

part 3

**The Relational Data
Model and SQL**

This page intentionally left blank

The Relational Data Model and Relational Database Constraints

This chapter opens Part 3 of the book, which covers relational databases. The relational data model was first introduced by Ted Codd of IBM Research in 1970 in a classic paper (Codd, 1970), and it 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 SQL/DS system on the MVS operating system by IBM and the Oracle DBMS. Since then, the model has been implemented in a large number of commercial systems, as well as a number of open source systems. Current popular commercial relational DBMSs (RDBMSs) include DB2 (from IBM), Oracle (from Oracle), Sybase DBMS (now from SAP), and SQLServer and Microsoft Access (from Microsoft). In addition, several open source systems, such as MySQL and PostgreSQL, are available.

Because of the importance of the relational model, all of Part 2 is devoted to this model and some of the languages associated with it. In Chapters 6 and 7, we describe some aspects of SQL, which is a comprehensive model and language that is the *standard* for commercial relational DBMSs. (Additional aspects of SQL will be covered in other chapters.) Chapter 8 covers the operations of the relational algebra and introduces the relational calculus—these are two formal languages associated with the relational model. The relational calculus is considered to be the basis for the SQL language, and the relational algebra is used in the internals of many database implementations for query processing and optimization (see Part 8 of the book).

Other features of the relational model are presented in subsequent parts of the book. Chapter 9 relates the relational model data structures to the constructs of the ER and EER models (presented in Chapters 3 and 4), and presents algorithms for designing a relational database schema by mapping a conceptual schema in the ER or EER model into a relational representation. These mappings are incorporated into many database design and CASE¹ tools. Chapters 10 and 11 in Part 4 discuss the programming techniques used to access database systems and the notion of connecting to relational databases via ODBC and JDBC standard protocols. We also introduce the topic of Web database programming in Chapter 11. Chapters 14 and 15 in Part 6 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*.

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 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, discusses how violations of integrity constraints are handled, and introduces the concept of a transaction. Section 5.4 summarizes the chapter.

This chapter and Chapter 8 focus on the formal foundations of the relational model, whereas Chapters 6 and 7 focus on the SQL practical relational model, which is the basis of most commercial and open source relational DBMSs. Many concepts are common between the formal and practical models, but a few differences exist that we shall point out.

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. It is called a **flat file** because each record has a simple linear or *flat* structure. For example, the database of files that was shown in Figure 1.2 is similar to the basic 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. A row represents a fact that typically corresponds to a real-world entity or relationship. The table name and column names are used to help to interpret 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, Student_number,

¹CASE stands for computer-aided software engineering.

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*—formally.

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 formal 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. The use of local phone numbers is quickly becoming obsolete, being replaced by standard ten-digit numbers.
- **Social_security_numbers**. The set of valid nine-digit Social Security numbers. (This is a unique identifier assigned to each person in the United States for employment, tax, and benefits purposes.)
- **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 in a company; each must be an integer value between 15 and 80.
- **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 $R(A_1, A_2, \dots, A_n)$, is made up of a relation name R and a list of attributes, A_1, A_2, \dots, 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 $\text{dom}(A_i)$. 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.

A relation of degree seven, which stores information about university students, would contain seven attributes describing each student as follows:

STUDENT(Name, Ssn, Home_phone, Address, Office_phone, Age, Gpa)

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

STUDENT(Name: string, Ssn: string, Home_phone: string, Address: string,
Office_phone: string, Age: integer, Gpa: real)

For this relation schema, STUDENT is the name of the relation, which has seven attributes. In the preceding 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: $\text{dom}(\text{Name}) = \text{Names}$; $\text{dom}(\text{Ssn}) = \text{Social_security_numbers}$; $\text{dom}(\text{Home_phone}) = \text{USA_phone_numbers}$ ³, $\text{dom}(\text{Office_phone}) = \text{USA_phone_numbers}$, and $\text{dom}(\text{Gpa}) = \text{Grade_point_averages}$. 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.

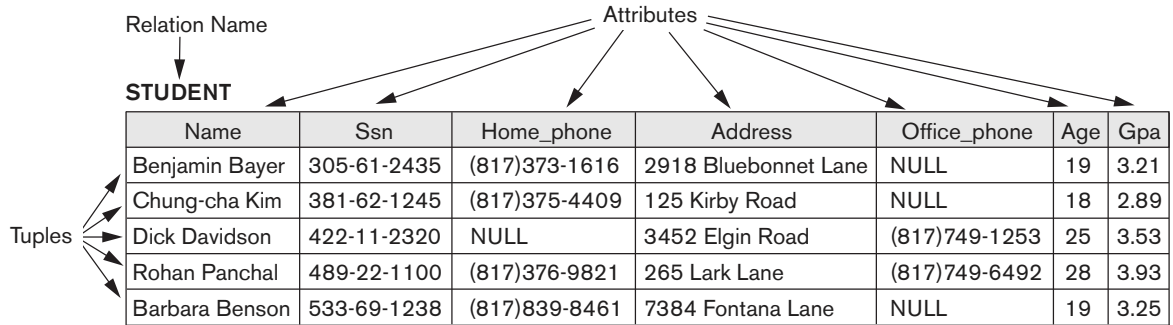
A **relation** (or **relation state**)⁴ r of the relation schema $R(A_1, A_2, \dots, A_n)$, also denoted by $r(R)$, is a set of n -tuples $r = \{t_1, t_2, \dots, t_m\}$. Each n -**tuple** t is an ordered list of n values $t = \langle v_1, v_2, \dots, v_n \rangle$, where each value v_i , $1 \leq i \leq n$, is an element of $\text{dom}(A_i)$ or is a special NULL value. (NULL values are discussed further below and in Section 5.1.2.) The i th value in tuple t , which corresponds to the attribute A_i , is referred to as $t[A_i]$ or $t.A_i$ (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 (or object). We 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.

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

³With the large increase in phone numbers caused by the proliferation of mobile phones, most metropolitan areas in the United States now have multiple area codes, so seven-digit local dialing has been discontinued in most areas. We changed this domain to `Usa_phone_numbers` instead of `Local_phone_numbers`, which would be a more general choice. This illustrates how database requirements can change over time.

⁴This 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.

**Figure 5.1**

The attributes and tuples of a relation STUDENT.

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

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

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

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

This product of cardinalities of all domains represents the total number of possible instances or tuples that can ever exist in any relation state $r(R)$. 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 state, by being transformed into another relation state. However, the schema R is relatively static and changes *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 attribute names indicate different **roles**, or interpretations, for the domain. For example, in the STUDENT relation, the same domain USA_phone_numbers plays the role of Home_phone, referring to the *home phone of a student*, and the role of Office_phone, referring to the *office phone of the student*. A third possible attribute (not shown) with the same domain could be Mobile_phone.

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. In other words, a relation is not sensitive to the ordering of tuples. 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 tuple orders can be specified on the same relation. For example, tuples in the STUDENT relation in Figure 5.1 could be ordered by values of Name, Ssn, Age, or some other attribute. The definition of a relation does not specify any order: There is *no preference* for one 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 more abstract 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 $R = \{A_1, A_2, \dots, A_n\}$ is a *set* of attributes (instead of an ordered list of attributes), and a relation state $r(R)$ is a finite set of mappings $r = \{t_1, t_2, \dots, t_m\}$, where each tuple t_i is a **mapping** from R to D , and D is the **union** (denoted by \cup) of the attribute domains; that is, $D = \text{dom}(A_1) \cup \text{dom}(A_2) \cup \dots \cup \text{dom}(A_n)$. 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

Figure 5.2

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

STUDENT

Name	Ssn	Home_phone	Address	Office_phone	Age	Gpa
Dick Davidson	422-11-2320	NULL	3452 Elgin Road	(817)749-1253	25	3.53
Barbara Benson	533-69-1238	(817)839-8461	7384 Fontana Lane	NULL	19	3.25
Rohan Panchal	489-22-1100	(817)376-9821	265 Lark Lane	(817)749-6492	28	3.93
Chung-cha Kim	381-62-1245	(817)375-4409	125 Kirby Road	NULL	18	2.89
Benjamin Bayer	305-61-2435	(817)373-1616	2918 Bluebonnet Lane	NULL	19	3.21

$t = \langle (\text{Name, Dick Davidson}), (\text{Ssn, 422-11-2320}), (\text{Home_phone, NULL}), (\text{Address, 3452 Elgin Road}), (\text{Office_phone, (817)749-1253}), (\text{Age, 25}), (\text{Gpa, 3.53}) \rangle$

$t = \langle (\text{Address, 3452 Elgin Road}), (\text{Name, Dick Davidson}), (\text{Ssn, 422-11-2320}), (\text{Age, 25}), (\text{Office_phone, (817)749-1253}), (\text{Gpa, 3.53}), (\text{Home_phone, NULL}) \rangle$

Figure 5.3

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

two tuples shown in Figure 5.3 are identical. This makes sense at an abstract level, since there really is no reason to prefer having one attribute value appear before another in a tuple. When the attribute name and value are included together in a tuple, it is known as **self-describing data**, because the description of each value (attribute name) is included in the tuple.

We will mostly use the **first definition** of relation, where the attributes are *ordered* in the relation schema and the values within tuples *are similarly 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 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 NULL values, 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 in 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 (also known as **value undefined**). 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 representing foreign students. It is possible to devise different codes for different meanings of

⁵We will use the alternative definition of relation when we discuss query processing and optimization in Chapter 18.

⁶We discuss this assumption in more detail in Chapter 14.

⁷Extensions of the relational model remove these restrictions. For example, object-relational systems (Chapter 12) allow complex-structured attributes, as do the **non-first normal form** or **nested** relational models.

NULL values. Incorporating different types of NULL values into relational model operations has proven difficult and is outside the scope of our presentation.

The exact meaning of a NULL value governs how it fares during arithmetic aggregations or comparisons with other values. For example, a comparison of two NULL values leads to ambiguities—if both Customer A and B have NULL addresses, it *does not mean* they have the same address. During database design, it is best to avoid NULL values as much as possible. We will discuss this further in Chapters 7 and 8 in the context of operations and queries, and in Chapter 14 in the context of database design and normalization.

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, Home_phone, Address, Office_phone, 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 (Student_ssn, Department_code) asserts that students major in academic disciplines. A tuple in this relation relates a student to his or her major discipline. 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. We introduced the entity–relationship (ER) model in detail in Chapter 3, where the entity and relationship concepts were described in detail. The mapping procedures in Chapter 9 show how different constructs of the ER/EER conceptual data models (see Part 2) 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. For example, the predicate STUDENT (Name, Ssn, ...) is true for the five tuples in relation STUDENT of Figure 5.1. These tuples represent five different propositions or facts in the real world. This interpretation is quite useful in the context of logical programming languages, such as Prolog, because it allows the relational model to be used within these languages (see Section 26.5). An assumption called **the closed world assumption** states that the only true facts in the universe are those present within the extension (state) of the relation(s). Any other combination of values makes the predicate false. This interpretation is useful when we consider queries on relations based on relational calculus in Section 8.6.

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 $R(A_1, A_2, \dots, A_n)$.

- The uppercase letters Q, R, S denote relation names.
- The lowercase 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.
- An n -tuple t in a relation $r(R)$ is denoted by $t = \langle v_1, v_2, \dots, v_n \rangle$, 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 $t[A_u, A_w, \dots, A_z]$ and $t.(A_u, A_w, \dots, A_z)$, where A_u, A_w, \dots, A_z is a list of attributes from R , refer to the subtuple of values $\langle v_u, v_w, \dots, v_z \rangle$ from t corresponding to the attributes specified in the list.

As an example, consider the tuple $t = \langle \text{'Barbara Benson'}, \text{'533-69-1238'}, \text{'(817)839-8461'}, \text{'7384 Fontana Lane'}, \text{NULL}, 19, 3.25 \rangle$ from the STUDENT relation in Figure 5.1; we have $t[\text{Name}] = \langle \text{'Barbara Benson'} \rangle$, and $t[\text{Ssn}, \text{Gpa}, \text{Age}] = \langle \text{'533-69-1238'}, 3.25, 19 \rangle$.

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** or **implicit 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** or **explicit 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 or in some other way. We call these **application-based** or **semantic constraints** or **business rules**.

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, relate to the meaning as well as behavior of attributes, and are difficult to express and enforce within the data model, so they are usually checked within the application programs that perform database updates. In some cases, these constraints can be specified as **assertions** in SQL (see Chapter 7).

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 14 and 15.

The schema-based constraints 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 $\text{dom}(A)$. 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 other special data types. Domains can also 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 relational standard in Section 6.1.

5.2.2 Key Constraints and Constraints on NULL Values

In the formal relational model, 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 properties:

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

Hence, a key is a superkey but not vice versa. A superkey may be a key (if it is minimal) or may not be a key (if it is not minimal). 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 together to have the uniqueness property.

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

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: License_number and Engine_serial_number. 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,

⁸Note that Ssn is also a superkey.

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

CAR

<u>License_number</u>	Engine_serial_number	Make	Model	Year
Texas ABC-739	A69352	Ford	Mustang	02
Florida TVP-347	B43696	Oldsmobile	Cutlass	05
New York MPO-22	X83554	Oldsmobile	Delta	01
California 432-TFY	C43742	Mercedes	190-D	99
California RSK-629	Y82935	Toyota	Camry	04
Texas RSK-629	U028365	Jaguar	XJS	04

Figure 5.4

The CAR relation, with two candidate keys: License_number and Engine_serial_number.

the choice of one to become the primary key is somewhat arbitrary; however, it is usually better to choose a primary key with a single attribute or a small number of attributes. The other candidate keys are designated as **unique keys** and are not underlined.

Another constraint on attributes specifies whether NULL values are or are not permitted. For example, if every STUDENT tuple must have a valid, non-NULL 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 $S = \{R_1, R_2, \dots, R_m\}$ and a set of **integrity constraints** IC. A **relational database state**¹⁰ DB of S is a set of relation states $DB = \{r_1, r_2, \dots, r_m\}$ 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 $COMPANY = \{EMPLOYEE, DEPARTMENT, DEPT_LOCATIONS, PROJECT, WORKS_ON, DEPENDENT\}$. In each relation schema, the underlined attribute represents the primary key. 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 4 through 6 for developing sample queries in different relational languages. (The data shown here is expanded and available for loading as a populated database from the Companion Website for the text, and can be used for the hands-on project exercises at the end of the chapters.)

When we refer to a relational database, we implicitly include both its schema and its current state. A database state that does not obey all the integrity constraints is

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

EMPLOYEE

Fname	Minit	Lname	<u>Ssn</u>	Bdate	Address	Sex	Salary	Super_ssn	Dno
-------	-------	-------	------------	-------	---------	-----	--------	-----------	-----

DEPARTMENT

Dname	<u>Dnumber</u>	Mgr_ssn	Mgr_start_date
-------	----------------	---------	----------------

DEPT_LOCATIONS

<u>Dnumber</u>	<u>Dlocation</u>
----------------	------------------

PROJECT

Pname	<u>Pnumber</u>	Plocation	Dnum
-------	----------------	-----------	------

WORKS_ON

<u>Essn</u>	<u>Pno</u>	Hours
-------------	------------	-------

DEPENDENT

<u>Essn</u>	<u>Dependent_name</u>	Sex	Bdate	Relationship
-------------	-----------------------	-----	-------	--------------

Figure 5.5

Schema diagram for the COMPANY relational database schema.

called **not valid**, and a state that satisfies all the constraints in the defined set of integrity constraints 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 SSN, and once in the role of the supervisor's SSN. We are required to give them distinct attribute names—Ssn and Super_ssn, respectively—because they appear in the same relation and in order to distinguish their meaning.

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 6.1 and 6.2.

Figure 5.6

One possible database state for the COMPANY relational database schema.

EMPLOYEE

Fname	Minit	Lname	Ssn	Bdate	Address	Sex	Salary	Super_ssn	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

DEPARTMENT

Dname	Dnumber	Mgr_ssn	Mgr_start_date
Research	5	333445555	1988-05-22
Administration	4	987654321	1995-01-01
Headquarters	1	888665555	1981-06-19

DEPT_LOCATIONS

Dnumber	Dlocation
1	Houston
4	Stafford
5	Bellaire
5	Sugarland
5	Houston

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

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 may not be able to distinguish them if we try 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, first we 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 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 $t_1[\text{FK}] = t_2[\text{PK}]$, 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, first we must have a clear understanding of the meaning or role that each attribute or 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 referencing 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 or will be assigned to a department later. For example, 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.

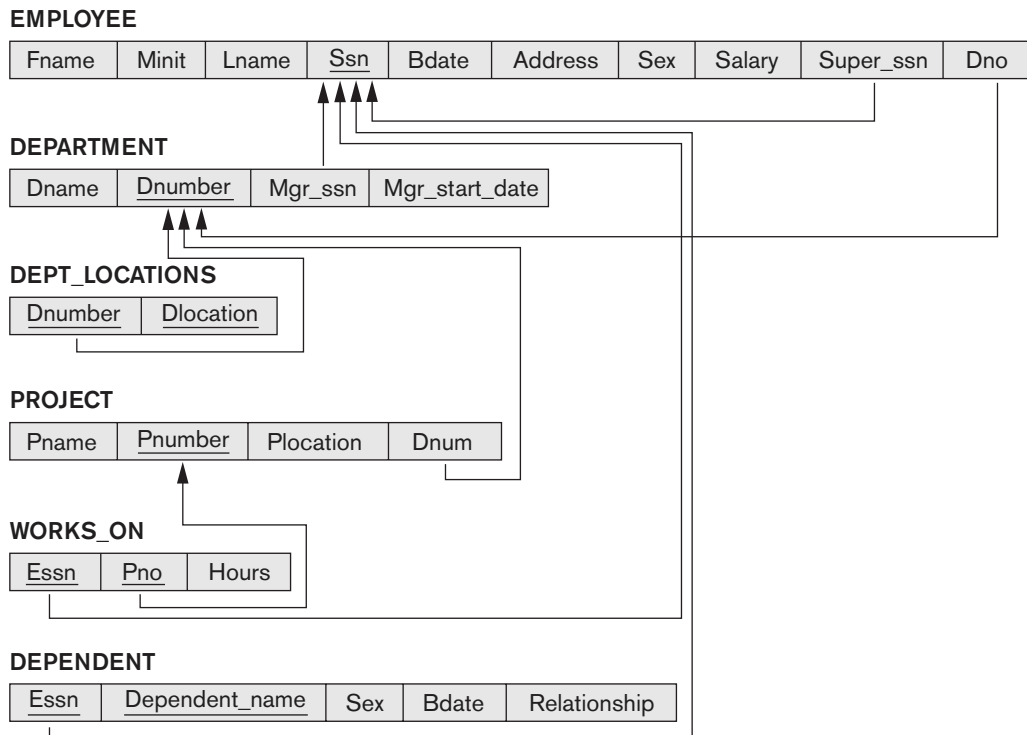
Notice that a foreign key can *refer to its own relation*. For example, the attribute Super_ssn in EMPLOYEE refers to the supervisor of an employee; this is another employee, represented by a tuple in the EMPLOYEE relation. Hence, Super_ssn is a foreign key that references the EMPLOYEE relation itself. 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 (that is, specified as part of its definition) if we want the DBMS to enforce these constraints on

Figure 5.7

Referential integrity constraints displayed on the COMPANY relational database schema.



the database states. Hence, the DDL includes provisions for specifying the various types of constraints so that the DBMS can automatically enforce them. In SQL, the CREATE TABLE statement of the SQL DDL allows the definition of primary key, unique key, NOT NULL, entity integrity, and referential integrity constraints, among other constraints (see Sections 6.1 and 6.2).

5.2.5 Other Types of Constraints

The preceding integrity constraints are included in the data definition language because they occur in most database applications. Another class of general constraints, sometimes called *semantic integrity constraints*, are not part of the DDL and have to be specified and enforced in a different way. 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, through the CREATE ASSERTION and CREATE TRIGGER statements, to specify some of these constraints (see Chapter 7). 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 sometimes difficult and complex to use, as we discuss in Section 26.1.

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 26.1.

5.3 Update Operations, Transactions, 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 8. 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 to retrieve information. 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. The **result relation** becomes the answer to (or result of) the user's query. Chapter 8 also introduces the language

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

called relational calculus, which is used to define a query declaratively without giving a specific order of operations.

In this section, we concentrate on the database **modification** or **update** operations. There are three basic operations that can change the states of relations in the database: Insert, Delete, and Update (or Modify). They insert new data, delete old data, or modify existing data records, respectively. **Insert** is used to insert one or more new 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 of these operations and the types of actions that may be taken if an operation causes a violation. We use the database shown in Figure 5.6 for examples and discuss only domain constraints, key constraints, entity integrity constraints, and the referential integrity constraints shown in Figure 5.7. For each type of operation, we give some examples 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. Domain constraints can be violated if an attribute value is given that does not appear in the corresponding domain or is not of the appropriate data type. 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 any part of 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.

- *Operation:*
Insert <'Cecilia', 'F', 'Kolonsky', NULL, '1960-04-05', '6357 Windy Lane, Katy, TX', F, 28000, NULL, 4> into EMPLOYEE.
Result: This insertion violates the entity integrity constraint (NULL for the primary key Ssn), so it is rejected.
- *Operation:*
Insert <'Alicia', 'J', 'Zelaya', '999887777', '1960-04-05', '6357 Windy Lane, Katy, TX', F, 28000, '987654321', 4> into EMPLOYEE.
Result: 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.
- *Operation:*
Insert <'Cecilia', 'F', 'Kolonsky', '677678989', '1960-04-05', '6357 Windswept, Katy, TX', F, 28000, '987654321', 7> into EMPLOYEE.
Result: This insertion violates the referential integrity constraint specified on Dno in EMPLOYEE because no corresponding referenced tuple exists in DEPARTMENT with Dnumber = 7.

- *Operation:*

Insert <'Cecilia', 'F', 'Kolonsky', '677678989', '1960-04-05', '6357 Windy Lane, Katy, TX', F, 28000, NULL, 4> into EMPLOYEE.

Result: 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 provide a reason to the user as to 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 the first operation, the DBMS could ask the user to provide a value for Ssn, and could then accept the insertion if a valid Ssn value is 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 Mgr_ssn that does not exist in the EMPLOYEE relation.

5.3.2 The Delete Operation

The **Delete** operation can violate only referential integrity. This occurs if the tuple being deleted is referenced by 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.

- *Operation:*

Delete the WORKS_ON tuple with Essn = '999887777' and Pno = 10.

Result: This deletion is acceptable and deletes exactly one tuple.

- *Operation:*

Delete the EMPLOYEE tuple with Ssn = '999887777'.

Result: This deletion is not acceptable, because there are tuples in WORKS_ON that refer to this tuple. Hence, if the tuple in EMPLOYEE is deleted, referential integrity violations will result.

- *Operation:*

Delete the EMPLOYEE tuple with Ssn = '333445555'.

Result: 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, called **restrict**, is to *reject the deletion*. The second option, called **cascade**, 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, called **set null** or **set default**, is to *modify the referencing attribute values* that cause the violation; each such value is either set to NULL or changed to

reference another default 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 Super_ssn = '333445555' and the tuple in DEPARTMENT with Mgr_ssn = '333445555' can have their Super_ssn and Mgr_ssn 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 database designer 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 DDL in Chapter 6.

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.

- *Operation:*
Update the salary of the EMPLOYEE tuple with Ssn = '999887777' to 28000.
Result: Acceptable.
- *Operation:*
Update the Dno of the EMPLOYEE tuple with Ssn = '999887777' to 1.
Result: Acceptable.
- *Operation:*
Update the Dno of the EMPLOYEE tuple with Ssn = '999887777' to 7.
Result: Unacceptable, because it violates referential integrity.
- *Operation:*
Update the Ssn of the EMPLOYEE tuple with Ssn = '999887777' to '987654321'.
Result: Unacceptable, because it violates primary key constraint by repeating a value that already exists as a primary key in another tuple; it violates referential integrity constraints because there are other relations that refer to the existing value of Ssn.

Updating an attribute that is *neither part of a primary key nor part of 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 set to 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 6.2).

5.3.4 The Transaction Concept

A database application program running against a relational database typically executes one or more *transactions*. A **transaction** is an executing program that includes some database operations, such as reading from the database, or applying insertions, deletions, or updates to the database. At the end of the transaction, it must leave the database in a valid or consistent state that satisfies all the constraints specified on the database schema. A single transaction may involve any number of retrieval operations (to be discussed as part of relational algebra and calculus in Chapter 8, and as a part of the language SQL in Chapters 6 and 7) and any number of update operations. These retrievals and updates will together form an atomic unit of work against the database. For example, a transaction to apply a bank withdrawal will typically read the user account record, check if there is a sufficient balance, and then update the record by the withdrawal amount.

A large number of commercial applications running against relational databases in **online transaction processing (OLTP)** systems are executing transactions at rates that reach several hundred per second. Transaction processing concepts, concurrent execution of transactions, and recovery from failures will be discussed in Chapters 20 to 22.

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. Then, we 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 a relation is not sensitive to the ordering of tuples. 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 ordering of attributes, but we continued to use the first definition, which requires attributes and tuple values to be ordered, for convenience. Then, we discussed values in tuples and introduced NULL values to represent missing or unknown information. We emphasized that NULL values should be avoided as much as possible.

We classified database constraints into inherent model-based constraints, explicit schema-based constraints, and semantic constraints or business rules. Then, we

discussed the schema constraints pertaining to the relational model, starting with domain constraints, then key constraints (including the concepts of superkey, key, and primary key), and the NOT NULL constraint on attributes. We defined relational databases and relational database schemas. Additional relational constraints include the entity integrity constraint, which prohibits primary key attributes from being NULL. We described the interrelation referential integrity constraint, which is used to maintain consistency of references among tuples from various relations.

The modification operations on the relational model are Insert, Delete, and Update. Each operation may violate certain types of constraints (refer to Section 5.3). Whenever an operation is applied, the resulting database state must be a valid state. Finally, we introduced the concept of a transaction, which is important in relational DBMSs because it allows the grouping of several database operations into a single atomic action on the database.

Review Questions

- 5.1. Define the following terms as they apply to the relational model of data: *domain*, *attribute*, *n-tuple*, *relation schema*, *relation state*, *degree of a relation*, *relational database schema*, and *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?
- 5.10. What is a transaction? How does it differ from an Update operation?

Exercises

- 5.11. 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', '1972-06-21', '2365 Newcastle Rd, Bellaire, TX', M, 58000, '888665555', 1> into EMPLOYEE.
- b. Insert <'ProductA', 4, 'Bellaire', 2> into PROJECT.
- c. Insert <'Production', 4, '943775543', '2007-10-01'> into DEPARTMENT.
- d. Insert <'677678989', NULL, '40.0'> into WORKS_ON.
- e. Insert <'453453453', 'John', 'M', '1990-12-12', '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 Mgr_ssn and Mgr_start_date of the DEPARTMENT tuple with Dnumber = 5 to '123456789' and '2007-10-01', respectively.
- j. Modify the Super_ssn 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'.

5.12. 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 FLIGHT_LEG has scheduled arrival and departure times, airports, and one or more LEG_INSTANCES—one for each Date on which the flight travels. FAREs are kept for each FLIGHT. For each FLIGHT_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 at 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.

5.13. Consider the relation CLASS(Course#, Univ_Section#, Instructor_name, Semester, Building_code, Room#, Time_period, Weekdays, Credit_hours). This represents classes taught in a university, with unique Univ_section#s. Identify what you think should be various candidate keys, and write in your own words the conditions or assumptions under which each candidate key would be valid.

AIRPORT

<u>Airport_code</u>	Name	City	State
---------------------	------	------	-------

FLIGHT

<u>Flight_number</u>	Airline	Weekdays
----------------------	---------	----------

FLIGHT_LEG

<u>Flight_number</u>	<u>Leg_number</u>	Departure_airport_code	Scheduled_departure_time
		Arrival_airport_code	Scheduled_arrival_time

LEG_INSTANCE

<u>Flight_number</u>	<u>Leg_number</u>	<u>Date</u>	Number_of_available_seats	Airplane_id
		Departure_airport_code	Departure_time	Arrival_airport_code
				Arrival_time

FARE

<u>Flight_number</u>	<u>Fare_code</u>	Amount	Restrictions
----------------------	------------------	--------	--------------

AIRPLANE_TYPE

<u>Airplane_type_name</u>	Max_seats	Company
---------------------------	-----------	---------

CAN_LAND

<u>Airplane_type_name</u>	<u>Airport_code</u>
---------------------------	---------------------

AIRPLANE

<u>Airplane_id</u>	Total_number_of_seats	Airplane_type
--------------------	-----------------------	---------------

SEAT_RESERVATION

<u>Flight_number</u>	<u>Leg_number</u>	<u>Date</u>	<u>Seat_number</u>	Customer_name	Customer_phone
----------------------	-------------------	-------------	--------------------	---------------	----------------

Figure 5.8

The AIRLINE relational database schema.

- 5.14. 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)

Here, Ord_amt refers to total dollar amount of an order; Odate is the date the order was placed; and Ship_date is the date an order (or part of 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. What other constraints can you think of for this database?

- 5.15.** 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)

A trip can be charged to one or more accounts. 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 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.

- 5.17.** Consider the following relations for a database that keeps track of automobile sales in a car dealership (OPTION refers to some optional equipment installed on an automobile):

CAR(Serial_no, Model, Manufacturer, Price)
 OPTION(Serial_no, Option_name, Price)
 SALE(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 sample tuples, and then give an example of an insertion in the SALE and SALESPERSON relations that *violates* the referential integrity constraints and of another insertion that does not.

- 5.18.** Database design often involves decisions about the storage of attributes. For example, a Social Security number can be stored as one attribute or split into three attributes (one for each of the three hyphen-delineated groups of

numbers in a Social Security number—XXX-XX-XXXX). However, Social Security numbers are usually represented as just one attribute. The decision is based on how the database will be used. This exercise asks you to think about specific situations where dividing the SSN is useful.

- 5.19.** Consider a STUDENT relation in a UNIVERSITY database with the following attributes (Name, Ssn, Local_phone, Address, Cell_phone, Age, Gpa). Note that the cell phone may be from a different city and state (or province) from the local phone. A possible tuple of the relation is shown below:

Name	Ssn	Local_phone	Address	Cell_phone	Age	Gpa
George Shaw	123-45-6789	555-1234	123 Main St.,	555-4321	19	3.75
William Edwards			Anytown, CA 94539			

- Identify the critical missing information from the Local_phone and Cell_phone attributes. (*Hint*: How do you call someone who lives in a different state or province?)
 - Would you store this additional information in the Local_phone and Cell_phone attributes or add new attributes to the schema for STUDENT?
 - Consider the Name attribute. What are the advantages and disadvantages of splitting this field from one attribute into three attributes (first name, middle name, and last name)?
 - What general guideline would you recommend for deciding when to store information in a single attribute and when to split the information?
 - Suppose the student can have between 0 and 5 phones. Suggest two different designs that allow this type of information.
- 5.20.** Recent changes in privacy laws have disallowed organizations from using Social Security numbers to identify individuals unless certain restrictions are satisfied. As a result, most U.S. universities cannot use SSNs as primary keys (except for financial data). In practice, Student_id, a unique identifier assigned to every student, is likely to be used as the primary key rather than SSN since Student_id can be used throughout the system.
- Some database designers are reluctant to use generated keys (also known as *surrogate keys*) for primary keys (such as Student_id) because they are artificial. Can you propose any natural choices of keys that can be used to identify the student record in a UNIVERSITY database?
 - Suppose that you are able to guarantee uniqueness of a natural key that includes last name. Are you guaranteed that the last name will not change during the lifetime of the database? If last name can change, what solutions can you propose for creating a primary key that still includes last name but remains unique?
 - What are the advantages and disadvantages of using generated (surrogate) keys?

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 (Association for Computing Machinery) 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. Date (2001) provides a retrospective review and analysis of the relational data model.

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) introduce 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) introduce various types of connections between relations to enhance its constraints. Extensions of the relational model are discussed in Chapters 11 and 26. Additional bibliographic notes for other aspects of the relational model and its languages, systems, extensions, and theory are given in Chapters 6 to 9, 14, 15, 23, and 30. Maier (1983) and Atzeni and De Antonellis (1993) provide an extensive theoretical treatment of the relational data model.

This page intentionally left blank

Basic SQL

The SQL language may be considered one of the major reasons for the commercial success of relational databases. 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, older network or hierarchical systems—to relational systems. This is because even if the users became dissatisfied with the particular relational DBMS product they were using, converting to another relational DBMS product was not expected to be too expensive and time-consuming because both systems followed the same language standards. In practice, of course, there are differences among various commercial relational DBMS packages. However, if the user is diligent in using only those features that are part of the standard, and if two relational DBMSs faithfully support the standard, then conversion between two systems should be 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), as long as both/all of the relational DBMSs support standard SQL.

This chapter presents the *practical* relational model, which is based on the SQL standard for *commercial* relational DBMSs, whereas Chapter 5 presented the most important concepts underlying the *formal* relational data model. In Chapter 8 (Sections 8.1 through 8.5), we shall discuss 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 18 and 19. However, the relational algebra operations are too low-level 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 describe in Section 8.6. However, the SQL syntax is more user-friendly than either of the two formal languages.

The name **SQL** is presently expanded as Structured Query Language. Originally, SQL was called SEQUEL (Structured English QUery Language) 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. The standardization of SQL is a joint effort by the American National Standards Institute (ANSI) and the International Standards Organization (ISO), and the first SQL standard is called SQL-86 or SQL1. A revised and much expanded standard called SQL-92 (also referred to as SQL2) was subsequently developed. The next standard that is well-recognized is SQL:1999, which started out as SQL3. Additional updates to the standard are SQL:2003 and SQL:2006, which added XML features (see Chapter 13) among other updates to the language. Another update in 2008 incorporated more object database features into SQL (see Chapter 12), and a further update is SQL:2011. We will try to cover the latest version of SQL as much as possible, but some of the newer features are discussed in later chapters. It is also not possible to cover the language in its entirety in this text. It is important to note that when new features are added to SQL, it usually takes a few years for some of these features to make it into the commercial SQL DBMSs.

SQL is a comprehensive database language: It has statements for data definitions, queries, and updates. 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 C/C++.¹

The later SQL standards (starting with **SQL:1999**) are divided into a **core** specification plus specialized **extensions**. The core is supposed to be implemented by all RDBMS vendors that are SQL compliant. The extensions can be implemented as optional modules to be purchased independently for specific database applications such as data mining, spatial data, temporal data, data warehousing, online analytical processing (OLAP), multimedia data, and so on.

Because the subject of SQL is both important and extensive, we devote two chapters to its basic features. In this chapter, Section 6.1 describes the SQL DDL commands for creating schemas and tables, and gives an overview of the basic data types in SQL. Section 6.2 presents how basic constraints such as key and referential integrity are specified. Section 6.3 describes the basic SQL constructs for

¹Originally, 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.

specifying retrieval queries, and Section 6.4 describes the SQL commands for insertion, deletion, and update.

In Chapter 7, we will describe more complex SQL retrieval queries, as well as the ALTER commands for changing the schema. We will also describe the CREATE ASSERTION statement, which allows the specification of more general constraints on the database, and the concept of triggers, which is presented in more detail in Chapter 26. We discuss the SQL facility for defining views on the database in Chapter 7. 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.

Section 6.5 lists some SQL features that are presented in other chapters of the book; these include object-oriented features in Chapter 12, XML in Chapter 13, transaction control in Chapter 20, active databases (triggers) in Chapter 26, online analytical processing (OLAP) features in Chapter 29, and security/authorization in Chapter 30. Section 6.6 summarizes the chapter. Chapters 10 and 11 discuss the various database programming techniques for programming with SQL.

6.1 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), types, 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 6.1.1 to place our discussion in perspective. Section 6.1.2 describes how tables are created, and Section 6.1.3 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 end-of-chapter bibliographic notes).

6.1.1 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 (in some systems, a *schema* is called a *database*). 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, types, 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'`. Note that each statement in SQL ends with a semicolon.

```
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, SQL uses the concept of a **catalog**—a named collection of schemas.² Database installations typically have a default environment and schema, so when a user connects and logs in to that database installation, the user can refer directly to tables and other constructs within that schema without having to specify a particular schema name. 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 type and domain definitions.

6.1.2 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 possibly 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 Chapter 7). Figure 6.1 shows sample data definition statements in SQL for the `COMPANY` relational database schema shown in Figure 3.7.

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 6.1, 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 table and its rows are actually created

²SQL also includes the concept of a *cluster* of catalogs.

```

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),
  Super_ssn      CHAR(9),
  Dno            INT                  NOT NULL,
  PRIMARY KEY (Ssn),
CREATE TABLE DEPARTMENT
( Dname          VARCHAR(15)          NOT NULL,
  Dnumber        INT                  NOT NULL,
  Mgr_ssn        CHAR(9)              NOT NULL,
  Mgr_start_date DATE,
  PRIMARY KEY (Dnumber),
  UNIQUE (Dname),
  FOREIGN KEY (Mgr_ssn) 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 6.1
SQL CREATE
TABLE data
definition statements
for defining the
COMPANY schema
from Figure 5.7.

and stored as a file by the DBMS. Base relations are distinguished from **virtual relations**, created through the CREATE VIEW statement (see Chapter 7), 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 table (relation).

It is important to note that in Figure 6.1, there are some *foreign keys that may cause errors* because they are specified either via circular references or because they refer to a table that has not yet been created. For example, the foreign key Super_ssn in the EMPLOYEE table is a circular reference because it refers to the EMPLOYEE table itself. The foreign key Dno in the EMPLOYEE table refers to the DEPARTMENT table, which has not been created yet. To deal with this type of problem, these constraints can be left out of the initial CREATE TABLE statement, and then added later using the ALTER TABLE statement (see Chapter 7). We displayed all the foreign keys in Figure 6.1 to show the complete COMPANY schema in one place.

6.1.3 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

³This 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.

⁴For nonalphabetic characters, there is a defined order.

in SQL. For example, 'abc' || 'XYZ' results in a single string 'abcXYZ'. Another variable-length string data type called CHARACTER LARGE OBJECT or CLOB is also available to specify columns that have large text values, such as documents. The CLOB maximum length can be specified in kilobytes (K), megabytes (M), or gigabytes (G). For example, CLOB(20M) specifies a maximum length of 20 megabytes.

- **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 Another variable-length bitstring data type called BINARY LARGE OBJECT or BLOB is also available to specify columns that have large binary values, such as images. As for CLOB, the maximum length of a BLOB can be specified in kilobits (K), megabits (M), or gigabits (G). For example, BLOB(30G) specifies a maximum length of 30 gigabits.
- 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 Chapter 7.
- 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 the SQL implementation. This implies that months should be between 1 and 12 and days must be between 01 and 31; furthermore, a day should be a valid day for the corresponding month. 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 '2014-09-27' 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 period (.) 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.

Some additional data types are discussed below. The list of types discussed here is not exhaustive; different implementations have added more data types to SQL.

- A **timestamp** data type (TIMESTAMP) includes 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

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

strings preceded by the keyword `TIMESTAMP`, with a blank space between data and time; for example, `TIMESTAMP '2014-09-27 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 6.1; alternatively, a domain can be declared, and the domain name can be 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 6.1 for the attributes `Ssn` and `Super_ssn` of `EMPLOYEE`, `Mgr_ssn` 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. Notice that domains may not be available in some implementations of SQL.

In SQL, there is also a **CREATE TYPE** command, which can be used to create user defined types or UDTs. These can then be used either as data types for attributes, or as the basis for creating tables. We shall discuss `CREATE TYPE` in detail in Chapter 12, because it is often used in conjunction with specifying object database features that have been incorporated into more recent versions of SQL.

6.2 Specifying Constraints in SQL

This section describes the basic constraints that can be specified in SQL as part of table creation. These include key and referential integrity constraints, restrictions on attribute domains and `NULL`s, and constraints on individual tuples within a relation using the `CHECK` clause. We discuss the specification of more general constraints, called assertions, in Chapter 7.

6.2.1 Specifying Attribute Constraints and Attribute Defaults

Because SQL allows `NULL`s 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 6.1.

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

```

CREATE TABLE EMPLOYEE
(
    ...,
    Dno          INT          NOT NULL          DEFAULT 1,
    CONSTRAINT EMPCHK
    PRIMARY KEY (Ssn),
    CONSTRAINT EMPSUPERFK
    FOREIGN KEY (Super_ssn) 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
(
    ...,
    Mgr_ssn CHAR(9)          NOT NULL          DEFAULT '888665555',
    ...,
    CONSTRAINT DEPTPK
    PRIMARY KEY(Dnumber),
    CONSTRAINT DEPTSK
    UNIQUE (Dname),
    CONSTRAINT DEPTMGRFK
    FOREIGN KEY (Mgr_ssn) 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 6.2

Example illustrating how default attribute values and referential integrity triggered actions are specified in SQL.

new tuple if an explicit value is not provided for that attribute. Figure 6.2 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 6.1) 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);

```

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

We can then use the created domain `D_NUM` as the attribute type for all attributes that refer to department numbers in Figure 6.1, such as `Dnumber` of `DEPARTMENT`, `Dnum` of `PROJECT`, `Dno` of `EMPLOYEE`, and so on.

6.2.2 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 6.1.⁷ 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, the primary key of `DEPARTMENT` can be specified as follows (instead of the way it is specified in Figure 6.1):

```
Dnumber INT PRIMARY KEY,
```

The **UNIQUE** clause specifies alternate (unique) keys, also known as candidate keys as illustrated in the `DEPARTMENT` and `PROJECT` table declarations in Figure 6.1. The **UNIQUE** clause can also be specified directly for a unique key if it is a single attribute, as in the following example:

```
Dname VARCHAR(15) UNIQUE,
```

Referential integrity is specified via the **FOREIGN KEY** clause, as shown in Figure 6.1. 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 updated. The default action that SQL takes for an integrity violation is to **reject** the update operation that will cause a violation, which is known as the **RESTRICT** option. However, the schema designer can specify an alternative action to be taken 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 6.2. Here, the database designer chooses `ON DELETE SET NULL` and `ON UPDATE CASCADE` for the foreign key `Super_ssn` of `EMPLOYEE`. This means that if the tuple for a *supervising employee* is *deleted*, the value of `Super_ssn` 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 `Super_ssn` for all employee tuples referencing the updated employee tuple.⁸

In general, the action taken by the DBMS for `SET NULL` or `SET DEFAULT` is the same for both `ON DELETE` and `ON UPDATE`: The value of the affected referencing attributes is changed to `NULL` for `SET NULL` and to the specified default value of the

⁷Key and referential integrity constraints were not included in early versions of SQL.

⁸Notice that the foreign key `Super_ssn` in the `EMPLOYEE` table is a circular reference and hence may have to be added later as a named constraint using the `ALTER TABLE` statement as we discussed at the end of Section 6.1.2.

referencing attribute 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 referencing foreign key attribute(s) to the updated (new) primary key value for all the 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 9.1), 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.

6.2.3 Giving Names to Constraints

Figure 6.2 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 Chapter 7. Giving names to constraints is optional. It is also possible to temporarily *defer* a constraint until the end of a transaction, as we shall discuss in Chapter 20 when we present transaction concepts.

6.2.4 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 **row-based** constraints because they apply to each row *individually* and are checked whenever a row is inserted or modified. For example, suppose that the DEPARTMENT table in Figure 6.1 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 <= Mgr_start_date);
```

The CHECK clause can also be used to specify more general constraints using the CREATE ASSERTION statement of SQL. We discuss this in Chapter 7 because it requires the full power of queries, which are discussed in Sections 6.3 and 7.1.

6.3 Basic Retrieval Queries in SQL

SQL has one basic statement for retrieving information from a database: the **SELECT** statement. The SELECT statement is *not the same as* the SELECT operation of relational algebra, which we shall discuss in Chapter 8. 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 these queries. In this section, we present the features of SQL for *simple retrieval queries*. Features of SQL for specifying more complex retrieval queries are presented in Section 7.1.

Before proceeding, we must point out an *important distinction* between the practical SQL model 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.

6.3.1 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:⁹

```
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 syntactic difference is the *not equal* operator. SQL has additional comparison operators that we will present gradually.

We illustrate the basic **SELECT** statement in SQL with some sample queries. The queries are labeled here with the same query numbers used in Chapter 8 for easy cross-reference.

⁹The **SELECT** and **FROM** clauses are required in all SQL queries. The **WHERE** is optional (see Section 6.3.3).

Query 0. Retrieve the birth date 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 individual 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.

The SELECT clause of SQL specifies the attributes whose values are to be retrieved, which are called the **projection attributes** in relational algebra (see Chapter 8) and the WHERE clause specifies the Boolean condition that must be true for any retrieved tuple, which is known as the **selection condition** in relational algebra. Figure 6.3(a) shows the result of query Q0 on the database of Figure 5.6.

We can think of an implicit **tuple variable** or *iterator* in the SQL query ranging or *looping* over each individual 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;
```

In the WHERE clause of Q1, the condition Dname = ‘Research’ is a **selection condition** that chooses the particular tuple of interest in the DEPARTMENT table, because Dname is an attribute of DEPARTMENT. The condition Dnumber = Dno is called a **join condition**, because it combines two tuples: one from DEPARTMENT and one from EMPLOYEE, whenever the value of Dnumber in DEPARTMENT is equal to the value of Dno in EMPLOYEE. The result of query Q1 is shown in Figure 6.3(b). In general, any number of selection and join conditions may be specified in a single SQL query.

A query that involves only selection and join conditions plus projection attributes is known as a **select-project-join** 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 birth date.

```
Q2:    SELECT    Pnumber, Dnum, Lname, Address, Bdate
        FROM      PROJECT, DEPARTMENT, EMPLOYEE
        WHERE     Dnum = Dnumber AND Mgr_ssn = Ssn AND
                  Plocation = ‘Stafford’
```

Figure 6.3

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.

(a)

<u>Bdate</u>	<u>Address</u>
1965-01-09	731Fondren, Houston, TX

(b)

<u>Fname</u>	<u>Lname</u>	<u>Address</u>
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)

<u>Pnumber</u>	<u>Dnum</u>	<u>Lname</u>	<u>Address</u>	<u>Bdate</u>
10	4	Wallace	291Berry, Bellaire, TX	1941-06-20
30	4	Wallace	291Berry, Bellaire, TX	1941-06-20

(d)

<u>E.Fname</u>	<u>E.Lname</u>	<u>S.Fname</u>	<u>S.Lname</u>
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)

<u>E.Fname</u>
123456789
333445555
999887777
987654321
666884444
453453453
987987987
888665555

(f)

<u>Ssn</u>	<u>Dname</u>
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)

<u>Fname</u>	<u>Minit</u>	<u>Lname</u>	<u>Ssn</u>	<u>Bdate</u>	<u>Address</u>	<u>Sex</u>	<u>Salary</u>	<u>Super_ssn</u>	<u>Dno</u>
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

The join condition $Dnum = Dnumber$ relates a project tuple to its controlling department tuple, whereas the join condition $Mgr_ssn = Ssn$ relates the controlling department tuple to the employee tuple who manages that department. Each tuple in the result will be a *combination* of one project, one department (that controls the project), and one employee (that manages the department). The projection attributes are used to choose the attributes to be displayed from each combined tuple. The result of query Q2 is shown in Figure 6.3(c).

6.3.2 Ambiguous Attribute Names, Aliasing, Renaming, and Tuple Variables

In SQL, the same name can be used for two (or more) attributes as long as the attributes are in *different tables*. If this is the case, and a multitable 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 same 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;
```

Fully qualified attribute names can be used for clarity even if there is no ambiguity in attribute names. Q1 can be rewritten as Q1' below with fully qualified attribute names. We can also rename the table names to shorter names by creating an *alias* for each table name to avoid repeated typing of long table names (see Q8 below).

```

Q1':   SELECT   EMPLOYEE.Fname, EMPLOYEE.LName,
                EMPLOYEE.Address
        FROM     EMPLOYEE, DEPARTMENT
        WHERE    DEPARTMENT.DName = 'Research' AND
                DEPARTMENT.Dnumber = EMPLOYEE.Dno;
```

The ambiguity of attribute names 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.Super_ssn = S.Ssn;
```

In this case, we are required 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 or subordinates; 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.Super_ssn = S.Ssn. Notice that this is an example of a one-level recursive query, as we will discuss in Section 8.4.2. In earlier versions of SQL, 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:1999 (see Chapter 7).

The result of query Q8 is shown in Figure 6.3(d). Whenever one or more aliases are given to a relation, we can use these names to represent different references to that same relation. This permits multiple references to the same relation within a query.

We can use this alias-naming or **renaming** 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 Q1 as in Q1B:

```
Q1B:  SELECT    E.Fname, E.LName, E.Address
      FROM      EMPLOYEE AS E, DEPARTMENT AS D
      WHERE     D.DName = 'Research' AND D.Dnumber = E.Dno;
```

6.3.3 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 6.3(e)), and Query 10 selects all combinations of an EMPLOYEE Ssn and a DEPARTMENT Dname, regardless of whether the employee works for the department or not (Figure 6.3(f)).

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 (see Chapter 8). If we specify all the attributes of EMPLOYEE and DEPARTMENT in Q10, we get the actual 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*. The * can also be prefixed by the relation name or alias; for example, EMPLOYEE.* refers to all attributes of the EMPLOYEE table.

Query Q1C retrieves all the attribute values of any EMPLOYEE who works in DEPARTMENT number 5 (Figure 6.3(g)), 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;

```

6.3.4 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 7.1.7) is applied to tuples, in most cases we do not want to eliminate duplicates.

Figure 6.4

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.

(a)	<table><tr><th>Salary</th></tr><tr><td>30000</td></tr><tr><td>40000</td></tr><tr><td>25000</td></tr><tr><td>43000</td></tr><tr><td>38000</td></tr><tr><td>25000</td></tr><tr><td>25000</td></tr><tr><td>55000</td></tr></table>	Salary	30000	40000	25000	43000	38000	25000	25000	55000	(b)	<table><tr><th>Salary</th></tr><tr><td>30000</td></tr><tr><td>40000</td></tr><tr><td>25000</td></tr><tr><td>43000</td></tr><tr><td>38000</td></tr><tr><td>55000</td></tr></table>	Salary	30000	40000	25000	43000	38000	55000	(c)	<table><tr><th>Fname</th><th>Lname</th></tr><tr><td></td><td></td></tr></table>	Fname	Lname		
Salary																									
30000																									
40000																									
25000																									
43000																									
38000																									
25000																									
25000																									
55000																									
Salary																									
30000																									
40000																									
25000																									
43000																									
38000																									
55000																									
Fname	Lname																								
				(d)	<table><tr><th>Fname</th><th>Lname</th></tr><tr><td>James</td><td>Borg</td></tr></table>	Fname	Lname	James	Borg																
Fname	Lname																								
James	Borg																								

An SQL table with a key is restricted to being a set, since the key value must be distinct in each tuple.¹⁰ 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 example, Q11 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 6.4(a). 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 6.4(b).

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 from mathematical *set theory*, which are also part of relational algebra (see Chapter 8). There are set union (**UNION**), set difference (**EXCEPT**),¹¹ 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*. These set operations apply only to *type-compatible relations*, so 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.

¹⁰In general, an SQL table is not required to have a key, although in most cases there will be one.

¹¹In some systems, the keyword MINUS is used for the set difference operation instead of EXCEPT.

(a)

R
A
a1
a2
a2
a3

(b)

T
A
a1
a1
a2
a2
a2
a3
a4
a5

(c)

T
A
a2
a3

(d)

T
A
a1
a2

Figure 6.5

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).

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.

```
Q4A:  ( SELECT  DISTINCT Pnumber
      FROM    PROJECT, DEPARTMENT, EMPLOYEE
      WHERE   Dnum = Dnumber AND Mgr_ssn = 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 6.5. Basically, each tuple—whether it is a duplicate or not—is considered as a different tuple when applying these operations.

6.3.5 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 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 1970s, we can use Query Q12A. Here, '7' 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.

```
Q12:  SELECT    Fname, Lname
        FROM      EMPLOYEE
        WHERE     Bdate LIKE '  7  _ _ _ _ _ _ _ _';
```

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. Notice that substring comparison implies that attribute values are not atomic (indivisible) values, as we had assumed in the formal relational model (see Section 5.1).

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% 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% raise.

```
Q13:  SELECT    E.Fname, E.Lname, 1.1 * E.Salary AS Increased_sal
        FROM      EMPLOYEE AS E, WORKS_ON AS W, PROJECT AS P
        WHERE     E.Ssn = W.Essn AND W.Pno = P.Pnumber AND
                  P.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, which 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)).

6.3.6 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 of the attributes that appear in the query result, by 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, then first name.

```
Q15:  SELECT  D.Dname, E.Lname, E.Fname, P.Pname
      FROM    DEPARTMENT AS D, EMPLOYEE AS E, WORKS_ON AS W,
              PROJECT AS P
      WHERE   D.Dnumber = E.Dno AND E.Ssn = W.Essn AND W.Pno =
              P.Pnumber
      ORDER BY D.Dname, E.Lname, E.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 alphabetical order on Dname and ascending order on Lname, Fname, the **ORDER BY** clause of Q15 can be written as

```
ORDER BY D.Dname DESC, E.Lname ASC, E.Fname ASC
```

6.3.7 Discussion and Summary of Basic SQL Retrieval Queries

A *simple* retrieval query in SQL can consist of up to four 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 list>
FROM    <table list>
[ WHERE <condition> ]
[ ORDER BY <attribute list> ];
```

The **SELECT** clause lists the attributes to be retrieved, and the **FROM** clause specifies all relations (tables) needed in the simple query. The **WHERE** clause identifies the conditions for selecting the tuples from these relations, including

join conditions if needed. **ORDER BY** specifies an order for displaying the results of a query. Two additional clauses **GROUP BY** and **HAVING** will be described in Section 7.1.8.

In Chapter 7, we will present more complex features of SQL retrieval queries. These include the following: nested queries that allow one query to be included as part of another query; aggregate functions that are used to provide summaries of the information in the tables; two additional clauses (**GROUP BY** and **HAVING**) that can be used to provide additional power to aggregate functions; and various types of joins that can combine records from various tables in different ways.

6.4 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.

6.4.1 The INSERT Command

In its simplest form, **INSERT** is used to add a single tuple (row) to a relation (table). 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 6.1, we can use U1:

```
U1:  INSERT INTO  EMPLOYEE
      VALUES      ('Richard', 'K', 'Marini', '653298653', '1962-12-30', '98
                  Oak Forest, Katy, TX', 'M', 37000, '653298653', 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 should support and enforce all the integrity constraints that can be specified in the DDL. For example, if we issue the command in U2 on the database shown in Figure 5.6, the DBMS should *reject* the operation because no DEPARTMENT tuple exists in the database with Dnumber = 2. Similarly, U2A would be *rejected* because no Ssn value is provided and it is the primary key, which cannot be NULL.

```

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 employee last name, project name, and hours per week for each employee working on a project, we can write the statements in U3A and U3B:

```

U3A:     CREATE TABLE      WORKS_ON_INFO
          ( Emp_name          VARCHAR(15),
            Proj_name         VARCHAR(15),
            Hours_per_week    DECIMAL(3,1) );

U3B:     INSERT INTO      WORKS_ON_INFO ( Emp_name, Proj_name,
          Hours_per_week )
SELECT      E.Lname, P.Pname, W.Hours
FROM        PROJECT P, WORKS_ON W, EMPLOYEE E
WHERE        P.Pnumber = W.Pno AND W.Essn = E.Ssn;

```

A table WORKS_ON_INFO is created by U3A and is loaded with the joined information retrieved from the database by the query in U3B. We can now query WORKS_ON_INFO as we would any other relation; when we do not need it anymore, we can remove it by using the DROP TABLE command (see Chapter 7). Notice that the WORKS_ON_INFO table may not be up to date; that is, if we update any of the PROJECT, WORKS_ON, or EMPLOYEE relations after issuing U3B, the information in WORKS_ON_INFO *may become outdated*. We have to create a view (see Chapter 7) to keep such a table up to date.

Most DBMSs have *bulk loading* tools that allow a user to load formatted data from a file into a table without having to write a large number of INSERT commands. The user can also write a program to read each record in the file, format it as a row in the table, and insert it using the looping constructs of a programming language (see Chapters 10 and 11, where we discuss database programming techniques).

Another variation for loading data is to create a new table TNEW that has the same attributes as an existing table T, and load some of the data currently in T into TNEW. The syntax for doing this uses the LIKE clause. For example, if we

want to create a table D5EMPS with a similar structure to the EMPLOYEE table and load it with the rows of employees who work in department 5, we can write the following SQL:

```
CREATE TABLE      D5EMPS LIKE EMPLOYEE
(SELECT           E.*
FROM             EMPLOYEE AS E
WHERE            E.Dno = 5) WITH DATA;
```

The clause WITH DATA specifies that the table will be created and loaded with the data specified in the query, although in some implementations it may be left out.

6.4.2 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 6.2.2).¹² 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. We must use the DROP TABLE command to remove the table definition (see Chapter 7). The DELETE commands in U4A to U4D, if applied independently to the database state shown in 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 = 5;
U4D:  DELETE FROM      EMPLOYEE;
```

6.4.3 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

¹²Other actions can be automatically applied through triggers (see Section 26.1) and other mechanisms.

constraints of the DDL (see Section 6.2.2). An additional **SET** clause in the UPDATE command specifies the attributes to be modified and 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% 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 = 5;
```

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.

6.5 Additional Features of SQL

SQL has a number of additional features that we have not described in this chapter but that we discuss elsewhere in the book. These are as follows:

- In Chapter 7, which is a continuation of this chapter, we will present the following SQL features: various techniques for specifying complex retrieval queries, including nested queries, aggregate functions, grouping, joined tables, outer joins, case statements, and recursive queries; SQL views, triggers, and assertions; and commands for schema modification.
- SQL has various techniques for writing programs in various programming languages that include SQL statements to access one or more databases. These include embedded (and dynamic) SQL, SQL/CLI (Call Level Interface) and its predecessor ODBC (Open Data Base Connectivity), and SQL/PSM (Persistent Stored Modules). We discuss these techniques in Chapter 10. We also describe 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 language (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. Many systems still have the `CREATE INDEX` commands; but they require a special privilege. We describe this in Chapter 17.

- 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 20 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 20, 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 26.1, 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, 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 12.
- SQL and relational databases can interact with new technologies such as XML (see Chapter 13) and OLAP/data warehouses (Chapter 29).

6.6 Summary

In this chapter, we introduced the SQL database language. This language and its variations have been implemented as interfaces to many commercial relational DBMSs, including Oracle's Oracle; ibm's DB2; Microsoft's SQL Server; and many other systems including Sybase and INGRES. Some open source systems also provide SQL, such as MySQL and PostgreSQL. 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, constraint specification, and view definition. We discussed the following features of SQL in this chapter: the data definition commands for creating tables, SQL basic data types, commands for constraint specification, simple retrieval queries, and database update commands. In the next chapter, we will present the following features of SQL: complex retrieval queries; views; triggers and assertions; and schema modification commands.

Review Questions

- 6.1. How do the relations (tables) in SQL differ from the relations defined formally in Chapter 3? Discuss the other differences in terminology. Why does SQL allow duplicate tuples in a table or in a query result?
- 6.2. List the data types that are allowed for SQL attributes.
- 6.3. How does SQL allow implementation of the entity integrity and referential integrity constraints described in Chapter 3? What about referential triggered actions?
- 6.4. Describe the four clauses in the syntax of a simple SQL retrieval query. Show what type of constructs can be specified in each of the clauses. Which are required and which are optional?

Exercises

- 6.5. 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.
- 6.6. Repeat Exercise 6.5, but use the AIRLINE database schema of Figure 5.8.
- 6.7. Consider the LIBRARY relational database schema shown in Figure 6.6. Choose the appropriate action (reject, cascade, set to NULL, set to default) for each referential integrity constraint, both for the *deletion* of a referenced tuple and for the *update* of a primary key attribute value in a referenced tuple. Justify your choices.
- 6.8. Write appropriate SQL DDL statements for declaring the LIBRARY relational database schema of Figure 6.6. Specify the keys and referential triggered actions.
- 6.9. 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?
- 6.10. Specify the following queries in SQL on the COMPANY relational database schema shown in Figure 5.5. Show the result of each query if it is applied to the COMPANY database in 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'.

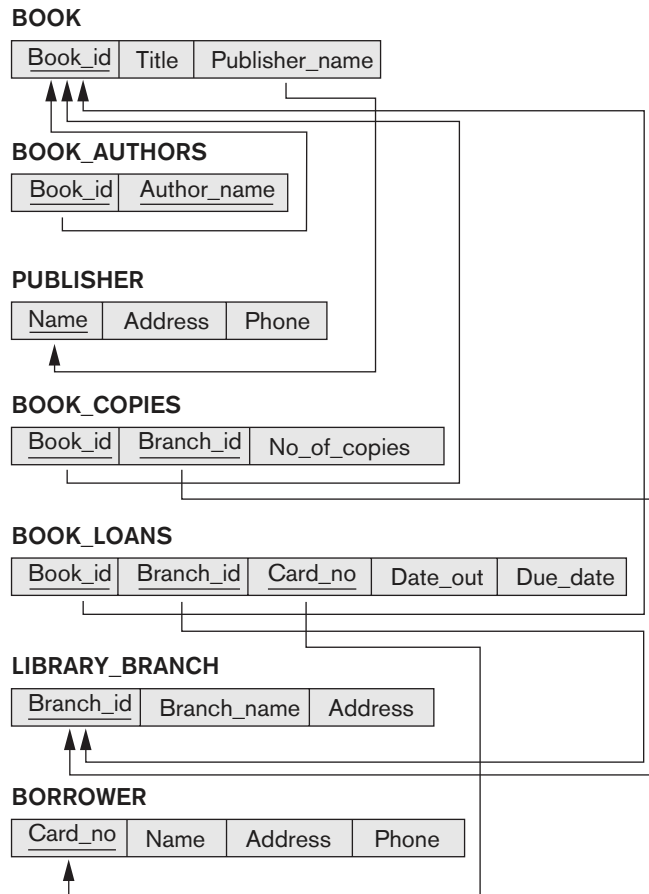


Figure 6.6
A relational database
schema for a
LIBRARY database.

- 6.11. Specify the updates of Exercise 3.11 using the SQL update commands.
- 6.12. 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 2007 and 2008.
 - 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 = 4) majoring in CS. A transcript includes course name, course number, credit hours, semester, year, and grade for each course completed by the student.

- 6.13.** Write SQL update statements to do the following on the database schema shown in Figure 1.2.
- Insert a new student, <'Johnson', 25, 1, 'Math'>, in the database.
 - Change the class of student 'Smith' to 2.
 - Insert a new course, <'Knowledge Engineering', 'cs4390', 3, 'cs'>.
 - Delete the record for the student whose name is 'Smith' and whose student number is 17.
- 6.14.** Design a relational database schema for a database application of your choice.
- Declare your relations using the SQL DDL.
 - Specify a number of queries in SQL that are needed by your database application.
 - Based on your expected use of the database, choose some attributes that should have indexes specified on them.
 - Implement your database, if you have a DBMS that supports SQL.
- 6.15.** Consider that the EMPLOYEE table's constraint EMPSUPERFK as specified in Figure 6.2 is changed to read as follows:

```
CONSTRAINT EMPSUPERFK
FOREIGN KEY (Super_ssn) REFERENCES EMPLOYEE(Ssn)
ON DELETE CASCADE ON UPDATE CASCADE;
```

Answer the following questions:

- What happens when the following command is run on the database state shown in Figure 5.6?

DELETE EMPLOYEE WHERE Lname = 'Borg'
 - Is it better to CASCADE or SET NULL in case of EMPSUPERFK constraint ON DELETE?
- 6.16.** Write SQL statements to create a table EMPLOYEE_BACKUP to back up the EMPLOYEE table shown in Figure 5.6.

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 & 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. We will give additional references to various aspects of SQL at the end of Chapter 7.

This page intentionally left blank

More SQL: Complex Queries, Triggers, Views, and Schema Modification

This chapter describes more advanced features of the SQL language for relational databases. We start in Section 7.1 by presenting more complex features of SQL retrieval queries, such as nested queries, joined tables, outer joins, aggregate functions, and grouping, and case statements. In Section 7.2, we describe the CREATE ASSERTION statement, which allows the specification of more general constraints on the database. We also introduce the concept of triggers and the CREATE TRIGGER statement, which will be presented in more detail in Section 26.1 when we present the principles of active databases. Then, in Section 7.3, we describe the SQL facility 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. Section 7.4 introduces the SQL ALTER TABLE statement, which is used for modifying the database tables and constraints. Section 7.5 is the chapter summary.

This chapter is a continuation of Chapter 6. The instructor may skip parts of this chapter if a less detailed introduction to SQL is intended.

7.1 More Complex SQL Retrieval Queries

In Section 6.3, we described some basic types of retrieval 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 retrievals from the database. We discuss several of these features in this section.

7.1.1 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* (value exists but is not known, or it is not known whether or not the value exists), value *not available* (value exists but is purposely withheld), or value *not applicable* (the attribute does not apply to this tuple or is undefined for this tuple). Consider the following examples to illustrate each of the meanings of NULL.

- 1. **Unknown value.** A person’s date of birth is not known, so it is represented by NULL in the database. An example of the other case of unknown would be NULL for a person’s home phone because it is not known whether or not the person has a home phone.
- 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 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 among the different meanings of NULL.

In general, each individual NULL value is considered to be different from every other NULL value in the various database records. When a record with NULL in one of its attributes 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 (Boolean) logic with values TRUE or FALSE. It is therefore necessary to define the results (or truth values) of three-valued logical expressions when the logical connectives AND, OR, and NOT are used. Table 7.1 shows the resulting values.

Table 7.1 Logical Connectives in Three-Valued Logic

(a)	AND	TRUE	FALSE	UNKNOWN
	TRUE	TRUE	FALSE	UNKNOWN
	FALSE	FALSE	FALSE	FALSE
	UNKNOWN	UNKNOWN	FALSE	UNKNOWN
(b)	OR	TRUE	FALSE	UNKNOWN
	TRUE	TRUE	TRUE	TRUE
	FALSE	TRUE	FALSE	UNKNOWN
	UNKNOWN	TRUE	UNKNOWN	UNKNOWN
(c)	NOT			
	TRUE	FALSE		
	FALSE	TRUE		
	UNKNOWN	UNKNOWN		

In Tables 7.1(a) and 7.1(b), the rows and columns represent the values of the results of comparison conditions, which would typically appear in the WHERE clause of an SQL query. Each expression result would have a value of TRUE, FALSE, or UNKNOWN. The result of combining the two values using the AND logical connective is shown by the entries in Table 7.1(a). Table 7.1(b) shows the result of using the OR logical connective. For example, the result of (FALSE AND UNKNOWN) is FALSE, whereas the result of (FALSE OR UNKNOWN) is UNKNOWN. Table 7.1(c) shows the result of the NOT logical operation. Notice that in standard Boolean logic, only TRUE or FALSE values are permitted; there is no UNKNOWN value.

In select-project-join queries, the general rule is that only those combinations of tuples that evaluate the logical expression in the WHERE clause 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 in Section 7.1.6.

SQL allows queries that check whether an attribute value is **NULL**. Rather than using = or <> to compare an attribute value to NULL, SQL uses the comparison operators **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 7.1.6). Query 18 illustrates NULL comparison by retrieving any employees who do not have a supervisor.

Query 18. Retrieve the names of all employees who do not have supervisors.

```
Q18:   SELECT    Fname, Lname
        FROM      EMPLOYEE
        WHERE     Super_ssn IS NULL;
```

7.1.2 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 another SQL query. That other query is called the **outer query**. These nested queries can also appear in the WHERE clause or the FROM clause or the SELECT clause or other SQL clauses as needed. 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 .

In Q4A, the first nested query selects the project numbers of projects that have an employee with last name 'Smith' involved as manager, whereas the second nested query selects the project numbers of projects that have an employee with last name '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.

```

Q4A:  SELECT  DISTINCT Pnumber
      FROM    PROJECT
      WHERE   Pnumber IN
              ( SELECT  Pnumber
                FROM    PROJECT, DEPARTMENT, EMPLOYEE
                WHERE   Dnum = Dnumber AND
                        Mgr_ssn = Ssn AND Lname = 'Smith' )

      OR

      Pnumber IN
              ( SELECT  Pno
                FROM    WORKS_ON, EMPLOYEE
                WHERE   Essn = Ssn AND Lname = 'Smith' );

```

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   Essn = '123456789' );

```

This query will select the Essns 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) within each tuple in WORKS_ON with the set of type-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 two keywords ANY and SOME have the same effect. 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 );

```


Notice that this query can also be specified using the MAX aggregate function (see Section 7.1.7).

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 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.

Query 16. Retrieve the name of each employee who has a dependent with the same first name and is the same sex as the employee.

```

Q16:   SELECT      E.Fname, E.Lname
        FROM        EMPLOYEE AS E
        WHERE       E.Ssn IN  ( SELECT      D.Essn
                                FROM        DEPENDENT AS D
                                WHERE       E.Fname = D.Dependent_name
                                AND E.Sex = D.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. If there were any unqualified references to Sex in the nested query, they would refer to the Sex attribute of DEPENDENT. However, we would not *have to* qualify the attributes Fname and Ssn of EMPLOYEE if they appeared in the nested query 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, as illustrated in Q16.

7.1.3 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;

```

7.1.4 The EXISTS and UNIQUE Functions in SQL

EXISTS and UNIQUE are Boolean functions that return TRUE or FALSE; hence, they can be used in a WHERE clause condition. The EXISTS function in SQL is used to check whether the result of a nested query is *empty* (contains no tuples) or not. The result of EXISTS is a Boolean value **TRUE** if the nested query result contains at least one tuple, or **FALSE** if the nested query result contains no tuples. We illustrate the use of EXISTS—and NOT EXISTS—with some examples. First, we formulate Query 16 in an alternative form that uses EXISTS as in Q16B:

```

Q16B:      SELECT    E.Fname, E.Lname
              FROM      EMPLOYEE AS E
              WHERE      EXISTS ( SELECT    *
                                FROM      DEPENDENT AS D
                                WHERE      E.Ssn = D.Essn AND E.Sex = D.Sex
                                        AND E.Fname = D.Dependent_name);

```

EXISTS and NOT EXISTS are typically 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 Essn, Sex, and Dependent_name as the EMPLOYEE tuple; if at least one tuple EXISTS in the result of the nested query, then select that EMPLOYEE tuple. EXISTS(Q) returns **TRUE** if there is *at least one tuple* in the result of the nested query Q, and 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 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 because the **WHERE**-clause condition will evaluate to **TRUE** in this case. 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 = Mgr_ssn );

```

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?

The query Q3: *Retrieve the name of each employee who works on all the projects controlled by department number 5* can be written using EXISTS and NOT EXISTS in SQL systems. We show two ways of specifying this query Q3 in SQL as Q3A and Q3B. This is an example of certain types of queries that require *universal quantification*, as we will discuss in Section 8.6.7. One way to write this query is to use the construct (S2 EXCEPT S1) as explained next, and checking whether the result is empty.¹ 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 with the outer query) 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 result MINUS (EXCEPT) the second subquery result is empty, it means that the employee works on all the projects and is therefore selected.

¹Recall that EXCEPT is the set difference operator. The keyword MINUS is also sometimes used, for example, in Oracle.

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 Q3A.

```

Q3B:  SELECT  Lname, Fname
        FROM    EMPLOYEE
        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 will write this query in tuple relation calculus (see Section 8.6.7).

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 (no duplicates) or a multiset (duplicates exist).

7.1.5 Explicit Sets and Renaming 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 in general, and it can be used in appropriate parts of a query. For example, Q8A shows how query Q8 from Section 4.3.2 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 for the query result.

```
Q8A:  SELECT  E.Lname AS Employee_name, S.Lname AS Supervisor_name
      FROM    EMPLOYEE AS E, EMPLOYEE AS S
      WHERE   E.Super_ssn = S.Ssn;
```

7.1.6 Joined Tables in SQL and Outer Joins

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 to specify the join of the EMPLOYEE and DEPARTMENT relations in the WHERE clause, 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 Sections 8.3.2 and 8.4.4 for more details on the various types of join operations in relational algebra).

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 the EMPLOYEE table), 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 called 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 who *have a supervisor* are included in the result;

an **EMPLOYEE** tuple whose value for `Super_ssn` is `NULL` is excluded. If the user requires that all employees be included, a different type of join called **OUTER JOIN** must be used explicitly (see Section 8.4.4 for the definition of **OUTER JOIN** in relational algebra). There are several variations of **OUTER JOIN**, as we shall see. In the SQL standard, this is handled by explicitly specifying the keyword **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.Super_ssn = S.Ssn);
```

In SQL, the options available for specifying joined tables include **INNER JOIN** (only pairs of tuples that match the join condition are retrieved, same as **JOIN**), **LEFT OUTER JOIN** (every tuple in the left table must appear in the result; if it does not have a matching tuple, it is padded with `NULL` values for the attributes of the right table), **RIGHT OUTER JOIN** (every tuple in the right table must appear in the result; if it does not have a matching tuple, it is padded with `NULL` values for the attributes of the left table), and **FULL OUTER JOIN**. In the latter three options, the keyword **OUTER** may be omitted. If the join attributes have the same name, one can 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 8.2.2), 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 allows the specification of the join of three or more tables as a single joined table, which is called a **multiway join**. For example, Q2A is a different way of specifying query Q2 from Section 6.3.1 using the concept of a joined table:

```
Q2A:  SELECT    Pnumber, Dnum, Lname, Address, Bdate
      FROM      ((PROJECT JOIN DEPARTMENT ON Dnum = Dnumber)
      JOIN EMPLOYEE ON Mgr_ssn = Ssn)
      WHERE     Plocation = 'Stafford';
```

Not all SQL implementations have implemented the new syntax of joined tables. In some systems, a different syntax was used to specify outer joins by using the comparison operators `+`, `= +`, and `++` for left, right, and full outer join, respectively, when specifying the join condition. For example, this syntax is available in Oracle. To specify the left outer join in Q8B using this syntax, we could write the query Q8C as follows:

```
Q8C:  SELECT    E.Lname, S.Lname
      FROM      EMPLOYEE E, EMPLOYEE S
      WHERE     E.Super_ssn ++ S.Ssn;
```

7.1.7 Aggregate Functions in SQL

Aggregate functions are used to summarize information from multiple tuples into a single-tuple summary. **Grouping** is used to create subgroups of tuples before summarization. Grouping and aggregation are required in many database

applications, and we will introduce their use in SQL through examples. A number of built-in aggregate functions exist: **COUNT**, **SUM**, **MAX**, **MIN**, and **AVG**.² The **COUNT** function returns the *number of tuples or values* as specified in a query. The functions **SUM**, **MAX**, **MIN**, and **AVG** can be 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.³ We illustrate the use of these functions with several queries.

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;

This query returns a *single-row* summary of all the rows in the **EMPLOYEE** table. We could use **AS** to rename the column names in the resulting single-row table; for example, as in Q19A.

Q19A: **SELECT** **SUM** (Salary) **AS** Total_Sal, **MAX** (Salary) **AS** Highest_Sal,
 MIN (Salary) **AS** Lowest_Sal, **AVG** (Salary) **AS** Average_Sal
 FROM EMPLOYEE;

If we want to get the preceding aggregate 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’;

²Additional aggregate functions for more advanced statistical calculation were added in SQL-99.

³Total 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.

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 only exception is for COUNT(*) because tuples instead of values are counted. In the previous examples, any Salary values that are NULL are not included in the aggregate function calculation. The general rule is as follows: when an aggregate function is applied to a collection of values, NULLs are removed from the collection before the calculation; if the collection becomes empty because all values are NULL, the aggregate function will return NULL (except in the case of COUNT, where it will return 0 for an empty collection of values).

The preceding examples summarize a *whole relation* (Q19, Q21, Q23) or a selected subset of tuples (Q20, Q22), and hence all produce a table with a single row or a single value. They illustrate how functions are applied to retrieve a summary value or summary tuple from a table. 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.

SQL also has aggregate functions SOME and ALL that can be applied to a collection of Boolean values; SOME returns TRUE if at least one element in the collection is TRUE, whereas ALL returns TRUE if all elements in the collection are TRUE.

7.1.8 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 to produce summary information about each group. 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 **GROUP BY** attribute **Dno**. Hence, each group contains the employees who work in the same department. 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 aggregate functions to be applied on each group of tuples. Figure 7.1(a) illustrates how grouping works and 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.

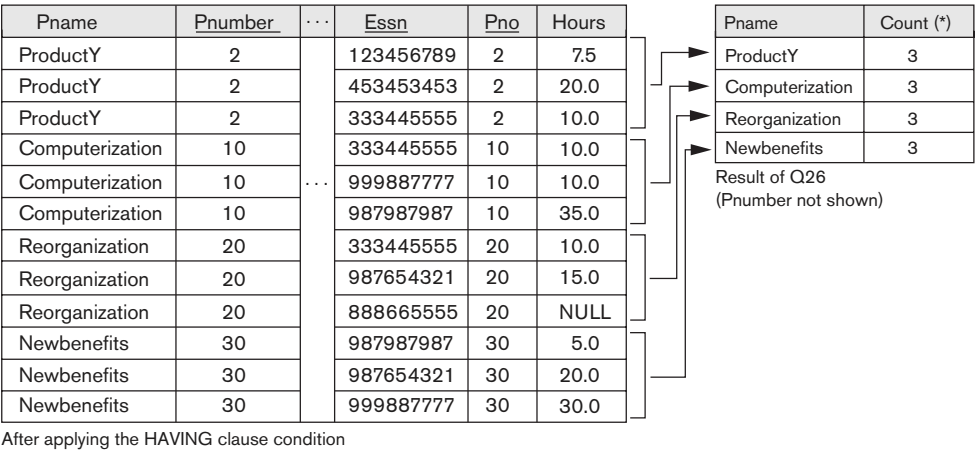
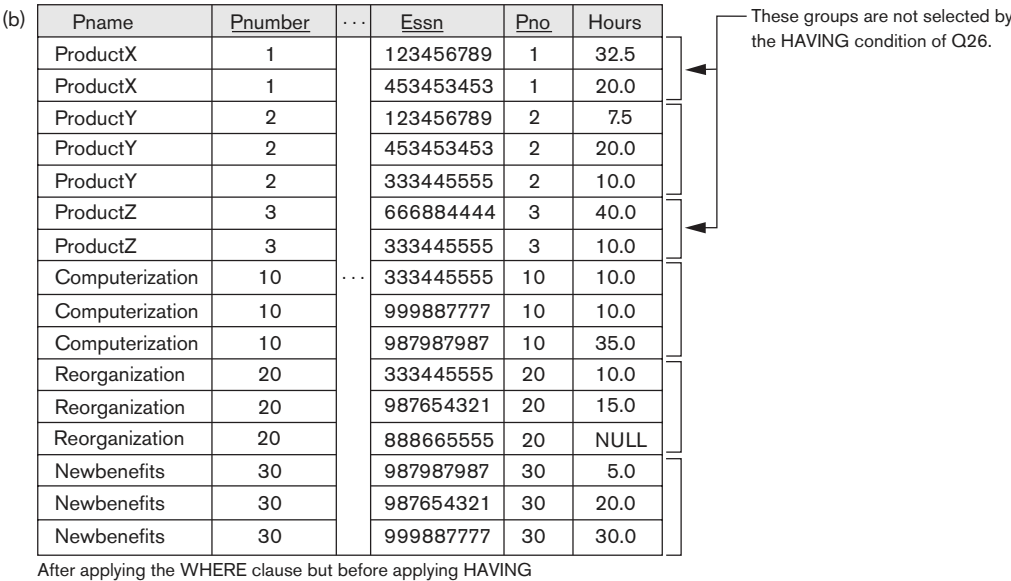
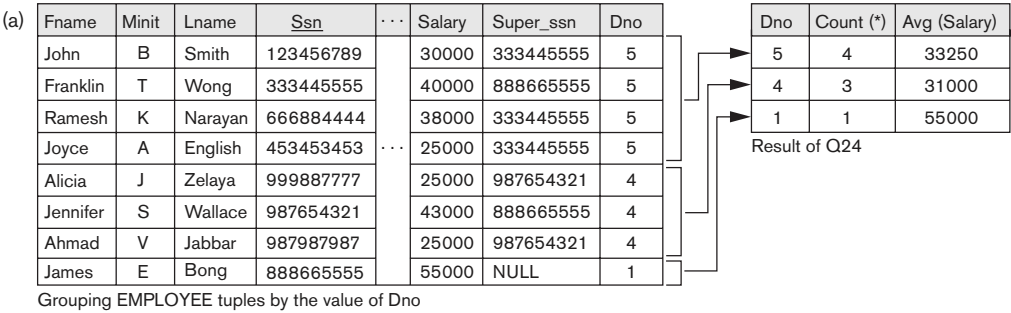
Query 25. For each project, retrieve the project number, the project name, and the number of employees who work on that project.

```
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 in the **WHERE** clause.

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 summary information regarding 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.

Figure 7.1
Results of GROUP BY and HAVING. (a) Q24. (b) Q26.



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 although selection conditions in the WHERE clause limit the *tuples* to which functions are applied, the HAVING clause serves to choose *whole groups*. Figure 7.1(b) illustrates the use of HAVING and displays the result of 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;
```

In Q27, 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 aggregate 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    Dno, COUNT (*)
FROM      EMPLOYEE
WHERE     Salary > 40000
GROUP BY Dno
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 or joined tuples; the HAVING clause is applied later, to select individual groups of tuples. In the incorrect query, 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      Dno, COUNT (*)
           FROM        EMPLOYEE
           WHERE       Salary>40000 AND Dno IN
                    ( SELECT      Dno
                      FROM        EMPLOYEE
                      GROUP BY   Dno
                      HAVING      COUNT (*) > 5)
           GROUP BY   Dno;

```

7.1.9 Other SQL Constructs: WITH and CASE

In this section, we illustrate two additional SQL constructs. The WITH clause allows a user to define a table that will only be used in a particular query; it is somewhat similar to creating a view (see Section 7.3) that will be used only in one query and then dropped. This construct was introduced as a convenience in SQL:99 and may not be available in all SQL based DBMSs. Queries using WITH can generally be written using other SQL constructs. For example, we can rewrite Q28 as Q28':

```

Q28':    WITH        BIGDEPTS (Dno) AS
           ( SELECT      Dno
             FROM        EMPLOYEE
             GROUP BY   Dno
             HAVING      COUNT (*) > 5)
           SELECT      Dno, COUNT (*)
           FROM        EMPLOYEE
           WHERE       Salary>40000 AND Dno IN BIGDEPTS
           GROUP BY   Dno;

```

In Q28', we defined in the WITH clause a temporary table BIG_DEPTS whose result holds the Dno's of departments with more than five employees, then used this table in the subsequent query. Once this query is executed, the temporary table BIGDEPTS is discarded.

SQL also has a CASE construct, which can be used when a value can be different based on certain conditions. This can be used in any part of an SQL query where a value is expected, including when querying, inserting or updating tuples. We illustrate this with an example. Suppose we want to give employees different raise amounts depending on which department they work for; for example, employees in department 5 get a \$2,000 raise, those in department 4 get \$1,500 and those in department 1 get \$3,000 (see Figure 5.6 for the employee tuples). Then we could re-write the update operation U6 from Section 6.4.3 as U6':

```

U6':      UPDATE     EMPLOYEE
           SET        Salary =
           CASE       WHEN      Dno = 5      THEN Salary + 2000
                     WHEN      Dno = 4      THEN Salary + 1500
                     WHEN      Dno = 1      THEN Salary + 3000
                     ELSE        Salary + 0 ;

```

In U6', the salary raise value is determined through the CASE construct based on the department number for which each employee works. The CASE construct can also be used when inserting tuples that can have different attributes being NULL depending on the type of record being inserted into a table, as when a specialization (see Chapter 4) is mapped into a single table (see Chapter 9) or when a union type is mapped into relations.

7.1.10 Recursive Queries in SQL

In this section, we illustrate how to write a recursive query in SQL. This syntax was added in SQL:99 to allow users the capability to specify a recursive query in a declarative manner. An example of a **recursive relationship** between tuples of the same type is the relationship between an employee and a supervisor. This relationship is described by the foreign key `Super_ssn` 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 a supervisory 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. In SQL:99, this query can be written as follows:

```

Q29:      WITH RECURSIVE  SUP_EMP (SupSsn, EmpSsn) AS
           ( SELECT        SupervisorSsn, Ssn
             FROM          EMPLOYEE
             UNION
             SELECT        E.Ssn, S.SupSsn
             FROM          EMPLOYEE AS E, SUP_EMP AS S
             WHERE         E.SupervisorSsn = S.EmpSsn)
           SELECT*
           FROM            SUP_EMP;
```

In Q29, we are defining a view `SUP_EMP` that will hold the result of the recursive query. The view is initially empty. It is first loaded with the first level (supervisor, supervisee) `Ssn` combinations via the first part (**SELECT** `SupervisorSsn, Ssn` **FROM** `EMPLOYEE`), which is called the **base query**. This will be combined via **UNION** with each successive level of supervisees through the second part, where the view contents are joined again with the base values to get the second level combinations, which are **UNIONed** with the first level. This is repeated with successive levels until a **fixed point** is reached, where no more tuples are added to the view. At this point, the result of the recursive query is in the view `SUP_EMP`.

7.1.11 Discussion and Summary of SQL Queries

A retrieval query in SQL can consist of up to six clauses, but only the first two—**SELECT** and **FROM**—are mandatory. The query can span several lines, and is ended by a semicolon. Query terms are separated by spaces, and parentheses can be used to group relevant parts of a query in the standard way. 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 selecting the 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.

In order to formulate queries correctly, it is useful to consider the steps that define the *meaning* or *semantics* of each query. A query is evaluated *conceptually*⁴ 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 to select and join tuples, 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 to execute. We discuss query processing and optimization in Chapters 18 and 19.

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 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

⁴The actual order of query evaluation is implementation dependent; this is just a way to conceptually view a query in order to correctly formulate it.

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 disable 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 so that the condition will always be `TRUE`. Thus, the assertion is violated if the result of the query is not empty. In the preceding 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 *individual* attributes and domains (see Section 6.2.1) and on *individual* tuples (see Section 6.2.4). A major difference between `CREATE ASSERTION` and the individual domain constraints and tuple constraints is that the `CHECK` clauses on individual attributes, domains, and tuples are checked in SQL *only when tuples are inserted or updated* in a specific table. 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 simple checks are implemented more efficiently by the DBMS.

7.2.2 Introduction to Triggers in SQL

Another important statement in SQL is `CREATE TRIGGER`. 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. 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. We discuss triggers in detail in Section 26.1 when we describe *active databases*. Here we just give a simple example of how triggers may be used.

Suppose we want to check whenever an employee's salary is greater than the salary of his or her direct supervisor in the COMPANY database (see Figures 5.5 and 5.6). Several events can trigger this rule: inserting a new employee record, changing an employee's salary, or changing an employee's supervisor. Suppose that the action to take would be to call an external stored procedure SALARY_VIOLATION,⁵ which will notify the supervisor. The trigger could then be written as in R5 below. Here we are using the syntax of the Oracle database system.

```
R5: CREATE TRIGGER SALARY_VIOLATION
BEFORE INSERT OR UPDATE OF SALARY, SUPERVISOR_SSN
ON EMPLOYEE
FOR EACH ROW
WHEN ( NEW.SALARY > ( SELECT SALARY FROM EMPLOYEE
WHERE SSN = NEW.SUPERVISOR_SSN ) )
INFORM_SUPERVISOR(NEW.Superervisor_ssn,
NEW.Ssn );
```

The trigger is given the name SALARY_VIOLATION, which can be used to remove or deactivate the trigger later. A typical trigger which is regarded as an ECA (Event, Condition, Action) rule has three components:

1. The **event(s)**: These are usually database update operations that are explicitly applied to the database. In this example the events are: inserting a new employee record, changing an employee's salary, or changing an employee's supervisor. The person who writes the trigger must make sure that all possible events are accounted for. In some cases, it may be necessary to write more than one trigger to cover all possible cases. These events are specified after the keyword **BEFORE** in our example, which means that the trigger should be executed before the triggering operation is executed. An alternative is to use the keyword **AFTER**, which specifies that the trigger should be executed after the operation specified in the event is completed.
2. The **condition** that determines whether the rule action should be executed: Once the triggering event has occurred, an *optional* condition may be evaluated. If *no condition* is specified, the action will be executed once the event occurs. If a condition is specified, it is first evaluated, and only *if it evaluates to true* will the rule action be executed. The condition is specified in the **WHEN** clause of the trigger.
3. The **action** to be taken: The action is usually a sequence of SQL statements, but it could also be a database transaction or an external program that will be automatically executed. In this example, the action is to execute the stored procedure **INFORM_SUPERVISOR**.

Triggers can be used in various applications, such as maintaining database consistency, monitoring database updates, and updating derived data automatically. A complete discussion is given in Section 26.1.

⁵Assuming that an appropriate external procedure has been declared. We discuss stored procedures in Chapter 10.

7.3 Views (Virtual Tables) in SQL

In this section we introduce the concept of a view in SQL. We show how views are specified, and then we discuss the problem of updating views and how views can be implemented by the DBMS.

7.3.1 Concept of a View in SQL

A **view** in SQL terminology is a single table that is derived from other tables.⁶ These other tables can be *base tables* or previously defined views. A view does not necessarily exist in physical form; it is considered to be a **virtual table**, in contrast to **base tables**, whose tuples are always physically 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, referring to the COMPANY database 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 three tables EMPLOYEE, WORKS_ON, and PROJECT every time we issue this query, we can define a view that is specified as the result of these joins. Then we can 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.

7.3.2 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 new 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 7.2 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_ems, Total_sal)
	AS SELECT	Dname, COUNT (*), SUM (Salary)
	FROM	DEPARTMENT, EMPLOYEE
	WHERE	Dnumber = Dno
	GROUP BY	Dname;

⁶As 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.

WORKS_ON1

Fname	Lname	Pname	Hours
-------	-------	-------	-------

DEPT_INFO

Dept_name	No_of_emps	Total_sal
-----------	------------	-----------

Figure 7.2

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 the ‘ProductX’ project, we can utilize the WORKS_ON1 view and specify the query as in QV1:

```
QV1:  SELECT  Fname, Lname
      FROM    WORKS_ON1
      WHERE   Pname = ‘ProductX’;
```

The same query would require the specification of two joins if specified on the base relations directly; 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 Section 7.3.4 and Chapter 30).

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 does not have to be realized or materialized at the time of *view definition* but rather at the time when 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 kept up-to-date. We will discuss various ways the DBMS can utilize to keep a view up-to-date in the next subsection.

If we do not need a view anymore, 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;
```

7.3.3 View Implementation, View Update, and Inline Views

The problem of how a DBMS can efficiently implement a view for efficient querying is complex. Two main approaches have been suggested. One strategy, called **query modification**, involves modifying or transforming the view query (submitted by the

user) 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 = 'ProductX';

```

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 view queries are going to be applied to the same view within a short period of time. The second strategy, called **view materialization**, involves physically creating a temporary or permanent view table when the view is first queried or created 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 the DBMS can determine what new tuples must be inserted, deleted, or modified in a *materialized view table* when a database update is applied to *one of the defining base tables*. The view is generally kept as a materialized (physically stored) table 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 table and recompute it from scratch when future queries reference the view.

Different strategies as to when a materialized view is updated are possible. The **immediate update** strategy updates a view as soon as the base tables are changed; the **lazy update** strategy updates the view when needed by a view query; and the **periodic update** strategy updates the view periodically (in the latter strategy, a view query may get a result that is not up-to-date).

A user can always issue a retrieval query against any view. However, issuing an INSERT, DELETE, or UPDATE command on a view table is in many cases not possible. 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*. Hence, it is often not possible for the DBMS to determine which of the updates is intended. 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. In addition, some of these updates will create

additional side effects that affect the result of other queries. For example, here are two possible updates, (a) and (b), on the base relations corresponding to the view update operation in UV1:

```
(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 instead 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 incorrect request is shown as UV2:

```
UV2:  UPDATE  DEPT_INFO
      SET      Total_sal = 100000
      WHERE    Dname = 'Research';
```

Generally, a view update is feasible when only *one possible update* on the base relations can accomplish the desired update operation on the view. Whenever an update on the view can be mapped to *more than one update* on the underlying base relations, it is usually not permitted. Some researchers have suggested that the DBMS have a certain procedure for choosing one of the possible updates as the most likely one. Some researchers have developed methods for choosing the most likely update, whereas other researchers prefer to have the user choose the desired update mapping during view definition. But these options are generally not available in most commercial DBMSs.

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** should be added at the end of the view definition if a view *is to be updated* by INSERT, DELETE, or UPDATE statements. This allows the system to reject operations that violate the SQL rules for view updates. The full set of SQL rules for when a view may be modified by the user are more complex than the rules stated earlier.

It is also possible to define a view table in the **FROM clause** of an SQL query. This is known as an **in-line view**. In this case, the view is defined within the query itself.

7.3.4 Views as Authorization Mechanisms

We describe SQL query authorization statements (GRANT and REVOKE) in detail in Chapter 30, when we present database security and authorization mechanisms. Here, we will just give a couple of simple examples to illustrate how views can be used to hide certain attributes or tuples from unauthorized users. Suppose a certain user is only allowed to see employee information for employees who work for department 5; then we can create the following view DEPT5EMP and grant the user the privilege to query the view but not the base table EMPLOYEE itself. This user will only be able to retrieve employee information for employee tuples whose Dno = 5, and will not be able to see other employee tuples when the view is queried.

```
CREATE VIEW    DEPT5EMP    AS
SELECT        *
FROM          EMPLOYEE
WHERE         Dno = 5;
```

In a similar manner, a view can restrict a user to only see certain columns; for example, only the first name, last name, and address of an employee may be visible as follows:

```
CREATE VIEW    BASIC_EMP_DATA    AS
SELECT        Fname, Lname, Address
FROM          EMPLOYEE;
```

Thus by creating an appropriate view and granting certain users access to the view and not the base tables, they would be restricted to retrieving only the data specified in the view. Chapter 30 discusses security and authorization in detail, including the GRANT and REVOKE statements of SQL.

7.4 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. This can be done while the database is operational and does not require recompilation of the database schema. Certain

checks must be done by the DBMS to ensure that the changes do not affect the rest of the database and make it inconsistent.

7.4.1 The DROP Command

The DROP command can be used to drop *named* schema elements, such as tables, domains, types, or constraints. One can also drop a whole schema if it is no longer needed by using the DROP SCHEMA command. 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. To use the RESTRICT option, the user must first individually drop each element in the schema, then drop the schema itself.

If a base relation within a schema is no longer needed, 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 6.1, 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 7.3) or by any other elements. With the CASCADE option, all such constraints, views, and other elements that reference the table being dropped are also dropped automatically from the schema, along with the table itself.

Notice that the DROP TABLE command not only deletes all the records in the table if successful, but also removes the *table definition* from the catalog. If it is desired to delete only the records but to leave the table definition for future use, then the DELETE command (see Section 6.4.2) should be used instead of DROP TABLE.

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

7.4.2 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 relation in the COMPANY schema (see Figure 6.1), we can use the command

```
ALTER TABLE COMPANY.EMPLOYEE ADD COLUMN 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 individually on each tuple (see Section 6.4.3). 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 schema elements) reference the column. For example, the following command removes the attribute Address from the EMPLOYEE base table:

```
ALTER TABLE COMPANY.EMPLOYEE DROP COLUMN 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 COLUMN Mgr_ssn  
DROP DEFAULT;  
ALTER TABLE COMPANY.DEPARTMENT ALTER COLUMN Mgr_ssn  
SET DEFAULT '333445555';
```

One can also change the constraints specified on a table by adding or dropping a named 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 6.2 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 CONSTRAINT** 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. It is also possible to create new tables and views within a database schema using the appropriate commands. There are many other details and options; we refer the interested reader to the SQL documents listed in the Selected Bibliography at the end of this chapter.

7.5 Summary

In this chapter we presented additional features of the SQL database language. We started in Section 7.1 by presenting more complex features of SQL retrieval queries, including nested queries, joined tables, outer joins, aggregate functions, and grouping. In Section 7.2, we described the CREATE ASSERTION statement, which allows the specification of more general constraints on the database, and introduced the

are shown in angled brackets `< ... >`, optional parts are shown in square brackets `[...]`, repetitions are shown in braces `{ ... }`, and alternatives are shown in parentheses `(... | ... | ...)`.⁷

Review Questions

- 7.1. Describe the six clauses in the syntax of an SQL retrieval query. 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?
- 7.2. Describe conceptually how an SQL retrieval query will be executed by specifying the conceptual order of executing each of the six clauses.
- 7.3. 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?
- 7.4. Discuss how each of the following constructs is used in SQL, and discuss the various options for each construct. Specify what each construct is useful for.
 - a. Nested queries
 - b. Joined tables and outer joins
 - c. Aggregate functions and grouping
 - d. Triggers
 - e. Assertions and how they differ from triggers
 - f. The SQL WITH clause
 - g. SQL CASE construct
 - h. Views and their updatability
 - i. Schema change commands

Exercises

- 7.5. Specify the following queries on the database in Figure 5.5 in SQL. Show the query results if each query is applied to the database state in Figure 5.6.
 - 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 making more than \$30,000, rather than all employees (as in Exercise 7.5a). Can we specify this query in SQL? Why or why not?

⁷The full syntax of SQL is described in many voluminous documents of hundreds of pages.

- 7.6. Specify the following queries in SQL on the database schema in Figure 1.2.
- Retrieve the names and major departments of all straight-A students (students who have a grade of A in all their courses).
 - Retrieve the names and major departments of all students who do not have a grade of A in any of their courses.
- 7.7. In SQL, specify the following queries on the database in Figure 5.5 using the concept of nested queries and other concepts described in this chapter.
- Retrieve the names of all employees who work in the department that has the employee with the highest salary among all employees.
 - Retrieve the names of all employees whose supervisor's supervisor has '888665555' for Ssn.
 - Retrieve the names of employees who make at least \$10,000 more than the employee who is paid the least in the company.
- 7.8. Specify the following views in SQL on the COMPANY database schema shown in Figure 5.5.
- A view that has the department name, manager name, and manager salary for every department
 - A view that has the employee name, supervisor name, and employee salary for each employee who works in the 'Research' department
 - 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
 - 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*
- 7.9. Consider the following view, DEPT_SUMMARY, defined on the COMPANY database in 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 in Figure 5.6.

- ```
SELECT *
FROM DEPT_SUMMARY;
```
- ```
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

Reisner (1977) describes a human factors evaluation of SEQUEL, a precursor of SQL, in which she found that users have some difficulty with specifying join conditions and grouping correctly. Date (1984) 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. 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) give a comprehensive treatment of the ANSI 1992 standard called 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.

There are many books that describe various aspects of SQL. For example, two references that describe SQL-99 are Melton and Simon (2002) and Melton (2003). Further SQL standards—SQL 2006 and SQL 2008—are described in a variety of technical reports; but no standard references exist.

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. In contrast, Chapters 6 and 7 described the *practical language* for the relational model, namely the SQL standard. Historically, the relational algebra and calculus were developed before the SQL language. SQL is primarily based on concepts from relational calculus and has been extended to incorporate some concepts from relational algebra as well. Because most relational DBMSs use SQL as their language, we presented the SQL language first.

Recall from Chapter 2 that a data model must include a set of operations to manipulate the database, in addition to the data model's concepts for defining the database's structure and constraints. We presented the structures and constraints of the formal relational model in Chapter 5. The basic set of operations for the formal relational model is the **relational algebra**. These operations enable a user to specify basic retrieval requests as *relational algebra expressions*. The result of a retrieval query is a new relation. 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 the query processing and optimization modules that are integral parts of relational database management systems (RDBMSs), as we shall discuss in Chapters 18 and 19. Third, some of its concepts are incorporated into the SQL standard

query language for RDBMSs. Although most commercial RDBMSs in use today do not provide user interfaces for relational algebra queries, the core operations and functions in the internal modules of most relational systems are based on relational algebra operations. We will define these operations in detail in Sections 8.1 through 8.4 of this chapter.

Whereas the algebra defines a set of operations for the relational model, the **relational calculus** provides a higher-level *declarative* language for specifying relational queries. In a relational calculus expression, there is *no order of operations* to specify how to retrieve the query result—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 standard query language (SQL) for RDBMSs has some of its foundations in a variation of relational calculus known as the tuple relational calculus.¹

The relational algebra is often considered to be an integral part of the relational data model. 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 (see Section 5.1). Set operations include UNION, INTERSECTION, SET DIFFERENCE, and CARTESIAN PRODUCT (also known as CROSS PRODUCT). The other group consists of operations developed specifically for relational databases—these include SELECT, PROJECT, and JOIN, among others. First, we describe the SELECT and PROJECT operations in Section 8.1 because they are **unary operations** that operate on single relations. Then we discuss set operations in Section 8.2. In Section 8.3, we discuss JOIN and other complex **binary operations**, which operate on two tables by combining related tuples (records) based on *join conditions*. 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, known as OUTER JOINS and OUTER UNIONS. These operations, which were added to the original relational algebra because of their importance to many database applications, are described in Section 8.4. We give examples of specifying queries that use relational operations in Section 8.5. Some of these same queries were used in Chapters 6 and 7. By using the same query numbers in this chapter, the reader can contrast how the same queries are written in the various query languages.

In Sections 8.6 and 8.7 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 8.6 and the *domain* relational calculus is described in Section 8.7. Some of the SQL constructs discussed in

¹SQL is based on tuple relational calculus, but also incorporates some of the operations from the relational algebra and its extensions, as illustrated in Chapters 6, 7, and 9.

Chapters 6 and 7 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 C we give an overview of the Query-By-Example (QBE) language, which is a graphical user-friendly relational language based on domain relational calculus. Section 8.8 summarizes the chapter.

For the reader who is interested in a less detailed introduction to formal relational languages, Sections 8.4, 8.6, and 8.7 may be skipped.

8.1 Unary Relational Operations: SELECT and PROJECT

8.1.1 The SELECT Operation

The SELECT operation is used to choose a *subset* of the tuples from a relation that satisfies a **selection condition**.³ We can consider the SELECT operation to be a *filter* that keeps only those tuples that satisfy a qualifying condition. Alternatively, we can consider the SELECT operation to *restrict* the tuples in a relation to only those tuples that satisfy the 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 filtered out. 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

$$\sigma_{\langle \text{selection condition} \rangle}(R)$$

where the symbol σ (sigma) is used to denote the SELECT operator and the selection condition is a Boolean expression (condition) 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 $\langle \text{selection condition} \rangle$ is made up of a number of **clauses** of the form

$\langle \text{attribute name} \rangle \langle \text{comparison op} \rangle \langle \text{constant value} \rangle$

²In this chapter no familiarity with first-order predicate calculus—which deals with quantified variables and values—is assumed.

³The SELECT operation is **different** from the SELECT clause of SQL. The SELECT operation chooses tuples from a table, and is sometimes called a RESTRICT or FILTER operation.

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 $\{=, <, \leq, >, \geq, \neq\}$, and <constant value> is a constant value from the attribute domain. Clauses can be connected by the standard 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 8.1(a).

Notice that all the comparison operators in the set $\{=, <, \leq, >, \geq, \neq\}$ can apply to attributes whose domains are *ordered values*, such as numeric or date domains. Domains of strings of characters are also considered to be 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 $\{=, \neq\}$ can be used. An example of an unordered domain is the domain $Color = \{\text{'red'}, \text{'blue'}, \text{'green'}, \text{'white'}, \text{'yellow'}, \dots\}$, 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.

Figure 8.1

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) $\pi_{Sex, Salary}(EMPLOYEE)$.

(a)

Fname	Minit	Lname	Ssn	Bdate	Address	Sex	Salary	Super_ssn	Dno
Franklin	T	Wong	333445555	1955-12-08	638 Voss, Houston, TX	M	40000	888665555	5
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

(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

In general, the result of a SELECT operation can be determined as follows. The $\langle \text{selection condition} \rangle$ is applied independently to each *individual 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,

$$\sigma_{\langle \text{cond1} \rangle}(\sigma_{\langle \text{cond2} \rangle}(R)) = \sigma_{\langle \text{cond2} \rangle}(\sigma_{\langle \text{cond1} \rangle}(R))$$

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

$$\sigma_{\langle \text{cond1} \rangle}(\sigma_{\langle \text{cond2} \rangle}(\dots (\sigma_{\langle \text{cond}_n \rangle}(R)) \dots)) = \sigma_{\langle \text{cond1} \rangle \text{ AND } \langle \text{cond2} \rangle \text{ AND } \dots \text{ AND } \langle \text{cond}_n \rangle}(R)$$

In SQL, the SELECT condition is typically specified in the *WHERE clause* of a query. For example, the following operation:

$$\sigma_{\text{Dno}=4 \text{ AND Salary}>25000}(\text{EMPLOYEE})$$

would correspond to the following SQL query:

```
SELECT      *
FROM        EMPLOYEE
WHERE       Dno=4 AND Salary>25000;
```

8.1.2 The PROJECT Operation

If we think of a relation as a table, the SELECT operation chooses 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. Therefore, the result of the PROJECT operation can 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 8.1(b). The general form of the PROJECT operation is

$$\pi_{\langle \text{attribute list} \rangle}(R)$$

where π (pi) is the symbol used to represent the PROJECT operation, and $\langle \text{attribute list} \rangle$ is the desired sublist 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 $\langle \text{attribute list} \rangle$ *in the same order as they appear in the list*. Hence, its **degree** is equal to the number of attributes in $\langle \text{attribute list} \rangle$.

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 distinct tuples, and hence a valid relation. 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 8.1(c). Notice that the tuple $\langle \text{'F', 25000} \rangle$ appears only once in Figure 8.1(c), even though this combination of values appears twice in the EMPLOYEE relation. Duplicate elimination involves sorting or some other technique to detect duplicates and thus adds more processing. If duplicates are not eliminated, the result would be a **multiset** or **bag** of tuples rather than a set. This was not permitted in the formal relational model but is allowed in SQL (see Section 6.3).

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

$$\pi_{\langle \text{list1} \rangle}(\pi_{\langle \text{list2} \rangle}(R)) = \pi_{\langle \text{list1} \rangle}(R)$$

as long as $\langle \text{list2} \rangle$ contains the attributes in $\langle \text{list1} \rangle$; otherwise, the left-hand side is an incorrect expression. It is also noteworthy that commutativity *does not* hold on PROJECT.

In SQL, the PROJECT attribute list is specified in the *SELECT clause* of a query. For example, the following operation:

$$\pi_{\text{Sex, Salary}}(\text{EMPLOYEE})$$

would correspond to the following SQL query:

```
SELECT    DISTINCT Sex, Salary
FROM      EMPLOYEE
```

Notice that if we remove the keyword **DISTINCT** from this SQL query, then duplicates will not be eliminated. This option is not available in the formal relational algebra, but the algebra can be extended to include this operation and allow relations to be multisets; we do not discuss these extensions here.

8.1.3 Sequences of Operations and the RENAME Operation

The relations shown in Figure 8.1 that depict operation results do not have any names. In general, for most queries, we need 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, also known as an **in-line expression**, as follows:

$$\pi_{Fname, Lname, Salary}(\sigma_{Dno=5}(EMPLOYEE))$$

Figure 8.2(a) shows the result of this in-line relational algebra expression. Alternatively, we can explicitly show the sequence of operations, giving a name to each intermediate relation, and using the **assignment operation**, denoted by \leftarrow (left arrow), as follows:

$$\begin{aligned} DEP5_EMPS &\leftarrow \sigma_{Dno=5}(EMPLOYEE) \\ RESULT &\leftarrow \pi_{Fname, Lname, Salary}(DEP5_EMPS) \end{aligned}$$

It is sometimes 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:

$$\begin{aligned} TEMP &\leftarrow \sigma_{Dno=5}(EMPLOYEE) \\ R(First_name, Last_name, Salary) &\leftarrow \pi_{Fname, Lname, Salary}(TEMP) \end{aligned}$$

These two operations are illustrated in Figure 8.2(b).

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—as a unary operator. The general RENAME operation when applied to a relation R of degree n is denoted by any of the following three forms:

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

(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	Super_ssn	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

First_name	Last_name	Salary
John	Smith	30000
Franklin	Wong	40000
Ramesh	Narayan	38000
Joyce	English	25000

Figure 8.2Results of a sequence of operations. (a) $\pi_{\text{Fname, Lname, Salary}}(\sigma_{\text{Dno}=5}(\text{EMPLOYEE}))$.

(b) Using intermediate relations and renaming of attributes.

where the symbol ρ (rho) is used to denote the RENAME operator, S is the new relation name, and B_1, B_2, \dots, 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, \dots, A_n) in that order, then each A_i is renamed as B_i .

In SQL, a single query typically represents a complex relational algebra expression. Renaming in SQL is accomplished by aliasing using **AS**, as in the following example:

```

SELECT      E.Fname AS First_name, E.Lname AS Last_name, E.Salary AS Salary
FROM        EMPLOYEE AS E
WHERE       E.Dno=5,

```

8.2 Relational Algebra Operations from Set Theory

8.2.1 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:⁴

```

DEP5_EMPS  $\leftarrow \sigma_{Dno=5}(EMPLOYEE)$ 
RESULT1  $\leftarrow \pi_{Ssn}(DEP5\_EMPS)$ 
RESULT2(Ssn)  $\leftarrow \pi_{Super\_ssn}(DEP5\_EMPS)$ 
RESULT  $\leftarrow RESULT1 \cup RESULT2$ 

```

The relation RESULT1 has the Ssn of all employees who work in department 5, whereas RESULT2 has the Ssn 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 8.3) while eliminating any duplicates. 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** or **EXCEPT**). 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* or *type compatibility*. Two relations $R(A_1, A_2, \dots, A_n)$ and $S(B_1, B_2, \dots, B_n)$ are said to be **union compatible** (or **type 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 .

RESULT1

Ssn
123456789
333445555
666884444
453453453

RESULT2

Ssn
333445555
888665555

RESULT

Ssn
123456789
333445555
666884444
453453453
888665555

Figure 8.3

Result of the UNION operation
 $RESULT \leftarrow RESULT1 \cup RESULT2$.

⁴As a single relational algebra expression, this becomes $Result \leftarrow \pi_{Ssn}(\sigma_{Dno=5}(EMPLOYEE)) \cup \pi_{Super_ssn}(\sigma_{Dno=5}(EMPLOYEE))$.

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 8.4 illustrates the three operations. The relations STUDENT and INSTRUCTOR in Figure 8.4(a) are union compatible and their tuples represent the names of students and the names of instructors, respectively. The result of the UNION operation in Figure 8.4(b) 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 8.4(c)) includes only those who are both students and instructors.

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

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

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

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

Figure 8.4

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.

(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

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

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

Figure 8.4(d) shows the names of students who are not instructors, and Figure 8.4(e) shows the names of instructors who are not students.

Note that INTERSECTION can be expressed in terms of union and set difference as follows:

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

In SQL, there are three operations—UNION, INTERSECT, and EXCEPT—that correspond to the set operations described here. In addition, there are multiset operations (UNION ALL, INTERSECT ALL, and EXCEPT ALL) that do not eliminate duplicates (see Section 6.3.4).

8.2.2 The CARTESIAN PRODUCT (CROSS PRODUCT) Operation

Next, we discuss the **CARTESIAN PRODUCT** operation—also known as **CROSS PRODUCT** or **CROSS JOIN**—which is denoted by \times . This is also a binary set operation, but the relations on which it is applied do *not* have to be union compatible. In its binary form, this set operation produces a new element by combining every member (tuple) from one relation (set) with every member (tuple) from the other relation (set). In general, the result of $R(A_1, A_2, \dots, A_n) \times S(B_1, B_2, \dots, B_m)$ is a relation Q with degree $n + m$ attributes $Q(A_1, A_2, \dots, A_n, B_1, B_2, \dots, B_m)$, 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 n -ary CARTESIAN PRODUCT operation is an extension of the above concept, which produces new tuples by concatenating all possible combinations of tuples from n underlying relations. The CARTESIAN PRODUCT operation applied by itself is generally meaningless. It is mostly 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 ←  $\sigma_{\text{Sex}='F'}$ (EMPLOYEE)
EMPNAMES ←  $\pi_{\text{Fname, Lname, Ssn}}$ (FEMALE_EMPS)
EMP_DEPENDENTS ← EMPNAMES  $\times$  DEPENDENT
ACTUAL_DEPENDENTS ←  $\sigma_{\text{Ssn}=\text{Essn}}$ (EMP_DEPENDENTS)
RESULT ←  $\pi_{\text{Fname, Lname, Dependent\_name}}$ (ACTUAL_DEPENDENTS)

```

The resulting relations from this sequence of operations are shown in Figure 8.5. The EMP_DEPENDENTS relation is the result of applying the CARTESIAN PRODUCT operation to EMPNAMES from Figure 8.5 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 (every dependent is

Figure 8.5

The CARTESIAN PRODUCT (CROSS PRODUCT) operation.

FEMALE_EMPS

Fname	Minit	Lname	Ssn	Bdate	Address	Sex	Salary	Super_ssn	Dno
Alicia	J	Zelaya	999887777	1968-07-19	3321Castle, Spring, TX	F	25000	987654321	4
Jennifer	S	Wallace	987654321	1941-06-20	291Berry, 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

combined with *every* female employee). We want to combine a female employee tuple only with her particular dependents—namely, the DEPENDENT tuples whose Essn value 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 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 SELECT *related tuples only* from the two relations by specifying an appropriate selection condition after the Cartesian product, as we did in the preceding example. Because this sequence of CARTESIAN PRODUCT followed by SELECT is quite commonly used to combine *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.

In SQL, CARTESIAN PRODUCT can be realized by using the CROSS JOIN option in joined tables (see Section 7.1.6). Alternatively, if there are two tables in the FROM clause and there is no corresponding join condition in the WHERE clause of the SQL query, the result will also be the CARTESIAN PRODUCT of the two tables (see Q10 in Section 6.3.3).

8.3 Binary Relational Operations: JOIN and DIVISION

8.3.1 The JOIN Operation

The JOIN operation, denoted by \bowtie , is used to combine *related tuples* from two relations into single “longer” tuples. This operation is very important for any relational database with more 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 Mgr_ssn value in the department tuple. We do this by using the JOIN operation and then projecting the result over the necessary attributes, as follows:

```
DEPT_MGR  $\leftarrow$  DEPARTMENT  $\bowtie_{\text{Mgr\_ssn}=\text{Ssn}}$  EMPLOYEE
RESULT  $\leftarrow \pi_{\text{Dname, Lname, Fname}}(\text{DEPT\_MGR})$ 
```

The first operation is illustrated in Figure 8.6. Note that Mgr_ssn is a foreign key of the DEPARTMENT relation that references Ssn, the primary key of the EMPLOYEE relation. This referential integrity constraint plays a role in having matching tuples in the referenced relation EMPLOYEE.

The JOIN operation can be specified as a CARTESIAN PRODUCT operation followed by a SELECT operation. However, JOIN is very important because it is used frequently when specifying database queries. Consider the earlier example

Figure 8.6

Result of the JOIN operation $\text{DEPT_MGR} \leftarrow \text{DEPARTMENT} \bowtie_{\text{Mgr_ssn}=\text{Ssn}} \text{EMPLOYEE}$.

DEPT_MGR

Dname	Dnumber	Mgr_ssn	...	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	...

illustrating CARTESIAN PRODUCT, which included the following sequence of operations:

$\text{EMP_DEPENDENTS} \leftarrow \text{EMP_NAMES} \times \text{DEPENDENT}$

$\text{ACTUAL_DEPENDENTS} \leftarrow \sigma_{\text{Ssn}=\text{Essn}}(\text{EMP_DEPENDENTS})$

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

$\text{ACTUAL_DEPENDENTS} \leftarrow \text{EMP_NAMES} \bowtie_{\text{Ssn}=\text{Essn}} \text{DEPENDENT}$

The general form of a JOIN operation on two relations⁵ $R(A_1, A_2, \dots, A_n)$ and $S(B_1, B_2, \dots, B_m)$ is

$R \bowtie_{\langle \text{join condition} \rangle} S$

The result of the JOIN is a relation Q with $n + m$ attributes $Q(A_1, A_2, \dots, A_n, B_1, B_2, \dots, B_m)$ 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

$\langle \text{condition} \rangle \text{ AND } \langle \text{condition} \rangle \text{ AND } \dots \text{ AND } \langle \text{condition} \rangle$

where each $\langle \text{condition} \rangle$ 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 or for which the join condition is FALSE *do not* appear in the result. In that sense, the JOIN operation does *not* necessarily preserve all of the information in the participating relations, because tuples that do not get combined with matching ones in the other relation do not appear in the result.

⁵Again, notice that R and S can be any relations that result from general *relational algebra expressions*.

8.3.2 Variations of JOIN: The EQUIJOIN and NATURAL 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 previous examples 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 8.6, the values of the attributes Mgr_ssn and Ssn are identical in every tuple of DEPT_MGR (the EQUIJOIN result) because the equality join condition specified on these two attributes *requires the values to be identical* in every tuple in the result. 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.⁶ 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.

Suppose we want to combine each PROJECT tuple with the DEPARTMENT tuple that controls the project. In the following example, first we rename the Dnumber attribute of DEPARTMENT to Dnum—so that it has the same name as the Dnum attribute in PROJECT—and then we apply NATURAL JOIN:

```
PROJ_DEPT ← PROJECT * ρ(Dname, Dnum, Mgr_ssn, Mgr_start_date)(DEPARTMENT)
```

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

```
DEPT ← ρ(Dname, Dnum, Mgr_ssn, Mgr_start_date)(DEPARTMENT)
PROJ_DEPT ← PROJECT * DEPT
```

The attribute Dnum is called the **join attribute** for the NATURAL JOIN operation, because it is the only attribute with the same name in both relations. The resulting relation is illustrated in Figure 8.7(a). 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 value* 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

```
DEPT_LOCS ← DEPARTMENT * DEPT_LOCATIONS
```

The resulting relation is shown in Figure 8.7(b), which combines each department with its locations and has one tuple for each location. In general, the join condition for NATURAL JOIN is constructed by equating *each pair of join attributes* that have the same name in the two relations and combining these conditions with **AND**. There can be a list of join attributes from each relation, and each corresponding pair must have the same name.

⁶NATURAL JOIN is basically an EQUIJOIN followed by the removal of the superfluous attributes.

(a)

PROJ_DEPT

Pname	<u>Pnumber</u>	Plocation	Dnum	Dname	Mgr_ssn	Mgr_start_date
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	Mgr_ssn	Mgr_start_date	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 8.7Results of two natural join operations. (a) proj_dept \leftarrow project \bowtie dept.(b) dept_locs \leftarrow department \bowtie dept_locations.

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 $R \bowtie_{\langle \text{join condition} \rangle} S$ 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, a single JOIN operation is used to combine data from two 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 join variation called *outer joins* (see Section 8.4.4). Informally, an *inner join* is a type of match-and-combine operation defined formally as a combination of CARTESIAN PRODUCT and SELECTION. Note that sometimes a join may be specified between a relation and itself, as we will illustrate in Section 8.4.3. 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:

$$((\text{PROJECT} \bowtie_{\text{Dnum}=\text{Dnumber}} \text{DEPARTMENT}) \bowtie_{\text{Mgr_ssn}=\text{Ssn}} \text{EMPLOYEE})$$

This combines each project tuple with its controlling department tuple into a single tuple, and then combines that tuple with an employee tuple that is the department manager. The net result is a consolidated relation in which each tuple contains this project-department-manager combined information.

In SQL, JOIN can be realized in several different ways. The first method is to specify the <join conditions> in the WHERE clause, along with any other selection conditions. This is very common and is illustrated by queries Q1, Q1A, Q1B, Q2, and Q8 in Sections 6.3.1 and 6.3.2, as well as by many other query examples in Chapters 6 and 7. The second way is to use a nested relation, as illustrated by queries Q4A and Q16 in Section 7.1.2. Another way is to use the concept of joined tables, as illustrated by the queries Q1A, Q1B, Q8B, and Q2A in Section 7.1.6. The construct of joined tables was added to SQL2 to allow the user to specify explicitly all the various types of joins, because the other methods were more limited. It also allows the user to clearly distinguish join conditions from the selection conditions in the WHERE clause.

8.3.3 A Complete Set of Relational Algebra Operations

It has been shown that the set of relational algebra operations $\{\sigma, \pi, \cup, \rho, -, \times\}$ 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:

$$R \bowtie_{\langle \text{condition} \rangle} S \equiv \sigma_{\langle \text{condition} \rangle} (R \times S)$$

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 include as separate operations because they are convenient to use and are very commonly applied in database applications. Other operations have been included in the basic relational algebra for convenience rather than necessity. We discuss one of these—the DIVISION operation—in the next section.

8.3.4 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:

```
SMITH ← σFname='John' AND Lname='Smith'(EMPLOYEE)
SMITH_PNOS ← πPno(WORKS_ON ⋈Essn=Ssn SMITH)
```

Next, create a relation that includes a tuple <Pno, Essn> whenever the employee whose Ssn is Essn works on the project whose number is Pno in the intermediate relation SSN_PNOS:

```
SSN_PNOS ← πEssn, Pno(WORKS_ON)
```

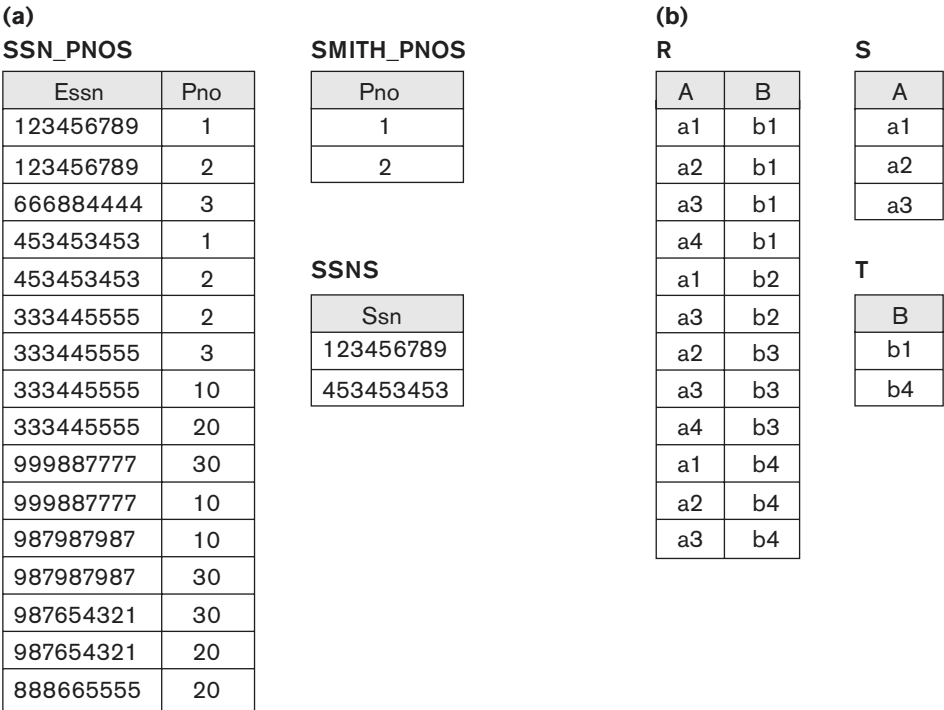
Finally, apply the DIVISION operation to the two relations, which gives the desired employees’ Social Security numbers:

```
SSNS(Ssn) ← SSN_PNOS ÷ SMITH_PNOS
RESULT ← πFname, Lname(SSNS * EMPLOYEE)
```

The preceding operations are shown in Figure 8.8(a).

In general, the DIVISION operation is applied to two relations $R(Z) \div S(X)$, where the attributes of S are a subset of the attributes of R ; that is, $X \subseteq Z$. Let Y be the set of attributes of R that are not attributes of S ; that is, $Y = Z - X$ (and hence $Z = X \cup Y$).

Figure 8.8
The DIVISION operation. (a) Dividing SSN_PNOS by SMITH_PNOS. (b) $T \leftarrow R \div S$.



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 S restrict the numerator relation R 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 as they can be computed by another operation, as illustrated in the SMITH_PNOS relation in the previous example.

Figure 8.8(b) 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:

$$\begin{aligned} T1 &\leftarrow \pi_Y(R) \\ T2 &\leftarrow \pi_Y(S \times T1) - R \\ T &\leftarrow T1 - T2 \end{aligned}$$

The DIVISION operation is defined for convenience for dealing with queries that involve *universal quantification* (see Section 8.6.7) 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 just illustrated (see Section 7.1.4, queries Q3A and Q3B). Table 8.1 lists the various basic relational algebra operations we have discussed.

8.3.5 Notation for Query Trees

In this section we describe a notation typically used in relational DBMSs (RDBMSs) to represent queries internally. The notation is called a *query tree* or sometimes it is known as a *query evaluation tree* or *query execution tree*. It includes the relational algebra operations being executed and is used as a possible data structure for the internal representation of the query in an RDBMS.

A **query tree** is a tree data structure that corresponds to a relational algebra expression. It represents the input relations of the query as *leaf nodes* of the tree, and represents the relational algebra operations as internal nodes. An execution of the query tree consists of executing an internal node operation whenever its operands (represented by its child nodes) are available, and then replacing that internal node by the relation that results from executing the operation. The execution terminates when the root node is executed and produces the result relation for the query.

Figure 8.9 shows a query tree for Query 2 (see Section 6.3.1): *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.* This query is specified

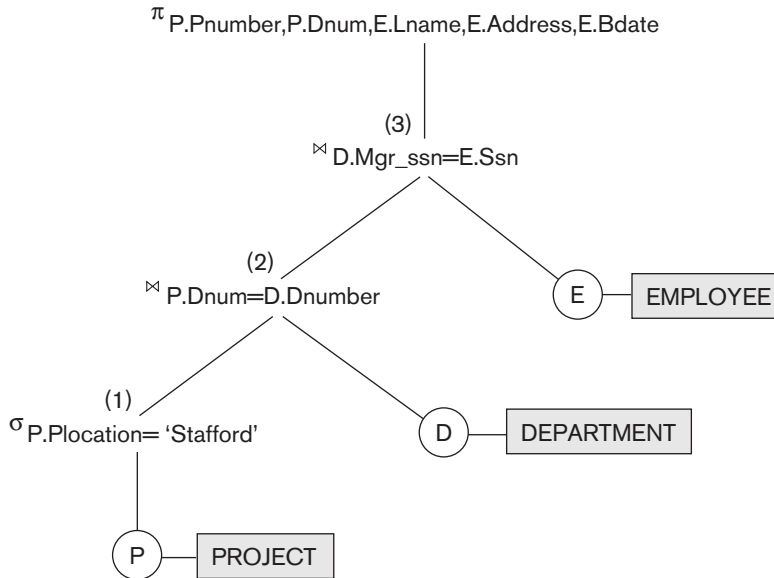
Table 8.1 Operations of Relational Algebra

OPERATION	PURPOSE	NOTATION
SELECT	Selects all tuples that satisfy the selection condition from a relation R .	$\sigma_{\langle \text{selection condition} \rangle}(R)$
PROJECT	Produces a new relation with only some of the attributes of R , and removes duplicate tuples.	$\pi_{\langle \text{attribute list} \rangle}(R)$
THETA JOIN	Produces all combinations of tuples from R_1 and R_2 that satisfy the join condition.	$R_1 \bowtie_{\langle \text{join condition} \rangle} R_2$
EQUIJOIN	Produces all the combinations of tuples from R_1 and R_2 that satisfy a join condition with only equality comparisons.	$R_1 \bowtie_{\langle \text{join condition} \rangle} R_2$, OR $R_1 \bowtie_{(\langle \text{join attributes } 1 \rangle), (\langle \text{join attributes } 2 \rangle)} R_2$
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.	$R_1^*_{\langle \text{join condition} \rangle} R_2$, OR $R_1^*_{(\langle \text{join attributes } 1 \rangle), (\langle \text{join attributes } 2 \rangle)} R_2$ 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 $Z = X \cup Y$.	$R_1(Z) \div R_2(Y)$

on the relational schema of Figure 5.5 and corresponds to the following relational algebra expression:

$$\pi_{\text{Pnumber, Dnum, Lname, Address, Bdate}}(((\sigma_{\text{Plocation}='Stafford'}(\text{PROJECT}))) \bowtie_{\text{Dnum=Dnumber}}(\text{DEPARTMENT})) \bowtie_{\text{Mgr_ssn=Ssn}}(\text{EMPLOYEE}))$$

In Figure 8.9, the three leaf nodes P, D, and E represent the three relations PROJECT, DEPARTMENT, and EMPLOYEE. The relational algebra operations in the expression are represented by internal tree nodes. The query tree signifies an explicit order of execution in the following sense. In order to execute Q2, the node marked (1) in Figure 8.9 must begin execution before node (2) because some resulting tuples of operation (1) must be available before we can begin to execute operation (2). Similarly,

**Figure 8.9**

Query tree corresponding to the relational algebra expression for Q2.

node (2) must begin to execute and produce results before node (3) can start execution, and so on. In general, a query tree gives a good visual representation and understanding of the query in terms of the relational operations it uses and is recommended as an additional means for expressing queries in relational algebra. We will revisit query trees when we discuss query processing and optimization in Chapters 18 and 19.

8.4 Additional Relational Operations

Some common database requests—which are needed in commercial applications for RDBMSs—cannot be performed with the original relational algebra operations described in Sections 8.1 through 8.3. In this section we define additional operations to express these requests. These operations enhance the expressive power of the original relational algebra.

8.4.1 Generalized Projection

The generalized projection operation extends the projection operation by allowing functions of attributes to be included in the projection list. The generalized form can be expressed as:

$$\pi_{F_1, F_2, \dots, F_n}(R)$$

where F_1, F_2, \dots, F_n are functions over the attributes in relation R and may involve arithmetic operations and constant values. This operation is helpful when developing reports where computed values have to be produced in the columns of a query result.

As an example, consider the relation

EMPLOYEE (Ssn, Salary, Deduction, Years_service)

A report may be required to show

Net Salary = Salary – Deduction,
 Bonus = 2000 * Years_service, and
 Tax = 0.25 * Salary

Then a generalized projection combined with renaming may be used as follows:

REPORT $\leftarrow \rho_{(Ssn, Net_salary, Bonus, Tax)}(\pi_{Ssn, Salary - Deduction, 2000 * Years_service, 0.25 * Salary}(EMPLOYEE))$

8.4.2 Aggregate Functions and Grouping

Another 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.

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 \mathcal{I} (pronounced *script F*)⁷, to specify these types of requests as follows:

$\langle \text{grouping attributes} \rangle \mathcal{I} \langle \text{function list} \rangle (R)$

where $\langle \text{grouping attributes} \rangle$ is a list of attributes of the relation specified in R , and $\langle \text{function list} \rangle$ is a list of ($\langle \text{function} \rangle \langle \text{attribute} \rangle$) pairs. In each such pair, $\langle \text{function} \rangle$ is one of the allowed functions—such as SUM, AVERAGE, MAXIMUM, MINIMUM, COUNT—and $\langle \text{attribute} \rangle$ 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:

$\rho_{R(Dno, No_of_employees, Average_sal)}(\mathcal{I} \text{ COUNT Ssn, AVERAGE Salary (EMPLOYEE)})$

⁷There is no single agreed-upon notation for specifying aggregate functions. In some cases a “script A” is used.

R				
(a)	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 8.10

The aggregate function operation.

- $\rho R(\text{Dno}, \text{No_of_employees}, \text{Average_sal})(\text{Dno} \bowtie \text{COUNT Ssn}, \text{AVERAGE Salary}(\text{EMPLOYEE}))$.
- $\text{Dno} \bowtie \text{COUNT Ssn}, \text{AVERAGE Salary}(\text{EMPLOYEE})$.
- $\bowtie \text{COUNT Ssn}, \text{AVERAGE Salary}(\text{EMPLOYEE})$.

The result of this operation on the EMPLOYEE relation of Figure 5.6 is shown in Figure 8.10(a).

In the preceding 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 $\langle \text{function} \rangle_ \langle \text{attribute} \rangle$.⁸ For example, Figure 8.10(b) shows the result of the following operation:

$$\text{Dno} \bowtie \text{COUNT Ssn}, \text{AVERAGE Salary}(\text{EMPLOYEE})$$

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 8.10(c) shows the result of the following operation:

$$\bowtie \text{COUNT Ssn}, \text{AVERAGE Salary}(\text{EMPLOYEE})$$

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.⁹ However, NULL values are not considered in the aggregation, as we discussed in Section 7.1.7. 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 mathematical system.

⁸Note that this is an arbitrary notation, consistent with what SQL would do.

⁹In SQL, the option of eliminating duplicates before applying the aggregate function is available by including the keyword DISTINCT (see Section 4.4.4).

8.4.3 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 `Super_ssn` 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.

It is relatively straightforward in the relational algebra to specify all employees supervised by e at a *specific level* by joining the table with itself one or more times. However, 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:

```
BORG_SSN ← πSsn(σFname='James' AND Lname='Borg'(EMPLOYEE))
SUPERVISION(Ssn1, Ssn2) ← πSsn, Super_ssn(EMPLOYEE)
RESULT1(Ssn) ← πSsn1(SUPERVISION ⋈Ssn2=Ssn 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:

```
RESULT2(Ssn) ← πSsn1(SUPERVISION ⋈Ssn2=Ssn 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:

```
RESULT ← RESULT2 ∪ RESULT1
```

The results of these queries are illustrated in Figure 8.11. 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 unless we know the maximum number of levels.¹⁰ An operation called the *transitive closure* of relations has been proposed to compute the recursive relationship as far as the recursion proceeds.

8.4.4 OUTER JOIN Operations

Next, we discuss some additional 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**

¹⁰The SQL3 standard includes syntax for recursive closure.

SUPERVISION

(Borg's Ssn is 888665555)

(Ssn)	(Super_ssn)
Ssn1	Ssn2
123456789	333445555
333445555	888665555
999887777	987654321
987654321	888665555
666884444	333445555
453453453	333445555
987987987	987654321
888665555	null

RESULT1

Ssn
333445555
987654321

(Supervised by Borg)

RESULT2

Ssn
123456789
999887777
666884444
453453453
987987987

(Supervised by
Borg's subordinates)

RESULT

Ssn
123456789
999887777
666884444
453453453
987987987
333445555
987654321

(RESULT1 \cup RESULT2)

Figure 8.11
A two-level recursive query.

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 type of join, where tuples with no match are eliminated, is known as an **inner join**. The join operations we described earlier in Section 8.3 are all inner joins. This amounts to the loss of information if the user wants the result of the JOIN to include all the tuples in one or more of the component relations.

A set of operations, called **outer joins**, were developed for the case where the user wants 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. For example, suppose that we want a list of all employee names as well as the name of the departments they manage *if they happen to manage a department*; if they do not manage one, we can indicate it

Figure 8.12

The result of a LEFT OUTER JOIN operation.

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

with a NULL value. We can apply an operation **LEFT OUTER JOIN**, denoted by \bowtie , to retrieve the result as follows:

```
TEMP  $\leftarrow$  (EMPLOYEE  $\bowtie_{\text{Ssn=Mgr\_ssn}}$  DEPARTMENT)
RESULT  $\leftarrow \pi_{\text{Fname, Minit, Lname, Dname}}(\text{TEMP})$ 
```

The **LEFT OUTER JOIN** operation keeps every tuple in the *first*, or *left*, relation R in $R \bowtie S$; 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 8.12.

A similar operation, **RIGHT OUTER JOIN**, denoted by \bowtie , keeps every tuple in the *second*, or *right*, relation S in the result of $R \bowtie S$. A third operation, **FULL OUTER JOIN**, denoted by \bowtie , 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 Section 7.1.6). These operations were provided later as an extension of relational algebra in response to the typical need in business applications to show related information from multiple tables exhaustively. Sometimes a complete reporting of data from multiple tables is required whether or not there are matching values.

8.4.5 The OUTER UNION Operation

The **OUTER UNION** operation was developed to take the union of tuples from two relations that have some common attributes, but are *not union (type) 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 $T(X, Y, Z)$. It is therefore the same as a **FULL OUTER JOIN** on the common attributes.

Two tuples t_1 in R and t_2 in S are said to **match** if $t_1[X] = t_2[X]$. 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 combination of values of the shared attributes—Name, Ssn, Department. The resulting 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, which represent a student who is also an instructor, will have values for all its attributes.¹¹

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 different 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 apply the OUTER UNION 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. For example, we could rename the attributes as MajorDept in STUDENT and WorkDept in INSTRUCTOR.

8.5 Examples of Queries in Relational Algebra

The following are additional examples to illustrate the use of the relational algebra operations. All examples refer to the database in 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.

```
RESEARCH_DEPT ←  $\sigma_{\text{Dname}=\text{'Research'}}$ (DEPARTMENT)
RESEARCH_EMPS ← (RESEARCH_DEPT  $\bowtie_{\text{Dnumber}=\text{Dno}}$  EMPLOYEE)
RESULT ←  $\pi_{\text{Fname, Lname, Address}}$ (RESEARCH_EMPS)
```

As a single in-line expression, this query becomes:

```
 $\pi_{\text{Fname, Lname, Address}}(\sigma_{\text{Dname}=\text{'Research'}}(\text{DEPARTMENT} \bowtie_{\text{Dnumber}=\text{Dno}} (\text{EMPLOYEE})))$ 
```

¹¹Note 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 to match the other join attribute name.

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.

```
STAFFORD_PROJS ←  $\sigma_{Plocation='Stafford'}(PROJECT)$ 
CONTR_DEPTS ←  $(STAFFORD\_PROJS \bowtie_{Dnum=Dnumber} DEPARTMENT)$ 
PROJ_DEPT_MGRS ←  $(CONTR\_DEPTS \bowtie_{Mgr\_ssn=Ssn} EMPLOYEE)$ 
RESULT ←  $\pi_{Pnumber, Dnum, Lname, Address, Bdate}(PROJ\_DEPT\_MGRS)$ 
```

In this example, we first select the projects located in Stafford, then join them with their controlling departments, and then join the result with the department managers. Finally, we apply a project operation on the desired attributes.

Query 3. Find the names of employees who work on *all* the projects controlled by department number 5.

```
DEPT5_PROJS ←  $\rho_{(Pno)}(\pi_{Pnumber}(\sigma_{Dnum=5}(PROJECT)))$ 
EMP_PROJ ←  $\rho_{(Ssn, Pno)}(\pi_{Essn, Pno}(WORKS\_ON))$ 
RESULT_EMP_SSNS ←  $EMP\_PROJ \div DEPT5\_PROJS$ 
RESULT ←  $\pi_{Lname, Fname}(RESULT\_EMP\_SSNS * EMPLOYEE)$ 
```

In this query, we first create a table DEPT5_PROJS that contains the project numbers of all projects controlled by department 5. Then we create a table EMP_PROJ that holds (Ssn, Pno) tuples, and apply the division operation. Notice that we renamed the attributes so that they will be correctly used in the division operation. Finally, we join the result of the division, which holds only Ssn values, with the EMPLOYEE table to retrieve the Fname, Lname attributes from 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.

```
SMITHS(Essn) ←  $\pi_{Ssn}(\sigma_{Lname='Smith'}(EMPLOYEE))$ 
SMITH_WORKER_PROJS ←  $\pi_{Pno}(WORKS\_ON * SMITHS)$ 
MGRS ←  $\pi_{Lname, Dnumber}(EMPLOYEE \bowtie_{Ssn=Mgr\_ssn} DEPARTMENT)$ 
SMITH_MANAGED_DEPTS(Dnum) ←  $\pi_{Dnumber}(\sigma_{Lname='Smith'}(MGRS))$ 
SMITH_MGR_PROJS(Pno) ←  $\pi_{Pnumber}(SMITH\_MANAGED\_DEPTS * PROJECT)$ 
RESULT ←  $(SMITH\_WORKER\_PROJS \cup SMITH\_MGR\_PROJS)$ 
```

In this query, we retrieved the project numbers for projects that involve an employee named Smith as a worker in SMITH_WORKER_PROJS. Then we retrieved the project numbers for projects that involve an employee named Smith as manager of the department that controls the project in SMITH_MGR_PROJS. Finally, we applied the

UNION operation on SMITH_WORKER_PROJS and SMITH_MGR_PROJS. As a single in-line expression, this query becomes:

$$\begin{aligned} & \pi_{Pno} (WORKS_ON \bowtie_{Essn=Ssn} (\pi_{Ssn} (\sigma_{Lname='Smith'}(EMPLOYEE)))) \cup \pi_{Pno} \\ & ((\pi_{Dnumber} (\sigma_{Lname='Smith'}(\pi_{Lname, Dnumber}(EMPLOYEE))) \bowtie \\ & Ssn=Mgr_ssn(DEPARTMENT)) \bowtie_{Dnum=ber=Dnum} PROJECT) \end{aligned}$$

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.

$$\begin{aligned} T1(Ssn, No_of_dependents) & \leftarrow_{Essn} \mathfrak{S} \text{ COUNT Dependent_name}(DEPENDENT) \\ T2 & \leftarrow \sigma_{No_of_dependents > 2}(T1) \\ \text{RESULT} & \leftarrow \pi_{Lname, Fname}(T2 * EMPLOYEE) \end{aligned}$$

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.

$$\begin{aligned} \text{ALL_EMPS} & \leftarrow \pi_{Ssn}(EMPLOYEE) \\ \text{EMPS_WITH_DEPS}(Ssn) & \leftarrow \pi_{Essn}(DEPENDENT) \\ \text{EMPS_WITHOUT_DEPS} & \leftarrow (\text{ALL_EMPS} - \text{EMPS_WITH_DEPS}) \\ \text{RESULT} & \leftarrow \pi_{Lname, Fname}(\text{EMPS_WITHOUT_DEPS} * EMPLOYEE) \end{aligned}$$

We first retrieve a relation with all employee Ssns in ALL_EMPS. Then we create a table with the Ssns of employees who have at least one dependent in EMPS_WITH_DEPS. Then we apply the SET DIFFERENCE operation to retrieve employees Ssns with no dependents in EMPS_WITHOUT_DEPS, and finally join this with EMPLOYEE to retrieve the desired attributes. As a single in-line expression, this query becomes:

$$\pi_{Lname, Fname}((\pi_{Ssn}(EMPLOYEE) - \rho_{Ssn}(\pi_{Essn}(DEPENDENT))) * EMPLOYEE)$$

Query 7. List the names of managers who have at least one dependent.

$$\begin{aligned} \text{MGRS}(Ssn) & \leftarrow \pi_{Mgr_ssn}(DEPARTMENT) \\ \text{EMPS_WITH_DEPS}(Ssn) & \leftarrow \pi_{Essn}(DEPENDENT) \\ \text{MGRS_WITH_DEPS} & \leftarrow (\text{MGRS} \cap \text{EMPS_WITH_DEPS}) \\ \text{RESULT} & \leftarrow \pi_{Lname, Fname}(\text{MGRS_WITH_DEPS} * EMPLOYEE) \end{aligned}$$

In this query, we retrieve the Ssns of managers in MGRS, and the Ssns of employees with at least one dependent in EMPS_WITH_DEPS, then we apply the SET INTERSECTION operation to get the Ssns of managers who have at least one dependent.

As we mentioned earlier, the same query can be specified in many different ways in relational algebra. In particular, 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 Q7 can be replaced by a NATURAL JOIN. As an exercise, try to do each of these sample queries using different operations.¹² We showed how to write queries as single relational algebra expressions for queries Q1, Q4, and Q6. Try to write the remaining queries as single expressions. In Chapters 6 and 7 and in Sections 8.6 and 8.7, we show how these queries are written in other relational languages.

8.6 The Tuple Relational Calculus

In this and the next section, we introduce another formal query language for the relational model called **relational calculus**. This section introduces the language known as **tuple relational calculus**, and Section 8.7 introduces a variation called **domain relational calculus**. In both variations of relational calculus, we write one **declarative** expression to specify a retrieval request; hence, there is no description of how, or *in what order*, 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 *in a particular order* of applying the operations; thus, 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.

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 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 8.4, 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. As we mentioned in the introduction to this chapter, the relational calculus is important for two reasons. First, it has a firm basis in mathematical logic. Second, the standard query language (SQL) for RDBMSs has its basic foundation in the tuple relational calculus.

Our examples refer to the database shown in Figures 5.6 and 5.7. We will use the same queries that were used in Section 8.5. Sections 8.6.6, 8.6.7, and 8.6.8 discuss dealing with universal quantifiers and safety of expression issues. Students interested in a basic introduction to tuple relational calculus may skip these sections.

¹²When queries are optimized (see Chapters 18 and 19), the system will choose a particular sequence of operations that corresponds to an execution strategy that can be executed efficiently.

8.6.1 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 $\text{COND}(t)$ is a conditional (Boolean) expression involving t that evaluates to either TRUE or FALSE for different assignments of tuples to the variable t . The result of such a query is the set of all tuples t that evaluate $\text{COND}(t)$ to TRUE. These tuples are said to **satisfy** $\text{COND}(t)$. For example, to find all employees whose salary is above \$50,000, we can write the following tuple calculus expression:

$$\{t \mid \text{EMPLOYEE}(t) \text{ AND } t.\text{Salary} > 50000\}$$

The condition $\text{EMPLOYEE}(t)$ specifies that the **range relation** of tuple variable t is EMPLOYEE . Each EMPLOYEE tuple t that satisfies the condition $t.\text{Salary} > 50000$ will be retrieved. Notice that $t.\text{Salary}$ references attribute Salary of tuple variable t ; this notation resembles how attribute names are qualified with relation names or aliases in SQL, as we saw in Chapter 6. In the notation of Chapter 5, $t.\text{Salary}$ is the same as writing $t[\text{Salary}]$.

The previous 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

$$t.\text{Fname}, t.\text{Lname} \mid \text{EMPLOYEE}(t) \text{ AND } t.\text{Salary} > 50000\}$$

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

- For each tuple variable t , the **range relation** R of t . This value is specified by a condition of the form $R(t)$. If we do not specify a range relation, then the variable t will range over all possible tuples “in the universe” as it is not restricted to any one relation.
- 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.

$$\text{Q0: } \{t.\text{Bdate}, t.\text{Address} \mid \text{EMPLOYEE}(t) \text{ AND } t.\text{Fname} = \text{'John'} \text{ AND } t.\text{Minit} = \text{'B'} \text{ AND } t.\text{Lname} = \text{'Smith'}\}$$

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.

8.6.2 Expressions and Formulas in Tuple Relational Calculus

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

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

where $t_1, t_2, \dots, t_n, t_{n+1}, \dots, t_{n+m}$ are tuple variables, each A_i is an attribute of the relation on which t_i ranges, and COND is a **condition** or **formula**¹³ 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 . It evaluates to TRUE if t_i is a tuple in the relation R , and evaluates to FALSE otherwise.
2. An atom of the form $t_i.A \text{ op } t_j.B$, 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 $t_i.A \text{ op } c$ or $c \text{ op } t_j.B$, 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.

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** (Boolean condition) is made up of one or more atoms connected via the logical operators **AND**, **OR**, and **NOT** and is defined recursively by Rules 1 and 2 as follows:

- **Rule 1:** Every atom is a formula.
- **Rule 2:** If F_1 and F_2 are formulas, then so are $(F_1 \text{ AND } F_2)$, $(F_1 \text{ OR } F_2)$, **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:

¹³Also called a **well-formed formula**, or **WFF**, in mathematical logic.

- a. $(F_1 \text{ AND } F_2)$ is TRUE if both F_1 and F_2 are TRUE; otherwise, it is FALSE.
- b. $(F_1 \text{ OR } F_2)$ 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.

8.6.3 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— $(F_1 \text{ AND } F_2)$, $(F_1 \text{ OR } F_2)$, **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 $F = (F_1 \text{ AND } F_2)$ or $F = (F_1 \text{ OR } F_2)$, 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:

$F_1: d.\text{Dname} = \text{'Research'}$
 $F_2: (\exists t)(d.\text{Dnumber} = t.\text{Dno})$
 $F_3: (\forall d)(d.\text{Mgr_ssn} = \text{'33344555'})$

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:

- **Rule 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.
- **Rule 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.

8.6.4 Sample Queries in Tuple Relational Calculus

We will use some of the same queries from Section 8.5 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. List the name and address of all employees who work for the ‘Research’ department.

Q1: $\{t.Fname, t.Lname, t.Address \mid EMPLOYEE(t) \text{ AND } (\exists d)(DEPARTMENT(d) \text{ AND } d.Dname = \text{‘Research’ AND } d.Dnumber = t.Dno)\}$

The *only free tuple variables* in a tuple relational calculus expression should be those that appear to the left of the bar (\mid). In Q1, t is the only free variable; it is then *bound successively* to each tuple. If a tuple *satisfies the conditions* specified after the bar 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 $d.Dname = \text{‘Research’}$ 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 is similar in purpose to the (INNER) JOIN operation (see Section 8.3).

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: $\{p.Pnumber, p.Dnum, m.Lname, m.Bdate, m.Address \mid PROJECT(p) \text{ AND } EMPLOYEE(m) \text{ AND } p.Plocation = \text{‘Stafford’ AND } ((\exists d)(DEPARTMENT(d) \text{ AND } p.Dnum = d.Dnumber \text{ AND } d.Mgr_ssn = m.Ssn))\}$

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 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: $\{e.Fname, e.Lname, s.Fname, s.Lname \mid EMPLOYEE(e) \text{ AND } EMPLOYEE(s) \text{ AND } e.Super_ssn = s.Ssn\}$

Query 3’. List the name of each employee who works on *some* project controlled by department number 5. This is a variation of Q3 in which *all* is changed to *some*. In this case we need two join conditions and two existential quantifiers.

Q0’: $\{e.Lname, e.Fname \mid EMPLOYEE(e) \text{ AND } ((\exists x)(\exists w)(PROJECT(x) \text{ AND } WORKS_ON(w) \text{ AND } x.Dnum = 5 \text{ AND } w.Essn = e.Ssn \text{ AND } x.Pnumber = w.Pno)))\}$

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: $\{ p.Pnumber \mid \text{PROJECT}(p) \text{ AND } (((\exists e)(\exists w)(\text{EMPLOYEE}(e) \text{ AND WORKS_ON}(w) \text{ AND } w.Pno=p.Pnumber \text{ AND } e.Lname='Smith' \text{ AND } e.Ssn=w.Essn)) \text{ OR } ((\exists m)(\exists d)(\text{EMPLOYEE}(m) \text{ AND DEPARTMENT}(d) \text{ AND } p.Dnum=d.Dnumber \text{ AND } d.Mgr_ssn=m.Ssn \text{ AND } m.Lname='Smith')))) \}$

Compare this with the relational algebra version of this query in Section 8.5. The UNION operation in relational algebra can usually be substituted with an OR connective in relational calculus.

8.6.5 Notation for Query Graphs

In this section, we describe a notation that has been proposed to represent relational calculus queries that do not involve complex quantification in a graphical form. These types of queries are known as **select-project-join queries** because they only involve these three relational algebra operations. The notation may be expanded to more general queries, but we do not discuss these extensions here. This graphical representation of a query is called a **query graph**. Figure 8.13 shows the query graph for Q2. Relations in the query are represented by **relation nodes**, which are displayed as single circles. Constant values, typically from the query selection conditions, are represented by **constant nodes**, which are displayed as double circles or ovals. Selection and join conditions are represented by the graph **edges** (the lines that connect the nodes), as shown in Figure 8.13. Finally, the attributes to be retrieved from each relation are displayed in square brackets above each relation.

The query graph representation does not indicate a particular order to specify which operations to perform first, and is hence a more neutral representation of a select-project-join query than the query tree representation (see Section 8.3.5), where the order of execution is implicitly specified. There is only a single query graph corresponding to each query. Although some query optimization techniques were based on query graphs, it is now generally accepted that query trees are preferable because,

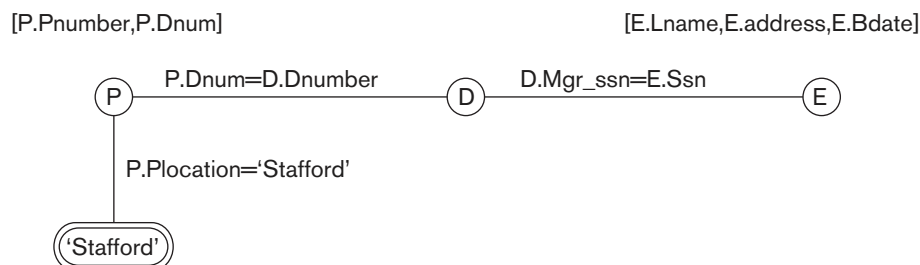


Figure 8.13
Query graph for Q2.

in practice, the query optimizer needs to show the order of operations for query execution, which is not possible in query graphs.

In the next section we discuss the relationship between the universal and existential quantifiers and show how one can be transformed into the other.

8.6.6 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 \equiv symbol stands for **equivalent to**:

$$\begin{aligned}
 (\forall x) (P(x)) &\equiv \text{NOT } (\exists x) (\text{NOT } (P(x))) \\
 (\exists x) (P(x)) &\equiv \text{NOT } (\forall x) (\text{NOT } (P(x))) \\
 (\forall x) (P(x) \text{ AND } Q(x)) &\equiv \text{NOT } (\exists x) (\text{NOT } (P(x)) \text{ OR NOT } (Q(x))) \\
 (\forall x) (P(x) \text{ OR } Q(x)) &\equiv \text{NOT } (\exists x) (\text{NOT } (P(x)) \text{ AND NOT } (Q(x))) \\
 (\exists x) (P(x)) \text{ OR } Q(x) &\equiv \text{NOT } (\forall x) (\text{NOT } (P(x)) \text{ AND NOT } (Q(x))) \\
 (\exists x) (P(x) \text{ AND } Q(x)) &\equiv \text{NOT } (\forall x) (\text{NOT } (P(x)) \text{ OR NOT } (Q(x)))
 \end{aligned}$$

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

$$\begin{aligned}
 (\forall x)(P(x)) &\Rightarrow (\exists x)(P(x)) \\
 \text{NOT } (\exists x)(P(x)) &\Rightarrow \text{NOT } (\forall x)(P(x))
 \end{aligned}$$

8.6.7 Using the Universal Quantifier in Queries

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 the query Q3.

Query 3. List the names of employees who work on *all* the projects controlled by department number 5. One way to specify this query is to use the universal quantifier as shown:

Q3: $\{e.\text{Lname}, e.\text{Fname} \mid \text{EMPLOYEE}(e) \text{ AND } ((\forall x)(\text{NOT}(\text{PROJECT}(x)) \text{ OR NOT } (x.\text{Dnum}=5) \text{ OR } ((\exists w)(\text{WORKS_ON}(w) \text{ AND } w.\text{Essn}=e.\text{Ssn} \text{ AND } x.\text{Pnumber}=w.\text{Pno}))))))\}$

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

Q3: $\{e.\text{Lname}, e.\text{Fname} \mid \text{EMPLOYEE}(e) \text{ AND } F'\}$
 $F' = ((\forall x)(\text{NOT}(\text{PROJECT}(x)) \text{ OR } F_1))$
 $F_1 = \text{NOT}(x.\text{Dnum}=5) \text{ OR } F_2$
 $F_2 = ((\exists w)(\text{WORKS_ON}(w) \text{ AND } w.\text{Essn}=e.\text{Ssn} \text{ AND } x.\text{Pnumber}=w.\text{Pno}))$

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. Therefore, we can 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 \neq 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 8.6.6, we can rephrase the query in Q3 as shown in Q3A, which uses a negated existential quantifier instead of the universal quantifier:

Q3A: $\{e.Lname, e.Fname \mid \text{EMPLOYEE}(e) \text{ AND } (\text{NOT } (\exists x) (\text{PROJECT}(x) \text{ AND } (x.Dnum=5) \text{ and } (\text{NOT } (\exists w)(\text{WORKS_ON}(w) \text{ AND } w.Essn=e.Ssn \text{ AND } x.Pnumber=w.Pno))))\}$

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

Query 6. List the names of employees who have no dependents.

Q6: $\{e.Fname, e.Lname \mid \text{EMPLOYEE}(e) \text{ AND } (\text{NOT } (\exists d)(\text{DEPENDENT}(d) \text{ AND } e.Ssn=d.Essn))\}$

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

Q6A: $\{e.Fname, e.Lname \mid \text{EMPLOYEE}(e) \text{ AND } ((\forall d)(\text{NOT}(\text{DEPENDENT}(d) \text{ OR NOT}(e.Ssn=d.Essn))))\}$

Query 7. List the names of managers who have at least one dependent.

Q7: $\{e.Fname, e.Lname \mid \text{EMPLOYEE}(e) \text{ AND } ((\exists d)(\exists p)(\text{DEPARTMENT}(d) \text{ AND } \text{DEPENDENT}(p) \text{ AND } e.Ssn=d.Mgr_ssn \text{ AND } p.Essn=e.Ssn))\}$

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

8.6.8 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

$\{t \mid \text{NOT } (\text{EMPLOYEE}(t))\}$

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. For example, the domain of $\{t \mid \text{NOT}(\text{EMPLOYEE}(t))\}$ 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 $\{t \mid \text{NOT}(\text{EMPLOYEE}(t))\}$ 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.

8.7 The Domain Relational Calculus

There is another type of relational calculus called the domain relational calculus, or simply **domain calculus**. Historically, while SQL (see Chapters 6 and 7), which was based on tuple relational calculus, was being developed by IBM Research at San Jose, California, another language called QBE (Query-By-Example), which is related to domain calculus, was being developed almost concurrently at the IBM T. J. Watson Research Center in Yorktown Heights, New York. The formal specification of the domain calculus was proposed after the development of the QBE language and 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

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

where $x_1, x_2, \dots, x_n, x_{n+1}, x_{n+2}, \dots, x_{n+m}$ 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 $R(x_1, x_2, \dots, x_j)$, 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 $\langle x_1, x_2, \dots, x_j \rangle$ must be a tuple in the relation whose name is R , where x_i is the value of the i th 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:

$$\{x_1, x_2, \dots, x_n \mid R(x_1 x_2 x_3) \text{ AND } \dots\}$$

instead of:

$$\{x_1, x_2, \dots, x_n \mid R(x_1, x_2, x_3) \text{ AND } \dots\}$$

2. An atom of the form $x_i \text{ op } x_j$, 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 $x_i \text{ op } c$ or $c \text{ op } x_j$, 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, \dots, x, y, z for domain variables.

Query 0. List the birth date and address of the employee whose name is ‘John B. Smith’.

Q0: $\{u, v \mid (\exists q) (\exists r) (\exists s) (\exists t) (\exists w) (\exists x) (\exists y) (\exists z)$
 $(\text{EMPLOYEE}(qrstuvwxyz) \text{ AND } q=\text{‘John’ AND } r=\text{‘B’ AND } s=\text{‘Smith’})\}$

We need ten variables for the EMPLOYEE relation, one to range over each of the domains of attributes of EMPLOYEE in order. Of the ten variables q, r, s, \dots, z , only u and v are free, because they appear to the left of the bar and hence should not be bound to a quantifier. 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 (\mid)—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 equal to ‘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.¹⁴

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:¹⁵

Q0A: $\{u, v \mid \text{EMPLOYEE}(\text{‘John’}, \text{‘B’}, \text{‘Smith’}, t, u, v, w, x, y, z)\}$

Query 1. Retrieve the name and address of all employees who work for the ‘Research’ department.

Q1: $\{q, s, v \mid (\exists z) (\exists l) (\exists m) (\text{EMPLOYEE}(qrstuvwxyz) \text{ AND}$
 $\text{DEPARTMENT}(lmno) \text{ AND } l=\text{‘Research’ AND } m=z)\}$

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 $l = \text{‘Research’}$, 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.

¹⁴Quantifying only the domain variables actually used in conditions and specifying a predicate such as $\text{EMPLOYEE}(qrstuvwxyz)$ without separating domain variables with commas is an abbreviated notation used for convenience; it is not the correct formal notation.

¹⁵Again, this is not a formally accurate notation.

Q2: $\{i, k, s, u, v \mid (\exists j)(\exists m)(\exists n)(\exists t)(\text{PROJECT}(hijk) \text{ AND } \text{EMPLOYEE}(qrstuvwxyz) \text{ AND } \text{DEPARTMENT}(lmno) \text{ AND } k=m \text{ AND } n=t \text{ AND } j=\text{'Stafford'})\}$

Query 6. List the names of employees who have no dependents.

Q6: $\{q, s \mid (\exists t)(\text{EMPLOYEE}(qrstuvwxyz) \text{ AND } (\text{NOT}(\exists l)(\text{DEPENDENT}(lmnop) \text{ AND } t=l)))\}$

Q6 can be restated using universal quantifiers instead of the existential quantifiers, as shown in Q6A:

Q6A: $\{q, s \mid (\exists t)(\text{EMPLOYEE}(qrstuvwxyz) \text{ AND } ((\forall l)(\text{NOT}(\text{DEPENDENT}(lmnop)) \text{ OR } \text{NOT}(t=l))))\}$

Query 7. List the names of managers who have at least one dependent.

Q7: $\{s, q \mid (\exists t)(\exists j)(\exists l)(\text{EMPLOYEE}(qrstuvwxyz) \text{ AND } \text{DEPARTMENT}(hijk) \text{ AND } \text{DEPENDENT}(lmnop) \text{ AND } t=j \text{ AND } l=t)\}$

As we mentioned earlier, it can be shown that any query that can be expressed in the basic 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 basic relational algebra.

The 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 Query Management Facility (QMF) interface option to DB2. The basic ideas used in QBE have been applied in several other commercial products. Because of its important place in the history of relational languages, we have included an overview of QBE in Appendix C.

8.8 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.

In Sections 8.1 through 8.3, we introduced the basic relational algebra operations and illustrated the types of queries for which each is used. First, we discussed the unary relational operators SELECT and PROJECT, as well as the RENAME operation. Then, we discussed binary set theoretic operations requiring that relations on which they are applied be union (or type) 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. Query trees were introduced as a graphical representation of relational algebra queries, which can also be used as the basis for internal data structures that the DBMS can use to represent a query.

We discussed some important types of queries that *cannot* be stated with the basic relational algebra operations but are important for practical situations. We introduced GENERALIZED PROJECTION to use functions of attributes in the projection list and the AGGREGATE FUNCTION operation to deal with aggregate types of statistical requests that summarize the information in the tables. We discussed recursive queries, for which there is no direct support in the algebra but which can be handled in a step-by-step approach, as we demonstrated. Then we 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 *declarative* 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 introduced query graphs as an internal representation for queries in relational calculus. We also discussed the existential quantifier (\exists) and the universal quantifier (\forall). 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 the basic relational algebra. There is no analog to grouping and aggregation functions in basic relational calculus, although some extensions have been suggested.

Review Questions

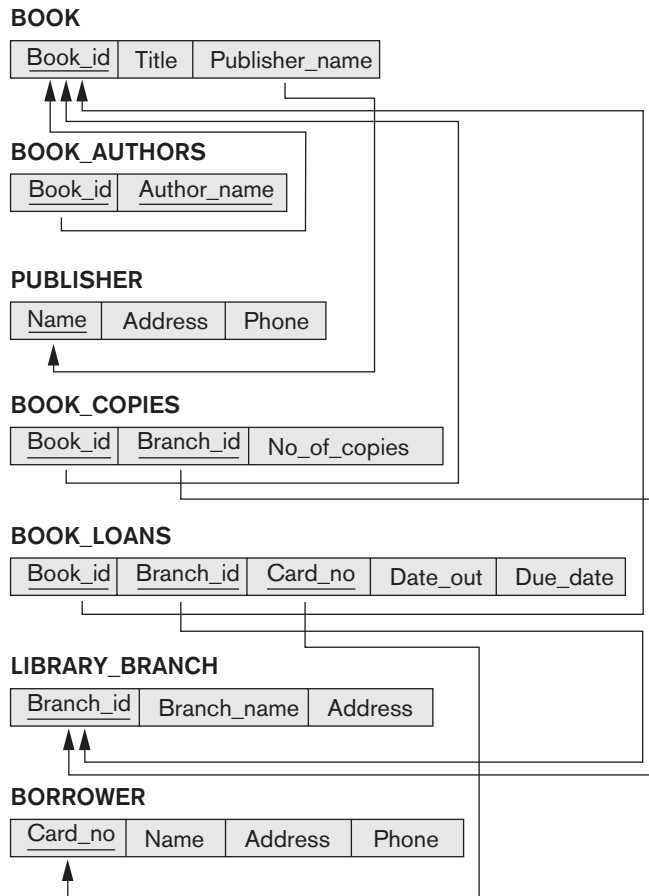
- 8.1. List the operations of relational algebra and the purpose of each.
- 8.2. What is union compatibility? Why do the UNION, INTERSECTION, and DIFFERENCE operations require that the relations on which they are applied be union compatible?
- 8.3. Discuss some types of queries for which renaming of attributes is necessary in order to specify the query unambiguously.
- 8.4. Discuss the various types of *inner join* operations. Why is theta join required?

- 8.5. What role does the concept of *foreign key* play when specifying the most common types of meaningful join operations?
- 8.6. What is the FUNCTION operation? For what is it used?
- 8.7. How are the OUTER JOIN operations different from the INNER JOIN operations? How is the OUTER UNION operation different from UNION?
- 8.8. In what sense does relational calculus differ from relational algebra, and in what sense are they similar?
- 8.9. How does tuple relational calculus differ from domain relational calculus?
- 8.10. Discuss the meanings of the existential quantifier (\exists) and the universal quantifier (\forall).
- 8.11. Define the following terms with respect to the tuple calculus: *tuple variable*, *range relation*, *atom*, *formula*, and *expression*.
- 8.12. Define the following terms with respect to the domain calculus: *domain variable*, *range relation*, *atom*, *formula*, and *expression*.
- 8.13. What is meant by a *safe expression* in relational calculus?
- 8.14. When is a query language called relationally complete?

Exercises

- 8.15. Show the result of each of the sample queries in Section 8.5 as it would apply to the database state in Figure 5.6.
- 8.16. Specify the following queries on the COMPANY relational 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 in 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.
- 8.17.** Consider the AIRLINE relational database schema shown in Figure 5.8, which was described in Exercise 5.12. 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 '2009-10-09'.
- 8.18.** Consider the LIBRARY relational database schema shown in Figure 8.14, which is used to keep track of books, borrowers, and book loans. Referential integrity constraints are shown as directed arcs in Figure 8.14, 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 Due_date 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.
- 8.19.** Specify the following queries in relational algebra on the database schema given in Exercise 5.14:

**Figure 8.14**

A relational database schema for a LIBRARY database.

- a. List the Order# and Ship_date for all orders shipped from Warehouse# 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 Cname, No_of_orders, 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.
- 8.20.** Specify the following queries in relational algebra on the database schema given in Exercise 5.15:
- a. Give the details (all attributes of trip relation) for trips that exceeded \$2,000 in expenses.

- b. Print the Ssns of salespeople who took trips to Honolulu.
- c. Print the total trip expenses incurred by the salesperson with SSN = '234-56-7890'.
- 8.21.** Specify the following queries in relational algebra on the database schema given in Exercise 5.16:
- List the number of courses taken by all students named John Smith in Winter 2009 (i.e., Quarter=W09).
 - 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.
 - List any department that has all its adopted books published by 'Pearson Publishing'.
- 8.22.** Consider the two tables $T1$ and $T2$ shown in Figure 8.15. Show the results of the following operations:
- $T1 \bowtie_{T1.P = T2.A} T2$
 - $T1 \bowtie_{T1.Q = T2.B} T2$
 - $T1 \bowtie_{T1.P = T2.A} T2$
 - $T1 \bowtie_{T1.Q = T2.B} T2$
 - $T1 \cup T2$
 - $T1 \bowtie_{(T1.P = T2.A \text{ AND } T1.R = T2.C)} T2$
- 8.23.** Specify the following queries in relational algebra on the database schema in Exercise 5.17:
- For the salesperson named 'Jane Doe', list the following information for all the cars she sold: Serial#, Manufacturer, Sale_price.
 - List the Serial# and Model of cars that have no options.
 - Consider the NATURAL JOIN operation between SALESPERSON and SALE. What is the meaning of a left outer join for these tables (do not change the order of relations)? Explain with an example.
 - Write a query in relational algebra involving selection and one set operation and say in words what the query does.
- 8.24.** Specify queries a, b, c, e, f, i, and j of Exercise 8.16 in both tuple and domain relational calculus.
- 8.25.** Specify queries a, b, c, and d of Exercise 8.17 in both tuple and domain relational calculus.

Figure 8.15

A database state for the relations $T1$ and $T2$.

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

- 8.26.** Specify queries c, d, and f of Exercise 8.18 in both tuple and domain relational calculus.
- 8.27.** 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?
- 8.28.** Rewrite the domain relational calculus queries that followed Q0 in Section 8.7 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.
- 8.29.** 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
- $(\forall x)(P(x)) \equiv \text{NOT}(\exists x)(\text{NOT}(P(x)))$.
 - $(\text{IF } P \text{ THEN } Q) \equiv (\text{NOT}(P) \text{ OR } Q)$.
- 8.30.** Show how you can specify the following relational algebra operations in both tuple and domain relational calculus.
- a. $\sigma_{A=C}(R(A, B, C))$
 - b. $\pi_{\langle A, B \rangle}(R(A, B, C))$
 - 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. $R(A, B) \div S(A)$
- 8.31.** Suggest extensions to the relational calculus so that it may express the following types of operations that were discussed in Section 8.4: (a) aggregate functions and grouping; (b) OUTER JOIN operations; (c) recursive closure queries.
- 8.32.** A nested query is a query within a query. More specifically, a nested query is a parenthesized query whose result can be used as a value in a number of places, such as instead of a relation. Specify the following queries on the database specified in Figure 5.5 using the concept of nested queries and the relational operators discussed in this chapter. Also show the result of each query as it would apply to the database state in Figure 5.6.
- a. List the names of all employees who work in the department that has the employee with the highest salary among all employees.

- b. List the names of all employees whose supervisor's supervisor has '888665555' for Ssn.
 - c. List the names of employees who make at least \$10,000 more than the employee who is paid the least in the company.
- 8.33.** State whether the following conclusions are true or false:
- a. **NOT** ($P(x)$ **OR** $Q(x)$) \rightarrow (**NOT** ($P(x)$) **AND** (**NOT** ($Q(x)$)))
 - b. **NOT** ($\exists x$) ($P(x)$) $\rightarrow \forall x$ (**NOT** ($P(x)$))
 - c. ($\exists x$) ($P(x)$) $\rightarrow \forall x$ ($(P(x))$)

Laboratory Exercises

- 8.34.** Specify and execute the following queries in relational algebra (RA) using the RA interpreter on the COMPANY database schema in Figure 5.5.
- a. List 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. List the names of employees who are directly supervised by Franklin Wong.
 - d. List the names of employees who work on every project.
 - e. List the names of employees who do not work on any project.
 - f. List the names and addresses of employees who work on at least one project located in Houston but whose department has no location in Houston.
 - g. List the names of department managers who have no dependents.
- 8.35.** Consider the following MAILORDER relational schema describing the data for a mail order company.

PARTS(Pno, Pname, Qoh, Price, Olevel)
 CUSTOMERS(Cno, Cname, Street, Zip, Phone)
 EMPLOYEES(Eno, Ename, Zip, Hdate)
 ZIP_CODES(Zip, City)
 ORDERS(Ono, Cno, Eno, Received, Shipped)
 ODETAILS(Ono, Pno, Qty)

Qoh stands for *quantity on hand*: the other attribute names are self-explanatory. Specify and execute the following queries using the RA interpreter on the MAILORDER database schema.

- a. Retrieve the names of parts that cost less than \$20.00.
- b. Retrieve the names and cities of employees who have taken orders for parts costing more than \$50.00.
- c. Retrieve the pairs of customer number values of customers who live in the same ZIP Code.

- d. Retrieve the names of customers who have ordered parts from employees living in Wichita.
- e. Retrieve the names of customers who have ordered parts costing less than \$20.00.
- f. Retrieve the names of customers who have not placed an order.
- g. Retrieve the names of customers who have placed exactly two orders.

8.36. Consider the following GRADEBOOK relational schema describing the data for a grade book of a particular instructor. (*Note:* The attributes A, B, C, and D of COURSES store grade cutoffs.)

CATALOG(Cno, Ctitle)
STUDENTS(Sid, Fname, Lname, Minit)
COURSES(Term, Sec_no, Cno, A, B, C, D)
ENROLLS(Sid, Term, Sec_no)

Specify and execute the following queries using the RA interpreter on the GRADEBOOK database schema.

- a. Retrieve the names of students enrolled in the Automata class during the fall 2009 term.
- b. Retrieve the Sid values of students who have enrolled in CSc226 and CSc227.
- c. Retrieve the Sid values of students who have enrolled in CSc226 or CSc227.
- d. Retrieve the names of students who have not enrolled in any class.
- e. Retrieve the names of students who have enrolled in all courses in the CATALOG table.

8.37. Consider a database that consists of the following relations.

SUPPLIER(Sno, Sname)
PART(Pno, Pname)
PROJECT(Jno, Jname)
SUPPLY(Sno, Pno, Jno)

The database records information about suppliers, parts, and projects and includes a ternary relationship between suppliers, parts, and projects. This relationship is a many-many-many relationship. Specify and execute the following queries using the RA interpreter.

- a. Retrieve the part numbers that are supplied to exactly two projects.
- b. Retrieve the names of suppliers who supply more than two parts to project 'J1'.
- c. Retrieve the part numbers that are supplied by every supplier.
- d. Retrieve the project names that are supplied by supplier 'S1' only.
- e. Retrieve the names of suppliers who supply at least two different parts each to at least two different projects.

- 8.38.** Specify and execute the following queries for the database in Exercise 5.16 using the RA interpreter.
- Retrieve the names of students who have enrolled in a course that uses a textbook published by Addison-Wesley-Longman.
 - Retrieve the names of courses in which the textbook has been changed at least once.
 - Retrieve the names of departments that adopt textbooks published by Addison-Wesley only.
 - Retrieve the names of departments that adopt textbooks written by Navathe and published by Addison-Wesley.
 - Retrieve the names of students who have never used a book (in a course) written by Navathe and published by Addison-Wesley.
- 8.39.** Repeat Laboratory Exercises 8.34 through 8.38 in domain relational calculus (DRC) by using the DRC interpreter.

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 (1977a). The experimental version of the Query-By-Example system is described in Zloof (1975). The ILL (Lacroix & Pirotte, 1977b) 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 Working Conference (e.g., Arisawa & Catarci (2000) or Zhou & Pu (2002)) present a number of proposals for such languages.

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

This chapter discusses how to *design a relational database schema* based on a conceptual schema design. Figure 3.1 presented a high-level view of the database design process. In this chapter we focus on the **logical database design** step of database design, which is also known as **data model mapping**. 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 through 8. Many computer-aided software engineering (CASE) tools are based on the ER or EER models, or other similar models, as we have discussed in Chapters 3 and 4. Many tools use ER or EER diagrams or variations to develop the schema graphically and collect information about the data types and constraints, then convert the ER/EER schema automatically into a relational database schema in the DDL of a specific relational DBMS. The design tools employ algorithms similar to the ones presented in this chapter.

We outline a seven-step algorithm in Section 9.1 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 9.2, we continue the mapping algorithm by describing how to map EER model constructs—specialization/generalization and union types (categories)—into relations. Section 9.3 summarizes the chapter.

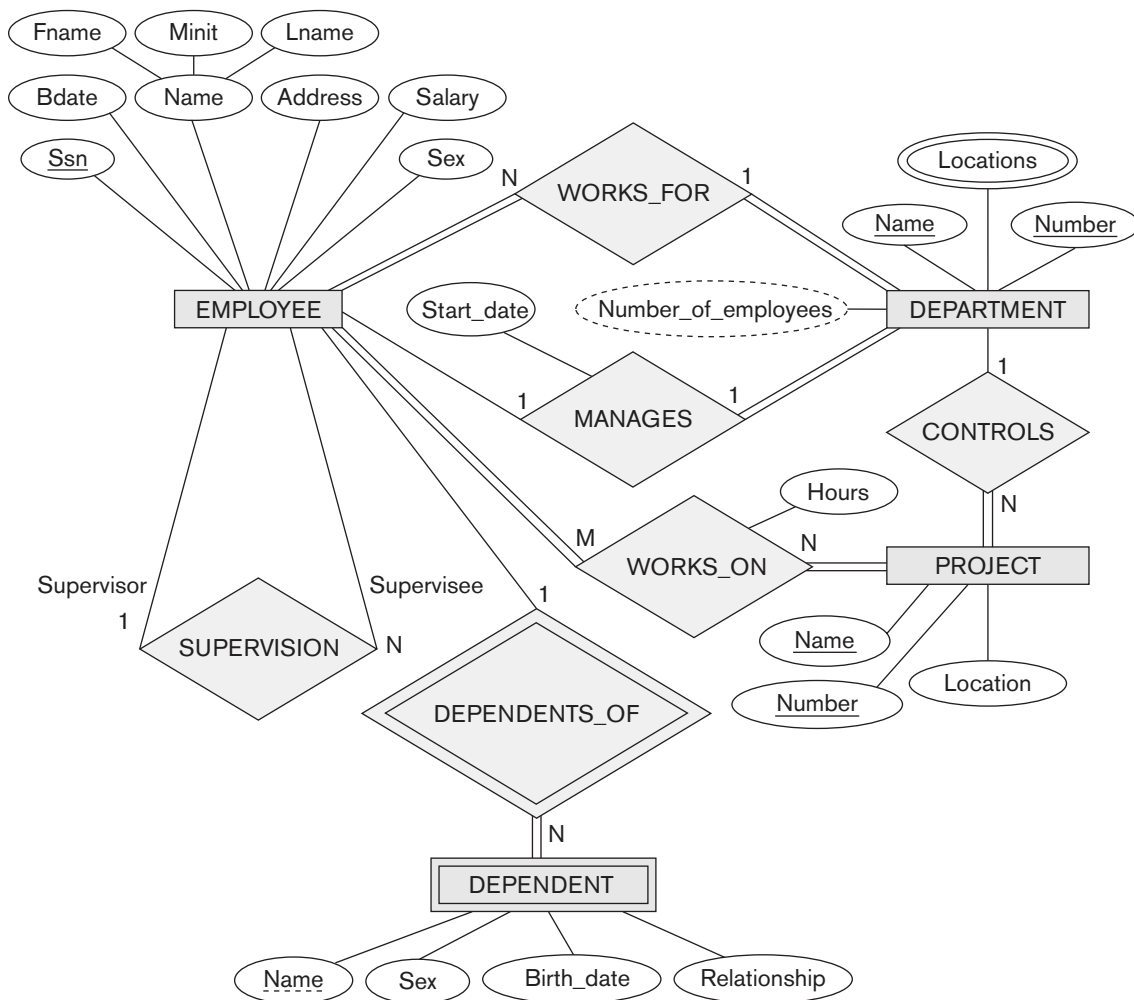
9.1 Relational Database Design Using ER-to-Relational Mapping

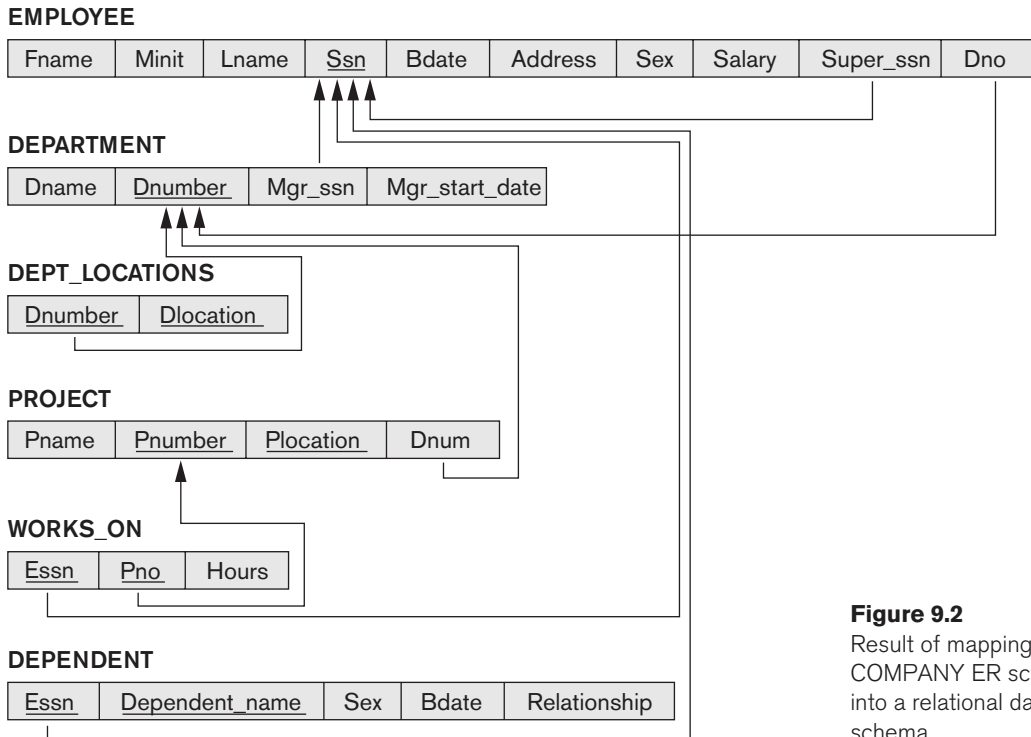
9.1.1 ER-to-Relational Mapping Algorithm

In this section we describe the steps of an algorithm for ER-to-relational mapping. We use the COMPANY database example to illustrate the mapping procedure. The COMPANY ER schema is shown again in Figure 9.1, and the corresponding COMPANY relational database schema is shown in Figure 9.2 to illustrate the

Figure 9.1

The ER conceptual schema diagram for the COMPANY database.



**Figure 9.2**

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

mapping steps. We assume that the mapping will create tables with simple single-valued attributes. The relational model constraints defined in Chapter 5, which include primary keys, unique keys (if any), and referential integrity constraints on the relations, will also be specified in the mapping results.

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 the primary key for R . If the chosen key of E is a composite, then 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 additional (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 9.2 to correspond to the regular entity types EMPLOYEE, DEPARTMENT, and PROJECT from Figure 9.1. The foreign key and relationship attributes, if any, are not included yet; they will be added during subsequent steps. These include

the attributes Super_ssn and Dno of EMPLOYEE, Mgr_ssn and Mgr_start_date 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 unique 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 represents an entity instance. The result after this mapping step is shown in Figure 9.3(a).

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 mapping 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 (see Figure 9.3(b)). 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 rename it Essn, although this is not

Figure 9.3
Illustration of some mapping steps.
(a) *Entity* relations after step 1.
(b) Additional *weak entity* relation after step 2.
(c) *Relationship* relations after step 5.
(d) Relation representing multivalued attribute after step 6.

(a)

Fname	Minit	Lname	<u>Ssn</u>	Bdate	Address	Sex	Salary
-------	-------	-------	------------	-------	---------	-----	--------

DEPARTMENT

Dname	<u>Dnumber</u>
-------	----------------

PROJECT

Pname	<u>Pnumber</u>	Plocation
-------	----------------	-----------

(b)

DEPENDENT

<u>Essn</u>	<u>Dependent_name</u>	Sex	Bdate	Relationship
-------------	-----------------------	-----	-------	--------------

(c)

WORKS_ON

<u>Essn</u>	<u>Pno</u>	Hours
-------------	------------	-------

(d)

DEPT_LOCATIONS

<u>Dnumber</u>	<u>Dlocation</u>
----------------	------------------

necessary. The primary key of the DEPENDENT relation is the combination {Essn, Dependent_name}, because Dependent_name (also renamed from Name in Figure 9.1) is the partial key of DEPENDENT.

It is common to choose the propagate (CASCADE) option for the referential triggered action (see Section 6.2) 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. The first approach 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 9.1 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 to Mgr_ssn. We also include the simple attribute Start_date of the MANAGES relationship type in the DEPARTMENT relation and rename it Mgr_start_date (see Figure 9.2).

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. This would be a bad choice, because if only 2% of employees manage a department, then 98% 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 creates redundancy and incurs a penalty for consistency maintenance.

2. **Merged relation approach:** An alternative mapping of a 1:1 relationship type is to merge the two entity types and the relationship into a single relation. This is possible when *both participations are total*, as this would indicate that the two tables will have the exact same number of tuples at all times.
3. **Cross-reference or relationship relation approach:** The third option 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 will 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 from T . The relation R will include the primary key attributes of S and T as foreign keys to S and T . The primary key of R will be one of the two foreign keys, and the other foreign key will be a unique key of R . The drawback is having an extra relation, and requiring extra join operations when combining related tuples from the tables.

Step 4: Mapping of Binary 1:N Relationship Types. There are two possible approaches: (1) the foreign key approach and (2) the cross-reference or relationship relation approach. The first approach is generally preferred as it reduces the number of tables.

1. **The foreign key approach:** 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 ; we do this 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 .

To apply this approach to our example, we map the 1:N relationship types `WORKS_FOR`, `CONTROLS`, and `SUPERVISION` from Figure 9.1. 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 `Super_ssn`. The `CONTROLS` relationship is mapped to the foreign key attribute `Dnum` of `PROJECT`, which references the primary key `Dnumber` of the `DEPARTMENT` relation. These foreign keys are shown in Figure 9.2.

2. **The relationship relation approach:** An alternative approach is to use the **relationship relation** (cross-reference) option as in the third option for binary 1:1 relationships. We create a separate relation R whose attributes are the primary keys of S and T , which will also be foreign keys to S and T . The primary key of R is the same as the primary 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. In the traditional relational model with no multivalued attributes, the only option for M:N relationships is the **relationship relation (cross-reference) option**. 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 9.1 by creating the relation WORKS_ON in Figure 9.2. We include the primary keys of the PROJECT and EMPLOYEE relations as foreign keys in WORKS_ON and rename them Pno and Essn, respectively (renaming is *not required*; it is a design choice). 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}. This **relationship relation** is shown in Figure 9.3(c).

The propagate (CASCADE) option for the referential triggered action (see Section 4.2) 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.

Although we can 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 is only recommended 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.

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 a multivalued 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 (see Figure 9.3(d)). The attribute Dlocation represents the multivalued attribute LOCATIONS of DEPARTMENT, whereas 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. It is important to note that in more recent versions of the relational model that allow array data types, the multivalued attribute can be mapped to an array attribute rather than requiring a separate table.

The propagate (CASCADE) option for the referential triggered action (see Section 6.2) 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 9.2 shows the COMPANY relational database schema obtained with steps 1 through 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 9.1 ; 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. We use the **relationship relation option**. For each n -ary relationship type R , where $n > 2$, create a new relationship 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 the discussion in Section 3.9.2 concerning constraints on n -ary relationships).

Consider the ternary relationship type SUPPLY in Figure 3.17, which relates a SUPPLIER s , PART p , and PROJECT j whenever s is currently supplying p to j ; this can be mapped to the relation SUPPLY shown in Figure 9.4, whose primary key is the combination of the three foreign keys {Sname, Part_no, Proj_name}.

9.1.2 Discussion and Summary of Mapping for ER Model Constructs

Table 9.1 summarizes the correspondences between ER and relational model constructs and constraints.

Figure 9.4
Mapping the n -ary relationship type SUPPLY from Figure 3.17(a).

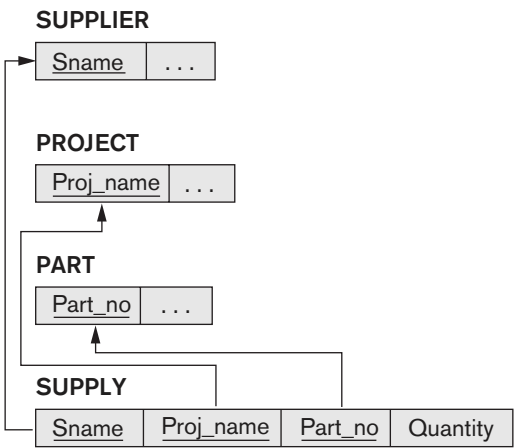


Table 9.1 Correspondence between ER and Relational Models

ER MODEL	RELATIONAL MODEL
Entity type	<i>Entity</i> relation
1:1 or 1:N relationship type	Foreign key (or <i>relationship</i> relation)
M:N relationship type	<i>Relationship</i> relation and <i>two</i> foreign keys
<i>n</i> -ary relationship type	<i>Relationship</i> relation and <i>n</i> 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

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 1:N relationship type is involved and the foreign key mapping is used, a single join operation is usually needed. When the relationship relation approach is used, such as 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 in Figure 9.2. Hence, we must apply the EQUIJOIN operation to the EMPLOYEE and WORKS_ON relations with the join condition EMPLOYEE.Ssn = WORKS_ON.Essn, and then apply another EQUIJOIN operation to the resulting relation and the PROJECT relation with join condition WORKS_ON.Pno = PROJECT.Pnumber. In general, when multiple relationships need to be traversed, numerous join operations must be specified. The 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 data. For example, the reader can try joining the PROJECT and DEPT_LOCATIONS relations on the condition Dlocation = Plocation and examine the result.

In the relational schema 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 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 in Figure 3.6; 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 a set of tuples of the form $\{ \langle '1', 'Houston' \rangle, \langle '4', 'Stafford' \rangle, \langle '5', 'Bellaire' \rangle, \langle '5', 'Sugarland' \rangle, \langle '5', 'Houston' \rangle \}$ from the DEPT_LOCATIONS relation in Figure 3.6. This is a serious drawback of the basic normalized or *flat* version of the relational model. The object data model and object-relational systems (see Chapter 12) do allow multivalued attributes by using the array type for the attribute.

9.2 Mapping EER Model Constructs to Relations

In this section, we discuss the mapping of EER model constructs to relations by extending the ER-to-relational mapping algorithm that was presented in Section 9.1.1.

9.2.1 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. The two main options are to map the whole specialization into a **single table**, or to map it into **multiple tables**. Within each option are variations that depend on the constraints on the specialization/generalization.

We can add a further step to our ER-to-relational mapping algorithm from Section 9.1.1, 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 discuss the conditions under which each option should be used. We use $\text{Attrs}(R)$ to denote *the attributes of a relation R*, and $\text{PK}(R)$ to denote *the primary key of R*. First we describe the mapping formally, then we illustrate it with examples.

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

- **Option 8A: Multiple relations—superclass and subclasses.** Create a relation L for C with attributes $\text{Attrs}(L) = \{k, a_1, \dots, a_n\}$ and $\text{PK}(L) = k$. Create a relation L_i for each subclass S_i , $1 \leq i \leq m$, with the attributes $\text{Attrs}(L_i) = \{k\} \cup \{\text{attributes of } S_i\}$ and $\text{PK}(L_i) = k$. 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 $\text{Attrs}(L_i) = \{\text{attributes of } S_i\} \cup \{k, a_1, \dots, a_n\}$ and $\text{PK}(L_i) = k$. 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). Additionally, it is only recommended if the specialization has the *disjointness constraint* (see Section 4.3.1). If the specialization is *overlapping*, the same entity may be duplicated in several relations.
- **Option 8C: Single relation with one type attribute.** Create a single relation L with attributes $\text{Attrs}(L) = \{k, a_1, \dots, a_n\} \cup \{\text{attributes of } S_1\} \cup \dots \cup \{\text{attributes of } S_m\} \cup \{t\}$ and $\text{PK}(L) = k$. The attribute t is called a **type** (or **discriminating**) attribute whose value 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 (local) attributes exist in the subclasses.
- **Option 8D: Single relation with multiple type attributes.** Create a single relation schema L with attributes $\text{Attrs}(L) = \{k, a_1, \dots, a_n\} \cup \{\text{attributes of } S_1\} \cup \dots \cup \{\text{attributes of } S_m\} \cup \{t_1, t_2, \dots, t_m\}$ and $\text{PK}(L) = k$. Each t_i , $1 \leq i \leq m$, is a **Boolean type attribute** indicating whether or not a tuple belongs to subclass S_i . This option is used for a specialization whose subclasses are *overlapping* (but will also work for a disjoint specialization).

Options 8A and 8B are the **multiple-relation options**, whereas options 8C and 8D are 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 (local) attributes of S_i , plus the primary key of the superclass C , which is propagated to L_i and becomes its primary key. It also becomes a foreign key to the superclass relation. 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 9.5(a) for the EER schema in Figure 4.4. Option 8A works for any constraints on the specialization: disjoint or overlapping, total or partial. Notice that the constraint

$$\pi_{\langle k \rangle}(L_i) \subseteq \pi_{\langle k \rangle}(L)$$

must hold for each L_i . This specifies a foreign key from each L_i to L .

In option 8B, the EQUIJOIN operation between each subclass and the superclass is *built into* the schema and the superclass relation L is done away with, as illustrated in Figure 9.5(b) for the EER specialization in Figure 4.3(b). 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

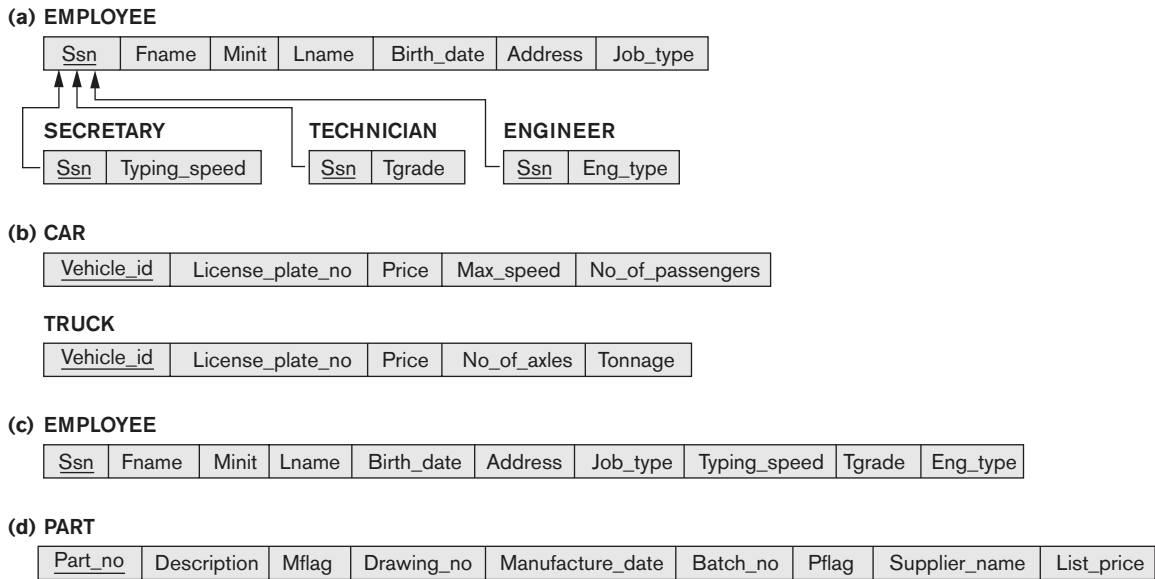


Figure 9.5
Options for mapping specialization or generalization. (a) Mapping the EER schema in Figure 4.4 using option 8A. (b) Mapping the EER schema in Figure 4.3(b) using option 8B. (c) Mapping the EER schema in Figure 4.4 using option 8C. (d) Mapping Figure 4.5 using option 8D with Boolean type fields Mflag and Pflag.

inherited attributes from the superclass C stored redundantly in more than one table 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 (see Section 6.4) 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 (local) attributes of these subclasses. These options are not recommended if many specific attributes are defined for the subclasses. If few local subclass attributes exist, however, these mappings are preferable to options 8A and 8B because they do away with the need to specify JOIN operations; therefore, they can yield a more efficient implementation for queries.

Option 8C is used to handle disjoint subclasses by including a single **type** (or **image** or **discriminating**) **attribute** t to indicate to which of the m subclasses each tuple belongs; hence, the domain of t could be $\{1, 2, \dots, m\}$. 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 itself serves the purpose of t and t is not needed; this option is illustrated in Figure 9.5(c) for the EER specialization in Figure 4.4.

Option 8D is designed to handle overlapping subclasses by including m Boolean **type** (or **flag**) 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 type attributes—*ls_a_secretary*, *ls_a_engineer*, and *ls_a_technician*—instead of the *Job_type* attribute in Figure 9.5(c). Figure 9.5(d) shows the mapping of the specialization from Figure 4.5 using option 8D.

For 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 9.6 shows one possible mapping into relations for the EER lattice in Figure 4.6. Here we used option 8A for PERSON/{EMPLOYEE, ALUMNUS, STUDENT}, and option 8C for EMPLOYEE/{STAFF, FACULTY, STUDENT_ASSISTANT} by including the type attribute *Employee_type*. We then used the single-table option 8D for STUDENT_ASSISTANT/{RESEARCH_ASSISTANT, TEACHING_ASSISTANT} by including the type attributes *Ta_flag* and *Ra_flag* in EMPLOYEE. We also used option 8D for STUDENT/STUDENT_ASSISTANT by including the type attributes *Student_assist_flag* in STUDENT, and for STUDENT/{GRADUATE_STUDENT, UNDERGRADUATE_STUDENT} by including the type attributes *Grad_flag* and *Undergrad_flag* in STUDENT. In Figure 9.6, all attributes whose names end with *type* or *flag* are type fields.

9.2.2 Mapping of Shared Subclasses (Multiple Inheritance)

A shared subclass, such as ENGINEERING_MANAGER in 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 (union type) as we discussed in Section 4.4. 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 9.6, options 8C and 8D are used for the shared subclass STUDENT_ASSISTANT. Option 8C is used in the EMPLOYEE relation (*Employee_type* attribute) and option 8D is used in the STUDENT relation (*Student_assist_flag* attribute).

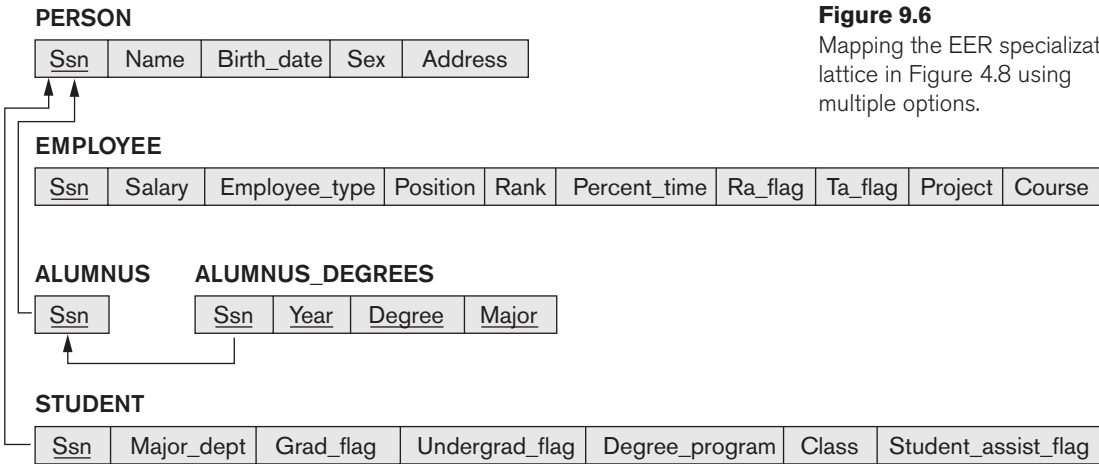


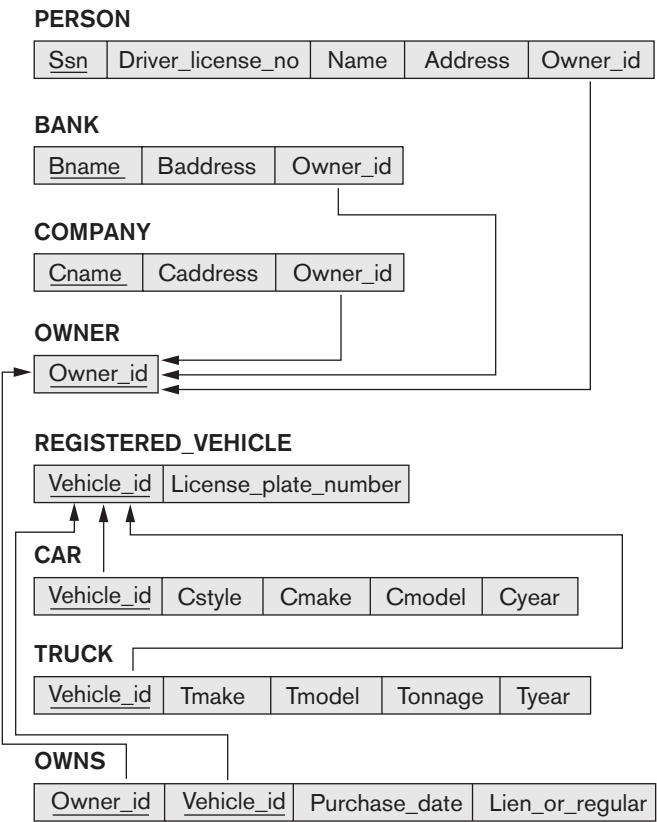
Figure 9.6
Mapping the EER specialization lattice in Figure 4.8 using multiple options.

9.2.3 Mapping of Categories (Union Types)

We 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 (see Section 4.4). An example is the OWNER category shown in Figure 4.8, 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 union type. The keys of the defining classes are different, so we cannot use any one of them exclusively to identify all entities in the relation. In our example in Figure 4.8, we create a relation OWNER to correspond to the OWNER category, as illustrated in Figure 9.7, and include any attributes of the category in this relation. The primary key of the OWNER relation is the surrogate key, which we called Owner_id. We also

Figure 9.7
Mapping the EER categories
(union types) in Figure 4.8 to
relations.



include the surrogate key attribute `Owner_id` 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 original 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 `Owner_id` attribute in its corresponding tuple in the PERSON (or BANK or COMPANY) relation, and it would not have a tuple in the OWNER relation. It is also recommended to add a type attribute (not shown in Figure 9.7) to the OWNER relation to indicate the particular entity type to which each tuple belongs (PERSON or BANK or COMPANY).

For a category whose superclasses have the same key, such as VEHICLE in Figure 4.8, there is no need for a surrogate key. The mapping of the REGISTERED_VEHICLE category, which illustrates this case, is also shown in Figure 9.7.

9.3 Summary

In Section 9.1, 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 9.1 summarized the correspondences between the ER and relational model constructs and constraints. Next, we added additional steps to the algorithm in Section 9.2 for mapping the constructs from the EER model into the relational model. Similar algorithms are incorporated into graphical database design tools to create a relational schema from a conceptual schema design automatically.

Review Questions

- 9.1. (a) 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.
(b) Discuss the options for mapping EER model constructs to relations, and the conditions under which each option could be used.

Exercises

- 9.2. Map the UNIVERSITY database schema shown in Figure 3.20 into a relational database schema.
- 9.3. Try to map the relational schema in Figure 6.14 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.

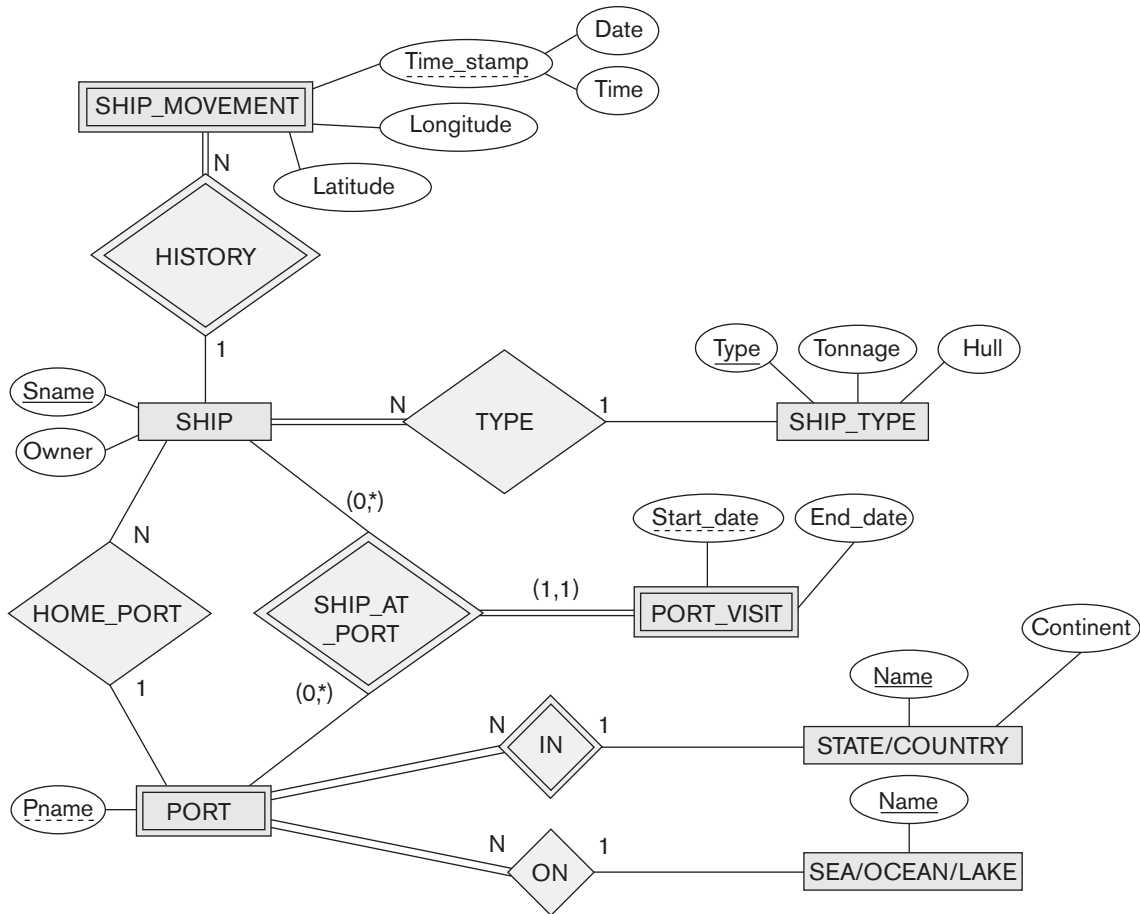


Figure 9.8

An ER schema for a SHIP_TRACKING database.

- 9.4. Figure 9.8 shows an ER schema for a database that can 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.
- 9.5. Map the BANK ER schema of Exercise 3.23 (shown in Figure 3.21) into a relational schema. Specify all primary keys and foreign keys. Repeat for the AIRLINE schema (Figure 3.20) of Exercise 3.19 and for the other schemas for Exercises 3.16 through 3.24.
- 9.6. Map the EER diagrams in Figures 4.9 and 4.12 into relational schemas. Justify your choice of mapping options.
- 9.7. Is it possible to successfully map a binary M:N relationship type without requiring a new relation? Why or why not?

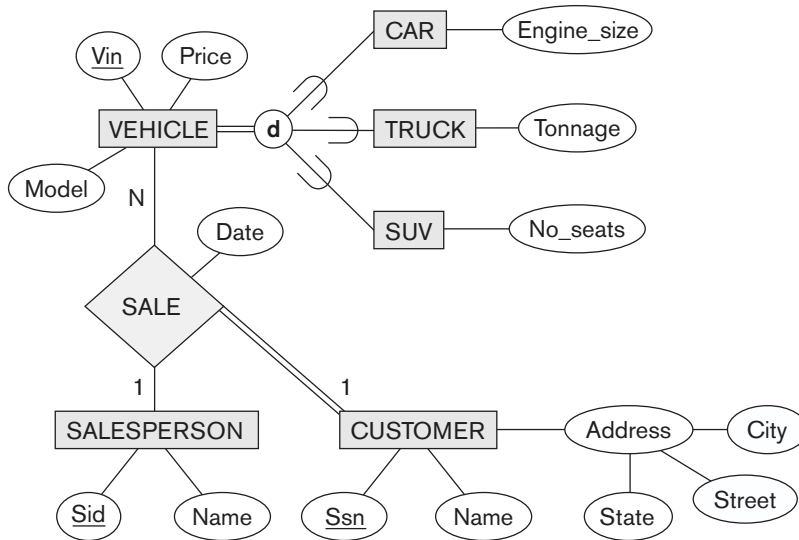


Figure 9.9
EER diagram for
a car dealer.

9.8. Consider the EER diagram in Figure 9.9 for a car dealer.

Map the EER schema into a set of relations. For the VEHICLE to CAR/TRUCK/SUV generalization, consider the four options presented in Section 9.2.1 and show the relational schema design under each of those options.

9.9. Using the attributes you provided for the EER diagram in Exercise 4.27, map the complete schema into a set of relations. Choose an appropriate option out of 8A thru 8D from Section 9.2.1 in doing the mapping of generalizations and defend your choice.

Laboratory Exercises

- 9.10.** Consider the ER design for the UNIVERSITY database that was modeled using a tool like ERwin or Rational Rose in Laboratory Exercise 3.31. Using the SQL schema generation feature of the modeling tool, generate the SQL schema for an Oracle database.
- 9.11.** Consider the ER design for the MAIL_ORDER database that was modeled using a tool like ERwin or Rational Rose in Laboratory Exercise 3.32. Using the SQL schema generation feature of the modeling tool, generate the SQL schema for an Oracle database.
- 9.12.** Consider the ER design for the CONFERENCE_REVIEW database that was modeled using a tool like ERwin or Rational Rose in Laboratory Exercise 3.34. Using the SQL schema generation feature of the modeling tool, generate the SQL schema for an Oracle database.

- 9.13. Consider the EER design for the `GRADE_BOOK` database that was modeled using a tool like ERwin or Rational Rose in Laboratory Exercise 4.28. Using the SQL schema generation feature of the modeling tool, generate the SQL schema for an Oracle database.
- 9.14. Consider the EER design for the `ONLINE_AUCTION` database that was modeled using a tool like ERwin or Rational Rose in Laboratory Exercise 4.29. Using the SQL schema generation feature of the modeling tool, generate the SQL schema for an Oracle database.

Selected Bibliography

The original ER-to-relational mapping algorithm was described in Chen's classic paper (Chen, 1976). Batini et al. (1992) discuss a variety of mapping algorithms from ER and EER models to legacy models and vice versa.