CHAPTER 6

Database Design Using the E-R Model

Up to this point in the text, we have assumed a given database schema and studied how queries and updates are expressed. We now consider how to design a database schema in the first place. In this chapter, we focus on the entity-relationship data model (E-R), which provides a means of identifying entities to be represented in the database and how those entities are related. Ultimately, the database design will be expressed in terms of a relational database design and an associated set of constraints. We show in this chapter how an E-R design can be transformed into a set of relation schemas and how some of the constraints can be captured in that design. Then, in Chapter 7, we consider in detail whether a set of relation schemas is a good or bad database design and study the process of creating good designs using a broader set of constraints. These two chapters cover the fundamental concepts of database design.

6.1 Overview of the Design Process

The task of creating a database application is a complex one, involving design of the database schema, design of the programs that access and update the data, and design of a security scheme to control access to data. The needs of the users play a central role in the design process. In this chapter, we focus on the design of the database schema, although we briefly outline some of the other design tasks later in the chapter.

6.1.1 Design Phases

For small applications, it may be feasible for a database designer who understands the application requirements to decide directly on the relations to be created, their attributes, and constraints on the relations. However, such a direct design process is difficult for real-world applications, since they are often highly complex. Often no one person understands the complete data needs of an application. The database designer must interact with users of the application to understand the needs of the application, represent them in a high-level fashion that can be understood by the users, and

then translate the requirements into lower levels of the design. A high-level data model serves the database designer by providing a conceptual framework in which to specify, in a systematic fashion, the data requirements of the database users, and a database structure that fulfills these requirements.

- The initial phase of database design is to characterize fully the data needs of the prospective database users. The database designer needs to interact extensively with domain experts and users to carry out this task. The outcome of this phase is a specification of user requirements. While there are techniques for diagrammatically representing user requirements, in this chapter we restrict ourselves to textual descriptions of user requirements.
- Next, the designer chooses a data model and, by applying the concepts of the chosen data model, translates these requirements into a conceptual schema of the database. The schema developed at this **conceptual-design** phase provides a detailed overview of the enterprise. The entity-relationship model, which we study in the rest of this chapter, is typically used to represent the conceptual design. Stated in terms of the entity-relationship model, the conceptual schema specifies the entities that are represented in the database, the attributes of the entities, the relationships among the entities, and constraints on the entities and relationships. Typically, the conceptual-design phase results in the creation of an entity-relationship diagram that provides a graphic representation of the schema.

  The designer reviews the schema to confirm that all data requirements are indeed satisfied and are not in conflict with one another. She can also examine the design to remove any redundant features. Her focus at this point is on describing the data and their relationships, rather than on specifying physical storage details.
- A fully developed conceptual schema also indicates the functional requirements of the enterprise. In a **specification of functional requirements**, users describe the kinds of operations (or transactions) that will be performed on the data. Example operations include modifying or updating data, searching for and retrieving specific data, and deleting data. At this stage of conceptual design, the designer can review the schema to ensure that it meets functional requirements.
- The process of moving from an abstract data model to the implementation of the database proceeds in two final design phases.
  - In the **logical-design phase**, the designer maps the high-level conceptual schema onto the implementation data model of the database system that will be used. The implementation data model is typically the relational data model, and this step typically consists of mapping the conceptual schema defined using the entity-relationship model into a relation schema.
  - Finally, the designer uses the resulting system-specific database schema in the subsequent **physical-design phase**, in which the physical features of the database

are specified. These features include the form of file organization and choice of index structures, discussed in Chapter 13 and Chapter 14.

The physical schema of a database can be changed relatively easily after an application has been built. However, changes to the logical schema are usually harder to carry out, since they may affect a number of queries and updates scattered across application code. It is therefore important to carry out the database design phase with care, before building the rest of the database application.

6.1.2 Design Alternatives

A major part of the database design process is deciding how to represent in the design the various types of "things" such as people, places, products, and the like. We use the term *entity* to refer to any such distinctly identifiable item. In a university database, examples of entities would include instructors, students, departments, courses, and course offerings. We assume that a course may have run in multiple semesters, as well as multiple times in a semester; we refer to each such offering of a course as a section. The various entities are related to each other in a variety of ways, all of which need to be captured in the database design. For example, a student takes a course offering, while an instructor teaches a course offering; teaches and takes are examples of relationships between entities.

In designing a database schema, we must ensure that we avoid two major pitfalls:

1. **Redundancy:** A bad design may repeat information. For example, if we store the course identifier and title of a course with each course offering, the title would be stored redundantly (i.e., multiple times, unnecessarily) with each course offering. It would suffice to store only the course identifier with each course offering, and to associate the title with the course identifier only once, in a course entity.

   Redundancy can also occur in a relational schema. In the university example we have used so far, we have a relation with section information and a separate relation with course information. Suppose that instead we have a single relation where we repeat all of the course information (course_id, title, dept_name, credits) once for each section (offering) of the course. Information about courses would then be stored redundantly.

   The biggest problem with such redundant representation of information is that the copies of a piece of information can become inconsistent if the information is updated without taking precautions to update all copies of the information. For example, different offerings of a course may have the same course identifier, but may have different titles. It would then become unclear what the correct title of the course is. Ideally, information should appear in exactly one place.

2. **Incompleteness:** A bad design may make certain aspects of the enterprise difficult or impossible to model. For example, suppose that, as in case (1) above, we only had entities corresponding to course offering, without having an entity

corresponding to courses. Equivalently, in terms of relations, suppose we have a single relation where we repeat all of the course information once for each section that the course is offered. It would then be impossible to represent information about a new course, unless that course is offered. We might try to make do with the problematic design by storing null values for the section information. Such a work-around is not only unattractive but may be prevented by primary-key constraints.

Avoiding bad designs is not enough. There may be a large number of good designs from which we must choose. As a simple example, consider a customer who buys a product. Is the sale of this product a relationship between the customer and the product? Alternatively, is the sale itself an entity that is related both to the customer and to the product? This choice, though simple, may make an important difference in what aspects of the enterprise can be modeled well. Considering the need to make choices such as this for the large number of entities and relationships in a real-world enterprise, it is not hard to see that database design can be a challenging problem. Indeed we shall see that it requires a combination of both science and "good taste."

6.2 The Entity-Relationship Model

The **entity-relationship** (**E-R**) **data model** was developed to facilitate database design by allowing specification of an *enterprise schema* that represents the overall logical structure of a database.

The E-R model is very useful in mapping the meanings and interactions of real-world enterprises onto a conceptual schema. Because of this usefulness, many database-design tools draw on concepts from the E-R model. The E-R data model employs three basic concepts: entity sets, relationship sets, and attributes. The E-R model also has an associated diagrammatic representation, the E-R diagram. As we saw briefly in Section 1.3.1, an **E-R diagram** can express the overall logical structure of a database graphically. E-R diagrams are simple and clear—qualities that may well account in large part for the widespread use of the E-R model.

The Tools section at the end of the chapter provides information about several diagram editors that you can use to create E-R diagrams.

6.2.1 Entity Sets

An **entity** is a "thing" or "object" in the real world that is distinguishable from all other objects. For example, each person in a university is an entity. An entity has a set of properties, and the values for some set of properties must uniquely identify an entity. For instance, a person may have a *person_id* property whose value uniquely identifies that person. Thus, the value 677-89-9011 for *person_id* would uniquely identify one particular person in the university. Similarly, courses can be thought of as entities, and *course_id* uniquely identifies a course entity in the university. An entity may be concrete, such

as a person or a book, or it may be abstract, such as a course, a course offering, or a flight reservation.

An **entity set** is a set of entities of the same type that share the same properties, or attributes. The set of all people who are instructors at a given university, for example, can be defined as the entity set *instructor*. Similarly, the entity set *student* might represent the set of all students in the university.

In the process of modeling, we often use the term *entity set* in the abstract, without referring to a particular set of individual entities. We use the term **extension** of the entity set to refer to the actual collection of entities belonging to the entity set. Thus, the set of actual instructors in the university forms the extension of the entity set *instructor*. This distinction is similar to the difference between a relation and a relation instance, which we saw in Chapter 2.

Entity sets do not need to be disjoint. For example, it is possible to define the entity set *person* consisting of all people in a university. A *person* entity may be an *instructor* entity, a *student* entity, both, or neither.

An entity is represented by a set of **attributes.** Attributes are descriptive properties possessed by each member of an entity set. The designation of an attribute for an entity set expresses that the database stores similar information concerning each entity in the entity set; however, each entity may have its own value for each attribute. Possible attributes of the *instructor* entity set are *ID*, *name*, *dept_name*, and *salary*. In real life, there would be further attributes, such as street number, apartment number, state, postal code, and country, but we generally omit them to keep our examples simple. Possible attributes of the *course* entity set are *course_id*, *title*, *dept_name*, and *credits*.

In this section we consider only attributes that are **simple**— those not divided into subparts. In Section 6.3, we discuss more complex situations where attributes can be composite and multivalued.

Each entity has a **value** for each of its attributes. For instance, a particular *instructor* entity may have the value 12121 for *ID*, the value Wu for *name*, the value Finance for *dept_name*, and the value 90000 for *salary*.

The *ID* attribute is used to identify instructors uniquely, since there may be more than one instructor with the same name. Historically, many enterprises found it convenient to use a government-issued identification number as an attribute whose value uniquely identifies the person. However, that is considered bad practice for reasons of security and privacy. In general, the enterprise would have to create and assign its own unique identifier for each instructor.

A database thus includes a collection of entity sets, each of which contains any number of entities of the same type. A database for a university may include a number of other entity sets. For example, in addition to keeping track of instructors and students, the university also has information about courses, which are represented by the entity set *course* with attributes *course_id*, *title*, *dept_name* and *credits*. In a real setting, a university database may keep dozens of entity sets.

An entity set is represented in an E-R diagram by a **rectangle**, which is divided into two parts. The first part, which in this text is shaded blue, contains the name of

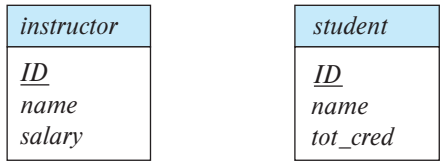


Figure 6.1 E-R diagram showing entity sets *instructor* and *student*.

the entity set. The second part contains the names of all the attributes of the entity set. The E-R diagram in Figure 6.1 shows two entity sets *instructor* and *student*. The attributes associated with *instructor* are *ID*, *name*, and *salary*. The attributes associated with *student* are *ID*, *name*, and *tot_cred*. Attributes that are part of the primary key are underlined (see Section 6.5).

6.2.2 Relationship Sets

A **relationship** is an association among several entities. For example, we can define a relationship *advisor* that associates instructor Katz with student Shankar. This relationship specifies that Katz is an advisor to student Shankar. A **relationship set** is a set of relationships of the same type.

Consider two entity sets *instructor* and *student*. We define the relationship set *advisor* to denote the associations between students and the instructors who act as their advisors. Figure 6.2 depicts this association. To keep the figure simple, only some of the attributes of the two entity sets are shown.

A **relationship instance** in an E-R schema represents an association between the named entities in the real-world enterprise that is being modeled. As an illustration, the individual *instructor* entity Katz, who has instructor *ID* 45565, and the *student* entity Shankar, who has student *ID* 12345, participate in a relationship instance of *advi-*

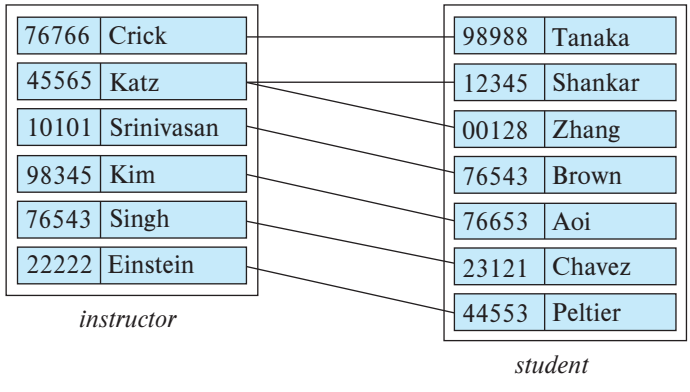


Figure 6.2 Relationship set *advisor* (only some attributes of *instructor* and *student* are shown).

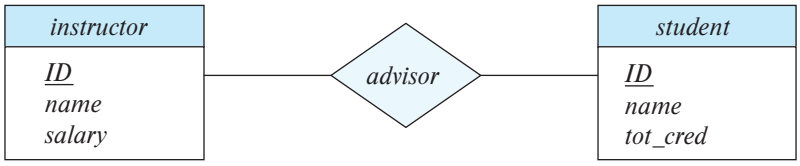


Figure 6.3 E-R diagram showing relationship set *advisor*.

sor. This relationship instance represents that in the university, the instructor Katz is advising student Shankar.

A relationship set is represented in an E-R diagram by a **diamond**, which is linked via **lines** to a number of different entity sets (rectangles). The E-R diagram in Figure 6.3 shows the two entity sets *instructor* and *student*, related through a binary relationship set *advisor*.

As another example, consider the two entity sets *student* and *section*, where *section* denotes an offering of a course. We can define the relationship set *takes* to denote the association between a student and a section in which that student is enrolled.

Although in the preceding examples each relationship set was an association between two entity sets, in general a relationship set may denote the association of more than two entity sets.

Formally, a **relationship set** is a mathematical relation on $n \geq 2$ (possibly nondistinct) entity sets. If $E_1, E_2, \ldots, E_n$ are entity sets, then a relationship set R is a subset of

$$\{(e_1, e_2, \ldots, e_n) \mid e_1 \in E_1, e_2 \in E_2, \ldots, e_n \in E_n\}$$

where $(e_1, e_2, \ldots, e_n)$ is a relationship instance.

The association between entity sets is referred to as participation; i.e., the entity sets $E_1, E_2, \ldots, E_n$ **participate** in relationship set R.

The function that an entity plays in a relationship is called that entity's **role**. Since entity sets participating in a relationship set are generally distinct, roles are implicit and are not usually specified. However, they are useful when the meaning of a relationship needs clarification. Such is the case when the entity sets of a relationship set are not distinct; that is, the same entity set participates in a relationship set more than once, in different roles. In this type of relationship set, sometimes called a **recursive** relationship set, explicit role names are necessary to specify how an entity participates in a relationship instance. For example, consider the entity set *course* that records information about all the courses offered in the university. To depict the situation where one course (C2) is a prerequisite for another course (C1) we have relationship set *prereq* that is modeled by ordered pairs of *course* entities. The first course of a pair takes the role of course C1, whereas the second takes the role of prerequisite course C2. In this way, all relationships of *prereq* are characterized by (C1, C2) pairs; (C2, C1) pairs are excluded. We indicate roles in E-R diagrams by labeling the lines that connect diamonds

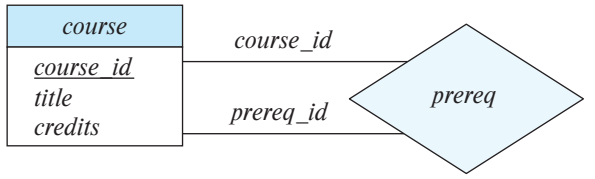


Figure 6.4 E-R diagram with role indicators.

to rectangles. Figure 6.4 shows the role indicators *course_id* and *prereq_id* between the *course* entity set and the *prereq* relationship set.

A relationship may also have attributes called **descriptive attributes**. As an example of descriptive attributes for relationships, consider the relationship set *takes* which relates entity sets *student* and *section*. We may wish to store a descriptive attribute *grade* with the relationship to record the grade that a student received in a course offering.

An attribute of a relationship set is represented in an E-R diagram by an **undivided rectangle**. We link the rectangle with a dashed line to the diamond representing that relationship set. For example, Figure 6.5 shows the relationship set *takes* between the entity sets *section* and *student*. We have the descriptive attribute *grade* attached to the relationship set *takes*. A relationship set may have multiple descriptive attributes; for example, we may also store a descriptive attribute *for_credit* with the *takes* relationship set to record whether a student has taken the section for credit, or is auditing (or sitting in on) the course.

Observe that the attributes of the two entity sets have been omitted from the E-R diagram in Figure 6.5, with the understanding that they are specified elsewhere in the complete E-R diagram for the university; we have already seen the attributes for *student*, and we will see the attributes of *section* later in this chapter. Complex E-R designs may need to be split into multiple diagrams that may be located in different pages. Relationship sets should be shown in only one location, but entity sets may be repeated in more than one location. The attributes of an entity set should be shown in the first occurrence. Subsequent occurrences of the entity set should be shown without attributes, to avoid repetition of information and the resultant possibility of inconsistency in the attributes shown in different occurrences.

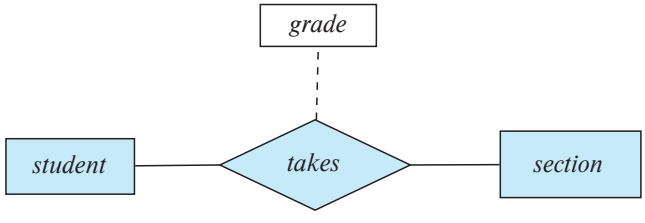


Figure 6.5 E-R diagram with an attribute attached to a relationship set.

It is possible to have more than one relationship set involving the same entity sets. For example, suppose that students may be teaching assistants for a course. Then, the entity sets *section* and *student* may participate in a relationship set *teaching_assistant*, in addition to participating in the *takes* relationship set.

The formal definition of a relationship set, which we saw earlier, defines a relationship set as a set of relationship instances. Consider the *takes* relationship between *student* and *section*. Since a set cannot have duplicates, it follows that a particular student can have only one association with a particular section in the *takes* relationship. Thus, a student can have only one grade associated with a section, which makes sense in this case. However, if we wish to allow a student to have more than one grade for the same section, we need to have an attribute *grades* which stores a set of grades; such attributes are called multivalued attributes, and we shall see them later in Section 6.3.

The relationship sets *advisor* and *takes* provide examples of a **binary relationship set**—that is, one that involves two entity sets. Most of the relationship sets in a database system are binary. Occasionally, however, relationship sets involve more than two entity sets. The number of entity sets that participate in a relationship set is the **degree of the relationship set**. A binary relationship set is of degree 2; a **ternary relationship set** is of degree 3.

As an example, suppose that we have an entity set *project* that represents all the research projects carried out in the university. Consider the entity sets *instructor*, *student*, and *project*. Each project can have multiple associated students and multiple associated instructors. Furthermore, each student working on a project must have an associated instructor who guides the student on the project. For now, we ignore the first two relationships, between project and instructor, and between project and student. Instead, we focus on the information about which instructor is guiding which student on a particular project.

To represent this information, we relate the three entity sets through a ternary relationship set *proj_guide*, which relates entity sets *instructor*, *student*, and *project*. An instance of *proj_guide* indicates that a particular student is guided by a particular instructor on a particular project. Note that a student could have different instructors as guides for different projects, which cannot be captured by a binary relationship between students and instructors.

Nonbinary relationship sets can be specified easily in an E-R diagram. Figure 6.6 shows the E-R diagram representation of the ternary relationship set *proj_guide*.

6.3 Complex Attributes

For each attribute, there is a set of permitted values, called the **domain**, or **value set**, of that attribute. The domain of attribute *course_id* might be the set of all text strings of a certain length. Similarly, the domain of attribute *semester* might be strings from the set {Fall, Winter, Spring, Summer}.

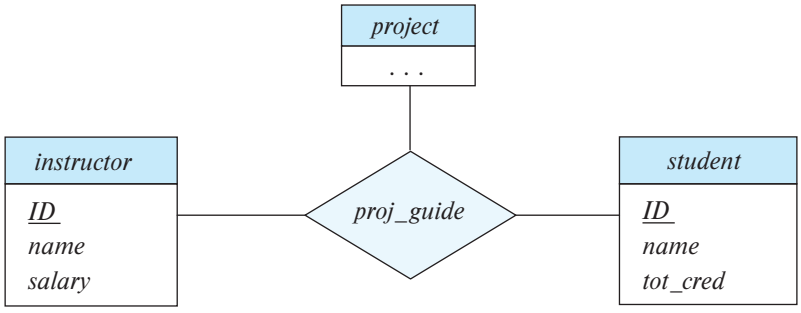


Figure 6.6 E-R diagram with a ternary relationship *proj_guide*.

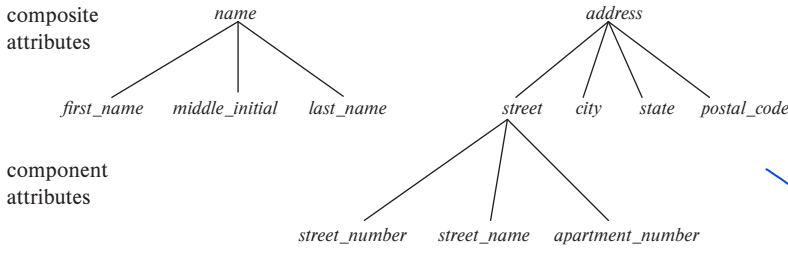


Figure 6.7 Composite attributes instructor *name* and *address*.

An attribute, as used in the E-R model, can be characterized by the following attribute types.

- **Simple** and **composite** attributes. In our examples thus far, the attributes have been **simple**; that is, they have not been divided into subparts. **Composite** attributes, on the other hand, can be divided into subparts (i.e., other attributes). For example, an attribute *name* could be structured as a composite attribute consisting of *first_name*, *middle_initial*, and *last_name*. Using composite attributes in a design schema is a good choice if a user will wish to refer to an entire attribute on some occasions, and to only a component of the attribute on other occasions. Suppose we were to add an address to the *student* entity-set. The address can be defined as the composite attribute *address* with the attributes *street*, *city*, *state*, and *postal_code*.[1] Composite attributes help us to group together related attributes, making the modeling cleaner.

  Note also that a composite attribute may appear as a hierarchy. In the composite attribute *address*, its component attribute *street* can be further divided into *street_number*, *street_name*, and *apartment_number*. Figure 6.7 depicts these examples of composite attributes for the *instructor* entity set.

[1] We assume the address format used in the United States, which includes a numeric postal code called a zip code.

- **Single-valued** and **multivalued** attributes. The attributes in our examples all have a single value for a particular entity. For instance, the *student_ID* attribute for a specific student entity refers to only one student *ID*. Such attributes are said to be **single valued**. There may be instances where an attribute has a set of values for a specific entity. Suppose we add to the *instructor* entity set a *phone_number* attribute. An *instructor* may have zero, one, or several phone numbers, and different instructors may have different numbers of phones. This type of attribute is said to be **multivalued**. As another example, we could add to the *instructor* entity set an attribute *dependent_name* listing all the dependents. This attribute would be multivalued, since any particular instructor may have zero, one, or more dependents.
- **Derived attributes**. The value for this type of attribute can be derived from the values of other related attributes or entities. For instance, let us say that the *instructor* entity set has an attribute *students_advised*, which represents how many students an instructor advises. We can derive the value for this attribute by counting the number of *student* entities associated with that instructor.

  As another example, suppose that the *instructor* entity set has an attribute *age* that indicates the instructor's age. If the *instructor* entity set also has an attribute *date_of_birth*, we can calculate *age* from *date_of_birth* and the current date. Thus, *age* is a derived attribute. In this case, *date_of_birth* may be referred to as a *base* attribute, or a *stored* attribute. The value of a derived attribute is not stored but is computed when required.

Figure 6.8 shows how composite attributes can be represented in the E-R notation. Here, a composite attribute *name* with component attributes *first_name*, *middle_initial*, and *last_name* replaces the simple attribute *name* of *instructor*. As another example, suppose we were to add an address to the *instructor* entity set. The address can be defined as the composite attribute *address* with the attributes *street*, *city*, *state*, and *postal_code*. The attribute *street* is itself a composite attribute whose component attributes are *street_number*, *street_name*, and *apartment_number*. The figure also illustrates a multivalued attribute *phone_number*, denoted by "{*phone_number*}", and a derived attribute *age*, depicted by "*age* ( )".

An attribute takes a **null** value when an entity does not have a value for it. The *null* value may indicate "not applicable"—that is, the value does not exist for the entity. For example, a person who has no middle name may have the *middle_initial* attribute set to *null*. *Null* can also designate that an attribute value is unknown. An unknown value may be either *missing* (the value does exist, but we do not have that information) or *not known* (we do not know whether or not the value actually exists).

For instance, if the *name* value for a particular instructor is *null*, we assume that the value is missing, since every instructor must have a name. A null value for the *apartment_number* attribute could mean that the address does not include an apartment number (not applicable), that an apartment number exists but we do not know what

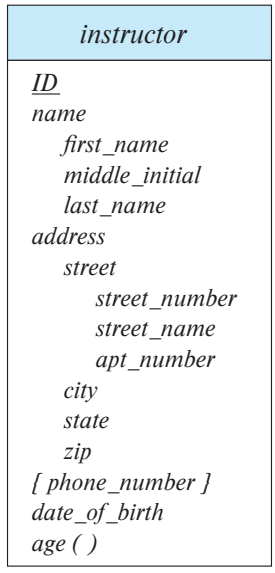


Figure 6.8 E-R diagram with composite, multivalued, and derived attributes.

it is (missing), or that we do not know whether or not an apartment number is part of the instructor's address (unknown).

6.4 Mapping Cardinalities

Mapping cardinalities, or cardinality ratios, express the number of entities to which another entity can be associated via a relationship set. Mapping cardinalities are most useful in describing binary relationship sets, although they can contribute to the description of relationship sets that involve more than two entity sets.

For a binary relationship set R between entity sets A and B, the mapping cardinality must be one of the following:

- **One-to-one**. An entity in A is associated with *at most* one entity in B, and an entity in B is associated with *at most* one entity in A. (See Figure 6.9a.)
- **One-to-many**. An entity in A is associated with any number (zero or more) of entities in B. An entity in B, however, can be associated with *at most* one entity in A. (See Figure 6.9b.)
- **Many-to-one**. An entity in A is associated with *at most* one entity in B. An entity in B, however, can be associated with any number (zero or more) of entities in A. (See Figure 6.10a.)

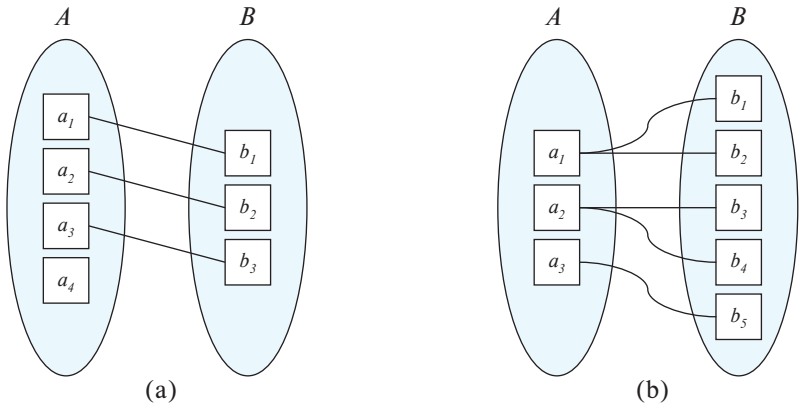


Figure 6.9 Mapping cardinalities. (a) One-to-one. (b) One-to-many.

- **Many-to-many**. An entity in A is associated with any number (zero or more) of entities in B, and an entity in B is associated with any number (zero or more) of entities in A. (See Figure 6.10b.)

The appropriate mapping cardinality for a particular relationship set obviously depends on the real-world situation that the relationship set is modeling.

As an illustration, consider the *advisor* relationship set. If a student can be advised by several instructors (as in the case of students advised jointly), the relationship set is many-to-many. In contrast, if a particular university imposes a constraint that a student can be advised by only one instructor, and an instructor can advise several students, then the relationship set from *instructor* to *student* must be one-to-many. Thus, mapping

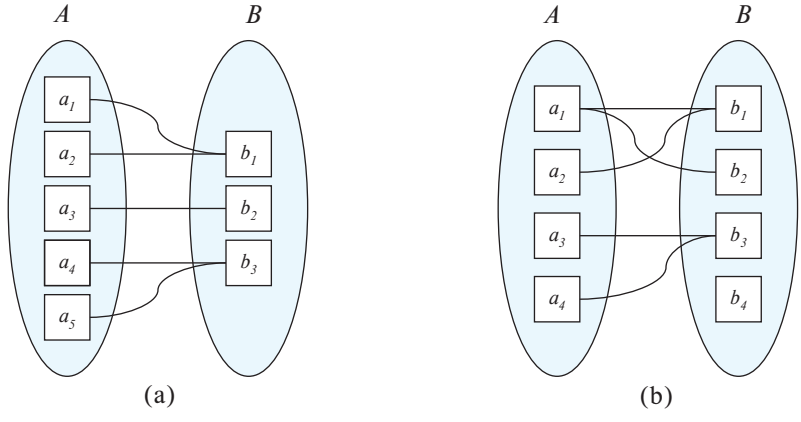


Figure 6.10 Mapping cardinalities. (a) Many-to-one. (b) Many-to-many.

cardinalities can be used to specify constraints on what relationships are permitted in the real world.

In the E-R diagram notation, we indicate cardinality constraints on a relationship by drawing either a directed line (→) or an undirected line (—) between the relationship set and the entity set in question. Specifically, for the university example:

- **One-to-one**. We draw a directed line from the relationship set to both entity sets. For example, in Figure 6.11a, the directed lines to *instructor* and *student* indicate that an instructor may advise at most one student, and a student may have at most one advisor.

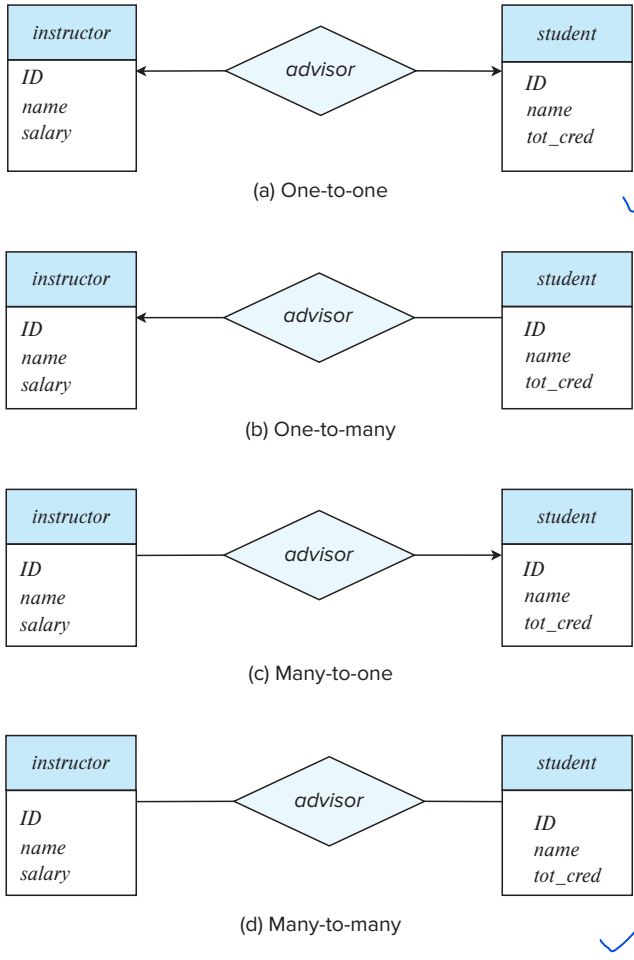


Figure 6.11 Relationship cardinalities.

- **One-to-many**. We draw a directed line from the relationship set to the "one" side of the relationship. Thus, in Figure 6.11b, there is a directed line from relationship set *advisor* to the entity set *instructor*, and an undirected line to the entity set *student*. This indicates that an instructor may advise many students, but a student may have at most one advisor.
- **Many-to-one**. We draw a directed line from the relationship set to the "one" side of the relationship. Thus, in Figure 6.11c, there is an undirected line from the relationship set *advisor* to the entity set *instructor* and a directed line to the entity set *student*. This indicates that an instructor may advise at most one student, but a student may have many advisors.
- **Many-to-many**. We draw an undirected line from the relationship set to both entity sets. Thus, in Figure 6.11d, there are undirected lines from the relationship set *advisor* to both entity sets *instructor* and *student*. This indicates that an instructor may advise many students, and a student may have many advisors.

The participation of an entity set E in a relationship set R is said to be **total** if every entity in E must participate in at least one relationship in R. If it is possible that some entities in E do not participate in relationships in R, the participation of entity set E in relationship R is said to be **partial**.

For example, a university may require every *student* to have at least one advisor; in the E-R model, this corresponds to requiring each entity to be related to at least one instructor through the *advisor* relationship. Therefore, the participation of *student* in the relationship set *advisor* is total. In contrast, an *instructor* need not advise any students. Hence, it is possible that only some of the *instructor* entities are related to the *student* entity set through the *advisor* relationship, and the participation of *instructor* in the *advisor* relationship set is therefore partial.

We indicate total participation of an entity in a relationship set using double lines. Figure 6.12 shows an example of the *advisor* relationship set where the double line indicates that a student must have an advisor.

E-R diagrams also provide a way to indicate more complex constraints on the number of times each entity participates in relationships in a relationship set. A line may have an associated minimum and maximum cardinality, shown in the form $l..h$, where l

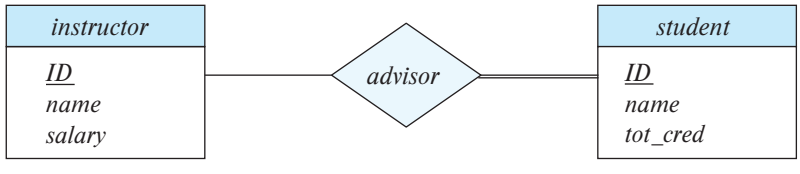


Figure 6.12 E-R diagram showing total participation.

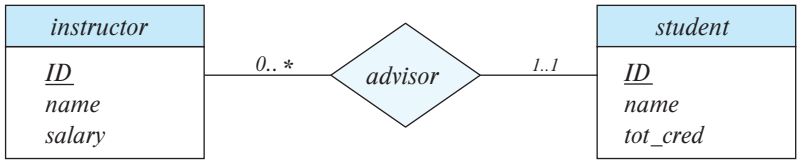


Figure 6.13 Cardinality limits on relationship sets.

is the minimum and h the maximum cardinality. A minimum value of 1 indicates total participation of the entity set in the relationship set; that is, each entity in the entity set occurs in at least one relationship in that relationship set. A maximum value of 1 indicates that the entity participates in at most one relationship, while a maximum value $*$ indicates no limit.

For example, consider Figure 6.13. The line between *advisor* and *student* has a cardinality constraint of 1..1, meaning the minimum and the maximum cardinality are both 1. That is, each student must have exactly one advisor. The limit 0.. $*$ on the line between *advisor* and *instructor* indicates that an instructor can have zero or more students. Thus, the relationship *advisor* is one-to-many from *instructor* to *student*, and further the participation of *student* in *advisor* is total, implying that a student must have an advisor.

It is easy to misinterpret the 0.. $*$ on the left edge and think that the relationship *advisor* is many-to-one from *instructor* to *student*—this is exactly the reverse of the correct interpretation.

If both edges have a maximum value of 1, the relationship is one-to-one. If we had specified a cardinality limit of 1.. $*$ on the left edge, we would be saying that each instructor must advise at least one student.

The E-R diagram in Figure 6.13 could alternatively have been drawn with a double line from *student* to *advisor*, and an arrow on the line from *advisor* to *instructor*, in place of the cardinality constraints shown. This alternative diagram would enforce exactly the same constraints as the constraints shown in the figure.

In the case of nonbinary relationship sets, we can specify some types of many-to-one relationships. Suppose a *student* can have at most one instructor as a guide on a project. This constraint can be specified by an arrow pointing to *instructor* on the edge from *proj_guide*.

We permit at most one arrow out of a nonbinary relationship set, since an E-R diagram with two or more arrows out of a nonbinary relationship set can be interpreted in two ways. We elaborate on this issue in Section 6.5.2.

6.5 Primary Key

We must have a way to specify how entities within a given entity set and relationships within a given relationship set are distinguished.

6.5.1 Entity Sets

Conceptually, individual entities are distinct; from a database perspective, however, the differences among them must be expressed in terms of their attributes.

Therefore, the values of the attribute values of an entity must be such that they can *uniquely identify* the entity. In other words, no two entities in an entity set are allowed to have exactly the same value for all attributes.

The notion of a *key* for a relation schema, as defined in Section 2.3, applies directly to entity sets. That is, a key for an entity is a set of attributes that suffice to distinguish entities from each other. The concepts of superkey, candidate key, and primary key are applicable to entity sets just as they are applicable to relation schemas.

Keys also help to identify relationships uniquely, and thus distinguish relationships from each other. Next, we define the corresponding notions of keys for relationship sets.

6.5.2 Relationship Sets

We need a mechanism to distinguish among the various relationships of a relationship set.

Let R be a relationship set involving entity sets $E_1, E_2, \ldots, E_n$. Let *primary-key*(E_i) denote the set of attributes that forms the primary key for entity set E_i. Assume for now that the attribute names of all primary keys are unique. The composition of the primary key for a relationship set depends on the set of attributes associated with the relationship set R.

If the relationship set R has no attributes associated with it, then the set of attributes

$$\textit{primary-key}(E_1) \cup \textit{primary-key}(E_2) \cup \cdots \cup \textit{primary-key}(E_n)$$

describes an individual relationship in set R.

If the relationship set R has attributes $a_1, a_2, \ldots, a_m$ associated with it, then the set of attributes

$$\textit{primary-key}(E_1) \cup \textit{primary-key}(E_2) \cup \cdots \cup \textit{primary-key}(E_n) \cup \{a_1, a_2, \ldots, a_m\}$$

describes an individual relationship in set R.

If the attribute names of primary keys are not unique across entity sets, the attributes are renamed to distinguish them; the name of the entity set combined with the name of the attribute would form a unique name. If an entity set participates more than once in a relationship set (as in the *prereq* relationship in Section 6.2.2), the role name is used instead of the name of the entity set, to form a unique attribute name.

Recall that a relationship set is a set of relationship instances, and each instance is uniquely identified by the entities that participate in it. Thus, in both of the preceding cases, the set of attributes

$$\textit{primary-key}(E_1) \cup \textit{primary-key}(E_2) \cup \cdots \cup \textit{primary-key}(E_n)$$

forms a superkey for the relationship set.

The choice of the primary key for a binary relationship set depends on the mapping cardinality of the relationship set. For many-to-many relationships, the preceding union of the primary keys is a minimal superkey and is chosen as the primary key. As an illustration, consider the entity sets *instructor* and *student*, and the relationship set *advisor*, in Section 6.2.2. Suppose that the relationship set is many-to-many. Then the primary key of *advisor* consists of the union of the primary keys of *instructor* and *student*.

For one-to-many and many-to-one relationships, the primary key of the "many" side is a minimal superkey and is used as the primary key. For example, if the relationship is many-to-one from *student* to *instructor*—that is, each student can have at most one advisor—then the primary key of *advisor* is simply the primary key of *student*. However, if an instructor can advise only one student—that is, if the *advisor* relationship is many-to-one from *instructor* to *student*—then the primary key of *advisor* is simply the primary key of *instructor*.

For one-to-one relationships, the primary key of either one of the participating entity sets forms a minimal superkey, and either one can be chosen as the primary key of the relationship set. However, if an instructor can advise only one student, and each student can be advised by only one instructor—that is, if the *advisor* relationship is one-to-one—then the primary key of either *student* or *instructor* can be chosen as the primary key for *advisor*.

For nonbinary relationships, if no cardinality constraints are present, then the superkey formed as described earlier in this section is the only candidate key, and it is chosen as the primary key. The choice of the primary key is more complicated if cardinality constraints are present. As we noted in Section 6.4, we permit at most one arrow out of a relationship set. We do so because an E-R diagram with two or more arrows out of a nonbinary relationship set can be interpreted in the two ways we describe below.

Suppose there is a relationship set R between entity sets E_1, E_2, E_3, E_4, and the only arrows are on the edges to entity sets E_3 and E_4. Then, the two possible interpretations are:

1. A particular combination of entities from E_1, E_2 can be associated with at most one combination of entities from E_3, E_4. Thus, the primary key for the relationship R can be constructed by the union of the primary keys of E_1 and E_2.
2. A particular combination of entities from E_1, E_2, E_3 can be associated with at most one combination of entities from E_4, and further a particular combination of entities from E_1, E_2, E_4 can be associated with at most one combination of entities from E_3, Then the union of the primary keys of E_1, E_2, and E_3 forms a candidate key, as does the union of the primary keys of E_1, E_2, and E_4.

Each of these interpretations has been used in practice and both are correct for particular enterprises being modeled. Thus, to avoid confusion, we permit only one arrow out of a nonbinary relationship set, in which case the two interpretations are equivalent.

In order to represent a situation where one of the multiple-arrow situations holds, the E-R design can be modified by replacing the non-binary relationship set with an entity set. That is, we treat each instance of the non-binary relationship set as an entity. Then we can relate each of those entities to corresponding instances of E_1, E_2, E_4 via separate relationship sets. A simpler approach is to use *functional dependencies*, which we study in Chapter 7 (Section 7.4). Functional dependencies which allow either of these interpretations to be specified simply in an unambiguous manner.

The primary key for the relationship set R is then the union of the primary keys of those participating entity sets E_i that do not have an incoming arrow from the relationship set R.

6.5.3 Weak Entity Sets

Consider a *section* entity, which is uniquely identified by a course identifier, semester, year, and section identifier. Section entities are related to course entities. Suppose we create a relationship set *sec_course* between entity sets *section* and *course*.

Now, observe that the information in *sec_course* is redundant, since *section* already has an attribute *course_id*, which identifies the course with which the section is related. One option to deal with this redundancy is to get rid of the relationship *sec_course*; however, by doing so the relationship between *section* and *course* becomes implicit in an attribute, which is not desirable.

An alternative way to deal with this redundancy is to not store the attribute *course_id* in the *section* entity and to only store the remaining attributes *sec_id*, *year*, and *semester*.[2] However, the entity set *section* then does not have enough attributes to identify a particular *section* entity uniquely; although each *section* entity is distinct, sections for different courses may share the same *sec_id*, *year*, and *semester*. To deal with this problem, we treat the relationship *sec_course* as a special relationship that provides extra information, in this case the *course_id*, required to identify *section* entities uniquely.

The notion of *weak entity set* formalizes the above intuition. A **weak entity set** is one whose existence is dependent on another entity set, called its **identifying entity set**; instead of associating a primary key with a weak entity, we use the primary key of the identifying entity, along with extra attributes, called **discriminator attributes** to uniquely identify a weak entity. An entity set that is not a weak entity set is termed a **strong entity set**.

Every weak entity must be associated with an identifying entity; that is, the weak entity set is said to be **existence dependent** on the identifying entity set. The identifying entity set is said to **own** the weak entity set that it identifies. The relationship associating the weak entity set with the identifying entity set is called the **identifying relationship**.

The identifying relationship is many-to-one from the weak entity set to the identifying entity set, and the participation of the weak entity set in the relationship is total.

[2] Note that the relational schema we eventually create from the entity set *section* does have the attribute *course_id*, for reasons that will become clear later, even though we have dropped the attribute *course_id* from the entity set *section*.

The identifying relationship set should not have any descriptive attributes, since any such attributes can instead be associated with the weak entity set.

In our example, the identifying entity set for *section* is *course*, and the relationship *sec_course*, which associates *section* entities with their corresponding *course* entities, is the identifying relationship. The primary key of *section* is formed by the primary key of the identifying entity set (that is, *course*), plus the discriminator of the weak entity set (that is, *section*). Thus, the primary key is {*course_id*, *sec_id*, *year*, *semester*}.

Note that we could have chosen to make *sec_id* globally unique across all courses offered in the university, in which case the *section* entity set would have had a primary key. However, conceptually, a *section* is still dependent on a *course* for its existence, which is made explicit by making it a weak entity set.

In E-R diagrams, a weak entity set is depicted via a double rectangle with the discriminator being underlined with a dashed line. The relationship set connecting the weak entity set to the identifying strong entity set is depicted by a double diamond. In Figure 6.14, the weak entity set *section* depends on the strong entity set *course* via the relationship set *sec_course*.

The figure also illustrates the use of double lines to indicate that the participation of the (weak) entity set *section* in the relationship *sec_course* is *total*, meaning that every section must be related via *sec_course* to some course. Finally, the arrow from *sec_course* to *course* indicates that each section is related to a single course.

In general, a weak entity set must have a total participation in its identifying relationship set, and the relationship is many-to-one toward the identifying entity set.

A weak entity set can participate in relationships other than the identifying relationship. For instance, the *section* entity could participate in a relationship with the *time_slot* entity set, identifying the time when a particular class section meets. A weak entity set may participate as owner in an identifying relationship with another weak entity set. It is also possible to have a weak entity set with more than one identifying entity set. A particular weak entity would then be identified by a combination of entities, one from each identifying entity set. The primary key of the weak entity set would consist of the union of the primary keys of the identifying entity sets, plus the discriminator of the weak entity set.

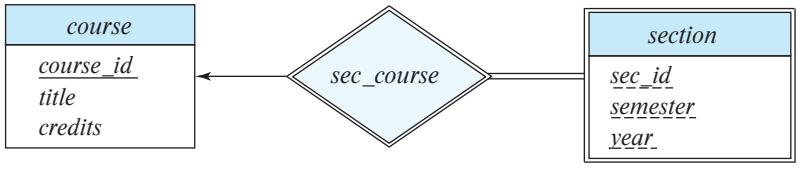


Figure 6.14 E-R diagram with a weak entity set.

6.6 Removing Redundant Attributes in Entity Sets

When we design a database using the E-R model, we usually start by identifying those entity sets that should be included. For example, in the university organization we have discussed thus far, we decided to include such entity sets as *student* and *instructor*. Once the entity sets are decided upon, we must choose the appropriate attributes. These attributes are supposed to represent the various values we want to capture in the database. In the university organization, we decided that for the *instructor* entity set, we will include the attributes *ID*, *name*, *dept_name*, and *salary*. We could have added the attributes *phone_number*, *office_number*, *home_page*, and others. The choice of what attributes to include is up to the designer, who has a good understanding of the structure of the enterprise.

Once the entities and their corresponding attributes are chosen, the relationship sets among the various entities are formed. These relationship sets may result in a situation where attributes in the various entity sets are redundant and need to be removed from the original entity sets. To illustrate, consider the entity sets *instructor* and *department*:

- The entity set *instructor* includes the attributes *ID*, *name*, *dept_name*, and *salary*, with *ID* forming the primary key.
- The entity set *department* includes the attributes *dept_name*, *building*, and *budget*, with *dept_name* forming the primary key.

We model the fact that each instructor has an associated department using a relationship set *inst_dept* relating *instructor* and *department*.

The attribute *dept_name* appears in both entity sets. Since it is the primary key for the entity set *department*, it is redundant in the entity set *instructor* and needs to be removed.

Removing the attribute *dept_name* from the *instructor* entity set may appear rather unintuitive, since the relation *instructor* that we used in the earlier chapters had an attribute *dept_name*. As we shall see later, when we create a relational schema from the E-R diagram, the attribute *dept_name* in fact gets added to the relation *instructor*, but only if each instructor has at most one associated department. If an instructor has more than one associated department, the relationship between instructors and departments is recorded in a separate relation *inst_dept*.

Treating the connection between instructors and departments uniformly as a relationship, rather than as an attribute of *instructor*, makes the logical relationship explicit, and it helps avoid a premature assumption that each instructor is associated with only one department.

Similarly, the *student* entity set is related to the *department* entity set through the relationship set *student_dept* and thus there is no need for a *dept_name* attribute in *student*.

As another example, consider course offerings (sections) along with the time slots of the offerings. Each time slot is identified by a *time_slot_id*, and has associated with it a set of weekly meetings, each identified by a day of the week, start time, and end time. We decide to model the set of weekly meeting times as a multivalued composite attribute. Suppose we model entity sets *section* and *time_slot* as follows:

- The entity set *section* includes the attributes *course_id*, *sec_id*, *semester*, *year*, *building*, *room_number*, and *time_slot_id*, with (*course_id*, *sec_id*, *year*, *semester*) forming the primary key.
- The entity set *time_slot* includes the attributes *time_slot_id*, which is the primary key,[3] and a multivalued composite attribute {(*day*, *start_time*, *end_time*)}.[4]

These entities are related through the relationship set *sec_time_slot*.

The attribute *time_slot_id* appears in both entity sets. Since it is the primary key for the entity set *time_slot*, it is redundant in the entity set *section* and needs to be removed.

As a final example, suppose we have an entity set *classroom*, with attributes *building*, *room_number*, and *capacity*, with *building* and *room_number* forming the primary key. Suppose also that we have a relationship set *sec_class* that relates *section* to *classroom*. Then the attributes {*building*, *room_number*} are redundant in the entity set *section*.

A good entity-relationship design does not contain redundant attributes. For our university example, we list the entity sets and their attributes below, with primary keys underlined:

- *classroom*: with attributes (*building*, *room_number*, *capacity*).
- *department*: with attributes (*dept_name*, *building*, *budget*).
- *course*: with attributes (*course_id*, *title*, *credits*).
- *instructor*: with attributes (*ID*, *name*, *salary*).
- *section*: with attributes (*course_id*, *sec_id*, *semester*, *year*).
- *student*: with attributes (*ID*, *name*, *tot_cred*).
- *time_slot*: with attributes (*time_slot_id*, {(*day*, *start_time*, *end_time*) }).

The relationship sets in our design are listed below:

- *inst_dept*: relating instructors with departments.
- *stud_dept*: relating students with departments.

[3] We shall see later on that the primary key for the relation created from the entity set *time_slot* includes *day* and *start_time*; however, *day* and *start_time* do not form part of the primary key of the entity set *time_slot*.

[4] We could optionally give a name, such as *meeting*, for the composite attribute containing *day*, *start_time*, and *end_time*.

- *teaches*: relating instructors with sections.
- *takes*: relating students with sections, with a descriptive attribute *grade*.
- *course_dept*: relating courses with departments.
- *sec_course*: relating sections with courses.
- *sec_class*: relating sections with classrooms.
- *sec_time_slot*: relating sections with time slots.
- *advisor*: relating students with instructors.
- *prereq*: relating courses with prerequisite courses.

You can verify that none of the entity sets has any attribute that is made redundant by one of the relationship sets. Further, you can verify that all the information (other than constraints) in the relational schema for our university database, which we saw earlier in Figure 2.9, has been captured by the above design, but with several attributes in the relational design replaced by relationships in the E-R design.

We are finally in a position to show (Figure 6.15) the E-R diagram that corresponds to the university enterprise that we have been using thus far in the text. This E-R diagram is equivalent to the textual description of the university E-R model, but with several additional constraints.

In our university database, we have a constraint that each instructor must have exactly one associated department. As a result, there is a double line in Figure 6.15 between *instructor* and *inst_dept*, indicating total participation of *instructor* in *inst_dept*; that is, each instructor must be associated with a department. Further, there is an arrow from *inst_dept* to *department*, indicating that each instructor can have at most one associated department.

Similarly, entity set *course* has a double line to relationship set *course_dept*, indicating that every course must be in some department, and entity set *student* has a double line to relationship set *stud_dept*, indicating that every student must be majoring in some department. In each case, an arrow points to the entity set *department* to show that a course (and, respectively, a student) can be related to only one department, not several.

Similarly, entity set *course* has a double line to relationship set *course_dept*, indicating that every course must be in some department, and entity set *student* has a double line to relationship set *stud_dept*, indicating that every student must be majoring in some department. In each case, an arrow points to the entity set *department* to show that a course (and, respectively, a student) can be related to only one department, not several.

Further, Figure 6.15 shows that the relationship set *takes* has a descriptive attribute *grade*, and that each student has at most one advisor. The figure also shows that *section* is a weak entity set, with attributes *sec_id*, *semester*, and *year* forming the discriminator; *sec_course* is the identifying relationship set relating weak entity set *section* to the strong entity set *course*.

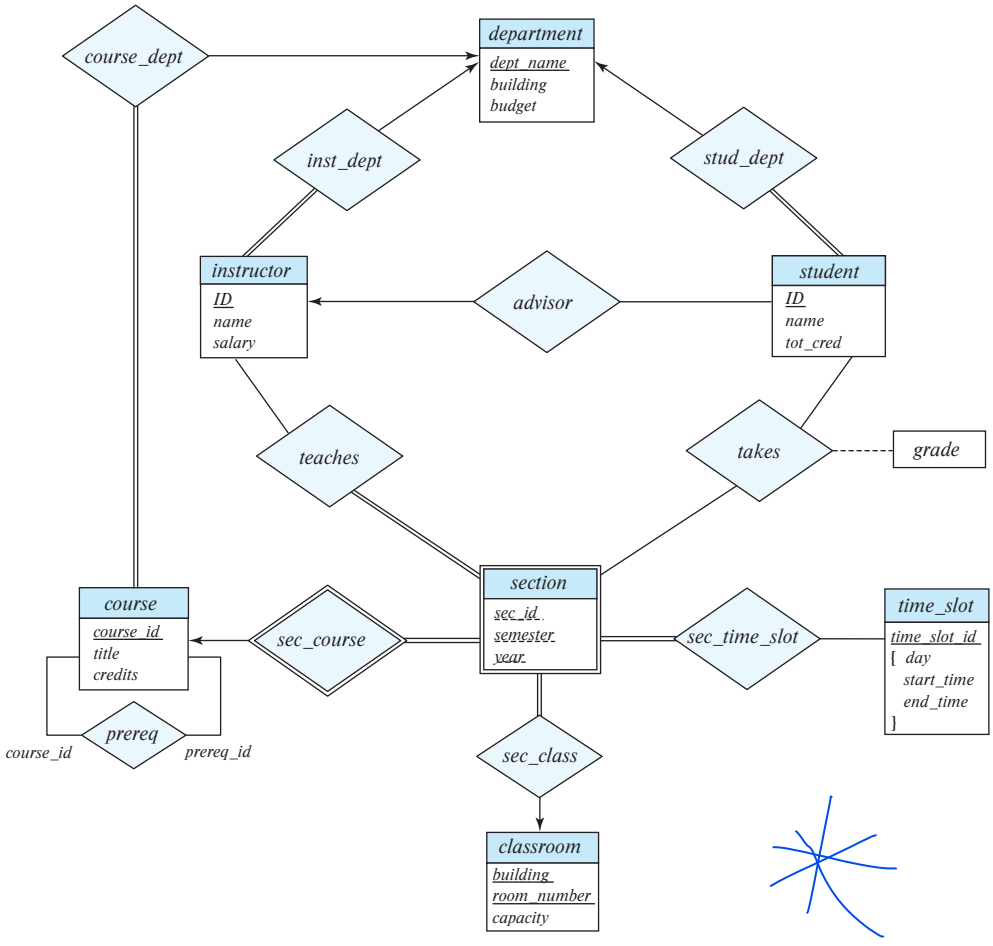


Figure 6.15 E-R diagram for a university enterprise.

In Section 6.7, we show how this E-R diagram can be used to derive the various relation schemas we use.

6.7 Reducing E-R Diagrams to Relational Schemas

Both the E-R model and the relational database model are abstract, logical representations of real-world enterprises. Because the two models employ similar design principles, we can convert an E-R design into a relational design. For each entity set and for each relationship set in the database design, there is a unique relation schema to which we assign the name of the corresponding entity set or relationship set.

In this section, we describe how an E-R schema can be represented by relation schemas and how constraints arising from the E-R design can be mapped to constraints on relation schemas.

6.7.1 Representation of Strong Entity Sets

Let E be a strong entity set with only simple descriptive attributes $a_1, a_2, \ldots, a_n$. We represent this entity with a schema called E with n distinct attributes. Each tuple in a relation on this schema corresponds to one entity of the entity set E.

For schemas derived from strong entity sets, the primary key of the entity set serves as the primary key of the resulting schema. This follows directly from the fact that each tuple corresponds to a specific entity in the entity set.

As an illustration, consider the entity set *student* of the E-R diagram in Figure 6.15. This entity set has three attributes: *ID*, *name*, *tot_cred*. We represent this entity set by a schema called *student* with three attributes:

student (*ID*, *name*, *tot_cred*)

Note that since student *ID* is the primary key of the entity set, it is also the primary key of the relation schema.

Continuing with our example, for the E-R diagram in Figure 6.15, all the strong entity sets, except *time_slot*, have only simple attributes. The schemas derived from these strong entity sets are depicted in Figure 6.16. Note that the *instructor*, *student*, and *course* schemas are different from the schemas we have used in the previous chapters (they do not contain the attribute *dept_name*). We shall revisit this issue shortly.

6.7.2 Representation of Strong Entity Sets with Complex Attributes

When a strong entity set has nonsimple attributes, things are a bit more complex. We handle composite attributes by creating a separate attribute for each of the component attributes; we do not create a separate attribute for the composite attribute itself. To illustrate, consider the version of the *instructor* entity set depicted in Figure 6.8. For the composite attribute *name*, the schema generated for *instructor* contains the attributes

classroom(*building*, *room_number*, *capacity*)
department(*dept_name*, *building*, *budget*)
course(*course_id*, *title*, *credits*)
instructor(*ID*, *name*, *salary*)
student(*ID*, *name*, *tot_cred*)

Figure 6.16 Schemas derived from the entity sets in the E-R diagram in Figure 6.15.

first_name, *middle_initial*, and *last_name*; there is no separate attribute or schema for *name*. Similarly, for the composite attribute *address*, the schema generated contains the attributes *street*, *city*, *state*, and *postal_code*. Since *street* is a composite attribute it is replaced by *street_number*, *street_name*, and *apt_number*.

Multivalued attributes are treated differently from other attributes. We have seen that attributes in an E-R diagram generally map directly into attributes for the appropriate relation schemas. Multivalued attributes, however, are an exception; new relation schemas are created for these attributes, as we shall see shortly.

Derived attributes are not explicitly represented in the relational data model. However, they can be represented as stored procedures, functions, or methods in other data models.

The relational schema derived from the version of entity set *instructor* with complex attributes, without including the multivalued attribute, is thus:

instructor (*ID*, *first_name*, *middle_initial*, *last_name*,
street_number, *street_name*, *apt_number*,
city, *state*, *postal_code*, *date_of_birth*)

For a multivalued attribute M, we create a relation schema R with an attribute A that corresponds to M and attributes corresponding to the primary key of the entity set or relationship set of which M is an attribute.

As an illustration, consider the E-R diagram in Figure 6.8 that depicts the entity set *instructor*, which includes the multivalued attribute *phone_number*. The primary key of *instructor* is *ID*. For this multivalued attribute, we create a relation schema

instructor_phone (*ID*, *phone_number*)

Each phone number of an instructor is represented as a unique tuple in the relation on this schema. Thus, if we had an instructor with *ID* 22222, and phone numbers 555-1234 and 555-4321, the relation *instructor_phone* would have two tuples (22222, 555-1234) and (22222, 555-4321).

We create a primary key of the relation schema consisting of all attributes of the schema. In the above example, the primary key consists of both attributes of the relation schema *instructor_phone*.

In addition, we create a foreign-key constraint on the relation schema created from the multivalued attribute. In that newly created schema, the attribute generated from the primary key of the entity set must reference the relation generated from the entity set. In the above example, the foreign-key constraint on the *instructor_phone* relation would be that attribute *ID* references the *instructor* relation.

In the case that an entity set consists of only two attributes—a single primary-key attribute B and a single multivalued attribute M— the relation schema for the entity set would contain only one attribute, namely, the primary-key attribute B. We can drop

this relation, while retaining the relation schema with the attribute B and attribute A that corresponds to M.

To illustrate, consider the entity set *time_slot* depicted in Figure 6.15. Here, *time_slot_id* is the primary key of the *time_slot* entity set, and there is a single multivalued attribute that happens also to be composite. The entity set can be represented by just the following schema created from the multivalued composite attribute:

$$time\_slot\ (\underline{time\_slot\_id}, \underline{day}, \underline{start\_time}, end\_time)$$

Although not represented as a constraint on the E-R diagram, we know that there cannot be two meetings of a class that start at the same time of the same day of the week but end at different times; based on this constraint, *end_time* has been omitted from the primary key of the *time_slot* schema.

The relation created from the entity set would have only a single attribute *time_slot_id*; the optimization of dropping this relation has the benefit of simplifying the resultant database schema, although it has a drawback related to foreign keys, which we briefly discuss in Section 6.7.4.

6.7.3 Representation of Weak Entity Sets

Let A be a weak entity set with attributes $a_1, a_2, \ldots, a_m$. Let B be the strong entity set on which A depends. Let the primary key of B consist of attributes $b_1, b_2, \ldots, b_n$. We represent the entity set A by a relation schema called A with one attribute for each member of the set:

$$\{a_1, a_2, \ldots, a_m\} \cup \{b_1, b_2, \ldots, b_n\}$$

For schemas derived from a weak entity set, the combination of the primary key of the strong entity set and the discriminator of the weak entity set serves as the primary key of the schema. In addition to creating a primary key, we also create a foreign-key constraint on the relation A, specifying that the attributes $b_1, b_2, \ldots, b_n$ reference the primary key of the relation B. The foreign-key constraint ensures that for each tuple representing a weak entity, there is a corresponding tuple representing the corresponding strong entity.

As an illustration, consider the weak entity set *section* in the E-R diagram of Figure 6.15. This entity set has the attributes: *sec_id*, *semester*, and *year*. The primary key of the *course* entity set, on which *section* depends, is *course_id*. Thus, we represent *section* by a schema with the following attributes:

$$section\ (\underline{course\_id}, \underline{sec\_id}, \underline{semester}, \underline{year})$$

The primary key consists of the primary key of the entity set *course*, along with the discriminator of *section*, which is *sec_id*, *semester*, and *year*. We also create a foreign-key

constraint on the *section* schema, with the attribute *course_id* referencing the primary key of the *course* schema.[5]

6.7.4 Representation of Relationship Sets

Let R be a relationship set, let $a_1, a_2, \ldots, a_m$ be the set of attributes formed by the union of the primary keys of each of the entity sets participating in R, and let the descriptive attributes (if any) of R be $b_1, b_2, \ldots, b_n$. We represent this relationship set by a relation schema called R with one attribute for each member of the set:

$$\{a_1, a_2, \ldots, a_m\} \cup \{b_1, b_2, \ldots, b_n\}$$

We described in Section 6.5, how to choose a primary key for a binary relationship set. The primary key attributes of the relationship set are also used as the primary key attributes of the relational schema R.

As an illustration, consider the relationship set *advisor* in the E-R diagram of Figure 6.15. This relationship set involves the following entity sets:

- *instructor*, with the primary key *ID*.
- *student*, with the primary key *ID*.

Since the relationship set has no attributes, the *advisor* schema has two attributes, the primary keys of *instructor* and *student*. Since both attributes have the same name, we rename them *i_ID* and s_*ID*. Since the *advisor* relationship set is many-to-one from *student* to *instructor* the primary key for the *advisor* relation schema is *s_ID*.

We also create foreign-key constraints on the relation schema R as follows: For each entity set E_i related by relationship set R, we create a foreign-key constraint from relation schema R, with the attributes of R that were derived from primary-key attributes of E_i referencing the primary key of the relation schema representing E_i.

Returning to our earlier example, we thus create two foreign-key constraints on the *advisor* relation, with attribute *i_ID* referencing the primary key of *instructor* and attribute *s_ID* referencing the primary key of *student*.

Applying the preceding techniques to the other relationship sets in the E-R diagram in Figure 6.15, we get the relational schemas depicted in Figure 6.17.

Observe that for the case of the relationship set *prereq*, the role indicators associated with the relationship are used as attribute names, since both roles refer to the same relation *course*.

Similar to the case of *advisor*, the primary key for each of the relations *sec_course*, *sec_time_slot*, *sec_class*, *inst_dept*, *stud_dept*, and *course_dept* consists of the primary key

[5]Optionally, the foreign-key constraint could have an "on delete cascade" specification, so that deletion of a *course* entity automatically deletes any *section* entities that reference the *course* entity. Without that specification, each section of a course would have to be deleted before the corresponding course can be deleted.

teaches (*ID*, *course_id*, *sec_id*, *semester*, *year*)
takes (*ID*, *course_id*, *sec_id*, *semester*, *year*, *grade*)
prereq (*course_id*, *prereq_id*)
advisor (*s_ID*, *i_ID*)
sec_course (*course_id*, *sec_id*, *semester*, *year*)
sec_time_slot (*course_id*, *sec_id*, *semester*, *year*, *time_slot_id*)
sec_class (*course_id*, *sec_id*, *semester*, *year*, *building*, *room_number*)
inst_dept (*ID*, *dept_name*)
stud_dept (*ID*, *dept_name*)
course_dept (*course_id*, *dept_name*)

Figure 6.17 Schemas derived from relationship sets in the E-R diagram in Figure 6.15.

of only one of the two related entity sets, since each of the corresponding relationships is many-to-one.

Foreign keys are not shown in Figure 6.17, but for each of the relations in the figure there are two foreign-key constraints, referencing the two relations created from the two related entity sets. Thus, for example, *sec_course* has foreign keys referencing *section* and *classroom*, *teaches* has foreign keys referencing *instructor* and *section*, and *takes* has foreign keys referencing *student* and *section*.

The optimization that allowed us to create only a single relation schema from the entity set *time_slot*, which had a multivalued attribute, prevents the creation of a foreign key from the relation schema *sec_time_slot* to the relation created from entity set *time_slot*, since we dropped the relation created from the entity set *time_slot*. We retained the relation created from the multivalued attribute and named it *time_slot*, but this relation may potentially have no tuples corresponding to a *time_slot_id*, or it may have multiple tuples corresponding to a *time_slot_id*; thus, *time_slot_id* in *sec_time_slot* cannot reference this relation.

The astute reader may wonder why we have not seen the schemas *sec_course*, *sec_time_slot*, *sec_class*, *inst_dept*, *stud_dept*, and *course_dept* in the previous chapters. The reason is that the algorithm we have presented thus far results in some schemas that can be either eliminated or combined with other schemas. We explore this issue next.

6.7.5 Redundancy of Schemas

A relationship set linking a weak entity set to the corresponding strong entity set is treated specially. As we noted in Section 6.5.3, these relationships are many-to-one and have no descriptive attributes. Furthermore, the of a weak entity set includes the primary key of the strong entity set. In the E-R diagram of Figure 6.14, the weak entity set *section* is dependent on the strong entity set *course* via the relationship set *sec_course*.

The primary key of *section* is {*course_id*, *sec_id*, *semester*, *year*}, and the primary key of *course* is *course_id*. Since *sec_course* has no descriptive attributes, the *sec_course* schema has attributes *course_id*, *sec_id*, *semester*, and *year*. The schema for the entity set *section* includes the attributes *course_id*, *sec_id*, *semester*, and *year* (among others). Every (*course_id*, *sec_id*, *semester*, *year*) combination in a *sec_course* relation would also be present in the relation on schema *section*, and vice versa. Thus, the *sec_course* schema is redundant.

In general, the schema for the relationship set linking a weak entity set to its corresponding strong entity set is redundant and does not need to be present in a relational database design based upon an E-R diagram.

6.7.6 Combination of Schemas

Consider a many-to-one relationship set *AB* from entity set *A* to entity set *B*. Using our relational-schema construction algorithm outlined previously, we get three schemas: *A*, *B*, and *AB*. Suppose further that the participation of *A* in the relationship is total; that is, every entity *a* in the entity set *A* must participate in the relationship *AB*. Then we can combine the schemas *A* and *AB* to form a single schema consisting of the union of attributes of both schemas. The primary key of the combined schema is the primary key of the entity set into whose schema the relationship set schema was merged.

To illustrate, let's examine the various relations in the E-R diagram of Figure 6.15 that satisfy the preceding criteria:

- *inst_dept*. The schemas *instructor* and *department* correspond to the entity sets *A* and *B*, respectively. Thus, the schema *inst_dept* can be combined with the *instructor* schema. The resulting *instructor* schema consists of the attributes {*ID*, *name*, *dept_name*, *salary*}.
- *stud_dept*. The schemas *student* and *department* correspond to the entity sets *A* and *B*, respectively. Thus, the schema *stud_dept* can be combined with the *student* schema. The resulting *student* schema consists of the attributes {*ID*, *name*, *dept_name*, *tot_cred*}.
- *course_dept*. The schemas *course* and *department* correspond to the entity sets *A* and *B*, respectively. Thus, the schema *course_dept* can be combined with the *course* schema. The resulting *course* schema consists of the attributes {*course_id*, *title*, *dept_name*, *credits*}.
- *sec_class*. The schemas *section* and *classroom* correspond to the entity sets *A* and *B*, respectively. Thus, the schema *sec_class* can be combined with the *section* schema. The resulting *section* schema consists of the attributes {*course_id*, *sec_id*, *semester*, *year*, *building*, *room_number*}.
- *sec_time_slot*. The schemas *section* and *time_slot* correspond to the entity sets *A* and *B* respectively, Thus, the schema *sec_time_slot* can be combined with the *section*

schema obtained in the previous step. The resulting *section* schema consists of the attributes {*course_id*, *sec_id*, *semester*, *year*, *building*, *room_number*, *time_slot_id*}.

In the case of one-to-one relationships, the relation schema for the relationship set can be combined with the schemas for either of the entity sets.

We can combine schemas even if the participation is partial by using null values. In the preceding example, if *inst_dept* were partial, then we would store null values for the *dept_name* attribute for those instructors who have no associated department.

Finally, we consider the foreign-key constraints that would have appeared in the schema representing the relationship set. There would have been foreign-key constraints referencing each of the entity sets participating in the relationship set. We drop the constraint referencing the entity set into whose schema the relationship set schema is merged, and add the other foreign-key constraints to the combined schema. For example, *inst_dept* has a foreign key constraint of the attribute *dept_name* referencing the *department* relation. This foreign constraint is enforced implicitly by the *instructor* relation when the schema for *inst_dept* is merged into *instructor*.

6.8 Extended E-R Features

Although the basic E-R concepts can model most database features, some aspects of a database may be more aptly expressed by certain extensions to the basic E-R model. In this section, we discuss the extended E-R features of specialization, generalization, higher- and lower-level entity sets, attribute inheritance, and aggregation.

To help with the discussions, we shall use a slightly more elaborate database schema for the university. In particular, we shall model the various people within a university by defining an entity set *person*, with attributes *ID*, *name*, *street*, and *city*.

6.8.1 Specialization

An entity set may include subgroupings of entities that are distinct in some way from other entities in the set. For instance, a subset of entities within an entity set may have attributes that are not shared by all the entities in the entity set. The E-R model provides a means for representing these distinctive entity groupings.

As an example, the entity set *person* may be further classified as one of the following:

- *employee*.
- *student*.

Each of these person types is described by a set of attributes that includes all the attributes of entity set *person* plus possibly additional attributes. For example, *employee* entities may be described further by the attribute *salary*, whereas *student* entities may

be described further by the attribute *tot_cred*. The process of designating subgroupings within an entity set is called **specialization**. The specialization of *person* allows us to distinguish among person entities according to whether they correspond to employees or students: in general, a person could be an employee, a student, both, or neither.

As another example, suppose the university divides students into two categories: graduate and undergraduate. Graduate students have an office assigned to them. Undergraduate students are assigned to a residential college. Each of these student types is described by a set of attributes that includes all the attributes of the entity set *student* plus additional attributes.

We can apply specialization repeatedly to refine a design. The university could create two specializations of *student*, namely *graduate* and *undergraduate*. As we saw earlier, student entities are described by the attributes *ID*, *name*, *street, city*, and *tot_cred*. The entity set *graduate* would have all the attributes of *student* and an additional attribute *office_number*. The entity set *undergraduate* would have all the attributes of *student*, and an additional attribute *residential_college*. As another example, university employees may be further classified as one of *instructor* or *secretary*.

Each of these employee types is described by a set of attributes that includes all the attributes of entity set *employee* plus additional attributes. For example, *instructor* entities may be described further by the attribute *rank* while *secretary* entities are described by the attribute *hours_per_week*. Further, *secretary* entities may participate in a relationship *secretary_for* between the *secretary* and *employee* entity sets, which identifies the employees who are assisted by a secretary.

An entity set may be specialized by more than one distinguishing feature. In our example, the distinguishing feature among employee entities is the job the employee performs. Another, coexistent, specialization could be based on whether the person is a temporary (limited_term) employee or a permanent employee, resulting in the entity sets *temporary_employee* and *permanent_employee*. When more than one specialization is formed on an entity set, a particular entity may belong to multiple specializations. For instance, a given employee may be a temporary employee who is a secretary.

In terms of an E-R diagram, specialization is depicted by a hollow arrow-head pointing from the specialized entity to the other entity (see Figure 6.18). We refer to this relationship as the ISA relationship, which stands for "is a" and represents, for example, that an instructor "is a" employee.

The way we depict specialization in an E-R diagram depends on whether an entity may belong to multiple specialized entity sets or if it must belong to at most one specialized entity set. The former case (multiple sets permitted) is called **overlapping specialization**, while the latter case (at most one permitted) is called **disjoint specialization**. For an overlapping specialization (as is the case for *student* and *employee* as specializations of *person*), two separate arrows are used. For a disjoint specialization (as is the case for *instructor* and *secretary* as specializations of *employee*), a single arrow is used. The specialization relationship may also be referred to as a **superclass-subclass** relationship. Higher- and lower-level entity sets are depicted as regular entity sets—that is, as rectangles containing the name of the entity set.

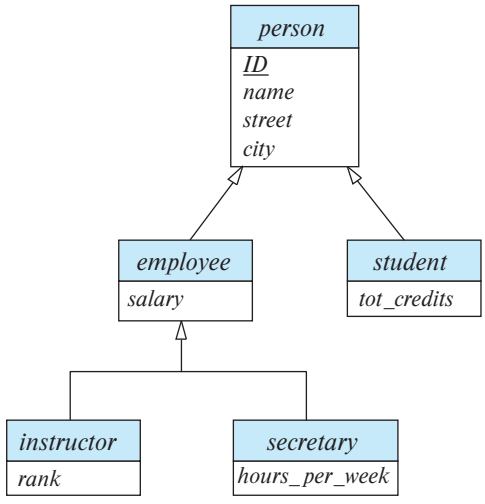


Figure 6.18 Specialization and generalization.

6.8.2 Generalization

The refinement from an initial entity set into successive levels of entity subgroupings represents a **top-down** design process in which distinctions are made explicit. The design process may also proceed in a **bottom-up** manner, in which multiple entity sets are synthesized into a higher-level entity set on the basis of common features. The database designer may have first identified:

- *instructor* entity set with attributes *instructor_id*, *instructor_name*, *instructor_salary*, and *rank*.
- *secretary* entity set with attributes *secretary_id*, *secretary_name*, *secretary_salary*, and *hours_per_week*.

There are similarities between the *instructor* entity set and the *secretary* entity set in the sense that they have several attributes that are conceptually the same across the two entity sets: namely, the identifier, name, and salary attributes. This commonality can be expressed by **generalization**, which is a containment relationship that exists between a *higher-level* entity set and one or more *lower-level* entity sets. In our example, *employee* is the higher-level entity set and *instructor* and *secretary* are lower-level entity sets. In this case, attributes that are conceptually the same had different names in the two lower-level entity sets. To create a generalization, the attributes must be given a common name and represented with the higher-level entity *person*. We can use the attribute names *ID*, *name*, *street*, and *city*, as we saw in the example in Section 6.8.1.

Higher- and lower-level entity sets also may be designated by the terms **superclass** and **subclass**, respectively. The *person* entity set is the superclass of the *employee* and *student* subclasses.

For all practical purposes, generalization is a simple inversion of specialization. We apply both processes, in combination, in the course of designing the E-R schema for an enterprise. In terms of the E-R diagram itself, we do not distinguish between specialization and generalization. New levels of entity representation are distinguished (specialization) or synthesized (generalization) as the design schema comes to express fully the database application and the user requirements of the database. Differences in the two approaches may be characterized by their starting point and overall goal.

Specialization stems from a single entity set; it emphasizes differences among entities within the set by creating distinct lower-level entity sets. These lower-level entity sets may have attributes, or may participate in relationships, that do not apply to all the entities in the higher-level entity set. Indeed, the reason a designer applies specialization is to represent such distinctive features. If *student* and *employee* have exactly the same attributes as *person* entities, and participate in exactly the same relationships as *person* entities, there would be no need to specialize the *person* entity set.

Generalization proceeds from the recognition that a number of entity sets share some common features (namely, they are described by the same attributes and participate in the same relationship sets). On the basis of their commonalities, generalization synthesizes these entity sets into a single, higher-level entity set. Generalization is used to emphasize the similarities among lower-level entity sets and to hide the differences; it also permits an economy of representation in that shared attributes are not repeated.

6.8.3 Attribute Inheritance

A crucial property of the higher- and lower-level entities created by specialization and generalization is **attribute inheritance**. The attributes of the higher-level entity sets are said to be **inherited** by the lower-level entity sets. For example, *student* and *employee* inherit the attributes of *person*. Thus, *student* is described by its *ID*, *name*, *street*, and *city* attributes, and additionally a *tot_cred* attribute; *employee* is described by its *ID*, *name*, *street*, and *city* attributes, and additionally a *salary* attribute. Attribute inheritance applies through all tiers of lower-level entity sets; thus, *instructor* and *secretary*, which are subclasses of *employee*, inherit the attributes *ID*, *name*, *street,* and *city* from *person*, in addition to inheriting *salary* from *employee*.

A lower-level entity set (or subclass) also inherits participation in the relationship sets in which its higher-level entity (or superclass) participates. Like attribute inheritance, participation inheritance applies through all tiers of lower-level entity sets. For example, suppose the *person* entity set participates in a relationship *person_dept* with *department*. Then, the *student*, *employee*, *instructor* and *secretary* entity sets, which are subclasses of the *person* entity set, also implicitly participate in the *person_dept* relationship with *department*. These entity sets can participate in any relationships in which the *person* entity set participates.

Whether a given portion of an E-R model was arrived at by specialization or generalization, the outcome is basically the same:

- A higher-level entity set with attributes and relationships that apply to all of its lower-level entity sets.
- Lower-level entity sets with distinctive features that apply only within a particular lower-level entity set.

In what follows, although we often refer to only generalization, the properties that we discuss belong fully to both processes.

Figure 6.18 depicts a **hierarchy** of entity sets. In the figure, *employee* is a lower-level entity set of *person* and a higher-level entity set of the *instructor* and *secretary* entity sets. In a hierarchy, a given entity set may be involved as a lower-level entity set in only one ISA relationship; that is, entity sets in this diagram have only **single inheritance**. If an entity set is a lower-level entity set in more than one ISA relationship, then the entity set has **multiple inheritance**, and the resulting structure is said to be a *lattice*.

6.8.4 Constraints on Specializations

To model an enterprise more accurately, the database designer may choose to place certain constraints on a particular generalization/specialization.

One type of constraint on specialization which we saw earlier specifies whether a specialization is disjoint or overlapping. Another type of constraint on a specialization/generalization is a **completeness constraint**, which specifies whether or not an entity in the higher-level entity set must belong to at least one of the lower-level entity sets within the generalization/specialization. This constraint may be one of the following:

- **Total specialization** or **generalization**. Each higher-level entity must belong to a lower-level entity set.
- **Partial specialization** or **generalization**. Some higher-level entities may not belong to any lower-level entity set.

Partial specialization is the default. We can specify total specialization in an E-R diagram by adding the keyword "total" in the diagram and drawing a dashed line from the keyword to the corresponding hollow arrowhead to which it applies (for a total specialization), or to the set of hollow arrowheads to which it applies (for an overlapping specialization).

The specialization of *person* to *student* or *employee* is total if the university does not need to represent any person who is neither a *student* nor an *employee*. However, if the university needs to represent such persons, then the specialization would be partial.

The completeness and disjointness constraints, do not depend on each other. Thus, specializations may be partial-overlapping, partial-disjoint, total-overlapping, and total-disjoint.

We can see that certain insertion and deletion requirements follow from the constraints that apply to a given generalization or specialization. For instance, when a total completeness constraint is in place, an entity inserted into a higher-level entity set must also be inserted into at least one of the lower-level entity sets. An entity that is deleted from a higher-level entity set must also be deleted from all the associated lower-level entity sets to which it belongs.

6.8.5 Aggregation

One limitation of the E-R model is that it cannot express relationships among relationships. To illustrate the need for such a construct, consider the ternary relationship *proj_guide*, which we saw earlier, between an *instructor*, *student* and *project* (see Figure 6.6).

Now suppose that each instructor guiding a student on a project is required to file a monthly evaluation report. We model the evaluation report as an entity *evaluation*, with a primary key *evaluation_id*. One alternative for recording the (*student*, *project*, *instructor*) combination to which an *evaluation* corresponds is to create a quaternary (4-way) relationship set *eval_for* between *instructor*, *student*, *project*, and *evaluation*. (A quaternary relationship is required—a binary relationship between *student* and *evaluation*, for example, would not permit us to represent the (*project*, *instructor*) combination to which an *evaluation* corresponds.) Using the basic E-R modeling constructs, we obtain the E-R diagram of Figure 6.19. (We have omitted the attributes of the entity sets, for simplicity.)

It appears that the relationship sets *proj_guide* and *eval_for* can be combined into one single relationship set. Nevertheless, we should not combine them into a single

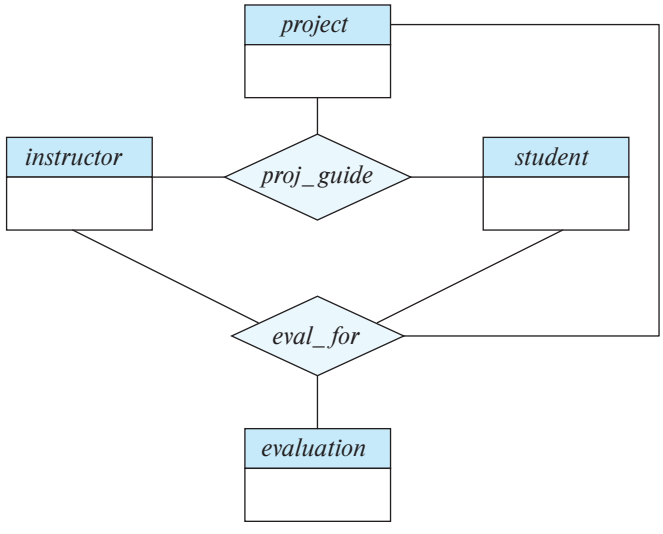


Figure 6.19 E-R diagram with redundant relationships.

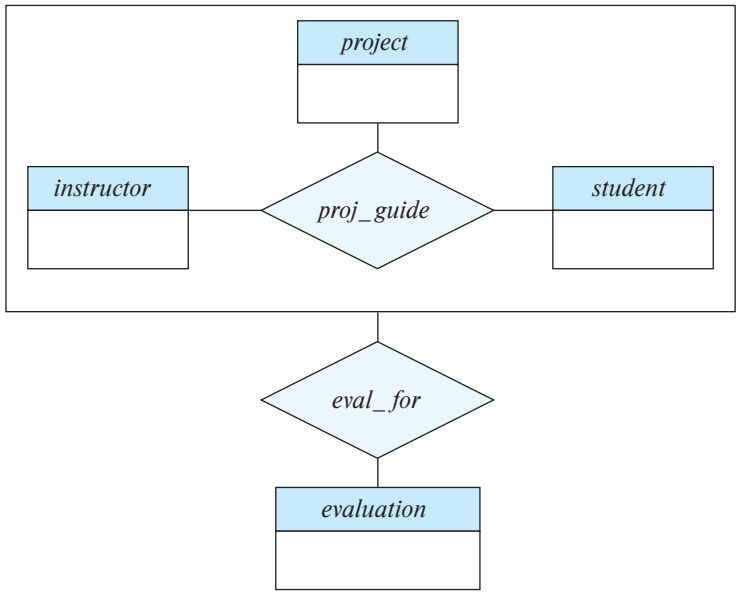


Figure 6.20 E-R diagram with aggregation.

relationship, since some *instructor*, *student*, *project* combinations may not have an associated *evaluation*.

There is redundant information in the resultant figure, however, since every *instructor*, *student*, *project* combination in *eval_for* must also be in *proj_guide*. If *evaluation* was modeled as a value rather than an entity, we could instead make *evaluation* a multivalued composite attribute of the relationship set *proj_guide*. However, this alternative may not be an option if an *evaluation* may also be related to other entities; for example, each evaluation report may be associated with a *secretary* who is responsible for further processing of the evaluation report to make scholarship payments.

The best way to model a situation such as the one just described is to use aggregation. **Aggregation** is an abstraction through which relationships are treated as higher-level entities. Thus, for our example, we regard the relationship set *proj_guide* (relating the entity sets *instructor*, *student*, and *project*) as a higher-level entity set called *proj_guide*. Such an entity set is treated in the same manner as is any other entity set. We can then create a binary relationship *eval_for* between *proj_guide* and *evaluation* to represent which (*student*, *project*, *instructor*) combination an *evaluation* is for. Figure 6.20 shows a notation for aggregation commonly used to represent this situation.

6.8.6 Reduction to Relation Schemas

We are in a position now to describe how the extended E-R features can be translated into relation schemas.

6.8.6.1 Representation of Generalization

There are two different methods of designing relation schemas for an E-R diagram that includes generalization. Although we refer to the generalization in Figure 6.18 in this discussion, we simplify it by including only the first tier of lower-level entity sets—that is, *employee* and *student*. We assume that *ID* is the primary key of *person*.

1. Create a schema for the higher-level entity set. For each lower-level entity set, create a schema that includes an attribute for each of the attributes of that entity set plus one for each attribute of the primary key of the higher-level entity set. Thus, for the E-R diagram of Figure 6.18 (ignoring the *instructor* and *secretary* entity sets) we have three schemas:

   person (*ID*, *name, street, city*)
   employee (*ID*, *salary*)
   student (*ID*, *tot_cred*)

   The primary-key attributes of the higher-level entity set become primary-key attributes of the higher-level entity set as well as all lower-level entity sets. These can be seen underlined in the preceding example.

   In addition, we create foreign-key constraints on the lower-level entity sets, with their primary-key attributes referencing the primary key of the relation created from the higher-level entity set. In the preceding example, the *ID* attribute of *employee* would reference the primary key of *person*, and similarly for *student*.

2. An alternative representation is possible, if the generalization is disjoint and complete—that is, if no entity is a member of two lower-level entity sets directly below a higher-level entity set, and if every entity in the higher-level entity set is also a member of one of the lower-level entity sets. Here, we do not create a schema for the higher-level entity set. Instead, for each lower-level entity set, we create a schema that includes an attribute for each of the attributes of that entity set plus one for *each* attribute of the higher-level entity set. Then, for the E-R diagram of Figure 6.18, we have two schemas:

   employee (*ID*, *name*, *street*, *city*, *salary*)
   student (*ID*, *name*, *street*, *city*, *tot_cred*)

   Both these schemas have *ID*, which is the primary-key attribute of the higher-level entity set *person*, as their primary key.

One drawback of the second method lies in defining foreign-key constraints. To illustrate the problem, suppose we have a relationship set *R* involving entity set *person*. With the first method, when we create a relation schema *R* from the relationship set, we also define a foreign-key constraint on *R*, referencing the schema *person*. Unfortunately, with the second method, we do not have a single relation to which a foreign-key

constraint on R can refer. To avoid this problem, we need to create a relation schema *person* containing at least the primary-key attributes of the *person* entity.

If the second method were used for an overlapping generalization, some values would be stored multiple times, unnecessarily. For instance, if a person is both an employee and a student, values for *street* and *city* would be stored twice.

If the generalization were disjoint but not complete—that is, if some person is neither an employee nor a student—then an extra schema

person (*ID*, *name, street, city*)

would be required to represent such people. However, the problem with foreign-key constraints mentioned above would remain. As an attempt to work around the problem, suppose employees and students are additionally represented in the *person* relation. Unfortunately, name, street, and city information would then be stored redundantly in the *person* relation and the *student* relation for students, and similarly in the *person* relation and the *employee* relation for employees. That suggests storing name, street, and city information only in the *person* relation and removing that information from *student* and *employee*. If we do that, the result is exactly the first method we presented.

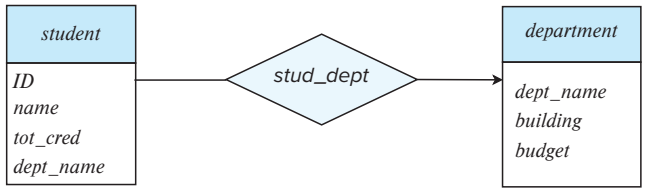
6.8.6.2 Representation of Aggregation

Designing schemas for an E-R diagram containing aggregation is straightforward. Consider Figure 6.20. The schema for the relationship set *eval_for* between the aggregation of *proj_guide* and the entity set *evaluation* includes an attribute for each attribute in the primary keys of the entity set *evaluation* and the relationship set *proj_guide*. It also includes an attribute for any descriptive attributes, if they exist, of the relationship set *eval_for*. We then transform the relationship sets and entity sets within the aggregated entity set following the rules we have already defined.

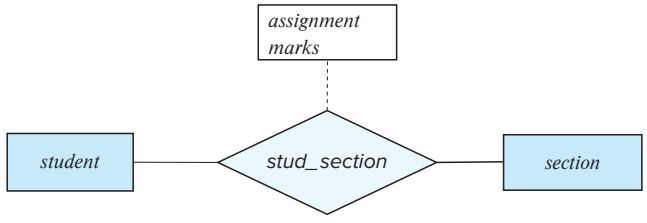
The rules we saw earlier for creating primary-key and foreign-key constraints on relationship sets can be applied to relationship sets involving aggregations as well, with the aggregation treated like any other entity set. The primary key of the aggregation is the primary key of its defining relationship set. No separate relation is required to represent the aggregation; the relation created from the defining relationship is used instead.

6.9 Entity-Relationship Design Issues

The notions of an entity set and a relationship set are not precise, and it is possible to define a set of entities and the relationships among them in a number of different ways. In this section, we examine basic issues in the design of an E-R database schema. Section 6.11 covers the design process in further detail.



(a) Incorrect use of attribute



(b) Erroneous use of relationship attributes

Figure 6.21 Example of erroneous E-R diagrams

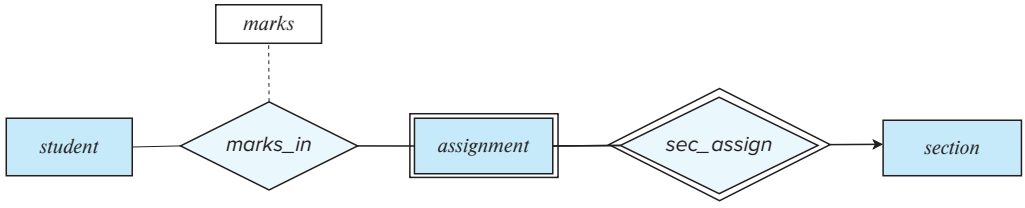
6.9.1 Common Mistakes in E-R Diagrams

A common mistake when creating E-R models is the use of the primary key of an entity set as an attribute of another entity set, instead of using a relationship. For example, in our university E-R model, it is incorrect to have *dept_name* as an attribute of *student*, as depicted in Figure 6.21a, even though it is present as an attribute in the relation schema for *student*. The relationship *stud_dept* is the correct way to represent this information in the E-R model, since it makes the relationship between *student* and *department* explicit, rather than implicit via an attribute. Having an attribute *dept_name* as well as a relationship *stud_dept* would result in duplication of information.

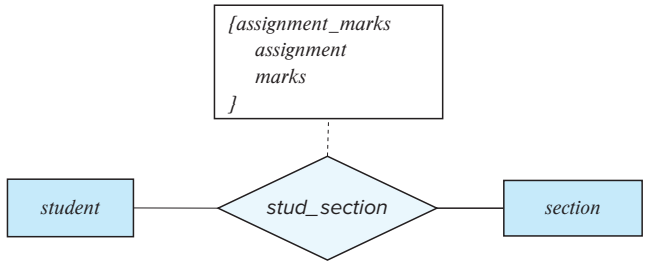
Another related mistake that people sometimes make is to designate the primary-key attributes of the related entity sets as attributes of the relationship set. For example, *ID* (the primary-key attributes of *student*) and *ID* (the primary key of *instructor*) should not appear as attributes of the relationship *advisor*. This should not be done since the primary-key attributes are already implicit in the relationship set.[6]

A third common mistake is to use a relationship with a single-valued attribute in a situation that requires a multivalued attribute. For example, suppose we decided to represent the marks that a student gets in different assignments of a course offering (*section*). A wrong way of doing this would be to add two attributes *assignment* and *marks* to the relationship *takes*, as depicted in Figure 6.21b. The problem with this design is that we can only represent a single assignment for a given student-section pair,

[6]When we create a relation schema from the E-R schema, the attributes may appear in a schema created from the *advisor* relationship set, as we shall see later; however, they should not appear in the *advisor* relationship set.



(c) Correct alternative to erroneous E-R diagram (b)



(d) Correct alternative to erroneous E-R diagram (b)

Figure 6.22 Correct versions of the E-R diagram of Figure 6.21.

since relationship instances must be uniquely identified by the participating entities, *student* and *section*.

One solution to the problem depicted in Figure 6.21c, shown in Figure 6.22a, is to model *assignment* as a weak entity identified by *section*, and to add a relationship *marks_in* between *assignment* and *student*; the relationship would have an attribute *marks*. An alternative solution, shown in Figure 6.22d, is to use a multivalued composite attribute {*assignment_marks*} to *takes*, where *assignment_marks* has component attributes *assignment* and *marks*. Modeling an assignment as a weak entity is preferable in this case, since it allows recording other information about the assignment, such as maximum marks or deadlines.

When an E-R diagram becomes too big to draw in a single piece, it makes sense to break it up into pieces, each showing part of the E-R model. When doing so, you may need to depict an entity set in more than one page. As discussed in Section 6.2.2, attributes of the entity set should be shown only once, in its first occurrence. Subsequent occurrences of the entity set should be shown without any attributes, to avoid repeating the same information at multiple places, which may lead to inconsistency.

6.9.2 Use of Entity Sets versus Attributes

Consider the entity set *instructor* with the additional attribute *phone_number* (Figure 6.23a.) It can be argued that a phone is an entity in its own right with attributes *phone*

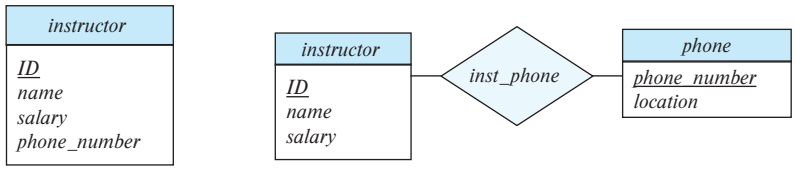


Figure 6.23 Alternatives for adding *phone* to the *instructor* entity set.

_number and *location*; the location may be the office or home where the phone is located, with mobile (cell) phones perhaps represented by the value "mobile." If we take this point of view, we do not add the attribute *phone_number* to the *instructor*. Rather, we create:

- A *phone* entity set with attributes *phone_number* and *location*.
- A relationship set *inst_phone*, denoting the association between instructors and the phones that they have.

This alternative is shown in Figure 6.23b.

What, then, is the main difference between these two definitions of an instructor? Treating a phone as an attribute *phone_number* implies that instructors have precisely one phone number each. Treating a phone as an entity *phone* permits instructors to have several phone numbers (including zero) associated with them. However, we could instead easily define *phone_number* as a multivalued attribute to allow multiple phones per instructor.

The main difference then is that treating a phone as an entity better models a situation where one may want to keep extra information about a phone, such as its location, or its type (mobile, IP phone, or plain old phone), or all who share the phone. Thus, treating phone as an entity is more general than treating it as an attribute and is appropriate when the generality may be useful.

In contrast, it would not be appropriate to treat the attribute *name* (of an instructor) as an entity; it is difficult to argue that *name* is an entity in its own right (in contrast to the phone). Thus, it is appropriate to have *name* as an attribute of the *instructor* entity set.

Two natural questions thus arise: What constitutes an attribute, and what constitutes an entity set? Unfortunately, there are no simple answers. The distinctions mainly depend on the structure of the real-world enterprise being modeled and on the semantics associated with the attribute in question.

6.9.3 Use of Entity Sets versus Relationship Sets

It is not always clear whether an object is best expressed by an entity set or a relationship set. In Figure 6.15, we used the *takes* relationship set to model the situation where a

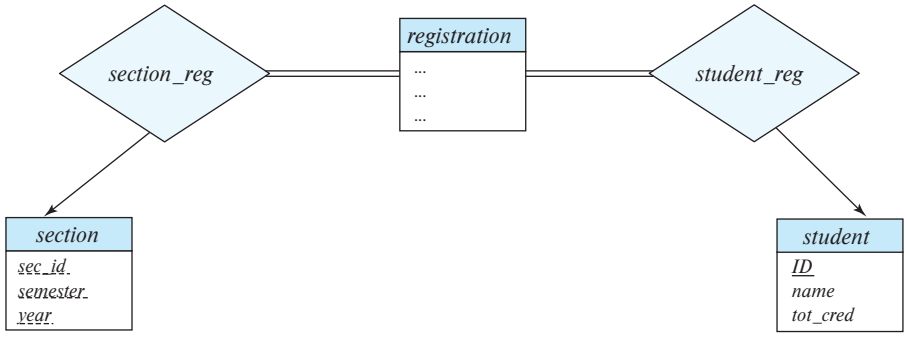


Figure 6.24 Replacement of *takes* by *registration* and two relationship sets.

student takes a (section of a) course. An alternative is to imagine that there is a course-registration record for each course that each student takes. Then, we have an entity set to represent the course-registration record. Let us call that entity set *registration*. Each *registration* entity is related to exactly one student and to exactly one section, so we have two relationship sets, one to relate course-registration records to students and one to relate course-registration records to sections. In Figure 6.24, we show the entity sets *section* and *student* from Figure 6.15 with the *takes* relationship set replaced by one entity set and two relationship sets:

- *registration*, the entity set representing course-registration records.
- *section_reg*, the relationship set relating *registration* and *course*.
- *student_reg*, the relationship set relating *registration* and *student*.

Note that we use double lines to indicate total participation by *registration* entities.

Both the approach of Figure 6.15 and that of Figure 6.24 accurately represent the university's information, but the use of *takes* is more compact and probably preferable. However, if the registrar's office associates other information with a course-registration record, it might be best to make it an entity in its own right.

One possible guideline in determining whether to use an entity set or a relationship set is to designate a relationship set to describe an action that occurs between entities. This approach can also be useful in deciding whether certain attributes may be more appropriately expressed as relationships.

6.9.4 Binary versus *n*-ary Relationship Sets

Relationships in databases are often binary. Some relationships that appear to be nonbinary could actually be better represented by several binary relationships. For instance, one could create a ternary relationship *parent*, relating a child to his/her mother and father. However, such a relationship could also be represented by two binary relationships, *mother* and *father*, relating a child to his/her mother and father separately. Using

the two relationships *mother* and *father* provides us with a record of a child's mother, even if we are not aware of the father's identity; a null value would be required if the ternary relationship *parent* were used. Using binary relationship sets is preferable in this case.

In fact, it is always possible to replace a nonbinary (n-ary, for $n > 2$) relationship set by a number of distinct binary relationship sets. For simplicity, consider the abstract ternary ($n = 3$) relationship set R, relating entity sets A, B, and C. We replace the relationship set R with an entity set E, and we create three relationship sets as shown in Figure 6.25:

- R_A, a many-to-one relationship set from E to A.
- R_B, a many-to-one relationship set from E to B.
- R_C, a many-to-one relationship set from E to C.

E is required to have total participation in each of R_A, R_B, and R_C. If the relationship set R had any attributes, these are assigned to entity set E; further, a special identifying attribute is created for E (since it must be possible to distinguish different entities in an entity set on the basis of their attribute values). For each relationship (a_i, b_i, c_i) in the relationship set R, we create a new entity e_i in the entity set E. Then, in each of the three new relationship sets, we insert a relationship as follows:

- (e_i, a_i) in R_A.
- (e_i, b_i) in R_B.
- (e_i, c_i) in R_C.

We can generalize this process in a straightforward manner to n-ary relationship sets. Thus, conceptually, we can restrict the E-R model to include only binary relationship sets. However, this restriction is not always desirable.

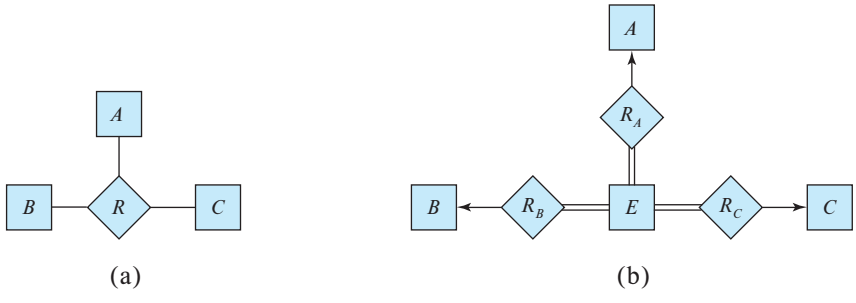


Figure 6.25 Ternary relationship versus three binary relationships.

- An identifying attribute may have to be created for the entity set created to represent the relationship set. This attribute, along with the extra relationship sets required, increases the complexity of the design and (as we shall see in Section 6.7) overall storage requirements.
- An *n*-ary relationship set shows more clearly that several entities participate in a single relationship.
- There may not be a way to translate constraints on the ternary relationship into constraints on the binary relationships. For example, consider a constraint that says that R is many-to-one from A, B to C; that is, each pair of entities from A and B is associated with at most one C entity. This constraint cannot be expressed by using cardinality constraints on the relationship sets R_A, R_B, and R_C.

Consider the relationship set *proj_guide* in Section 6.2.2, relating *instructor*, *student*, and *project*. We cannot directly split *proj_guide* into binary relationships between *instructor* and *project* and between *instructor* and *student*. If we did so, we would be able to record that instructor Katz works on projects A and B with students Shankar and Zhang; however, we would not be able to record that Katz works on project A with student Shankar and works on project B with student Zhang, but does not work on project A with Zhang or on project B with Shankar.

The relationship set *proj_guide* can be split into binary relationships by creating a new entity set as described above. However, doing so would not be very natural.

6.10 Alternative Notations for Modeling Data

A diagrammatic representation of the data model of an application is a very important part of designing a database schema. Creation of a database schema requires not only data modeling experts, but also domain experts who know the requirements of the application but may not be familiar with data modeling. An intuitive diagrammatic representation is particularly important since it eases communication of information between these groups of experts.

A number of alternative notations for modeling data have been proposed, of which E-R diagrams and UML class diagrams are the most widely used. There is no universal standard for E-R diagram notation, and different books and E-R diagram software use different notations.

In the rest of this section, we study some of the alternative E-R diagram notations, as well as the UML class diagram notation. To aid in comparison of our notation with these alternatives, Figure 6.26 summarizes the set of symbols we have used in our E-R diagram notation.

6.10.1 Alternative E-R Notations

Figure 6.27 indicates some of the alternative E-R notations that are widely used. One alternative representation of attributes of entities is to show them in ovals connected

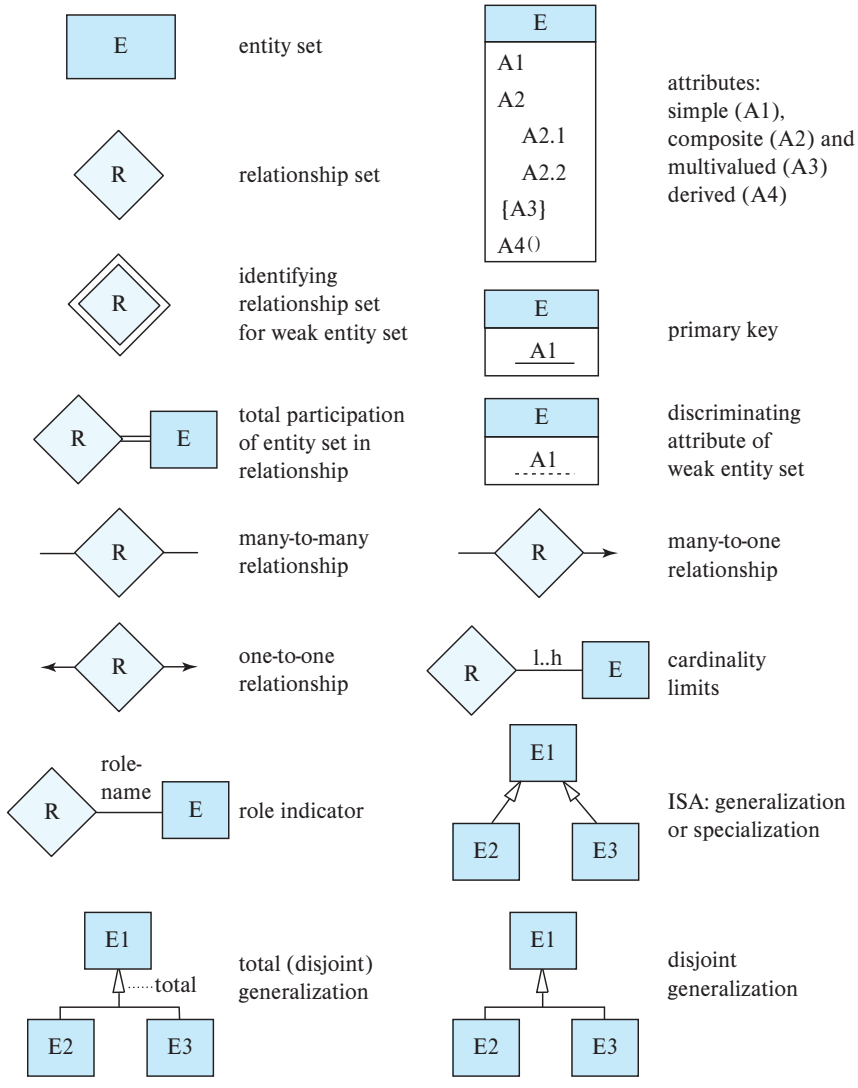


Figure 6.26 Symbols used in the E-R notation.

to the box representing the entity; primary key attributes are indicated by underlining them. The above notation is shown at the top of the figure. Relationship attributes can be similarly represented, by connecting the ovals to the diamond representing the relationship.

Cardinality constraints on relationships can be indicated in several different ways, as shown in Figure 6.27. In one alternative, shown on the left side of the figure, labels $*$ and 1 on the edges out of the relationship are used for depicting many-to-many, one-

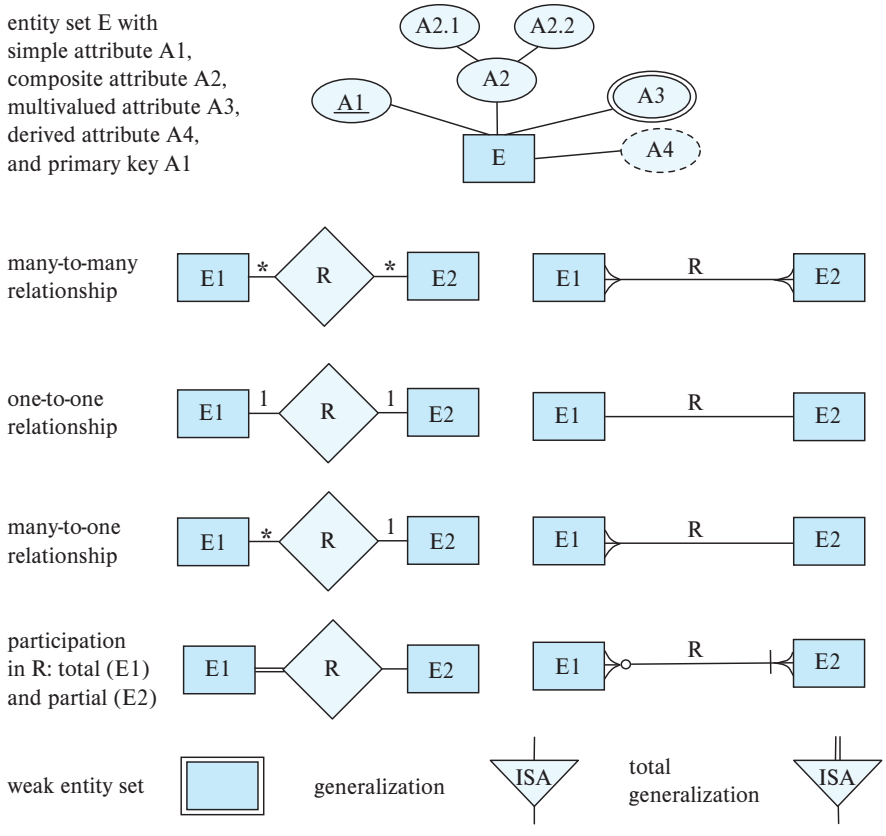


Figure 6.27 Alternative E-R notations.

to-one, and many-to-one relationships. The case of one-to-many is symmetric to many-to-one and is not shown.

In another alternative notation shown on the right side of Figure 6.27, relationship sets are represented by lines between entity sets, without diamonds; only binary relationships can be modeled thus. Cardinality constraints in such a notation are shown by "crow's-foot" notation, as in the figure. In a relationship R between $E1$ and $E2$, crow's feet on both sides indicate a many-to-many relationship, while crow's feet on just the $E1$ side indicate a many-to-one relationship from $E1$ to $E2$. Total participation is specified in this notation by a vertical bar. Note however, that in a relationship R between entities $E1$ and $E2$, if the participation of $E1$ in R is total, the vertical bar is placed on the opposite side, adjacent to entity $E2$. Similarly, partial participation is indicated by using a circle, again on the opposite side.

The bottom part of Figure 6.27 shows an alternative representation of generalization, using triangles instead of hollow arrowheads.

In prior editions of this text up to the fifth edition, we used ovals to represent attributes, with triangles representing generalization, as shown in Figure 6.27. The notation using ovals for attributes and diamonds for relationships is close to the original form of E-R diagrams used by Chen in his paper that introduced the notion of E-R modeling. That notation is now referred to as Chen's notation.

The U.S. National Institute for Standards and Technology defined a standard called IDEF1X in 1993. IDEF1X uses the crow's-foot notation, with vertical bars on the relationship edge to denote total participation and hollow circles to denote partial participation, and it includes other notations that we have not shown.

With the growth in the use of Unified Markup Language (UML), described in Section 6.10.2, we have chosen to update our E-R notation to make it closer to the form of UML class diagrams; the connections will become clear in Section 6.10.2. In comparison with our previous notation, our new notation provides a more compact representation of attributes, and it is also closer to the notation supported by many E-R modeling tools, in addition to being closer to the UML class diagram notation.

There are a variety of tools for constructing E-R diagrams, each of which has its own notational variants. Some of the tools even provide a choice between several E-R notation variants. See the tools section at the end of the chapter for references.

One key difference between entity sets in an E-R diagram and the relation schemas created from such entities is that attributes in the relational schema corresponding to E-R relationships, such as the *dept_name* attribute of *instructor*, are not shown in the entity set in the E-R diagram. Some data modeling tools allow designers to choose between two views of the same entity, one an entity view without such attributes, and other a relational view with such attributes.

6.10.2 The Unified Modeling Language UML

Entity-relationship diagrams help model the data representation component of a software system. Data representation, however, forms only one part of an overall system design. Other components include models of user interactions with the system, specification of functional modules of the system and their interaction, etc. The **Unified Modeling Language** (**UML**) is a standard developed under the auspices of the **Object Management Group** (**OMG**) for creating specifications of various components of a software system. Some of the parts of UML are:

- **Class diagram**. A class diagram is similar to an E-R diagram. Later in this section we illustrate a few features of class diagrams and how they relate to E-R diagrams.
- **Use case diagram**. Use case diagrams show the interaction between users and the system, in particular the steps of tasks that users perform (such as withdrawing money or registering for a course).
- **Activity diagram**. Activity diagrams depict the flow of tasks between various components of a system.

- **Implementation diagram**. Implementation diagrams show the system components and their interconnections, both at the software component level and the hardware component level.

We do not attempt to provide detailed coverage of the different parts of UML here. Instead we illustrate some features of that part of UML that relates to data modeling through examples. See the Further Reading section at the end of the chapter for references on UML.

Figure 6.28 shows several E-R diagram constructs and their equivalent UML class diagram constructs. We describe these constructs below. UML actually models objects, whereas E-R models entities. Objects are like entities, and have attributes, but additionally provide a set of functions (called methods) that can be invoked to compute values on the basis of attributes of the objects, or to update the object itself. Class diagrams can depict methods in addition to attributes. We cover objects in Section 8.2. UML does not support composite or multivalued attributes, and derived attributes are equivalent to methods that take no parameters. Since classes support encapsulation, UML allows attributes and methods to be prefixed with a "+", "-", or "#", which denote respectively public, private, and protected access. Private attributes can only be used in methods of the class, while protected attributes can be used only in methods of the class and its subclasses; these should be familiar to anyone who knows Java, C++, or C#.

In UML terminology, relationship sets are referred to as **associations**; we shall refer to them as relationship sets for consistency with E-R terminology. We represent binary relationship sets in UML by just drawing a line connecting the entity sets. We write the relationship set name adjacent to the line. We may also specify the role played by an entity set in a relationship set by writing the role name on the line, adjacent to the entity set. Alternatively, we may write the relationship set name in a box, along with attributes of the relationship set, and connect the box by a dotted line to the line depicting the relationship set. This box can then be treated as an entity set, in the same way as an aggregation in E-R diagrams, and can participate in relationships with other entity sets.

Since UML version 1.3, UML supports nonbinary relationships, using the same diamond notation used in E-R diagrams. Nonbinary relationships could not be directly represented in earlier versions of UML—they had to be converted to binary relationships by the technique we have seen earlier in Section 6.9.4. UML allows the diamond notation to be used even for binary relationships, but most designers use the line notation.

Cardinality constraints are specified in UML in the same way as in E-R diagrams, in the form $l..h$, where l denotes the minimum and h the maximum number of relationships an entity can participate in. However, you should be aware that the positioning of the constraints is exactly the reverse of the positioning of constraints in E-R diagrams, as shown in Figure 6.28. The constraint $0..*$ on the $E2$ side and $0..1$ on the $E1$ side means that each $E2$ entity can participate in at most one relationship, whereas each $E1$ entity can participate in many relationships; in other words, the relationship is many-to-one from $E2$ to $E1$.

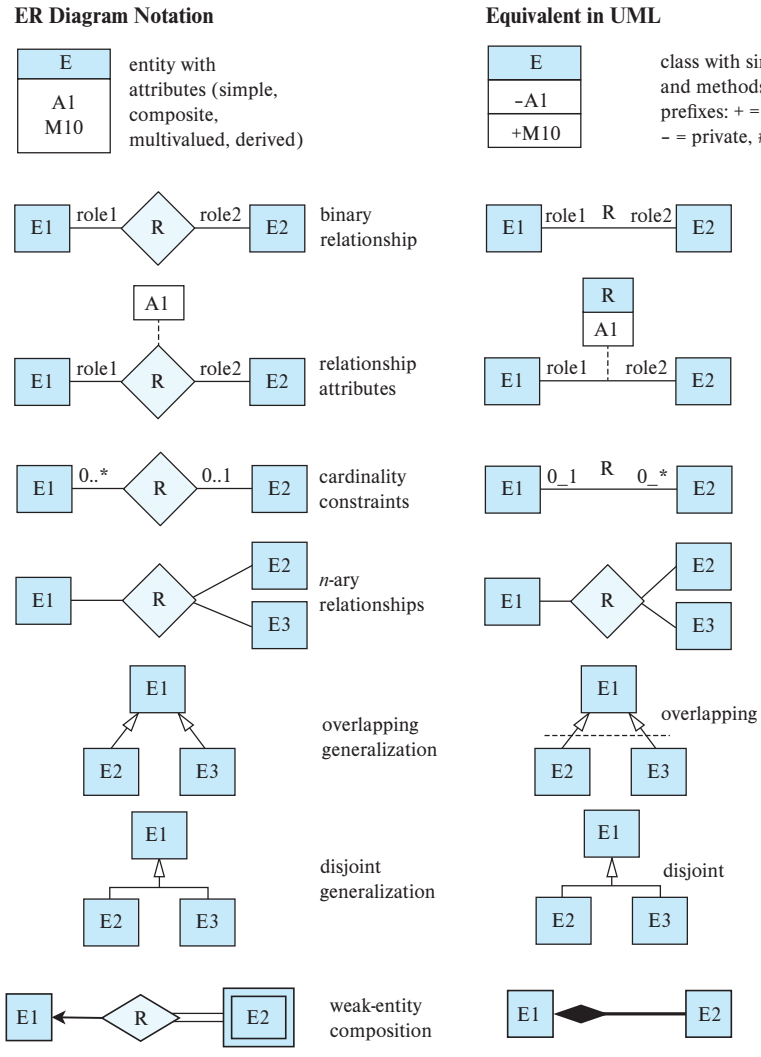


Figure 6.28 Symbols used in the UML class diagram notation.

Single values such as 1 or $*$ may be written on edges; the single value 1 on an edge is treated as equivalent to 1..1, while $*$ is equivalent to 0.. $*$. UML supports generalization; the notation is basically the same as in our E-R notation, including the representation of disjoint and overlapping generalizations.

UML class diagrams include several other notations that approximately correspond to the E-R notations we have seen. A line between two entity sets with a small shaded diamond at one end in UML specifies "composition" in UML. The composition relationship between $E2$ and $E1$ in Figure 6.28 indicates that $E2$ is existence dependent on $E1$; this is roughly equivalent to denoting $E2$ as a weak entity set that is existence

dependent on the identifying entity set $E1$. (The term *aggregation* in UML denotes a variant of composition where $E2$ is contained in $E1$ but may exist independently, and it is denoted using a small hollow diamond.)

UML class diagrams also provide notations to represent object-oriented language features such as interfaces. See the Further Reading section for more information on UML class diagrams.

6.11 Other Aspects of Database Design

Our extensive discussion of schema design in this chapter may create the false impression that schema design is the only component of a database design. There are indeed several other considerations that we address more fully in subsequent chapters, and survey briefly here.

6.11.1 Functional Requirements

All enterprises have rules on what kinds of functionality are to be supported by an enterprise application. These could include transactions that update the data, as well as queries to view data in a desired fashion. In addition to planning the functionality, designers have to plan the interfaces to be built to support the functionality.

Not all users are authorized to view all data, or to perform all transactions. An authorization mechanism is very important for any enterprise application. Such authorization could be at the level of the database, using database authorization features. But it could also be at the level of higher-level functionality or interfaces, specifying who can use which functions/interfaces.

6.11.2 Data Flow, Workflow

Database applications are often part of a larger enterprise application that interacts not only with the database system but also with various specialized applications. As an example, consider a travel-expense report. It is created by an employee returning from a business trip (possibly by means of a special software package) and is subsequently routed to the employee's manager, perhaps other higher-level managers, and eventually to the accounting department for payment (at which point it interacts with the enterprise's accounting information systems).

The term *workflow* refers to the combination of data and tasks involved in processes like those of the preceding examples. Workflows interact with the database system as they move among users and users perform their tasks on the workflow. In addition to the data on which workflows operate, the database may store data about the workflow itself, including the tasks making up a workflow and how they are to be routed among users. Workflows thus specify a series of queries and updates to the database that may be taken into account as part of the database-design process. Put in other terms, modeling the

enterprise requires us not only to understand the semantics of the data but also the business processes that use those data.

6.11.3 Schema Evolution

Database design is usually not a one-time activity. The needs of an organization evolve continually, and the data that it needs to store also evolve correspondingly. During the initial database-design phases, or during the development of an application, the database designer may realize that changes are required at the conceptual, logical, or physical schema levels. Changes in the schema can affect all aspects of the database application. A good database design anticipates future needs of an organization and ensures that the schema requires minimal changes as the needs evolve.

It is important to distinguish between fundamental constraints that are expected to be permanent and constraints that are anticipated to change. For example, the constraint that an instructor-id identify a unique instructor is fundamental. On the other hand, a university may have a policy that an instructor can have only one department, which may change at a later date if joint appointments are allowed. A database design that only allows one department per instructor might require major changes if joint appointments are allowed. Such joint appointments can be represented by adding an extra relationship without modifying the *instructor* relation, as long as each instructor has only one primary department affiliation; a policy change that allows more than one primary affiliation may require a larger change in the database design. A good design should account not only for current policies, but should also avoid or minimize the need for modifications due to changes that are anticipated or have a reasonable chance of happening.

Finally, it is worth noting that database design is a human-oriented activity in two senses: the end users of the system are people (even if an application sits between the database and the end users); and the database designer needs to interact extensively with experts in the application domain to understand the data requirements of the application. All of the people involved with the data have needs and preferences that should be taken into account in order for a database design and deployment to succeed within the enterprise.

6.12 Summary

- Database design mainly involves the design of the database schema. The entity-relationship (E-R) data model is a widely used data model for database design. It provides a convenient graphical representation to view data, relationships, and constraints.
- The E-R model is intended primarily for the database-design process. It was developed to facilitate database design by allowing the specification of an enterprise schema. Such a schema represents the overall logical structure of the database. This overall structure can be expressed graphically by an E-R diagram.

- An entity is an object that exists in the real world and is distinguishable from other objects. We express the distinction by associating with each entity a set of attributes that describes the object.
- A relationship is an association among several entities. A relationship set is a collection of relationships of the same type, and an entity set is a collection of entities of the same type.
- The terms superkey, candidate key, and primary key apply to entity and relationship sets as they do for relation schemas. Identifying the primary key of a relationship set requires some care, since it is composed of attributes from one or more of the related entity sets.
- Mapping cardinalities express the number of entities to which another entity can be associated via a relationship set.
- An entity set that does not have sufficient attributes to form a primary key is termed a weak entity set. An entity set that has a primary key is termed a strong entity set.
- The various features of the E-R model offer the database designer numerous choices in how to best represent the enterprise being modeled. Concepts and objects may, in certain cases, be represented by entities, relationships, or attributes. Aspects of the overall structure of the enterprise may be best described by using weak entity sets, generalization, specialization, or aggregation. Often, the designer must weigh the merits of a simple, compact model versus those of a more precise, but more complex one.
- A database design specified by an E-R diagram can be represented by a collection of relation schemas. For each entity set and for each relationship set in the database, there is a unique relation schema that is assigned the name of the corresponding entity set or relationship set. This forms the basis for deriving a relational database design from an E-R diagram.
- Specialization and generalization define a containment relationship between a higher-level entity set and one or more lower-level entity sets. Specialization is the result of taking a subset of a higher-level entity set to form a lower-level entity set. Generalization is the result of taking the union of two or more disjoint (lower-level) entity sets to produce a higher-level entity set. The attributes of higher-level entity sets are inherited by lower-level entity sets.
- Aggregation is an abstraction in which relationship sets (along with their associated entity sets) are treated as higher-level entity sets, and can participate in relationships.
- Care must be taken in E-R design. There are a number of common mistakes to avoid. Also, there are choices among the use of entity sets, relationship sets, and

attributes in representing aspects of the enterprise whose correctness may depend on subtle details specific to the enterprise.

- UML is a popular modeling language. UML class diagrams are widely used for modeling classes, as well as for general-purpose data modeling.

Review Terms

- Design Process
  - Conceptual-design
  - Logical-design
  - Physical-design
- Entity-relationship (E-R) data model
- Entity and entity set
  - Simple and composite attributes
  - Single-valued and multivalued attributes
  - Derived attribute
- Key
  - Superkey
  - Candidate key
  - Primary key
- Relationship and relationship set
  - Binary relationship set
  - Degree of relationship set
  - Descriptive attributes
  - Superkey, candidate key, and primary key
  - Role
  - Recursive relationship set
- E-R diagram
- Mapping cardinality:
  - One-to-one relationship
  - One-to-many relationship
  - Many-to-one relationship
  - Many-to-many relationship
- Total and partial participation
- Weak entity sets and strong entity sets
  - Discriminator attributes
  - Identifying relationship
- Specialization and generalization
- Aggregation
- Design choices
- United Modeling Language (UML)

Practice Exercises

6.1 Construct an E-R diagram for a car insurance company whose customers own one or more cars each. Each car has associated with it zero to any number of recorded accidents. Each insurance policy covers one or more cars and has one or more premium payments associated with it. Each payment is for a particular period of time, and has an associated due date, and the date when the payment was received.

6.2 Consider a database that includes the entity sets *student*, *course*, and *section* from the university schema and that additionally records the marks that students receive in different exams of different sections.

a. Construct an E-R diagram that models exams as entities and uses a ternary relationship as part of the design.

b. Construct an alternative E-R diagram that uses only a binary relationship between *student* and *section*. Make sure that only one relationship exists between a particular *student* and *section* pair, yet you can represent the marks that a student gets in different exams.

6.3 Design an E-R diagram for keeping track of the scoring statistics of your favorite sports team. You should store the matches played, the scores in each match, the players in each match, and individual player scoring statistics for each match. Summary statistics should be modeled as derived attributes with an explanation as to how they are computed.

6.4 Consider an E-R diagram in which the same entity set appears several times, with its attributes repeated in more than one occurrence. Why is allowing this redundancy a bad practice that one should avoid?

6.5 An E-R diagram can be viewed as a graph. What do the following mean in terms of the structure of an enterprise schema?

a. The graph is disconnected.

b. The graph has a cycle.

6.6 Consider the representation of the ternary relationship of Figure 6.29a using the binary relationships illustrated in Figure 6.29b (attributes not shown).

a. Show a simple instance of E, A, B, C, R_A, R_B, and R_C that cannot correspond to any instance of A, B, C, and R.

b. Modify the E-R diagram of Figure 6.29b to introduce constraints that will guarantee that any instance of E, A, B, C, R_A, R_B, and R_C that satisfies the constraints will correspond to an instance of A, B, C, and R.

c. Modify the preceding translation to handle total participation constraints on the ternary relationship.

6.7 A weak entity set can always be made into a strong entity set by adding to its attributes the primary-key attributes of its identifying entity set. Outline what sort of redundancy will result if we do so.

6.8 Consider a relation such as *sec_course*, generated from a many-to-one relationship set *sec_course*. Do the primary and foreign key constraints created on the relation enforce the many-to-one cardinality constraint? Explain why.

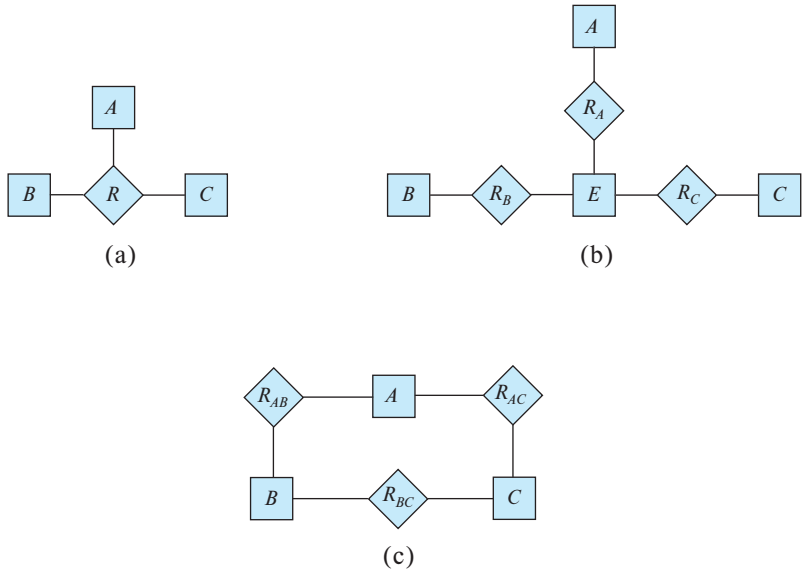


Figure 6.29 Representation of a ternary relationship using binary relationships.

6.9 Suppose the *advisor* relationship set were one-to-one. What extra constraints are required on the relation *advisor* to ensure that the one-to-one cardinality constraint is enforced?

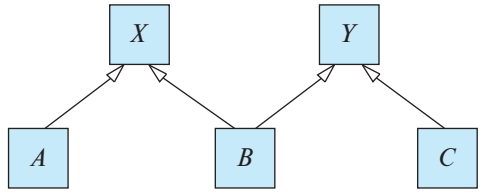
6.10 Consider a many-to-one relationship R between entity sets A and B. Suppose the relation created from R is combined with the relation created from A. In SQL, attributes participating in a foreign key constraint can be null. Explain how a constraint on total participation of A in R can be enforced using **not null** constraints in SQL.

6.11 In SQL, foreign key constraints can reference only the primary key attributes of the referenced relation or other attributes declared to be a superkey using the **unique** constraint. As a result, total participation constraints on a many-to-many relationship set (or on the "one" side of a one-to-many relationship set) cannot be enforced on the relations created from the relationship set, using primary key, foreign key, and not null constraints on the relations.

a. Explain why.

b. Explain how to enforce total participation constraints using complex check constraints or assertions (see Section 4.4.8). (Unfortunately, these features are not supported on any widely used database currently.)

6.12 Consider the following lattice structure of generalization and specialization (attributes not shown).



For entity sets A, B, and C, explain how attributes are inherited from the higher-level entity sets X and Y. Discuss how to handle a case where an attribute of X has the same name as some attribute of Y.

6.13 An E-R diagram usually models the state of an enterprise at a point in time. Suppose we wish to track *temporal changes*, that is, changes to data over time. For example, Zhang may have been a student between September 2015 and May 2019, while Shankar may have had instructor Einstein as advisor from May 2018 to December 2018, and again from June 2019 to January 2020. Similarly, attribute values of an entity or relationship, such as *title* and *credits* of *course*, *salary*, or even *name* of *instructor*, and *tot_cred* of *student*, can change over time.

One way to model temporal changes is as follows: We define a new data type called **valid_time**, which is a time interval, or a set of time intervals. We then associate a *valid_time* attribute with each entity and relationship, recording the time periods during which the entity or relationship is valid. The end time of an interval can be infinity; for example, if Shankar became a student in September 2018, and is still a student, we can represent the end time of the *valid_time* interval as infinity for the Shankar entity. Similarly, we model attributes that can change over time as a set of values, each with its own *valid_time*.

a. Draw an E-R diagram with the *student* and *instructor* entities, and the *advisor* relationship, with the above extensions to track temporal changes.

b. Convert the E-R diagram discussed above into a set of relations.

It should be clear that the set of relations generated is rather complex, leading to difficulties in tasks such as writing queries in SQL. An alternative approach, which is used more widely, is to ignore temporal changes when designing the E-R model (in particular, temporal changes to attribute values), and to modify the relations generated from the E-R model to track temporal changes.

Exercises

6.14 Explain the distinctions among the terms *primary key*, *candidate key*, and *superkey*.

6.15 Construct an E-R diagram for a hospital with a set of patients and a set of medical doctors. Associate with each patient a log of the various tests and examinations conducted.

6.16 Extend the E-R diagram of Exercise 6.3 to track the same information for all teams in a league.

6.17 Explain the difference between a weak and a strong entity set.

6.18 Consider two entity sets A and B that both have the attribute X (among others whose names are not relevant to this question).

a. If the two Xs are completely unrelated, how should the design be improved?

b. If the two Xs represent the same property and it is one that applies both to A and to B, how should the design be improved? Consider three subcases:

- X is the primary key for A but not B
- X is the primary key for both A and B
- X is not the primary key for A nor for B

6.19 We can convert any weak entity set to a strong entity set by simply adding appropriate attributes. Why, then, do we have weak entity sets?

6.20 Construct appropriate relation schemas for each of the E-R diagrams in:

a. Exercise 6.1.

b. Exercise 6.2.

c. Exercise 6.3.

d. Exercise 6.15.

6.21 Consider the E-R diagram in Figure 6.30, which models an online bookstore.

a. Suppose the bookstore adds Blu-ray discs and downloadable video to its collection. The same item may be present in one or both formats, with differing prices. Draw the part of the E-R diagram that models this addition, showing just the parts related to video.

b. Now extend the full E-R diagram to model the case where a shopping basket may contain any combination of books, Blu-ray discs, or downloadable video.

6.22 Design a database for an automobile company to provide to its dealers to assist them in maintaining customer records and dealer inventory and to assist sales staff in ordering cars.

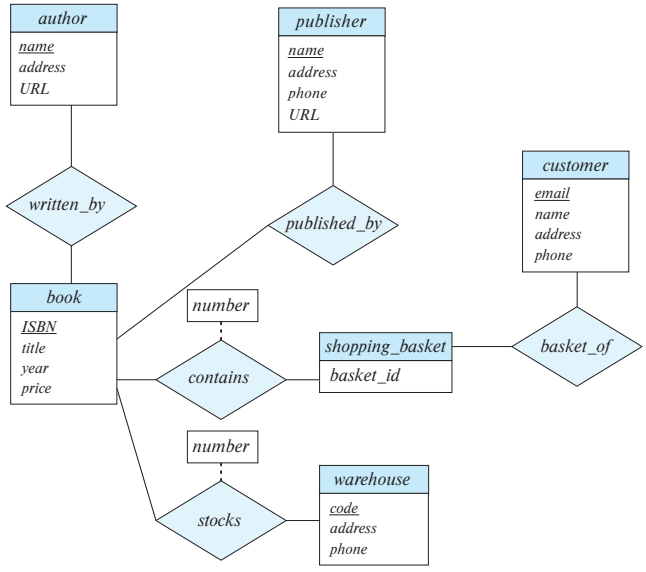


Figure 6.30 E-R diagram for modeling an online bookstore.

Each vehicle is identified by a vehicle identification number (VIN). Each individual vehicle is a particular model of a particular brand offered by the company (e.g., the XF is a model of the car brand Jaguar of Tata Motors). Each model can be offered with a variety of options, but an individual car may have only some (or none) of the available options. The database needs to store information about models, brands, and options, as well as information about individual dealers, customers, and cars.

Your design should include an E-R diagram, a set of relational schemas, and a list of constraints, including primary-key and foreign-key constraints.

6.23 Design a database for a worldwide package delivery company (e.g., DHL or FedEx). The database must be able to keep track of customers who ship items and customers who receive items; some customers may do both. Each package must be identifiable and trackable, so the database must be able to store the location of the package and its history of locations. Locations include trucks, planes, airports, and warehouses.

Your design should include an E-R diagram, a set of relational schemas, and a list of constraints, including primary-key and foreign-key constraints.

6.24 Design a database for an airline. The database must keep track of customers and their reservations, flights and their status, seat assignments on individual flights, and the schedule and routing of future flights.

Your design should include an E-R diagram, a set of relational schemas, and a list of constraints, including primary-key and foreign-key constraints.

6.25 In Section 6.9.4, we represented a ternary relationship (repeated in Figure 6.29a) using binary relationships, as shown in Figure 6.29b. Consider the alternative shown in Figure 6.29c. Discuss the relative merits of these two alternative representations of a ternary relationship by binary relationships.

6.26 Design a generalization – specialization hierarchy for a motor vehicle sales company. The company sells motorcycles, passenger cars, vans, and buses. Justify your placement of attributes at each level of the hierarchy. Explain why they should not be placed at a higher or lower level.

6.27 Explain the distinction between disjoint and overlapping constraints.

6.28 Explain the distinction between total and partial constraints.

Tools

Many database systems provide tools for database design that support E-R diagrams. These tools help a designer create E-R diagrams, and they can automatically create corresponding tables in a database. See bibliographical notes of Chapter 1 for references to database-system vendors' web sites.

There are also several database-independent data modeling tools that support E-R diagrams and UML class diagrams.

Dia, which is a free diagram editor that runs on multiple platforms such as Linux and Windows, supports E-R diagrams and UML class diagrams. To represent entities with attributes, you can use either classes from the UML library or tables from the Database library provided by Dia, since the default E-R notation in Dia represents attributes as ovals. The free online diagram editor *LucidChart* allows you to create E-R diagrams with entities represented in the same ways as we do. To create relationships, we suggest you use diamonds from the Flowchart shape collection. *Draw.io* is another online diagram editor that supports E-R diagrams.

Commercial tools include IBM Rational Rose Modeler, Microsoft Visio, ERwin Data Modeler, Poseidon for UML, and SmartDraw.

Further Reading

The E-R data model was introduced by [Chen (1976)]. The Integration Definition for Information Modeling (IDEF1X) standard [NIST (1993)] released by the United States National Institute of Standards and Technology (NIST) defined standards for E-R diagrams. However, a variety of E-R notations are in use today.

[Thalheim (2000)] provides a detailed textbook coverage of research in E-R modeling.

As of 2018, the current UML version was 2.5, which was released in June 2015. See www.uml.org for more information on UML standards and tools.

Bibliography

[Chen (1976)] P. P. Chen, "The Entity-Relationship Model: Toward a Unified View of Data", *ACM Transactions on Database Systems*, Volume 1, Number 1 (1976), pages 9–36.

[NIST (1993)] NIST, "Integration Definition for Information Modeling (IDEF1X)", Technical Report Federal Information Processing Standards Publication 184, National Institute of Standards and Technology (NIST) (1993).

[Thalheim (2000)] B. Thalheim, *Entity-Relationship Modeling: Foundations of Database Technology*, Springer Verlag (2000).

Credits

The photo of the sailboats in the beginning of the chapter is due to ©Pavel Nesvadba/Shutterstock.

CHAPTER 6

Database Design using the E-R Model

Practice Exercises

6.1 Construct an E-R diagram for a car insurance company whose customers own one or more cars each. Each car has associated with it zero to any number of recorded accidents. Each insurance policy covers one or more cars and has one or more premium payments associated with it. Each payment is for a particular period of time, and has an associated due date, and the date when the payment was received.

Answer:

One possible E-R diagram is shown in Figure 6.101. Payments are modeled as weak entities since they are related to a specific policy.

Note that the participation of accident in the relationship *participated* is not total, since it is possible that there is an accident report where the participating car is unknown.

6.2 Consider a database that includes the entity sets *student*, *course*, and *section* from the university schema and that additionally records the marks that students receive in different exams of different sections.

a. Construct an E-R diagram that models exams as entities and uses a ternary relationship as part of the design.

b. Construct an alternative E-R diagram that uses only a binary relationship between *student* and *section*. Make sure that only one relationship exists between a particular *student* and *section* pair, yet you can represent the marks that a student gets in different exams.

Answer:

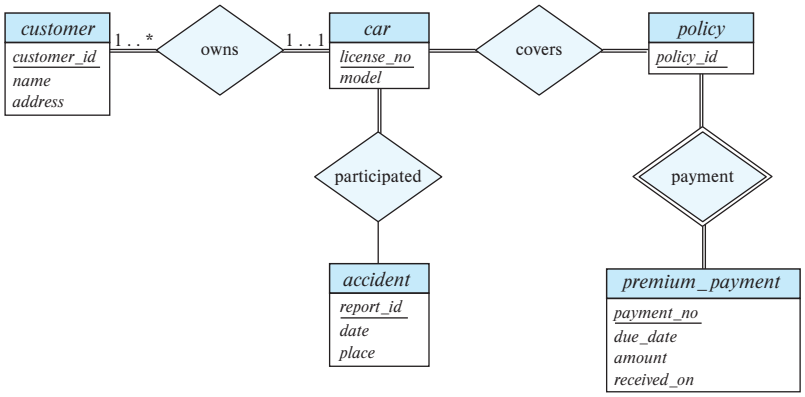


Figure 6.101 E-R diagram for a car insurance company.

a. The E-R diagram is shown in Figure 6.102. Note that an alternative is to model examinations as weak entities related to a section, rather than as strong entities. The marks relationship would then be a binary relationship between *student* and *exam*, without directly involving *section*.

b. The E-R diagram is shown in Figure 6.103. Note that here we have not modeled the name, place, and time of the exam as part of the relationship attributes. Doing so would result in duplication of the information, once per student, and we would not be able to record this information without an associated student. If we wish to represent this information, we need to retain a separate entity corresponding to each exam.

6.3 Design an E-R diagram for keeping track of the scoring statistics of your favorite sports team. You should store the matches played, the scores in each match, the players in each match, and individual player scoring statistics for each match.

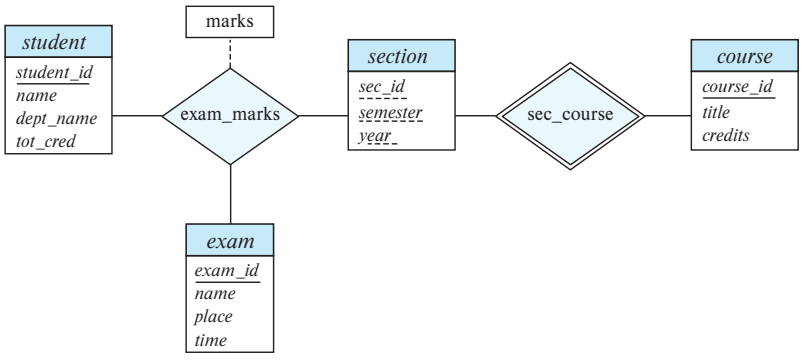


Figure 6.102 E-R diagram for marks database.

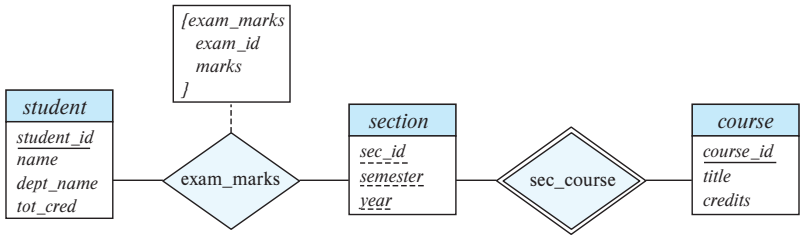


Figure 6.103 Another E-R diagram for marks database.

Summary statistics should be modeled as derived attributes with an explanation as to how they are computed.

Answer:

The diagram is shown in Figure 6.104. The derived attribute *season_score* is computed by summing the score values associated with the *player* entity set via the *played* relationship set.

6.4 Consider an E-R diagram in which the same entity set appears several times, with its attributes repeated in more than one occurrence. Why is allowing this redundancy a bad practice that one should avoid?

Answer:

The reason an entity set would appear more than once is if one is drawing a diagram that spans multiple pages.

The different occurrences of an entity set may have different sets of attributes, leading to an inconsistent diagram. Instead, the attributes of an entity set should be specified only once. All other occurrences of the entity should omit attributes. Since it is not possible to have an entity set without any attributes, an occurrence of an entity set without attributes clearly indicates that the attributes are specified elsewhere.

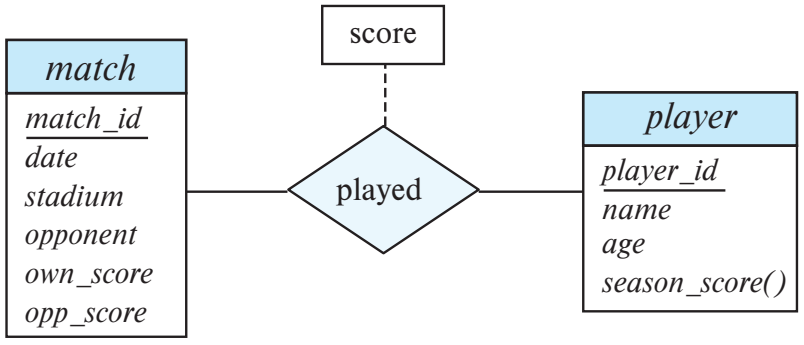


Figure 6.104 E-R diagram for favorite team statistics.

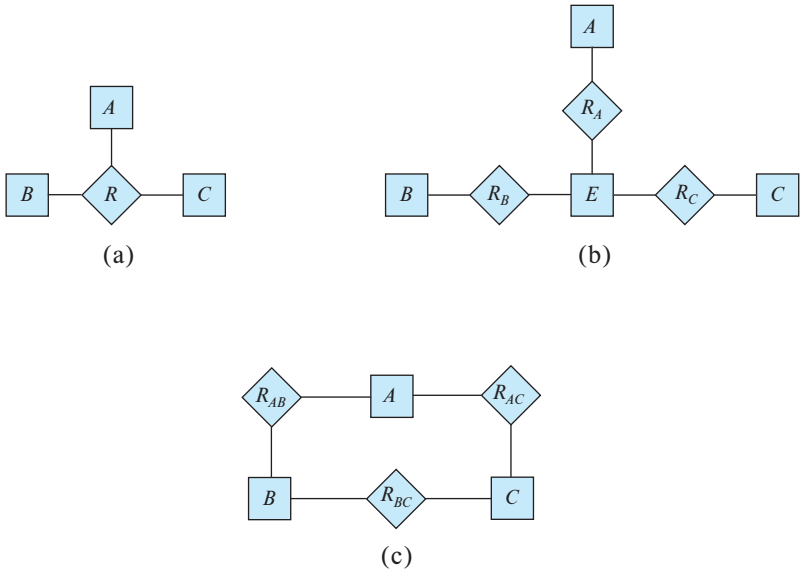


Figure 6.29 Representation of a ternary relationship using binary relationships.

6.5 An E-R diagram can be viewed as a graph. What do the following mean in terms of the structure of an enterprise schema?

a. The graph is disconnected.

b. The graph has a cycle.

Answer:

a. If a pair of entity sets are connected by a path in an E-R diagram, the entity sets are related, though perhaps indirectly. A disconnected graph implies that there are pairs of entity sets that are unrelated to each other. In an enterprise, we can say that the two parts of the enterprise are completely independent of each other. If we split the graph into connected components, we have, in effect, a separate database corresponding to each independent part of the enterprise.

b. As indicated in the answer to the previous part, a path in the graph between a pair of entity sets indicates a (possibly indirect) relationship between the two entity sets. If there is a cycle in the graph, then every pair of entity sets on the cycle are related to each other in at least two distinct ways. If the E-R diagram is acyclic, then there is a unique path between every pair of entity sets and thus a unique relationship between every pair of entity sets.

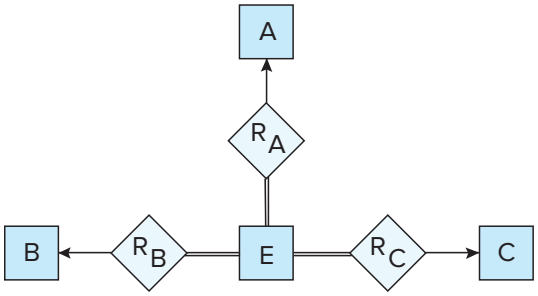


Figure 6.105 E-R diagram for Exercise Exercise 6.6b.

6.6 Consider the representation of the ternary relationship of Figure 6.29a using the binary relationships illustrated in Figure 6.29b (attributes not shown).

a. Show a simple instance of E, A, B, C, R_A, R_B, and R_C that cannot correspond to any instance of A, B, C, and R.

b. Modify the E-R diagram of Figure 6.29b to introduce constraints that will guarantee that any instance of E, A, B, C, R_A, R_B, and R_C that satisfies the constraints will correspond to an instance of A, B, C, and R.

c. Modify the preceding translation to handle total participation constraints on the ternary relationship.

Answer:

a. Let $E = \{e_1, e_2\}$, $A = \{a_1, a_2\}$, $B = \{b_1\}$, $C = \{c_1\}$, $R_A = \{(e_1, a_1), (e_2, a_2)\}$, $R_B = \{(e_1, b_1)\}$, and $R_C = \{(e_1, c_1)\}$. We see that because of the tuple (e_2, a_2), no instance of A, B, C, and R exists that corresponds to E, R_A, R_B and R_C.

b. See Figure 6.105. The idea is to introduce total participation constraints between E and the relationships R_A, R_B, R_C so that every tuple in E has a relationship with A, B, and C.

c. Suppose A totally participates in the relationhip R, then introduce a total participation constraint between A and R_A, and similarly for B and C.

6.7 A weak entity set can always be made into a strong entity set by adding to its attributes the primary-key attributes of its identifying entity set. Outline what sort of redundancy will result if we do so.

Answer:

The primary key of a weak entity set can be inferred from its relationship with the strong entity set. If we add primary-key attributes to the weak entity set, they will be present in both the entity set, and the relationship set and they have to be the same. Hence there will be redundancy.

6.8 Consider a relation such as *sec_course*, generated from a many-to-one relationship set *sec_course*. Do the primary and foreign key constraints created on the relation enforce the many-to-one cardinality constraint? Explain why.

Answer:
In this example, the primary key of *section* consists of the attributes (*course_id*, *sec_id*, *semester*, *year*), which would also be the primary key of *sec_course*, while *course_id* is a foreign key from *sec_course* referencing *course*. These constraints ensure that a particular *section* can only correspond to one *course*, and thus the many-to-one cardinality constraint is enforced.
However, these constraints cannot enforce a total participation constraint, since a course or a section may not participate in the *sec_course* relationship.

6.9 Suppose the *advisor* relationship set were one-to-one. What extra constraints are required on the relation *advisor* to ensure that the one-to-one cardinality constraint is enforced?

Answer:
In addition to declaring *s_ID* as primary key for *advisor*, we declare *i_ID* as a superkey for *advisor* (this can be done in SQL using the **unique** constraint on *i_ID*).

6.10 Consider a many-to-one relationship *R* between entity sets *A* and *B*. Suppose the relation created from *R* is combined with the relation created from *A*. In SQL, attributes participating in a foreign key constraint can be null. Explain how a constraint on total participation of *A* in *R* can be enforced using **not null** constraints in SQL.

Answer:
The foreign-key attribute in *R* corresponding to the primary key of *B* should be made **not null**. This ensures that no tuple of *A* which is not related to any entry in *B* under *R* can come in *R*. For example, say **a** is a tuple in *A* which has no corresponding entry in *R*. This means when *R* is combined with *A*, it would have a foreign-key attribute corresponding to *B* as **null**, which is not allowed.

6.11 In SQL, foreign key constraints can reference only the primary key attributes of the referenced relation or other attributes declared to be a superkey using the **unique** constraint. As a result, total participation constraints on a many-to-many relationship set (or on the "one" side of a one-to-many relationship set) cannot be enforced on the relations created from the relationship set, using primary key, foreign key, and not null constraints on the relations.

a. Explain why.

b. Explain how to enforce total participation constraints using complex check constraints or assertions (see Section 4.4.8). (Unfortunately, these features are not supported on any widely used database currently.)

Answer:

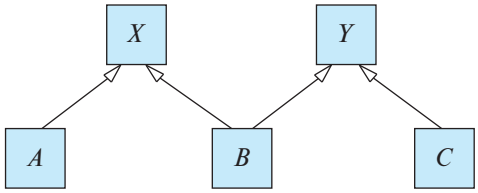
a. For the many-to-many case, the relationship set must be represented as a separate relation that cannot be combined with either participating entity. Now, there is no way in SQL to ensure that a primary-key value occurring in an entity $E1$ also occurs in a many-to-many relationship R, since the corresponding attribute in R is not unique; SQL foreign keys can only refer to the primary key or some other unique key.
Similarly, for the one-to-many case, there is no way to ensure that an attribute on the one side appears in the relation corresponding to the many side, for the same reason.

b. Let the relation R be many-to-one from entity A to entity B with a and b as their respective primary keys. We can put the following check constraints on the "one" side relation B:

constraint *total_part* **check** (b in (**select** b **from** A));
set constraints *total_part* **deferred**;

Note that the constraint should be set to deferred so that it is only checked at the end of the transaction; otherwise if we insert a b value in B before it is inserted in A, the constraint would be violated, and if we insert it in A before we insert it in B, a foreign-key violation would occur.

6.12 Consider the following lattice structure of generalization and specialization (attributes not shown).



For entity sets A, B, and C, explain how attributes are inherited from the higher-level entity sets X and Y. Discuss how to handle a case where an attribute of X has the same name as some attribute of Y.

Answer:
A inherits all the attributes of X, plus it may define its own attributes. Similarly, C inherits all the attributes of Y plus its own attributes. B inherits the attributes of both X and Y. If there is some attribute *name* which belongs to both X and Y, it may be referred to in B by the qualified name $X.name$ or $Y.name$.

6.13 An E-R diagram usually models the state of an enterprise at a point in time. Suppose we wish to track *temporal changes*, that is, changes to data over time. For example, Zhang may have been a student between September 2015 and

May 2019, while Shankar may have had instructor Einstein as advisor from May 2018 to December 2018, and again from June 2019 to January 2020. Similarly, attribute values of an entity or relationship, such as *title* and *credits* of *course*, *salary*, or even *name* of *instructor*, and *tot_cred* of *student*, can change over time.

One way to model temporal changes is as follows: We define a new data type called **valid_time**, which is a time interval, or a set of time intervals. We then associate a *valid_time* attribute with each entity and relationship, recording the time periods during which the entity or relationship is valid. The end time of an interval can be infinity; for example, if Shankar became a student in September 2018, and is still a student, we can represent the end time of the *valid_time* interval as infinity for the Shankar entity. Similarly, we model attributes that can change over time as a set of values, each with its own *valid_time*.

a. Draw an E-R diagram with the *student* and *instructor* entities, and the *advisor* relationship, with the above extensions to track temporal changes.

b. Convert the E-R diagram discussed above into a set of relations.

It should be clear that the set of relations generated is rather complex, leading to difficulties in tasks such as writing queries in SQL. An alternative approach, which is used more widely, is to ignore temporal changes when designing the E-R model (in particular, temporal changes to attribute values), and to modify the relations generated from the E-R model to track temporal changes.

Answer:

.

a. The E-R diagram is shown in Figure 6.106.
The primary key attributes *student_id* and *instructor_id* are assumed to be immutable, that is, they are not allowed to change with time. All other attributes are assumed to potentially change with time.

Note that the diagram uses multivalued composite attributes such as *valid_times* or *name*, with subattributes such as *start_time* or *value*. The *value* attribute is a subattribute of several attributes such as *name, tot_cred* and *salary*, and refers to the name, total credits or salary during a particular interval of time.

b. The generated relations are as shown below. Each multivalued attribute has turned into a relation, with the relation name consisting of the original relation name concatenated with the name of the multivalued attribute. The relation corresponding to the entity has only the primary-key attribute, and this is needed to ensure uniqueness.

student(student_id)
student_valid_times(student_id, start_time, end_time)
student_name(student_id, value, start_time, end_time
student_dept_name(student_id, value, start_time, end_time
student_tot_cred(student_id, value, start_time, end_time
instructor(instructor_id)
instructor_valid_times(instructor_id, start_time, end_time)
instructor_name(instructor_id, value, start_time, end_time
instructor_dept_name(instructor_id, value, start_time, end_time
instructor_salary(instructor_id, value, start_time, end_time
advisor(student_id, instructor_id, start_time, end_time)

The primary keys shown are derived directly from the E-R diagram. If we add the additional constraint that time intervals cannot overlap (or even the weaker condition that one start time cannot have two end times), we can remove the *end_time* from all the above primary keys.

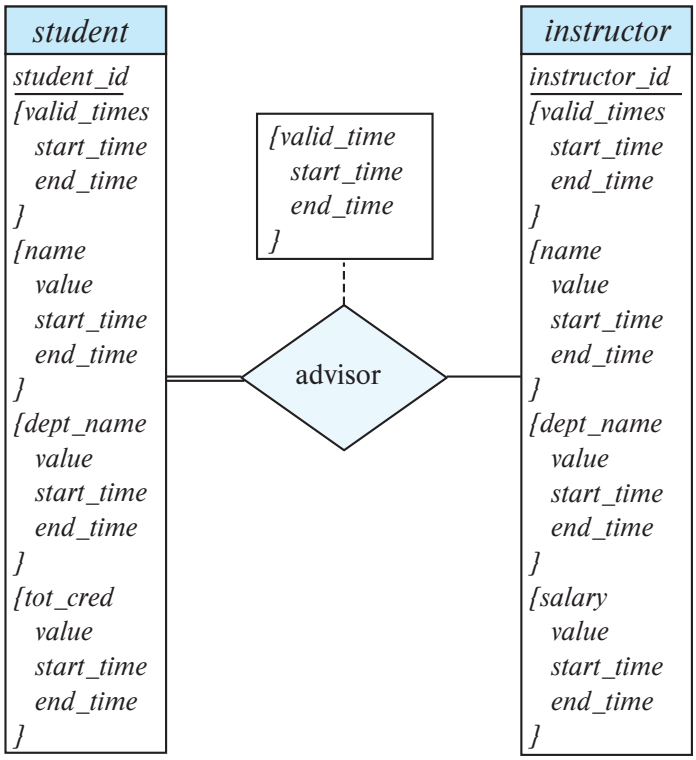


Figure 6.106 E-R diagram for Exercise 6.13

CHAPTER 5

The Basic Parts of Speech in SQL

ith the Structured Query Language (SQL), you tell Oracle which information you want it to **select**, **insert**, **update**, or **delete**. In fact, these four verbs are the primary words you will use to give Oracle instructions. As of Oracle9*i*, you can use an additional command, **merge**, to perform **insert**s and **update**s with a single command. As of Oracle Database 10*g*, the **merge** command capabilities have been extended to include greater control over the **insert**s, **update**s, and even **delete**s performed within a single statement. Also as of Oracle Database 10*g*, these basic commands are further enhanced to support flashback version queries and other features.

In Part I, you saw what is meant by "relational," how tables are organized into columns and rows, and how to instruct Oracle to select certain columns from a table and show you the information in them, row by row. In this and the following chapters, you will learn how to do this more completely for the different datatypes supported by Oracle. In this part, you will learn how to interact with SQL*Plus, a powerful Oracle product that can take your instructions for Oracle, check them for correctness, submit them to Oracle, and then modify or reformat the response Oracle gives, based on orders or directions you've set in place.

It may be a little confusing at first to understand the difference between what SQL*Plus is doing and what Oracle is doing, especially because the error messages that Oracle produces are simply passed on to you by SQL*Plus, but you will see as you work through this book where the differences lie. As you get started, just think of SQL*Plus as a coworker—an assistant who follows your instructions and helps you do your work more quickly. You interact with this coworker by typing on your keyboard.

You may follow the examples in this and subsequent chapters by typing the commands shown. Your Oracle and SQL*Plus programs should respond just as they do in these examples. However, you do need to make certain that the tables used in this book have been loaded into your copy of Oracle.

You can understand what is described in this book without actually typing it in yourself; for example, you can use the commands shown with your own tables. It will probably be clearer and easier, though, if you have the same tables loaded into Oracle as the ones used here and practice using the same queries.

The CD contains instructions on loading the tables. Assuming that you have loaded the demo tables into an Oracle database, you can connect to SQL*Plus and begin working by typing this:

```
sqlplus
```

(If you want to run SQL*Plus from your desktop client machine, select the SQL Plus program option from the Application Development menu option under the Oracle software menu option.) This starts SQL*Plus. (Note that you don't type the * that is in the middle of the official product name, and the asterisk doesn't appear in the program name, either. Because Oracle is careful to guard who can access the data it stores, it requires that you enter an ID and password to connect to it. Oracle will display a copyright message and then ask for your username and password. Log into your database using the account and password you created to hold the sample tables (such as practice/practice). If you provide a valid username and password, SQL*Plus will announce that you're connected to Oracle and then will display this prompt:

```
SQL>
```

You are now in SQL*Plus, and it awaits your instructions. If the command fails, there are several potential reasons: Oracle is not in your path, you are not authorized to use SQL*Plus, or Oracle hasn't been installed properly on your computer. If you get the message

```
ERROR: ORA-1017: invalid username/password; logon denied
```

either you've entered the username or password incorrectly or your username has not yet been set up properly on your copy of Oracle. After three unsuccessful attempts to enter a username and password that Oracle recognizes, SQL*Plus will terminate the attempt to log on, showing this message:

```
unable to CONNECT to ORACLE after 3 attempts, exiting SQL*Plus
```

If you get this message, contact your company's database administrator. Assuming everything is in order, and the SQL> prompt has appeared, you may now begin working with SQL*Plus.

When you want to quit working and leave SQL*Plus, type this:

```
quit
```

Style

First, some comments on style. SQL*Plus doesn't care whether the SQL commands you type are in uppercase or lowercase. For example, the command

```
SeLeCt feaTURE, section, PAGE FROM newsPaPeR;
```

will produce exactly the same result as this one:

```
select Feature, Section, Page from NEWSPAPER;
```

Case matters only when SQL*Plus or Oracle is checking an alphanumeric value for equality. If you tell Oracle to find a row where Section = 'f', and Section is really equal to 'F', Oracle won't find it (because *f* and *F* are not identical). Aside from this usage, case is completely irrelevant. (Incidentally, 'F', as used here, is called a *literal,* meaning that you want Section to be tested literally against the letter *F*, not a column *named* F. The single quote marks enclosing the letter tell Oracle that this is a literal and not a column name.)

As a matter of style, this book follows certain conventions about case to make the text easier to read:

- **select**, **from**, **where**, **order by**, **having**, and **group by** will always be lowercase and boldface in the body of the text.
- SQL*Plus commands also will be lowercase and boldface (for example, **column**, **set**, **save**, **ttitle**, and so on).
- **IN**, **BETWEEN**, **UPPER**, and other SQL operators and functions will be uppercase and boldface.

- Column names will be mixed uppercase and lowercase without boldface (for example, Feature, EastWest, Longitude, and so on).
- Table names will be uppercase without boldface (for example, NEWSPAPER, WEATHER, LOCATION, and so on).

You may want to follow similar conventions in creating your own queries, or your company already may have standards it would like you to use. You may even choose to invent your own. Regardless, the goal of any such standards should always be to make your work simple to read and understand.

Creating the NEWSPAPER Table

The examples in this book are based on the tables created by the scripts located on the CD. Each table is created via the **create table** command, which specifies the names of the columns in the table, as well as the characteristics of those columns. Here is the **create table** command for the NEWSPAPER table, which is used in many of the examples in this chapter:

```
create table NEWSPAPER (
Feature      VARCHAR2(15) not null,
Section      CHAR(1),
Page         NUMBER
);
```

In later chapters in this book, you'll see how to interpret all the clauses of this command. For now, you can read it as, "Create a table called NEWSPAPER. It will have three columns, named Feature (a variable-length character column), Section (a fixed-length character column), and Page (a numeric column). The values in the Feature column can be up to 15 characters long, and every row must have a value for Feature. Section values will all be one character long."

In later chapters, you'll see how to extend this simple command to add constraints, indexes, and storage clauses. For now, the NEWSPAPER table will be kept simple so that the examples can focus on SQL.

Using SQL to Select Data from Tables

Figure 5-1 shows a table of features from a local newspaper. If this were an Oracle table, rather than just paper and ink on the front of the local paper, SQL*Plus would display it for you if you typed this:

```
select Feature, Section, Page from NEWSPAPER;

FEATURE         S       PAGE
--------------- - ----------
National News   A          1
Sports          D          1
Editorials      A         12
Business        E          1
Weather         C          2
```

```
Television      B          7
Births          F          7
Classified      F          8
Modern Life     B          1
Comics          C          4
Movies          B          4
Bridge          B          2
Obituaries      F          6
Doctor Is In    F          6

14 rows selected.
```

NOTE

Depending on your configuration, the listing may have a page break in it. If that happens, use the ***set pagesize*** *command to increase the size of each displayed page of results. See the Alphabetical Reference for details on this command.*

What's different between the table you created and the one shown in the output in Figure 5-1? Both tables have the same information, but the format differs and the order of rows may be different.

Feature	Section	Page
Births	F	7
Bridge	B	2
Business	E	1
Classified	F	8
Comics	C	4
Doctor Is In	F	6
Editorials	A	12
Modern Life	B	1
Movies	B	4
National News	A	1
Obituaries	F	6
Sports	D	1
Television	B	7
Weather	C	2

FIGURE 5-1. *A NEWSPAPER table*

For example, the column headings differ slightly. In fact, they even differ slightly from the columns you just asked for in the **select** statement.

Notice that the column named Section shows up as just the letter *S*. Also, although you used uppercase and lowercase letters to type the column headings in the command

```
select Feature, Section, Page from NEWSPAPER;
```

the column headings came back with all the letters in uppercase.

These changes are the result of the assumptions SQL*Plus makes about how information should be presented. You can change these assumptions, and you probably will, but until you give SQL*Plus different orders, this is how it changes what you input:

- It changes all the column headings to uppercase.
- It allows columns to be only as wide as a column is defined to be in Oracle.
- It squeezes out any spaces if the column heading is a function. (This will be demonstrated in Chapter 7.)

The first point is obvious. The column names you used were shifted to uppercase. The second point is not obvious. How *are* the columns defined? To find out, ask Oracle. Simply tell SQL*Plus to **describe** the table, as shown here:

```
describe NEWSPAPER

Name                           Null?    Type
------------------------------ -------- ------------
FEATURE                        NOT NULL VARCHAR2(15)
SECTION                                 CHAR(1)
PAGE                                    NUMBER
```

This display is a descriptive table that lists the columns and their definitions for the NEWSPAPER table; the **describe** command works for any table. Note that the details in this description match the **create table** command given earlier in this chapter.

The first column tells the names of the columns in the table being described.

The second column (Null?) is really a rule about the column named to its left. When the NEWSPAPER table was created, the **NOT NULL** rule instructed Oracle not to allow any user to add a new row to the table if he or she left the Feature column empty (**NULL** means empty). Of course, in a table such as NEWSPAPER, it probably would have been worthwhile to use the same rule for all three columns. What good is it to know the title of a feature without also knowing what section it's in and what page it's on? But, for the sake of this example, only Feature was created with the rule that it could not be **NULL**.

Because Section and Page have nothing in the Null? column, they are allowed to be empty in any row of the NEWSPAPER table.

The third column (Type) tells the basic nature of the individual columns. Feature is a VARCHAR2 (variable-length character) column that can be up to 15 characters (letters, numbers, symbols, or spaces) long.

Section is a character column as well, but it is only one character long. The creator of the table knew that newspaper sections in the local paper are only a single letter, so the column was defined

to be only as wide as it needed to be. It was defined using the CHAR datatype, which is used for fixed-length character strings. When SQL*Plus went to display the results of your query

```
select Feature, Section, Page from NEWSPAPER;
```

it knew from Oracle that Section was a maximum of only one character. It assumed that you did not want to use up more space than this, so it displayed a column just one character wide and used as much of the column name as it could—in this case, just the letter *S*.

The third column in the NEWSPAPER table is Page, which is simply a number. Notice that the Page column shows up as ten spaces wide, even though no pages use more than two digits—numbers usually are not defined as having a maximum width, so SQL*Plus assumes a maximum just to get started.

You also may have noticed that the heading for the only column composed solely of numbers, Page, was *right-justified*—that is, it sits over on the right side of the column, whereas the headings for columns that contain characters sit over on the left. This is standard alignment for column headings in SQL*Plus. As with other column features, you'll see in Chapter 6 how to change alignment as needed.

Finally, SQL*Plus tells you how many rows it found in Oracle's NEWSPAPER table. (Notice the "14 rows selected" notation at the bottom of the display.) This is called *feedback.* You can make SQL*Plus stop giving feedback by setting the **feedback** option, as shown here:

```
set feedback off
```

Alternatively, you can set a minimum number of rows for **feedback** to work:

```
set feedback 25
```

This last example tells Oracle that you don't want to know how many rows have been displayed until there have been at least 25. Unless you tell SQL*Plus differently, **feedback** is set to 6.

The **set** command is a SQL*Plus command, which means that it is an instruction telling SQL*Plus how to act. There are many SQL*Plus options, such as **feedback**, that you can set. Several of these will be shown and used in this chapter and in the chapters to follow. For a complete list, look up **set** in the Alphabetical Reference section of this book.

The **set** command has a counterpart named **show** that allows you to see what instructions you've given to SQL*Plus. For instance, you can check the setting of **feedback** by typing

```
show feedback
```

SQL*Plus will respond with the following:

```
FEEDBACK ON for 25 or more rows
```

The width used to display numbers also is changed by the **set** command. You check it by typing

```
show numwidth
```

SQL*Plus will reply as shown here:

```
numwidth 9
```

Because 9 is a wide width for displaying page numbers that never contain more than two digits, shrink the display by typing

```
set numwidth 5
```

However, this means that all number columns will be five digits wide. If you anticipate having numbers with more than five digits, you must use a number higher than 5. Individual columns in the display also can be set independently. This will be covered in Chapter 6.

select, from, where, and order by

You will use four primary keywords in SQL when selecting information from an Oracle table: **select**, **from**, **where**, and **order by**. You will use **select** and **from** in every Oracle query you do.

The **select** keyword tells Oracle which columns you want, and **from** tells Oracle the name(s) of the table(s) those columns are in. The NEWSPAPER table example showed how these keywords are used. In the first line that you entered, a comma follows each column name except the last. You'll notice that a correctly typed SQL query reads pretty much like an English sentence. A query in SQL*Plus usually ends with a semicolon (sometimes called the *SQL terminator*). The **where** keyword tells Oracle what qualifiers you'd like to put on the information it is selecting. For example, if you input

```
select Feature, Section, Page from NEWSPAPER
 where Section = 'F';

FEATURE         S  PAGE
--------------- - -----
Births          F     7
Classified      F     8
Obituaries      F     6
Doctor Is In    F     6
```

Oracle checks each row in the NEWSPAPER table before sending the row back to you. It skips over those without the single letter *F* in their Section column. It returns those where the Section entry is 'F', and SQL*Plus displays them to you.

To tell Oracle that you want the information it returns sorted in the order you specify, use **order by**. You can be as elaborate as you like about the order you request. Consider these examples:

```
select Feature, Section, Page from NEWSPAPER
 where Section = 'F'
 order by Feature;

FEATURE         S  PAGE
--------------- - -----
Births          F     7
Classified      F     8
Doctor Is In    F     6
Obituaries      F     6
```

They are nearly reversed when ordered by page, as shown here:

```
select Feature, Section, Page from NEWSPAPER
 where Section = 'F'
 order by Page;

FEATURE         S  PAGE
--------------- - -----
Obituaries      F     6
Doctor Is In    F     6
Births          F     7
Classified      F     8
```

In the next example, Oracle first puts the features in order by page (see the previous listing to observe the order they are in when they are ordered only by page). It then puts them in further order by feature, listing Doctor Is In ahead of Obituaries.

```
select Feature, Section, Page from NEWSPAPER
 where Section = 'F'
 order by Page, Feature;

FEATURE         S  PAGE
--------------- - -----
Doctor Is In    F     6
Obituaries      F     6
Births          F     7
Classified      F     8
```

Using **order by** also can reverse the normal order, like this:

```
select Feature, Section, Page from NEWSPAPER
 where Section = 'F'
 order by Page desc, Feature;

FEATURE         S  PAGE
--------------- - -----
Classified      F     8
Births          F     7
Doctor Is In    F     6
Obituaries      F     6
```

The **desc** keyword stands for *descending.* Because it followed the word Page in the **order by** line, it put the page numbers in descending order. It would have the same effect on the Feature column if it followed the word Feature in the **order by** line.

Notice that each of these keywords—**select**, **from**, **where**, and **order by—**has its own way of structuring the words that follow it. The groups of words including these keywords are often called *clauses,* as shown in Figure 5-2.

```
Select Feature, Section, Page        <--select clause
  from NEWSPAPER                     <--from clause
  where Section = 'F'                <--where clause
```

FIGURE 5-2. *Clauses*

Logic and Value

Just as the **order by** clause can have several parts, so can the **where** clause, but with a significantly greater degree of sophistication. You control the extent to which you use **where** through the careful use of logical instructions to Oracle on what you expect it to return to you. These instructions are expressed using mathematical symbols called *logical operators.* These are explained shortly, and they also are listed in the Alphabetical Reference section of this book.

The following is a simple example in which the values in the Page column are tested to see if any equals 6. Every row where this is true is returned to you. Any row in which Page is not equal to 6 is skipped (in other words, those rows for which Page = 6 is false).

```
select Feature, Section, Page
  from NEWSPAPER
 where Page = 6;

FEATURE         S  PAGE
--------------- - -----
Obituaries      F     6
Doctor Is In    F     6
```

The equal sign is called a *logical operator,* because it operates by making a logical test that compares the values on either side of it—in this case, the value of Page and the value 6—to see if they are equal.

In this example, no quotes are placed around the value being checked because the column the value is compared to (the Page column) is defined as a NUMBER datatype. Number values do not require quotes around them during comparisons.

Single-Value Tests

You can use one of several logical operators to test against a single value, as shown in the upcoming sidebar "Logical Tests Against a Single Value." Take a few examples from any of the expressions listed in this sidebar. They all work similarly and can be combined at will, although they must follow certain rules about how they'll act together.

Logical Tests Against a Single Value

All these operators work with letters or numbers, and with columns or literals.

Equal, Greater Than, Less Than, Not Equal

Page=	6	Page is equal to 6.
Page>	6	Page is greater than 6.
Page>=	6	Page is greater than or equal to 6.
Page<	6	Page is less than 6.
Page<=	6	Page is less than or equal to 6.
Page!=	6	Page is not equal to 6.
Page^=	6	Page is not equal to 6.
Page<>	6	Page is not equal to 6.

Because some keyboards lack an exclamation mark (!) or a caret (^), Oracle allows three ways of typing the not equal operator. The final alternative, <>, qualifies as a not equal operator because it permits only numbers less than 6 (in this example) or greater than 6, but not 6 itself.

LIKE

Feature LIKE 'Mo%'	Feature begins with the letters *Mo*.
Feature LIKE '_ _ I%'	Feature has an *I* in the third position.
Feature LIKE '%o%o%'	Feature has two *o*'s in it.

LIKE performs pattern matching. An underline character (_) represents exactly one character. A percent sign (%) represents any number of characters, including zero characters.

IS NULL, IS NOT NULL

Precipitation IS NULL	Precipitation is unknown.
Precipitation IS NOT NULL	Precipitation is known.

NULL tests to see if data exists in a column for a row. If the column is completely empty, it is said to be "null." The word **IS** must be used with **NULL** and **NOT NULL**; equal, greater-than, and less-than signs do not work with **NULL** and **NOT NULL.**

Equal, Greater Than, Less Than, Not Equal

Logical tests can compare values, both for equality and for relative value. Here, a simple test is made for all sections equal to 'B':

```
select Feature, Section, Page
  from NEWSPAPER
 where Section = 'B';

FEATURE         S  PAGE
--------------- - -----
Television      B     7
Modern Life     B     1
Movies          B     4
Bridge          B     2
```

The following is the test for all pages greater than 4:

```
select Feature, Section, Page
  from NEWSPAPER
 where Page > 4;

FEATURE         S  PAGE
--------------- - -----
Editorials      A    12
Television      B     7
Births          F     7
Classified      F     8
Obituaries      F     6
Doctor Is In    F     6
```

The following is the test for sections greater than 'B' (this means later in the alphabet than the letter *B*):

```
select Feature, Section, Page
  from NEWSPAPER
 where Section > 'B';

FEATURE         S  PAGE
--------------- - -----
Sports          D     1
Business        E     1
Weather         C     2
Births          F     7
Classified      F     8
Comics          C     4
Obituaries      F     6
Doctor Is In    F     6
```

Just as a test can be made for greater than, so can a test be made for less than, as shown here (all page numbers less than 8):

```
select Feature, Section, Page
  from NEWSPAPER
 where Page < 8;

FEATURE         S       PAGE
--------------- - ----------
National News   A          1
Sports          D          1
Business        E          1
Weather         C          2
Television      B          7
Births          F          7
Modern Life     B          1
Comics          C          4
Movies          B          4
Bridge          B          2
Obituaries      F          6
Doctor Is In    F          6
```

The opposite of the test for equality is the not equal test, as given here:

```
select Feature, Section, Page
  from NEWSPAPER
 where Page <> 1;

FEATURE         S       PAGE
--------------- - ----------
Editorials      A         12
Weather         C          2
Television      B          7
Births          F          7
Classified      F          8
Comics          C          4
Movies          B          4
Bridge          B          2
Obituaries      F          6
Doctor Is In    F          6
```

NOTE

Be careful when using the greater-than and less-than operators against numbers that are stored in character datatype columns. All values in VARCHAR2 and CHAR columns will be treated as characters during comparisons. Therefore, numbers that are stored in those types of columns will be compared as if they were character strings, not numbers. If the column's datatype is NUMBER, then 12 is greater than 9. If it is a character column, then 9 is greater than 12, because the character '9' is greater than the character '1'.

LIKE

One of the most powerful features of SQL is a marvelous pattern-matching operator called **LIKE**, which is able to search through the rows of a database column for values that look like a pattern you describe. It uses two special characters to denote which kind of matching to do: a percent sign, called a *wildcard,* and an underline, called a *position marker.* To look for all the features that begin with the letters *Mo,* use the following:

```
select Feature, Section, Page from NEWSPAPER
 where Feature LIKE 'Mo%';

FEATURE         S       PAGE
--------------- - ----------
Modern Life     B          1
Movies          B          4
```

The percent sign (%) means anything is acceptable here: one character, a hundred characters, or no characters. If the first letters are *Mo,* **LIKE** will find the feature. If the query had used 'MO%' as its search condition instead, then no rows would have been returned, due to Oracle's case-sensitivity in data values.

If you want to find those features that have the letter *i* in the third position of their titles, and you don't care which two characters precede the *i* or what set of characters follows, using two underlines (_ _) specifies that any character in those two positions is acceptable. Position three must have a lowercase *i,* and the percent sign after that says anything is okay.

```
select Feature, Section, Page from NEWSPAPER
 where Feature LIKE '__i%';

FEATURE         S       PAGE
--------------- - ----------
Editorials      A         12
Bridge          B          2
Obituaries      F          6
```

Multiple percent signs also can be used. To find those words with two lowercase *o*'s anywhere in the Feature title, three percent signs are used, as shown here:

```
select Feature, Section, Page from NEWSPAPER
 where Feature LIKE '%o%o%';

FEATURE         S       PAGE
--------------- - ----------
Doctor Is In    F          6
```

For the sake of comparison, the following is the same query, but it is looking for two *i*'s:

```
select Feature, Section, Page from NEWSPAPER
 where Feature LIKE '%i%i%';

FEATURE         S       PAGE
--------------- - ----------
Editorials      A         12
Television      B          7
Classified      F          8
Obituaries      F          6
```

This pattern-matching feature can play an important role in making an application friendlier by simplifying searches for names, products, addresses, and other partially remembered items. In Chapter 8, you will see how to use advanced regular expression searches available as of Oracle Database 10*g*.

NULL and NOT NULL

The NEWSPAPER table has no columns in it that are **NULL**, even though the **describe** you did on it showed that they are allowed. The following query on the COMFORT table contains, among other data, the precipitation for San Francisco, California, and Keene, New Hampshire, for four sample dates during 2003:

```
select City, SampleDate, Precipitation
  from COMFORT;

CITY          SAMPLEDAT PRECIPITATION
------------- --------- -------------
SAN FRANCISCO 21-MAR-03            .5
SAN FRANCISCO 22-JUN-03            .1
SAN FRANCISCO 23-SEP-03            .1
SAN FRANCISCO 22-DEC-03           2.3
KEENE         21-MAR-03           4.4
KEENE         22-JUN-03           1.3
KEENE         23-SEP-03
KEENE         22-DEC-03           3.9
```

You can find out the city and dates on which precipitation was not measured with this query:

```
select City, SampleDate, Precipitation
  from COMFORT
 where Precipitation IS NULL;

CITY          SAMPLEDAT PRECIPITATION
------------- --------- -------------
KEENE         23-SEP-03
```

IS NULL essentially instructs Oracle to identify columns in which the data is missing. You don't know for that day whether the value should be 0, 1, or 5 inches. Because it is unknown, the value

in the column is not set to 0; it stays empty. By using **NOT**, you also can find those cities and dates for which data exists, with this query:

```
select City, SampleDate, Precipitation
  from COMFORT
 where Precipitation IS NOT NULL;

CITY          SAMPLEDAT PRECIPITATION
------------- --------- -------------
SAN FRANCISCO 21-MAR-03            .5
SAN FRANCISCO 22-JUN-03            .1
SAN FRANCISCO 23-SEP-03            .1
SAN FRANCISCO 22-DEC-03           2.3
KEENE         21-MAR-03           4.4
KEENE         22-JUN-03           1.3
KEENE         22-DEC-03           3.9
```

Oracle lets you use the relational operators (=, !=, and so on) with **NULL**, but this kind of comparison will not return meaningful results. Use **IS** or **IS NOT** for comparing values to **NULL**.

Simple Tests Against a List of Values

If there are logical operators that test against a single value, are there others that will test against many values, such as a list? The sidebar "Logical Tests Against a List of Values" shows just such a group of operators.

Logical Tests Against a List of Values

Logical tests with numbers:

Page IN (1,2,3)	Page is in the list (1,2,3)
Page NOT IN (1,2,3)	Page is not in the list (1,2,3)
Page BETWEEN 6 AND 10	Page is equal to 6, 10, or anything in between
Page NOT BETWEEN 6 AND 10	Page is below 6 or above 10

With letters (or characters):

Section IN ('A','C','F')	Section is in the list ('A','C','F')
Section NOT IN ('A','C','F')	Section is not in the list ('A','C','F')
Section BETWEEN 'B' AND 'D'	Section is equal to 'B', 'D', or anything in between (alphabetically)
Section NOT BETWEEN 'B' AND 'D'	Section is below 'B' or above 'D' (alphabetically)

Here are a few examples of how these logical operators are used:

```
select Feature, Section, Page
  from NEWSPAPER
 where Section IN ('A','B','F');

FEATURE         S       PAGE
--------------- - ----------
National News   A          1
Editorials      A         12
Television      B          7
Births          F          7
Classified      F          8
Modern Life     B          1
Movies          B          4
Bridge          B          2
Obituaries      F          6
Doctor Is In    F          6

select Feature, Section, Page
  from NEWSPAPER
 where Section NOT IN ('A','B','F');

FEATURE         S       PAGE
--------------- - ----------
Sports          D          1
Business        E          1
Weather         C          2
Comics          C          4

select Feature, Section, Page
  from NEWSPAPER
 where Page BETWEEN 7 and 10;

FEATURE         S       PAGE
--------------- - ----------
Television      B          7
Births          F          7
Classified      F          8
```

These logical tests also can be combined, as in this case:

```
select Feature, Section, Page
  from NEWSPAPER
 where Section = 'F'
  AND Page > 7;

FEATURE         S       PAGE
--------------- - ----------
Classified      F          8
```

The **AND** command has been used to combine two logical expressions and requires any row Oracle examines to pass *both* tests; both Section = 'F' and Page > 7 must be true for a row to be returned to you. Alternatively, **OR** can be used, which will return rows to you if *either* logical expression turns out to be true:

```
select Feature, Section, Page
  from NEWSPAPER
 where Section = 'F'
    OR Page > 7;

FEATURE         S  PAGE
--------------- - -----
Editorials      A    12
Births          F     7
Classified      F     8
Obituaries      F     6
Doctor Is In    F     6
```

There are some sections here that qualify even though they are not equal to 'F' because their page is greater than 7, and there are others whose page is less than or equal to 7 but whose section is equal to 'F'.

Finally, choose those features in Section F between pages 7 and 10 with this query:

```
select Feature, Section, Page
  from NEWSPAPER
 where Section = 'F'
   and Page BETWEEN 7 AND 10;

FEATURE         S  PAGE
--------------- - -----
Births          F     7
Classified      F     8
```

There are a few additional *many-value operators* whose use is more complex; they will be covered in Chapter 9. They also can be found, along with those just discussed, in the Alphabetical Reference section of this book.

Combining Logic

Both **AND** and **OR** follow the commonsense meanings of the words. They can be combined in a virtually unlimited number of ways, but you must use care, because **AND**s and **OR**s get convoluted very easily.

Suppose you want to find the features in the paper that the editors tend to bury, those that are placed somewhere past page 2 of section A or B. You might try this:

```
select Feature, Section, Page
  from NEWSPAPER
 where Section = 'A'
    or Section = 'B'
   and Page > 2;
```

```
FEATURE         S  PAGE
--------------- - -----
National News   A     1
Editorials      A    12
Television      B     7
Movies          B     4
```

Note that the result you got back from Oracle is not what you wanted. Somehow, page 1 of section A was included. Why is this happening? Is there a way to get Oracle to answer the question correctly? Although both **AND** and **OR** are logical connectors, **AND** is stronger. It binds the logical expressions on either side of it more strongly than **OR** does (technically, **AND** is said to have *higher precedence)*, which means the **where** clause

```
where Section = 'A'
   or Section = 'B'
  and Page > 2;
```

is interpreted to read, "where Section = 'A', or where Section = 'B' *and* Page > 2." If you look at the failed example just given, you'll see how this interpretation affected the result. The **AND** is always acted on first.

You can break this bonding by using parentheses that enclose those expressions you want to be interpreted together. Parentheses override the normal precedence:

```
select Feature, Section, Page
  from NEWSPAPER
 where Page > 2
   and ( Section = 'A' or Section = 'B' );

FEATURE            S PAGE
------------------ - ----
Editorials         A   12
Television         B    7
Movies             B    4
```

The result is exactly what you wanted in the first place. Note that although you can type this with the sections listed first, the result is identical because the parentheses tell Oracle what to interpret together. Compare this to the different results caused by changing the order in the first example, where parentheses were not used.

Another Use for where: Subqueries

What if the logical operators in the previous sidebars, "Logical Tests Against a Single Value" and "Logical Tests Against a List of Values," could be used not just with a single literal value (such as 'F') or a typed list of values (such as 4,2,7 or 'A','C','F'), but with values brought back by an Oracle query? In fact, this is a powerful feature of SQL.

Imagine that you are the author of the "Doctor Is In" feature, and each newspaper that publishes your column sends along a copy of the table of contents that includes your piece. Of course, each editor rates your importance a little differently, and places you in a section he or she deems suited to your feature. Without knowing ahead of time where your feature is, or with what other features

you are placed, how could you write a query to find out where a particular local paper places you? You might do this:

```
select Section from NEWSPAPER
 where Feature = 'Doctor Is In';

S
-
F
```

The result is 'F'. Knowing this, you could do this query:

```
select FEATURE from NEWSPAPER
 where Section = 'F';

FEATURE
---------------
Births
Classified
Obituaries
Doctor Is In
```

You're in there with births, deaths, and classified ads. Could the two separate queries have been combined into one? Yes, as shown here:

```
select FEATURE from NEWSPAPER
 where Section = (select Section from NEWSPAPER
                   where Feature = 'Doctor Is In');

FEATURE
---------------
Births
Classified
Obituaries
Doctor Is In
```

Single Values from a Subquery

In effect, the **select** in parentheses (called a *subquery*) brought back a single value, F. The main query then treated this value as if it were a literal 'F', as was used in the previous query. Remember that the equal sign is a single-value test. It can't work with lists, so if your subquery returned more than one row, you'd get an error message like this:

```
select * from NEWSPAPER
 where Section = (select Section from NEWSPAPER
                   where Page = 1);
where Section = (select Section from NEWSPAPER
                 *
ERROR at line 2:
ORA-01427:  single-row subquery returns more than one row
```

All the logical operators that test single values can work with subqueries, as long as the subquery returns a single row. For instance, you can ask for all the features in the paper where the section is *less than* (that is, earlier in the alphabet) the section that carries your column. The asterisk in this **select** shows a shorthand way to request all the columns in a table without listing them individually. They will be displayed in the order in which they were created in the table.

```
select * from NEWSPAPER
 where Section < (select Section from NEWSPAPER
                  where Feature = 'Doctor Is In');

FEATURE         S  PAGE
--------------- - -----
National News   A     1
Sports          D     1
Editorials      A    12
Business        E     1
Weather         C     2
Television      B     7
Modern Life     B     1
Comics          C     4
Movies          B     4
Bridge          B     2

10 rows selected.
```

Ten other features rank ahead of your medical advice in this local paper.

Lists of Values from a Subquery

Just as the single-value logical operators can be used on a subquery, so can the many-value operators. If a subquery returns one or more rows, the value in the column for each row will be stacked up in a list. For example, suppose you want to know the cities and countries where it is cloudy. You could have a table of complete weather information for all cities, and a LOCATION table for all cities and their countries, as shown here:

```
select City, Country from LOCATION;

CITY                      COUNTRY
------------------------- --------------------------
ATHENS                    GREECE
CHICAGO                   UNITED STATES
CONAKRY                   GUINEA
LIMA                      PERU
MADRAS                    INDIA
MANCHESTER                ENGLAND
MOSCOW                    RUSSIA
PARIS                     FRANCE
SHENYANG                  CHINA
ROME                      ITALY
TOKYO                     JAPAN
```

```
SYDNEY                    AUSTRALIA
SPARTA                    GREECE
MADRID                    SPAIN

select City, Condition from WEATHER;

CITY        CONDITION
----------- -----------
LIMA        RAIN
PARIS       CLOUDY
MANCHESTER  FOG
ATHENS      SUNNY
CHICAGO     RAIN
SYDNEY      SUNNY
SPARTA      CLOUDY
```

First, you'd discover which cities were cloudy:

```
select City from WEATHER
 where Condition = 'CLOUDY';

CITY
-----------
PARIS
SPARTA
```

Then, you would build a list including those cities and use it to query the LOCATION table:

```
select City, Country from LOCATION
 where City IN ('PARIS', 'SPARTA');

CITY                      COUNTRY
------------------------- -------------------------
PARIS                     FRANCE
SPARTA                    GREECE
```

The same task can be accomplished by a subquery, where the **select** in parentheses builds a list of cities that are tested by the **IN** operator, as shown here:

```
select City, Country from LOCATION
 where City IN (select City from WEATHER
                 where Condition = 'CLOUDY');

CITY                      COUNTRY
------------------------- -------------------------
PARIS                     FRANCE
SPARTA                    GREECE
```

The other many-value operators work similarly. The fundamental task is to build a subquery that produces a list that can be logically tested. The following are some relevant points:

- The subquery must either have only one column or compare its selected columns to multiple columns in parentheses in the main query (covered in Chapter 13).
- The subquery must be enclosed in parentheses.
- Subqueries that produce only one row can be used with *either* single- or many-value operators.
- Subqueries that produce more than one row can be used *only* with many-value operators.

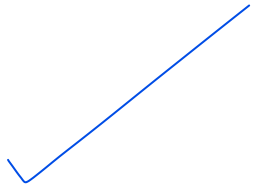
Combining Tables

If you've normalized your data, you'll probably need to combine two or more tables to get all the information you want.

Suppose you are the oracle at Delphi. The Athenians come to ask about the forces of nature that might affect the expected attack by the Spartans, as well as the direction from which they are likely to appear:

```
select City, Condition, Temperature from WEATHER;

CITY        CONDITION   TEMPERATURE
----------- ----------- -----------
LIMA        RAIN                 45
PARIS       CLOUDY               81
MANCHESTER  FOG                  66
ATHENS      SUNNY                97
CHICAGO     RAIN                 66
SYDNEY      SUNNY                69
SPARTA      CLOUDY               74
```



You realize your geography is rusty, so you query the LOCATION table:

```
select City, Longitude, EastWest, Latitude, NorthSouth
  from LOCATION;

CITY                      LONGITUDE E LATITUDE N
------------------------- --------- - -------- -
ATHENS                        23.43 E    37.58 N
CHICAGO                       87.38 W    41.53 N
CONAKRY                       13.43 W     9.31 N
LIMA                          77.03 W    12.03 S
MADRAS                        80.17 E    13.05 N
MANCHESTER                     2.15 W     53.3 N
MOSCOW                        37.35 E    55.45 N
PARIS                           2.2 E    48.52 N
SHENYANG                     123.27 E    41.48 N
ROME                          12.29 E    41.54 N
TOKYO                        139.46 E    35.42 N
SYDNEY                       151.13 E    33.52 S
SPARTA                        22.27 E    37.05 N
MADRID                         3.41 W    40.24 N
```

This is much more than you need, and it doesn't have any weather information. Yet these two tables, WEATHER and LOCATION, have a column in common: City. You can therefore put the information from the two tables together by joining them. You merely use the **where** clause to tell Oracle what the two tables have in common:

```
select WEATHER.City, Condition, Temperature, Latitude,
       NorthSouth, Longitude, EastWest
  from WEATHER, LOCATION
 where WEATHER.City = LOCATION.City;

CITY        CONDITION   TEMPERATURE LATITUDE N LONGITUDE E
----------- ----------- ----------- -------- - --------- -
ATHENS      SUNNY                97    37.58 N     23.43 E
CHICAGO     RAIN                 66    41.53 N     87.38 W
LIMA        RAIN                 45    12.03 S     77.03 W
MANCHESTER  FOG                  66     53.3 N      2.15 W
PARIS       CLOUDY               81    48.52 N       2.2 E
SYDNEY      SUNNY                69    33.52 S    151.13 E
SPARTA      CLOUDY               74    37.05 N     22.27 E
```

Notice that the only rows in this combined table are those where the same city is in *both* tables. The **where** clause is still executing your logic, as it did earlier in the case of the NEWSPAPER table. The logic you gave described the relationship between the two tables. It says, "Select those rows in the WEATHER table and the LOCATION table where the cities are equal." If a city is only in one table, it would have nothing to be equal to in the other table. The notation used in the **select** statement is TABLE.ColumnName—in this case, WEATHER.City.

The **select** clause has chosen those columns from the two tables that you'd like to see displayed; any columns in either table that you did not ask for are simply ignored. If the first line had simply said

```
select City, Condition, Temperature, Latitude
```

then Oracle would not have known to which city you were referring. Oracle would tell you that the column name City was ambiguous. The correct wording in the **select** clause is WEATHER.City or LOCATION.City. In this example, it won't make a bit of difference which of these alternatives is used, but you will encounter cases where the choice of identically named columns from two or more tables will contain very different data.

The **where** clause also requires the names of the tables to accompany the identical column name by which the tables are combined: "where weather dot city equals location dot city" (that is, where the City column in the WEATHER table equals the City column in the LOCATION table).

Consider that the combination of the two tables looks like a single table with seven columns and seven rows. Everything that you excluded is gone. There is no Humidity column here, even though it is a part of the WEATHER table. There is no Country column here, even though it is a part of the LOCATION table. And of the 14 cities in the LOCATION table, only those that are in the WEATHER table are included in this table. Your **where** clause didn't allow the others to be selected.

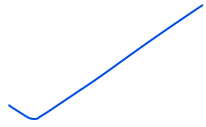
A table that is built from columns in one or more tables is sometimes called a *projection table,* or a *result table.*

Creating a View

There is even more here than meets the eye. Not only does this look like a new table, but you can give it a name and treat it like one. This is called "creating a view." A *view* provides a way of hiding the logic that created the joined table just displayed. It works this way:

```
create view INVASION AS
select WEATHER.City, Condition, Temperature, Latitude,
       NorthSouth, Longitude, EastWest
  from WEATHER, LOCATION
 where WEATHER.City = LOCATION.City;

View created.
```



Now you can act as if INVASION were a real table with its own rows and columns. You can even ask Oracle to describe it to you:

```
describe INVASION

Name                           Null?    Type
------------------------------ -------- ------------
CITY                                    VARCHAR2(11)
CONDITION                               VARCHAR2(9)
TEMPERATURE                             NUMBER
LATITUDE                                NUMBER
NORTHSOUTH                              CHAR(1)
LONGITUDE                               NUMBER
EASTWEST                                CHAR(1)
```

You can query it, too (note that you will not have to specify which table the City columns were from, because that logic is hidden inside the view):

```
select City, Condition, Temperature, Latitude, NorthSouth,
       Longitude, EastWest
  from INVASION;

CITY        CONDITION   TEMPERATURE LATITUDE N LONGITUDE E
----------- ----------- ----------- -------- - --------- -
ATHENS      SUNNY                97    37.58 N     23.43 E
CHICAGO     RAIN                 66    41.53 N     87.38 W
LIMA        RAIN                 45    12.03 S     77.03 W
MANCHESTER  FOG                  66     53.3 N      2.15 W
PARIS       CLOUDY               81    48.52 N       2.2 E
SYDNEY      SUNNY                69    33.52 S    151.13 E
SPARTA      CLOUDY               74    37.05 N     22.27 E
```

There will be some Oracle functions you won't be able to use on a view that you can use on a plain table, but they are few and mostly involve modifying rows and indexing tables, which will be discussed in later chapters. For the most part, a view behaves and can be manipulated just like any other table.

NOTE
Views do not contain any data. Tables contain data. Although you can create "materialized views" that contain data, they are truly tables, not views.

Suppose now you realize that you don't really need information about Chicago or other cities outside of Greece, so you change the query. Will the following work?

```
select City, Condition, Temperature, Latitude, NorthSouth,
       Longitude, EastWest
  from INVASION
 where Country = 'GREECE';
```

SQL*Plus passes back this message from Oracle:

```
where Country = 'GREECE'
      *
ERROR at line 4:
ORA-00904: "COUNTRY": invalid identifier
```

Why? Because even though Country is a real column in one of the tables behind the view called INVASION, it was not in the **select** clause when the view was created. It is as if it does not exist. So, you must go back to the **create view** statement and include only the country of Greece there:

```
create or replace view INVASION as
select WEATHER.City, Condition, Temperature, Latitude,
       NorthSouth, Longitude, EastWest
  from WEATHER, LOCATION
 where WEATHER.City = LOCATION.City
   and Country = 'GREECE';

View created.
```

Using the **create or replace view** command allows you to create a new version of a view without first dropping the old one. This command will make it easier to administer users' privileges to access the view, as will be described in Chapter 18.

The logic of the **where** clause has now been expanded to include both joining two tables, and a single-value test on a column in one of those tables. Now, query Oracle. You'll get this response:

```
select City, Condition, Temperature, Latitude, NorthSouth,
       Longitude, EastWest
  from INVASION;

CITY        CONDITION   TEMPERATURE LATITUDE N LONGITUDE E
----------- ----------- ----------- -------- - --------- -
ATHENS      SUNNY                97    37.58 N     23.43 E
SPARTA      CLOUDY               74    37.05 N     22.27 E
```

This allows you to warn the Athenians that the Spartans are likely to appear from the southwest but will be overheated and tired from their march. With a little trigonometry, you could even make Oracle calculate how far they will have marched.

Expanding the View

The power of views to hide or even modify data can be used for a variety of useful purposes. Very complex reports can be built up by the creation of a series of simple views, and specific individuals or groups can be restricted to seeing only certain pieces of the whole table.

In fact, any qualifications you can put into a query can become part of a view. You could, for instance, let supervisors looking at a payroll table see only their own salaries and those of the people working for them, or you could restrict operating divisions in a company to seeing only their own financial results, even though the table actually contains results for all divisions. Most importantly, views are *not* snapshots of the data at a certain point in the past. They are dynamic and always reflect the data in the underlying tables. The instant data in a table is changed, any views created with that table change as well.

For example, you may create a view that restricts values based on column values. As shown here, a query that restricts the LOCATION table on the Country column could be used to limit the rows that are visible via the view:

```
create or replace view PERU_LOCATIONS as
select * from LOCATION
 where Country = 'PERU';
```

A user querying PERU_LOCATIONS would not be able to see any rows from any country other than Peru.

The queries used to define views may also reference *pseudo-columns.* A pseudo-column is a "column" that returns a value when it is selected, but is not an actual column in a table. The User pseudo-column, when selected, will always return the Oracle username that executed the query. So, if a column in the table contains usernames, those values can be compared against the User pseudo-column to restrict its rows, as shown in the following listing. In this example, the NAME table is queried. If the value of its Name column is the same as the name of the user entering the query, then rows will be returned.

```
create or replace view RESTRICTED_NAMES as
select * from NAME
 where Name = User;
```

This type of view is very useful when users require access to selected rows in a table. It prevents them from seeing any rows that do not match their Oracle username.

Views are powerful tools. There will be more to come on the subject of views in Chapter 17.

The **where** clause can be used to join two tables based on a common column. The resulting set of data can be turned into a view (with its own name), which can be treated as if it were a regular table itself. The power of a view is in its ability to limit or change the way data is seen by a user, even though the underlying tables themselves are not affected.

CHAPTER 18

Basic Oracle Security

nformation is vital to success, but when damaged or in the wrong hands, it can threaten success. Oracle provides extensive security features to safeguard your information from both unauthorized viewing and intentional or inadvertent damage. This security is provided by granting or revoking privileges on a person-by-person and privilege-by-privilege basis, and is in addition to (and independent of) any security your computer system already has. Oracle uses the **create user**, **create role**, and **grant** commands to control data access.

In this chapter, you will see the basic security features available in Oracle. See Chapter 19 for coverage of advanced Oracle security features such as virtual private databases.

Users, Roles, and Privileges

Every Oracle user has a name and password and owns any tables, views, and other resources that he or she creates. An Oracle *role* is a set of *privileges* (or the type of access that each user needs, depending on his or her status and responsibilities). You can grant or bestow specific privileges to roles and then assign roles to the appropriate users. A user can also grant privileges directly to other users.

Database system privileges let you execute specific sets of commands. The CREATE TABLE privilege, for example, lets you create tables. The privilege GRANT ANY PRIVILEGE allows you to grant any system privilege.

Database object privileges give you the ability to perform some operation on various objects. The DELETE privilege, for example, lets you delete rows from tables and views. The SELECT privilege allows you to query with a **select** from tables, views, sequences, and snapshots (materialized views).

See "Privilege" in the Alphabetical Reference at the end of this book for a complete list of system and object privileges.

Creating a User

The Oracle system comes with many users already created, including SYSTEM and SYS. The SYS user owns the core internal tables Oracle uses to manage the database; SYSTEM owns additional tables and views. You can log onto the SYSTEM user to create other users because SYSTEM has that privilege.

When installing Oracle, you (or a system administrator) first create a user for yourself. This is the simplest format for the **create user** command:

```
create user user identified
    {by password | externally | globally as 'extnm'};
```

Many other account characteristics can be set via this command; see the **create user** command in the Alphabetical Reference for details.

To connect your computer system's userid and password into Oracle's security so that only one logon is required for host-based users, use **externally** instead of giving a password. A system administrator (who has a great many privileges) may want the extra security of having a separate password. Let's call the system administrator Dora in the following examples:

```
create user Dora identified by avocado;
```

Dora's account now exists and is secured by a password.

You also can set up the user with specific tablespaces and *quotas* (limits) for space and resource usage. See **create user** in the Alphabetical Reference and Chapters 17 and 20 for a discussion of tablespaces and resources.

To change a password, use the **alter user** command:

```
alter user Dora identified by psyche;
```

Now Dora has the password "psyche" instead of "avocado." However, Dora cannot log into her account until she first has the CREATE SESSION system privilege:

```
grant CREATE SESSION to Dora;
```

You'll see additional examples of system privilege grants later in this chapter.

Password Management

Passwords can expire, and accounts may be locked due to repeated failed attempts to connect. When you change your password, a password history may be maintained to prevent reuse of previous passwords.

The expiration characteristics of your account's password are determined by the profile assigned to your account. Profiles, which are created by the **create profile** command, are managed by the DBA (database administrator, discussed later in the chapter). See the CREATE PROFILE entry in the Alphabetical Reference for full details on the **create profile** command.

Relative to passwords and account access, profiles can enforce the following:

- The "lifetime" of your password, which determines how frequently you must change it
- The grace period following your password's "expiration date" during which you can change the password
- The number of consecutive failed connect attempts allowed before the account is automatically "locked"
- The number of days the account will remain locked
- The number of days that must pass before you can reuse a password
- The number of password changes that must take place before you can reuse a password

Additional password management features allow the minimum length and complexity of passwords to be enforced.

In addition to the **alter user** command, you can use the **password** command in SQL*Plus to change your password. If you use the **password** command, your new password will not be displayed on the screen as you type. Database administrators can change any user's password via the **password** command; other users can change only their own password.

When you enter the **password** command, you will be prompted for the old and new passwords, as shown in the following listing:

```
connect dora/psyche
password
```

```
Changing password for dora
Old password:
New password:
Retype new password:
```

When the password has been successfully changed, you will receive the feedback:

```
Password changed
```

You can set another user's password from a DBA account. Simply append the username to the **password** command. You will not be asked for the old password:

```
password dora
Changing password for dora
New password:
Retype new password:
```

Enforcing Password Expiration

You can use profiles to manage the expiration, reuse, and complexity of passwords. You can limit the lifetime of a password, and lock an account whose password is too old. You can also force a password to be at least moderately complex, and lock any account that has repeated failed login attempts.

For example, if you set the FAILED_LOGIN_ATTEMPTS resource of the user's profile to 5, then four consecutive failed login attempts will be allowed for the account; the fifth will cause the account to be locked.

NOTE

If the correct password is supplied on the fifth attempt, the "failed login attempt count" is reset to 0, allowing for five more consecutive unsuccessful login attempts before the account is locked.

In the following listing, the LIMITED_PROFILE profile is created for use by the user JANE:

```
create profile LIMITED_PROFILE limit
FAILED_LOGIN_ATTEMPTS 5;

create user JANE identified by EYRE
profile LIMITED_PROFILE;

grant CREATE SESSION to JANE;
```

If there are five consecutive failed connection attempts to the JANE account, the account will be automatically locked by Oracle. When you then use the correct password for the JANE account, you will receive an error.

```
connect jane/eyre
ERROR:
ORA-28000: the account is locked
```

To unlock the account, use the **account unlock** clause of the **alter user** command (from a DBA account), as shown in the following listing:

```
alter user JANE account unlock;
```

Following the unlocking of the account, connections to the JANE account will once again be allowed. You can manually lock an account via the **account lock** clause of the **alter user** command.

```
alter user JANE account lock;
```

NOTE
*You can specify **account lock** as part of the **create user** command.*

If an account becomes locked due to repeated connection failures, it will automatically become unlocked when its profile's PASSWORD_LOCK_TIME value is exceeded. For example, if PASSWORD_LOCK_TIME is set to 1, the JANE account in the previous example would be locked for one day, after which the account would be unlocked again.

You can establish a maximum lifetime for a password via the PASSWORD_LIFE_TIME resource within profiles. For example, you could force users of the LIMITED_PROFILE profile to change their passwords every 30 days:

```
alter profile LIMITED_PROFILE limit
PASSWORD_LIFE_TIME 30;
```

In this example, the **alter profile** command is used to modify the LIMITED_PROFILE profile. The PASSWORD_LIFE_TIME value is set to 30, so each account that uses that profile will have its password expire after 30 days. If your password has expired, you must change it the next time you log in, unless the profile has a specified grace period for expired passwords. The grace period parameter is called PASSWORD_GRACE_TIME. If the password is not changed within the grace period, the account expires.

NOTE
If you are going to use the PASSWORD_LIFE_TIME parameter, you need to give the users a way to change their passwords easily.

An "expired" account is different from a "locked" account. A locked account, as discussed earlier in this section, may be automatically unlocked by the passage of time. An expired account, however, requires manual intervention by the DBA to be reenabled.

NOTE
If you use the password expiration features, make sure the accounts that own your applications have different profile settings; otherwise, the accounts may become locked and the application may become unusable.

To reenable an expired account, execute the **alter user** command. In this example, the DBA manually expires JANE's password:

```
alter user jane password expire;

User altered.
```

Next, JANE attempts to connect to her account. When she provides her password, she is immediately prompted for a new password for the account.

```
connect jane/eyre
ERROR:
ORA-28001: the password has expired

Changing password for jane
New password:
Retype new password:
Password changed
Connected.
```

You can also force users to change their passwords when they first access their accounts, via the **password expire** clause of the **create user** command. The **create user** command does not, however, allow you to set an expiration date for the new password set by the user; to do that, you must use the PASSWORD_LIFE_TIME profile parameter shown in the previous examples.

To see the password expiration date of any account, query the Expiry_Date column of the DBA_USERS data dictionary view. Users who want to see the password expiration date for their accounts can query the Expiry_Date column of the USER_USERS data dictionary view (via either SQL*Plus or a client-based query tool).

Enforcing Password Reuse Limitations

To prevent a password from being reused, you can use one of two profile parameters: PASSWORD_REUSE_MAX or PASSWORD_REUSE_TIME. These two parameters are mutually exclusive; if you set a value for one of them, the other must be set to UNLIMITED.

The PASSWORD_REUSE_TIME parameter specifies the number of days that must pass before a password can be reused. For example, if you set PASSWORD_REUSE_TIME to 60, you cannot reuse the same password within 60 days.

The PASSWORD_REUSE_MAX parameter specifies the number of password changes that must occur before a password can be reused. If you attempt to reuse the password before the limit is reached, Oracle will reject your password change.

For example, you can set PASSWORD_REUSE_MAX for the LIMITED_PROFILE profile created earlier in this chapter:

```
alter profile LIMITED_PROFILE limit
PASSWORD_REUSE_MAX 3
PASSWORD_REUSE_TIME UNLIMITED;
```

If the user JANE now attempts to reuse a recent password, the password change attempt will fail. For example, suppose she changes her password, as in the following line:

```
alter user JANE identified by austen;
```

And then she changes it again:

```
alter user JANE identified by whitley;
```

During her next password change, she attempts to reuse a recent password, and the attempt fails:

```
alter user jane identified by austen;
alter user jane identified by austen
*
ERROR at line 1:
ORA-28007: the password cannot be reused
```

She cannot reuse any of her recent passwords; she will need to come up with a new password.

Standard Roles

At the beginning of this chapter, we created a user named Dora. Now that Dora has an account, what can she do in Oracle? At this point, nothing—Dora has no system privileges other than CREATE SESSION. In many cases, application users receive privileges via roles. You can group system privileges and object accesses into roles specific to your application users' needs. You can create your own roles for application access, and you can use Oracle's default roles for some system access requirements. The most important standard roles created during database creation are

CONNECT, RESOURCE, and DBA	Provided for compatibility with previous versions of Oracle database software.
DELETE_CATALOG_ROLE, EXECUTE_CATALOG_ROLE, SELECT_CATALOG_ROLE	These roles are provided for accessing data dictionary views and packages.
EXP_FULL_DATABASE, IMP_FULL_DATABASE	These roles are provided for convenience in using the Import and Export utilities.
AQ_USER_ROLE, AQ_ADMINISTRATOR_ROLE	These roles are needed for Oracle Advanced Queuing.
SNMPAGENT	This role is used by the Enterprise Manager Intelligent Agent.
RECOVERY_CATALOG_OWNER	This role is required for the creation of a recovery catalog schema owner.
SCHEDULER_ADMIN	This role allows the grantee to execute the DBMS_SCHEDULER package's procedures; should be restricted to DBAs.

CONNECT, RESOURCE, and DBA are provided for backward compatibility and should no longer be used. CONNECT gives users the ability to log in and perform basic functions. Users with CONNECT may create tables, views, sequences, clusters, synonyms, and database links. Users with RESOURCE can create their own tables, sequences, procedures, triggers, datatypes,

operators, indextypes, indexes, and clusters. Users with the RESOURCE role also receive the UNLIMITED TABLESPACE system privilege, allowing them to bypass quotas on all tablespaces. Users with the DBA role can perform database administration functions, including creating and altering users, tablespaces, and objects.

In place of CONNECT, RESOURCE, and DBA, you should create your own roles that have privileges to execute specific system privileges. In the following sections you will see how to grant privileges to users and roles.

Format for the grant Command

Here is the format for the **grant** command for system privileges (see the Alphabetical Reference for the full syntax diagram):

```
grant {system privilege | role | all [privileges] }
      [, {system privilege | role | all [privileges] }. . .]
      to {user | role} [, {user | role}]. . .
      [identified by password ]
      [with admin option]
```

You can grant any system privilege or role to another user, to another role, or to **public**. The **with admin option** clause permits the grantee to bestow the privilege or role on other users or roles. The **all** clause grants the user or role all privileges except the SELECT ANY DICTIONARY system privilege.

The grantor can revoke a role from a user as well.

Revoking Privileges

Privileges granted can be taken away. The **revoke** command is similar to the **grant** command:

```
revoke {system privilege | role | all [privileges] }
       [, {system privilege | role | all [privileges] }. . .]
       from {user | role} [, {user | role}]. . .
```

An individual with the DBA role can revoke CONNECT, RESOURCE, DBA, or any other privilege or role from anyone, including another DBA. This, of course, is dangerous, and is why DBA privileges should be given neither lightly nor to more than a tiny minority who really need them.

NOTE

Revoking everything from a given user does not eliminate that user from Oracle, nor does it destroy any tables that user had created; it simply prohibits that user's access to them. Other users with access to the tables will still have exactly the same access they've always had.

To remove a user and all the resources owned by that user, use the **drop user** command, like this:

```
drop user username [cascade];
```

The **cascade** option drops the user along with all the objects owned by the user, including referential integrity constraints. The **cascade** option invalidates views, synonyms, stored procedures, functions, or packages that refer to objects in the dropped user's schema. If you don't use the **cascade** option and there are still objects owned by the user, Oracle does not drop the user and instead returns an error message.

What Users Can Grant

A user can grant privileges on any object he or she owns. The database administrator can grant any system privilege.

Suppose that user Dora owns the COMFORT table and is a database administrator. Create two new users, Bob and Judy, with these privileges:

```
create user Judy identified by sarah;

User created.

grant CREATE SESSION to Judy;

Grant succeeded.

create user Bob identified by carolyn;

User created.

grant CREATE SESSION, CREATE TABLE, CREATE VIEW,
CREATE SYNONYM to bob;

Grant succeeded.

alter user bob
default tablespace users
quota 5m on users;

User altered.
```

This sequence of commands gives both Judy and Bob the ability to connect to Oracle, and gives Bob some extra capabilities. But can either do anything with Dora's tables? Not without explicit access.

To give others access to your tables, use a second form of the **grant** command:

```
grant { object privilege | all [privileges] }
    [(column [, column] . . .)]
    [, { object privilege | all [privileges] }
    [(column [, column] . . .)] ] . . .
    on object to {user | role}
    [with grant option]
    [with hierarchy option];
```

The privileges a user can grant include these:

- On the user's tables, views, and materialized views:

  `FLASHBACK`
  `INSERT`
  `UPDATE` (all or specific columns)
  `DELETE`
  `SELECT`

  The INSERT, UPDATE, and DELETE privileges can only be granted on materialized views if they are updateable. See Chapter 24 for details on the creation of materialized views.

- On tables, you can also grant:

  `ALTER` (table—all or specific columns)
  `DEBUG`
  `REFERENCES`
  `INDEX` (columns in a table)
  `ON COMMIT REFRESH`
  `QUERY REWRITE`
  `ALL` (of the items previously listed)

- On procedures, functions, packages, abstract datatypes, libraries, indextypes, and operators:

  `EXECUTE`
  `DEBUG`

- On sequences:

  `SELECT`
  `ALTER`

- On directories (for BFILE LOB datatypes and external tables):

  `READ`
  `WRITE`

- On abstract datatypes and views:

  `UNDER` (for the ability to create subviews under a view, or subtypes under a type)

When you execute another user's procedure or function, it is typically executed using the privileges of its owner. This means you don't need explicit access to the data the procedure or function uses; you see only the result of the execution, not the underlying data. You can also create stored procedures that execute under the privileges of the invoking user instead of the procedure owner; see Chapter 31 for details.

Dora gives Bob SELECT access to the COMFORT table:

```
grant select on COMFORT to Bob;

Grant succeeded.
```

The **with grant option** clause of the **grant** command allows the recipient of that grant to pass along the privileges he or she has received to other users. If the user Dora grants privileges on her

tables to the user Bob using **with grant option**, Bob can make grants on Dora's tables to other users (Bob can only pass along those privileges—such as SELECT—that he has been granted). If you intend to create views based on another user's tables and grant access to those views to other users, you first must be granted access **with grant option** to the base tables.

Moving to Another User with connect

To test the success of her grant, Dora connects to Bob's username with the **connect** command. This command may be used via one of the following methods:

- By entering both the username and password on the same line as the command
- By entering the command alone and then responding to prompts
- By entering the command and username and responding to the prompt for the password

The latter two methods suppress the display of the password and are therefore inherently more secure. The following listing shows a sample connection to the database:

```
connect Bob/carolyn

Connected.
```

Once connected, Dora selects from the table to which Bob has been given SELECT access.

NOTE
Unless a synonym is used, the table name must be preceded by the username of the table's owner. Without this, Oracle will say the table does not exist.

```
select * from Dora.COMFORT;

CITY          SAMPLEDAT  NOON MIDNIGHT PRECIPITATION
------------- --------- ----- -------- -------------
SAN FRANCISCO 21-MAR-03  62.5     42.3            .5
SAN FRANCISCO 22-JUN-03  51.1     71.9            .1
SAN FRANCISCO 23-SEP-03           61.5            .1
SAN FRANCISCO 22-DEC-03  52.6     39.8           2.3
KEENE         21-MAR-03  39.9     -1.2           4.4
KEENE         22-JUN-03  85.1     66.7           1.3
KEENE         23-SEP-03  99.8     82.6
KEENE         22-DEC-03  -7.2     -1.2           3.9
```

A view named COMFORT is created, which is simply a straight **select** from the table Dora.COMFORT:

```
create view COMFORT as select * from Dora.COMFORT;

View created.
```

Selecting from this view will produce exactly the same results as selecting from Dora.COMFORT:

```
select * from COMFORT;

CITY          SAMPLEDAT  NOON MIDNIGHT PRECIPITATION
------------- --------- ----- -------- -------------
SAN FRANCISCO 21-MAR-03  62.5     42.3            .5
SAN FRANCISCO 22-JUN-03  51.1     71.9            .1
SAN FRANCISCO 23-SEP-03           61.5            .1
SAN FRANCISCO 22-DEC-03  52.6     39.8           2.3
KEENE         21-MAR-03  39.9     -1.2           4.4
KEENE         22-JUN-03  85.1     66.7           1.3
KEENE         23-SEP-03  99.8     82.6
KEENE         22-DEC-03  -7.2     -1.2           3.9
```

Now Dora returns to her own username and creates a view that selects only a part of the COMFORT table:

```
connect Dora/psyche
Connected.

create view SOMECOMFORT as
select * from COMFORT
 where City = 'KEENE';

View created.
```

She then grants both SELECT and UPDATE privileges to Bob *on this view,* and revokes all privileges from Bob (via the **revoke** command) for the whole COMFORT table:

```
grant select, update on SOMECOMFORT to Bob;

Grant succeeded.

revoke all on COMFORT from Bob;

Revoke succeeded.
```

Dora then reconnects to Bob's username to test the effects of this change:

```
connect Bob/carolyn
Connected.

select * from COMFORT;
              *
ERROR at line 1:
ORA-04063: view "BOB.COMFORT" has errors
```

Attempting to select from his COMFORT view fails because the underlying table named in Bob's view was the Dora.COMFORT table. Not surprisingly, attempting to select from Dora.COMFORT

would produce exactly the same message. Next, an attempt to select from Dora.SOMECOMFORT is made:

```
select * from Dora.SOMECOMFORT;

CITY          SAMPLEDAT  NOON MIDNIGHT PRECIPITATION
------------- --------- ----- -------- -------------
KEENE         21-MAR-03  39.9     -1.2           4.4
KEENE         22-JUN-03  85.1     66.7           1.3
KEENE         23-SEP-03  99.8     82.6
KEENE         22-DEC-03  -7.2     -1.2           3.9
```

This works perfectly well, even though direct access to the COMFORT table had been revoked, because Dora gave Bob access to a portion of COMFORT through the SOMECOMFORT view. It is just that portion of the table related to KEENE.

This shows a powerful security feature of Oracle: You can create a view using virtually any restrictions you like or any computations in the columns, and then give access to the view, rather than to the underlying tables, to other users. They will see only the information the view presents. This can even be extended to be user specific. The "Security by User" section later in this chapter gives complete details on this feature.

Now, the view LITTLECOMFORT is created under Bob's username, on top of the view SOMECOMFORT:

```
create view LITTLECOMFORT as select * from Dora.SOMECOMFORT;

View created.
```

And the row for September 23, 2003, is updated:

```
update LITTLECOMFORT set Noon = 88
 where SampleDate = '23-SEP-03';

1 row updated.
```

When the view LITTLECOMFORT is queried, it shows the effect of the **update**:

```
select * from LITTLECOMFORT;

CITY          SAMPLEDAT  NOON MIDNIGHT PRECIPITATION
------------- --------- ----- -------- -------------
KEENE         21-MAR-03  39.9     -1.2           4.4
KEENE         22-JUN-03  85.1     66.7           1.3
KEENE         23-SEP-03    88     82.6
KEENE         22-DEC-03  -7.2     -1.2           3.9
```

A query of Dora.SOMECOMFORT would show the same results, as would a query of COMFORT itself if Dora made it on her own username. The **update** was successful against the underlying table, even though it went through two views (LITTLECOMFORT and SOMECOMFORT) to reach it.

NOTE
You need to grant users SELECT access to any table in which they can update or delete records. This is in keeping with the evolving ANSI standard and reflects the fact that a user who only has the UPDATE or DELETE privilege on a table could use the database's feedback comments to discover information about the underlying data.

create synonym

An alternative method to creating a view that includes an entire table or view from another user is to create a synonym:

```
create synonym BOBCOMFORT for Dora.SOMECOMFORT;
```

This synonym can be treated exactly like a view. See the **create synonym** command in the Alphabetical Reference.

Using Ungranted Privileges

Let's say an attempt is made to delete the row just updated:

```
delete from LITTLECOMFORT where SampleDate = '23-SEP-03';
              *
ERROR at line 1:
ORA-01031: insufficient privileges
```

Bob has not been given DELETE privileges by Dora, so the attempt fails.

Passing Privileges

Bob can grant authority for other users to access his tables, but he cannot bestow on other users access to tables that don't belong to him. Here, he attempts to give INSERT authority to Judy:

```
grant insert on Dora.SOMECOMFORT to Judy;
      *
ERROR at line 1:
ORA-01031: insufficient privileges
```

Because Bob does not have this authority, he fails to give it to Judy. Next, Bob tries to pass on a privilege he does have, SELECT:

```
grant select on Dora.SOMECOMFORT to Judy;
      *
ERROR at line 1:
ORA-01031: insufficient privileges
```

He cannot grant this privilege, either, because the view SOMECOMFORT does not belong to him. If he had been granted access to SOMECOMFORT **with grant option**, then the preceding **grant** command would have succeeded. The view LITTLECOMFORT does belong to him, though, so he can try to pass authority to that view on to Judy:

```
grant select on LITTLECOMFORT to Judy;

ERROR at line 1:
ORA-01720: grant option does not exist for 'DORA.SOMECOMFORT'
```

Because the LITTLECOMFORT view relies on one of Dora's views, and Bob was not granted SELECT **with grant option** on that view, Bob's grant fails.

In addition, a new table, owned by Bob, is created and loaded with the current information from his view LITTLECOMFORT:

```
create table NOCOMFORT as
select * from LITTLECOMFORT;

Table created.
```

The SELECT privilege on the NOCOMFORT table is granted to Judy:

```
grant select on NOCOMFORT to Judy;

Grant succeeded.
```

Queries are made from Judy's account against the table for which Bob granted the SELECT privilege to Judy:

```
connect Judy/sarah
select * from Bob.NOCOMFORT;

CITY          SAMPLEDAT  NOON MIDNIGHT PRECIPITATION
------------- --------- ----- -------- -------------
KEENE         21-MAR-03  39.9     -1.2           4.4
KEENE         22-JUN-03  85.1     66.7           1.3
KEENE         23-SEP-03  88       82.6
KEENE         22-DEC-03  -7.2     -1.2           3.9
```

This grant was successful. The NOCOMFORT table was created, and is owned, by Bob's username. He was able to successfully give access to this table.

If Dora wants Bob to be able to pass on his privileges to others, she can add another clause to the **grant** statement:

```
connect Dora/psyche
grant select, update on SOMECOMFORT to Bob with grant option;

Grant succeeded.
```

The **with grant option** clause enables Bob to pass on access to SOMECOMFORT to Judy through his LITTLECOMFORT view. Note that attempts to access a table to which a user does not have the SELECT privilege always result in the following message:

```
ERROR at line 1:
ORA-00942: table or view does not exist
```

This message appears (rather than a message about not having access privilege) so that a person without permission to query a table doesn't know that it even exists.

Creating a Role

In addition to the default roles shown earlier in this chapter, you can create your own roles within Oracle. The roles you create can comprise table or system privileges or a combination of both. In the following sections, you will see how to create and administer roles.

To create a role, you need to have the CREATE ROLE system privilege. The syntax for role creation is shown in the following listing:

```
create role role_name
[not identified
|identified {by password | using [schema.]package
|externally | globally }];
```

When a role is first created, it has no privileges associated with it. Password options for roles are discussed in "Adding a Password to a Role," later in this chapter.

Two sample **create role** commands are shown in the following example:

```
create role CLERK;
create role MANAGER;
```

The first command creates a role called CLERK, which will be used in the examples in the following sections of this chapter. The second command creates a role called MANAGER, which will also be featured in those examples.

Granting Privileges to a Role

Once a role has been created, you may grant privileges to it. The syntax for the **grant** command is the same for roles as it is for users. When granting privileges to roles, you use the role name in the **to** clause of the **grant** command, as shown in the following listing:

```
grant select on COMFORT to CLERK;
```

As shown in this example, the role name takes the place of the username in the **grant** command. The privilege to select from the COMFORT table will now be available to any user of the CLERK role.

If you are a DBA, or have been granted the GRANT ANY PRIVILEGE system role, you may grant system privileges—such as CREATE SESSION, CREATE SYNONYM, and CREATE VIEW—to roles. These privileges will then be available to any user of your role.

The ability to log into the database is given via the CREATE SESSION system privilege. In the following example, this privilege is granted to the CLERK role. This privilege is also granted to the MANAGER role, along with the CREATE VIEW system privilege.

```
grant CREATE SESSION to CLERK;
grant CREATE SESSION, CREATE VIEW to MANAGER;
```

Granting a Role to Another Role

Roles can be granted to other roles. You can do this via the **grant** command, as shown in the following example:

```
grant CLERK to MANAGER;
```

NOTE
You can't have circular grants, so you cannot now grant MANAGER to CLERK.

In this example, the CLERK role is granted to the MANAGER role. Even though you have not directly granted any table privileges to the MANAGER role, it will now inherit any privileges that have been granted to the CLERK role. Organizing roles in this way is a common design for hierarchical organizations.

When granting a role to another role (or to a user, as in the following section), you may grant the role using the **with admin option** clause, as shown in the following listing:

```
grant CLERK to MANAGER with admin option;
```

If the **with admin option** clause is used, the grantee has the authority to grant the role to other users or roles. The grantee can also alter or drop the role.

Granting a Role to Users

Roles can be granted to users. When granted to users, roles can be thought of as named sets of privileges. Instead of granting each privilege to each user, you grant the privileges to the role and then grant the role to each user. This greatly simplifies the administrative tasks involved in the management of privileges.

NOTE
Privileges that are granted to users via roles cannot be used as the basis for views, procedures, functions, packages, or foreign keys. When creating these types of database objects, you must rely on direct grants of the necessary privileges.

You can grant a role to a user via the **grant** command, as shown in the following example:

```
grant CLERK to Bob;
```

The user Bob in this example will have all the privileges that were granted to the CLERK role (CREATE SESSION and SELECT privileges on COMFORT).

When granting a role to a user, you may grant the role using the **with admin option** clause, as shown in the following listing:

```
grant MANAGER to Dora with admin option;
```

Dora now has the authority to grant the MANAGER role to other users or roles, or to alter or drop the role.

Adding a Password to a Role

You can use the **alter role** command for only one purpose: to change the authority needed to enable it. By default, roles do not have passwords associated with them. To enable security for a role, use the **identified** keyword in the **alter role** command. There are two ways to implement this security.

First, you can use the **identified by** clause of the **alter role** command to specify a password, as shown in the following listing:

```
alter role MANAGER identified by cygnusxi;
```

Anytime a user tries to activate that role, the password will be required. If, however, that role is set up as a default role for the user, no password will be required for that role when the user logs in. See the upcoming "Enabling and Disabling Roles" section of this chapter for more details on these topics.

Roles can be tied to operating system privileges as well. If this capability is available on your operating system, you use the **identified externally** clause of the **alter role** command. When the role is enabled, Oracle will check the operating system to verify your access. Altering a role to use this security feature is shown in the following example:

```
alter role MANAGER identified externally;
```

In most UNIX systems, the verification process uses the /etc/group file. To use this file for any operating system, the OS_ROLES database startup parameter in the init.ora file must be set to TRUE.

The following example of this verification process is for a database instance called "Local" on a UNIX system. The server's /etc/group file may contain the following entry:

```
ora_local_manager_d:NONE:1:dora
```

This entry grants the MANAGER role to the account named Dora. The **_d** suffix indicates that this role is to be granted by default when Dora logs in. An **_a** suffix would indicate that this role is to be enabled **with admin option**. If this role were also the user's default role, the suffix would be **_ad**. If more than one user were granted this role, the additional usernames would be appended to the /etc/group entry, as shown in the following listing:

```
ora_local_manager_d:NONE:1:dora,judy
```

If you use this option, roles in the database will be enabled via the operating system.

Removing a Password from a Role

To remove a password from a role, use the **not identified** clause of the **alter role** command, as shown in the following listing. By default, roles do not have passwords.

```
alter role MANAGER not identified;
```

Enabling and Disabling Roles

When a user's account is altered, a list of default roles for that user can be created via the **default role** clause of the **alter user** command. The default action of this command sets all of a user's roles as default roles, enabling all of them every time the user logs in.

NOTE
The maximum number of roles a user can have enabled at any time is set via the MAX_ENABLED_ROLES database initialization parameter.

The syntax for this portion of the **alter user** command is as follows:

```
alter user username
default role {[role1, role2]
[all | all except role1, role2][NONE]};
```

As shown by this syntax, a user can be altered to have, by default, specific roles enabled, all roles enabled, all except specific roles enabled, or no roles enabled. For example, the following **alter user** command will enable the CLERK role whenever Bob logs in:

```
alter user Bob
default role CLERK;
```

To enable a nondefault role, use the **set role** command, as shown in this example:

```
set role CLERK;
```

To see what roles are enabled within your current session, select from the SESSION_ROLES data dictionary view. Query SESSION_PRIVS to see the currently enabled system privileges.

You may also use the **all** and **all except** clauses that were available in the **alter user** command:

```
set role all;
set role all except CLERK;
```

If a role has a password associated with it, that password must be specified via an **identified by** clause:

```
set role MANAGER identified by cygnusxi;
```

To disable a role in your session, use the **set role none** command, as shown in the following listing. This will disable all roles in your current session. Once all the roles have been disabled, reenable the ones you want.

```
set role none;
```

Because you may find it necessary to execute a **set role none** command from time to time, you may want to grant the CREATE SESSION privilege to users directly rather than via roles.

Revoking Privileges from a Role

To revoke a privilege from a role, use the **revoke** command, described earlier in this chapter. Specify the privilege, object name (if it is an object privilege), and role name, as shown in the following example:

```
revoke SELECT on COMFORT from CLERK;
```

Users of the CLERK role will then be unable to query the COMFORT table.

Dropping a Role

To drop a role, use the **drop role** command, as shown in the following example:

```
drop role MANAGER;
drop role CLERK;
```

The roles you specify, and their associated privileges, will be removed from the database entirely. **Grant**s and **revoke**s of system and object privileges take effect immediately. **Grant**s and **revoke**s of roles are only observed when a current user issues a **set role** command or a new user session is started.

Granting UPDATE to Specific Columns

You may want to grant users the SELECT privilege to more columns than you want to grant them the UPDATE privilege. Because SELECT columns can be restricted through a view, to further restrict the columns that can be updated requires a special form of the user's **grant** command. Here is an example for two columns in the COMFORT table:

```
grant update (Noon, Midnight) on COMFORT to Judy;
```

Revoking Object Privileges

If object privileges can be granted, they can also be taken away. This is similar to the **grant** command:

```
revoke { object privilege | all [privileges]}
       [(column [, column]. . . )]
[, { object privilege | all [privileges]}
       [(column [, column]. . . )]]. . .]
on object
from {user | role} [,{user | role}]
[cascade constraints] [force];
```

revoke all removes any of the privileges listed previously, from SELECT through INDEX; revoking individual privileges will leave intact others that had also been granted. The **with grant option** is revoked along with the privilege to which it was attached; the **revoke** then cascades so users who received their access via the **with grant option** user also have their privileges revoked.

If a user defines referential integrity constraints on the object, Oracle drops these constraints if you revoke privileges on the object using the **cascade constraints** option. The **force** option applies

to user-defined datatypes (see Chapter 33). The **force** option revokes the EXECUTE privilege on user-defined datatype objects with table or type dependencies; all dependent objects will be marked as invalid and the data in dependent tables will become inaccessible until the necessary privilege is regranted.

Security by User

Access to tables can be granted specifically, table by table and view by view, to each user or role. There is, however, an additional technique that will simplify this process in some cases. Recall the BOOKSHELF_CHECKOUT records:

```
select Name, SUBSTR(Title,1,30) from BOOKSHELF_CHECKOUT;

NAME                      SUBSTR(TITLE,1,30)
------------------------- ------------------------------
JED HOPKINS               INNUMERACY
GERHARDT KENTGEN          WONDERFUL LIFE
DORAH TALBOT              EITHER/OR
EMILY TALBOT              ANNE OF GREEN GABLES
PAT LAVAY                 THE SHIPPING NEWS
ROLAND BRANDT             THE SHIPPING NEWS
ROLAND BRANDT             THE DISCOVERERS
ROLAND BRANDT             WEST WITH THE NIGHT
EMILY TALBOT              MIDNIGHT MAGIC
EMILY TALBOT              HARRY POTTER AND THE GOBLET OF
PAT LAVAY                 THE MISMEASURE OF MAN
DORAH TALBOT              POLAR EXPRESS
DORAH TALBOT              GOOD DOG, CARL
GERHARDT KENTGEN          THE MISMEASURE OF MAN
FRED FULLER               JOHN ADAMS
FRED FULLER               TRUMAN
JED HOPKINS               TO KILL A MOCKINGBIRD
DORAH TALBOT              MY LEDGER
GERHARDT KENTGEN          MIDNIGHT MAGIC
```

To enable each borrower to access this table, but restrict the access given to a view of only each borrower's own single row, you could create 19 separate views, each with a different name in the **where** clause, and you could make separate grants to each of these views for each borrower. Alternatively, you could create a view whose **where** clause contains User, the pseudo-column, like this:

```
create or replace view MY_CHECKOUT as
select * from BOOKSHELF_CHECKOUT
 where SUBSTR(Name,1,INSTR(Name,' ')-1) = User;
```

The owner of the BOOKSHELF_CHECKOUT table can create this view and grant SELECT on it to users. A user can then query the BOOKSHELF_CHECKOUT table through the MY_CHECKOUT view. The **where** clause will find his or her username in the Name column (see the **where** clause) and return just that user's records.

In the following example, a user name Jed is created and granted access to the MY_CHECKOUT view. Selecting from that view yields just those books checked out by him. This example assumes that the schema owning the BOOKSHELF_CHECKOUT table is named PRACTICE and is a database administrator:

```
create user jed identified by hop;

User created.

grant CREATE SESSION to jed;

Grant succeeded.

grant select on MY_CHECKOUT to jed;

Grant succeeded.

connect jed/hop

Connected.

select * from Practice.MY_CHECKOUT;

NAME
-------------------------
TITLE
------------------------------------------------------------
CHECKOUTD RETURNEDD
--------- ---------
JED HOPKINS
INNUMERACY
01-JAN-02 22-JAN-02

JED HOPKINS
TO KILL A MOCKINGBIRD
15-FEB-02 01-MAR-02
```

Whenever MY_CHECKOUT is queried, it will return records depending on the pseudo-column User—the user of SQL*Plus at the moment the view is queried.

Granting Access to the Public

Rather than grant access to every worker, the **grant** command can be generalized to the public:

```
grant select on MY_CHECKOUT to public;
```

This gives everyone access, including users created after this **grant** was made. However, each user will still have to access the table using the owner's username as a prefix. To avoid this, a DBA may create a *public synonym* (which creates a name accessible to all users that stands for Practice.MY_CHECKOUT):

```
create public synonym MY_CHECKOUT for Practice.MY_CHECKOUT;
```

From this point forward, anyone can access MY_CHECKOUT without prefixing it with the schema owner (Practice in this case). This approach gives tremendous flexibility for security. Workers could see only their own salaries, for instance, in a table that contains salaries for everyone. If, however, a user creates a table or view with the same name as a public synonym, any future SQL statements by this user will act on this new table or view, and not on the one by the same name to which the public has access.

NOTE
All these accesses—and all attempts at system-level operations, such as logins—may be audited. See AUDIT in the Alphabetical Reference for an introduction to Oracle's security-auditing capabilities.

Granting Limited Resources

When granting resource quotas in an Oracle database, you use the **quota** parameter of the **create user** or **alter user** command, as shown in the following listing. In this example, Bob is granted a quota of 100MB in the USERS tablespace:

```
alter user Bob
quota 100M on USERS;
```

A user's space quota may be set when the user is created, via the **create user** command. If you want to remove limits on a user's space quota, you can grant that user the UNLIMITED TABLESPACE system privilege. See the **create user** and **alter user** commands in the Alphabetical Reference for further details on these commands.

Profiles can also be used to enforce other resource limits, such as the amount of CPU time or idle time a user's requests to Oracle can take. A profile detailing these resource limits is created and then assigned to one or more users. See the **create profile** and **alter user** command entries in the Alphabetical Reference for full details.

CHAPTER 30

Triggers

trigger defines an action the database should take when some database-related event occurs. Triggers may be used to supplement declarative referential integrity, to enforce complex business rules, or to audit changes to data. The code within a trigger, called the *trigger body,* is made up of PL/SQL blocks (see Chapter 29).

The execution of triggers is transparent to the user. Triggers are executed by the database when specific types of data manipulation commands are performed on specific tables. Such commands may include **insert**s, **update**s, and **delete**s. Updates of specific columns may also be used as triggering events. Triggering events may also include DDL commands and database events such as shutdowns and logins.

Because of their flexibility, triggers may supplement referential integrity; they should not be used to replace it. When enforcing the business rules in an application, you should first rely on the declarative referential integrity available with Oracle; use triggers to enforce rules that cannot be coded through referential integrity.

Required System Privileges

To create a trigger on a table, you must be able to alter that table. Therefore, you must either own the table, or have the ALTER privilege for the table, or have the ALTER ANY TABLE system privilege. In addition, you must have the CREATE TRIGGER system privilege; to create triggers in another user's schema you must have the CREATE ANY TRIGGER system privilege. The CREATE TRIGGER system privilege is part of the RESOURCE role provided with Oracle.

NOTE
The RESOURCE role is provided in Oracle Database 10g only for backward compatibility and should not be used regularly.

To *alter* a trigger, you must either own the trigger or have the ALTER ANY TRIGGER system privilege. You may also enable or disable triggers by altering the tables they are based on, which requires that you have either the ALTER privilege for that table or the ALTER ANY TABLE system privilege. For information on altering triggers, see "Enabling and Disabling Triggers," later in this chapter.

To create a trigger on a database-level event, you must have the ADMINISTER DATABASE TRIGGER system privilege.

Required Table Privileges

Triggers may reference tables other than the one that initiated the triggering event. For example, if you use triggers to audit changes to data in the BOOKSHELF table, then you may insert a record into a different table (say, BOOKSHELF_AUDIT) every time a record is changed in BOOKSHELF. To do this, you need to have privileges to insert into BOOKSHELF_AUDIT (to perform the triggered transaction).

NOTE
The privileges needed for triggered transactions cannot come from roles; they must be granted directly to the creator of the trigger.

Types of Triggers

A trigger's type is defined by the type of triggering transaction and by the level at which the trigger is executed. In the following sections, you will see descriptions of these classifications, along with relevant restrictions.

Row-Level Triggers

Row-level triggers execute once for each row affected by a DML statement. For the BOOKSHELF table auditing example described earlier, each row that is changed in the BOOKSHELF table may be processed by the trigger. Row-level triggers are the most common type of trigger; they are often used in data auditing applications. Row-level triggers are also useful for keeping distributed data in sync. Materialized views, which use internal row-level triggers for this purpose, are described in Chapter 24.

Row-level triggers are created using the **for each row** clause in the **create trigger** command. The syntax for triggers is shown in "Trigger Syntax," later in this chapter.

Statement-Level Triggers

Statement-level triggers execute once for each DML statement. For example, if a single INSERT statement inserts 500 rows into the BOOKSHELF table, a statement-level trigger on that table would only be executed once. Statement-level triggers therefore are not often used for data-related activities; they are normally used to enforce additional security measures on the types of actions that may be performed on a table.

Statement-level triggers are the default type of trigger created via the **create trigger** command. The syntax for triggers is shown in "Trigger Syntax," later in this chapter.

BEFORE and AFTER Triggers

Because triggers are executed by events, they may be set to occur immediately before or after those events. Since the events that execute triggers include database DML statements, triggers can be executed immediately before or after **insert**s, **update**s, and **delete**s. For database-level events, additional restrictions apply; you cannot trigger an event to occur before a login or startup takes place.

Within the trigger, you can reference the old and new values involved in the DML statement. The access required for the old and new data may determine which type of trigger you need. "Old" refers to the data as it existed prior to the DML statement; **update**s and **delete**s usually reference old values. "New" values are the data values that the DML statement creates (such as the columns in an inserted record).

If you need to set a column value in an inserted row via your trigger, then you need to use a BEFORE INSERT trigger to access the "new" values. Using an AFTER INSERT trigger would not allow you to set the inserted value, since the row will already have been inserted into the table.

Row-level AFTER triggers are frequently used in auditing applications, since they do not fire until the row has been modified. The row's successful modification implies that it has passed the referential integrity constraints defined for that table.

INSTEAD OF Triggers

You can use INSTEAD OF triggers to tell Oracle what to do *instead of* performing the actions that invoked the trigger. For example, you could use an INSTEAD OF trigger on a view to redirect **insert**s

into a table or to **update** multiple tables that are part of a view. You can use INSTEAD OF triggers on either object views (see Chapter 33) or relational views.

For example, if a view involves a join of two tables, your ability to use the **update** command on records in the view is limited. However, if you use an INSTEAD OF trigger, you can tell Oracle how to **update**, **delete**, or **insert** records in the view's underlying tables when a user attempts to change values via the view. The code in the INSTEAD OF trigger is executed *in place of* the **insert**, **update**, or **delete** command you enter.

NOTE
You can access or change LOB data within BEFORE and INSTEAD OF triggers.

In this chapter, you will see how to implement basic triggers. INSTEAD OF triggers, which were initially introduced to support object views, are described in Chapter 33.

Schema Triggers

You can create triggers on schema-level operations such as **create table**, **alter table**, **drop table**, **audit**, **rename**, **truncate**, and **revoke**. You can even create a **before ddl** trigger. For the most part, schema-level triggers provide two capabilities: preventing DDL operations and providing additional security monitoring when DDL operations occur.

Database-Level Triggers

You can create triggers to be fired on database events, including errors, logins, logoffs, shutdowns, and startups. You can use this type of trigger to automate database maintenance or auditing actions. Virtual Private Databases, as described in Chapter 19, rely on database-level triggers to establish session-context variable values.

Trigger Syntax

The full syntax for the **create trigger** command is shown in the Alphabetical Reference section of this book. The following listing contains an abbreviated version of the command syntax:

```
create [or replace] trigger [schema .] trigger
{ before | after | instead of }
{ dml_event_clause
| { ddl_event [or ddl_event]...
  | database_event [or database_event]...
  }
  on { [schema .] schema | database }
}
[when ( condition ) ]
{ pl/sql_block | call_procedure_statement }
```

The syntax options available depend on the type of trigger in use. For example, a trigger on a DML event will use the *dml_event_clause*, which follows this syntax:

```
{ delete | insert | update [of column [, column]...] }
[or { delete | insert | update [of column [, column]...] }]...
on { [schema .] table | [nested table nested_table_column of] [schema .] view
}
[referencing_clause] [for each row]
```

Clearly, there is a great deal of flexibility in the design of a trigger. The **before** and **after** keywords indicate whether the trigger should be executed before or after the triggering event. If the **instead of** clause is used, the trigger's code will be executed instead of the event that caused the trigger to be invoked. The **delete**, **insert**, and **update** keywords (the last of which may include a column list) indicate the type of data manipulation that will constitute a triggering event. When referring to the old and new values of columns, you can use the defaults ("old" and "new") or you can use the **referencing** clause to specify other names.

When the **for each row** clause is used, the trigger will be a row-level trigger; otherwise, it will be a statement-level trigger. The **when** clause is used to further restrict when the trigger is executed. The restrictions enforced in the **when** clause may include checks of old and new data values.

For example, suppose we want to track any changes to the Rating value in the BOOKSHELF table whenever rating values are lowered. First, we'll create a table that will store the audit records:

```
drop table BOOKSHELF_AUDIT;
create table BOOKSHELF_AUDIT
(Title       VARCHAR2(100),
Publisher    VARCHAR2(20),
CategoryName VARCHAR2(20),
Old_Rating   VARCHAR2(2),
New_Rating   VARCHAR2(2),
Audit_Date   DATE);
```

The following row-level BEFORE UPDATE trigger will be executed only if the Rating value is lowered. This example also illustrates the use of the **new** keyword, which refers to the new value of the column, and the **old** keyword, which refers to the old value of the column.

```
create or replace trigger BOOKSHELF_BEF_UPD_ROW
before update on BOOKSHELF
for each row
when (new.Rating < old.Rating)
begin
  insert into BOOKSHELF_AUDIT
   (Title, Publisher, CategoryName,
    Old_Rating, New_Rating, Audit_Date)
  values
   (:old.Title, :old.Publisher, :old.CategoryName,
    :old.Rating, :new.Rating, Sysdate);
end;
/
```

Breaking this **create trigger** command into its components makes it easier to understand. First, the trigger is named:

```
create or replace trigger BOOKSHELF_BEF_UPD_ROW
```

The name of the trigger contains the name of the table it acts upon and the type of trigger it is. (See "Naming Triggers," later in this chapter, for information on naming conventions.)

This trigger applies to the BOOKSHELF table; it will be executed before **update** transactions have been committed to the database:

```
before update on BOOKSHELF
```

Because the **for each row** clause is used, the trigger will apply to each row changed by the **update** statement. If this clause is not used, then the trigger will execute at the statement level.

```
for each row
```

The **when** clause adds further criteria to the triggering condition. The triggering event not only must be an **update** of the BOOKSHELF table, but also must reflect a lowering of the Rating value:

```
when (new.Rating < old.Rating)
```

The PL/SQL code shown in the following listing is the trigger body. The commands shown here are to be executed for every **update** of the BOOKSHELF table that passes the **when** condition. For this to succeed, the BOOKSHELF_AUDIT table must exist, and the owner of the trigger must have been granted privileges (directly, not via roles) on that table. This example **insert**s the old values from the BOOKSHELF record into the BOOKSHELF_AUDIT table before the BOOKSHELF record is updated.

```
begin
  insert into BOOKSHELF_AUDIT
   (Title, Publisher, CategoryName,
    Old_Rating, New_Rating, Audit_Date)
  values
   (:old.Title, :old.Publisher, :old.CategoryName,
    :old.Rating, :new.Rating, Sysdate);
end;
```

NOTE

*When the **new** and **old** keywords are referenced in the PL/SQL block, they are preceded by colons (:).*

This example is typical of auditing triggers. The auditing activity is completely transparent to the user who performs the **update** of the BOOKSHELF table. However, the transaction against the BOOKSHELF table is dependent on the successful execution of the trigger.

Combining DML Trigger Types

Triggers for multiple **insert**, **update**, and **delete** commands on a table can be combined into a single trigger, provided they are all at the same level (row level or statement level). The following

example shows a trigger that is executed whenever an **insert** or an **update** occurs. Several points (shown in bold) should stand out in this example:

- The **update** portion of the trigger occurs only when the Rating column's value is updated.
- An **if** clause is used within the PL/SQL block to determine which of the two commands invoked the trigger.
- The column to be changed is specified in the *dml_event_clause* instead of in the **when** clause, as in prior examples.

```
drop trigger BOOKSHELF_BEF_UPD_ROW;

create or replace trigger BOOKSHELF_BEF_UPD_INS_ROW
before insert or update of Rating on BOOKSHELF
for each row
begin
 if INSERTING then
  insert into BOOKSHELF_AUDIT
   (Title, Publisher, CategoryName,
    New_Rating, Audit_Date)
  values
   (:new.Title, :new.Publisher, :new.CategoryName,
    :new.Rating, Sysdate);
 else -- if not inserting then we are updating the Rating
  insert into BOOKSHELF_AUDIT
   (Title, Publisher, CategoryName,
    Old_Rating, New_Rating, Audit_Date)
  values
   (:old.Title, :old.Publisher, :old.CategoryName,
    :old.Rating, :new.Rating, Sysdate);
 end if;
end;
/
```

Again, look at the trigger's component parts. First, it is named and identified as a BEFORE INSERT and BEFORE UPDATE (of Rating) trigger, executing **for each row**:

```
create or replace trigger BOOKSHELF_BEF_UPD_INS_ROW
before insert or update of Rating on BOOKSHELF
for each row
```

The trigger body then follows. In the first part of the trigger body, shown in the following listing, the type of transaction is checked via an **if** clause. Valid transaction types are INSERTING, DELETING, and UPDATING. In this case, the trigger checks to see if the record is being inserted into the BOOKSHELF table. If it is, then the first part of the trigger body is executed. The INSERTING portion of the trigger body inserts the new values of the record into the BOOKSHELF_AUDIT table.

```
begin
 if INSERTING then
```

```
  insert into BOOKSHELF_AUDIT
   (Title, Publisher, CategoryName,
    New_Rating, Audit_Date)
  values
   (:new.Title, :new.Publisher, :new.CategoryName,
    :new.Rating, Sysdate);
```

Other transaction types can then be checked. In this example, because the trigger executed, the transaction must be either an **insert** or an **update** of the Rating column. Since the **if** clause in the first half of the trigger body checks for **insert**s, and the trigger is only executed for **insert**s and **update**s, the only conditions that should execute the second half of the trigger body are **update**s of Rating. Therefore, no additional **if** clauses are necessary to determine the DML event type. This portion of the trigger body is the same as in the previous example: Prior to being updated, the old values in the row are written to the BOOKSHELF_AUDIT table.

```
else -- if not inserting then we are updating the Rating
  insert into BOOKSHELF_AUDIT
   (Title, Publisher, CategoryName,
    Old_Rating, New_Rating, Audit_Date)
  values
   (:old.Title, :old.Publisher, :old.CategoryName,
    :old.Rating, :new.Rating, Sysdate);
```

Combining trigger types in this manner may help you to coordinate trigger development among multiple developers, since it consolidates all the database events that depend on a single table.

Setting Inserted Values

You may use triggers to set column values during **insert**s and **update**s. The previous examples in this chapter set the BOOKSHELF_AUDIT.Audit_Date value to the result of the SYSDATE function. You may also use updates to support different application needs, such as storing an uppercase version of a value along with the mixed-case version entered by users. In that case, you may have partially denormalized your table to include a column for the derived data. Storing this data in an uppercase format (for this example, in the column UpperPerson) allows you to display data to the users in its natural format while using the uppercase column during queries.

Since the uppercase version of the value is derived data, it may become out of sync with the user-entered column. Unless your application supplies a value for the uppercase version during **insert**s, that column's value will be **NULL** when a new row is entered.

To avoid this synchronization problem, you can use a table-level trigger. Put a BEFORE INSERT and a BEFORE UPDATE trigger on the table; they will act at the row level. As shown in the following listing, this approach can set a new value for UpperName every time the Name column's value is changed in BOOKSHELF_CHECKOUT:

```
alter table BOOKSHELF_CHECKOUT add (UpperName VARCHAR2(25));

create or replace trigger BOOKSHELF_CHECKOUT_BUI_ROW
before insert or update of Name on BOOKSHELF_CHECKOUT
for each row
begin
```

```
  :new.UpperName := UPPER(:new.Name);
end;
/
```

In this example, the trigger body determines the value for UpperName by using the UPPER function on the Name column. This trigger will be executed every time a row is inserted into BOOKSHELF_CHECKOUT and every time the Name column is updated. The Name and UpperName columns will thus be kept in sync.

Maintaining Duplicated Data

The method of setting values via triggers, shown in the preceding section, can be combined with the remote data access methods described in Chapter 23. As with materialized views, you may replicate all or some of the rows in a table.

For example, you may want to create and maintain a second copy of your application's audit log. By doing so, you safeguard against a single application's erasing the audit log records created by multiple applications. Duplicate audit logs are frequently used in security monitoring.

Consider the BOOKSHELF_AUDIT table used in the examples in this chapter. A second table, BOOKSHELF_AUDIT_DUP, could be created, possibly in a remote database. For this example, assume that a database link called AUDIT_LINK can be used to connect the user to the database in which BOOKSHELF_AUDIT_DUP resides (refer to Chapter 23 for details on database links).

```
drop table BOOKSHELF_AUDIT_DUP;
create table BOOKSHELF_AUDIT_DUP
(Title       VARCHAR2(100),
Publisher    VARCHAR2(20),
CategoryName VARCHAR2(20),
Old_Rating   VARCHAR2(2),
New_Rating   VARCHAR2(2),
Audit_Date   DATE);
```

To automate populating the BOOKSHELF_AUDIT_DUP table, the following trigger could be placed on the BOOKSHELF_AUDIT table:

```
create or replace trigger BOOKSHELF_AUDIT_AFT_INS_ROW
after insert on BOOKSHELF_AUDIT
for each row
begin
 insert into BOOKSHELF_AUDIT_DUP@AUDIT_LINK
   (Title, Publisher, CategoryName,
    New_Rating, Audit_Date)
   values (:new.Title, :new.Publisher, :new.CategoryName,
    :new.New_Rating, :new.Audit_Date);
end;
/
```

As the trigger header shows, this trigger executes for each row that is inserted into the BOOKSHELF_AUDIT table. It inserts a single record into the BOOKSHELF_AUDIT_DUP table in the database defined by the AUDIT_LINK database link. AUDIT_LINK may point to a database located on a remote server. If there is any problem with the remote insert initiated via the trigger, the local insert will fail.

For asynchronous replication requirements, consider using materialized views instead (see Chapter 24).

You can see the actions of these triggers by inserting a row into BOOKSHELF:

```
insert into BOOKSHELF
(Title, Publisher, CategoryName, Rating) values
('HARRY POTTER AND THE CHAMBER OF SECRETS',
'SCHOLASTIC','CHILDRENFIC','4');

1 row created.


select Title from BOOKSHELF_AUDIT;

TITLE
--------------------------------------------------
HARRY POTTER AND THE CHAMBER OF SECRETS


select Title from BOOKSHELF_AUDIT_DUP;

TITLE
--------------------------------------------------
HARRY POTTER AND THE CHAMBER OF SECRETS
```

Customizing Error Conditions

Within a single trigger, you may establish different error conditions. For each of the error conditions you define, you may select an error message that appears when the error occurs. The error numbers and messages that are displayed to the user are set via the RAISE_APPLICATION_ERROR procedure, which may be called from within any trigger.

The following example shows a statement-level BEFORE DELETE trigger on the BOOKSHELF table. When a user attempts to **delete** a record from the BOOKSHELF table, this trigger is executed and checks two system conditions: that the day of the week is neither Saturday nor Sunday, and that the Oracle username of the account performing the **delete** begins with the letters "LIB." The trigger's components will be described following the listing.

```
create or replace trigger BOOKSHELF_BEF_DEL
before delete on BOOKSHELF
declare
  weekend_error EXCEPTION;
  not_library_user EXCEPTION;
begin
 if TO_CHAR(SysDate,'DY') = 'SAT' or
    TO_CHAR(SysDate,'DY') = 'SUN' THEN
   RAISE weekend_error;
end if;
if SUBSTR(User,1,3) <> 'LIB' THEN
   RAISE not_library_user;
end if;
```

```
EXCEPTION
  WHEN weekend_error THEN
    RAISE_APPLICATION_ERROR (-20001,
     'Deletions not allowed on weekends');
  WHEN not_library_user THEN
    RAISE_APPLICATION_ERROR (-20002,
     'Deletions only allowed by Library users');
end;
/
```

The header of the trigger defines it as a statement-level BEFORE DELETE trigger:

```
create or replace trigger BOOKSHELF_BEF_DEL
before delete on BOOKSHELF
```

There are no **when** clauses in this trigger, so the trigger body is executed for all **delete**s.

The next portion of the trigger declares the names of the two exceptions that are defined within this trigger:

```
declare
  weekend_error EXCEPTION;
  not_library_user EXCEPTION;
```

The first part of the trigger body contains an **if** clause that uses the TO_CHAR function on the SYSDATE function. If the current day is either Saturday or Sunday, then the WEEKEND_ERROR error condition is raised. This error condition, called an *exception*, must be defined within the trigger body.

```
if TO_CHAR(SysDate,'DY') = 'SAT' or
   TO_CHAR(SysDate,'DY') = 'SUN' THEN
   RAISE weekend_error;
end if;
```

A second **if** clause checks the User pseudo-column to see if its first three letters are "LIB." If the username does not begin with "LIB," then the NOT_LIBRARY_USER exception is raised. In this example, the <> operator is used; this is equivalent to != (meaning "not equals").

```
if SUBSTR(User,1,3) <> 'LIB' THEN
   RAISE not_library_user;
end if;
```

The final portion of the trigger body tells the trigger how to handle the exceptions. It begins with the keyword **exception**, followed by a **when** clause for each of the exceptions. Each of the exceptions in this trigger calls the RAISE_APPLICATION_ERROR procedure, which takes two input parameters: the error number (which must be between –20001 and –20999), and the error message to be displayed. In this example, two different error messages are defined, one for each of the defined exceptions:

```
EXCEPTION
  WHEN weekend_error THEN
    RAISE_APPLICATION_ERROR (-20001,
```

```
     'Deletions not allowed on weekends');
  WHEN not_library_user THEN
    RAISE_APPLICATION_ERROR (-20002,
     'Deletions only allowed by Library users');
```

The use of the RAISE_APPLICATION_ERROR procedure gives you great flexibility in managing the error conditions that may be encountered within your trigger. For a further description of procedures, see Chapter 31.

NOTE
*The exceptions will still be raised even if the **delete** operations find no rows to delete.*

When a non-"LIB" user attempts to delete a row from BOOKSHELF on a weekday, this is the result:

```
delete from BOOKSHELF
where Title = 'MY LEDGER';

delete from BOOKSHELF
            *
ERROR at line 1:
ORA-20002: Deletions only allowed by Library users
ORA-06512: at "PRACTICE.BOOKSHELF_BEF_DEL", line 17
ORA-04088: error during execution of trigger 'PRACTICE.BOOKSHELF_BEF_DEL'
```

If you attempt to delete a row on a Saturday or Sunday, the ORA-20001 error is returned instead. As soon as an exception is encountered, Oracle leaves the main executable commands section of the trigger and processes the exception. You will only see the first exception encountered.

Calling Procedures Within Triggers

Rather than creating a large block of code within a trigger body, you can save the code as a stored procedure and call the procedure from within the trigger, by using the **call** command. For example, if you create an INSERT_BOOKSHELF_AUDIT_DUP procedure that inserts rows into BOOKSHELF_AUDIT_DUP, you can call it from a trigger on the BOOKSHELF_AUDIT table, as shown in the following listing:

```
create or replace trigger BOOKSHELF_AFT_INS_ROW
after insert on BOOKSHELF_AUDIT
for each row
begin
call INSERT_BOOKSHELF_AUDIT_DUP(:new.Title, :new.Publisher,
   :new.CategoryName, :new.Old_Rating, :new.New_Rating,
   :new.Audit_Date);
end;
/
```

Naming Triggers

The name of a trigger should clearly indicate the table it applies to, the DML commands that trigger it, its before/after status, and whether it is a row-level or statement-level trigger. Since a trigger name cannot exceed 30 characters in length, you need to use a standard set of abbreviations when naming. In general, the trigger name should include as much of the table name as possible. Thus, when creating a BEFORE UPDATE, row-level trigger on a table named BOOKSHELF_CHECKOUT, you should not name the trigger BEFORE_UPDATE_ROW_LEVEL_BC. A better name would be BOOKSHELF_CHECKOUT_BEF_UPD_ROW.

Creating DDL Event Triggers

You can create triggers that are executed when a DDL event occurs. If you are planning to use this feature solely for security purposes, you should investigate using the **audit** command instead. For example, you can use a DDL event trigger to execute the trigger code for **create**, **alter**, and **drop** commands performed on a cluster, function, index, package, procedure, role, sequence, synonym, table, tablespace, trigger, type, user, or view. If you use the **on schema** clause, the trigger will execute for any new data dictionary objects created in your schema. The following example will execute a procedure named INSERT_AUDIT_RECORD whenever objects are created within your schema:

```
create or replace trigger CREATE_DB_OBJECT_AUDIT
after create on schema
begin
  call INSERT_AUDIT_RECORD (ora_dict_obj_name);
end;
/
```

As shown in this example, you can reference system attributes such as the object name. The available attributes are listed in Table 30-1.

To protect the objects within a schema, you may want to create a trigger that is executed for each attempted **drop table** command. That trigger will have to be a BEFORE DROP trigger:

```
create or replace trigger PREVENT_DROP
  before drop on Practice.schema
begin
  if ora_dict_obj_owner = 'PRACTICE'
     and ora_dict_obj_name like 'BOO%'
     and ora_dict_obj_type = 'TABLE'
  then
     RAISE_APPLICATION_ERROR (
        -20002, 'Operation not permitted.');
  end if;
end;
/
```

Note that the trigger references the event attributes within its **if** clauses. Attempting to drop a table in the Practice schema whose name starts with BOO results in the following:

```
drop table BOOKSHELF_AUDIT_DUP;
```

```
drop table BOOKSHELF_AUDIT_DUP
*
ERROR at line 1:
ORA-00604: error occurred at recursive SQL level 1
ORA-20002: Operation not permitted.
ORA-06512: at line 6
```

You can use the RAISE_APPLICATION_ERROR procedure to further customize the error message displayed to the user.

Creating Database Event Triggers

Like DML events, database events can execute triggers. When a database event occurs (a shutdown, startup, or error), you can execute a trigger that references the attributes of the event (as with DDL events). You could use a database event trigger to perform system maintenance functions immediately after each database startup.

For example, the following trigger pins packages on each database startup. *Pinning packages* is an effective way of keeping large PL/SQL objects in the shared pool of memory, improving performance, and enhancing database stability. This trigger, PIN_ON_STARTUP, will run each time the database is started. You should create this trigger while connected as a user with ADMINISTER DATABASE TRIGGER privilege.

```
rem  while connected as a user with the
rem  ADMINISTER DATABASE TRIGGER system privilege:

create or replace trigger PIN_ON_STARTUP
after startup on database
begin
   DBMS_SHARED_POOL.KEEP (
      'SYS.STANDARD', 'P');
end;
/
```

This example shows a simple trigger that will be executed immediately after a database startup. You can modify the list of packages in the trigger body to include those most used by your applications.

Startup and shutdown triggers can access the ora_instance_num, ora_database_name, ora_login_user, and ora_sysevent attributes listed in Table 30-1. See the Alphabetical Reference for the full **create trigger** command syntax.

Enabling and Disabling Triggers

Unlike declarative integrity constraints (such as NOT NULL and PRIMARY KEY), triggers do not necessarily affect all rows in a table. They only affect operations of the specified type, and then only while the trigger is enabled. Any data created prior to a trigger's creation will not be affected by the trigger.

By default, a trigger is enabled when it is created. However, there are situations in which you may want to disable a trigger. The two most common reasons involve data loads. During large data loads, you may want to disable triggers that would execute during the load. Disabling the triggers during data loads may dramatically improve the performance of the load. After the data

has been loaded, you need to manually perform the data manipulation that the trigger would have performed had it been enabled during the data load.

The second data load–related reason for disabling a trigger occurs when a data load fails and has to be performed a second time. In such a case, it is likely that the data load partially succeeded—and thus the trigger was executed for a portion of the records loaded. During a subsequent load, the same records would be inserted. Thus, it is possible that the same trigger will be executed twice for the same **insert** operation (when that **insert** operation occurs during both loads). Depending on the nature of the operations and the triggers, this may not be desirable. If the trigger was enabled during the failed load, then it may need to be disabled prior to the start of a second data-load process. After the data has been loaded, you need to manually perform the data manipulation that the trigger would have performed had it been enabled during the data load.

To enable a trigger, use the **alter trigger** command with the **enable** keyword. To use this command, you must either own the table or have the ALTER ANY TRIGGER system privilege. A sample **alter trigger** command is shown in the following listing:

```
alter trigger  BOOKSHELF_BEF_UPD_INS_ROW enable;
```

A second method of enabling triggers uses the **alter table** command, with the **enable all triggers** clause. You may not enable specific triggers with this command; you must use the **alter trigger** command to do that. The following example shows the usage of the **alter table** command:

```
alter table BOOKSHELF enable all triggers;
```

To use the **alter table** command, you must either own the table or have the ALTER ANY TABLE system privilege.

You can disable triggers using the same basic commands (requiring the same privileges) with modifications to their clauses. For the **alter trigger** command, use the **disable** clause:

```
alter trigger BOOKSHELF_BEF_UPD_INS_ROW disable;
```

For the **alter table** command, use the **disable all triggers** clause:

```
alter table BOOKSHELF disable all triggers;
```

You can manually compile triggers. Use the **alter trigger compile** command to manually compile existing triggers that have become invalid. Since triggers have dependencies, they can become invalid if an object the trigger depends on changes. The **alter trigger debug** command allows PL/SQL information to be generated during trigger recompilation.

Replacing Triggers

The status of a trigger is the only portion that can be altered. To alter a trigger's body, the trigger must be re-created or replaced. When replacing a trigger, use the **create or replace trigger** command (refer to "Trigger Syntax" and the examples earlier in the chapter).

Dropping Triggers

Triggers may be dropped via the **drop trigger** command. To drop a trigger, you must either own the trigger or have the DROP ANY TRIGGER system privilege. An example of this command is shown in Table 30-1:

```
drop trigger BOOKSHELF_BEF_UPD_INS_ROW;
```

Attribute	Type	Description	Example
ora_client_ip_address	VARCHAR2	Returns the IP address of the client in a LOGON event when the underlying protocol is TCP/IP.	`if (ora_sysevent = 'LOGON') then addr := ora_client_ip_address; end if;`
ora_database_name	VARCHAR2(50)	Database name.	`Declare db_name VARCHAR2(50); begin db_name := ora_database_name; end;`
ora_des_encrypted_password	VARCHAR2	The DES encrypted password of the user being created or altered.	`if (ora_dict_obj_type = 'USER') then insert into event_table (ora_des_encrypted_password); end if;`
ora_dict_obj_name	VARCHAR(30)	Name of the dictionary object on which the DDL operation occurred.	`insert into event_table ('Changed object is ' || ora_dict_obj_name');`
ora_dict_obj_name_list (name_list OUT ora_name_list_t)	BINARY_INTEGER	Returns the list of object names of objects being modified in the event.	`if (ora_sysevent = 'ASSOCIATE STATISTICS') then number_modified := ora_dict_obj_name_list (name_list); end if;`
ora_dict_obj_owner	VARCHAR(30)	Owner of the dictionary object on which the DDL operation occurred.	`insert into event_table ('object owner is' || ora_dict_obj_owner');`
ora_dict_obj_owner_list(owner_list OUT ora_name_list_t)	BINARY_INTEGER	Returns the list of object owners of objects being modified in the event.	`if (ora_sysevent = 'ASSOCIATE STATISTICS') then number_of_modified_objects := ora_dict_obj_owner_list(owner_list); end if;`

TABLE 30-1. *System-Defined Event Attributes*

Attribute	Type	Description	Example
ora_dict_obj_type	VARCHAR(20)	Type of the dictionary object on which the DDL operation occurred.	`insert into event_table ('This object is a ' || ora_dict_obj_type);`
ora_grantee(user_list OUT ora_name_list_t)	BINARY_INTEGER	Returns the grantees of a grant event in the OUT parameter; returns the number of grantees in the return value.	`if (ora_sysevent = 'GRANT') then number_of_users := ora_grantee(user_list); end if;`
ora_instance_num	NUMBER	Instance number.	`if (ora_instance_num = 1) then insert into event_table ('1'); end if;`
ora_is_alter_column(column_name IN VARCHAR2)	BOOLEAN	Returns true if the specified column is altered.	`if (ora_sysevent = 'ALTER' and ora_dict_obj_type = 'TABLE') then alter_column := ora_is_alter_column('FOO'); end if;`
ora_is_creating_nested_table	BOOLEAN	Returns true if the current event is creating a nested table.	`if (ora_sysevent = 'CREATE' and ora_dict_obj_type = 'TABLE' and ora_is_creating_nested_table) then insert into event_tab values ('A nested table is created'); end if;`
ora_is_drop_column(column_name IN VARCHAR2)	BOOLEAN	Returns true if the specified column is dropped.	`if (ora_sysevent = 'ALTER' and ora_dict_obj_type = 'TABLE') then drop_column := ora_is_drop_column('FOO'); end if;`
ora_is_servererror	BOOLEAN	Returns true if given error is on error stack, FALSE otherwise.	`if (ora_is_servererror(error_number)) then insert into event_table ('Server error!!'); end if;`
ora_login_user	VARCHAR2(30)	Login username.	`select ora_login_user from dual;`

ora_partition_pos	BINARY_INTEGER	In an INSTEAD OF trigger for CREATE TABLE, the position within the SQL text where you could insert a PARTITION clause.	`-- Retrieve ora_sql_txt into -- sql_text variable first.`
			
			`n := ora_partition_pos; new_stmt := substr(sql_text, 1, n-1) || ' ' || my_partition_clause || ' ' || substr(sql_text, n));`

TABLE 30-1. *System-Defined Event Attributes* (continued)

Attribute	Type	Description	Example
ora_privilege_list(privilege_list OUT ora_name_ list_t)	BINARY_INTEGER	Returns the list of privileges being granted by the grantee or the list of privileges revoked from the revokees in the OUT parameter; returns the number of privileges in the return value.	`if (ora_sysevent = 'GRANT' or ora_sysevent = 'REVOKE') then number_of_privileges := ora_privilege_list(priv_list); end if;`
ora_revokee (user_list OUT ora_name_ list_t)	BINARY_INTEGER	Returns the revokees of a revoke event in the OUT parameter; returns the number of revokees in the return value.	`if (ora_sysevent = 'REVOKE') then number_of_users := ora_ revokee(user_list);`
ora_server_error	NUMBER	Given a position (1 for top of stack), it returns the error number at that position on error stack.	`insert into event_table ('top stack error ' || ora_server_ error(1));`
ora_server_error_depth	BINARY_INTEGER	Returns the total number of error messages on the error stack.	`n := ora_server_error_depth; -- This value is used with -- other functions such as -- ora_server_error`
ora_server_error_msg (position in binary_integer)	VARCHAR2	Given a position (1 for top of stack), it returns the error message at that position on error stack.	`insert into event_table ('top stack error message' || ora_ server_error_msg(1));`
ora_server_error_num_params (position in binary_integer)	BINARY_INTEGER	Given a position (1 for top of stack), it returns the number of strings that have been substituted into the error message using a format like "%s".	`n := ora_server_error_num_ params(1);`
ora_server_error_param (position in binary_integer, param in binary_integer)	VARCHAR2	Given a position (1 for top of stack) and a parameter number, returns the matching "%s", "%d", and so on substitution value in the error message.	`-- E.g. the 2nd %s in a message -- like "Expected %s, found %s" param := ora_server_error_ param(1,2);`

TABLE 30-1. *System-Defined Event Attributes* (continued)

Attribute	Type	Description	Example

ora_sql_txt (sql_text out ora_name_list_t)	BINARY_INTEGER	Returns the SQL text of the triggering statement in the OUT parameter. If the statement is long, it is broken up into multiple PL/SQL table elements. The function return value specifies how many elements are in the PL/SQL table.	`sql_text ora_name_list_t;`
			`stmt VARCHAR2(2000);`
			`...`
			`n := ora_sql_txt(sql_text);`
			`for i in 1..n loop`
			`  stmt := stmt || sql_text(i);`
			`end loop;`
			`insert into event_table ('text of triggering statement: ' || stmt);`

ora_sysevent	VARCHAR2(20)	System event firing the trigger: Event name is same as that in the syntax.	`insert into event_table (ora_sysevent);`

ora_with_grant_option	BOOLEAN	Returns true if the privileges are granted with grant option.	`if (ora_sysevent = 'GRANT' and ora_with_grant_option = TRUE)`
			`  then insert into event_table ('with grant option');`
			`end if;`

space_error_info(error_number OUT NUMBER, error_type OUT VARCHAR2, object_owner OUT VARCHAR2, table_space_name OUT VARCHAR2, object_name OUT VARCHAR2, sub_object_name OUT VARCHAR2)	BOOLEAN	Returns true if the error is related to an out-of-space condition, and fills in the OUT parameters with information about the object that caused the error.	`if (space_error_info(eno, typ, owner, ts, obj, subobj) = TRUE)`
			`then.put_line('The object ' || run out of space.');`
			`dbms_outputobj || ' owned by ' || owner || ' has`
			`end if;`

TABLE 30-1. *System-Defined Event Attributes* (continued)

DESCRIPTION **CREATE PROCEDURE** creates the specification and body of a procedure. A procedure may have parameters, named arguments of a certain datatype. The PL/SQL block defines the behavior of the procedure as a series of declarations, PL/SQL program statements, and exceptions.

The **IN** qualifier means that you have to specify a value for the argument when you call the procedure. The **OUT** qualifier means that the procedure passes a value back to the caller through this argument. The **IN OUT** qualifier combines the meaning of both **IN** and **OUT**—you specify a value, and the procedure replaces it with a value. If you don't have any qualifier, the argument defaults to **IN**. The difference between a function and a procedure is that a function returns a value to the calling environment in addition to any arguments that are **OUT** arguments.

The PL/SQL block can refer to programs in an external C library or to a Java call specification. The *invoker_rights* (**AUTHID**) clause lets you specify whether the procedure executes with the privileges of the function owner or the privileges of the user who is calling the function.

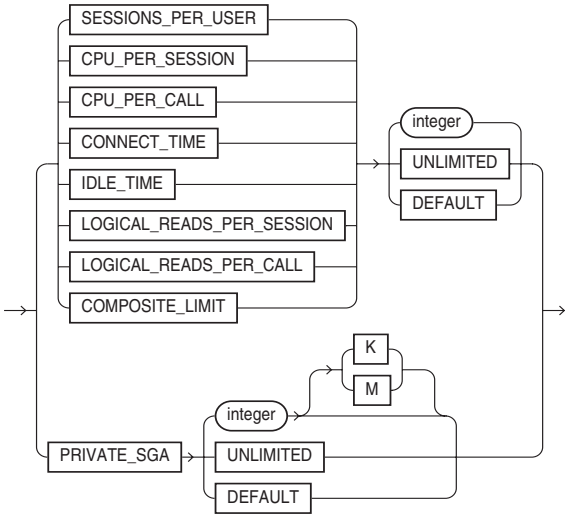
CREATE PROFILE

SEE ALSO **ALTER PROFILE**, **ALTER RESOURCE COST**, **ALTER SYSTEM**, **ALTER USER**, **CREATE USER**, **DROP PROFILE**, Chapter 18

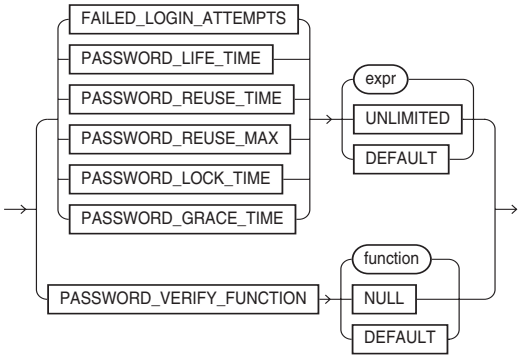
FORMAT

create_profile::=

resource_parameters::=



password_parameters::=



DESCRIPTION **CREATE PROFILE** creates a set of limits on the use of database resources. When you associate the profile with a user with **CREATE USER** or **ALTER USER**, you can control what the user does via those limits. To use **CREATE PROFILE**, you must enable resource limits through the initialization parameter RESOURCE_LIMIT or through the **ALTER SYSTEM** command.

SESSIONS_PER_USER limits the user to *integer* concurrent SQL sessions. CPU_PER_SESSION limits the CPU time in hundredths of seconds. CPU_PER_CALL limits the CPU time for a parse, execute, or fetch call in hundredths of seconds. CONNECT_TIME limits elapsed time of a session in minutes. IDLE_TIME disconnects a user after this number of minutes; this does not apply while a query is running. LOGICAL_READS_PER_SESSION limits the number of blocks read per session; LOGICAL_READS_PER_CALL does the same thing for parse, execute, or fetch calls. PRIVATE_SGA limits the amount of space you can allocate in the SGA as private; the **K** and **M** options apply only to this limit. COMPOSITE_LIMIT limits the total resource cost for a session in service units based on a weighted sum of CPU, connect time, logical reads, and private SGA resources.

UNLIMITED means there is no limit on a particular resource. DEFAULT picks up the limit from the DEFAULT profile, which you can change through the **ALTER PROFILE** command.

If a user exceeds a limit, Oracle aborts and rolls back the transaction, then ends the session. You must have the CREATE PROFILE system privilege to create a profile. You associate a profile to a user with the **ALTER USER** command.

CREATE ROLE

SEE ALSO **ALTER ROLE**, **ALTER USER**, **CREATE USER**, **DROP ROLE**, **GRANT**, **REVOKE**, ROLE, **SET ROLE**, Chapter 18

FORMAT

```
CREATE ROLE role
[ NOT IDENTIFIED
| IDENTIFIED { BY password |
  USING [schema .] package | EXTERNALLY | GLOBALLY }];
```

DESCRIPTION With **CREATE ROLE**, you can create a named role or set of privileges. When you grant the role to a user, you grant him or her all the privileges of that role. You first create the role with **CREATE ROLE**, then grant privileges to the role using the **GRANT** command. When a user wants to access something that the role allows, he or she enables the role with **SET ROLE**. Alternatively, the role can be set as the user's DEFAULT role via the **ALTER USER** or **CREATE USER** command.