (or simple components of composite attributes) as attributes of S. Notice that we cannot represent an M:N relationship type by a single foreign key attribute in one of the participating relations (as we did for 1:1 or 1:N relationship types) because of the M:N cardinality ratio; we must create a separate *relationship relation* S.

In our example, we map the M:N relationship type WORKS_ON from Figure 7.1 by creating the relation WORKS_ON in Figure 7.2. We include the primary keys of the PROJECT

and EMPLOYEE relations as foreign keys in WORKS_ON and rename them PNO and ESSN, respectively. We also include an attribute HOURS in WORKS_ON to represent the HOURS attribute of the relationship type. The primary key of the WORKS_ON relation is the combination of the foreign key attributes {ESSN, PNO}.

The propagate (CASCADE) option for the referential triggered action (see Section 8.2) should be specified on the foreign keys in the relation corresponding to the relationship R, since each relationship instance has an existence dependency on each of the entities it relates. This can be used for both ON UPDATE and ON DELETE.

Notice that we can always map 1:1 or 1:N relationships in a manner similar to M:N relationships by using the cross-reference (relationship relation) approach, as we discussed earlier. This alternative is particularly useful when few relationship instances exist, in order to avoid null values in foreign keys. In this case, the primary key of the relationship relation will be *only one* of the foreign keys that reference the participating entity relations. For a 1:N relationship, the primary key of the relationship relation will be the foreign key that references the entity relation on the N-side. For a 1:1 relationship, either foreign key can be used as the primary key of the relationship relation as long as no null entries are present in that relation.

Step 6: Mapping of Multivalued Attributes. For each multivalued attribute A, create a new relation R. This relation R will include an attribute corresponding to A, plus the primary key attribute K—as a foreign key in R—of the relation that represents the entity type or relationship type that has A as an attribute. The primary key of R is the combination of A and K. If the multivalued attribute is composite, we include its simple components.

In our example, we create a relation DEPT_LOCATIONS. The attribute DLOCATION represents the multivalued attribute LOCATIONS of DEPARTMENT, while DNUMBER—as foreign key—represents the primary key of the DEPARTMENT relation. The primary key of DEPT_LOCATIONS is the combination of {DNUMBER, DLOCATION}. A separate tuple will exist in DEPT_LOCATIONS for each location that a department has.

The propagate (CASCADE) option for the referential triggered action (see Section 8.2) should be specified on the foreign key in the relation R corresponding to the multivalued attribute for both ON UPDATE and ON DELETE. We should also note that the key of R when mapping a composite, multivalued attribute requires some analysis of the meaning of the component attributes. In some cases when a multivalued attribute is composite, only some of the component attributes are required to be part of the key of R; these attributes are similar to a partial key of a weak entity type that corresponds to the multivalued attribute (see Section 3.5).

Figure 7.2 shows the COMPANY relational database schema obtained through steps 1 to 6, and Figure 5.6 shows a sample database state. Notice that we did not yet discuss the mapping of n-ary relationship types ($n > 2$), because none exist in Figure 7.1; these are mapped in a similar way to M:N relationship types by including the following additional step in the mapping algorithm.

Step 7: Mapping of N-ary Relationship Types. For each n-ary relationship type R, where $n > 2$, create a new relation S to represent R. Include as foreign key

attributes in S the primary keys of the relations that represent the participating entity types. Also include any simple attributes of the n-ary relationship type (or simple components of composite attributes) as attributes of S. The primary key of S is usually a combination of all the foreign keys that reference the relations representing the participating entity types. However, if the cardinality constraints on any of the entity types E participating in R is 1, then the primary key of S should not include the foreign key attribute that references the relation E' corresponding to E (see Section 4.7).

For example, consider the relationship type SUPPLY of Figure 4.11a. This can be mapped to the relation SUPPLY shown in Figure 7.3, whose primary key is the combination of the three foreign keys {SNAME, PARTNO, PROJNAME}.

7.1.2 Discussion and Summary of Mapping for Model Constructs

Table 7.1 summarizes the correspondences between ER and relational model constructs and constraints.

One of the main points to note in a relational schema, in contrast to an ER schema, is that relationship types are not represented explicitly; instead, they are represented by having two attributes A and B, one a primary key and the other a foreign key (over the same domain) included in two relations S and T. Two tuples in S and T are related when they have the same value for A and B. By using the EQUIJOIN operation (or NATURAL JOIN if the two join attributes have the same name) over $S.A$ and $T.B$, we can combine all pairs of related tuples from S and T and materialize the relationship. When a binary 1:1 or

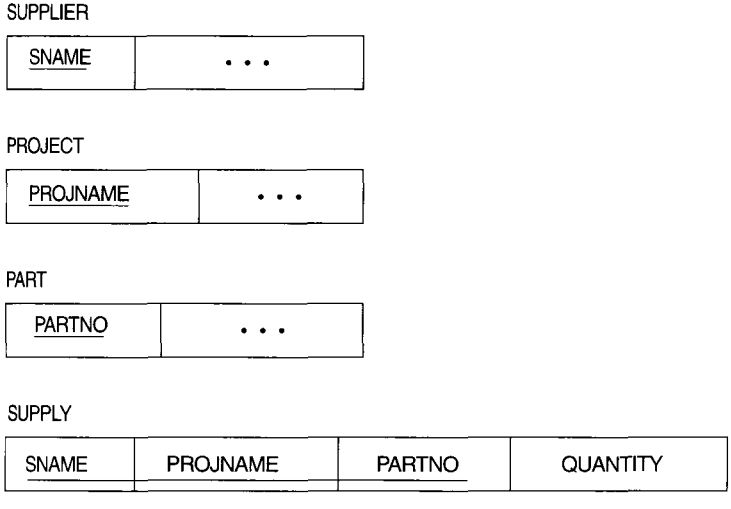


FIGURE 7.3 Mapping the n-ary relationship type SUPPLY from Figure 4.11a.

TABLE 7.1 CORRESPONDENCE BETWEEN ER AND RELATIONAL MODELS

ER MODEL	RELATIONAL MODEL
Entity type	"Entity" relation
1:1 or 1:N relationship type	Foreign key (or "relationship" relation)
M:N relationship type	"Relationship" relation and two foreign keys
n-ary relationship type	"Relationship" relation and n foreign keys
Simple attribute	Attribute
Composite attribute	Set of simple component attributes
Multivalued attribute	Relation and foreign key
Value set	Domain
Key attribute	Primary (or secondary) key

1:N relationship type is involved, a single join operation is usually needed. For a binary M:N relationship type, two join operations are needed, whereas for *n*-ary relationship types, *n* joins are needed to fully materialize the relationship instances.

For example, to form a relation that includes the employee name, project name, and hours that the employee works on each project, we need to connect each EMPLOYEE tuple to the related PROJECT tuples via the WORKS_ON relation of Figure 7.2. Hence, we must apply the EQUIJOIN operation to the EMPLOYEE and WORKS_ON relations with the join condition SSN = ESSN, and then apply another EQUIJOIN operation to the resulting relation and the PROJECT relation with join condition PNO = PNUMBER. In general, when multiple relationships need to be traversed, numerous join operations must be specified. A relational database user must always be aware of the foreign key attributes in order to use them correctly in combining related tuples from two or more relations. This is sometimes considered to be a drawback of the relational data model because the foreign key/primary key correspondences are not always obvious upon inspection of relational schemas. If an equijoin is performed among attributes of two relations that do not represent a foreign key/primary key relationship, the result can often be meaningless and may lead to spurious (invalid) data. For example, the reader can try joining the PROJECT and DEPT_LOCATIONS relations on the condition DLOCATION = PLOCATION and examine the result (see also Chapter 10).

Another point to note in the relational schema is that we create a separate relation for *each* multivalued attribute. For a particular entity with a set of values for the multivalued attribute, the key attribute value of the entity is repeated once for each value of the multivalued attribute in a separate tuple. This is because the basic relational model does *not* allow multiple values (a list, or a set of values) for an attribute in a single tuple. For example, because department 5 has three locations, three tuples exist in the DEPT_LOCATIONS relation of Figure 5.6; each tuple specifies one of the locations. In our example, we apply EQUIJOIN to DEPT_LOCATIONS and DEPARTMENT on the DNUMBER attribute to get the values of all locations along with other DEPARTMENT attributes. In the resulting relation, the values of the other department attributes are repeated in separate tuples for every location that a department has.

The basic relational algebra does not have a NEST or COMPRESS operation that would produce from the DEPT_LOCATIONS relation of Figure 5.6 a set of tuples of the form {<1, Houston>, <4, Stafford>, <5, {Bellaire, Sugarland, Houston}>}. This is a serious drawback of the basic normalized or "flat" version of the relational model. On this score, the object-oriented model and the legacy hierarchical and network models have better facilities than does the relational model. The nested relational model and object-relational systems (see Chapter 22) attempt to remedy this.

7.2 Mapping EER Model Constructs to Relations

We now discuss the mapping of EER model constructs to relations by extending the ER-to-relational mapping algorithm that was presented in Section 7.1.1.

7.2.1 Mapping of Specialization or Generalization

There are several options for mapping a number of subclasses that together form a specialization (or alternatively, that are generalized into a superclass), such as the {SECRETARY, TECHNICIAN, ENGINEER} subclasses of EMPLOYEE in Figure 4.4. We can add a further step to our ER-to-relational mapping algorithm from Section 7.1.1, which has seven steps, to handle the mapping of specialization. Step 8, which follows, gives the most common options; other mappings are also possible. We then discuss the conditions under which each option should be used. We use Attrs(R) to denote *the attributes of relation R*, and PK(R) to denote the *primary key of R*.

Step 8: Options for Mapping Specialization or Generalization. Convert each specialization with m subclasses $\{S_1, S_2, \ldots, S_m\}$ and (generalized) superclass C, where the attributes of C are $\{k, a_1, \ldots a_n\}$ and k is the (primary) key, into relation schemas using one of the four following options:

- ***Option 8A: Multiple relations—Superclass and subclasses.*** Create a relation L for C with attributes Attrs(L) = $\{k, a_1, \ldots, a_n\}$ and PK(L) = k. Create a relation L_i for each subclass S_i, $1 \leq i \leq m$, with the attributes Attrs(L_i) = $\{k\} \cup$ {attributes of S_i} and PK(L_i) = k. This option works for any specialization (total or partial, disjoint or overlapping).
- ***Option 8B: Multiple relations—Subclass relations only.*** Create a relation L_i for each subclass S_i, $1 \leq i \leq m$, with the attributes Attrs(L_i) = {attributes of S_i} $\cup$ $\{k, a_1, \ldots, a_n\}$ and PK(L_i) = k. This option only works for a specialization whose subclasses are *total* (every entity in the superclass must belong to (at least) one of the subclasses).
- ***Option 8C: Single relation with one type attribute.*** Create a single relation L with attributes Attrs(L) = $\{k, a_1, \ldots, a_n\} \cup$ {attributes of S_1} $\cup \ldots \cup$ {attributes of S_m} $\cup$ $\{t\}$ and PK(L) = k. The attribute t is called a **type** (or **discriminating**) attribute that

indicates the subclass to which each tuple belongs, if any. This option works only for a specialization whose subclasses are *disjoint*, and has the potential for generating many null values if many specific attributes exist in the subclasses.

- ***Option 8D: Single relation with multiple type attributes.*** Create a single relation schema L with attributes $\text{Attrs}(L) = \{k, a_1, \ldots, a_n\} \cup \{\text{attributes of } S_1\} \cup \ldots \cup \{\text{attributes of } S_m\} \cup \{t_1, t_2, \ldots, t_m\}$ and $\text{PK}(L) = k$. Each t_i, $1 \leq i \leq m$, is a **Boolean type attribute** indicating whether a tuple belongs to subclass S_i. This option works for a specialization whose subclasses are *overlapping* (but will also work for a disjoint specialization).

Options 8A and 8B can be called the **multiple-relation options,** whereas options 8C and 8D can be called the **single-relation options.** Option 8A creates a relation L for the superclass C and its attributes, plus a relation L_i for each subclass S_i; each L_i includes the specific (or local) attributes of S_i, plus the primary key of the superclass C, which is propagated to L_i and becomes its primary key. An EQUIJOIN operation on the primary key between any L_i and L produces all the specific and inherited attributes of the entities in S_i. This option is illustrated in Figure 7.4a for the EER schema in Figure 4.4. Option 8A

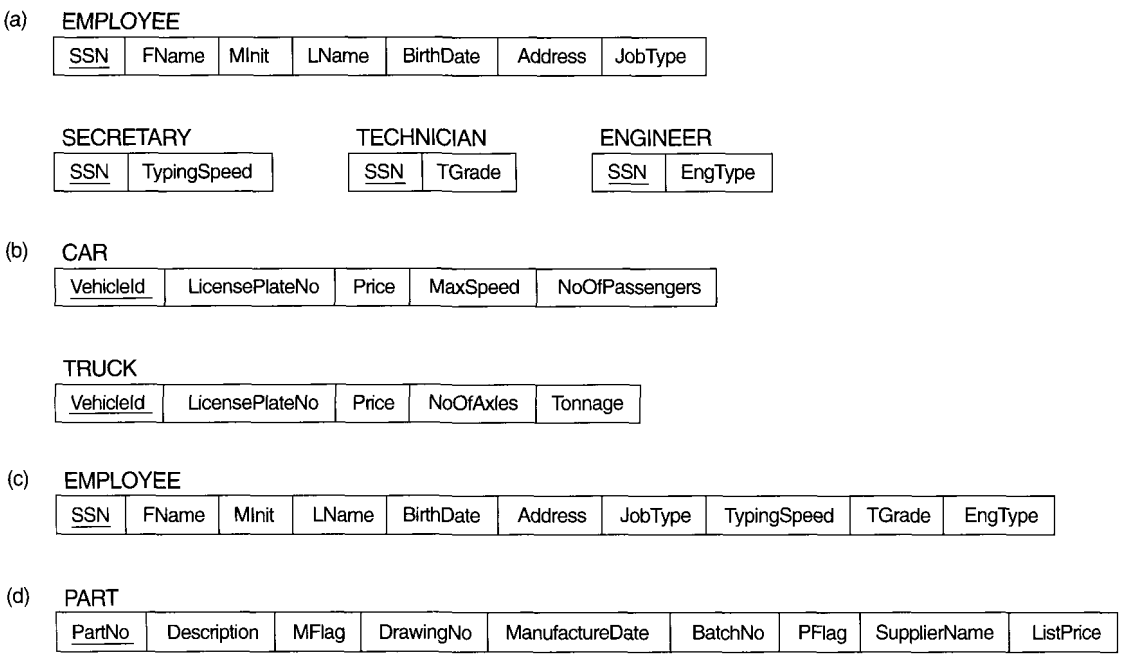


FIGURE 7.4 Options for mapping specialization or generalization. (a) Mapping the EER schema in Figure 4.4 using option 8A. (b) Mapping the EER schema in Figure 4.3b using option 8B. (c) Mapping the EER schema in Figure 4.4 using option 8C. (d) Mapping Figure 4.5 using option 8D with Boolean type fields MFlag and PFlag.

works for any constraints on the specialization: disjoint or overlapping, total or partial. Notice that the constraint

$$\pi_{<K>}(L_i) \subseteq \pi_{<K>}(L)$$

must hold for each L_i. This specifies a foreign key from each L_i to L, as well as an *inclusion dependency* $L_i.k < L.k$ (see Section 11.5).

In option 8B, the EQUIJOIN operation is *built into* the schema, and the relation L is done away with, as illustrated in Figure 7.4b for the EER specialization in Figure 4.3b. This option works well only when *both* the disjoint and total constraints hold. If the specialization is not total, an entity that does not belong to any of the subclasses S_i is lost. If the specialization is not disjoint, an entity belonging to more than one subclass will have its inherited attributes from the superclass C stored redundantly in more than one L_i. With option 8B, no relation holds all the entities in the superclass C; consequently, we must apply an OUTER UNION (or FULL OUTER JOIN) operation to the L_i relations to retrieve all the entities in C. The result of the outer union will be similar to the relations under options 8C and 8D except that the type fields will be missing. Whenever we search for an arbitrary entity in C, we must search all the m relations L_i.

Options 8C and 8D create a single relation to represent the superclass C and all its subclasses. An entity that does not belong to some of the subclasses will have null values for the specific attributes of these subclasses. These options are hence not recommended if many specific attributes are defined for the subclasses. If few specific subclass attributes exist, however, these mappings are preferable to options 8A and 8B because they do away with the need to specify EQUIJOIN and OUTER UNION operations and hence can yield a more efficient implementation.

Option 8C is used to handle disjoint subclasses by including a single **type** (or **image** or **discriminating**) **attribute** t to indicate the subclass to which each tuple belongs; hence, the domain of t could be $\{1, 2, \ldots, m\}$. If the specialization is partial, t can have null values in tuples that do not belong to any subclass. If the specialization is attribute-defined, that attribute serves the purpose of t and t is not needed; this option is illustrated in Figure 7.4c for the EER specialization in Figure 4.4.

Option 8D is designed to handle overlapping subclasses by including m *Boolean* type fields, one for *each* subclass. It can also be used for disjoint subclasses. Each type field t_i can have a domain {yes, no}, where a value of yes indicates that the tuple is a member of subclass S_i. If we use this option for the EER specialization in Figure 4.4, we would include three types attributes—IsASecretary, IsAEngineer, and IsATechnician—instead of the JobType attribute in Figure 7.4c. Notice that it is also possible to create a single type attribute of m *bits* instead of the m type fields.

When we have a multilevel specialization (or generalization) hierarchy or lattice, we do not have to follow the same mapping option for all the specializations. Instead, we can use one mapping option for part of the hierarchy or lattice and other options for other parts. Figure 7.5 shows one possible mapping into relations for the EER lattice of Figure 4.6. Here we used option 8A for PERSON/{EMPLOYEE, ALUMNUS, STUDENT}, option 8C for EMPLOYEE/{STAFF, FACULTY, STUDENT_ASSISTANT}, and option 8D for STUDENT_ASSISTANT/{RESEARCH_ASSISTANT, TEACHING_ASSISTANT}, STUDENT/STUDENT_ASSISTANT (in STUDENT), and STUDENT/{GRADUATE_STUDENT, UNDERGRADUATE_STUDENT}. In Figure 7.5, all attributes whose names end with 'Type' or 'Flag' are type fields.

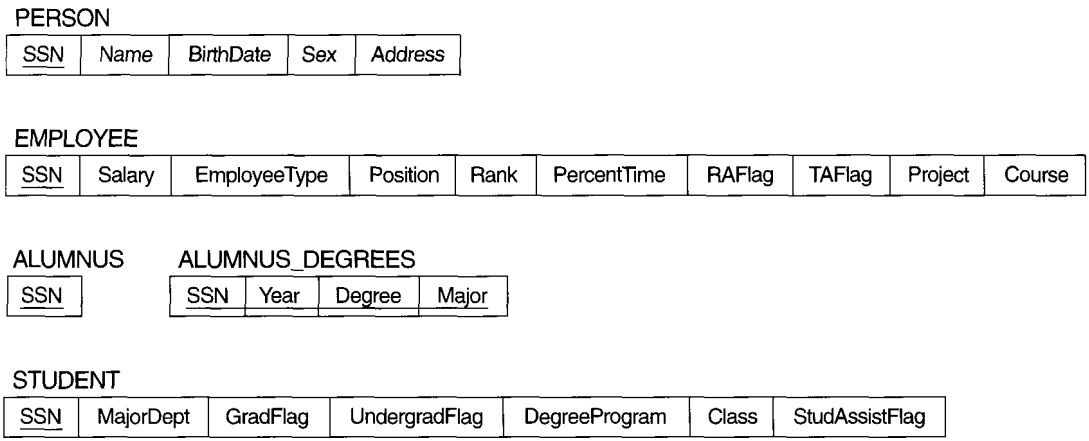


FIGURE 7.5 Mapping the EER specialization lattice in Figure 4.6 using multiple options.

7.2.2 Mapping of Shared Subclasses (Multiple Inheritance)

A shared subclass, such as ENGINEERING_MANAGER of Figure 4.6, is a subclass of several superclasses, indicating multiple inheritance. These classes must all have the same key attribute; otherwise, the shared subclass would be modeled as a category. We can apply any of the options discussed in step 8 to a shared subclass, subject to the restrictions discussed in step 8 of the mapping algorithm. In Figure 7.5, both options 8C and 8D are used for the shared subclass STUDENT_ASSISTANT. Option 8C is used in the EMPLOYEE relation (EmployeeType attribute) and option 8D is used in the STUDENT relation (StudAssistFlag attribute).

7.2.3 Mapping of Categories (Union Types)

We now add another step to the mapping procedure—step 9—to handle categories. A category (or union type) is a subclass of the *union* of two or more superclasses that can have different keys because they can be of different entity types. An example is the OWNER category shown in Figure 4.7, which is a subset of the union of three entity types PERSON, BANK, and COMPANY. The other category in that figure, REGISTERED_VEHICLE, has two superclasses that have the same key attribute.

Step 9: Mapping of Union Types (Categories). For mapping a category whose defining superclasses have different keys, it is customary to specify a new key attribute, called a **surrogate key,** when creating a relation to correspond to the category. This is because the keys of the defining classes are different, so we cannot use any one of them exclusively to identify all entities in the category. In our example of Figure 4.7, we can create a relation OWNER to correspond to the OWNER category, as illustrated in Figure 7.6, and include any attributes of the category in this relation. The primary key of the OWNER relation

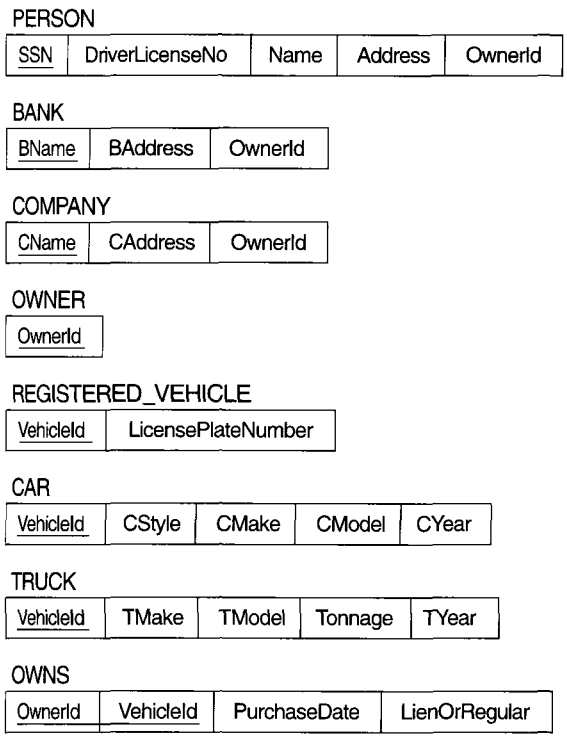


FIGURE 7.6 Mapping the EER categories (union types) in Figure 4.7 to relations.

is the surrogate key, which we called OwnerId. We also include the surrogate key attribute OwnerId as foreign key in each relation corresponding to a superclass of the category, to specify the correspondence in values between the surrogate key and the key of each superclass. Notice that if a particular PERSON (or BANK or COMPANY) entity is not a member of OWNER, it would have a null value for its OwnerId attribute in its corresponding tuple in the PERSON (or BANK or COMPANY) relation, and it would not have a tuple in the OWNER relation.

For a category whose superclasses have the same key, such as VEHICLE in Figure 4.7, there is no need for a surrogate key. The mapping of the REGISTERED_VEHICLE category, which illustrates this case, is also shown in Figure 7.6.

7.3 SUMMARY

In Section 7.1, we showed how a conceptual schema design in the ER model can be mapped to a relational database schema. An algorithm for ER-to-relational mapping was given and illustrated by examples from the COMPANY database. Table 7.1 summarized the correspondences between the ER and relational model constructs and constraints. We then added additional steps to the algorithm in Section 7.2 for mapping the constructs from the EER model into the

relational model. Similar algorithms are incorporated into graphical database design tools to automatically create a relational schema from a conceptual schema design.

Review Questions

7.1. Discuss the correspondences between the ER model constructs and the relational model constructs. Show how each ER model construct can be mapped to the relational model, and discuss any alternative mappings.

7.2. Discuss the options for mapping EER model constructs to relations.

Exercises

7.3. Try to map the relational schema of Figure 6.12 into an ER schema. This is part of a process known as *reverse engineering*, where a conceptual schema is created for an existing implemented database. State any assumptions you make.

7.4. Figure 7.7 shows an ER schema for a database that may be used to keep track of transport ships and their locations for maritime authorities. Map this schema into a relational schema, and specify all primary keys and foreign keys.

7.5. Map the BANK ER schema of Exercise 3.23 (shown in Figure 3.17) into a relational schema. Specify all primary keys and foreign keys. Repeat for the AIRLINE schema

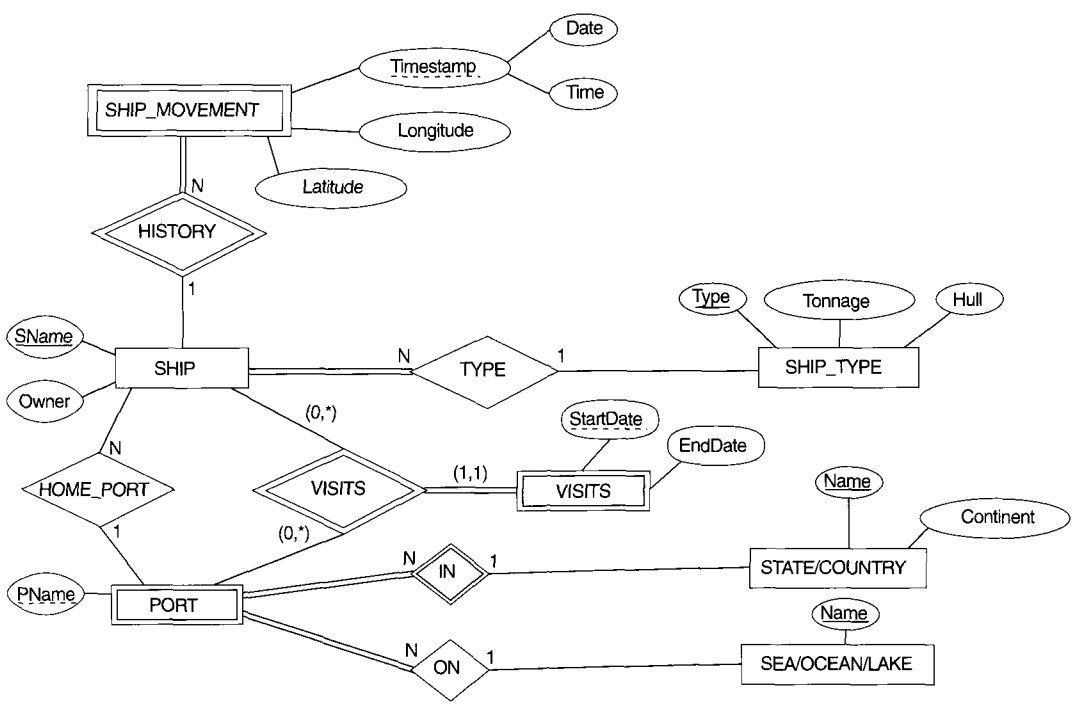


FIGURE 7.7 An ER schema for a SHIP_TRACKING database.

(Figure 3.16) of Exercise 3.19 and for the other schemas for Exercises 3.16 through 3.24.

7.6. Map the EER diagrams in Figures 4.10 and 4.17 into relational schemas. Justify your choice of mapping options.

Selected Bibliography

The original ER-to-relational mapping algorithm was described in Chen's classic paper (Chen 1976) that presented the original ER model.

8

SQL-99: Schema Definition, Basic Constraints, and Queries

The SQL language may be considered one of the major reasons for the success of relational databases in the commercial world. Because it became a standard for relational databases, users were less concerned about migrating their database applications from other types of database systems—for example, network or hierarchical systems—to relational systems. The reason is that even if users became dissatisfied with the particular relational DBMS product they chose to use, converting to another relational DBMS product would not be expected to be too expensive and time-consuming, since both systems would follow the same language standards. In practice, of course, there are many differences between various commercial relational DBMS packages. However, if the user is diligent in using only those features that are part of the standard, and if both relational systems faithfully support the standard, then conversion between the two systems should be much simplified. Another advantage of having such a standard is that users may write statements in a database application program that can access data stored in two or more relational DBMSs without having to change the database sublanguage (SQL) if both relational DBMSs support standard SQL.

This chapter presents the main features of the SQL standard for *commercial* relational DBMSs, whereas Chapter 5 presented the most important concepts underlying the *formal* relational data model. In Chapter 6 (Sections 6.1 through 6.5) we discussed the *relational algebra* operations, which are very important for understanding the types of requests that may be specified on a relational database. They are also important for query processing and optimization in a relational DBMS, as we shall see in Chapters 15 and 16. However, the

relational algebra operations are considered to be too technical for most commercial DBMS users because a query in relational algebra is written as a sequence of operations that, when executed, produces the required result. Hence, the user must specify how—that is, *in what order*—to execute the query operations. On the other hand, the SQL language provides a higher-level *declarative* language interface, so the user only specifies *what* the result is to be, leaving the actual optimization and decisions on how to execute the query to the DBMS. Although SQL includes some features from relational algebra, it is based to a greater extent on the *tuple relational calculus*, which we described in Section 6.6. However, the SQL syntax is more user-friendly than either of the two formal languages.

The name **SQL** is derived from Structured Query Language. Originally, SQL was called SEQUEL (for Structured English *QUE*ry *L*anguage) and was designed and implemented at IBM Research as the interface for an experimental relational database system called SYSTEM R. SQL is now the standard language for commercial relational DBMSs. A joint effort by ANSI (the American National Standards Institute) and ISO (the International Standards Organization) has led to a standard version of SQL (ANSI 1986), called SQL-86 or SQL1. A revised and much expanded standard called SQL2 (also referred to as SQL-92) was subsequently developed. The next version of the standard was originally called SQL3, but is now called SQL-99. We will try to cover the latest version of SQL as much as possible.

SQL is a comprehensive database language: It has statements for data definition, query, and update. Hence, it is both a DDL *and* a DML. In addition, it has facilities for defining views on the database, for specifying security and authorization, for defining integrity constraints, and for specifying transaction controls. It also has rules for embedding SQL statements into a general-purpose programming language such as Java or COBOL or C/C++.[1] We will discuss most of these topics in the following subsections.

Because the specification of the SQL standard is expanding, with more features in each version of the standard, the latest **SQL-99** standard is divided into a **core** specification plus optional specialized **packages.** The core is supposed to be implemented by all RDBMS vendors that are SQL-99 compliant. The packages can be implemented as optional modules to be purchased independently for specific database applications such as data mining, spatial data, temporal data, data warehousing, on-line analytical processing (OLAP), multimedia data, and so on. We give a summary of some of these packages—and where they are discussed in the book—at the end of this chapter.

Because SQL is very important (and quite large) we devote two chapters to its basic features. In this chapter, Section 8.1 describes the SQL DDL commands for creating schemas and tables, and gives an overview of the basic data types in SQL. Section 8.2 presents how basic constraints such as key and referential integrity are specified. Section 8.3 discusses statements for modifying schemas, tables, and constraints. Section 8.4 describes the basic SQL constructs for specifying retrieval queries, and Section 8.5 goes over more complex features of SQL queries, such as aggregate functions and grouping. Section 8.6 describes the SQL commands for insertion, deletion, and updating of data.

1. Originally, SQL had statements for creating and dropping indexes on the files that represent relations, but these have been dropped from the SQL standard for some time.

Section 8.7 lists some SQL features that are presented in other chapters of the book; these include transaction control in Chapter 17, security/authorization in Chapter 23, active databases (triggers) in Chapter 24, object-oriented features in Chapter 22, and OLAP (OnLine Analytical Processing) features in Chapter 28. Section 8.8 summarizes the chapter.

In the next chapter, we discuss the concept of views (virtual tables), and then describe how more general constraints may be specified as assertions or checks. This is followed by a description of the various database programming techniques for programming with SQL.

For the reader who desires a less comprehensive introduction to SQL, parts of Section 8.5 may be skipped.

8.1 SQL Data Definition and Data Types

SQL uses the terms **table, row,** and **column** for the formal relational model terms *relation*, *tuple*, and *attribute*, respectively. We will use the corresponding terms interchangeably. The main SQL command for data definition is the CREATE statement, which can be used to create schemas, tables (relations), and domains (as well as other constructs such as views, assertions, and triggers). Before we describe the relevant CREATE statements, we discuss schema and catalog concepts in Section 8.1.1 to place our discussion in perspective. Section 8.1.2 describes how tables are created, and Section 8.1.3 describes the most important data types available for attribute specification. Because the SQL specification is very large, we give a description of the most important features. Further details can be found in the various SQL standards documents (see bibliographic notes).

8.1.1 Schema and Catalog Concepts in SQL

Early versions of SQL did not include the concept of a relational database schema; all tables (relations) were considered part of the same schema. The concept of an SQL schema was incorporated starting with SQL2 in order to group together tables and other constructs that belong to the same database application. An **SQL schema** is identified by a **schema name,** and includes an **authorization identifier** to indicate the user or account who owns the schema, as well as **descriptors** for *each element* in the schema. Schema **elements** include tables, constraints, views, domains, and other constructs (such as authorization grants) that describe the schema. A schema is created via the CREATE SCHEMA statement, which can include all the schema elements' definitions. Alternatively, the schema can be assigned a name and authorization identifier, and the elements can be defined later. For example, the following statement creates a schema called COMPANY, owned by the user with authorization identifier JSMITH:

CREATE SCHEMA COMPANY **AUTHORIZATION** JSMITH;

In general, not all users are authorized to create schemas and schema elements. The privilege to create schemas, tables, and other constructs must be explicitly granted to the relevant user accounts by the system administrator or DBA.

In addition to the concept of a schema, SQL2 uses the concept of a **catalog**—a named collection of schemas in an SQL environment. An SQL **environment** is basically an installation of an SQL-compliant RDBMS on a computer system.[2] A catalog always contains a special schema called INFORMATION_SCHEMA, which provides information on all the schemas in the catalog and all the element descriptors in these schemas. Integrity constraints such as referential integrity can be defined between relations only if they exist in schemas within the same catalog. Schemas within the same catalog can also share certain elements, such as domain definitions.

8.1.2 The CREATE TABLE Command in SQL

The **CREATE TABLE** command is used to specify a new relation by giving it a name and specifying its attributes and initial constraints. The attributes are specified first, and each attribute is given a name, a data type to specify its domain of values, and any attribute constraints, such as NOT NULL. The key, entity integrity, and referential integrity constraints can be specified within the CREATE TABLE statement after the attributes are declared, or they can be added later using the ALTER TABLE command (see Section 8.3). Figure 8.1 shows sample data definition statements in SQL for the relational database schema shown in Figure 5.7.

Typically, the SQL schema in which the relations are declared is implicitly specified in the environment in which the CREATE TABLE statements are executed. Alternatively, we can explicitly attach the schema name to the relation name, separated by a period. For example, by writing

CREATE TABLE COMPANY.EMPLOYEE . . .

rather than

CREATE TABLE EMPLOYEE . . .

as in Figure 8.1, we can explicitly (rather than implicitly) make the EMPLOYEE table part of the COMPANY schema.

The relations declared through CREATE TABLE statements are called **base tables** (or base relations); this means that the relation and its tuples are actually created and stored as a file by the DBMS. Base relations are distinguished from **virtual relations,** created through the CREATE VIEW statement (see Section 9.2), which may or may not correspond to an actual physical file. In SQL the attributes in a base table are considered to be *ordered in the sequence in which they are specified* in the CREATE TABLE statement. However, rows (tuples) are not considered to be ordered within a relation.

2. SQL also includes the concept of a *cluster* of catalogs within an environment, but it is not very clear if so many levels of nesting are required in most applications.

(a)

```
CREATE TABLE EMPLOYEE
      ( FNAME            VARCHAR(15)        NOT NULL,
        MINIT            CHAR,
        LNAME            VARCHAR(15)        NOT NULL,
        SSN              CHAR(9)            NOT NULL,
        BDATE            DATE,
        ADDRESS          VARCHAR(30),
        SEX              CHAR,
        SALARY           DECIMAL(10,2),
        SUPERSSN         CHAR(9),
        DNO              INT                NOT NULL,
    PRIMARY KEY (SSN),
    FOREIGN KEY (SUPERSSN) REFERENCES EMPLOYEE(SSN),
    FOREIGN KEY (DNO) REFERENCES DEPARTMENT(DNUMBER) );

CREATE TABLE DEPARTMENT
      ( DNAME            VARCHAR(15)        NOT NULL,
        DNUMBER          INT                NOT NULL,
        MGRSSN           CHAR(9)            NOT NULL,
        MGRSTARTDATE     DATE,
    PRIMARY KEY (DNUMBER),
    UNIQUE (DNAME),
    FOREIGN KEY (MGRSSN) REFERENCES EMPLOYEE(SSN) );

CREATE TABLE DEPT_LOCATIONS
      ( DNUMBER          INT                NOT NULL,
        DLOCATION        VARCHAR(15)        NOT NULL,
    PRIMARY KEY (DNUMBER, DLOCATION),
    FOREIGN KEY (DNUMBER) REFERENCES DEPARTMENT(DNUMBER) );

CREATE TABLE PROJECT
      ( PNAME            VARCHAR(15)        NOT NULL,
        PNUMBER          INT                NOT NULL,
        PLOCATION        VARCHAR(15),
        DNUM             INT                NOT NULL,
    PRIMARY KEY (PNUMBER),
    UNIQUE (PNAME),
    FOREIGN KEY (DNUM) REFERENCES DEPARTMENT(DNUMBER) );

CREATE TABLE WORKS_ON
      ( ESSN             CHAR(9)            NOT NULL,
        PNO              INT                NOT NULL,
        HOURS            DECIMAL(3,1)       NOT NULL,
    PRIMARY KEY (ESSN, PNO),
    FOREIGN KEY (ESSN) REFERENCES EMPLOYEE(SSN),
    FOREIGN KEY (PNO) REFERENCES PROJECT(PNUMBER) );

CREATE TABLE DEPENDENT
      ( ESSN             CHAR(9)            NOT NULL,
        DEPENDENT_NAME   VARCHAR(15)        NOT NULL,
        SEX              CHAR,
        BDATE            DATE,
        RELATIONSHIP     VARCHAR(8),
    PRIMARY KEY (ESSN, DEPENDENT_NAME),
    FOREIGN KEY (ESSN) REFERENCES EMPLOYEE(SSN) );
```

FIGURE 8.1 SQL CREATE TABLE data definition statements for defining the COMPANY schema from Figure 5.7

8.1.3 Attribute Data Types and Domains in SQL

The basic **data types** available for attributes include numeric, character string, bit string, boolean, date, and time.

- **Numeric** data types include integer numbers of various sizes (INTEGER or INT, and SMALLINT) and floating-point (real) numbers of various precision (FLOAT or REAL, and DOUBLE PRECISION). Formatted numbers can be declared by using DECIMAL(i,j)—or DEC(i,j) or NUMERIC(i,j)—where i, the *precision*, is the total number of decimal digits and j, the *scale*, is the number of digits after the decimal point. The default for scale is zero, and the default for precision is implementation-defined.
- **Character-string** data types are either fixed length—CHAR(n) or CHARACTER(n), where n is the number of characters—or varying length—VARCHAR(n) or CHAR VARYING(n) or CHARACTER VARYING(n), where n is the maximum number of characters. When specifying a literal string value, it is placed between single quotation marks (apostrophes), and it is *case sensitive* (a distinction is made between uppercase and lowercase).[3] For fixed-length strings, a shorter string is padded with blank characters to the right. For example, if the value 'Smith' is for an attribute of type CHAR(10), it is padded with five blank characters to become 'Smith ' if needed. Padded blanks are generally ignored when strings are compared. For comparison purposes, strings are considered ordered in alphabetic (or lexicographic) order; if a string *str1* appears before another string *str2* in alphabetic order, then *str1* is considered to be less than *str2*.[4] There is also a concatenation operator denoted by || (double vertical bar) that can concatenate two strings in SQL. For example, 'abc' || 'XYZ' results in a single string 'abcXYZ'.
- **Bit-string** data types are either of fixed length n—BIT(n)—or varying length—BIT VARYING(n), where n is the maximum number of bits. The default for n, the length of a character string or bit string, is 1. Literal bit strings are placed between single quotes but preceded by a B to distinguish them from character strings; for example, B'10101'.[5]
- A **boolean** data type has the traditional values of TRUE or FALSE. In SQL, because of the presence of NULL values, a three-valued logic is used, so a third possible value for a boolean data type is UNKNOWN. We discuss the need for UNKNOWN and the three-valued logic in Section 8.5.1.
- New data types for **date** and **time** were added in SQL2. The DATE data type has ten positions, and its components are YEAR, MONTH, and DAY in the form YYYY-MM-DD. The TIME data type has at least eight positions, with the components HOUR, MINUTE, and SECOND in the form HH:MM:SS. Only valid dates and times should be allowed by

3. This is not the case with SQL keywords, such as CREATE or CHAR. With keywords, SQL is *case insensitive*, meaning that SQL treats uppercase and lowercase letters as equivalent in keywords.

4. For nonalphabetic characters, there is a defined order.

5. Bit strings whose length is a multiple of 4 can also be specified in *hexadecimal* notation, where the literal string is preceded by X and each hexadecimal character represents 4 bits.

the SQL implementation. The < (less than) comparison can be used with dates or times—an *earlier* date is considered to be smaller than a later date, and similarly with time. Literal values are represented by single-quoted strings preceded by the keyword DATE or TIME; for example, DATE '2002-09-27' or TIME '09:12:47'. In addition, a data type TIME(i), where i is called *time fractional seconds precision*, specifies $i + 1$ additional positions for TIME—one position for an additional separator character, and i positions for specifying decimal fractions of a second. A TIME WITH TIME ZONE data type includes an additional six positions for specifying the *displacement* from the standard universal time zone, which is in the range +13:00 to –12:59 in units of HOURS:MINUTES. If WITH TIME ZONE is not included, the default is the local time zone for the SQL session.

- A **timestamp** data type (TIMESTAMP) includes both the DATE and TIME fields, plus a minimum of six positions for decimal fractions of seconds and an optional WITH TIME ZONE qualifier. Literal values are represented by single-quoted strings preceded by the keyword TIMESTAMP, with a blank space between data and time; for example, TIMESTAMP '2002-09-27 09:12:47 648302'.
- Another data type related to DATE, TIME, and TIMESTAMP is the INTERVAL data type. This specifies an **interval**—a *relative value* that can be used to increment or decrement an absolute value of a date, time, or timestamp. Intervals are qualified to be either YEAR/MONTH intervals or DAY/TIME intervals.
- The format of DATE, TIME, and TIMESTAMP can be considered as a special type of string. Hence, they can generally be used in string comparisons by being **cast** (or **coerced** or converted) into the equivalent strings.

It is possible to specify the data type of each attribute directly, as in Figure 8.1; alternatively, a domain can be declared, and the domain name used with the attribute specification. This makes it easier to change the data type for a domain that is used by numerous attributes in a schema, and improves schema readability. For example, we can create a domain SSN_TYPE by the following statement:

```
CREATE DOMAIN SSN_TYPE AS CHAR(9);
```

We can use SSN_TYPE in place of CHAR(9) in Figure 8.1 for the attributes SSN and SUPERSSN of EMPLOYEE, MGRSSN of DEPARTMENT, ESSN of WORKS_ON, and ESSN of DEPENDENT. A domain can also have an optional default specification via a DEFAULT clause, as we discuss later for attributes.

8.2 Specifying Basic Constraints in SQL

We now describe the basic constraints that can be specified in SQL as part of table creation. These include key and referential integrity constraints, as well as restrictions on attribute domains and NULLs, and constraints on individual tuples within a relation. We discuss the specification of more general constraints, called assertions, in Secion 9.1.

8.2.1 Specifying Attribute Constraints and Attribute Defaults

Because SQL allows NULLs as attribute values, a *constraint* NOT NULL may be specified if NULL is not permitted for a particular attribute. This is always implicitly specified for the attributes that are part of the *primary key* of each relation, but it can be specified for any other attributes whose values are required not to be NULL, as shown in Figure 8.1.

It is also possible to define a *default value* for an attribute by appending the clause DEFAULT <value> to an attribute definition. The default value is included in any new tuple if an explicit value is not provided for that attribute. Figure 8.2 illustrates an example of specifying a default manager for a new department and a default department for a new employee. If no default clause is specified, the default *default value* is NULL for attributes *that do not have* the NOT NULL constraint.

Another type of constraint can restrict attribute or domain values using the CHECK clause following an attribute or domain definition.[6] For example, suppose that department numbers are restricted to integer numbers between 1 and 20; then, we can change the attribute declaration of DNUMBER in the DEPARTMENT table (see Figure 8.1) to the following:

```
DNUMBER INT NOT NULL CHECK (DNUMBER > 0 AND DNUMBER < 21);
```

The CHECK clause can also be used in conjunction with the CREATE DOMAIN statement. For example, we can write the following statement:

```
CREATE DOMAIN D_NUM AS INTEGER CHECK
(D_NUM > 0 AND D_NUM < 21);
```

We can then use the created domain D_NUM as the attribute type for all attributes that refer to department numbers in Figure 8.1, such as DNUMBER of DEPARTMENT, DNUM of PROJECT, DNO of EMPLOYEE, and so on.

8.2.2 Specifying Key and Referential Integrity Constraints

Because keys and referential integrity constraints are very important, there are special clauses within the CREATE TABLE statement to specify them. Some examples to illustrate the specification of keys and referential integrity are shown in Figure 8.1.[7] The PRIMARY KEY clause specifies one or more attributes that make up the primary key of a relation. If a primary key has a *single* attribute, the clause can follow the attribute directly. For example,

6. The CHECK clause can also be used for other purposes, as we shall see.

7. Key and referential integrity constraints were not included in early versions of SQL. In some earlier implementations, keys were specified implicitly at the internal level via the CREATE INDEX command.

```
CREATE TABLE EMPLOYEE
    (...,
     DNO            INT   NOT NULL   DEFAULT 1,
    CONSTRAINT EMPPK
     PRIMARY KEY (SSN),
    CONSTRAINT EMPSUPERFK
     FOREIGN KEY (SUPERSSN) REFERENCES EMPLOYEE(SSN)
                 ON DELETE SET NULL   ON UPDATE CASCADE,
    CONSTRAINT EMPDEPTFK
     FOREIGN KEY (DNO) REFERENCES DEPARTMENT(DNUMBER)
                 ON DELETE SET DEFAULT   ON UPDATE CASCADE );

CREATE TABLE DEPARTMENT
    (...,
     MGRSSN   CHAR(9) NOT NULL DEFAULT '888665555',
     ...,
     CONSTRAINT DEPTPK
      PRIMARY KEY (DNUMBER),
     CONSTRAINT DEPTSK
      UNIQUE (DNAME),
     CONSTRAINT DEPTMGRFK
      FOREIGN KEY (MGRSSN) REFERENCES EMPLOYEE(SSN)
           ON DELETE SET DEFAULT   ON UPDATE CASCADE );

CREATE TABLE DEPT_LOCATIONS
    (...,
     PRIMARY KEY (DNUMBER, DLOCATION),
     FOREIGN KEY (DNUMBER) REFERENCES DEPARTMENT(DNUMBER)
      ON DELETE CASCADE   ON UPDATE CASCADE );
```

FIGURE 8.2 Example illustrating how default attribute values and referential triggerred actions are specified in SQL

the primary key of DEPARTMENT can be specified as follows (instead of the way it is specified in Figure 8.1):

```
DNUMBER INT PRIMARY KEY;
```

The UNIQUE clause specifies alternate (secondary) keys, as illustrated in the DEPARTMENT and PROJECT table declarations in Figure 8.1.

Referential integrity is specified via the FOREIGN KEY clause, as shown in Figure 8.1. As we discussed in Section 5.2.4, a referential integrity constraint can be violated when tuples are inserted or deleted, or when a foreign key or primary key attribute value is modified. The default action that SQL takes for an integrity violation is to **reject** the update operation that will cause a violation. However, the schema designer can specify an alternative action to be taken if a referential integrity constraint is violated, by attaching a **referential triggered action** clause to any foreign key constraint. The options include

SET NULL, CASCADE, and SET DEFAULT. An option must be qualified with either ON DELETE or ON UPDATE. We illustrate this with the examples shown in Figure 8.2. Here, the database designer chooses SET NULL ON DELETE and CASCADE ON UPDATE for the foreign key SUPERSSN of EMPLOYEE. This means that if the tuple for a supervising employee is *deleted*, the value of SUPERSSN is automatically set to NULL for all employee tuples that were referencing the deleted employee tuple. On the other hand, if the SSN value for a supervising employee is *updated* (say, because it was entered incorrectly), the new value is *cascaded* to SUPERSSN for all employee tuples referencing the updated employee tuple.

In general, the action taken by the DBMS for SET NULL or SET DEFAULT is the same for both ON DELETE or ON UPDATE: The value of the affected referencing attributes is changed to NULL for SET NULL, and to the specified default value for SET DEFAULT. The action for CASCADE ON DELETE is to delete all the referencing tuples, whereas the action for CASCADE ON UPDATE is to change the value of the foreign key to the updated (new) primary key value for all referencing tuples. It is the responsibility of the database designer to choose the appropriate action and to specify it in the database schema. As a general rule, the CASCADE option is suitable for "relationship" relations (see Section 7.1), such as WORKS_ON; for relations that represent multivalued attributes, such as DEPT_LOCATIONS; and for relations that represent weak entity types, such as DEPENDENT.

8.2.3 Giving Names to Constraints

Figure 8.2 also illustrates how a constraint may be given a **constraint name,** following the keyword **CONSTRAINT.** The names of all constraints within a particular schema must be unique. A constraint name is used to identify a particular constraint in case the constraint must be dropped later and replaced with another constraint, as we discuss in Section 8.3. Giving names to constraints is optional.

8.2.4 Specifying Constraints on Tuples Using CHECK

In addition to key and referential integrity constraints, which are specified by special keywords, other *table constraints* can be specified through additional CHECK clauses at the end of a CREATE TABLE statement. These can be called **tuple-based** constraints because they apply to each tuple *individually* and are checked whenever a tuple is inserted or modified. For example, suppose that the DEPARTMENT table in Figure 8.1 had an additional attribute DEPT_CREATE_DATE, which stores the date when the department was created. Then we could add the following CHECK clause at the end of the CREATE TABLE statement for the DEPARTMENT table to make sure that a manager's start date is later than the department creation date:

```
CHECK (DEPT_CREATE_DATE < MGRSTARTDATE);
```

The CHECK clause can also be used to specify more general constraints using the CREATE ASSERTION statement of SQL. We discuss this in Section 9.1 because it requires the full power of queries, which are discussed in Sections 8.4 and 8.5.

8.3 SCHEMA CHANGE STATEMENTS IN SQL

In this section, we give an overview of the **schema evolution commands** available in SQL, which can be used to alter a schema by adding or dropping tables, attributes, constraints, and other schema elements.

8.3.1 The DROP Command

The DROP command can be used to drop *named* schema elements, such as tables, domains, or constraints. One can also drop a schema. For example, if a whole schema is not needed any more, the DROP SCHEMA command can be used. There are two *drop behavior* options: CASCADE and RESTRICT. For example, to remove the COMPANY database schema and all its tables, domains, and other elements, the CASCADE option is used as follows:

DROP SCHEMA COMPANY **CASCADE;**

If the RESTRICT option is chosen in place of CASCADE, the schema is dropped only if it has *no elements* in it; otherwise, the DROP command will not be executed.

If a base relation within a schema is not needed any longer, the relation and its definition can be deleted by using the DROP TABLE command. For example, if we no longer wish to keep track of dependents of employees in the COMPANY database of Figure 8.1, we can get rid of the DEPENDENT relation by issuing the following command:

DROP TABLE DEPENDENT **CASCADE;**

If the RESTRICT option is chosen instead of CASCADE, a table is dropped only if it is *not referenced* in any constraints (for example, by foreign key definitions in another relation) or views (see Section 9.2). With the CASCADE option, all such constraints and views that reference the table are dropped automatically from the schema, along with the table itself.

The DROP command can also be used to drop other types of named schema elements, such as constraints or domains.

8.3.2 The ALTER Command

The definition of a base table or of other named schema elements can be changed by using the ALTER command. For base tables, the possible *alter table actions* include adding or dropping a column (attribute), changing a column definition, and adding or dropping table constraints. For example, to add an attribute for keeping track of jobs of employees to the EMPLOYEE base relations in the COMPANY schema, we can use the command

ALTER TABLE COMPANY.EMPLOYEE **ADD** JOB VARCHAR(12);

We must still enter a value for the new attribute JOB for each individual EMPLOYEE tuple. This can be done either by specifying a default clause or by using the UPDATE command (see Section 8.6). If no default clause is specified, the new attribute will have NULLs in all

the tuples of the relation immediately after the command is executed; hence, the NOT NULL constraint is *not allowed* in this case.

To drop a column, we must choose either CASCADE or RESTRICT for drop behavior. If CASCADE is chosen, all constraints and views that reference the column are dropped automatically from the schema, along with the column. If RESTRICT is chosen, the command is successful only if no views or constraints (or other elements) reference the column. For example, the following command removes the attribute ADDRESS from the EMPLOYEE base table:

ALTER TABLE COMPANY.EMPLOYEE **DROP** ADDRESS **CASCADE;**

It is also possible to alter a column definition by dropping an existing default clause or by defining a new default clause. The following examples illustrate this clause:

ALTER TABLE COMPANY.DEPARTMENT **ALTER** MGRSSN **DROP DEFAULT;**

ALTER TABLE COMPANY.DEPARTMENT **ALTER** MGRSSN **SET DEFAULT** "333445555";

One can also change the constraints specified on a table by adding or dropping a constraint. To be dropped, a constraint must have been given a name when it was specified. For example, to drop the constraint named EMPSUPERFK in Figure 8.2 from the EMPLOYEE relation, we write:

ALTER TABLE COMPANY.EMPLOYEE
DROP CONSTRAINT EMPSUPERFK **CASCADE;**

Once this is done, we can redefine a replacement constraint by adding a new constraint to the relation, if needed. This is specified by using the ADD keyword in the ALTER TABLE statement followed by the new constraint, which can be named or unnamed and can be of any of the table constraint types discussed.

The preceding subsections gave an overview of the schema evolution commands of SQL. There are many other details and options, and we refer the interested reader to the SQL documents listed in the bibliographical notes. The next two sections discuss the querying capabilities of SQL.

8.4 Basic Queries in SQL

SQL has one basic statement for retrieving information from a database: the SELECT statement. The SELECT statement *has no relationship* to the SELECT operation of relational algebra, which was discussed in Chapter 6. There are many options and flavors to the SELECT statement in SQL, so we will introduce its features gradually. We will use example queries specified on the schema of Figure 5.5 and will refer to the sample database state shown in Figure 5.6 to show the results of some of the example queries.

Before proceeding, we must point out an important distinction between SQL and the formal relational model discussed in Chapter 5: SQL allows a table (relation) to have two or more tuples that are identical in all their attribute values. Hence, in general, an SQL table is not a *set of tuples*, because a set does not allow two identical members; rather, it is a **multiset** (sometimes called a *bag*) of tuples. Some SQL relations are *constrained to be sets* because a key constraint has been declared or because the DISTINCT option has been used with the SELECT statement (described later in this section). We should be aware of this distinction as we discuss the examples.

8.4.1 The SELECT-FROM-WHERE Structure of Basic SQL Queries

Queries in SQL can be very complex. We will start with simple queries, and then progress to more complex ones in a step-by-step manner. The basic form of the SELECT statement, sometimes called a **mapping** or a **select-from-where block,** is formed of the three clauses SELECT, FROM, and WHERE and has the following form:

```
SELECT   <attribute list>
FROM     <table list>
WHERE    <condition>;
```

where

- <attribute list> is a list of attribute names whose values are to be retrieved by the query.
- <table list> is a list of the relation names required to process the query.
- <condition> is a conditional (Boolean) expression that identifies the tuples to be retrieved by the query.

In SQL, the basic logical comparison operators for comparing attribute values with one another and with literal constants are =, <, <=, >, >=, and <>. These correspond to the relational algebra operators =, <, ≤, >, ≥, and ≠, respectively, and to the C/C++ programming language operators =, <, <=, >, >=, and !=. The main difference is the *not equal* operator. SQL has many additional comparison operators that we shall present gradually as needed.

We now illustrate the basic SELECT statement in SQL with some example queries. The queries are labeled here with the same query numbers that appear in Chapter 6 for easy cross reference.

QUERY 0

Retrieve the birthdate and address of the employee(s) whose name is 'John B. Smith'.

```
Q0: SELECT BDATE, ADDRESS
    FROM   EMPLOYEE
    WHERE  FNAME='John' AND MINIT='B' AND LNAME='Smith';
```

This query involves only the EMPLOYEE relation listed in the FROM clause. The query *selects* the EMPLOYEE tuples that satisfy the condition of the WHERE clause, then *projects* the result on the BDATE and ADDRESS attributes listed in the SELECT clause. Q0 is similar to the following relational algebra expression, except that duplicates, if any, would *not* be eliminated:

$$\pi_{\text{BDATE, ADDRESS}}(\sigma_{\text{FNAME='John' AND MINIT='B' AND LNAME='Smith'}}(\text{EMPLOYEE}))$$

Hence, a simple SQL query with a single relation name in the FROM clause is similar to a SELECT-PROJECT pair of relational algebra operations. The SELECT clause of SQL specifies the *projection attributes*, and the WHERE clause specifies the *selection condition*. The only difference is that in the SQL query we may get duplicate tuples in the result, because the constraint that a relation is a set is not enforced. Figure 8.3a shows the result of query Q0 on the database of Figure 5.6.

The query Q0 is also similar to the following tuple relational calculus expression, except that duplicates, if any, would again *not* be eliminated in the SQL query:

Q0: {t.BDATE, t.ADDRESS | EMPLOYEE(t) **AND** t.FNAME='John' **AND** t.MINIT='B' **AND** t.LNAME='Smith'}

Hence, we can think of an implicit tuple variable in the SQL query ranging over each tuple in the EMPLOYEE table and evaluating the condition in the WHERE clause. Only those tuples that satisfy the condition—that is, those tuples for which the condition evaluates to TRUE after substituting their corresponding attribute values—are selected.

QUERY 1

Retrieve the name and address of all employees who work for the 'Research' department.

```
Q1: SELECT  FNAME, LNAME, ADDRESS
    FROM    EMPLOYEE, DEPARTMENT
    WHERE   DNAME='Research' AND DNUMBER=DNO;
```

Query Q1 is similar to a SELECT-PROJECT-JOIN sequence of relational algebra operations. Such queries are often called **select-project-join queries.** In the WHERE clause of Q1, the condition DNAME = 'Research' is a **selection condition** and corresponds to a SELECT operation in the relational algebra. The condition DNUMBER = DNO is a **join condition,** which corresponds to a JOIN condition in the relational algebra. The result of query Q1 is shown in Figure 8.3b. In general, any number of select and join conditions may be specified in a single SQL query. The next example is a select-project-join query with *two* join conditions.

QUERY 2

For every project located in 'Stafford', list the project number, the controlling department number, and the department manager's last name, address, and birthdate.

```
Q2: SELECT  PNUMBER, DNUM, LNAME, ADDRESS, BDATE
    FROM    PROJECT, DEPARTMENT, EMPLOYEE
```

(a)

BDATE	ADDRESS
1965-01-09	731 Fondren, Houston, TX

(b)

FNAME	LNAME	ADDRESS
John	Smith	731 Fondren, Houston, TX
Franklin	Wong	638 Voss, Houston, TX
Ramesh	Narayan	975 Fire Oak, Humble, TX
Joyce	English	5631 Rice, Houston, TX

(c)

PNUMBER	DNUM	LNAME	ADDRESS	BDATE
10	4	Wallace	291 Berry, Bellaire, TX	1941-06-20
30	4	Wallace	291 Berry, Bellaire, TX	1941-06-20

(d)

E.FNAME	E.LNAME	S.FNAME	S.LNAME
John	Smith	Franklin	Wong
Franklin	Wong	James	Borg
Alicia	Zelaya	Jennifer	Wallace
Jennifer	Wallace	James	Borg
Ramesh	Narayan	Franklin	Wong
Joyce	English	Franklin	Wong
Ahmad	Jabbar	Jennifer	Wallace

(e)

SSN
123456789
333445555
999887777
987654321
666884444
453453453
987987987
888665555

(f)

SSN	DNAME
123456789	Research
333445555	Research
999887777	Research
987654321	Research
666884444	Research
453453453	Research
987987987	Research
888665555	Research
123456789	Administration
333445555	Administration
999887777	Administration
987654321	Administration
666884444	Administration
453453453	Administration
987987987	Administration
888665555	Administration
123456789	Headquarters
333445555	Headquarters
999887777	Headquarters
987654321	Headquarters
666884444	Headquarters
453453453	Headquarters
987987987	Headquarters
888665555	Headquarters

(g)

FNAME	MINIT	LNAME	SSN	BDATE	ADDRESS	SEX	SALARY	SUPERSSN	DNO
John	B	Smith	123456789	1965-09-01	731 Fondren, Houston, TX	M	30000	333445555	5
Franklin	T	Wong	333445555	1955-12-08	638 Voss, Houston, TX	M	40000	888665555	5
Ramesh	K	Narayan	666884444	1962-09-15	975 Fire Oak, Humble, TX	M	38000	333445555	5
Joyce	A	English	453453453	1972-07-31	5631 Rice, Houston, TX	F	25000	333445555	5

FIGURE 8.3 Results of SQL queries when applied to the COMPANY database state shown in Figure 5.6. (a) Q0. (b) Q1. (c) Q2. (d) Q8. (e) Q9. (f) Q10. (g) Q1C

```
    WHERE  DNUM=DNUMBER AND MGRSSN=SSN AND
           PLOCATION='Stafford';
```

The join condition DNUM = DNUMBER relates a project to its controlling department, whereas the join condition MGRSSN = SSN relates the controlling department to the employee who manages that department. The result of query Q2 is shown in Figure 8.3c.

8.4.2 Ambiguous Attribute Names, Aliasing, and Tuple Variables

In SQL the same name can be used for two (or more) attributes as long as the attributes are in *different relations*. If this is the case, and a query refers to two or more attributes with the same name, we must **qualify** the attribute name with the relation name to prevent ambiguity. This is done by *prefixing* the relation name to the attribute name and separating the two by a period. To illustrate this, suppose that in Figures 5.5 and 5.6 the DNO and LNAME attributes of the EMPLOYEE relation were called DNUMBER and NAME, and the DNAME attribute of DEPARTMENT was also called NAME; then, to prevent ambiguity, query Q1 would be rephrased as shown in Q1A. We must prefix the attributes NAME and DNUMBER in Q1A to specify which ones we are referring to, because the attribute names are used in both relations:

```
Q1A: SELECT  FNAME, EMPLOYEE.NAME, ADDRESS
     FROM    EMPLOYEE, DEPARTMENT
     WHERE   DEPARTMENT.NAME='Research' AND
             DEPARTMENT.DNUMBER=EMPLOYEE.DNUMBER;
```

Ambiguity also arises in the case of queries that refer to the same relation twice, as in the following example.

QUERY 8

For each employee, retrieve the employee's first and last name and the first and last name of his or her immediate supervisor.

```
Q8: SELECT  E.FNAME, E.LNAME, S.FNAME, S.LNAME
    FROM    EMPLOYEE AS E, EMPLOYEE AS S
    WHERE   E.SUPERSSN=S.SSN;
```

In this case, we are allowed to declare alternative relation names E and S, called **aliases** or **tuple variables,** for the EMPLOYEE relation. An alias can follow the keyword AS, as shown in Q8, or it can directly follow the relation name—for example, by writing EMPLOYEE E, EMPLOYEE S in the FROM clause of Q8. It is also possible to rename the relation attributes within the query in SQL by giving them aliases. For example, if we write

```
EMPLOYEE AS E(FN, MI, LN, SSN, BD, ADDR, SEX, SAL, SSSN, DNO)
```

in the FROM clause, FN becomes an alias for FNAME, MI for MINIT, LN for LNAME, and so on.

In Q8, we can think of E and S as two *different copies* of the EMPLOYEE relation; the first, E, represents employees in the role of supervisees; the second, S, represents employees in the role of supervisors. We can now join the two copies. Of course, in reality there is *only one* EMPLOYEE relation, and the join condition is meant to join the relation with itself by matching the tuples that satisfy the join condition E.SUPERSSN = S.SSN. Notice that this is an example of a one-level recursive query, as we discussed in Section 6.4.2. In earlier versions of SQL, as in relational algebra, it was not possible to specify a general recursive query, with

an unknown number of levels, in a single SQL statement. A construct for specifying recursive queries has been incorporated into SQL-99, as described in Chapter 22.

The result of query Q8 is shown in Figure 8.3d. Whenever one or more aliases are given to a relation, we can use these names to represent different references to that relation. This permits multiple references to the same relation within a query. Notice that, if we want to, we can use this alias-naming mechanism in any SQL query to specify tuple variables for every table in the WHERE clause, whether or not the same relation needs to be referenced more than once. In fact, this practice is recommended since it results in queries that are easier to comprehend. For example, we could specify query Q1A as in Q1B:

```
Q1B: SELECT  E.FNAME, E.NAME, E.ADDRESS
     FROM    EMPLOYEE E, DEPARTMENT D
     WHERE   D.NAME='Research' AND D.DNUMBER=E.DNUMBER;
```

If we specify tuple variables for every table in the WHERE clause, a select-project-join query in SQL closely resembles the corresponding tuple relational calculus expression (except for duplicate elimination). For example, compare Q1B with the following tuple relational calculus expression:

```
Q1: {e.FNAME, e.LNAME, e.ADDRESS | EMPLOYEE(e) AND (∃d)
    (DEPARTMENT(d) AND d.DNAME='Research' AND d.DNUMBER=e.DNO) }
```

Notice that the main difference—other than syntax—is that in the SQL query, the existential quantifier is not specified explicitly.

8.4.3 Unspecified WHERE Clause and Use of the Asterisk

We discuss two more features of SQL here. A *missing* WHERE clause indicates no condition on tuple selection; hence, *all tuples* of the relation specified in the FROM clause qualify and are selected for the query result. If more than one relation is specified in the FROM clause and there is no WHERE clause, then the CROSS PRODUCT—*all possible tuple combinations*—of these relations is selected. For example, Query 9 selects all EMPLOYEE SSNs (Figure 8.3e), and Query 10 selects all combinations of an EMPLOYEE SSN and a DEPARTMENT DNAME (Figure 8.3f).

QUERIES 9 AND 10

Select all EMPLOYEE SSNs (Q9), and all combinations of EMPLOYEE SSN and DEPARTMENT DNAME (Q10) in the database.

```
Q9:  SELECT  SSN
     FROM    EMPLOYEE;

Q10: SELECT  SSN, DNAME
     FROM    EMPLOYEE, DEPARTMENT;
```

It is extremely important to specify every selection and join condition in the WHERE clause; if any such condition is overlooked, incorrect and very large relations may result. Notice that Q10 is similar to a CROSS PRODUCT operation followed by a PROJECT operation in relational algebra. If we specify all the attributes of EMPLOYEE and DEPARTMENT in Q10, we get the CROSS PRODUCT (except for duplicate elimination, if any).

To retrieve all the attribute values of the selected tuples, we do not have to list the attribute names explicitly in SQL; we just specify an *asterisk* (*), which stands for *all the attributes*. For example, query Q1C retrieves all the attribute values of any EMPLOYEE who works in DEPARTMENT number 5 (Figure 8.3g), query Q1D retrieves all the attributes of an EMPLOYEE and the attributes of the DEPARTMENT in which he or she works for every employee of the 'Research' department, and Q10A specifies the CROSS PRODUCT of the EMPLOYEE and DEPARTMENT relations.

```
Q1C:  SELECT *
      FROM   EMPLOYEE
      WHERE  DNO=5;

Q1D:  SELECT *
      FROM   EMPLOYEE, DEPARTMENT
      WHERE  DNAME='Research' AND DNO=DNUMBER;

Q10A: SELECT *
      FROM   EMPLOYEE, DEPARTMENT;
```

8.4.4 Tables as Sets in SQL

As we mentioned earlier, SQL usually treats a table not as a set but rather as a **multiset;** *duplicate tuples can appear more than once* in a table, and in the result of a query. SQL does not automatically eliminate duplicate tuples in the results of queries, for the following reasons:

- Duplicate elimination is an expensive operation. One way to implement it is to sort the tuples first and then eliminate duplicates.
- The user may want to see duplicate tuples in the result of a query.
- When an aggregate function (see Section 8.5.7) is applied to tuples, in most cases we do not want to eliminate duplicates.

An SQL table with a key is restricted to being a set, since the key value must be distinct in each tuple.[8] If we *do want* to eliminate duplicate tuples from the result of an SQL query, we use the keyword **DISTINCT** in the SELECT clause, meaning that only distinct tuples should remain in the result. In general, a query with SELECT DISTINCT eliminates duplicates, whereas a query with SELECT ALL does not. Specifying SELECT with neither ALL nor DISTINCT—as in our previous examples—is equivalent to SELECT ALL. For

8. In general, an SQL table is not required to have a key, although in most cases there will be one.

example, Query 11 retrieves the salary of every employee; if several employees have the same salary, that salary value will appear as many times in the result of the query, as shown in Figure 8.4a. If we are interested only in distinct salary values, we want each value to appear only once, regardless of how many employees earn that salary. By using the keyword DISTINCT as in Q11A, we accomplish this, as shown in Figure 8.4b.

QUERY 11

Retrieve the salary of every employee (Q11) and all distinct salary values (Q11A).

```
Q11:  SELECT ALL SALARY
      FROM   EMPLOYEE;

Q11A: SELECT DISTINCT SALARY
      FROM   EMPLOYEE;
```

SQL has directly incorporated some of the set operations of relational algebra. There are set union (UNION), set difference (EXCEPT), and set intersection (INTERSECT) operations. The relations resulting from these set operations are sets of tuples; that is, *duplicate tuples are eliminated from the result*. Because these set operations apply only to *union-compatible relations*, we must make sure that the two relations on which we apply the operation have the same attributes and that the attributes appear in the same order in both relations. The next example illustrates the use of UNION.

QUERY 4

Make a list of all project numbers for projects that involve an employee whose last name is 'Smith', either as a worker or as a manager of the department that controls the project.

```
Q4:  (SELECT DISTINCT PNUMBER
      FROM   PROJECT, DEPARTMENT, EMPLOYEE
```

(a)

SALARY
30000
40000
25000
43000
38000
25000
25000
55000

(b)

SALARY
30000
40000
25000
43000
38000
55000

(c)

FNAME	LNAME

(d)

FNAME	LNAME
James	Borg

FIGURE 8.4 Results of additional SQL queries when applied to the COMPANY database state shown in Figure 5.6. (a) Q11. (b) Q11A. (c) Q16. (d) Q18.

```
WHERE   DNUM=DNUMBER AND MGRSSN=SSN AND LNAME='Smith')
UNION
(SELECT DISTINCT PNUMBER
FROM    PROJECT, WORKS_ON, EMPLOYEE
WHERE   PNUMBER=PNO AND ESSN=SSN AND LNAME='Smith');
```

The first SELECT query retrieves the projects that involve a 'Smith' as manager of the department that controls the project, and the second retrieves the projects that involve a 'Smith' as a worker on the project. Notice that if several employees have the last name 'Smith', the project names involving any of them will be retrieved. Applying the UNION operation to the two SELECT queries gives the desired result.

SQL also has corresponding multiset operations, which are followed by the keyword ALL (UNION ALL, EXCEPT ALL, INTERSECT ALL). Their results are multisets (duplicates are not eliminated). The behavior of these operations is illustrated by the examples in Figure 8.5. Basically, each tuple—whether it is a duplicate or not—is considered as a different tuple when applying these operations.

8.4.5 Substring Pattern Matching and Arithmetic Operators

In this section we discuss several more features of SQL. The first feature allows comparison conditions on only parts of a character string, using the LIKE comparison operator. This

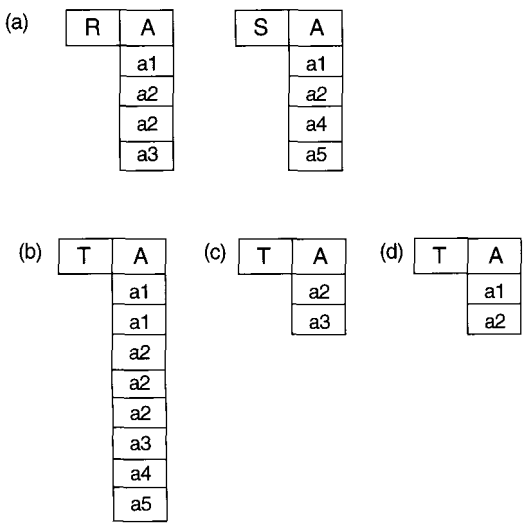


FIGURE 8.5 The results of SQL multiset operations. (a) Two tables, R(A) and S(A). (b) R(A) UNION ALL S(A). (c) R(A) EXCEPT ALL S(A). (d) R(A) INTERSECT ALL S(A).

can be used for string **pattern matching.** Partial strings are specified using two reserved characters: % replaces an arbitrary number of zero or more characters, and the underscore (_) replaces a single character. For example, consider the following query.

QUERY 12

Retrieve all employees whose address is in Houston, Texas.

```
Q12:  SELECT  FNAME, LNAME
      FROM    EMPLOYEE
      WHERE   ADDRESS LIKE '%Houston,TX%';
```

To retrieve all employees who were born during the 1950s, we can use Query 12A. Here, '5' must be the third character of the string (according to our format for date), so we use the value '_ _ 5 _ _ _ _ _ _ _', with each underscore serving as a placeholder for an arbitrary character.

QUERY 12A

Find all employees who were born during the 1950s.

```
Q12A: SELECT  FNAME, LNAME
      FROM    EMPLOYEE
      WHERE   BDATE LIKE '_ _ 5 _ _ _ _ _ _ _';
```

If an underscore or % is needed as a literal character in the string, the character should be preceded by an *escape character*, which is specified after the string using the keyword ESCAPE. For example, 'AB_CD\%EF' ESCAPE '\' represents the literal string 'AB_CD%EF', because \ is specified as the escape character. Any character not used in the string can be chosen as the escape character. Also, we need a rule to specify apostrophes or single quotation marks (") if they are to be included in a string, because they are used to begin and end strings. If an apostrophe (') is needed, it is represented as two consecutive apostrophes (") so that it will not be interpreted as ending the string.

Another feature allows the use of arithmetic in queries. The standard arithmetic operators for addition (+), subtraction (–), multiplication (*), and division (/) can be applied to numeric values or attributes with numeric domains. For example, suppose that we want to see the effect of giving all employees who work on the 'ProductX' project a 10 percent raise; we can issue Query 13 to see what their salaries would become. This example also shows how we can rename an attribute in the query result using AS in the SELECT clause.

QUERY 13

Show the resulting salaries if every employee working on the 'ProductX' project is given a 10 percent raise.

```
Q13:  SELECT  FNAME, LNAME, 1.1*SALARY AS INCREASED_SAL
      FROM    EMPLOYEE, WORKS_ON, PROJECT
```

```
WHERE   SSN=ESSN AND PNO=PNUMBER AND
        PNAME='ProductX';
```

For string data types, the concatenate operator | | can be used in a query to append two string values. For date, time, timestamp, and interval data types, operators include incrementing (+) or decrementing (–) a date, time, or timestamp by an interval. In addition, an interval value is the result of the difference between two date, time, or timestamp values. Another comparison operator that can be used for convenience is BETWEEN, which is illustrated in Query 14.

QUERY 14

Retrieve all employees in department 5 whose salary is between $30,000 and $40,000.

```
Q14:  SELECT  *
      FROM    EMPLOYEE
      WHERE   (SALARY BETWEEN 30000 AND 40000) AND DNO = 5;
```

The condition (SALARY BETWEEN 30000 AND 40000) in Q14 is equivalent to the condition ((SALARY >= 30000) AND (SALARY <= 40000)).

8.4.6 Ordering of Query Results

SQL allows the user to order the tuples in the result of a query by the values of one or more attributes, using the ORDER BY clause. This is illustrated by Query 15.

QUERY 15

Retrieve a list of employees and the projects they are working on, ordered by department and, within each department, ordered alphabetically by last name, first name.

```
Q15:  SELECT    DNAME, LNAME, FNAME, PNAME
      FROM      DEPARTMENT, EMPLOYEE, WORKS_ON, PROJECT
      WHERE     DNUMBER=DNO AND SSN=ESSN AND PNO=PNUMBER
      ORDER BY  DNAME, LNAME, FNAME;
```

The default order is in ascending order of values. We can specify the keyword DESC if we want to see the result in a descending order of values. The keyword ASC can be used to specify ascending order explicitly. For example, if we want descending order on DNAME and ascending order on LNAME, FNAME, the ORDER BY clause of Q15 can be written as

```
ORDER BY DNAME DESC, LNAME ASC, FNAME ASC
```

8.5 More Complex SQL Queries

In the previous section, we described some basic types of queries in SQL. Because of the generality and expressive power of the language, there are many additional features that allow users to specify more complex queries. We discuss several of these features in this section.

8.5.1 Comparisons Involving NULL and Three-Valued Logic

SQL has various rules for dealing with NULL values. Recall from Section 5.1.2 that NULL is used to represent a missing value, but that it usually has one of three different interpretations—value unknown (exists but is not known), value not available (exists but is purposely withheld), or attribute not applicable (undefined for this tuple). Consider the following examples to illustrate each of the three meanings of NULL.

1. *Unknown value:* A particular person has a date of birth but it is not known, so it is represented by NULL in the database.
2. *Unavailable or withheld value:* A person has a home phone but does not want it to be listed, so it is withheld and represented as NULL in the database.
3. *Not applicable attribute:* An attribute LastCollegeDegree would be NULL for a person who has no college degrees, because it does not apply to that person.

It is often not possible to determine which of the three meanings is intended; for example, a NULL for the home phone of a person can have any of the three meanings. Hence, SQL does not distinguish between the different meanings of NULL.

In general, each NULL is considered to be different from every other NULL in the database. When a NULL is involved in a comparison operation, the result is considered to be UNKNOWN (it may be TRUE or it may be FALSE). Hence, SQL uses a three-valued logic with values TRUE, FALSE, and UNKNOWN instead of the standard two-valued logic with values TRUE or FALSE. It is therefore necessary to define the results of three-valued logical expressions when the logical connectives AND, OR, and NOT are used. Table 8.1 shows the resulting values.

In select-project-join queries, the general rule is that only those combinations of tuples that evaluate the logical expression of the query to TRUE are selected. Tuple combinations that evaluate to FALSE or UNKNOWN are not selected. However, there are exceptions to that rule for certain operations, such as outer joins, as we shall see.

SQL allows queries that check whether an attribute value is NULL. Rather than using = or <> to compare an attribute value to NULL, SQL uses IS or IS NOT. This is because SQL considers each NULL value as being distinct from every other NULL value, so equality comparison is not appropriate. It follows that when a join condition is specified, tuples with NULL values for the join attributes are not included in the result (unless it is an OUTER JOIN; see Section 8.5.6). Query 18 illustrates this; its result is shown in Figure 8.4d.

TABLE 8.1 LOGICAL CONNECTIVES IN THREE-VALUED LOGIC

AND	TRUE	FALSE	UNKNOWN
TRUE	TRUE	FALSE	UNKNOWN
FALSE	FALSE	FALSE	FALSE
UNKNOWN	UNKNOWN	FALSE	UNKNOWN

OR	TRUE	FALSE	UNKNOWN
TRUE	TRUE	TRUE	TRUE
FALSE	TRUE	FALSE	UNKNOWN
UNKNOWN	TRUE	UNKNOWN	UNKNOWN

NOT	
TRUE	FALSE
FALSE	TRUE
UNKNOWN	UNKNOWN

QUERY 18

Retrieve the names of all employees who do not have supervisors.

```
Q18: SELECT FNAME, LNAME
     FROM   EMPLOYEE
     WHERE  SUPERSSN IS NULL;
```

8.5.2 Nested Queries, Tuples, and Set/Multiset Comparisons

Some queries require that existing values in the database be fetched and then used in a comparison condition. Such queries can be conveniently formulated by using **nested queries,** which are complete select-from-where blocks within the WHERE clause of another query. That other query is called the **outer query.** Query 4 is formulated in Q4 without a nested query, but it can be rephrased to use nested queries as shown in Q4A. Q4A introduces the comparison operator IN, which compares a value v with a set (or multiset) of values V and evaluates to TRUE if v is one of the elements in V.

```
Q4A: SELECT  DISTINCT PNUMBER
     FROM    PROJECT
     WHERE   PNUMBER IN (SELECT  PNUMBER
                         FROM    PROJECT, DEPARTMENT,
                                 EMPLOYEE
                         WHERE   DNUM=DNUMBER AND
```

```
                                MGRSSN=SSN AND
                                LNAME='Smith')
    OR
    PNUMBER IN (SELECT  PNO
                FROM    WORKS_ON, EMPLOYEE
                WHERE   ESSN=SSN AND
                        LNAME='Smith');
```

The first nested query selects the project numbers of projects that have a 'Smith' involved as manager, while the second selects the project numbers of projects that have a 'Smith' involved as worker. In the outer query, we use the OR logical connective to retrieve a PROJECT tuple if the PNUMBER value of that tuple is in the result of either nested query.

If a nested query returns a single attribute *and* a single tuple, the query result will be a single (scalar) value. In such cases, it is permissible to use = instead of IN for the comparison operator. In general, the nested query will return a **table** (relation), which is a set or multiset of tuples.

SQL allows the use of **tuples** of values in comparisons by placing them within parentheses. To illustrate this, consider the following query:

```
SELECT DISTINCT ESSN
FROM   WORKS_ON
WHERE  (PNO, HOURS) IN (SELECT PNO, HOURS FROM WORKS_ON
                        WHERE SSN='123456789');
```

This query will select the social security numbers of all employees who work the same (project, hours) combination on some project that employee 'John Smith' (whose SSN = '123456789') works on. In this example, the IN operator compares the subtuple of values in parentheses (PNO, HOURS) for each tuple in WORKS_ON with the set of union-compatible tuples produced by the nested query.

In addition to the IN operator, a number of other comparison operators can be used to compare a single value v (typically an attribute name) to a set or multiset V (typically a nested query). The = ANY (or = SOME) operator returns TRUE if the value v is equal to *some value* in the set V and is hence equivalent to IN. The keywords ANY and SOME have the same meaning. Other operators that can be combined with ANY (or SOME) include >, >=, <, <=, and <>. The keyword ALL can also be combined with each of these operators. For example, the comparison condition (v > ALL V) returns TRUE if the value v is greater than *all* the values in the set (or multiset) V. An example is the following query, which returns the names of employees whose salary is greater than the salary of all the employees in department 5:

```
SELECT LNAME, FNAME
FROM   EMPLOYEE
WHERE  SALARY > ALL (SELECT SALARY FROM EMPLOYEE
                     WHERE DNO=5);
```

In general, we can have several levels of nested queries. We can once again be faced with possible ambiguity among attribute names if attributes of the same name exist—one in a relation in the FROM clause of the *outer query*, and another in a relation in the FROM clause of the *nested query*. The rule is that a reference to an *unqualified attribute* refers to the relation declared in the **innermost nested query.** For example, in the SELECT clause and WHERE clause of the first nested query of Q4A, a reference to any unqualified attribute of the PROJECT relation refers to the PROJECT relation specified in the FROM clause of the nested query. To refer to an attribute of the PROJECT relation specified in the outer query, we can specify and refer to an *alias* (tuple variable) for that relation. These rules are similar to scope rules for program variables in most programming languages that allow nested procedures and functions. To illustrate the potential ambiguity of attribute names in nested queries, consider Query 16, whose result is shown in Figure 8.4c.

QUERY 16

Retrieve the name of each employee who has a dependent with the same first name and same sex as the employee.

```
Q16: SELECT E.FNAME, E.LNAME
     FROM   EMPLOYEE AS E
     WHERE  E.SSN IN (SELECT ESSN
                      FROM   DEPENDENT
                      WHERE  E.FNAME=DEPENDENT_NAME
                       AND E.SEX=SEX);
```

In the nested query of Q16, we must qualify E.SEX because it refers to the SEX attribute of EMPLOYEE from the outer query, and DEPENDENT also has an attribute called SEX. All unqualified references to SEX in the nested query refer to SEX of DEPENDENT. However, we do not *have to* qualify FNAME and SSN because the DEPENDENT relation does not have attributes called FNAME and SSN, so there is no ambiguity.

It is generally advisable to create tuple variables (aliases) for *all the tables referenced in an SQL query* to avoid potential errors and ambiguities.

8.5.3 Correlated Nested Queries

Whenever a condition in the WHERE clause of a nested query references some attribute of a relation declared in the outer query, the two queries are said to be **correlated.** We can understand a correlated query better by considering that the *nested query is evaluated once for each tuple (or combination of tuples) in the outer query*. For example, we can think of Q16 as follows: For *each* EMPLOYEE tuple, evaluate the nested query, which retrieves the ESSN values for all DEPENDENT tuples with the same sex and name as that EMPLOYEE tuple; if the SSN value of the EMPLOYEE tuple is *in* the result of the nested query, then select that EMPLOYEE tuple.

In general, a query written with nested select-from-where blocks and using the = or IN comparison operators can *always* be expressed as a single block query. For example, Q16 may be written as in Q16A:

```
Q16A: SELECT  E.FNAME, E.LNAME
      FROM    EMPLOYEE AS E, DEPENDENT AS D
      WHERE   E.SSN=D.ESSN AND E.SEX=D.SEX AND
              E.FNAME=D.DEPENDENT_NAME;
```

The original SQL implementation on SYSTEM R also had a CONTAINS comparison operator, which was used to compare two sets or multisets. This operator was subsequently dropped from the language, possibly because of the difficulty of implementing it efficiently. Most commercial implementations of SQL do *not* have this operator. The CONTAINS operator compares two sets of values and returns TRUE if one set contains all values in the other set. Query 3 illustrates the use of the CONTAINS operator.

QUERY 3

Retrieve the name of each employee who works on *all* the projects controlled by department number 5.

```
Q3: SELECT  FNAME, LNAME
    FROM    EMPLOYEE
    WHERE   (     (SELECT  PNO
                  FROM     WORKS_ON
                  WHERE    SSN=ESSN)
                  CONTAINS
                  (SELECT  PNUMBER
                  FROM     PROJECT
                  WHERE    DNUM=5) );
```

In Q3, the second nested query (which is not correlated with the outer query) retrieves the project numbers of all projects controlled by department 5. For *each* employee tuple, the first nested query (which is correlated) retrieves the project numbers on which the employee works; if these contain all projects controlled by department 5, the employee tuple is selected and the name of that employee is retrieved. Notice that the CONTAINS comparison operator has a similar function to the DIVISION operation of the relational algebra (see Section 6.3.4) and to universal quantification in relational calculus (see Section 6.6.6). Because the CONTAINS operation is not part of SQL, we have to use other techniques, such as the EXISTS function, to specify these types of queries, as described in Section 8.5.4.

8.5.4 The EXISTS and UNIQUE Functions in SQL

The EXISTS function in SQL is used to check whether the result of a correlated nested query is empty (contains no tuples) or not. We illustrate the use of EXISTS—and NOT

EXISTS—with some examples. First, we formulate Query 16 in an alternative form that uses EXISTS. This is shown as Q16B:

```
Q16B: SELECT  E.FNAME, E.LNAME
      FROM    EMPLOYEE AS E
      WHERE   EXISTS (SELECT *
                      FROM   DEPENDENT
                      WHERE  E.SSN=ESSN AND E.SEX=SEX
                             AND E.FNAME=DEPENDENT_NAME);
```

EXISTS and NOT EXISTS are usually used in conjunction with a correlated nested query. In Q16B, the nested query references the SSN, FNAME, and SEX attributes of the EMPLOYEE relation from the outer query. We can think of Q16B as follows: For each EMPLOYEE tuple, evaluate the nested query, which retrieves all DEPENDENT tuples with the same social security number, sex, and name as the EMPLOYEE tuple; if at least one tuple EXISTS in the result of the nested query, then select that EMPLOYEE tuple. In general, EXISTS(Q) returns **TRUE** if there is *at least one tuple* in the result of the nested query Q, and it returns **FALSE** otherwise. On the other hand, NOT EXISTS(Q) returns **TRUE** if there are *no tuples* in the result of nested query Q, and it returns **FALSE** otherwise. Next, we illustrate the use of NOT EXISTS.

QUERY 6

Retrieve the names of employees who have no dependents.

```
Q6:  SELECT  FNAME, LNAME
     FROM    EMPLOYEE
     WHERE   NOT EXISTS (SELECT *
                         FROM   DEPENDENT
                         WHERE  SSN=ESSN);
```

In Q6, the correlated nested query retrieves all DEPENDENT tuples related to a particular EMPLOYEE tuple. If *none exist*, the EMPLOYEE tuple is selected. We can explain Q6 as follows: For *each* EMPLOYEE tuple, the correlated nested query selects all DEPENDENT tuples whose ESSN value matches the EMPLOYEE SSN; if the result is empty, no dependents are related to the employee, so we select that EMPLOYEE tuple and retrieve its FNAME and LNAME.

QUERY 7

List the names of managers who have at least one dependent.

```
Q7:  SELECT  FNAME, LNAME
     FROM    EMPLOYEE
     WHERE   EXISTS     (SELECT *
                         FROM   DEPENDENT
                         WHERE  SSN=ESSN)
```

```
AND
EXISTS    (SELECT  *
           FROM    DEPARTMENT
           WHERE   SSN=MGRSSN);
```

One way to write this query is shown in Q7, where we specify two nested correlated queries; the first selects all DEPENDENT tuples related to an EMPLOYEE, and the second selects all DEPARTMENT tuples managed by the EMPLOYEE. If at least one of the first and at least one of the second exists, we select the EMPLOYEE tuple. Can you rewrite this query using only a single nested query or no nested queries?

Query 3 ("Retrieve the name of each employee who works on *all* the projects controlled by department number 5," see Section 8.5.3) can be stated using EXISTS and NOT EXISTS in SQL systems. There are two options. The first is to use the well-known set theory transformation that (*S1* CONTAINS *S2*) is logically equivalent to (*S2* EXCEPT *S1*) is empty.[9] This option is shown as Q3A.

```
Q3A: SELECT  FNAME, LNAME
     FROM    EMPLOYEE
     WHERE   NOT EXISTS
     (       (SELECT  PNUMBER
              FROM    PROJECT
              WHERE   DNUM=5)
     EXCEPT
              (SELECT  PNO
              FROM    WORKS_ON
              WHERE   SSN=ESSN) );
```

In Q3A, the first subquery (which is not correlated) selects all projects controlled by department 5, and the second subquery (which is correlated) selects all projects that the particular employee being considered works on. If the set difference of the first subquery MINUS (EXCEPT) the second subquery is empty, it means that the employee works on all the projects and is hence selected.

The second option is shown as Q3B. Notice that we need two-level nesting in Q3B and that this formulation is quite a bit more complex than Q3, which used the CONTAINS comparison operator, and Q3A, which uses NOT EXISTS and EXCEPT. However, CONTAINS is not part of SQL, and not all relational systems have the EXCEPT operator even though it is part of SQL-99.

```
Q3B: SELECT  LNAME, FNAME
     FROM    EMPLOYEE
```

9. Recall that EXCEPT is the set difference operator.

```
WHERE   NOT EXISTS
(SELECT *
FROM    WORKS_ON B
WHERE   (B.PNO IN   (SELECT  PNUMBER
                     FROM    PROJECT
                     WHERE   DNUM=5) )
        AND
        NOT EXISTS  (SELECT  *
                     FROM    WORKS_ON C
                     WHERE   C.ESSN=SSN
                     AND     C.PNO=B.PNO) );
```

In Q3B, the outer nested query selects any WORKS_ON (B) tuples whose PNO is of a project controlled by department 5, *if* there is not a WORKS_ON (C) tuple with the same PNO and the same SSN as that of the EMPLOYEE tuple under consideration in the outer query. If no such tuple exists, we select the EMPLOYEE tuple. The form of Q3B matches the following rephrasing of Query 3: Select each employee such that there does not exist a project controlled by department 5 that the employee does not work on. It corresponds to the way we wrote this query in tuple relation calculus in Section 6.6.6.

There is another SQL function, UNIQUE(Q), which returns TRUE if there are no duplicate tuples in the result of query Q; otherwise, it returns FALSE. This can be used to test whether the result of a nested query is a set or a multiset.

8.5.5 Explicit Sets and Renaming of Attributes in SQL

We have seen several queries with a nested query in the WHERE clause. It is also possible to use an **explicit set of values** in the WHERE clause, rather than a nested query. Such a set is enclosed in parentheses in SQL.

QUERY 17

Retrieve the social security numbers of all employees who work on project numbers 1, 2, or 3.

```
Q17:  SELECT  DISTINCT ESSN
      FROM    WORKS_ON
      WHERE   PNO IN (1, 2, 3);
```

In SQL, it is possible to rename any attribute that appears in the result of a query by adding the qualifier AS followed by the desired new name. Hence, the AS construct can be used to alias both attribute and relation names, and it can be used in both the SELECT and FROM clauses. For example, Q8A shows how query Q8 can be slightly changed to retrieve the last name of each employee and his or her supervisor, while renaming the resulting

attribute names as EMPLOYEE_NAME and SUPERVISOR_NAME. The new names will appear as column headers in the query result.

```
Q8A: SELECT  E.LNAME AS EMPLOYEE_NAME, S.LNAME AS
             SUPERVISOR_NAME
     FROM    EMPLOYEE AS E, EMPLOYEE AS S
     WHERE   E.SUPERSSN=S.SSN;
```

8.5.6 Joined Tables in SQL

The concept of a **joined table** (or **joined relation**) was incorporated into SQL to permit users to specify a table resulting from a join operation *in the FROM clause* of a query. This construct may be easier to comprehend than mixing together all the select and join conditions in the WHERE clause. For example, consider query Q1, which retrieves the name and address of every employee who works for the 'Research' department. It may be easier first to specify the join of the EMPLOYEE and DEPARTMENT relations, and then to select the desired tuples and attributes. This can be written in SQL as in Q1A:

```
Q1A: SELECT  FNAME, LNAME, ADDRESS
     FROM    (EMPLOYEE JOIN DEPARTMENT ON DNO=DNUMBER)
     WHERE   DNAME='Research';
```

The FROM clause in Q1A contains a single *joined table*. The attributes of such a table are all the attributes of the first table, EMPLOYEE, followed by all the attributes of the second table, DEPARTMENT. The concept of a joined table also allows the user to specify different types of join, such as NATURAL JOIN and various types of OUTER JOIN. In a NATURAL JOIN on two relations R and S, no join condition is specified; an implicit equijoin condition for *each pair of attributes with the same name* from R and S is created. Each such pair of attributes is included only once in the resulting relation (see Section 6.4.3).

If the names of the join attributes are not the same in the base relations, it is possible to rename the attributes so that they match, and then to apply NATURAL JOIN. In this case, the AS construct can be used to rename a relation and all its attributes in the FROM clause. This is illustrated in Q1B, where the DEPARTMENT relation is renamed as DEPT and its attributes are renamed as DNAME, DNO (to match the name of the desired join attribute DNO in EMPLOYEE), MSSN, and MSDATE. The implied join condition for this NATURAL JOIN is EMPLOYEE.DNO = DEPT.DNO, because this is the only pair of attributes with the same name after renaming.

```
Q1B: SELECT FNAME, LNAME, ADDRESS
     FROM   (EMPLOYEE NATURAL JOIN
             (DEPARTMENT AS DEPT (DNAME, DNO, MSSN, MSDATE)))
     WHERE  DNAME='Research;
```

The default type of join in a joined table is an **inner join,** where a tuple is included in the result only if a matching tuple exists in the other relation. For example, in query

Q8A, only employees that *have a supervisor* are included in the result; an EMPLOYEE tuple whose value for SUPERSSN is NULL is excluded. If the user requires that all employees be included, an OUTER JOIN must be used explicitly (see Section 6.4.3 for the definition of OUTER JOIN). In SQL, this is handled by explicitly specifying the OUTER JOIN in a joined table, as illustrated in Q8B:

```
Q8B: SELECT  E.LNAME AS EMPLOYEE_NAME,
             S.LNAME AS SUPERVISOR_NAME
     FROM    (EMPLOYEE AS E LEFT OUTER JOIN EMPLOYEE AS S
             ON E.SUPERSSN=S.SSN);
```

The options available for specifying joined tables in SQL include INNER JOIN (same as JOIN), LEFT OUTER JOIN, RIGHT OUTER JOIN, and FULL OUTER JOIN. In the latter three options, the keyword OUTER may be omitted. If the join attributes have the same name, one may also specify the natural join variation of outer joins by using the keyword NATURAL before the operation (for example, NATURAL LEFT OUTER JOIN). The keyword CROSS JOIN is used to specify the Cartesian product operation (see Section 6.2.2), although this should be used only with the utmost care because it generates all possible tuple combinations.

It is also possible to *nest* join specifications; that is, one of the tables in a join may itself be a joined table. This is illustrated by Q2A, which is a different way of specifying query Q2, using the concept of a joined table:

```
Q2A: SELECT  PNUMBER, DNUM, LNAME, ADDRESS, BDATE
     FROM    ((PROJECT JOIN DEPARTMENT ON DNUM=DNUMBER)
             JOIN EMPLOYEE ON MGRSSN=SSN)
     WHERE   PLOCATION='Stafford';
```

8.5.7 Aggregate Functions in SQL

In Section 6.4.1, we introduced the concept of an aggregate function as a relational operation. Because grouping and aggregation are required in many database applications, SQL has features that incorporate these concepts. A number of built-in functions exist: **COUNT, SUM, MAX, MIN,** and **AVG.**[10] The COUNT function returns the number of tuples or values as specified in a query. The functions SUM, MAX, MIN, and AVG are applied to a set or multiset of numeric values and return, respectively, the sum, maximum value, minimum value, and average (mean) of those values. These functions can be used in the SELECT clause or in a HAVING clause (which we introduce later). The functions MAX and MIN can also be used with attributes that have nonnumeric domains if the domain values have a *total ordering* among one another.[11] We illustrate the use of these functions with example queries.

10. Additional aggregate functions for more advanced statistical calculation have been added in SQL-99.

11. Total order means that for any two values in the domain, it can be determined that one appears before the other in the defined order; for example, DATE, TIME, and TIMESTAMP domains have total orderings on their values, as do alphabetic strings.

QUERY 19

Find the sum of the salaries of all employees, the maximum salary, the minimum salary, and the average salary.

```
Q19: SELECT  SUM (SALARY), MAX (SALARY), MIN (SALARY),
             AVG (SALARY)
     FROM    EMPLOYEE;
```

If we want to get the preceding function values for employees of a specific department—say, the 'Research' department—we can write Query 20, where the EMPLOYEE tuples are restricted by the WHERE clause to those employees who work for the 'Research' department.

QUERY 20

Find the sum of the salaries of all employees of the 'Research' department, as well as the maximum salary, the minimum salary, and the average salary in this department.

```
Q20: SELECT  SUM (SALARY), MAX (SALARY), MIN (SALARY),
             AVG (SALARY)
     FROM    (EMPLOYEE JOIN DEPARTMENT ON DNO=DNUMBER)
     WHERE   DNAME='Research';
```

QUERIES 21 AND 22

Retrieve the total number of employees in the company (Q21) and the number of employees in the 'Research' department (Q22).

```
Q21: SELECT  COUNT (*)
     FROM    EMPLOYEE;

Q22: SELECT  COUNT (*)
     FROM    EMPLOYEE, DEPARTMENT
     WHERE   DNO=DNUMBER AND DNAME='Research';
```

Here the asterisk (*) refers to the *rows* (tuples), so COUNT (*) returns the number of rows in the result of the query. We may also use the COUNT function to count values in a column rather than tuples, as in the next example.

QUERY 23

Count the number of distinct salary values in the database.

```
Q23: SELECT  COUNT (DISTINCT SALARY)
     FROM    EMPLOYEE;
```

If we write COUNT(SALARY) instead of COUNT(DISTINCT SALARY) in Q23, then duplicate values will not be eliminated. However, any tuples with NULL for SALARY will not be counted. In general, NULL values are **discarded** when aggregate functions are applied to a particular column (attribute).

The preceding examples summarize *a whole relation* (Q19, Q21, Q23) or a selected subset of tuples (Q20, Q22), and hence all produce single tuples or single values. They illustrate how functions are applied to retrieve a summary value or summary tuple from the database. These functions can also be used in selection conditions involving nested queries. We can specify a correlated nested query with an aggregate function, and then use the nested query in the WHERE clause of an outer query. For example, to retrieve the names of all employees who have two or more dependents (Query 5), we can write the following:

```
Q5:  SELECT  LNAME, FNAME
     FROM    EMPLOYEE
     WHERE   (SELECT  COUNT (*)
              FROM    DEPENDENT
              WHERE   SSN=ESSN)  >=  2;
```

The correlated nested query counts the number of dependents that each employee has; if this is greater than or equal to two, the employee tuple is selected.

8.5.8 Grouping: The GROUP BY and HAVING Clauses

In many cases we want to apply the aggregate functions *to subgroups of tuples in a relation*, where the subgroups are based on some attribute values. For example, we may want to find the average salary of employees *in each department* or the number of employees who work *on each project*. In these cases we need to **partition** the relation into nonoverlapping subsets (or **groups**) of tuples. Each group (partition) will consist of the tuples that have the same value of some attribute(s), called the **grouping attribute(s).** We can then apply the function to each such group independently. SQL has a GROUP BY clause for this purpose. The GROUP BY clause specifies the grouping attributes, which should *also appear in the* SELECT clause, so that the value resulting from applying each aggregate function to a group of tuples appears along with the value of the grouping attribute(s).

QUERY 24

For each department, retrieve the department number, the number of employees in the department, and their average salary.

```
Q24: SELECT    DNO, COUNT (*), AVG (SALARY)
     FROM      EMPLOYEE
     GROUP BY  DNO;
```

In Q24, the EMPLOYEE tuples are partitioned into groups—each group having the same value for the grouping attribute DNO. The COUNT and AVG functions are applied to each

such group of tuples. Notice that the SELECT clause includes only the grouping attribute and the functions to be applied on each group of tuples. Figure 8.6a illustrates how grouping works on Q24; it also shows the result of Q24.

If NULLs exist in the grouping attribute, then a **separate group** is created for all tuples with a *NULL value in the grouping attribute*. For example, if the EMPLOYEE table had some tuples that had NULL for the grouping attribute DNO, there would be a separate group for those tuples in the result of Q24.

QUERY 25

For each project, retrieve the project number, the project name, and the number of employees who work on that project.

```
Q25: SELECT    PNUMBER, PNAME, COUNT (*)
     FROM      PROJECT, WORKS_ON
     WHERE     PNUMBER=PNO
     GROUP BY  PNUMBER, PNAME;
```

Q25 shows how we can use a join condition in conjunction with GROUP BY. In this case, the grouping and functions are applied *after* the joining of the two relations. Sometimes we want to retrieve the values of these functions only for *groups that satisfy certain conditions*. For example, suppose that we want to modify Query 25 so that only projects with more than two employees appear in the result. SQL provides a **HAVING** clause, which can appear in conjunction with a GROUP BY clause, for this purpose. HAVING provides a condition on the group of tuples associated with each value of the grouping attributes. Only the groups that satisfy the condition are retrieved in the result of the query. This is illustrated by Query 26.

QUERY 26

For each project *on which more than two employees work*, retrieve the project number, the project name, and the number of employees who work on the project.

```
Q26: SELECT    PNUMBER, PNAME, COUNT (*)
     FROM      PROJECT, WORKS_ON
     WHERE     PNUMBER=PNO
     GROUP BY  PNUMBER, PNAME
     HAVING    COUNT (*) > 2;
```

Notice that, while selection conditions in the WHERE clause limit the *tuples* to which functions are applied, the HAVING clause serves to choose *whole groups*. Figure 8.6b illustrates the use of HAVING and displays the result of Q26.

(a)

FNAME	MINIT	LNAME	SSN	. . .	SALARY	SUPERSSN	DNO
John	B	Smith	123456789		30000	333445555	5
Franklin	T	Wong	333445555		40000	888665555	5
Ramesh	K	Narayan	666884444		38000	333445555	5
Joyce	A	English	453453453	. . .	25000	333445555	5
Alicia	J	Zelaya	999887777		25000	987654321	4
Jennifer	S	Wallace	987654321		43000	888665555	4
Ahmad	V	Jabbar	987987987		25000	987654321	4
James	E	Bong	888665555		55000	null	1

Grouping EMPLOYEE tuples by the value of DNO.

DNO	COUNT (*)	AVG (SALARY)
5	4	33250
4	3	31000
1	1	55000

Result of Q24.

(b)

PNAME	PNUMBER		ESSN	PNO	HOURS
ProductX	1		123456789	1	32.5
ProductX	1		453453453	1	20.0
ProductY	2		123456789	2	7.5
ProductY	2		453453453	2	20.0
ProductY	2		333445555	2	10.0
ProductZ	3		666884444	3	40.0
ProductZ	3		333445555	3	10.0
Computerization	10	. . .	333445555	10	10.0
Computerization	10		999887777	10	10.0
Computerization	10		987987987	10	35.0
Reorganization	20		333445555	20	10.0
Reorganization	20		987654321	20	15.0
Reorganization	20		888665555	20	null
Newbenefits	30		987987987	30	5.0
Newbenefits	30		987654321	30	20.0
Newbenefits	30		999887777	30	30.0

These groups are not selected by the HAVING condition of Q26.

After applying the WHERE clause but before applying HAVING.

PNAME	PNUMBER		ESSN	PNO	HOURS
ProductY	2		123456789	2	7.5
ProductY	2		453453453	2	20.0
ProductY	2		333445555	2	10.0
Computerization	10	. . .	333445555	10	10.0
Computerization	10		999887777	10	10.0
Computerization	10		987987987	10	35.0
Reorganization	20		333445555	20	10.0
Reorganization	20		987654321	20	15.0
Reorganization	20		888665555	20	null
Newbenefits	30		987987987	30	5.0
Newbenefits	30		987654321	30	20.0
Newbenefits	30		999887777	30	30.0

After applying the HAVING clause condition.

PNAME	COUNT (*)
ProductY	3
Computerization	3
Reorganization	3
Newbenefits	3

Result of Q26 (PNUMBER not shown).

FIGURE 8.6 Results of GROUP BY and HAVING. (a) Q24. (b) Q26.

QUERY 27

For each project, retrieve the project number, the project name, and the number of employees from department 5 who work on the project.

```
Q27: SELECT   PNUMBER, PNAME, COUNT (*)
     FROM     PROJECT, WORKS_ON, EMPLOYEE
     WHERE    PNUMBER=PNO AND SSN=ESSN AND DNO=5
     GROUP BY PNUMBER, PNAME;
```

Here we restrict the tuples in the relation (and hence the tuples in each group) to those that satisfy the condition specified in the WHERE clause—namely, that they work in department number 5. Notice that we must be extra careful when two different conditions apply (one to the function in the SELECT clause and another to the function in the HAVING clause). For example, suppose that we want to count the *total* number of employees whose salaries exceed $40,000 in each department, but only for departments where more than five employees work. Here, the condition (SALARY > 40000) applies only to the COUNT function in the SELECT clause. Suppose that we write the following *incorrect* query:

```
SELECT   DNAME, COUNT (*)
FROM     DEPARTMENT, EMPLOYEE
WHERE    DNUMBER=DNO AND SALARY>40000
GROUP BY DNAME
HAVING   COUNT (*) > 5;
```

This is incorrect because it will select only departments that have more than five employees *who each earn more than* $40,000. The rule is that the WHERE clause is executed first, to select individual tuples; the HAVING clause is applied later, to select individual groups of tuples. Hence, the tuples are already restricted to employees who earn more than $40,000, *before* the function in the HAVING clause is applied. One way to write this query correctly is to use a nested query, as shown in Query 28.

QUERY 28

For each department that has more than five employees, retrieve the department number and the number of its employees who are making more than $40,000.

```
Q28: SELECT   DNUMBER, COUNT (*)
     FROM     DEPARTMENT, EMPLOYEE
     WHERE    DNUMBER=DNO AND SALARY>40000 AND
              DNO IN   (SELECT   DNO
                        FROM     EMPLOYEE
                        GROUP BY DNO
                        HAVING   COUNT (*) > 5)
     GROUP BY DNUMBER;
```

8.5.9 Discussion and Summary of SQL Queries

A query in SQL can consist of up to six clauses, but only the first two—SELECT and FROM—are mandatory. The clauses are specified in the following order, with the clauses between square brackets [. . .] being optional:

SELECT <ATTRIBUTE AND FUNCTION LIST>

FROM <*TABLE LIST*>

[**WHERE** <CONDITION>]

[**GROUP BY** <GROUPING ATTRIBUTE(S)>]

[**HAVING** <*GROUP CONDITION*>]

[**ORDER BY** <ATTRIBUTE LIST>];

The SELECT clause lists the attributes or functions to be retrieved. The FROM clause specifies all relations (tables) needed in the query, including joined relations, but not those in nested queries. The WHERE clause specifies the conditions for selection of tuples from these relations, including join conditions if needed. GROUP BY specifies grouping attributes, whereas HAVING specifies a condition on the groups being selected rather than on the individual tuples. The built-in aggregate functions COUNT, SUM, MIN, MAX, and AVG are used in conjunction with grouping, but they can also be applied to all the selected tuples in a query without a GROUP BY clause. Finally, ORDER BY specifies an order for displaying the result of a query.

A query is evaluated *conceptually*[12] by first applying the FROM clause (to identify all tables involved in the query or to materialize any joined tables), followed by the WHERE clause, and then by GROUP BY and HAVING. Conceptually, ORDER BY is applied at the end to sort the query result. If none of the last three clauses (GROUP BY, HAVING, and ORDER BY) are specified, we can *think conceptually* of a query as being executed as follows: For *each combination of tuples*—one from each of the relations specified in the FROM clause—evaluate the WHERE clause; if it evaluates to TRUE, place the values of the attributes specified in the SELECT clause from this tuple combination in the result of the query. Of course, this is not an efficient way to implement the query in a real system, and each DBMS has special query optimization routines to decide on an execution plan that is efficient. We discuss query processing and optimization in Chapters 15 and 16.

In general, there are numerous ways to specify the same query in SQL. This flexibility in specifying queries has advantages and disadvantages. The main advantage is that users can choose the technique with which they are most comfortable when specifying a query. For example, many queries may be specified with join conditions in the WHERE clause, or by using joined relations in the FROM clause, or with some form of nested queries and the IN comparison operator. Some users may be more comfortable with one approach, whereas others may be more comfortable with another. From the programmer's and the

12. The actual order of query evaluation is implementation dependent; this is just a way to conceptually view a query in order to correctly formulate it.

system's point of view regarding query optimization, it is generally preferable to write a query with as little nesting and implied ordering as possible.

The disadvantage of having numerous ways of specifying the same query is that this may confuse the user, who may not know which technique to use to specify particular types of queries. Another problem is that it may be more efficient to execute a query specified in one way than the same query specified in an alternative way. Ideally, this should not be the case: The DBMS should process the same query in the same way regardless of how the query is specified. But this is quite difficult in practice, since each DBMS has different methods for processing queries specified in different ways. Thus, an additional burden on the user is to determine which of the alternative specifications is the most efficient. Ideally, the user should worry only about specifying the query correctly. It is the responsibility of the DBMS to execute the query efficiently. In practice, however, it helps if the user is aware of which types of constructs in a query are more expensive to process than others (see Chapter 16).

8.6 Insert, Delete, and Update Statements in SQL

In SQL, three commands can be used to modify the database: INSERT, DELETE, and UPDATE. We discuss each of these in turn.

8.6.1 The INSERT Command

In its simplest form, INSERT is used to add a single tuple to a relation. We must specify the relation name and a list of values for the tuple. The values should be listed *in the same order* in which the corresponding attributes were specified in the CREATE TABLE command. For example, to add a new tuple to the EMPLOYEE relation shown in Figure 5.5 and specified in the CREATE TABLE EMPLOYEE . . . command in Figure 8.1, we can use U1:

```
U1: INSERT INTO  EMPLOYEE
    VALUES       ('Richard', 'K', 'Marini', '653298653', '1962-12-30', '98
                 Oak Forest,Katy,TX', 'M', 37000, '987654321', 4);
```

A second form of the INSERT statement allows the user to specify explicit attribute names that correspond to the values provided in the INSERT command. This is useful if a relation has many attributes but only a few of those attributes are assigned values in the new tuple. However, the values must include all attributes with NOT NULL specification *and* no default value. Attributes with NULL allowed or DEFAULT values are the ones that can be *left out*. For example, to enter a tuple for a new EMPLOYEE for whom we know only the FNAME, LNAME, DNO, and SSN attributes, we can use U1A:

```
U1A: INSERT INTO  EMPLOYEE (FNAME, LNAME, DNO, SSN)
     VALUES       ('Richard', 'Marini', 4, '653298653');
```

Attributes not specified in U1A are set to their DEFAULT or to NULL, and the values are listed in the same order as the *attributes are listed in the* INSERT command itself. It is also possible to insert into a relation *multiple tuples* separated by commas in a single INSERT command. The attribute values forming *each tuple* are enclosed in parentheses.

A DBMS that fully implements SQL-99 should support and enforce all the integrity constraints that can be specified in the DDL. However, some DBMSs do not incorporate all the constraints, in order to maintain the efficiency of the DBMS and because of the complexity of enforcing all constraints. If a system does not support some constraint—say, referential integrity—the users or programmers must enforce the constraint. For example, if we issue the command in U2 on the database shown in Figure 5.6, a DBMS not supporting referential integrity will do the insertion even though no DEPARTMENT tuple exists in the database with DNUMBER = 2. It is the responsibility of the user to check that any such constraints *whose checks are not implemented by the* DBMS are not violated. However, the DBMS must implement checks to enforce all the SQL integrity constraints *it supports*. A DBMS enforcing NOT NULL will reject an INSERT command in which an attribute declared to be NOT NULL does not have a value; for example, U2A would be *rejected* because no SSN value is provided.

```
U2:  INSERT INTO   EMPLOYEE (FNAME, LNAME, SSN, DNO)
     VALUES        ('Robert', 'Hatcher', '980760540', 2);
     (* U2 is rejected if referential integrity checking is provided by dbms *)

U2A: INSERT INTO   EMPLOYEE (FNAME, LNAME, DNO)
     VALUES        ('Robert', 'Hatcher', 5);
     (* U2A is rejected if not null checking is provided by dbms *)
```

A variation of the INSERT command inserts multiple tuples into a relation in conjunction with creating the relation and loading it with the *result of a query*. For example, to create a temporary table that has the name, number of employees, and total salaries for each department, we can write the statements in U3A and U3B:

```
U3A: CREATE TABLE  DEPTS_INFO
     (DEPT_NAME    VARCHAR(15),
     NO_OF_EMPS    INTEGER,
     TOTAL_SAL     INTEGER);
U3B: INSERT INTO   DEPTS_INFO (DEPT_NAME, NO_OF_EMPS,
                   TOTAL_SAL)
     SELECT        DNAME, COUNT (*), SUM (SALARY)
     FROM          (DEPARTMENT JOIN EMPLOYEE ON
                   DNUMBER=DNO)
     GROUP BY      DNAME;
```

A table DEPTS_INFO is created by U3A and is loaded with the summary information retrieved from the database by the query in U3B. We can now query DEPTS_INFO as we

would any other relation; when we do not need it any more, we can remove it by using the DROP TABLE command. Notice that the DEPTS_INFO table may not be up to date; that is, if we update either the DEPARTMENT or the EMPLOYEE relations after issuing U3B, the information in DEPTS_INFO *becomes outdated*. We have to create a view (see Section 9.2) to keep such a table up to date.

8.6.2 The DELETE Command

The DELETE command removes tuples from a relation. It includes a WHERE clause, similar to that used in an SQL query, to select the tuples to be deleted. Tuples are explicitly deleted from only one table at a time. However, the deletion may propagate to tuples in other relations if *referential triggered actions* are specified in the referential integrity constraints of the DDL (see Section 8.2.2).[13] Depending on the number of tuples selected by the condition in the WHERE clause, zero, one, or several tuples can be deleted by a single DELETE command. A missing WHERE clause specifies that all tuples in the relation are to be deleted; however, the table remains in the database as an empty table.[14] The DELETE commands in U4A to U4D, if applied independently to the database of Figure 5.6, will delete zero, one, four, and all tuples, respectively, from the EMPLOYEE relation:

```
U4A: DELETE FROM  EMPLOYEE
     WHERE        LNAME='Brown';

U4B: DELETE FROM  EMPLOYEE
     WHERE        SSN='123456789';

U4C: DELETE FROM  EMPLOYEE
     WHERE        DNO IN (SELECT  DNUMBER
                          FROM    DEPARTMENT
                          WHERE   DNAME='Research');

U4D: DELETE FROM  EMPLOYEE;
```

8.6.3 The UPDATE Command

The UPDATE command is used to modify attribute values of one or more selected tuples. As in the DELETE command, a WHERE clause in the UPDATE command selects the tuples to be modified from a single relation. However, updating a primary key value may propagate to the foreign key values of tuples in other relations if such a *referential triggered action* is specified in the referential integrity constraints of the DDL (see Section 8.2.2). An additional SET clause in the UPDATE command specifies the attributes to be modified and

13. Other actions can be automatically applied through triggers (see Section 24.1) and other mechanisms.

14. We must use the DROP TABLE command to remove the table definition (see Section 8.3.1).

their new values. For example, to change the location and controlling department number of project number 10 to 'Bellaire' and 5, respectively, we use U5:

```
U5: UPDATE  PROJECT
    SET     PLOCATION = 'Bellaire', DNUM = 5
    WHERE   PNUMBER=10;
```

Several tuples can be modified with a single UPDATE command. An example is to give all employees in the 'Research' department a 10 percent raise in salary, as shown in U6. In this request, the modified SALARY value depends on the original SALARY value in each tuple, so two references to the SALARY attribute are needed. In the SET clause, the reference to the SALARY attribute on the right refers to the old SALARY value *before modification*, and the one on the left refers to the new SALARY value *after modification*:

```
U6: UPDATE  EMPLOYEE
    SET     SALARY = SALARY *1.1
    WHERE   DNO IN (SELECT    DNUMBER
                    FROM      DEPARTMENT
                    WHERE     DNAME='Research');
```

It is also possible to specify NULL or DEFAULT as the new attribute value. Notice that each UPDATE command explicitly refers to a single relation only. To modify multiple relations, we must issue several UPDATE commands.

8.7 Additional Features of SQL

SQL has a number of additional features that we have not described in this chapter but discuss elsewhere in the book. These are as follows:

- SQL has the capability to specify more general constraints, called assertions, using the CREATE ASSERTION statement. This is described in Section 9.1.
- SQL has language constructs for specifying views, also known as virtual tables, using the CREATE VIEW statement. Views are derived from the base tables declared through the CREATE TABLE statement, and are discussed in Section 9.2.
- SQL has several different techniques for writing programs in various programming languages that can include SQL statements to access one or more databases. These include embedded (and dynamic) SQL, SQL/CLI (Call Language Interface) and its predecessor ODBC (Open Data Base Connectivity), and SQL/PSM (Program Stored Modules). We discuss the differences among these techniques in Section 9.3, then discuss each technique in Sections 9.4 through 9.6. We also discuss how to access SQL databases through the Java programming language using JDBC and SQLJ.
- Each commercial RDBMS will have, in addition to the SQL commands, a set of commands for specifying physical database design parameters, file structures for relations, and access paths such as indexes. We called these commands a *storage definition lan-*

guage (SDL) in Chapter 2. Earlier versions of SQL had commands for **creating indexes,** but these were removed from the language because they were not at the conceptual schema level (see Chapter 2).

- SQL has transaction control commands. These are used to specify units of database processing for concurrency control and recovery purposes. We discuss these commands in Chapter 17 after we discuss the concept of transactions in more detail.
- SQL has language constructs for specifying the *granting and revoking of privileges* to users. Privileges typically correspond to the right to use certain SQL commands to access certain relations. Each relation is assigned an owner, and either the owner or the DBA staff can grant to selected users the privilege to use an SQL statement—such as SELECT, INSERT, DELETE, or UPDATE—to access the relation. In addition, the DBA staff can grant the privileges to create schemas, tables, or views to certain users. These SQL commands—called **GRANT** and **REVOKE**—are discussed in Chapter 23 where we discuss database security and authorization.
- SQL has language constructs for creating triggers. These are generally referred to as **active database** techniques, since they specify actions that are automatically triggered by events such as database updates. We discuss these features in Section 24.1, where we discuss active database concepts.
- SQL has incorporated many features from object-oriented models to have more powerful capabilities, leading to enhanced relational systems known as **object-relational.** Capabilities such as creating complex-structured attributes (also called **nested relations**), specifying abstract data types (called **UDTs** or user-defined types) for attributes and tables, creating **object identifiers** for referencing tuples, and specifying **operations** on types are discussed in Chapter 22.
- SQL and relational databases can interact with new technologies such as XML (eXtended Markup Language; see Chapter 26) and OLAP (On Line Analytical Processing for Data Warehouses; see Chapter 28).

8.8 SUMMARY

In this chapter we presented the SQL database language. This language or variations of it have been implemented as interfaces to many commercial relational DBMSs, including Oracle, IBM's DB2 and SQL/DS, Microsoft's SQL Server and ACCESS, INGRES, INFORMIX, and SYBASE. The original version of SQL was implemented in the experimental DBMS called SYSTEM R, which was developed at IBM Research. SQL is designed to be a comprehensive language that includes statements for data definition, queries, updates, view definition, and constraint specification. We discussed many of these in separate sections of this chapter. In the final section we discussed additional features that are described elsewhere in the book. Our emphasis was on the SQL-99 standard.

Table 8.2 summarizes the syntax (or structure) of various SQL statements. This summary is not meant to be comprehensive nor to describe every possible SQL construct; rather, it is meant to serve as a quick reference to the major types of constructs available

TABLE 8.2 SUMMARY OF SQL SYNTAX

```
CREATE TABLE <table name> (<column name> <column type> [<attribute constraint>]
                          {, <column name> <column type> [<attribute constraint>] }
                          [<table constraint> {,<table constraint>}])

DROP TABLE <table name>

ALTER TABLE <table name> ADD <column name> <column type>

SELECT [DISTINCT] <attribute list>
FROM (<table name> { <alias>} | <joined table>) {, (<table name> { <alias>} | <joined table>) }
[WHERE <condition>]
[GROUP BY <grouping attributes> [HAVING <group selection condition> ] ]
[ORDER BY <column name> [<order>] {, <column name> [<order>] } ]

<attribute list>::= (* | ( <column name> | <function>(([DISTINCT]<column name> | *)))
                    {,( <column name> | <function>(([DISTINCT] <column name> | *)) } ) )
<grouping attributes>::= <column name> { , <column name>}
<order>::= (ASC | DESC)

INSERT INTO <table name> [( <column name>{, <column name>} ) ]
(VALUES ( <constant value> , { <constant value>} ){,(<constant value>{,<constant value>})}
| <select statement>)

DELETE FROM <table name>
[WHERE <selection condition>]

UPDATE <table name>
SET <column name>=<value expression> { , <column name>=<value expression> }
[WHERE <selection condition>]

CREATE [UNIQUE] INDEX <index name>
ON <table name> ( <column name> [ <order> ] { , <column name> [ <order> ] } )
[CLUSTER]

DROP INDEX <index name>

CREATE VIEW <view name> [ ( <column name> { , <column name> } ) ]
AS <select statement>

DROP VIEW <view name>
```

*The last two commands are not part of standard SQL2.

in SQL. We use BNF notation, where nonterminal symbols are shown in angled brackets <...>, optional parts are shown in square brackets [...], repetitions are shown in braces {...}, and alternatives are shown in parentheses (... | ... | ...).[15]

Review Questions

8.1. How do the relations (tables) in SQL differ from the relations defined formally in Chapter 5? Discuss the other differences in terminology. Why does SQL allow duplicate tuples in a table or in a query result?

8.2. List the data types that are allowed for SQL attributes.

8.3. How does SQL allow implementation of the entity integrity and referential integrity constraints described in Chapter 5? What about referential triggered actions?

8.4. Describe the six clauses in the syntax of an SQL query, and show what type of constructs can be specified in each of the six clauses. Which of the six clauses are required and which are optional?

8.5. Describe conceptually how an SQL query will be executed by specifying the conceptual order of executing each of the six clauses.

8.6. Discuss how NULLs are treated in comparison operators in SQL. How are NULLs treated when aggregate functions are applied in an SQL query? How are NULLs treated if they exist in grouping attributes?

Exercises

8.7. Consider the database shown in Figure 1.2, whose schema is shown in Figure 2.1. What are the referential integrity constraints that should hold on the schema? Write appropriate SQL DDL statements to define the database.

8.8. Repeat Exercise 8.7, but use the AIRLINE database schema of Figure 5.8.

8.9. Consider the LIBRARY relational database schema of Figure 6.12. Choose the appropriate action (reject, cascade, set to null, set to default) for each referential integrity constraint, both for *deletion* of a referenced tuple, and for *update* of a primary key attribute value in a referenced tuple. Justify your choices.

8.10. Write appropriate SQL DDL statements for declaring the LIBRARY relational database schema of Figure 6.12. Specify appropriate keys and referential triggered actions.

8.11. Write SQL queries for the LIBRARY database queries given in Exercise 6.18.

8.12. How can the key and foreign key constraints be enforced by the DBMS? Is the enforcement technique you suggest difficult to implement? Can the constraint checks be executed efficiently when updates are applied to the database?

8.13. Specify the queries of Exercise 6.16 in SQL. Show the result of each query if it is applied to the COMPANY database of Figure 5.6.

8.14. Specify the following additional queries on the database of Figure 5.5 in SQL. Show the query results if each query is applied to the database of Figure 5.6.

15. The full syntax of SQL-99 is described in many voluminous documents of hundreds of pages.

a. For each department whose average employee salary is more than $30,000, retrieve the department name and the number of employees working for that department.
b. Suppose that we want the number of *male* employees in each department rather than all employees (as in Exercise 8.14a). Can we specify this query in SQL? Why or why not?

8.15. Specify the updates of Exercise 5.10, using the SQL update commands.

8.16. Specify the following queries in SQL on the database schema of Figure 1.2.
a. Retrieve the names of all senior students majoring in 'CS' (computer science).
b. Retrieve the names of all courses taught by Professor King in 1998 and 1999.
c. For each section taught by Professor King, retrieve the course number, semester, year, and number of students who took the section.
d. Retrieve the name and transcript of each senior student (Class = 5) majoring in CS. A transcript includes course name, course number, credit hours, semester, year, and grade for each course completed by the student.
e. Retrieve the names and major departments of all straight-A students (students who have a grade of A in all their courses).
f. Retrieve the names and major departments of all students who do not have a grade of A in any of their courses.

8.17. Write SQL update statements to do the following on the database schema shown in Figure 1.2.
a. Insert a new student, <'Johnson', 25, 1, 'MATH'>, in the database.
b. Change the class of student 'Smith' to 2.
c. Insert a new course, <'Knowledge Engineering', 'CS4390', 3, 'CS'>.
d. Delete the record for the student whose name is 'Smith' and whose student number is 17.

8.18. Specify the queries and updates of Exercises 6.17 and 5.11, which refer to the AIRLINE database (see Figure 5.8), in SQL.

8.19. a. Design a relational database schema for your database application.
b. Declare your relations, using the SQL DDL.
c. Specify a number of queries in SQL that are needed by your database application.
d. Based on your expected use of the database, choose some attributes that should have indexes specified on them.
e. Implement your database, if you have a DBMS that supports SQL.

8.20. Specify the answers to Exercises 6.19 through 6.21 and Exercise 6.23 in SQL.

Selected Bibliography

The SQL language, originally named SEQUEL, was based on the language SQUARE (Specifying Queries as Relational Expressions), described by Boyce et al. (1975). The syntax of SQUARE was modified into SEQUEL (Chamberlin and Boyce 1974) and then into SEQUEL 2 (Chamberlin et al. 1976), on which SQL is based. The original implementation of SEQUEL was done at IBM Research, San Jose, California.

Reisner (1977) describes a human factors evaluation of SEQUEL in which she found that users have some difficulty with specifying join conditions and grouping correctly.

Date (1984b) contains a critique of the SQL language that points out its strengths and shortcomings. Date and Darwen (1993) describes SQL2. ANSI (1986) outlines the original SQL standard, and ANSI (1992) describes the SQL2 standard. Various vendor manuals describe the characteristics of SQL as implemented on DB2, SQL/DS, Oracle, INGRES, INFORMIX, and other commercial DBMS products. Melton and Simon (1993) is a comprehensive treatment of SQL2. Horowitz (1992) discusses some of the problems related to referential integrity and propagation of updates in SQL2.

The question of view updates is addressed by Dayal and Bernstein (1978), Keller (1982), and Langerak (1990), among others. View implementation is discussed in Blakeley et al. (1989). Negri et al. (1991) describes formal semantics of SQL queries.

9

More SQL: Assertions, Views, and Programming Techniques

In the previous chapter, we described several aspects of the SQL language, the standard for relational databases. We described the SQL statements for data definition, schema modification, queries, and updates. We also described how common constraints such as key and referential integrity are specified. In this chapter, we present several additional aspects of SQL. We start in Section 9.1 by describing the CREATE ASSERTION statement, which allows the specification of more general constraints on the database. Then, in Section 9.2, we describe the SQL facilities for defining views on the database. Views are also called *virtual* or *derived tables* because they present the user with what appear to be tables; however, the information in those tables is derived from previously defined tables.

The next several sections of this chapter discuss various techniques for accessing databases from programs. Most database access in practical situations is through software programs that implement **database applications.** This software is usually developed in a general-purpose programming language such as JAVA, COBOL, or C/C++. Recall from Section 2.3.1 that when database statements are included in a program, the general-purpose programming language is called the *host language*, whereas the database language—SQL, in our case—is called the *data sublanguage*. In some cases, special *database programming languages* are developed specifically for writing database applications. Although many of these were developed as research prototypes, some notable database programming languages have widespread use, such as ORACLE's PL/SQL (Programming Language/SQL).

We start our presentation of database programming in Section 9.3 with an overview of the different techniques developed for accessing a database from programs. Then, in Section 9.4, we discuss the rules for embedding SQL statements into a general-purpose programming language, generally known as *embedded SQL*. This section also briefly discusses *dynamic SQL*, in which queries can be dynamically constructed at runtime, and presents the basics of the SQLJ variation of embedded SQL that was developed specifically for the programming language JAVA. In Section 9.5, we discuss the technique known as SQL/CLI (Call Level Interface), in which a library of procedures and functions is provided for accessing the database. Various sets of library functions have been proposed. The SQL/CLI set of functions is the one given in the SQL standard. Another library of functions is ODBC (Open Data Base Connectivity). We do not describe ODBC because it is considered to be the predecessor to SQL/CLI. A third library of functions—which we do describe—is JDBC; this was developed specifically for accessing databases from JAVA. Finally, in Section 9.6, we discuss SQL/PSM (Persistent Stored Modules), which is a part of the SQL standard that allows program modules—procedures and functions—to be stored by the DBMS and accessed through SQL. Section 9.7 summarizes the chapter.

9.1 Specifying General Constraints as Assertions

In SQL, users can specify general constraints—those that do not fall into any of the categories described in Section 8.2—via **declarative assertions,** using the CREATE ASSERTION statement of the DDL. Each assertion is given a constraint name and is specified via a condition similar to the WHERE clause of an SQL query. For example, to specify the constraint that "the salary of an employee must not be greater than the salary of the manager of the department that the employee works for" in SQL, we can write the following assertion:

```
CREATE ASSERTION SALARY_CONSTRAINT
CHECK ( NOT EXISTS
        (SELECT *
        FROM    EMPLOYEE E, EMPLOYEE M, DEPARTMENT D
        WHERE   E.SALARY>M.SALARY AND
                E.DNO=D.DNUMBER AND
                D.MGRSSN=M.SSN) );
```

The constraint name SALARY_CONSTRAINT is followed by the keyword CHECK, which is followed by a **condition** in parentheses that must hold true on every database state for the assertion to be satisfied. The constraint name can be used later to refer to the constraint or to modify or drop it. The DBMS is responsible for ensuring that the condition is not violated. Any WHERE clause condition can be used, but many constraints can be specified using the EXISTS and NOT EXISTS style of SQL conditions. Whenever some tuples in the database cause the condition of an ASSERTION statement to evaluate to FALSE, the constraint is **violated.** The constraint is **satisfied** by a database state if *no combination of tuples* in that database state violates the constraint.

The basic technique for writing such assertions is to specify a query that selects any tuples *that violate the desired condition.* By including this query inside a NOT EXISTS clause, the assertion will specify that the result of this query must be empty. Thus, the assertion is violated if the result of the query is not empty. In our example, the query selects all employees whose salaries are greater than the salary of the manager of their department. If the result of the query is not empty, the assertion is violated.

Note that the CHECK clause and constraint condition can also be used to specify constraints on attributes and domains (see Section 8.2.1) and on tuples (see Section 8.2.4). A major difference between CREATE ASSERTION and the other two is that the CHECK clauses on attributes, domains, and tuples are checked in SQL *only when tuples are inserted or updated.* Hence, constraint checking can be implemented more efficiently by the DBMS in these cases. The schema designer should use CHECK on attributes, domains, and tuples only when he or she is sure that the constraint can *only be violated by insertion or updating of tuples.* On the other hand, the schema designer should use CREATE ASSERTION only in cases where it is not possible to use CHECK on attributes, domains, or tuples, so that checks are implemented more efficiently by the DBMS.

Another statement related to CREATE ASSERTION in SQL is CREATE TRIGGER, but triggers are used in a different way. In many cases it is convenient to specify the type of action to be taken when certain events occur and when certain conditions are satisfied. Rather than offering users only the option of aborting an operation that causes a violation—as with CREATE ASSERTION—the DBMS should make other options available. For example, it may be useful to specify a condition that, if violated, causes some user to be informed of the violation. A manager may want to be informed if an employee's travel expenses exceed a certain limit by receiving a message whenever this occurs. The action that the DBMS must take in this case is to send an appropriate message to that user. The condition is thus used to **monitor** the database. Other actions may be specified, such as executing a specific *stored procedure* or triggering other updates. The CREATE TRIGGER statement is used to implement such actions in SQL. A **trigger** specifies an **event** (such as a particular database update operation), a **condition,** and an **action.** The action is to be executed automatically if the condition is satisfied when the event occurs. We discuss triggers in detail in Section 24.1 when we describe *active databases.*

9.2 Views (Virtual Tables) in SQL

In this section we introduce the concept of a view in SQL. We then show how views are specified, and we discuss the problem of updating a view, and how a view can be implemented by the DBMS.

9.2.1 Concept of a View in SQL

A **view** in SQL terminology is a single table that is derived from other tables.[1] These other tables could be base tables or previously defined views. A view does not necessarily exist in

1. As used in SQL, the term *view* is more limited than the term *user view* discussed in Chapters 1 and 2, since a user view would possibly include many relations.

physical form; it is considered a **virtual table,** in contrast to base tables, whose tuples are actually stored in the database. This limits the possible update operations that can be applied to views, but it does not provide any limitations on querying a view.

We can think of a view as a way of specifying a table that we need to reference frequently, even though it may not exist physically. For example, in Figure 5.5 we may frequently issue queries that retrieve the employee name and the project names that the employee works on. Rather than having to specify the join of the EMPLOYEE, WORKS_ON, and PROJECT tables every time we issue that query, we can define a view that is a result of these joins. We can then issue queries on the view, which are specified as single-table retrievals rather than as retrievals involving two joins on three tables. We call the EMPLOYEE, WORKS_ON, and PROJECT tables the **defining tables** of the view.

9.2.2 Specification of Views in SQL

In SQL, the command to specify a view is CREATE VIEW. The view is given a (virtual) table name (or view name), a list of attribute names, and a query to specify the contents of the view. If none of the view attributes results from applying functions or arithmetic operations, we do not have to specify attribute names for the view, since they would be the same as the names of the attributes of the defining tables in the default case. The views in V1 and V2 create virtual tables whose schemas are illustrated in Figure 9.1 when applied to the database schema of Figure 5.5.

```
V1: CREATE VIEW    WORKS_ON1
    AS SELECT      FNAME, LNAME, PNAME, HOURS
       FROM        EMPLOYEE, PROJECT, WORKS_ON
       WHERE       SSN=ESSN AND PNO=PNUMBER;
V2: CREATE VIEW    DEPT_INFO(DEPT_NAME,NO_OF_EMPS,TOTAL_SAL)
    AS SELECT      DNAME, COUNT (*), SUM (SALARY)
       FROM        DEPARTMENT, EMPLOYEE
       WHERE       DNUMBER=DNO
       GROUP BY    DNAME;
```

WORKS_ON1

FNAME	LNAME	PNAME	HOURS

DEPT_INFO

DEPT_NAME	NO_OF_EMPS	TOTAL_SAL

FIGURE 9.1 Two views specified on the database schema of Figure 5.5.

In V1, we did not specify any new attribute names for the view WORKS_ON1 (although we could have); in this case, WORKS_ON1 *inherits* the names of the view attributes from the defining tables EMPLOYEE, PROJECT, and WORKS_ON. View V2 explicitly specifies new attribute names for the view DEPT_INFO, using a one-to-one correspondence between the attributes specified in the CREATE VIEW clause and those specified in the SELECT clause of the query that defines the view.

We can now specify SQL queries on a view—or virtual table—in the same way we specify queries involving base tables. For example, to retrieve the last name and first name of all employees who work on 'ProjectX', we can utilize the WORKS_ON1 view and specify the query as in QV1:

```
QV1: SELECT  FNAME, LNAME
     FROM    WORKS_ON1
     WHERE   PNAME='ProjectX';
```

The same query would require the specification of two joins if specified on the base relations; one of the main advantages of a view is to simplify the specification of certain queries. Views are also used as a security and authorization mechanism (see Chapter 23).

A view is supposed to be *always up to date*; if we modify the tuples in the base tables on which the view is defined, the view must automatically reflect these changes. Hence, the view is not realized at the time of *view definition* but rather at the time we *specify a query* on the view. It is the responsibility of the DBMS and not the user to make sure that the view is up to date.

If we do not need a view any more, we can use the **DROP VIEW** command to dispose of it. For example, to get rid of the view V1, we can use the SQL statement in V1A:

```
V1A: DROP VIEW  WORKS_ON1;
```

9.2.3 View Implementation and View Update

The problem of efficiently implementing a view for querying is complex. Two main approaches have been suggested. One strategy, called **query modification,** involves modifying the view query into a query on the underlying base tables. For example, the query QV1 would be automatically modified to the following query by the DBMS:

```
SELECT  FNAME, LNAME
FROM    EMPLOYEE, PROJECT, WORKS_ON
WHERE   SSN=ESSN AND PNO=PNUMBER
        AND PNAME='ProjectX';
```

The disadvantage of this approach is that it is inefficient for views defined via complex queries that are time-consuming to execute, especially if multiple queries are applied to the view within a short period of time. The other strategy, called **view materialization,** involves physically creating a temporary view table when the view is first queried and keeping that table on the assumption that other queries on the view will follow. In this case, an efficient strategy for automatically updating the view table when

the base tables are updated must be developed in order to keep the view up to date. Techniques using the concept of **incremental update** have been developed for this purpose, where it is determined what new tuples must be inserted, deleted, or modified in a materialized view table when a change is applied to one of the defining base tables. The view is generally kept as long as it is being queried. If the view is not queried for a certain period of time, the system may then automatically remove the physical view table and recompute it from scratch when future queries reference the view.

Updating of views is complicated and can be ambiguous. In general, an update on a view defined on a *single table* without any *aggregate functions* can be mapped to an update on the underlying base table under certain conditions. For a view involving joins, an update operation may be mapped to update operations on the underlying base relations in *multiple ways*. To illustrate potential problems with updating a view defined on multiple tables, consider the WORKS_ON1 view, and suppose that we issue the command to update the PNAME attribute of 'John Smith' from 'ProductX' to 'ProductY'. This view update is shown in UV1:

```
UV1: UPDATE  WORKS_ON1
     SET     PNAME = 'ProductY'
     WHERE   LNAME='Smith' AND FNAME='John' AND
             PNAME='ProductX';
```

This query can be mapped into several updates on the base relations to give the desired update effect on the view. Two possible updates, (a) and (b), on the base relations corresponding to UV1 are shown here:

```
(a): UPDATE  WORKS_ON
     SET     PNO =    (SELECT   PNUMBER
                       FROM     PROJECT
                       WHERE    PNAME='ProductY')
     WHERE   ESSN IN  (SELECT   SSN
                       FROM     EMPLOYEE
                       WHERE    LNAME='Smith' AND FNAME='John')
             AND
             PNO =    (SELECT   PNUMBER
                       FROM     PROJECT
                       WHERE    PNAME='ProductX');

(b): UPDATE  PROJECT SET  PNAME = 'ProductY'
     WHERE   PNAME = 'ProductX';
```

Update (a) relates 'John Smith' to the 'ProductY' PROJECT tuple in place of the 'ProductX' PROJECT tuple and is the most likely desired update. However, (b) would also give the desired update effect on the view, but it accomplishes this by changing the name of the 'ProductX' tuple in the PROJECT relation to 'ProductY'. It is quite unlikely that the

user who specified the view update UV1 wants the update to be interpreted as in (b), since it also has the side effect of changing all the view tuples with PNAME = 'ProductX'.

Some view updates may not make much sense; for example, modifying the TOTAL_SAL attribute of the DEPT_INFO view does not make sense because TOTAL_SAL is defined to be the sum of the individual employee salaries. This request is shown as UV2:

```
UV2: UPDATE  DEPT_INFO
     SET     TOTAL_SAL=100000
     WHERE   DNAME='Research';
```

A large number of updates on the underlying base relations can satisfy this view update.

A view update is feasible when only *one possible update* on the base relations can accomplish the desired update effect on the view. Whenever an update on the view can be mapped to *more than one update* on the underlying base relations, we must have a certain procedure for choosing the desired update. Some researchers have developed methods for choosing the most likely update, while other researchers prefer to have the user choose the desired update mapping during view definition.

In summary, we can make the following observations:

- A view with a single defining table is updatable if the view attributes contain the primary key of the base relation, as well as all attributes with the NOT NULL constraint *that do not have* default values specified.
- Views defined on multiple tables using joins are generally not updatable.
- Views defined using grouping and aggregate functions are not updatable.

In SQL, the clause **WITH CHECK OPTION** must be added at the end of the view definition if a view *is to be updated*. This allows the system to check for view updatability and to plan an execution strategy for view updates.

9.3 Database Programming: Issues and Techniques

We now turn our attention to the techniques that have been developed for accessing databases from programs and, in particular, to the issue of how to access SQL databases from application programs. Our presentation of SQL so far has focused on the language constructs for various database operations—from schema definition and constraint specification to querying, to updating, and the specification of views. Most database systems have an **interactive interface** where these SQL commands can be typed directly into a monitor and input to the database system. For example, in a computer system where the ORACLE RDBMS is installed, the command SQLPLUS will start the interactive interface. The user can type SQL commands or queries directly over several lines, ended by a semicolon and the Enter key (that is, "; <cr>"). Alternatively, a **file of commands** can be created and executed through the interactive interface by typing @<filename>. The system will execute the commands written in the file and display the results, if any.

The interactive interface is quite convenient for schema and constraint creation or for occasional ad hoc queries. However, the majority of database interactions in practice are executed through programs that have been carefully designed and tested. These programs are generally known as **application programs** or **database applications,** and are used as *canned transactions* by the end users, as discussed in Section 1.4.3. Another very common use of database programming is to access a database through an application program that implements a **Web interface,** for example, for making airline reservations or department store purchases. In fact, the vast majority of Web electronic commerce applications include some database access commands.

In this section, we first give an overview of the main approaches to database programming. Then we discuss some of the problems that occur when trying to access a database from a general-purpose programming language, and discuss the typical sequence of commands for interacting with a database from a software program.

9.3.1 Approaches to Database Programming

Several techniques exist for including database interactions in application programs. The main approaches for database programming are the following:

1. *Embedding database commands in a general-purpose programming language:* In this approach, database statements are **embedded** into the host programming language, but they are identified by a special prefix. For example, the prefix for embedded SQL is the string EXEC SQL, which precedes all SQL commands in a host language program.[2] A **precompiler** or **preproccessor** first scans the source program code to identify database statements and extract them for processing by the DBMS. They are replaced in the program by function calls to the DBMS-generated code.
2. *Using a library of database functions:* A **library of functions** is made available to the host programming language for database calls. For example, there could be functions to connect to a database, execute a query, execute an update, and so on. The actual database query and update commands, and any other necessary information, are included as parameters in the function calls. This approach provides what is known as an **Application Programming Interface** (API) for accessing a database from application programs.
3. *Designing a brand-new language:* A **database programming language** is designed from scratch to be compatible with the database model and query language. Additional programming structures such as loops and conditional statements are added to the database language to convert it into a full-fledged programming language.

In practice, the first two approaches are more common, since many applications are already written in general-purpose programming languages but require some database access. The third approach is more appropriate for applications that have intensive database interaction. One of the main problems with the first two approaches is *impedance mismatch*, which does not occur in the third approach. We discuss this next.

2. Other prefixes are sometimes used, but this is the most common one.

9.3.2 Impedance Mismatch

Impedance mismatch is the term used to refer to the problems that occur because of differences between the database model and the programming language model. For example, the practical relational model has three main constructs: attributes and their data types, tuples (records), and tables (sets or multisets of records). The first problem that may occur is that the data types of the programming language differ from the attribute data types in the data model. Hence, it is necessary to have a binding for each host programming language that specifies for each attribute type the compatible programming language types. It is necessary to have a **binding** for each programming language because different languages have different data types; for example, the data types available in C and JAVA are different, and both differ from the SQL data types.

Another problem occurs because the results of most queries are sets or multisets of tuples, and each tuple is formed of a sequence of attribute values. In the program, it is often necssary to access the individual data values within individual tuples for printing or processing. Hence, a binding is needed to map the *query result data structure*, which is a table, to an appropriate data structure in the programming language. A mechanism is needed to loop over the tuples in a query result in order to access a single tuple at a time and to extract individual values from the tuple. A **cursor** or **iterator variable** is used to loop over the tuples in a query result. Individual values within each tuple are typically extracted into distinct program variables of the appropriate type.

Impedance mismatch is less of a problem when a special database programming language is designed that uses the same data model and data types as the database model. One example of such a language is ORACLE's PL/SQL. For object databases, the object data model (see Chapter 20) is quite similar to the data model of the JAVA programming language, so the impedance mismatch is greatly reduced when JAVA is used as the host language for accessing a JAVA-compatible object database. Several database programming languages have been implemented as research prototypes (see bibliographic notes).

9.3.3 Typical Sequence of Interaction in Database Programming

When a programmer or software engineer writes a program that requires access to a database, it is quite common for the program to be running on one computer system while the database is installed on another. Recall from Section 2.5 that a common architecture for database access is the client/server model, where a **client program** handles the logic of a software application, but includes some calls to one or more **database servers** to access or update the data.[3] When writing such a program, a common sequence of interaction is the following:

1. When the client program requires access to a particular database, the program must first *establish* or *open* a **connection** to the database server. Typically, this

3. As we discussed in Section 2.5, there are two-tier and three-tier architectures; to keep our discussion simple, we will assume a two-tier client/server architecture here. We discuss additional variations of these architectures in Chapter 25.

involves specifying the Internet address (URL) of the machine where the database server is located, plus providing a login account name and password for database access.

2. Once the connection is established, the program can interact with the database by submitting queries, updates, and other database commands. In general, most types of SQL statements can be included in an application program.
3. When the program no longer needs access to a particular database, it should *terminate* or *close* the connection to the database.

A program can access multiple databases if needed. In some database programming approaches, only one connection can be active at a time, whereas in other approaches multiple connections can be established at the same time.

In the next three sections, we discuss examples of each of the three approaches to database programming. Section 9.4 describes how SQL is *embedded* into a programming language. Section 9.5 discusses how *function calls* are used to access the database, and Section 9.6 discusses an extension to SQL called SQL/PSM that allows *general-purpose programming* constructs for defining modules (procedures and functions) that are stored within the database system.[4]

9.4 Embedded SQL, Dynamic SQL, and SQLJ

9.4.1 Retrieving Single Tuples with Embedded SQL

In this section, we give an overview of how SQL statements can be embedded in a general-purpose programming language such as C, ADA, COBOL, or PASCAL. The programming language is called the **host language.** Most SQL statements—including data or constraint definitions, queries, updates, or view definitions—can be embedded in a host language program. An embedded SQL statement is distinguished from programming language statements by prefixing it with the keywords EXEC SQL so that a **preprocessor** (or **precompiler**) can separate embedded SQL statements from the host language code. The SQL statements can be terminated by a semicolon (;) or a matching END-EXEC.

To illustrate the concepts of embedded SQL, we will use C as the host programming language. Within an embedded SQL command, we may refer to specially declared C program variables. These are called **shared variables** because they are used in both the C program and the embedded SQL statements. Shared variables are prefixed by a colon (:) *when they appear in an SQL statement.* This distinguishes program variable names from the names of database schema constructs such as attributes and relations. It also allows program variables to have the same names as attribute names, since they are distinguishable by the ":" prefix in the SQL statement.

4. Although SQL/PSM is not considered to be a full-fledged programming language, it illustrates how typical general-purpose programming constructs—such as loops and conditional structures—can be incorporated into SQL.

Names of database schema constructs—such as attributes and relations—can only be used within the SQL commands, but shared program variables can be used elsewhere in the C program without the ":" prefix.

Suppose that we want to write C programs to process the COMPANY database of Figure 5.5. We need to declare program variables to match the types of the database attributes that the program will process. The programmer can choose the names of the program variables; they may or may not have names that are identical to their corresponding attributes. We will use the C program variables declared in Figure 9.2 for all our examples, and we will show C program segments without variable declarations. Shared variables are declared within a **declare section** in the program, as shown in Figure 9.2 (lines 1 through 7).[5] A few of the common bindings of C types to SQL types are as follows. The SQL types INTEGER, SMALLINT, REAL, and DOUBLE are mapped to the C types `long`, `short`, `float`, and `double`, respectively. Fixed-length and varying-length strings (CHAR[*i*], VARCHAR[*i*]) in SQL can be mapped to arrays of characters (`char` [*i*+1], `varchar` [*i*+1]) in C that are one character longer than the SQL type, because strings in C are terminated by a "\0" (null) character, which is not part of the character string itself.[6]

Notice that the only embedded SQL commands in Figure 9.2 are lines 1 and 7, which tell the precompiler to take note of the C variable names between BEGIN DECLARE and END DECLARE because they can be included in embedded SQL statements—as long as they are preceded by a colon (:). Lines 2 through 5 are regular C program declarations. The C program variables declared in lines 2 through 5 correspond to the attributes of the EMPLOYEE and DEPARTMENT tables from the COMPANY database of Figure 5.5 that was declared by the SQL DDL in Figure 8.1. The variables declared in line 6—SQLCODE and SQLSTATE—are used to communicate errors and exception conditions between the database system and the program. Line 0 shows a program variable `loop` that will not be used in any embedded SQL statements, so it is declared outside the SQL declare section.

```
0)  int loop ;
1)  EXEC SQL BEGIN DECLARE SECTION ;
2)  varchar dname [16], fname [16], lname [16], address [31] ;
3)  char ssn [10], bdate [11], sex [2], minit [2] ;
4)  float salary, raise ;
5)  int dno, dnumber ;
6)  int SQLCODE ; char SQLSTATE [6] ;
7)  EXEC SQL END DECLARE SECTION ;
```

FIGURE 9.2 C program variables used in the embedded SQL examples E1 and E2.

5. We use line numbers in our code segments for easy reference; these numbers are not part of the actual code.

6. SQL strings can also be mapped to `char*` types in C.

Connecting to the Database. The SQL command for establishing a connection to a database has the following form:

CONNECT TO <server name> **AS** <connection name>
AUTHORIZATION <user account name and password> ;

In general, since a user or program can access several database servers, several connections can be established, but only one connection can be active at any point in time. The programmer or user can use the <connection name> to change from the currently active connection to a different one by using the following command:

SET CONNECTION <connection name> ;

Once a connction is no longer needed, it can be terminated by the following command:

DISCONNECT <connection name> ;

In the examples in this chapter, we assume that the appropriate connection has already been established to the COMPANY database, and that it is the currently active connection.

Communicating between the Program and the DBMS Using SQLCODE and SQLSTATE. The two special **communication variables** that are used by the DBMS to communicate exception or error conditions to the program are SQLCODE and SQLSTATE. The **SQLCODE** variable shown in Figure 9.2 is an integer variable. After each database command is executed, the DBMS returns a value in SQLCODE. A value of 0 indicates that the statement was executed successfully by the DBMS. If SQLCODE > 0 (or, more specifically, if SQLCODE = 100), this indicates that no more data (records) are available in a query result. If SQLCODE < 0, this indicates some error has occurred. In some systems—for example, in the ORACLE RDBMS—SQLCODE is a field in a record structure called SQLCA (SQL communication area), so it is referenced as SQLCA.SQLCODE. In this case, the definition of SQLCA must be included in the C program by including the following line:

```
EXEC SQL include SQLCA ;
```

In later versions of the SQL standard, a communication variable called **SQLSTATE** was added, which is a string of five characters. A value of "00000" in SQLSTATE indicates no error or exception; other values indicate various errors or exceptions. For example, "02000" indicates "no more data" when using SQLSTATE. Currently, both SQLSTATE and SQLCODE are available in the SQL standard. Many of the error and exception codes returned in SQLSTATE are supposed to be standardized for all SQL vendors and platforms,[7] whereas the codes returned in SQLCODE are not standardized but are defined by the DBMS vendor. Hence, it is generally better to use SQLSTATE, because this makes error handling in the application programs independent of a particular DBMS. As an exercise, the reader should rewrite the examples given later in this chapter using SQLSTATE instead of SQLCODE.

7. In particular, SQLSTATE codes starting with the characters 0 through 4 or A through H are supposed to be standardized, whereas other values can be implementation-defined.

Example of Embedded SQL Programming. Our first example to illustrate embedded SQL programming is a repeating program segment (loop) that reads a social security number of an employee and prints out some information from the corresponding EMPLOYEE record in the database. The C program code is shown as program segment E1 in Figure 9.3. The program reads (inputs) a social security number value and then retrieves the EMPLOYEE tuple with that social security number from the database via the embedded SQL command. The **INTO clause** (line 5) specifies the program variables into which attribute values from the database are retrieved. C program variables in the INTO clause are prefixed with a colon (:), as we discussed earlier.

Line 7 in E1 illustrates the communication between the database and the program through the special variable SQLCODE. If the value returned by the DBMS in SQLCODE is 0, the previous statement was executed without errors or exception conditions. Line 7 checks this and assumes that if an error occurred, it was because no EMPLOYEE tuple existed with the given social security number; it therefore outputs a message to that effect (line 8).

In E1 a *single tuple* is selected by the embedded SQL query; that is why we are able to assign its attribute values directly to C program variables in the INTO clause in line 5. In general, an SQL query can retrieve many tuples. In that case, the C program will typically go through the retrieved tuples and process them one at a time. A *cursor* is used to allow tuple-at-a-time processing by the host language program. We describe cursors next.

9.4.2 Retrieving Multiple Tuples with Embedded SQL Using Cursors

We can think of a **cursor** as a pointer that points to a *single tuple* (*row*) from the result of a query that retrieves multiple tuples. The cursor is declared when the SQL query command is declared in the program. Later in the program, an **OPEN CURSOR** command fetches the query result from the database and sets the cursor to a position *before the first row* in the

```
//Program Segment E1:
0)  loop = 1 ;
1)  while (loop) {
2)    prompt("Enter a Social Security Number: ", ssn) ;
3)    EXEC SQL
4)      select FNAME, MINIT, LNAME, ADDRESS, SALARY
5)      into :fname, :minit, :lname, :address, :salary
6)      from EMPLOYEE where SSN = :ssn ;
7)    if (SQLCODE == 0) printf(fname, minit, lname, address, salary)
8)      else printf("Social Security Number does not exist: ", ssn) ;
9)    prompt("More Social Security Numbers (enter 1 for Yes, 0 for No): ", loop) ;
10)   }
```

FIGURE 9.3 Program segment E1, a C program segment with embedded SQL.

result of the query. This becomes the **current row** for the cursor. Subsequently, FETCH commands are issued in the program; each FETCH moves the cursor to the *next row* in the result of the query, making it the current row and copying its attribute values into the C (host language) program variables specified in the FETCH command by an INTO clause. The cursor variable is basically an **iterator** that iterates (loops) over the tuples in the query result—one tuple at a time. This is similar to traditional record-at-a-time file processing.

To determine when all the tuples in the result of the query have been processed, the communication variable SQLCODE (or, alternatively, SQLSTATE) is checked. If a FETCH command is issued that results in moving the cursor past the last tuple in the result of the query, a positive value (SQLCODE > 0) is returned in SQLCODE, indicating that no data (tuple) was found (or the string "02000" is returned in SQLSTATE). The programmer uses this to terminate a loop over the tuples in the query result. In general, numerous cursors can be opened at the same time. A CLOSE CURSOR command is issued to indicate that we are done with processing the result of the query associated with that cursor.

An example of using cursors is shown in Figure 9.4, where a cursor called EMP is declared in line 4. We assume that appropriate C program variables have been declared as in Figure 9.2. The program segment in E2 reads (inputs) a department name (line 0), retrieves its department number (lines 1 to 3), and then retrieves the employees who

```
//Program Segment E2:
0)   prompt("Enter the Department Name: ", dname) ;
1)   EXEC SQL
2)      select DNUMBER into :dnumber
3)      from DEPARTMENT where DNAME = :dname ;
4)   EXEC SQL DECLARE EMP CURSOR FOR
5)      select SSN, FNAME, MINIT, LNAME, SALARY
6)      from EMPLOYEE where DNO = :dnumber
7)      FOR UPDATE OF SALARY ;
8)   EXEC SQL OPEN EMP ;
9)   EXEC SQL FETCH from EMP into :ssn, :fname, :minit, :lname, :salary ;
10)  while (SQLCODE == 0) {
11)     printf("Employee name is:", fname, minit, lname)
12)     prompt("Enter the raise amount: ", raise) ;
13)     EXEC SQL
14)        update EMPLOYEE
15)        set SALARY = SALARY + :raise
16)        where CURRENT OF EMP ;
17)     EXEC SQL FETCH from EMP into :ssn, :fname, :minit, :lname, :salary ;
18)     }
19)  EXEC SQL CLOSE EMP ;
```

FIGURE 9.4 Program segment E2, a C program segment that uses cursors with embedded SQL for update purposes.

work in that department via a cursor. A loop (lines 10 to 18) then iterates over each employee record, one at a time, and prints the employee name. The program then reads a raise amount for that employee (line 12) and updates the employee's salary in the database by the raise amount (lines 14 to 16).

When a cursor is defined for rows that are to be modified (updated), we must add the clause **FOR UPDATE OF** in the cursor declaration and list the names of any attributes that will be updated by the program. This is illustrated in line 7 of code segment E2. If rows are to be deleted, the keywords **FOR UPDATE** must be added without specifying any attributes. In the embedded UPDATE (or DELETE) command, the condition **WHERE CURRENT OF** <cursor name> specifies that the current tuple referenced by the cursor is the one to be updated (or deleted), as in line 16 of E2.

Notice that declaring a cursor and associating it with a query (lines 4 through 7 in E2) does not execute the query; the query is executed only when the OPEN <cursor name> command (line 8) is executed. Also notice that there is no need to include the **FOR UPDATE OF** clause in line 7 of E2 if the results of the query are to be used *for retrieval purposes only* (no update or delete).

Several options can be specified when declaring a cursor. The general form of a cursor declaration is as follows:

DECLARE <cursor name> [**INSENSITIVE**] [**SCROLL**] **CURSOR**
[**WITH HOLD**] **FOR** <query specification>
[**ORDER BY** <ordering specification>]
[**FOR READ ONLY** | **FOR UPDATE** [**OF** <attribute list>]] ;

We already briefly discussed the options listed in the last line. The default is that the query is for retrieval purposes (FOR READ ONLY). If some of the tuples in the query result are to be updated, we need to specify FOR UPDATE OF <attribute list> and list the attributes that may be updated. If some tuples are to be deleted, we need to specify FOR UPDATE without any attributes listed.

When the optional keyword SCROLL is specified in a cursor declaration, it is possible to position the cursor in other ways than for purely sequential access. A **fetch orientation** can be added to the FETCH command, whose value can be one of NEXT, PRIOR, FIRST, LAST, ABSOLUTE *i*, and RELATIVE *i*. In the latter two commands, *i* must evaluate to an integer value that specifies an absolute tuple position or a tuple position relative to the current cursor position, respectively. The default fetch orientation, which we used in our examples, is NEXT. The fetch orientation allows the programmer to move the cursor around the tuples in the query result with greater flexibility, providing random access by position or access in reverse order. When SCROLL is specified on the cursor, the general form of a FETCH command is as follows, with the parts in square brackets being optional:

FETCH [[<fetch orientation>] **FROM**] <cursor name> **INTO** <fetch target list> ;

The ORDER BY clause orders the tuples so that the FETCH command will fetch them in the specified order. It is specified in a similar manner to the corresponding clause for SQL queries (see Section 8.4.6). The last two options when declaring a cursor (INSENSITIVE and WITH HOLD) refer to transaction characteristics of database programs, which we discuss in Chapter 17.

9.4.3 Specifying Queries at Runtime Using Dynamic SQL

In the previous examples, the embedded SQL queries were written as part of the host program source code. Hence, any time we want to write a different query, we must write a new program, and go through all the steps involved (compiling, debugging, testing, and so on). In some cases, it is convenient to write a program that can execute different SQL queries or updates (or other operations) *dynamically at runtime*. For example, we may want to write a program that accepts an SQL query typed from the monitor, executes it, and displays its result, such as the interactive interfaces available for most relational DBMSs. Another example is when a user-friendly interface generates SQL queries dynamically for the user based on point-and-click operations on a graphical schema (for example, a QBE-like interface; see Appendix D). In this section, we give a brief overview of **dynamic SQL**, which is one technique for writing this type of database program, by giving a simple example to illustrate how dynamic SQL can work.

Program segment E3 in Figure 9.5 reads a string that is input by the user (that string should be an SQL update command) into the string variable `sqlupdatestring` in line 3. It then prepares this as an SQL command in line 4 by associating it with the SQL variable `sqlcommand`. Line 5 then executes the command. Notice that in this case no syntax check or other types of checks on the command are possible *at compile time*, since the command is not available until runtime. This contrasts with our previous examples of embedded SQL, where the query could be checked at compile time because its text was in the program source code.

Although including a dynamic update command is relatively straightforward in dynamic SQL, a dynamic query is much more complicated. This is because in the general case we do not know the type or the number of attributes to be retrieved by the SQL query when we are writing the program. A complex data structure is sometimes needed to allow for different numbers and types of attributes in the query result if no prior information is known about the dynamic query. Techniques similar to those that we discuss in Section 9.5 can be used to assign query results (and query parameters) to host program variables.

In E3, the reason for separating PREPARE and EXECUTE is that if the command is to be executed multiple times in a program, it can be prepared only once. Preparing the command generally involves syntax and other types of checks by the system, as well as

```
//Program Segment E3:
0)    EXEC SQL BEGIN DECLARE SECTION ;
1)    varchar sqlupdatestring [256] ;
2)    EXEC SQL END DECLARE SECTION ;
      ...
3)    prompt("Enter the Update Command: ", sqlupdatestring) ;
4)    EXEC SQL PREPARE sqlcommand FROM :sqlupdatestring ;
5)    EXEC SQL EXECUTE sqlcommand ;
      ...
```

FIGURE 9.5 Program segment E3, a C program segment that uses dynamic SQL for updating a table.

generating the code for executing it. It is possible to combine the PREPARE and EXECUTE commands (lines 4 and 5 in E3) into a single statement by writing

```
EXEC SQL EXECUTE IMMEDIATE :sqlupdatestring ;
```

This is useful if the command is to be executed only once. Alternatively, one can separate the two to catch any errors after the PREPARE statement, if any.

9.4.4 SQLJ: Embedding SQL Commands in JAVA

In the previous sections, we gave an overview of how SQL commands can be embedded in a traditional programming language, using the C language in our examples. We now turn our attention to how SQL can be embedded in an object-oriented programming language,[8] in particular, the JAVA language. SQLJ is a standard that has been adopted by several vendors for embedding SQL in JAVA. Historically, SQLJ was developed after JDBC, which is used for accessing SQL databases from JAVA using function calls. We discuss JDBC in Section 9.5.2. In our discussion, we focus on SQLJ as it is used in the ORACLE RDBMS. An SQLJ translator will generally convert SQL statements into JAVA, which can then be executed through the JDBC interface. Hence, it is necessary to install a *JDBC driver* when using SQLJ.[9] In this section, we focus on how to use SQLJ concepts to write embedded SQL in a JAVA program.

Before being able to process SQLJ with JAVA in ORACLE, it is necessary to import several class libraries, shown in Figure 9.6. These include the JDBC and IO classes (lines 1 and 2), plus the additional classes listed in lines 3, 4, and 5. In addition, the program must first connect to the desired database using the function call `getConnection`, which is one of the methods of the `oracle` class in line 5 of Figure 9.6. The format of this function call, which returns an object of type *default context*,[10] is as follows:

```
public static DefaultContext
getConnection(String url, String user, String password, Boolean
autoCommit)
throws SQLException ;
```

For example, we can write the statements in lines 6 through 8 in Figure 9.6 to connect to an ORACLE database located at the URL <url name> using the login of <user name> and <password> with automatic commitment of each command,[11] and then set this connection as the **default context** for subsequent commands.

---

8. This section assumes familiarity with object-oriented concepts and basic JAVA concepts. If readers lack this familiarity, they should postpone this section until after reading Chapter 20.

9. We discuss JDBC drivers in Section 9.5.2.

10. A *default context*, when set, applies to subsequent commands in the program until it is changed.

11. *Automatic commitment* roughly means that each command is applied to the database after it is executed. The alternative is that the programmer wants to execute several related database commands and then commit them together. We discuss commit concepts in Chapter 17 when we describe database transactions.

```
1)   import java.sql.* ;
2)   import java.io.* ;
3)   import sqlj.runtime.* ;
4)   import sqlj.runtime.ref.* ;
5)   import oracle.sqlj.runtime.* ;
     ...
6)   DefaultContext cntxt =
7)      oracle.getConnection("<url name>", "<user name>", "<password>", true) ;
8)   DefaultContext.setDefaultContext(cntxt) ;
     ...
```

FIGURE 9.6 Importing classes needed for including SQLJ in JAVA programs in ORACLE, and establishing a connection and default context.

In the following examples, we will not show complete JAVA classes or programs since it is not our intention to teach JAVA. Rather, we will show program segments that illustrate the use of SQLJ. Figure 9.7 shows the JAVA program variables used in our examples. Program segment J1 in Figure 9.8 reads an employee's social security number and prints some of the employee's information from the database.

Notice that because JAVA already uses the concept of **exceptions** for error handling, a special exception called `SQLException` is used to return errors or exception conditions after executing an SQL database command. This plays a similar role to SQLCODE and SQLSTATE in embedded SQL. JAVA has many types of predefined exceptions. Each JAVA operation (function) must specify the exceptions that can be **thrown**—that is, the exception conditions that may occur while executing the JAVA code of that operation. If a defined exception occurs, the system transfers control to the JAVA code specified for exception handling. In J1, exception handling for an `SQLException` is specified in lines 7 and 8. Exceptions that can be thrown by the code in a particular operation should be specified as part of the operation declaration or *interface*—for example, in the following format:

```
<operation return type> <operation name>(<parameters>) throws
SQLException, IOException ;
```

In SQLJ, the embedded SQL commands within a JAVA program are preceded by `#sql`, as illustrated in J1 line 3, so that they can be identified by the preprocessor. SQLJ uses an *INTO clause*—similar to that used in embedded SQL—to return the attribute values retrieved from the database by an SQL query into JAVA program variables. The program variables are preceded by colons (:) in the SQL statement, as in embedded SQL.

```
1)   string dname, ssn , fname, fn, lname, ln, bdate, address  ;
2)   char sex, minit, mi ;
3)   double salary, sal ;
4)   integer dno, dnumber ;
```

FIGURE 9.7 JAVA program variables used in SQLJ examples J1 and J2.

```
//Program Segment J1:
1)  ssn = readEntry("Enter a Social Security Number: ") ;
2)  try {
3)     #sql{select FNAME, MINIT, LNAME, ADDRESS, SALARY
4)        into :fname, :minit, :lname, :address, :salary
5)        from EMPLOYEE where SSN = :ssn} ;
6)  } catch (SQLException se) {
7)     System.out.println("Social Security Number does not exist: " + ssn) ;
8)     Return ;
9)     }
10) System.out.println(fname + " " + minit + " " + lname + " " + address + " " +
    salary)
```

FIGURE 9.8 Program segment J1, a JAVA program segment with SQLJ.

In J1 a single tuple is selected by the embedded SQLJ query; that is why we are able to assign its attribute values directly to JAVA program variables in the INTO clause in line 4. For queries that retrieve many tuples, SQLJ uses the concept of an *iterator*, which is somewhat similar to a cursor in embedded SQL.

9.4.5 Retrieving Multiple Tuples in SQLJ Using Iterators

In SQLJ, an **iterator** is a type of object associated with a collection (set or multiset) of tuples in a query result.[12] The iterator is associated with the tuples and attributes that appear in a query result. There are two types of iterators:

1. A **named iterator** is associated with a query result by listing the attribute *names and types* that appear in the query result.
2. A **positional iterator** lists only the *attribute types* that appear in the query result.

In both cases, the list should be *in the same order* as the attributes that are listed in the SELECT clause of the query. However, looping over a query result is different for the two types of iterators, as we shall see. First, we show an example of using a *named* iterator in Figure 9.9, program segment J2A. Line 9 in Figure 9.9 shows how a named iterator type Emp is declared. Notice that the names of the attributes in a named iterator type must match the names of the attributes in the SQL query result. Line 10 shows how an iterator object e of type Emp is created in the program and then associated with a query (lines 11 and 12).

When the iterator object is associated with a query (lines 11 and 12 in Figure 9.9), the program fetches the query result from the database and sets the iterator to a position *before the first row* in the result of the query. This becomes the **current row** for the iterator. Subsequently, **next** operations are issued on the iterator; each moves the iterator to the *next row* in the result of the query, making it the current row. If the row exists, the

12. We discuss iterators in more detail in Chapter 21 when we discuss object databases.

```
//Program Segment J2A:
0)   dname = readEntry("Enter the Department Name: ") ;
1)   try {
2)      #sql{select DNUMBER into :dnumber
3)         from DEPARTMENT where DNAME = :dname} ;
4)   }  catch (SQLException se) {
5)      System.out.println("Department does not exist: " + dname) ;
6)      Return ;
7)      }
8)   System.out.println("Employee information for Department: " + dname) ;
9)   #sql iterator Emp(String ssn, String fname, String minit, String lname,
     double salary) ;
10)  Emp e = null ;
11)  #sql e = {select ssn, fname, minit, lname, salary
12)            from EMPLOYEE where DNO = :dnumber} ;
13)  while (e.next()) {
14)     System.out.println(e.ssn + " " + e.fname + " " + e.minit + " " +
           e.lname + " " + e.salary) ;
15)     } ;
16)  e.close() ;
```

FIGURE 9.9 Program segment J2A, a JAVA program segment that uses a named iterator to print employee information in a particular department.

operation retrieves the attribute values for that row into the corresponding program variables. If no more rows exist, the `next` operation returns null, and can thus be used to control the looping.

In Figure 9.9, the command `(e.next())` in line 13 performs two functions: It gets the next tuple in the query result and controls the while loop. Once we are done with the query result, the command `e.close()` (line 16) closes the iterator.

Next, consider the same example using *positional* iterators as shown in Figure 9.10 (program segment J2B). Line 9 in Figure 9.10 shows how a positional iterator type `Emppos` is declared. The main difference between this and the named iterator is that there are no attribute names in the positional iterator—only attribute types. They still must be compatible with the attribute types in the SQL query result and in the same order. Line 10 shows how a positional iterator variable `e` of type `Emppos` is created in the program and then associated with a query (lines 11 and 12).

The positional iterator behaves in a manner that is more similar to embedded SQL (see Section 9.4.2). A **fetch <iterator variable> into <program variables>** command is needed to get the next tuple in a query result. The first time `fetch` is executed, it gets the first tuple (line 13 in Figure 9.10). Line 16 gets the next tuple until no more tuples exist in the query result. To control the loop, a positional iterator function `e.endFetch()` is used. This function is set to a value of TRUE when the iterator is initially associated with an SQL query (line 11), and is set to FALSE each time a fetch command returns a valid tuple from the query result. It is set to TRUE again when a fetch command does not find any more tuples. Line 14 shows how the looping is controlled by negation.

```
//Program Segment J2B:
0)  dname = readEntry("Enter the Department Name: ") ;
1)  try {
2)     #sql{select DNUMBER into :dnumber
3)        from DEPARTMENT where DNAME = :dname} ;
4)  } catch (SQLException se) {
5)     System.out.println("Department does not exist: " + dname) ;
6)     Return ;
7)     }
8)  System.out.println("Employee information for Department: " + dname) ;
9)  #sql iterator Emppos(String, String, String, String, double) ;
10) Emppos e = null ;
11) #sql e ={select ssn, fname, minit, lname, salary
12)    from EMPLOYEE where DNO = :dnumber} ;
13) #sql {fetch :e into :ssn, :fn, :mi, :ln, :sal} ;
14) while (!e.endFetch()) {
15)    System.out.println(ssn + " " + fn + " " + mi + " " + ln + " " + sal) ;
16)    #sql {fetch :e into :ssn, :fn, :mi, :ln, :sal} ;
17)    } ;
18) e.close() ;
```

FIGURE 9.10 Program segment J2B, a JAVA program segment that uses a positional iterator to print employee information in a particular department.

9.5 Database Programming with Function Calls: SQL/CLI and JDBC

Embedded SQL (see Section 9.4) is sometimes referred to as a **static** database programming approach because the query text is written within the program and cannot be changed without recompiling or reprocessing the source code. The use of function calls is a more **dynamic** approach for database programming than embedded SQL. We already saw one dynamic database programming technique—dynamic SQL—in Section 9.4.3. The techniques discussed here provide another approach to dynamic database programming. A **library of functions,** also known as an **application programming interface (API)**, is used to access the database. Although this provides more flexibility because no preprocessor is needed, one drawback is that syntax and other checks on SQL commands have to be done at runtime. Another drawback is that it sometimes requires more complex programming to access query results because the types and numbers of attributes in a query result may not be known in advance.

In this section, we give an overview of two function call interfaces. We first discuss **SQL/CLI** (Call Level Interface), which is part of the SQL standard. This was developed as a follow-up to the earlier technique know as ODBC (Open Data Base Connectivity). We use C as the host language in our SQL/CLI examples. Then we give an overview of **JDBC,** which is the call function interface for accessing databases from JAVA. Although it is commonly assumed that JDBC stands for Java Data Base Connectivity, JDBC is just a registered trademark of Sun Microsystems, not an acronym.

The main advantage of using a function call interface is that it makes it easier to access multiple databases within the same application program, even if they are stored under different DBMS packages. We discuss this further in Section 9.5.2 when we discuss JAVA database programming with JDBC, although this advantage also applies to database programming with SQL/CLI and ODBC (see Section 9.5.1).

9.5.1 Database Programming with SQL/CLI Using C as the Host Language

Before using the function calls in SQL/CLI, it is necessary to install the appropriate library packages on the database server. These packages are obtained from the vendor of the DBMS being used. We now give an overview of how SQL/CLI can be used in a C program. We shall illustrate our presentation with the example program segment CLI1 shown in Figure 9.11.

When using SQL/CLI, the SQL statements are dynamically created and passed as string parameters in the function calls. Hence, it is necessary to keep track of the information about host program interactions with the database in runtime data structures, because the database commands are processed at runtime. The information is kept in four types of

```
//Program CLI1:
0)   #include sqlcli.h ;
1)   void printSal() {
2)   SQLHSTMT stmt1 ;
3)   SQLHDBC con1 ;
4)   SQLHENV env1 ;
5)   SQLRETURN ret1, ret2, ret3, ret4 ;
6)   ret1 = SQLAllocHandle(SQL_HANDLE_ENV, SQL_NULL_HANDLE, &env1) ;
7)   if (!ret1) ret2 = SQLAllocHandle(SQL_HANDLE_DBC, env1, &con1) else exit ;
8)   if (!ret2) ret3 = SQLConnect(con1, "dbs", SQL_NTS, "js", SQL_NTS, "xyz", SQL_NTS)
else exit ;
9)   if (!ret3) ret4 = SQLAllocHandle(SQL_HANDLE_STMT, con1, &stmt1) else exit ;
10)  SQLPrepare(stmt1, "select LNAME, SALARY from EMPLOYEE where SSN = ?", SQL_NTS) ;
11)  prompt("Enter a Social Security Number: ", ssn) ;
12)  SQLBindParameter(stmt1, 1, SQL_CHAR, &ssn, 9, &fetchlen1) ;
13)  ret1 = SQLExecute(stmt1) ;
14)  if (!ret1) {
15)      SQLBindCol(stmt1, 1, SQL_CHAR, &lname, 15, &fetchlen1) ;
16)      SQLBindCol(stmt1, 2, SQL_FLOAT, &salary, 4, &fetchlen2) ;
17)      ret2 = SQLFetch(stmt1) ;
18)      if (!ret2) printf(ssn, lname, salary)
19)          else printf("Social Security Number does not exist: ", ssn) ;
20)      }
21)  }
```

FIGURE 9.11 Program segment CLI1, a C program segment with SQL/CLI.

records, represented as *structs* in C data types. An **environment record** is used as a container to keep track of one or more database connections and to set environment information. A **connection record** keeps track of the information needed for a particular database connection. A **statement record** keeps track of the information needed for one SQL statement. A **description record** keeps track of the information about tuples or parameters—for example, the number of attributes and their types in a tuple, or the number and types of parameters in a function call.

Each record is accessible to the program through a C pointer variable—called a **handle** to the record. The handle is returned when a record is first created. To create a record and return its handle, the following SQL/CLI function is used:

```
SQLAllocHandle(<handle_type>, <handle_1>, <handle_2>)
```

In this function, the parameters are as follows:

- `<handle_type>` indicates the type of record being created. The possible values for this parameter are the keywords SQL_HANDLE_ENV, SQL_HANDLE_DBC, SQL_HANDLE_STMT, or SQL_HANDLE_DESC, for an environment, connection, statement, or description record, respectively.
- `<handle_1>` indicates the container within which the new handle is being created. For example, for a connection record this would be the environment within which the connection is being created, and for a statement record this would be the connection for that statement.
- `<handle_2>` is the pointer (handle) to the newly created record of type `<handle_type>`.

When writing a C program that will include database calls through SQL/CLI, the following are the typical steps that are taken. We illustrate the steps by referring to the example CLI1 in Figure 9.11, which reads a social security number of an employee and prints the employee's last name and salary.

1. The *library of functions* comprising SQL/CLI must be included in the C program. This is called `sqlcli.h`, and is included using line 0 in Figure 9.11.
2. Declare *handle variables* of types SQLHSTMT, SQLHDBC, SQLHENV, and SQLHDESC for the statements, connections, environments, and descriptions needed in the program, respectively (lines 2 to 4).[13] Also declare variables of type SQLRETURN (line 5) to hold the return codes from the SQL/CLI function calls. A return code of 0 (zero) indicates *successful execution* of the function call.
3. An *environment record* must be set up in the program using `SQLAllocHandle`. The function to do this is shown in line 6. Because an environment record is not contained in any other record, the parameter `<handle_1>` is the null handle SQL_NULL_HANDLE (null pointer) when creating an environment. The handle (pointer) to the newly created environment record is returned in variable `env1` in line 6.
4. A *connection record* is set up in the program using `SQLAllocHandle`. In line 7, the connection record created has the handle `con1` and is contained in the environ-

13. We will not show description records here, to keep our presentation simple.

ment env1. A **connection** is then established in con1 to a particular server database using the SQLConnect function of SQL/CLI (line 8). In our example, the database server name we are connecting to is "dbs", and the account name and password for login are "js" and "xyz", respectively.

5. A *statement record* is set up in the program using SQLAllocHandle. In line 9, the statement record created has the handle stmt1 and uses the connection con1.
6. The statement is *prepared* using the SQL/CLI function SQLPrepare. In line 10, this assigns the SQL statement string (the query in our example) to the statement handle stmt1. The question mark (?) symbol in line 10 represents a **statement parameter,** which is a value to be determined at runtime—typically by binding it to a C program variable. In general, there could be several parameters. They are distinguished by the order of appearance of the question marks in the statement (the first ? represents parameter 1, the second ? represents parameter 2, and so on). The last parameter in SQLPrepare should give the length of the SQL statement string in bytes, but if we enter the keyword SQL_NTS, this indicates that the string holding the query is a *null-terminated string* so that SQL can calculate the string length automatically. This also applies to other string parameters in the function calls.
7. Before executing the query, any parameters should be bound to program variables using the SQL/CLI function SQLBindParameter. In Figure 9.11, the parameter (indicated by ?) to the prepared query referenced by stmt1 is bound to the C program variable ssn in line 12. If there are n parameters in the SQL statement, we should have n SQLBindParameter function calls, each with a different parameter position $(1, 2, \ldots, n)$.
8. Following these preparations, we can now execute the SQL statement referenced by the handle stmt1 using the function SQLExecute (line 13). Notice that although the query will be executed in line 13, the query results have not yet been assigned to any C program variables.
9. In order to determine where the result of the query is returned, one common technique is the **bound columns** approach. Here, each column in a query result is bound to a C program variable using the SQLBindCol function. The columns are distinguished by their order of appearance in the SQL query. In Figure 9.11 lines 15 and 16, the two columns in the query (LNAME and SALARY) are bound to the C program variables lname and salary, respectively.[14]
10. Finally, in order to retrieve the column values into the C program variables, the function SQLFetch is used (line 17). This function is similar to the FETCH command of embedded SQL. If a query result has a collection of tuples, each SQLFetch call gets the next tuple and returns its column values into the bound

14. An alternative technique known as **unbound columns** uses different SQL/CLI functions, namely SQLGetCol or SQLGetData, to retrieve columns from the query result without previously binding them; these are applied after the SQLFetch command in step 17.

program variables. SQLFetch returns an exception (nonzero) code if there are no more tuples.[15]

As we can see, using dynamic function calls requires a lot of preparation to set up the SQL statements and to bind parameters and query results to the appropriate program variables.

In CLI1 a *single tuple* is selected by the SQL query. Figure 9.12 shows an example of retrieving multiple tuples. We assume that appropriate C program variables have been declared as in Figure 9.12. The program segment in CLI2 reads (inputs) a department number and then retrieves the employees who work in that department. A loop then iterates over each employee record, one at a time, and prints the employee's last name and salary.

9.5.2 JDBC: SQL Function Calls for JAVA Programming

We now turn our attention to how SQL can be called from the JAVA object-oriented programming language.[16] The function libraries for this access are known as **JDBC.**[17] The JAVA programming language was designed to be platform independent—that is, a program should be able to run on any type of computer system that has a JAVA interpreter installed. Because of this portability, many RDBMS vendors provide JDBC drivers so that it is possible to access their systems via JAVA programs. A **JDBC driver** is basically an implementation of the function calls specified in the JDBC API (Application Programming Interface) for a particular vendor's RDBMS. Hence, a JAVA program with JDBC function calls can access any RDBMS that has a JDBC driver available.

Because JAVA is object-oriented, its function libraries are implemented as **classes.** Before being able to process JDBC function calls with JAVA, it is necessary to import the **JDBC class libraries,** which are called `java.sql.*`. These can be downloaded and installed via the Web.[18]

JDBC is designed to allow a single JAVA program to connect to several different databases. These are sometimes called the **data sources** accessed by the JAVA program. These data sources could be stored using RDBMSs from different vendors and could reside on different machines. Hence, different data source accesses within the same JAVA program may require JDBC drivers from different vendors. To achieve this flexibility, a special JDBC class called the **driver manager** class is employed, which keeps track of the installed drivers. A driver should be *registered* with the driver

15. If unbound program variables are used, SQLFetch returns the tuple into a temporary program area. Each subsequent SQLGetCol (or SQLGetData) returns one attribute value in order.

16. This section assumes familiarity with object-oriented concepts and basic JAVA concepts. If readers lack this familiarity, they should postpone this section until after reading Chapter 20.

17. As we mentioned earlier, **JDBC** is a registered trademark of Sun Microsystems, although it is commonly thought to be an acronym for Java Data Base Connectivity.

18. These are available from several Web sites—for example, through the Web site at the URL http://industry.java.sun.com/products/jdbc/drivers.

manager before it is used. The operations (methods) of the driver manager class include `getDriver`, `registerDriver`, and `deregisterDriver`. These can be used to add and remove drivers dynamically. Other functions set up and close connections to data sources, as we shall see.

To load a JDBC driver explicitly, the generic JAVA function for loading a class can be used. For example, to load the JDBC driver for the ORACLE RDBMS, the following command can be used:

```
Class.forName("oracle.jdbc.driver.OracleDriver")
```

This will register the driver with the driver manager and make it available to the program. It is also possible to load and register the driver(s) needed in the command line that runs the program, for example, by including the following in the command line:

```
-Djdbc.drivers = oracle.jdbc.driver
```

The following are typical steps that are taken when writing a JAVA application program with database access through JDBC function calls. We illustrate the steps by referring to the example JDBC1 in Figure 9.13, which reads a social security number of an employee and prints the employee's last name and salary.

```
//Program Segment CLI2:
0)   #include sqlcli.h ;
1)   void printDepartmentEmps() {
2)   SQLHSTMT stmt1 ;
3)   SQLHDBC con1 ;
4)   SQLHENV env1 ;
5)   SQLRETURN ret1, ret2, ret3, ret4 ;
6)   ret1 = SQLAllocHandle(SQL_HANDLE_ENV, SQL_NULL_HANDLE, &env1) ;
7)   if (!ret1) ret2 = SQLAllocHandle(SQL_HANDLE_DBC, env1, &con1) else exit ;
8)   if (!ret2) ret3 = SQLConnect(con1, "dbs", SQL_NTS, "js", SQL_NTS, "xyz", SQL_NTS)
else exit ;
9)   if (!ret3) ret4 = SQLAllocHandle(SQL_HANDLE_STMT, con1, &stmt1) else exit ;
10)  SQLPrepare(stmt1, "select LNAME, SALARY from EMPLOYEE where DNO = ?", SQL_NTS) ;
11)  prompt("Enter the Department Number: ", dno) ;
12)  SQLBindParameter(stmt1, 1, SQL_INTEGER, &dno, 4, &fetchlen1) ;
13)  ret1 = SQLExecute(stmt1) ;
14)  if (!ret1) {
15)      SQLBindCol(stmt1, 1, SQL_CHAR, &lname, 15, &fetchlen1) ;
16)      SQLBindCol(stmt1, 2, SQL_FLOAT, &salary, 4, &fetchlen2) ;
17)      ret2 = SQLFetch(stmt1) ;
18)      while (!ret2) {
19)          printf(lname, salary) ;
20)          ret2 = SQLFetch(stmt1) ;
21)          }
22)      }
23)  }
```

FIGURE 9.12 Program segment CLI2, a C program segment that uses SQL/CLI for a query with a collection of tuples in its result.

1. The JDBC *library of classes* must be imported into the JAVA program. These classes are called `java.sql.*`, and can be imported using line 1 in Figure 9.13. Any additional JAVA class libraries needed by the program must also be imported.
2. Load the JDBC driver as discussed previously (lines 4 to 7). The JAVA exception in line 5 occurs if the driver is not loaded successfully.
3. Create appropriate variables as needed in the JAVA program (lines 8 and 9).
4. A **connection object** is created using the `getConnection` function of the `DriverManager` class of JDBC. In lines 12 and 13, the connection object is created by using the function call `getConnection(urlstring)`, where `urlstring` has the form

   ```
   jdbc:oracle:<driverType>:<dbaccount>/<password>
   ```

   An alternative form is

   ```
   getConnection(url, dbaccount, password)
   ```

   Various properties can be set for a connection object, but they are mainly related to transactional properties, which we discuss in Chapter 17.
5. A **statement object** is created in the program. In JDBC, there is a basic statement class, `Statement`, with two specialized subclasses: `PreparedStatement` and `CallableStatement`. This example illustrates how `PreparedStatement` objects are created and used. The next example (Figure 9.14) illustrates the other type of `Statement` objects. In line 14, a query string with a single parameter—indicated by the "?" symbol—is created in the variable `stmt1`. In line 15, an object `p` of type `PreparedStatement` is created based on the query string in `stmt1` and using the connection object `conn`. In general, the programmer should use `PreparedStatement` objects if a query is to be executed multiple times, since it would be prepared, checked, and compiled only once, thus saving this cost for the additional executions of the query.
6. The question mark (?) symbol in line 14 represents a **statement parameter,** which is a value to be determined at runtime, typically by binding it to a JAVA program variable. In general, there could be several parameters, distinguished by the order of appearance of the question marks (first ? represents parameter 1, second ? represents parameter 2, and so on) in the statement, as discussed previously.
7. Before executing a `PreparedStatement` query, any parameters should be bound to program variables. Depending on the type of the parameter, functions such as `setString`, `setInteger`, `setDouble`, and so on are applied to the `PreparedStatement` object to set its parameters. In Figure 9.13, the parameter (indicated by ?) in object `p` is bound to the JAVA program variable `ssn` in line 18. If there are n parameters in the SQL statement, we should have n `Set...` functions, each with a different parameter position (1, 2, ..., n). Generally, it is advisable to clear all parameters before setting any new values (line 17).
8. Following these preparations, we can now execute the SQL statement referenced by the object `p` using the function `executeQuery` (line 19). There is a generic function `execute` in JDBC, plus two specialized functions: `executeUpdate` and `executeQuery`. `executeUpdate` is used for SQL insert, delete, or update statements,

```
//Program JDBC1:
0)   import java.io.* ;
1)   import java.sql.*
     ...
2)   class getEmpInfo {
3)      public static void main (String args []) throws SQLException, IOException {
4)      try { Class.forName("oracle.jdbc.driver.OracleDriver")
5)      } catch (ClassNotFoundException x) {
6)         System.out.println ("Driver could not be loaded") ;
7)         }
8)      String dbacct, passwrd, ssn, lname ;
9)      Double salary ;
10)     dbacct = readentry("Enter database account:") ;
11)     passwrd = readentry("Enter pasword:") ;
12)     Connection conn = DriverManager.getConnection
13)        ("jdbc:oracle:oci8:" + dbacct + "/" + passwrd) ;
14)     String stmt1 = "select LNAME, SALARY from EMPLOYEE where SSN = ?" ;
15)     PreparedStatement p = conn.prepareStatement(stmt1) ;
16)     ssn = readentry("Enter a Social Security Number: ") ;
17)     p.clearParameters() ;
18)     p.setString(1, ssn) ;
19)     ResultSet r = p.executeQuery() ;
20)     while (r.next()) {
21)        lname = r.getString(1) ;
22)        salary = r.getDouble(2) ;
23)        system.out.println(lname + salary) ;
24)     }  }
25)  }
```

FIGURE 9.13 Program segment JDBC1, a JAVA program segment with JDBC.

and returns an integer value indicating the number of tuples that were affected. `executeQuery` is used for SQL retrieval statements, and returns an object of type `ResultSet`, which we discuss next.

9. In line 19, the result of the query is returned in an object `r` of type `ResultSet`. This resembles a two-dimensional array or a table, where the tuples are the rows and the attributes returned are the columns. A `ResultSet` object is similar to a cursor in embedded SQL and an iterator in SQLJ. In our example, when the query is executed, `r` refers to a tuple before the first tuple in the query result. The `r.next()` function (line 20) moves to the next tuple (row) in the `ResultSet` object and returns `null` if there are no more objects. This is used to control the looping. The programmer can refer to the attributes in the current tuple using various `get...` functions that depend on the type of each attribute (for example, `getString`, `getInteger`, `getDouble`, and so on). The programmer can either use the attribute positions (1, 2) or the actual attribute names (`"LNAME"`, `"SALARY"`)

with the get... functions. In our examples, we used the positional notation in lines 21 and 22.

In general, the programmer can check for SQL exceptions after each JDBC function call.

Notice that JDBC does not distinguish between queries that return single tuples and those that return multiple tuples, unlike some of the other techniques. This is justifiable because a single tuple result set is just a special case.

In example JDBC1, a *single tuple* is selected by the SQL query, so the loop in lines 20 to 24 is executed at most once. The next example, shown in Figure 9.14, illustrates the retrieval of multiple tuples. The program segment in JDBC2 reads (inputs) a department number and then retrieves the employees who work in that department. A loop then iterates over each employee record, one at a time, and prints the employee's last name and salary. This example also illustrates how we can execute a query directly, without having to prepare it as in the previous example. This technique is preferred for queries

```
//Program Segment JDBC2:
0)   import java.io.* ;
1)   import java.sql.*
     ...
2)   class printDepartmentEmps {
3)      public static void main (String args []) throws SQLException, IOException {
4)      try { Class.forName("oracle.jdbc.driver.OracleDriver")
5)      } catch (ClassNotFoundException x) {
6)         System.out.println ("Driver could not be loaded") ;
7)         }
8)      String dbacct, passwrd, lname ;
9)      Double salary ;
10)     Integer dno ;
11)     dbacct = readentry("Enter database account:") ;
12)     passwrd = readentry("Enter pasword:") ;
13)     Connection conn = DriverManager.getConnection
14)        ("jdbc:oracle:oci8:" + dbacct + "/" + passwrd) ;
15)     dno = readentry("Enter a Department Number: ") ;
16)     String q = "select LNAME, SALARY from EMPLOYEE where DNO = " +
        dno.tostring() ;
17)     Statement s = conn.createStatement() ;
18)     ResultSet r = s.executeQuery(q) ;
19)     while (r.next()) {
20)        lname = r.getString(1) ;
21)        salary = r.getDouble(2) ;
22)        system.out.println(lname + salary) ;
23)     } }
24)  }
```

FIGURE 9.14 Program segment JDBC2, a JAVA program segment that uses JDBC for a query with a collection of tuples in its result.

that will be executed only once, since it is simpler to program. In line 17 of Figure 9.14, the programmer creates a `Statement` object (instead of `PreparedStatement`, as in the previous example) without associating it with a particular query string. The query string `q` is passed to the statement object `s` when it is executed in line 18.

This concludes our brief introduction to JDBC. The interested reader is referred to the Web site http://java.sun.com/docs/books/tutorial/jdbc/, which contains many further details on JDBC.

9.6 Database Stored Procedures and SQL/PSM

We conclude this chapter with two additional topics related to database programming. In Section 9.6.1, we discuss the concept of stored procedures, which are program modules that are stored by the DBMS at the database server. Then in Section 9.6.2, we discuss the extensions to SQL that are specified in the standard to include general-purpose programming constructs in SQL. These extensions are known as SQL/PSM (SQL/Persistent Stored Modules) and can be used to write stored procedures. SQL/PSM also serves as an example of a database programming language that extends a database model and language—namely, SQL—with some programming constructs, such as conditional statements and loops.

9.6.1 Database Stored Procedures and Functions

In our presentation of database programming techniques so far, there was an implicit assumption that the database application program was running on a client machine that is different from the machine on which the database server—and the main part of the DBMS software package—is located. Although this is suitable for many applications, it is sometimes useful to create database program modules—procedures or functions—that are stored and executed by the DBMS at the database server. These are historically known as database **stored procedures,** although they can be functions or procedures. The term used in the SQL standard for stored procedures is **persistent stored modules,** because these programs are stored persistently by the DBMS, similarly to the persistent data stored by the DBMS.

Stored procedures are useful in the following circumstances:

- If a database program is needed by several applications, it can be stored at the server and invoked by any of the application programs. This reduces duplication of effort and improves software modularity.
- Executing a program at the server can reduce data transfer and hence communication cost between the client and server in certain situations.
- These procedures can enhance the modeling power provided by views by allowing more complex types of derived data to be made available to the database users. In addition, they can be used to check for complex constraints that are beyond the specification power of assertions and triggers.

In general, many commercial DBMSs allow stored procedures and functions to be written in a general-purpose programming language. Alternatively, a stored procedure can

be made of simple SQL commands such as retrievals and updates. The general form of declaring a stored procedures is as follows:

```
CREATE PROCEDURE <procedure name> ( <parameters> )
<local declarations>
<procedure body> ;
```

The parameters and local declarations are optional, and are specified only if needed. For declaring a function, a return type is necessary, so the declaration form is

```
CREATE FUNCTION <function name> ( <parameters> )
RETURNS <return type>
<local declarations>
<function body> ;
```

If the procedure (or function) is written in a general-purpose programming language, it is typical to specify the language, as well as a file name where the program code is stored. For example, the following format can be used:

```
CREATE PROCEDURE <procedure name> ( <parameters> )
LANGUAGE <programming language name>
EXTERNAL NAME <file path name> ;
```

In general, each parameter should have a **parameter type** that is one of the SQL data types. Each parameter should also have a **parameter mode,** which is one of IN, OUT, or INOUT. These correspond to parameters whose values are input only, output (returned) only, or both input and output, respectively.

Because the procedures and functions are stored persistently by the DBMS, it should be possible to call them from the various SQL interfaces and programming techniques. The **CALL statement** in the SQL standard can be used to invoke a stored procedure—either from an interactive interface or from embedded SQL or SQLJ. The format of the statement is as follows:

```
CALL <procedure or function name> ( <argument list> ) ;
```

If this statement is called from JDBC, it should be assigned to a statement object of type `CallableStatement` (see Section 9.5.2).

9.6.2 SQL/PSM: Extending SQL for Specifying Persistent Stored Modules

SQL/PSM is the part of the SQL standard that specifies how to write persistent stored modules. It includes the statements to create functions and procedures that we described in the previous section. It also includes additional programming constructs to enhance the power of SQL for the purpose of writing the code (or body) of stored procedures and functions.

In this section, we discuss the SQL/PSM constructs for conditional (branching) statements and for looping statements. These will give a flavor of the type of constructs

that SQL/PSM has incorporated.[19] Then we give an example to illustrate how these constructs can be used.

The conditional branching statement in SQL/PSM has the following form:

```
IF <condition> THEN <statement list>
        ELSEIF <condition> THEN <statement list>
        ...
        ELSEIF <condition> THEN <statement list>
        ELSE <statement list>
        END IF ;
```

Consider the example in Figure 9.15, which illustrates how the conditional branch structure can be used in an SQL/PSM function. The function returns a string value (line 1) describing the size of a department based on the number of employees. There is one IN integer parameter, `deptno`, which gives a department number. A local variable `NoOfEmps` is declared in line 2. The query in lines 3 and 4 returns the number of employees in the department, and the conditional branch in lines 5 to 8 then returns one of the values {"HUGE", "LARGE", "MEDIUM", "SMALL"} based on the number of employees.

SQL/PSM has several constructs for looping. There are standard while and repeat looping structures, which have the following forms:

```
WHILE <condition> DO
        <statement list>
END WHILE ;
```

```
//Function PSM1:
0)   CREATE FUNCTION DeptSize(IN deptno INTEGER)
1)   RETURNS VARCHAR [7]
2)   DECLARE NoOfEmps INTEGER ;
3)   SELECT COUNT(*) INTO NoOfEmps
4)   FROM EMPLOYEE WHERE DNO = deptno ;
5)   IF NoOfEmps > 100 THEN RETURN "HUGE"
6)      ELSEIF NoOfEmps > 25 THEN RETURN "LARGE"
7)      ELSEIF NoOfEmps > 10 THEN RETURN "MEDIUM"
8)      ELSE RETURN "SMALL"
9)   END IF ;
```

FIGURE 9.15 Declaring a function in SQL/PSM.

19. We only give a brief introduction to SQL/PSM here. There are many other features in the SQL/PSM standard.

```
REPEAT
        <statement list>
UNTIL <condition>
END REPEAT ;
```

There is also a cursor-based looping structure. The statement list in such a loop is executed once for each tuple in the query result. This has the following form:

```
FOR <loop name> AS <cursor name> CURSOR FOR <query> DO
        <statement list>
END FOR ;
```

Loops can have names, and there is a `LEAVE <loop name>` statement to break a loop when a condition is satisfied. SQL/PSM has many other features, but they are outside the scope of our presentation.

9.7 SUMMARY

In this chapter we presented additional features of the SQL database language. In particular, we presented an overview of the most important techniques for database programming. We started in Section 9.1 by presenting the features for specifying general constraints as assertions. Then we discussed the concept of a view in SQL. We then discussed the various approaches to database application programming in Sections 9.3 to 9.6.

Review Questions

9.1. How does SQL allow implementation of general integrity constraints?
9.2. What is a view in SQL, and how is it defined? Discuss the problems that may arise when one attempts to update a view. How are views typically implemented?
9.3. List the three main approaches to database programming. What are the advantages and disadvantages of each approach?
9.4. What is the impedance mismatch problem? Which of the three programming approaches minimizes this problem?
9.5. Describe the concept of a cursor and how it is used in embedded SQL.
9.6. What is SQLJ used for? Describe the two types of iterators available in SQLJ.

Exercises

9.7. Consider the database shown in Figure 1.2, whose schema is shown in Figure 2.1. Write a program segment to read a student's name and print his or her grade point average, assuming that A=4, B=3, C=2, and D=1 points. Use embedded SQL with C as the host language.
9.8. Repeat Exercise 9.7, but use SQLJ with JAVA as the host language.

9.9. Consider the LIBRARY relational database schema of Figure 6.12. Write a program segment that retrieves the list of books that became overdue yesterday and that prints the book title and borrower name for each. Use embedded SQL with C as the host language.

9.10. Repeat Exercise 9.9, but use SQLJ with JAVA as the host language.

9.11. Repeat Exercises 9.7 and 9.9, but use SQL/CLI with C as the host language.

9.12. Repeat Exercises 9.7 and 9.9, but use JDBC with JAVA as the host language.

9.13. Repeat Exercise 9.7, but write a function in SQL/PSM.

9.14. Specify the following views in SQL on the COMPANY database schema shown in Figure 5.5.

a. A view that has the department name, manager name, and manager salary for every department.

b. A view that has the employee name, supervisor name, and employee salary for each employee who works in the 'Research' department.

c. A view that has the project name, controlling department name, number of employees, and total hours worked per week on the project for each project.

d. A view that has the project name, controlling department name, number of employees, and total hours worked per week on the project for each project *with more than one employee working on it*.

9.15. Consider the following view, DEPT_SUMMARY, defined on the COMPANY database of Figure 5.6:

```
CREATE VIEW   DEPT_SUMMARY (D, C, TOTAL_S, AVERAGE_S)
AS SELECT     DNO, COUNT (*), SUM (SALARY), AVG (SALARY)
   FROM       EMPLOYEE
   GROUP BY   DNO;
```

State which of the following queries and updates would be allowed on the view. If a query or update would be allowed, show what the corresponding query or update on the base relations would look like, and give its result when applied to the database of Figure 5.6.

a.
```
SELECT   *
FROM     DEPT_SUMMARY;
```

b.
```
SELECT   D, C
FROM     DEPT_SUMMARY
WHERE    TOTAL_S > 100000;
```

c.
```
SELECT   D, AVERAGE_S
FROM     DEPT_SUMMARY
WHERE    C > (SELECT C FROM DEPT_SUMMARY WHERE D=4);
```

d.
```
UPDATE   DEPT_SUMMARY
SET      D=3
WHERE    D=4;
```

e.
```
DELETE   FROM DEPT_SUMMARY
WHERE    C > 4;
```

Selected Bibliography

The question of view updates is addressed by Dayal and Bernstein (1978), Keller (1982), and Langerak (1990), among others. View implementation is discussed in Blakeley et al. (1989). Negri et al. (1991) describes formal semantics of sql queries.