PART 2

DATABASE DESIGN

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 part, we focus primarily on the design of the database schema. We also outline some of the other design tasks.

The entity-relationship (E-R) model described in Chapter 6 is a high-level data model. Instead of representing all data in tables, it distinguishes between basic objects, called *entities*, and *relationships* among these objects. It is often used as a first step in database-schema design.

Relational database design—the design of the relational schema— was covered informally in earlier chapters. There are, however, principles that can be used to distinguish good database designs from bad ones. These are formalized by means of several "normal forms" that offer different trade-offs between the possibility of inconsistencies and the efficiency of certain queries. Chapter 7 describes the formal design of relational schemas.

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

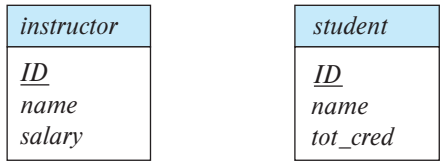


Figure 6.1 E-R diagram showing entity sets *instructor* and *student*.

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-*

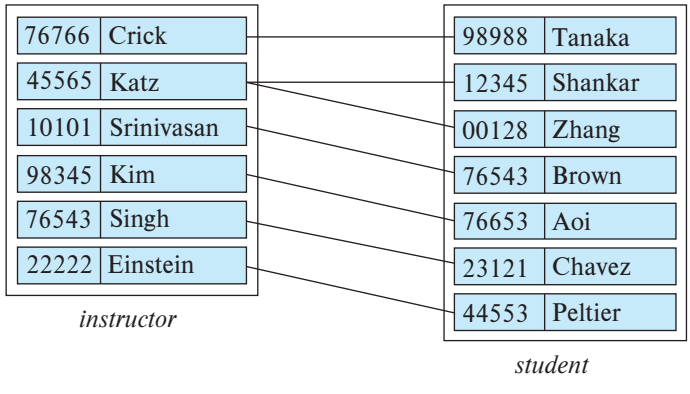


Figure 6.2 Relationship set *advisor* (only some attributes of *instructor* and *student* are shown).

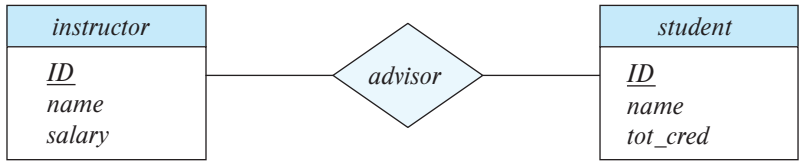


Figure 6.3 E-R diagram showing relationship set *advisor*.

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

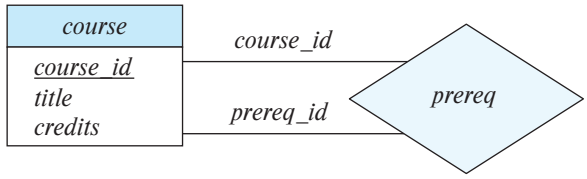


Figure 6.4 E-R diagram with role indicators.

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.

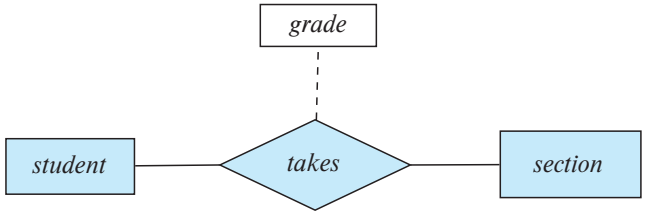


Figure 6.5 E-R diagram with an attribute attached to a relationship set.

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}.

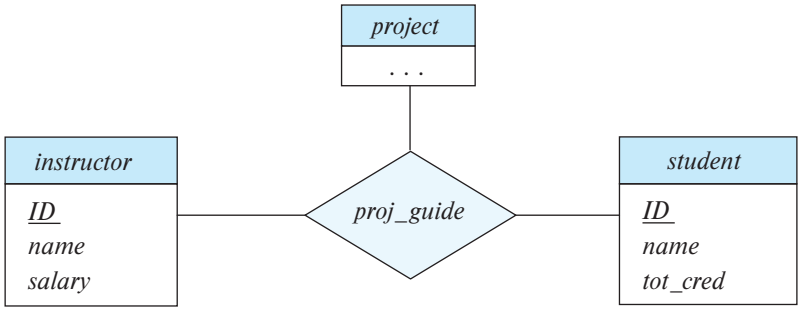


Figure 6.6 E-R diagram with a ternary relationship *proj_guide*.

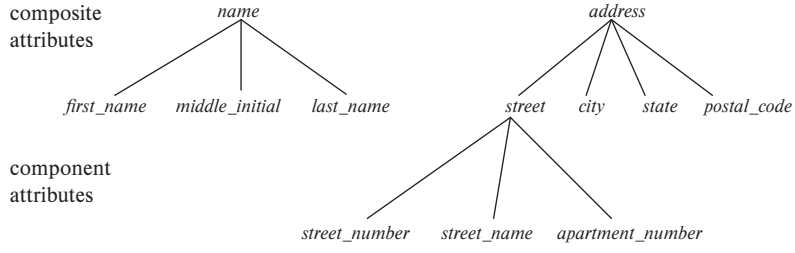


Figure 6.7 Composite attributes instructor *name* and *address*.

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

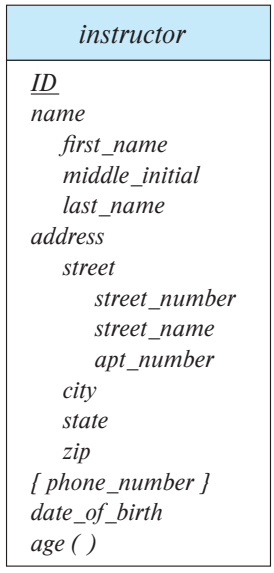


Figure 6.8 E-R diagram with composite, multivalued, and derived attributes.

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.)

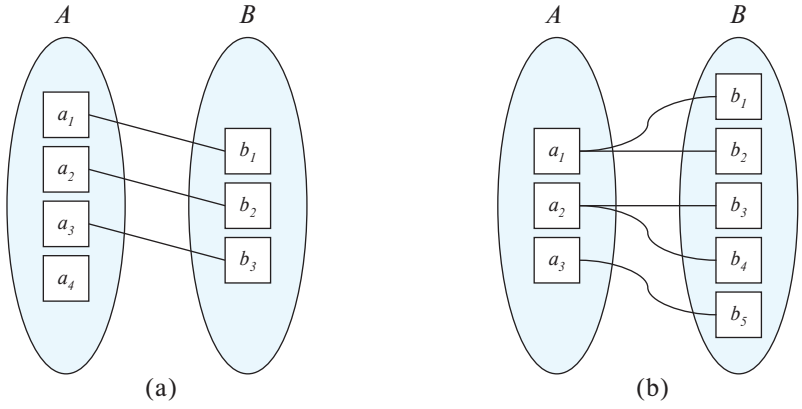


Figure 6.9 Mapping cardinalities. (a) One-to-one. (b) One-to-many.

- **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

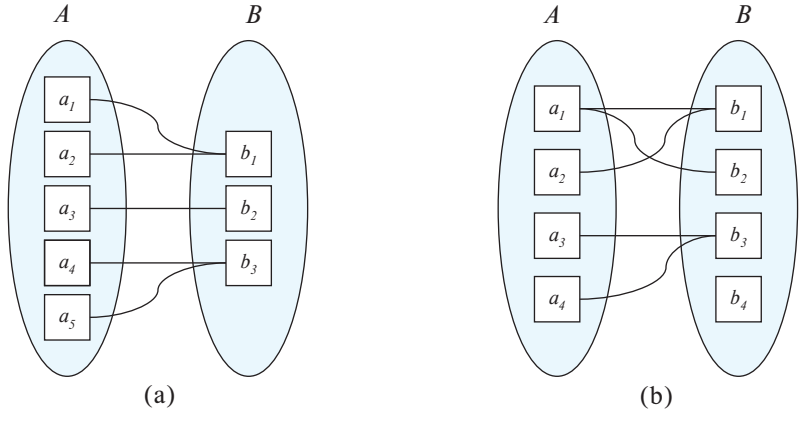
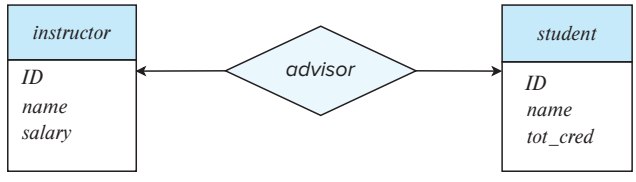


Figure 6.10 Mapping cardinalities. (a) Many-to-one. (b) Many-to-many.

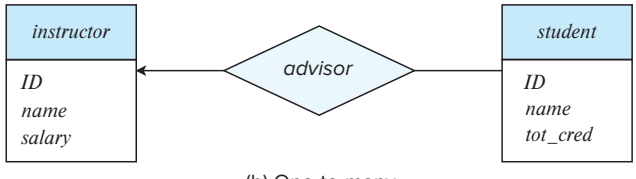
cardinalities can be used to specify constraints on what relationships are permitted in the real world.

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:

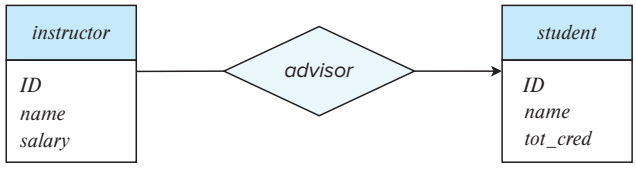
- **One-to-one**. We draw a directed line from the relationship set to both entity sets. For example, in Figure 6.11a, the directed lines to *instructor* and *student* indicate that an instructor may advise at most one student, and a student may have at most one advisor.



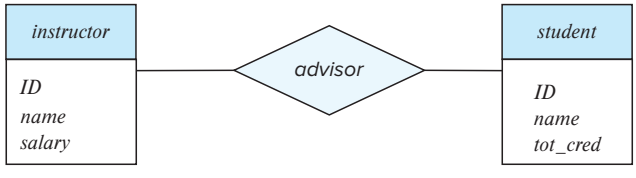
(a) One-to-one



(b) One-to-many



(c) Many-to-one



(d) Many-to-many

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

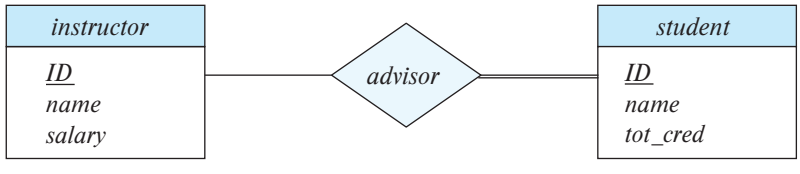


Figure 6.12 E-R diagram showing total participation.

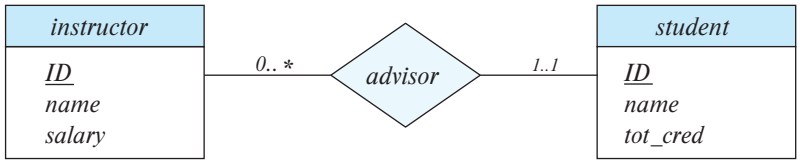


Figure 6.13 Cardinality limits on relationship sets.

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.

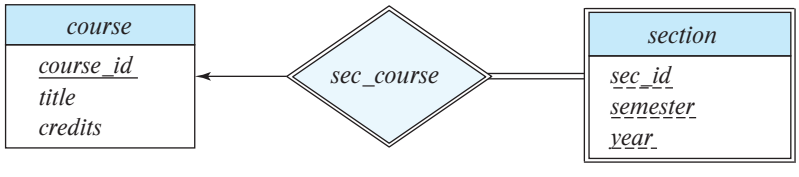


Figure 6.14 E-R diagram with a weak entity set.

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*.

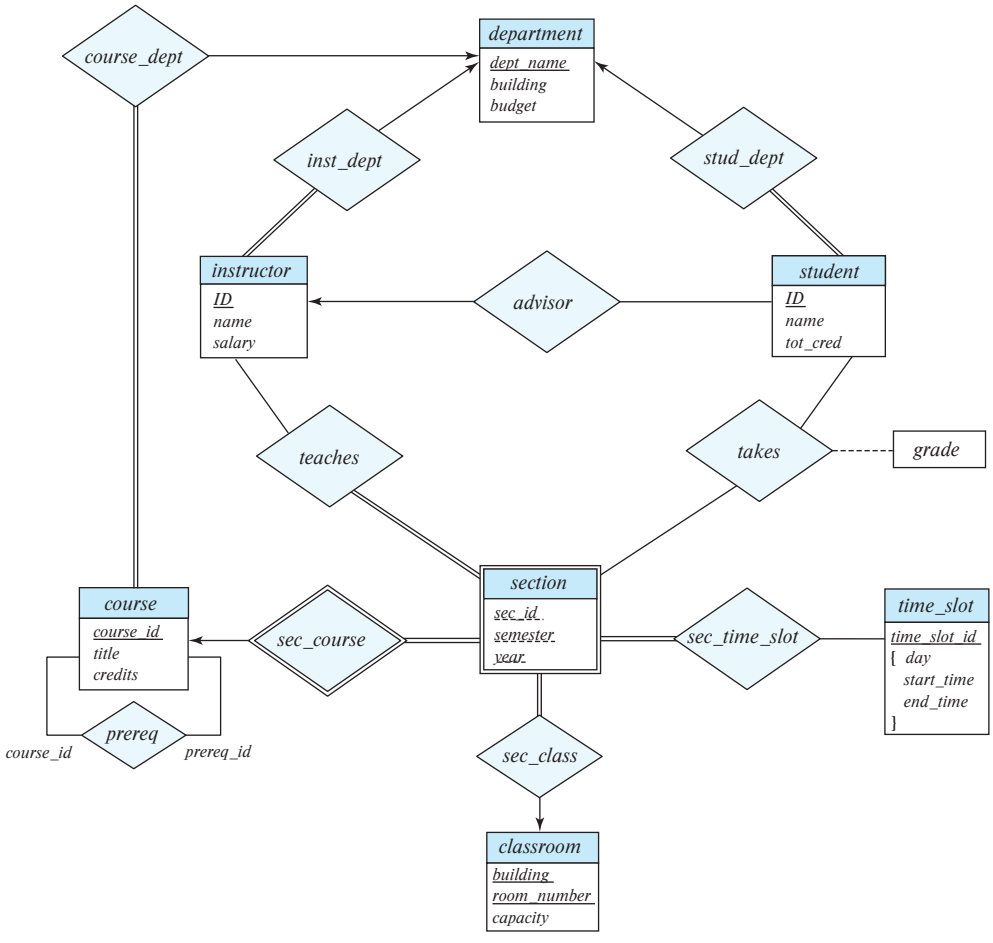


Figure 6.15 E-R diagram for a university enterprise.

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.

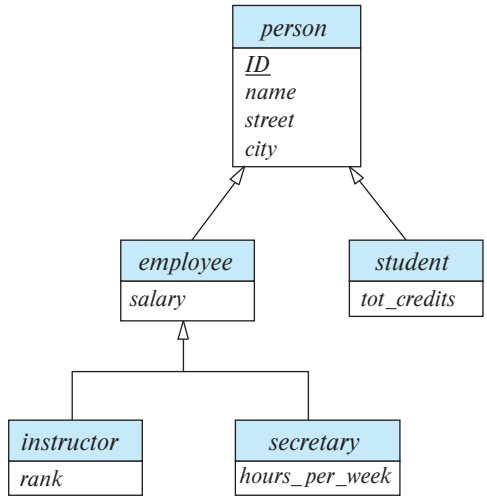


Figure 6.18 Specialization and generalization.

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

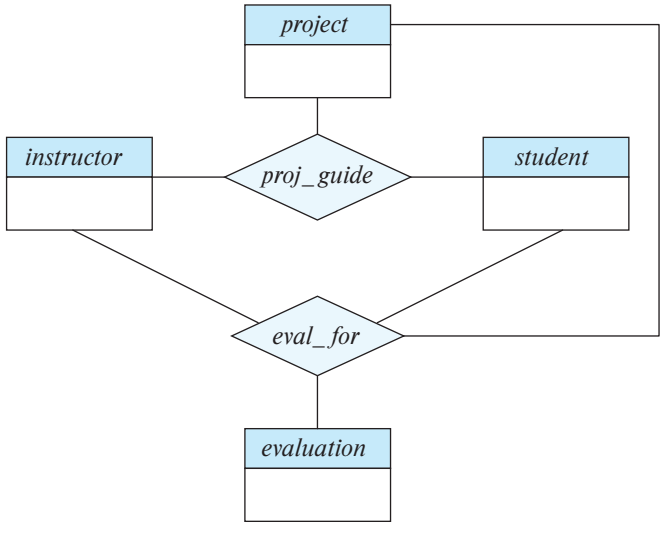


Figure 6.19 E-R diagram with redundant relationships.

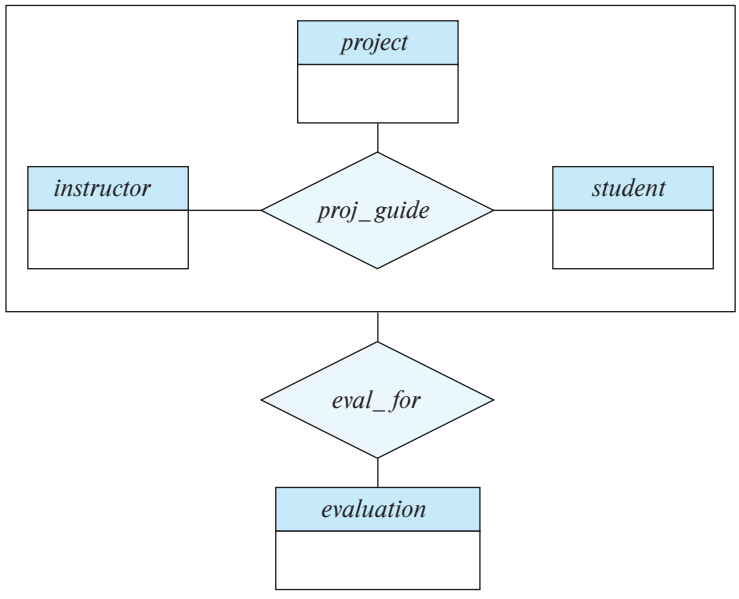


Figure 6.20 E-R diagram with aggregation.

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.

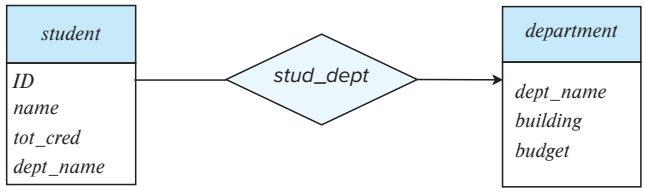
6.8.6.2 Representation of Aggregation

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.

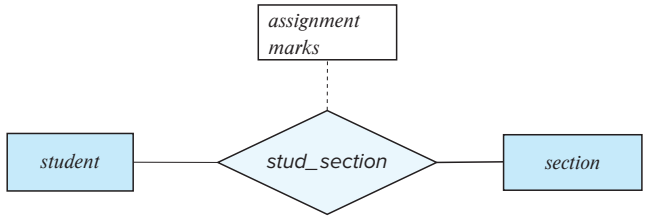
The rules we saw earlier for creating primary-key and foreign-key constraints on relationship sets can be applied to relationship sets involving aggregations as well, with the aggregation treated like any other entity set. The primary key of the aggregation is the primary key of its defining relationship 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

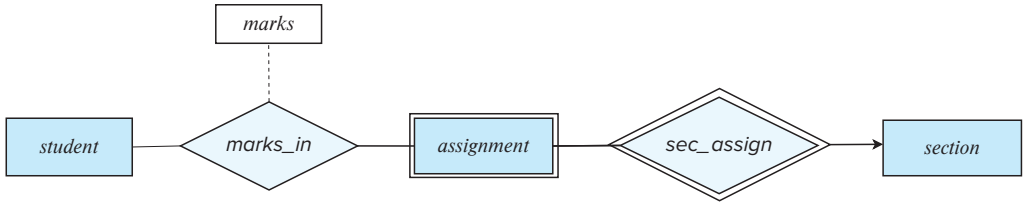
6.9.1 Common Mistakes in E-R Diagrams

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.

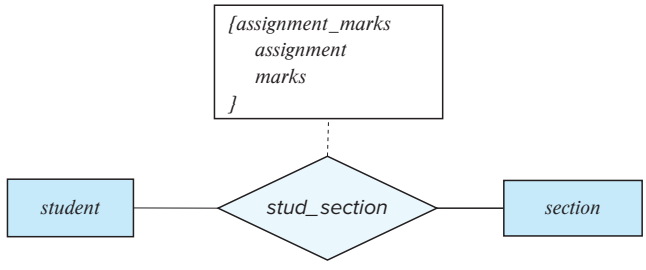
Another related mistake that people sometimes make is to designate the primary-key attributes of the related entity sets as attributes of the relationship set. For example, *ID* (the primary-key attributes of *student*) and *ID* (the primary key of *instructor*) should not appear as attributes of the 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*

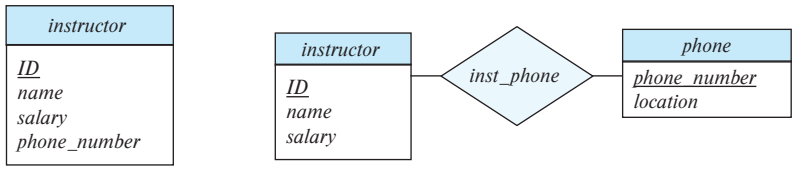


Figure 6.23 Alternatives for adding *phone* to the *instructor* entity set.

_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

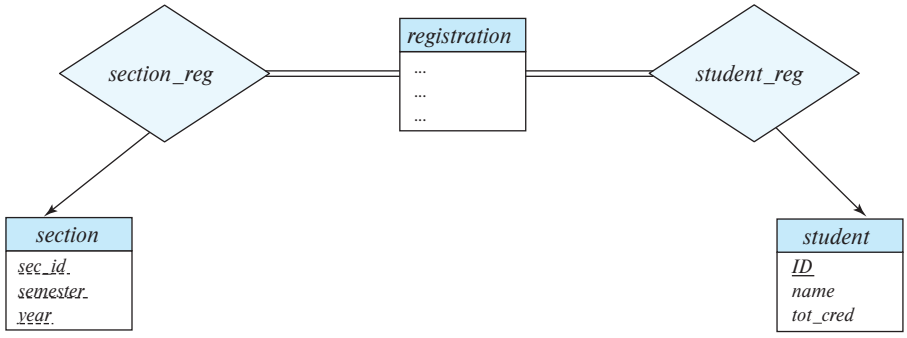


Figure 6.24 Replacement of *takes* by *registration* and two relationship sets.

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.

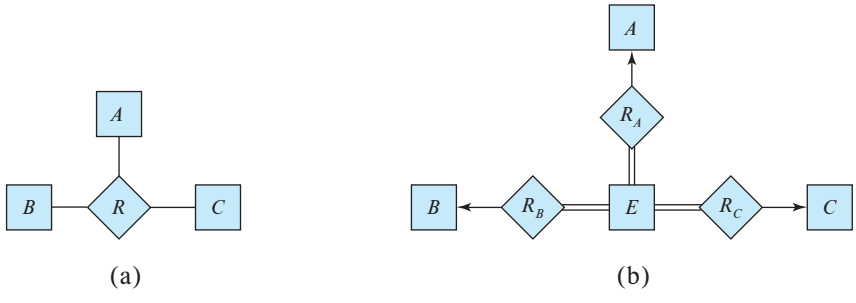


Figure 6.25 Ternary relationship versus three binary relationships.

- 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

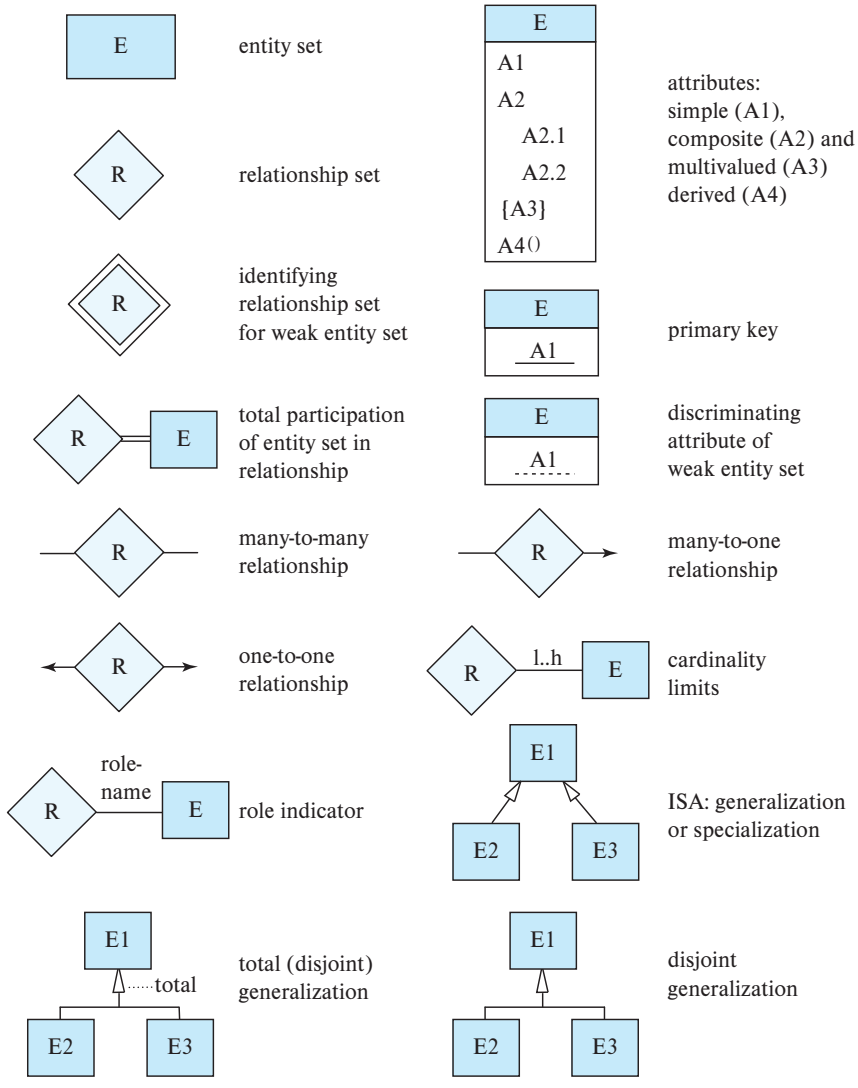


Figure 6.26 Symbols used in the E-R notation.

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-

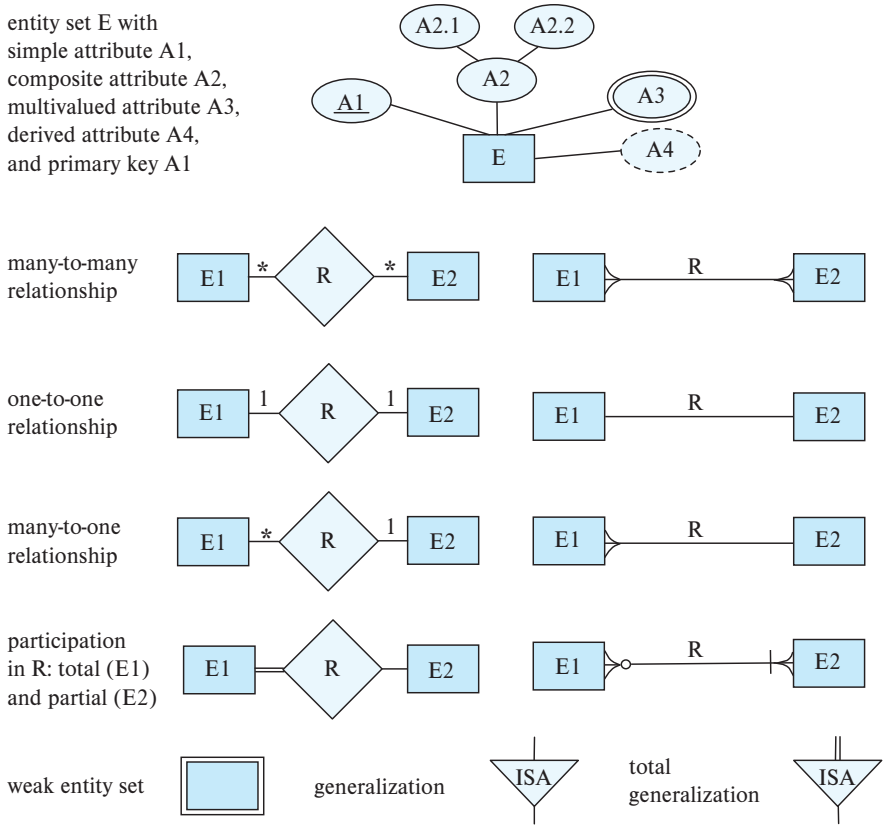


Figure 6.27 Alternative E-R notations.

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 $E1$.

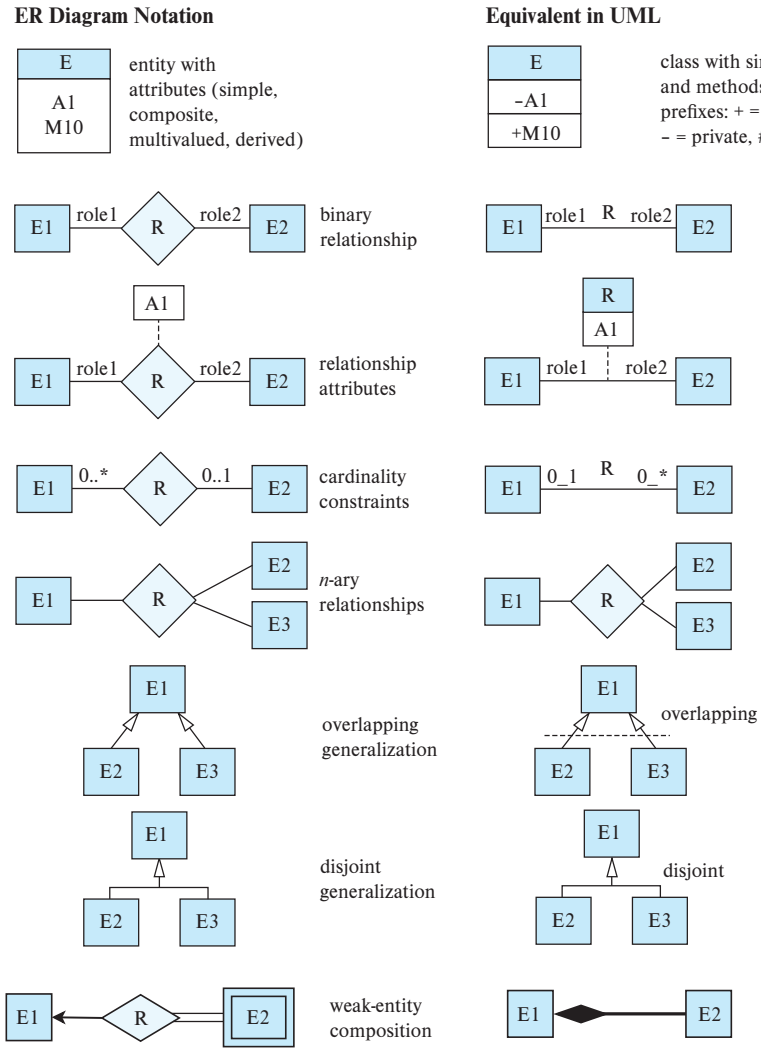


Figure 6.28 Symbols used in the UML class diagram notation.

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.

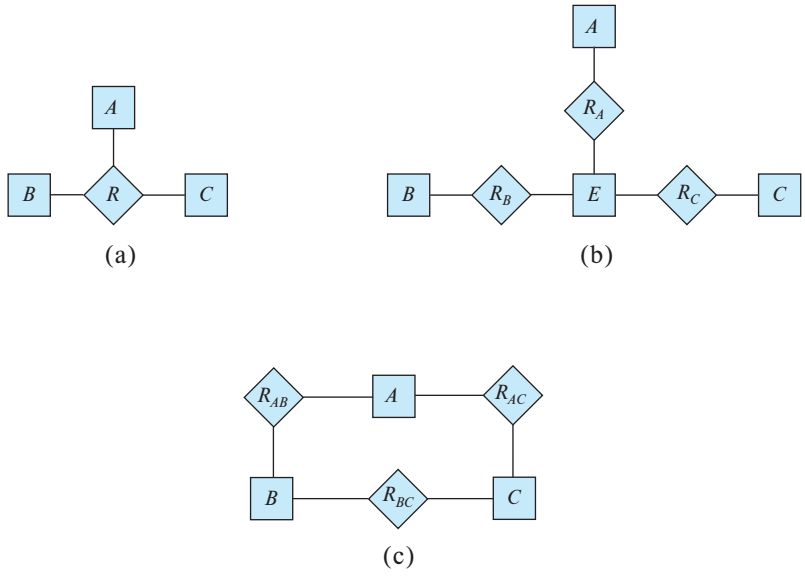


Figure 6.29 Representation of a ternary relationship using binary relationships.

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?

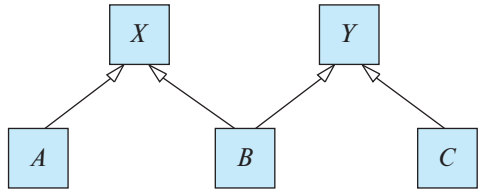
6.10 Consider a many-to-one relationship R between entity sets A and B. Suppose the relation created from R is combined with the relation created from A. In SQL, attributes participating in a foreign key constraint can be null. Explain how a constraint on total participation of A in R can be enforced 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.

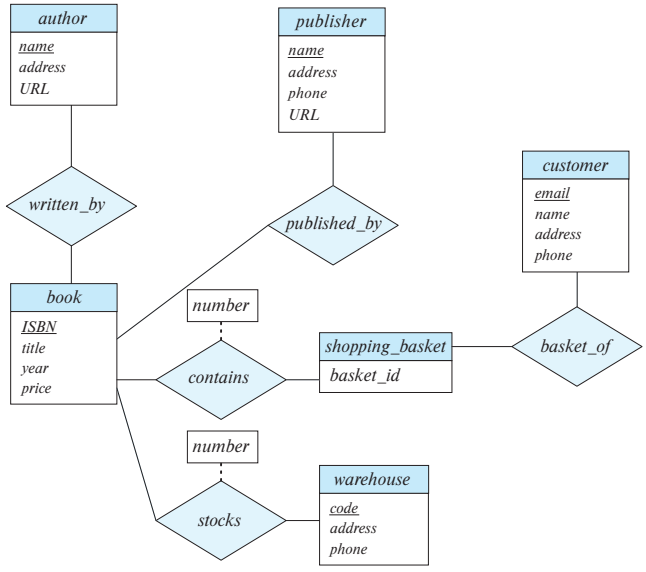


Figure 6.30 E-R diagram for modeling an online bookstore.

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 7

Relational Database Design

In this chapter, we consider the problem of designing a schema for a relational database. Many of the issues in doing so are similar to design issues we considered in Chapter 6 using the E-R model.

In general, the goal of relational database design is to generate a set of relation schemas that allows us to store information without unnecessary redundancy, yet also allows us to retrieve information easily. This is accomplished by designing schemas that are in an appropriate *normal form*. To determine whether a relation schema is in one of the desirable normal forms, we need information about the real-world enterprise that we are modeling with the database. Some of this information exists in a well-designed E-R diagram, but additional information about the enterprise may be needed as well.

In this chapter, we introduce a formal approach to relational database design based on the notion of functional dependencies. We then define normal forms in terms of functional dependencies and other types of data dependencies. First, however, we view the problem of relational design from the standpoint of the schemas derived from a given entity-relationship design.

7.1 Features of Good Relational Designs

Our study of entity-relationship design in Chapter 6 provides an excellent starting point for creating a relational database design. We saw in Section 6.7 that it is possible to generate a set of relation schemas directly from the E-R design. The goodness (or badness) of the resulting set of schemas depends on how good the E-R design was in the first place. Later in this chapter, we shall study precise ways of assessing the desirability of a collection of relation schemas. However, we can go a long way toward a good design using concepts we have already studied. For ease of reference, we repeat the schemas for the university database in Figure 7.1.

Suppose that we had started out when designing the university enterprise with the schema *in_dep*.

in_dep (*ID*, *name*, *salary*, *dept_name*, *building*, *budget*)

classroom(*building*, *room_number*, *capacity*)
department(*dept_name*, *building*, *budget*)
course(*course_id*, *title*, *dept_name*, *credits*)
instructor(*ID*, *name*, *dept_name*, *salary*)
section(*course_id*, *sec_id*, *semester*, *year*, *building*, *room_number*, *time_slot_id*)
teaches(*ID*, *course_id*, *sec_id*, *semester*, *year*)
student(*ID*, *name*, *dept_name*, *tot_cred*)
takes(*ID*, *course_id*, *sec_id*, *semester*, *year*, *grade*)
advisor(*s_ID*, *i_ID*)
time_slot(*time_slot_id*, *day*, *start_time*, *end_time*)
prereq(*course_id*, *prereq_id*)

Figure 7.1 Database schema for the university example.

This represents the result of a natural join on the relations corresponding to *instructor* and *department*. This seems like a good idea because some queries can be expressed using fewer joins, until we think carefully about the facts about the university that led to our E-R design.

Let us consider the instance of the *in_dep* relation shown in Figure 7.2. Notice that we have to repeat the department information ("building" and "budget") once for each instructor in the department. For example, the information about the Comp. Sci. department (Taylor, 100000) is included in the tuples of instructors Katz, Srinivasan, and Brandt.

It is important that all these tuples agree as to the budget amount since otherwise our database would be inconsistent. In our original design using *instructor* and *department*, we stored the amount of each budget exactly once. This suggests that using *in_dep* is a bad idea since it stores the budget amounts redundantly and runs the risk that some user might update the budget amount in one tuple but not all, and thus create inconsistency.

Even if we decided to live with the redundancy problem, there is still another problem with the *in_dep* schema. Suppose we are creating a new department in the university. In the alternative design above, we cannot represent directly the information concerning a department (*dept_name*, *building*, *budget*) unless that department has at least one instructor at the university. This is because tuples in the *in_dep* table require values for *ID*, *name*, and *salary*. This means that we cannot record information about the newly created department until the first instructor is hired for the new department. In the old design, the schema *department* can handle this, but under the revised design, we would have to create a tuple with a null value for *building* and *budget*. In some cases null values are troublesome, as we saw in our study of SQL. However, if we decide that

ID	*name*	*salary*	*dept_name*	*building*	*budget*
22222	Einstein	95000	Physics	Watson	70000
12121	Wu	90000	Finance	Painter	120000
32343	El Said	60000	History	Painter	50000
45565	Katz	75000	Comp. Sci.	Taylor	100000
98345	Kim	80000	Elec. Eng.	Taylor	85000
76766	Crick	72000	Biology	Watson	90000
10101	Srinivasan	65000	Comp. Sci.	Taylor	100000
58583	Califieri	62000	History	Painter	50000
83821	Brandt	92000	Comp. Sci.	Taylor	100000
15151	Mozart	40000	Music	Packard	80000
33456	Gold	87000	Physics	Watson	70000
76543	Singh	80000	Finance	Painter	120000

Figure 7.2 The *in_dep* relation.

this is not a problem to us in this case, then we can proceed to use the revised design, though, as we noted, we would still have the redundancy problem.

7.1.1 Decomposition

The only way to avoid the repetition-of-information problem in the *in_dep* schema is to decompose it into two schemas (in this case, the *instructor* and *department* schemas). Later on in this chapter we shall present algorithms to decide which schemas are appropriate and which ones are not. In general, a schema that exhibits repetition of information may have to be decomposed into several smaller schemas.

Not all decompositions of schemas are helpful. Consider an extreme case in which all schemas consist of one attribute. No interesting relationships of any kind could be expressed. Now consider a less extreme case where we choose to decompose the *employee* schema (Section 6.8):

$$employee\ (ID, name, street, city, salary)$$

into the following two schemas:

$$employee1\ (ID, name)$$
$$employee2\ (name, street, city, salary)$$

The flaw in this decomposition arises from the possibility that the enterprise has two employees with the same name. This is not unlikely in practice, as many cultures have certain highly popular names. Each person would have a unique employee-id, which is why *ID* can serve as the primary key. As an example, let us assume two employees,

both named Kim, work at the university and have the following tuples in the relation on schema *employee* in the original design:

(57766, Kim, Main, Perryridge, 75000)
(98776, Kim, North, Hampton, 67000)

Figure 7.3 shows these tuples, the resulting tuples using the schemas resulting from the decomposition, and the result if we attempted to regenerate the original tuples using a natural join. As we see in the figure, the two original tuples appear in the result along with two new tuples that incorrectly mix data values pertaining to the two employees named Kim. Although we have more tuples, we actually have less information in the following sense. We can indicate that a certain street, city, and salary pertain to someone named Kim, but we are unable to distinguish which of the Kims. Thus, our decomposition is unable to represent certain important facts about the university

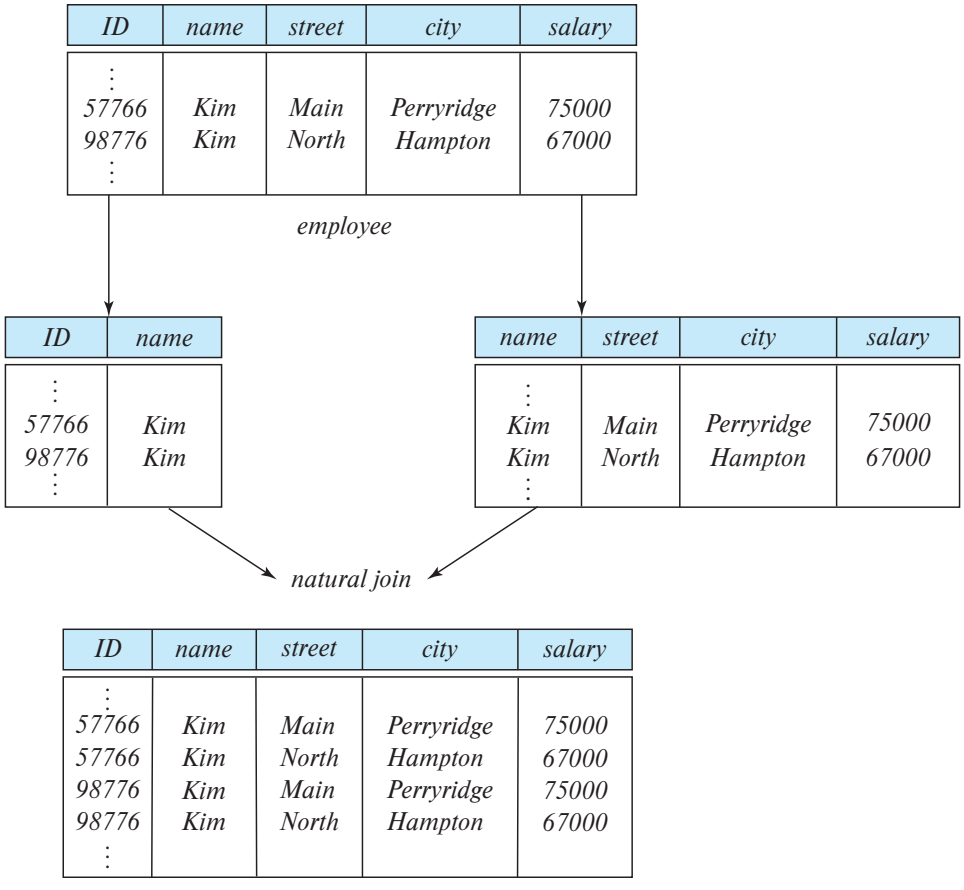


Figure 7.3 Loss of information via a bad decomposition.

employees. We would like to avoid such decompositions. We shall refer to such decompositions as being **lossy decompositions**, and, conversely, to those that are not as **lossless decompositions**.

For the remainder of the text we shall insist that all decompositions should be lossless decompositions.

7.1.2 Lossless Decomposition

Let R be a relation schema and let R_1 and R_2 form a decomposition of R—that is, viewing R, R_1, and R_2 as sets of attributes, $R = R_1 \cup R_2$. We say that the decomposition is a **lossless decomposition** if there is no loss of information by replacing R with two relation schemas R_1 and R_2. Loss of information occurs if it is possible to have an instance of a relation $r(R)$ that includes information that cannot be represented if instead of the instance of $r(R)$ we must use instances of $r_1(R_1)$ and $r_2(R_2)$. More precisely, we say the decomposition is lossless if, for all legal (we shall formally define "legal" in Section 7.2.2.) database instances, relation r contains the same set of tuples as the result of the following SQL query:[1]

```
select *
from (select R1 from r)
     natural join
     (select R2 from r)
```

This is stated more succinctly in the relational algebra as:

$$\Pi_{R_1}(r) \bowtie \Pi_{R_2}(r) = r$$

In other words, if we project r onto R_1 and R_2, and compute the natural join of the projection results, we get back exactly r.

Conversely, a decomposition is lossy if when we compute the natural join of the projection results, we get a proper superset of the original relation. This is stated more succinctly in the relational algebra as:

$$r \subset \Pi_{R_1}(r) \bowtie \Pi_{R_2}(r)$$

Let us return to our decomposition of the *employee* schema into *employee1* and *employee2* (Figure 7.3) and a case where two or more employees have the same name. The result of *employee1* natural join *employee2* is a superset of the original relation *employee*, but the decomposition is lossy since the join result has lost information about which employee identifiers correspond to which addresses and salaries.

[1] The definition of lossless is stated assuming that no attribute that appears on the left side of a functional dependency can have a null value. This is explored further in Exercise 7.10.

It may seem counterintuitive that we have *more* tuples but *less* information, but that is indeed the case. The decomposed version is unable to represent the *absence* of a connection between a name and an address or salary, and absence of a connection is indeed information.

7.1.3 Normalization Theory

We are now in a position to define a general methodology for deriving a set of schemas each of which is in "good form"; that is, does not suffer from the repetition-of-information problem.

The method for designing a relational database is to use a process commonly known as **normalization**. The goal is to generate a set of relation schemas that allows us to store information without unnecessary redundancy, yet also allows us to retrieve information easily. The approach is:

- Decide if a given relation schema is in "good form." There are a number of different forms (called *normal form*s), which we cover in Section 7.3.
- If a given relation schema is not in "good form," then we decompose it into a number of smaller relation schemas, each of which is in an appropriate normal form. The decomposition must be a lossless decomposition.

To determine whether a relation schema is in one of the desirable normal forms, we need additional information about the real-world enterprise that we are modeling with the database. The most common approach is to use **functional dependencies**, which we cover in Section 7.2.

7.2 Decomposition Using Functional Dependencies

A database models a set of entities and relationships in the real world. There are usually a variety of constraints (rules) on the data in the real world. For example, some of the constraints that are expected to hold in a university database are:

1. Students and instructors are uniquely identified by their ID.
2. Each student and instructor has only one name.
3. Each instructor and student is (primarily) associated with only one department.[2]
4. Each department has only one value for its budget, and only one associated building.

[2] An instructor, in most real universities, can be associated with more than one department, for example, via a joint appointment or in the case of adjunct faculty. Similarly, a student may have two (or more) majors or a minor. Our simplified university schema models only the primary department associated with each instructor or student.

An instance of a relation that satisfies all such real-world constraints is called a **legal instance** of the relation; a legal instance of a database is one where all the relation instances are legal instances.

7.2.1 Notational Conventions

In discussing algorithms for relational database design, we shall need to talk about arbitrary relations and their schema, rather than talking only about examples. Recalling our introduction to the relational model in Chapter 2, we summarize our notation here.

- In general, we use Greek letters for sets of attributes (e.g., α). We use an uppercase Roman letter to refer to a relation schema. We use the notation $r(R)$ to show that the schema R is for relation r.

  A relation schema is a set of attributes, but not all sets of attributes are schemas. When we use a lowercase Greek letter, we are referring to a set of attributes that may or may not be a schema. A Roman letter is used when we wish to indicate that the set of attributes is definitely a schema.
- When a set of attributes is a superkey, we may denote it by K. A superkey pertains to a specific relation schema, so we use the terminology "K is a superkey for R."
- We use a lowercase name for relations. In our examples, these names are intended to be realistic (e.g., *instructor*), while in our definitions and algorithms, we use single letters, like r.
- The notation $r(R)$ thus refers to the relation r with schema R. When we write $r(R)$, we thus refer both to the relation and its schema.
- A relation, has a particular value at any given time; we refer to that as an instance and use the term "instance of r." When it is clear that we are talking about an instance, we may use simply the relation name (e.g., r).

For simplicity, we assume that attribute names have only one meaning within the database schema.

7.2.2 Keys and Functional Dependencies

Some of the most commonly used types of real-world constraints can be represented formally as keys (superkeys, candidate keys, and primary keys), or as functional dependencies, which we define below.

In Section 2.3, we defined the notion of a *superkey* as a set of one or more attributes that, taken collectively, allows us to identify uniquely a tuple in the relation. We restate that definition here as follows: Given $r(R)$, a subset K of R is a **superkey** of $r(R)$ if, in any legal instance of $r(R)$, for all pairs t_1 and t_2 of tuples in the instance of r if $t_1 \neq t_2$, then $t_1[K] \neq t_2[K]$. That is, no two tuples in any legal instance of relation $r(R)$ may

have the same value on attribute set K.[3] If no two tuples in r have the same value on K, then a K-value uniquely identifies a tuple in r.

Whereas a superkey is a set of attributes that uniquely identifies an entire tuple, a functional dependency allows us to express constraints that uniquely identify the values of certain attributes. Consider a relation schema $r(R)$, and let $\alpha \subseteq R$ and $\beta \subseteq R$.

- Given an instance of $r(R)$, we say that the instance **satisfies** the **functional dependency** $\alpha \rightarrow \beta$ if for all pairs of tuples t_1 and t_2 in the instance such that $t_1[\alpha] = t_2[\alpha]$, it is also the case that $t_1[\beta] = t_2[\beta]$.
- We say that the functional dependency $\alpha \rightarrow \beta$ **holds** on schema $r(R)$ if, every legal instance of $r(R)$ satisfies the functional dependency.

Using the functional-dependency notation, we say that K *is a superkey for* $r(R)$ if the functional dependency $K \rightarrow R$ holds on $r(R)$. In other words, K is a superkey if, for every legal instance of $r(R)$, for every pair of tuples t_1 and t_2 from the instance, whenever $t_1[K] = t_2[K]$, it is also the case that $t_1[R] = t_2[R]$ (i.e., $t_1 = t_2$).[4]

Functional dependencies allow us to express constraints that we cannot express with superkeys. In Section 7.1, we considered the schema:

$$in\_dep\ (ID, name, salary, dept\_name, building, budget)$$

in which the functional dependency $dept\_name \rightarrow budget$ holds because for each department (identified by $dept\_name$) there is a unique budget amount.

We denote the fact that the pair of attributes (ID, $dept\_name$) forms a superkey for in_dep by writing:

$$ID, dept\_name \rightarrow name, salary, building, budget$$

We shall use functional dependencies in two ways:

1. To test instances of relations to see whether they *satisfy* a given set F of functional dependencies.
2. To specify constraints on the set of legal relations. We shall thus concern ourselves with *only* those relation instances that satisfy a given set of functional dependencies. If we wish to constrain ourselves to relations on schema $r(R)$ that satisfy a set F of functional dependencies, we say that F **holds** on $r(R)$.

[3] In our discussion of functional dependencies, we use equality (=) in the normal mathematical sense, not the three-valued-logic sense of SQL. Said differently, in discussing functional dependencies, we assume no null values.

[4] Note that we assume here that relations are sets. SQL deals with multisets, and a **primary key** declaration in SQL for a set of attributes K requires not only that $t_1 = t_2$ if $t_1[K] = t_2[K]$, but also that there be no duplicate tuples. SQL also requires that attributes in the set K cannot be assigned a *null* value.

A	B	C	D
a_1	b_1	c_1	d_1
a_1	b_2	c_1	d_2
a_2	b_2	c_2	d_2
a_2	b_3	c_2	d_3
a_3	b_3	c_2	d_4

Figure 7.4 Sample instance of relation r.

Let us consider the instance of relation r of Figure 7.4, to see which functional dependencies are satisfied. Observe that $A \rightarrow C$ is satisfied. There are two tuples that have an A value of a_1. These tuples have the same C value—namely, c_1. Similarly, the two tuples with an A value of a_2 have the same C value, c_2. There are no other pairs of distinct tuples that have the same A value. The functional dependency $C \rightarrow A$ is not satisfied, however. To see that it is not, consider the tuples $t_1 = (a_2, b_3, c_2, d_3)$ and $t_2 = (a_3, b_3, c_2, d_4)$. These two tuples have the same C values, c_2, but they have different A values, a_2 and a_3, respectively. Thus, we have found a pair of tuples t_1 and t_2 such that $t_1[C] = t_2[C]$, but $t_1[A] \neq t_2[A]$.

Some functional dependencies are said to be **trivial** because they are satisfied by all relations. For example, $A \rightarrow A$ is satisfied by all relations involving attribute A. Reading the definition of functional dependency literally, we see that, for all tuples t_1 and t_2 such that $t_1[A] = t_2[A]$, it is the case that $t_1[A] = t_2[A]$. Similarly, $AB \rightarrow A$ is satisfied by all relations involving attribute A. In general, a functional dependency of the form $\alpha \rightarrow \beta$ is trivial if $\beta \subseteq \alpha$.

It is important to realize that an instance of a relation may satisfy some functional dependencies that are not required to hold on the relation's schema. In the instance of the *classroom* relation of Figure 7.5, we see that *room_number* $\rightarrow$ *capacity* is satisfied. However, we believe that, in the real world, two classrooms in different buildings can have the same room number but with different room capacity. Thus, it is possible, at some time, to have an instance of the *classroom* relation in which *room_number* $\rightarrow$ *capacity* is not satisfied. So, we would not include *room_number* $\rightarrow$ *capacity* in the set of

building	*room_number*	*capacity*
Packard	101	500
Painter	514	10
Taylor	3128	70
Watson	100	30
Watson	120	50

Figure 7.5 An instance of the *classroom* relation.

functional dependencies that hold on the schema for the *classroom* relation. However, we would expect the functional dependency *building*, *room_number* → *capacity* to hold on the *classroom* schema.

Because we assume that attribute names have only one meaning in the database schema, if we state that a functional dependency $\alpha \rightarrow \beta$ holds as a constraint on the database, then for any schema R such that $\alpha \subseteq R$ and $\beta \subseteq R$, $\alpha \rightarrow \beta$ must hold.

Given that a set of functional dependencies F holds on a relation $r(R)$, it may be possible to infer that certain other functional dependencies must also hold on the relation. For example, given a schema $r(A, B, C)$, if functional dependencies $A \rightarrow B$ and $B \rightarrow C$ hold on r, we can infer the functional dependency $A \rightarrow C$ must also hold on r. This is because, given any value of A, there can be only one corresponding value for B, and for that value of B, there can only be one corresponding value for C. We study in Section 7.4.1, how to make such inferences.

We shall use the notation F^+ to denote the **closure** of the set F, that is, the set of all functional dependencies that can be inferred given the set F. F^+ contains all of the functional dependencies in F.

7.2.3 Lossless Decomposition and Functional Dependencies

We can use functional dependencies to show when certain decompositions are lossless. Let R, R_1, R_2, and F be as above. R_1 and R_2 form a lossless decomposition of R if at least one of the following functional dependencies is in F^+:

- $R_1 \cap R_2 \rightarrow R_1$
- $R_1 \cap R_2 \rightarrow R_2$

In other words, if $R_1 \cap R_2$ forms a superkey for either R_1 or R_2, the decomposition of R is a lossless decomposition. We can use attribute closure to test efficiently for superkeys, as we have seen earlier.

To illustrate this, consider the schema

in_dep (*ID*, *name*, *salary*, *dept_name*, *building*, *budget*)

that we decomposed in Section 7.1 into the *instructor* and *department* schemas:

instructor (*ID*, *name*, *dept_name*, *salary*)
department (*dept_name*, *building*, *budget*)

Consider the intersection of these two schemas, which is *dept_name*. We see that because *dept_name*→ *dept_name*, *building*, *budget*, the lossless-decomposition rule is satisfied.

For the general case of decomposition of a schema into multiple schemas at once, the test for lossless decomposition is more complicated. See the Further Reading section at the end of this chapter for references on this topic.

While the test for binary decomposition is clearly a sufficient condition for lossless decomposition, it is a necessary condition only if all constraints are functional dependencies. We shall see other types of constraints later (in particular, a type of constraint called multivalued dependencies discussed in Section 7.6.1) that can ensure that a decomposition is lossless even if no functional dependencies are present.

Suppose we decompose a relation schema $r(R)$ into $r_1(R_1)$ and $r_2(R_2)$, where $R_1 \cap R_2 \rightarrow R_1$.[5] Then the following SQL constraints must be imposed on the decomposed schema to ensure their contents are consistent with the original schema.

- $R_1 \cap R_2$ is the primary key of r_1.
  This constraint enforces the functional dependency.
- $R_1 \cap R_2$ is a foreign key from r_2 referencing r_1.
  This constraint ensures that each tuple in r_2 has a matching tuple in r_1, without which it would not appear in the natural join of r_1 and r_2.

If r_1 or r_2 is decomposed further, as long as the decomposition ensures that all attributes in $R_1 \cap R_2$ are in one relation, the primary or foreign-key constraint on r_1 or r_2 would be inherited by that relation.

7.3 Normal Forms

As stated in Section 7.1.3, there are a number of different normal forms that are used in designing relational databases. In this section, we cover two of the most common ones.

7.3.1 Boyce–Codd Normal Form

One of the more desirable normal forms that we can obtain is **Boyce–Codd normal form** (**BCNF**). It eliminates all redundancy that can be discovered based on functional dependencies, though, as we shall see in Section 7.6, there may be other types of redundancy remaining.

7.3.1.1 Definition

A relation schema R is in BCNF with respect to a set F of functional dependencies if, for all functional dependencies in F^+ of the form $\alpha \rightarrow \beta$, where $\alpha \subseteq R$ and $\beta \subseteq R$, at least one of the following holds:

[5] The case for $R_1 \cap R_2 \rightarrow R_2$ is symmetrical, and ignored.

- $\alpha \rightarrow \beta$ is a trivial functional dependency (i.e., $\beta \subseteq \alpha$).
- α is a superkey for schema R.

A database design is in BCNF if each member of the set of relation schemas that constitutes the design is in BCNF.

We have already seen in Section 7.1 an example of a relational schema that is not in BCNF:

$$in\_dep\ (ID, name, salary, dept\_name, building, budget)$$

The functional dependency $dept\_name \rightarrow budget$ holds on *in_dep*, but *dept_name* is not a superkey (because a department may have a number of different instructors). In Section 7.1 we saw that the decomposition of *in_dep* into *instructor* and *department* is a better design. The *instructor* schema is in BCNF. All of the nontrivial functional dependencies that hold, such as:

$$ID \rightarrow name, dept\_name, salary$$

include *ID* on the left side of the arrow, and *ID* is a superkey (actually, in this case, the primary key) for *instructor*. (In other words, there is no nontrivial functional dependency with any combination of *name*, *dept_name*, and *salary*, without *ID*, on the left side.) Thus, *instructor* is in BCNF.

Similarly, the *department* schema is in BCNF because all of the nontrivial functional dependencies that hold, such as:

$$dept\_name \rightarrow building, budget$$

include *dept_name* on the left side of the arrow, and *dept_name* is a superkey (and the primary key) for *department*. Thus, *department* is in BCNF.

We now state a general rule for decomposing schemas that are not in BCNF. Let R be a schema that is not in BCNF. Then there is at least one nontrivial functional dependency $\alpha \rightarrow \beta$ such that α is not a superkey for R. We replace R in our design with two schemas:

- $(\alpha \cup \beta)$
- $(R - (\beta - \alpha))$

In the case of *in_dep* above, $\alpha =$ *dept_name*, $\beta = \{$*building*, *budget*$\}$, and *in_dep* is replaced by

- $(\alpha \cup \beta)$ = (*dept_name*, *building*,*budget*)
- $(R - (\beta - \alpha))$ = (*ID*, *name*, *dept_name*, *salary*)

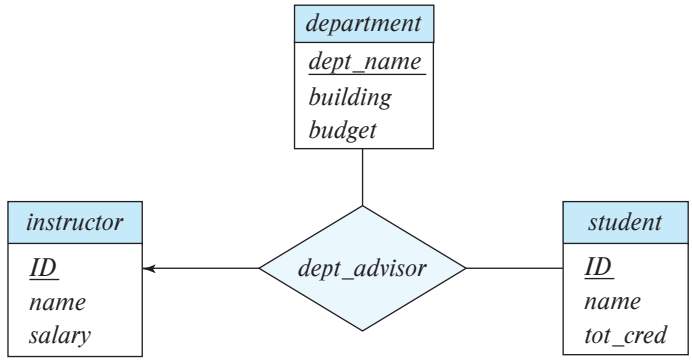


Figure 7.6 The *dept_advisor* relationship set.

In this example, it turns out that $\beta - \alpha = \beta$. We need to state the rule as we did so as to deal correctly with functional dependencies that have attributes that appear on both sides of the arrow. The technical reasons for this are covered later in Section 7.5.1.

When we decompose a schema that is not in BCNF, it may be that one or more of the resulting schemas are not in BCNF. In such cases, further decomposition is required, the eventual result of which is a set of BCNF schemas.

7.3.1.2 BCNF and Dependency Preservation

We have seen several ways in which to express database consistency constraints: primary-key constraints, functional dependencies, **check** constraints, assertions, and triggers. Testing these constraints each time the database is updated can be costly and, therefore, it is useful to design the database in a way that constraints can be tested efficiently. In particular, if testing a functional dependency can be done by considering just one relation, then the cost of testing this constraint is low. We shall see that, in some cases, decomposition into BCNF can prevent efficient testing of certain functional dependencies.

To illustrate this, suppose that we make a small change to our university organization. In the design of Figure 6.15, a student may have only one advisor. This follows from the relationship set *advisor* being many-to-one from *student* to *advisor*. The "small" change we shall make is that an instructor can be associated with only a single department, and a student may have more than one advisor, but no more than one from a given department.[6]

One way to implement this change using the E-R design is by replacing the *advisor* relationship set with a ternary relationship set, *dept_advisor*, involving entity sets *instructor*, *student*, and *department* that is many-to-one from the pair {*student*, *instructor*} to *department* as shown in Figure 7.6. The E-R diagram specifies the constraint that

[6] Such an arrangement makes sense for students with a double major.

"a student may have more than one advisor, but at most one corresponding to a given department."

With this new E-R diagram, the schemas for the *instructor*, *department*, and *student* relations are unchanged. However, the schema derived from the *dept_advisor* relationship set is now:

dept_advisor (*s_ID*, *i_ID*, *dept_name*)

Although not specified in the E-R diagram, suppose we have the additional constraint that "an instructor can act as advisor for only a single department."

Then, the following functional dependencies hold on *dept_advisor*:

$$i\_ID \rightarrow dept\_name$$
$$s\_ID, dept\_name \rightarrow i\_ID$$

The first functional dependency follows from our requirement that "an instructor can act as an advisor for only one department." The second functional dependency follows from our requirement that "a student may have at most one advisor for a given department."

Notice that with this design, we are forced to repeat the department name once for each time an instructor participates in a *dept_advisor* relationship. We see that *dept_advisor* is not in BCNF because *i_ID* is not a superkey. Following our rule for BCNF decomposition, we get:

(*s_ID*, *i_ID*)
(*i_ID*, *dept_name*)

Both the above schemas are BCNF. (In fact, you can verify that any schema with only two attributes is in BCNF by definition.)

Note, however, that in our BCNF design, there is no schema that includes all the attributes appearing in the functional dependency *s_ID*, *dept_name* → *i_ID*. The only dependency that can be enforced on the individual decomposed relations is *ID* → *dept_name*. The functional dependency *s_ID*, *dept_name* → *i_ID* can only be checked by computing the join of the decomposed relations.[7]

Because our design does not permit the enforcement of this functional dependency without a join, we say that our design is *not* **dependency preserving** (we provide a formal definition of dependency preservation in Section 7.4.4). Because dependency preservation is usually considered desirable, we consider another normal form, weaker than BCNF, that will allow us to preserve dependencies. That normal form is called the third normal form.[8]

[7] Technically, it is possible that a dependency whose attributes do not all appear in any one schema is still implicitly enforced, because of the presence of other dependencies that imply it logically. We address that case in Section 7.4.4.

[8] You may have noted that we skipped second normal form. It is of historical significance only and, in practice, one of third normal form or BCNF is always a better choice. We explore second normal form in Exercise 7.19. First normal form pertains to attribute domains, not decomposition. We discuss it in Section 7.8.

7.3.2 Third Normal Form

BCNF requires that all nontrivial dependencies be of the form $\alpha \rightarrow \beta$, where α is a superkey. Third normal form (3NF) relaxes this constraint slightly by allowing certain nontrivial functional dependencies whose left side is not a superkey. Before we define 3NF, we recall that a candidate key is a minimal superkey—that is, a superkey no proper subset of which is also a superkey.

A relation schema R is in **third normal form** with respect to a set F of functional dependencies if, for all functional dependencies in F^+ of the form $\alpha \rightarrow \beta$, where $\alpha \subseteq R$ and $\beta \subseteq R$, at least one of the following holds:

- $\alpha \rightarrow \beta$ is a trivial functional dependency.
- α is a superkey for R.
- Each attribute A in $\beta - \alpha$ is contained in a candidate key for R.

Note that the third condition above does not say that a single candidate key must contain all the attributes in $\beta - \alpha$; each attribute A in $\beta - \alpha$ may be contained in a *different* candidate key.

The first two alternatives are the same as the two alternatives in the definition of BCNF. The third alternative in the 3NF definition seems rather unintuitive, and it is not obvious why it is useful. It represents, in some sense, a minimal relaxation of the BCNF conditions that helps ensure that every schema has a dependency-preserving decomposition into 3NF. Its purpose will become more clear later, when we study decomposition into 3NF.

Observe that any schema that satisfies BCNF also satisfies 3NF, since each of its functional dependencies would satisfy one of the first two alternatives. BCNF is therefore a more restrictive normal form than is 3NF.

The definition of 3NF allows certain functional dependencies that are not allowed in BCNF. A dependency $\alpha \rightarrow \beta$ that satisfies only the third alternative of the 3NF definition is not allowed in BCNF but is allowed in 3NF.[9]

Now, let us again consider the schema for the *dept_advisor* relation, which has the following functional dependencies:

$$i\_ID \rightarrow dept\_name$$
$$s\_ID, dept\_name \rightarrow i\_ID$$

In Section 7.3.1.2, we argued that the functional dependency "$i\_ID \rightarrow dept\_name$" caused the *dept_advisor* schema not to be in BCNF. Note that here $\alpha = i\_ID$, $\beta = dept\_name$, and $\beta - \alpha = dept\_name$. Since the functional dependency $s\_ID, dept\_name \rightarrow$

[9] These dependencies are examples of **transitive dependencies** (see Practice Exercise 7.18). The original definition of 3NF was in terms of transitive dependencies. The definition we use is equivalent but easier to understand.

i_ID holds on *dept_advisor*, the attribute *dept_name* is contained in a candidate key and, therefore, *dept_advisor* is in 3NF.

We have seen the trade-off that must be made between BCNF and 3NF when there is no dependency-preserving BCNF design. These trade-offs are described in more detail in Section 7.3.3.

7.3.3 Comparison of BCNF and 3NF

Of the two normal forms for relational database schemas, 3NF and BCNF there are advantages to 3NF in that we know that it is always possible to obtain a 3NF design without sacrificing losslessness or dependency preservation. Nevertheless, there are disadvantages to 3NF: We may have to use null values to represent some of the possible meaningful relationships among data items, and there is the problem of repetition of information.

Our goals of database design with functional dependencies are:

1. BCNF.
2. Losslessness.
3. Dependency preservation.

Since it is not always possible to satisfy all three, we may be forced to choose between BCNF and dependency preservation with 3NF.

It is worth noting that SQL does not provide a way of specifying functional dependencies, except for the special case of declaring superkeys by using the **primary key** or **unique** constraints. It is possible, although a little complicated, to write assertions that enforce a functional dependency (see Practice Exercise 7.9); unfortunately, currently no database system supports the complex assertions that are required to enforce arbitrary functional dependencies, and the assertions would be expensive to test. Thus even if we had a dependency-preserving decomposition, if we use standard SQL we can test efficiently only those functional dependencies whose left-hand side is a key.

Although testing functional dependencies may involve a join if the decomposition is not dependency preserving, if the database system supports materialized views, we could in principle reduce the cost by storing the join result as materialized view; however, this approach is feasible only if the database system supports **primary key** constraints or **unique** constraints on materialized views. On the negative side, there is a space and time overhead due to the materialized view, but on the positive side, the application programmer need not worry about writing code to keep redundant data consistent on updates; it is the job of the database system to maintain the materialized view, that is, keep it up to date when the database is updated. (In Section 16.5, we outline how a database system can perform materialized view maintenance efficiently.)

Unfortunately, most current database systems limit constraints on materialized views or do not support them at all. Even if such constraints are allowed, there is an additional requirement: the database must update the view and check the constraint

immediately (as part of the same transaction) when an underlying relation is updated. Otherwise, a constraint violation may get detected well after the update has been performed and the transaction that caused the violation has committed.

In summary, even if we are not able to get a dependency-preserving BCNF decomposition, it is still preferable to opt for BCNF, since checking functional dependencies other than primary key constraints is difficult in SQL.

7.3.4 Higher Normal Forms

Using functional dependencies to decompose schemas may not be sufficient to avoid unnecessary repetition of information in certain cases. Consider a slight variation in the *instructor* entity-set definition in which we record with each instructor a set of children's names and a set of landline phone numbers that may be shared by multiple people. Thus, *phone_number* and *child_name* would be multivalued attributes and, following our rules for generating schemas from an E-R design, we would have two schemas, one for each of the multivalued attributes, *phone_number* and *child_name*:

(*ID*, *child_name*)
(*ID*, *phone_number*)

If we were to combine these schemas to get

(*ID*, *child_name*, *phone_number*)

we would find the result to be in BCNF because no nontrivial functional dependencies hold. As a result we might think that such a combination is a good idea. However, such a combination is a bad idea, as we can see by considering the example of an instructor with two children and two phone numbers. For example, let the instructor with *ID* 99999 have two children named "David" and "William" and two phone numbers, 512-555-1234 and 512-555-4321. In the combined schema, we must repeat the phone numbers once for each dependent:

(99999, David, 512-555-1234)
(99999, David, 512-555-4321)
(99999, William, 512-555-1234)
(99999, William, 512-555-4321)

If we did not repeat the phone numbers, and we stored only the first and last tuples, we would have recorded the dependent names and the phone numbers, but the resultant tuples would imply that David corresponded to 512-555-1234, while William corresponded to 512-555-4321. This would be incorrect.

Because normal forms based on functional dependencies are not sufficient to deal with situations like this, other dependencies and normal forms have been defined. We cover these in Section 7.6 and Section 7.7.

7.4 Functional-Dependency Theory

We have seen in our examples that it is useful to be able to reason systematically about functional dependencies as part of a process of testing schemas for BCNF or 3NF.

7.4.1 Closure of a Set of Functional Dependencies

We shall see that, given a set F of functional dependencies on a schema, we can prove that certain other functional dependencies also hold on the schema. We say that such functional dependencies are "logically implied" by F. When testing for normal forms, it is not sufficient to consider the given set of functional dependencies; rather, we need to consider *all* functional dependencies that hold on the schema.

More formally, given a relation schema $r(R)$, a functional dependency f on R is **logically implied** by a set of functional dependencies F on R if every instance of a relation $r(R)$ that satisfies F also satisfies f.

Suppose we are given a relation schema $r(A, B, C, G, H, I)$ and the set of functional dependencies:

$$A \rightarrow B$$
$$A \rightarrow C$$
$$CG \rightarrow H$$
$$CG \rightarrow I$$
$$B \rightarrow H$$

The functional dependency:

$$A \rightarrow H$$

is logically implied. That is, we can show that, whenever a relation instance satisfies our given set of functional dependencies, $A \rightarrow H$ must also be satisfied by that relation instance. Suppose that t_1 and t_2 are tuples such that:

$$t_1[A] = t_2[A]$$

Since we are given that $A \rightarrow B$, it follows from the definition of functional dependency that:

$$t_1[B] = t_2[B]$$

Then, since we are given that $B \rightarrow H$, it follows from the definition of functional dependency that:

$$t_1[H] = t_2[H]$$

Therefore, we have shown that, whenever t_1 and t_2 are tuples such that $t_1[A] = t_2[A]$, it must be that $t_1[H] = t_2[H]$. But that is exactly the definition of $A \rightarrow H$.

Let F be a set of functional dependencies. The **closure** of F, denoted by F^+, is the set of all functional dependencies logically implied by F. Given F, we can compute F^+ directly from the formal definition of functional dependency. If F were large, this process would be lengthy and difficult. Such a computation of F^+ requires arguments of the type just used to show that $A \rightarrow H$ is in the closure of our example set of dependencies.

Axioms, or rules of inference, provide a simpler technique for reasoning about functional dependencies. In the rules that follow, we use Greek letters (α, β, γ, ...) for sets of attributes and uppercase Roman letters from the beginning of the alphabet for individual attributes. We use $\alpha\beta$ to denote $\alpha \cup \beta$.

We can use the following three rules to find logically implied functional dependencies. By applying these rules *repeatedly*, we can find all of F^+, given F. This collection of rules is called **Armstrong's axioms** in honor of the person who first proposed it.

- **Reflexivity rule**. If α is a set of attributes and $\beta \subseteq \alpha$, then $\alpha \rightarrow \beta$ holds.
- **Augmentation rule**. If $\alpha \rightarrow \beta$ holds and γ is a set of attributes, then $\gamma\alpha \rightarrow \gamma\beta$ holds.
- **Transitivity rule**. If $\alpha \rightarrow \beta$ holds and $\beta \rightarrow \gamma$ holds, then $\alpha \rightarrow \gamma$ holds.

Armstrong's axioms are **sound**, because they do not generate any incorrect functional dependencies. They are **complete**, because, for a given set F of functional dependencies, they allow us to generate all F^+. The Further Reading section provides references for proofs of soundness and completeness.

Although Armstrong's axioms are complete, it is tiresome to use them directly for the computation of F^+. To simplify matters further, we list additional rules. It is possible to use Armstrong's axioms to prove that these rules are sound (see Practice Exercise 7.4, Practice Exercise 7.5, and Exercise 7.27).

- **Union rule**. If $\alpha \rightarrow \beta$ holds and $\alpha \rightarrow \gamma$ holds, then $\alpha \rightarrow \beta\gamma$ holds.
- **Decomposition rule**. If $\alpha \rightarrow \beta\gamma$ holds, then $\alpha \rightarrow \beta$ holds and $\alpha \rightarrow \gamma$ holds.
- **Pseudotransitivity rule**. If $\alpha \rightarrow \beta$ holds and $\gamma\beta \rightarrow \delta$ holds, then $\alpha\gamma \rightarrow \delta$ holds.

Let us apply our rules to the example of schema $R = (A, B, C, G, H, I)$ and the set F of functional dependencies $\{A \rightarrow B, A \rightarrow C, CG \rightarrow H, CG \rightarrow I, B \rightarrow H\}$. We list several members of F^+ here:

- $A \rightarrow H$. Since $A \rightarrow B$ and $B \rightarrow H$ hold, we apply the transitivity rule. Observe that it was much easier to use Armstrong's axioms to show that $A \rightarrow H$ holds than it was to argue directly from the definitions, as we did earlier in this section.
- $CG \rightarrow HI$. Since $CG \rightarrow H$ and $CG \rightarrow I$, the union rule implies that $CG \rightarrow HI$.

```
F+ = F
apply the reflexivity rule /* Generates all trivial dependencies */
repeat
    for each functional dependency f in F+
        apply the augmentation rule on f
        add the resulting functional dependencies to F+
    for each pair of functional dependencies f1 and f2 in F+
        if f1 and f2 can be combined using transitivity
            add the resulting functional dependency to F+
until F+ does not change any further
```

Figure 7.7 A procedure to compute F^+.

- $AG \rightarrow I$. Since $A \rightarrow C$ and $CG \rightarrow I$, the pseudotransitivity rule implies that $AG \rightarrow I$ holds.

  Another way of finding that $AG \rightarrow I$ holds is as follows: We use the augmentation rule on $A \rightarrow C$ to infer $AG \rightarrow CG$. Applying the transitivity rule to this dependency and $CG \rightarrow I$, we infer $AG \rightarrow I$.

Figure 7.7 shows a procedure that demonstrates formally how to use Armstrong's axioms to compute F^+. In this procedure, when a functional dependency is added to F^+, it may be already present, and in that case there is no change to F^+. We shall see an alternative way of computing F^+ in Section 7.4.2.

The left-hand and right-hand sides of a functional dependency are both subsets of R. Since a set of size n has 2^n subsets, there are a total of $2^n \times 2^n = 2^{2n}$ possible functional dependencies, where n is the number of attributes in R. Each iteration of the repeat loop of the procedure, except the last iteration, adds at least one functional dependency to F^+. Thus, the procedure is guaranteed to terminate, though it may be very lengthy.

7.4.2 Closure of Attribute Sets

We say that an attribute B is **functionally determined** by α if $\alpha \rightarrow B$. To test whether a set α is a superkey, we must devise an algorithm for computing the set of attributes functionally determined by α. One way of doing this is to compute F^+, take all functional dependencies with α as the left-hand side, and take the union of the right-hand sides of all such dependencies. However, doing so can be expensive, since F^+ can be large.

An efficient algorithm for computing the set of attributes functionally determined by α is useful not only for testing whether α is a superkey, but also for several other tasks, as we shall see later in this section.

Let α be a set of attributes. We call the set of all attributes functionally determined by α under a set F of functional dependencies the closure of α under F; we denote it by α^+. Figure 7.8 shows an algorithm, written in pseudocode, to compute α^+. The input is a set F of functional dependencies and the set α of attributes. The output is stored in the variable *result*.

To illustrate how the algorithm works, we shall use it to compute $(AG)^+$ with the functional dependencies defined in Section 7.4.1. We start with $result = AG$. The first time that we execute the **repeat** loop to test each functional dependency, we find that:

- $A \rightarrow B$ causes us to include B in *result*. To see this fact, we observe that $A \rightarrow B$ is in F, $A \subseteq result$ (which is AG), so $result := result \cup B$.
- $A \rightarrow C$ causes *result* to become $ABCG$.
- $CG \rightarrow H$ causes *result* to become $ABCGH$.
- $CG \rightarrow I$ causes *result* to become $ABCGHI$.

The second time that we execute the **repeat** loop, no new attributes are added to *result*, and the algorithm terminates.

Let us see why the algorithm of Figure 7.8 is correct. The first step is correct because $\alpha \rightarrow \alpha$ always holds (by the reflexivity rule). We claim that, for any subset β of *result*, $\alpha \rightarrow \beta$. Since we start the **repeat** loop with $\alpha \rightarrow result$ being true, we can add γ to *result* only if $\beta \subseteq result$ and $\beta \rightarrow \gamma$. But then $result \rightarrow \beta$ by the reflexivity rule, so $\alpha \rightarrow \beta$ by transitivity. Another application of transitivity shows that $\alpha \rightarrow \gamma$ (using $\alpha \rightarrow \beta$ and $\beta \rightarrow \gamma$). The union rule implies that $\alpha \rightarrow result \cup \gamma$, so α functionally determines any new result generated in the **repeat** loop. Thus, any attribute returned by the algorithm is in α^+.

It is easy to see that the algorithm finds all of α^+. Consider an attribute A in α^+ that is not yet in *result* at any point during the execution. There must be a way to prove that $result \rightarrow A$ using the axioms. Either $result \rightarrow A$ is in F itself (making the proof trivial and ensuring A is added to *result*) or there must a proof step using transitivity to show

```
result := α;
repeat
    for each functional dependency β → γ in F do
        begin
            if β ⊆ result then result := result ∪ γ;
        end
until (result does not change)
```

Figure 7.8 An algorithm to compute α^+, the closure of α under F.

for some attribute B that $result \rightarrow B$. If it happens that $A = B$, then we have shown that A is added to *result*. If not, $B \neq A$ is added. Then repeating this argument, we see that A must eventually be added to *result*.

It turns out that, in the worst case, this algorithm may take an amount of time quadratic in the size of F. There is a faster (although slightly more complex) algorithm that runs in time linear in the size of F; that algorithm is presented as part of Practice Exercise 7.8.

There are several uses of the attribute closure algorithm:

- To test if α is a superkey, we compute α^+ and check if α^+ contains all attributes in R.
- We can check if a functional dependency $\alpha \rightarrow \beta$ holds (or, in other words, is in F^+), by checking if $\beta \subseteq \alpha^+$. That is, we compute α^+ by using attribute closure, and then check if it contains β. This test is particularly useful, as we shall see later in this chapter.
- It gives us an alternative way to compute F^+: For each $\gamma \subseteq R$, we find the closure γ^+, and for each $S \subseteq \gamma^+$, we output a functional dependency $\gamma \rightarrow S$.

7.4.3 Canonical Cover

Suppose that we have a set of functional dependencies F on a relation schema. Whenever a user performs an update on the relation, the database system must ensure that the update does not violate any functional dependencies, that is, all the functional dependencies in F are satisfied in the new database state.

The system must roll back the update if it violates any functional dependencies in the set F.

We can reduce the effort spent in checking for violations by testing a simplified set of functional dependencies that has the same closure as the given set. Any database that satisfies the simplified set of functional dependencies also satisfies the original set, and vice versa, since the two sets have the same closure. However, the simplified set is easier to test. We shall see how the simplified set can be constructed in a moment. First, we need some definitions.

An attribute of a functional dependency is said to be **extraneous** if we can remove it without changing the closure of the set of functional dependencies.

- Removing an attribute from the left side of a functional dependency could make it a stronger constraint. For example, if we have $AB \rightarrow C$ and remove B, we get the possibly stronger result $A \rightarrow C$. It may be stronger because $A \rightarrow C$ logically implies $AB \rightarrow C$, but $AB \rightarrow C$ does not, on its own, logically imply $A \rightarrow C$. But, depending on what our set F of functional dependencies happens to be, we may be able to remove B from $AB \rightarrow C$ safely. For example, suppose that the set

$F = \{AB \rightarrow C, A \rightarrow D, D \rightarrow C\}$. Then we can show that F logically implies $A \rightarrow C$, making B extraneous in $AB \rightarrow C$.

- Removing an attribute from the right side of a functional dependency could make it a weaker constraint. For example, if we have $AB \rightarrow CD$ and remove C, we get the possibly weaker result $AB \rightarrow D$. It may be weaker because using just $AB \rightarrow D$, we can no longer infer $AB \rightarrow C$. But, depending on what our set F of functional dependencies happens to be, we may be able to remove C from $AB \rightarrow CD$ safely. For example, suppose that $F = \{AB \rightarrow CD, A \rightarrow C\}$. Then we can show that even after replacing $AB \rightarrow CD$ by $AB \rightarrow D$, we can still infer $AB \rightarrow C$ and thus $AB \rightarrow CD$.

The formal definition of **extraneous attributes** is as follows: Consider a set F of functional dependencies and the functional dependency $\alpha \rightarrow \beta$ in F.

- **Removal from the left side:** Attribute A is extraneous in α if $A \in \alpha$ and F logically implies $(F - \{\alpha \rightarrow \beta\}) \cup \{(\alpha - A) \rightarrow \beta\}$.
- **Removal from the right side:** Attribute A is extraneous in β if $A \in \beta$ and the set of functional dependencies $(F - \{\alpha \rightarrow \beta\}) \cup \{\alpha \rightarrow (\beta - A)\}$ logically implies F.

Beware of the direction of the implications when using the definition of extraneous attributes: If you reverse the statement, the implication will *always* hold. That is, $(F - \{\alpha \rightarrow \beta\}) \cup \{(\alpha - A) \rightarrow \beta\}$ always logically implies F, and also F always logically implies $(F - \{\alpha \rightarrow \beta\}) \cup \{\alpha \rightarrow (\beta - A)\}$.

Here is how we can test efficiently if an attribute is extraneous. Let R be the relation schema, and let F be the given set of functional dependencies that hold on R. Consider an attribute A in a dependency $\alpha \rightarrow \beta$.

- If $A \in \beta$, to check if A is extraneous, consider the set

  $$F' = (F - \{\alpha \rightarrow \beta\}) \cup \{\alpha \rightarrow (\beta - A)\}$$

  and check if $\alpha \rightarrow A$ can be inferred from F'. To do so, compute α^+ (the closure of α) under F'; if α^+ includes A, then A is extraneous in β.
- If $A \in \alpha$, to check if A is extraneous, let $\gamma = \alpha - \{A\}$, and check if $\gamma \rightarrow \beta$ can be inferred from F. To do so, compute γ^+ (the closure of γ) under F; if γ^+ includes all attributes in β, then A is extraneous in α.

For example, suppose F contains $AB \rightarrow CD$, $A \rightarrow E$, and $E \rightarrow C$. To check if C is extraneous in $AB \rightarrow CD$, we compute the attribute closure of AB under $F' = \{AB \rightarrow D$, $A \rightarrow E$, $E \rightarrow C\}$. The closure is $ABCDE$, which includes CD, so we infer that C is extraneous.

$F_c = F$
repeat
  Use the union rule to replace any dependencies in F_c of the form
    $\alpha_1 \rightarrow \beta_1$ and $\alpha_1 \rightarrow \beta_2$ with $\alpha_1 \rightarrow \beta_1 \beta_2$.
  Find a functional dependency $\alpha \rightarrow \beta$ in F_c with an extraneous
    attribute either in α or in β.
    /* Note: the test for extraneous attributes is done using F_c, not F */
  If an extraneous attribute is found, delete it from $\alpha \rightarrow \beta$ in F_c.
until (F_c does not change)

Figure 7.9 Computing canonical cover.

Having defined the concept of extraneous attributes, we can explain how we can construct a simplified set of functional dependencies equivalent to a given set of functional dependencies.

A **canonical cover** F_c for F is a set of dependencies such that F logically implies all dependencies in F_c, and F_c logically implies all dependencies in F. Furthermore, F_c must have the following properties:

- No functional dependency in F_c contains an extraneous attribute.
- Each left side of a functional dependency in F_c is unique. That is, there are no two dependencies $\alpha_1 \rightarrow \beta_1$ and $\alpha_2 \rightarrow \beta_2$ in F_c such that $\alpha_1 = \alpha_2$.

A canonical cover for a set of functional dependencies F can be computed as described in Figure 7.9. It is important to note that when checking if an attribute is extraneous, the check uses the dependencies in the current value of F_c, and **not** the dependencies in F. If a functional dependency contains only one attribute in its right-hand side, for example $A \rightarrow C$, and that attribute is found to be extraneous, we would get a functional dependency with an empty right-hand side. Such functional dependencies should be deleted.

Since the algorithm permits a choice of any extraneous attribute, it is possible that there may be several possible canonical covers for a given F. Any such F_c is equally acceptable. Any canonical cover of F, F_c, can be shown to have the same closure as F; hence, testing whether F_c is satisfied is equivalent to testing whether F is satisfied. However, F_c is minimal in a certain sense—it does not contain extraneous attributes, and it combines functional dependencies with the same left side. It is cheaper to test F_c than it is to test F itself.

We now consider an example. Assume we are given the following set F of functional dependencies on schema (A, B, C):

$$A \rightarrow BC$$
$$B \rightarrow C$$
$$A \rightarrow B$$
$$AB \rightarrow C$$

Let us compute a canonical cover for F.

- There are two functional dependencies with the same set of attributes on the left side of the arrow:

  $$A \rightarrow BC$$
  $$A \rightarrow B$$

  We combine these functional dependencies into $A \rightarrow BC$.
- A is extraneous in $AB \rightarrow C$ because F logically implies $(F - \{AB \rightarrow C\}) \cup \{B \rightarrow C\}$. This assertion is true because $B \rightarrow C$ is already in our set of functional dependencies.
- C is extraneous in $A \rightarrow BC$, since $A \rightarrow BC$ is logically implied by $A \rightarrow B$ and $B \rightarrow C$.

Thus, our canonical cover is:

$$A \rightarrow B$$
$$B \rightarrow C$$

Given a set F of functional dependencies, it may be that an entire functional dependency in the set is extraneous, in the sense that dropping it does not change the closure of F. We can show that a canonical cover F_c of F contains no such extraneous functional dependency. Suppose that, to the contrary, there were such an extraneous functional dependency in F_c. The right-side attributes of the dependency would then be extraneous, which is not possible by the definition of canonical covers.

As we noted earlier, a canonical cover might not be unique. For instance, consider the set of functional dependencies $F = \{A \rightarrow BC, B \rightarrow AC$, and $C \rightarrow AB\}$. If we apply the test for extraneous attributes to $A \rightarrow BC$, we find that both B and C are extraneous under F. However, it is incorrect to delete both! The algorithm for finding the canonical cover picks one of the two and deletes it. Then,

1. If C is deleted, we get the set $F' = \{A \rightarrow B, B \rightarrow AC$, and $C \rightarrow AB\}$. Now, B is not extraneous on the right side of $A \rightarrow B$ under F'. Continuing the algorithm, we find A and B are extraneous in the right side of $C \rightarrow AB$, leading to two choices of canonical cover:

```
compute F+;
for each schema Ri in D do
    begin
        Fi := the restriction of F+ to Ri;
    end
F' := ∅
for each restriction Fi do
    begin
        F' = F' ∪ Fi
    end
compute F'+;
if (F'+ = F+) then return (true)
                else return (false);
```

Figure 7.10 Testing for dependency preservation.

$$F_c = \{A \rightarrow B, B \rightarrow C, C \rightarrow A\}$$
$$F_c = \{A \rightarrow B, B \rightarrow AC, C \rightarrow B\}.$$

2. If B is deleted, we get the set $\{A \rightarrow C, B \rightarrow AC$, and $C \rightarrow AB\}$. This case is symmetrical to the previous case, leading to two more choices of canonical cover:

$$F_c = \{A \rightarrow C, C \rightarrow B, \text{and } B \rightarrow A\}$$
$$F_c = \{A \rightarrow C, B \rightarrow C, \text{and } C \rightarrow AB\}.$$

As an exercise, can you find one more canonical cover for F?

7.4.4 Dependency Preservation

Using the theory of functional dependencies, there is a way to describe dependency preservation that is simpler than the ad hoc approach we used in Section 7.3.1.2.

Let F be a set of functional dependencies on a schema R, and let $R_1, R_2, \ldots, R_n$ be a decomposition of R. The **restriction of F to R_i** is the set F_i of all functional dependencies in F^+ that include *only* attributes of R_i. Since all functional dependencies in a restriction involve attributes of only one relation schema, it is possible to test such a dependency for satisfaction by checking only one relation.

Note that the definition of restriction uses all dependencies in F^+, not just those in F. For instance, suppose $F = \{A \rightarrow B, B \rightarrow C\}$, and we have a decomposition into AC and AB. The restriction of F to AC includes $A \rightarrow C$, since $A \rightarrow C$ is in F^+, even though it is not in F.

The set of restrictions $F_1, F_2, \ldots, F_n$ is the set of dependencies that can be checked efficiently. We now must ask whether testing only the restrictions is sufficient. Let $F' = F_1 \cup F_2 \cup \cdots \cup F_n$. F' is a set of functional dependencies on schema R, but, in general, $F' \neq F$. However, even if $F' \neq F$, it may be that $F'^+ = F^+$. If the latter is true, then every dependency in F is logically implied by F', and, if we verify that F' is satisfied, we have verified that F is satisfied. We say that a decomposition having the property $F'^+ = F^+$ is a **dependency-preserving decomposition**.

Figure 7.10 shows an algorithm for testing dependency preservation. The input is a set $D = \{R_1, R_2, \ldots, R_n\}$ of decomposed relation schemas, and a set F of functional dependencies. This algorithm is expensive since it requires computation of F^+. Instead of applying the algorithm of Figure 7.10, we consider two alternatives.

First, note that if each member of F can be tested on one of the relations of the decomposition, then the decomposition is dependency preserving. This is an easy way to show dependency preservation; however, it does not always work. There are cases where, even though the decomposition is dependency preserving, there is a dependency in F that cannot be tested in any one relation in the decomposition. Thus, this alternative test can be used only as a sufficient condition that is easy to check; if it fails we cannot conclude that the decomposition is not dependency preserving; instead we will have to apply the general test.

We now give a second alternative test for dependency preservation that avoids computing F^+. We explain the intuition behind the test after presenting the test. The test applies the following procedure to each $\alpha \rightarrow \beta$ in F.

```
result = α
repeat
    for each R_i in the decomposition
        t = (result ∩ R_i)^+ ∩ R_i
        result = result ∪ t
until (result does not change)
```

The attribute closure here is under the set of functional dependencies F. If *result* contains all attributes in β, then the functional dependency $\alpha \rightarrow \beta$ is preserved. The decomposition is dependency preserving if and only if the procedure shows that all the dependencies in F are preserved.

The two key ideas behind the preceding test are as follows:

- The first idea is to test each functional dependency $\alpha \rightarrow \beta$ in F to see if it is preserved in F' (where F' is as defined in Figure 7.10). To do so, we compute the closure of α under F'; the dependency is preserved exactly when the closure includes β. The decomposition is dependency preserving if (and only if) all the dependencies in F are found to be preserved.

- The second idea is to use a modified form of the attribute-closure algorithm to compute closure under F', without actually first computing F'. We wish to avoid computing F' since computing it is quite expensive. Note that F' is the union of all F_i, where F_i is the restriction of F on R_i. The algorithm computes the attribute closure of $(result \cap R_i)$ with respect to F, intersects the closure with R_i, and adds the resultant set of attributes to *result*; this sequence of steps is equivalent to computing the closure of *result* under F_i. Repeating this step for each i inside the while loop gives the closure of *result* under F'.

  To understand why this modified attribute-closure approach works correctly, we note that for any $\gamma \subseteq R_i$, $\gamma \rightarrow \gamma^+$ is a functional dependency in F^+, and $\gamma \rightarrow \gamma^+ \cap R_i$ is a functional dependency that is in F_i, the restriction of F^+ to R_i. Conversely, if $\gamma \rightarrow \delta$ were in F_i, then δ would be a subset of $\gamma^+ \cap R_i$.

This test takes polynomial time, instead of the exponential time required to compute F^+.

7.5 Algorithms for Decomposition Using Functional Dependencies

Real-world database schemas are much larger than the examples that fit in the pages of a book. For this reason, we need algorithms for the generation of designs that are in appropriate normal form. In this section, we present algorithms for BCNF and 3NF.

7.5.1 BCNF Decomposition

The definition of BCNF can be used directly to test if a relation is in BCNF. However, computation of F^+ can be a tedious task. We first describe simplified tests for verifying if a relation is in BCNF. If a relation is not in BCNF, it can be decomposed to create relations that are in BCNF. Later in this section, we describe an algorithm to create a lossless decomposition of a relation, such that the decomposition is in BCNF.

7.5.1.1 Testing for BCNF

Testing of a relation schema R to see if it satisfies BCNF can be simplified in some cases:

- To check if a nontrivial dependency $\alpha \rightarrow \beta$ causes a violation of BCNF, compute α^+ (the attribute closure of α), and verify that it includes all attributes of R; that is, it is a superkey for R.
- To check if a relation schema R is in BCNF, it suffices to check only the dependencies in the given set F for violation of BCNF, rather than check all dependencies in F^+.

  We can show that if none of the dependencies in F causes a violation of BCNF, then none of the dependencies in F^+ will cause a violation of BCNF, either.

```
result := {R};
done := false;
while (not done) do
   if (there is a schema R_i in result that is not in BCNF)
     then begin
             let α → β be a nontrivial functional dependency that holds
             on R_i such that α+ does not contain R_i and α ∩ β = ∅;
             result := (result − R_i) ∪ (R_i − β) ∪ ( α, β);
          end
     else done := true;
```

Figure 7.11 BCNF decomposition algorithm.

Unfortunately, the latter procedure does not work when a relation schema is decomposed. That is, it *does not* suffice to use F when we test a relation schema R_i, in a decomposition of R, for violation of BCNF. For example, consider relation schema (A, B, C, D, E), with functional dependencies F containing $A \rightarrow B$ and $BC \rightarrow D$. Suppose this were decomposed into (A, B) and (A, C, D, E). Now, neither of the dependencies in F contains only attributes from (A, C, D, E), so we might be misled into thinking that it is in BCNF. In fact, there is a dependency $AC \rightarrow D$ in F^+ (which can be inferred using the pseudotransitivity rule from the two dependencies in F) that shows that (A, C, D, E) is not in BCNF. Thus, we may need a dependency that is in F^+, but is not in F, to show that a decomposed relation is not in BCNF.

An alternative BCNF test is sometimes easier than computing every dependency in F^+. To check if a relation schema R_i in a decomposition of R is in BCNF, we apply this test:

- For every subset α of attributes in R_i, check that α^+ (the attribute closure of α under F) either includes no attribute of $R_i - \alpha$, or includes all attributes of R_i.

If the condition is violated by some set of attributes α in R_i, consider the following functional dependency, which can be shown to be present in F^+:

$$\alpha \rightarrow (\alpha^+ - \alpha) \cap R_i.$$

This dependency shows that R_i violates BCNF.

7.5.1.2 BCNF Decomposition Algorithm

We are now able to state a general method to decompose a relation schema so as to satisfy BCNF. Figure 7.11 shows an algorithm for this task. If R is not in BCNF, we can decompose R into a collection of BCNF schemas $R_1, R_2, \ldots, R_n$ by the algorithm.

The algorithm uses dependencies that demonstrate violation of BCNF to perform the decomposition.

The decomposition that the algorithm generates is not only in BCNF, but is also a lossless decomposition. To see why our algorithm generates only lossless decompositions, we note that, when we replace a schema R_i with $(R_i - \beta)$ and (α, β), the dependency $\alpha \rightarrow \beta$ holds, and $(R_i - \beta) \cap (\alpha, \beta) = \alpha$.

If we did not require $\alpha \cap \beta = \emptyset$, then those attributes in $\alpha \cap \beta$ would not appear in the schema $(R_i - \beta)$, and the dependency $\alpha \rightarrow \beta$ would no longer hold.

It is easy to see that our decomposition of *in_dep* in Section 7.3.1 would result from applying the algorithm. The functional dependency *dept_name* → *building*, *budget* satisfies the $\alpha \cap \beta = \emptyset$ condition and would therefore be chosen to decompose the schema.

The **BCNF decomposition algorithm** takes time exponential to the size of the initial schema, since the algorithm for checking whether a relation in the decomposition satisfies BCNF can take exponential time. There is an algorithm that can compute a BCNF decomposition in polynomial time; however, the algorithm may "overnormalize," that is, decompose a relation unnecessarily.

As a longer example of the use of the BCNF decomposition algorithm, suppose we have a database design using the *class* relation, whose schema is as shown below:

class (*course_id*, *title*, *dept_name*, *credits*, *sec_id*, *semester*, *year*, *building*, *room_number*, *capacity*, *time_slot_id*)

The set of functional dependencies that we need to hold on this schema are:

course_id → *title*, *dept_name*, *credits*
building, *room_number* → *capacity*
course_id, *sec_id*, *semester*, *year*→ *building*, *room_number*, *time_slot_id*

A candidate key for this schema is {*course_id*, *sec_id*, *semester*, *year*}.

We can apply the algorithm of Figure 7.11 to the *class* example as follows:

- The functional dependency:

  course_id → *title*, *dept_name*, *credits*

  holds, but *course_id* is not a superkey. Thus, *class* is not in BCNF. We replace *class* with two relations with the following schemas:

  course (*course_id*, *title*, *dept_name*, *credits*)
  class-1 (*course_id*, *sec_id*, *semester*, *year*, *building*, *room_number* *capacity*, *time_slot_id*)

The only nontrivial functional dependencies that hold on *course* include *course_id* on the left side of the arrow. Since *course_id* is a superkey for *course*, *course* is in BCNF.

- A candidate key for *class-1* is {*course_id*, *sec_id*, *semester*, *year*}. The functional dependency:

$$building, room\_number \rightarrow capacity$$

holds on *class-1*, but {*building*, *room_number*} is not a superkey for *class-1*. We replace *class-1* two relations with the following schemas:

$$classroom\ (building, room\_number, capacity)$$
$$section\ (course\_id, sec\_id, semester, year, building, room\_number, time\_slot\_id)$$

These two schemas are in BCNF.

Thus, the decomposition of *class* results in the three relation schemas *course*, *classroom*, and *section*, each of which is in BCNF. These correspond to the schemas that we have used in this and previous chapters. You can verify that the decomposition is lossless and dependency preserving.

7.5.2 3NF Decomposition

Figure 7.12 shows an algorithm for finding a dependency-preserving, lossless decomposition into 3NF. The set of dependencies F_c used in the algorithm is a canonical cover for F. Note that the algorithm considers the set of schemas R_j, $j = 1, 2, \ldots, i$; initially $i = 0$, and in this case the set is empty.

Let us apply this algorithm to our example of *dept_advisor* from Section 7.3.2, where we showed that:

$$dept\_advisor\ (s\_ID, i\_ID, dept\_name)$$

is in 3NF even though it is not in BCNF. The algorithm uses the following functional dependencies in F:

$$f_1: i\_ID \rightarrow dept\_name$$
$$f_2: s\_ID, dept\_name \rightarrow i\_ID$$

There are no extraneous attributes in any of the functional dependencies in F, so F_c contains f_1 and f_2. The algorithm then generates as R_1 the schema, (*i_ID dept_name*), and as R_2 the schema (*s_ID*, *dept_name*, *i_ID*). The algorithm then finds that R_2 contains a candidate key, so no further relation schema is created.

```
let F_c be a canonical cover for F;
i := 0;
for each functional dependency α → β in F_c
    i := i + 1;
    R_i := α β;
if none of the schemas R_j, j = 1, 2, ..., i contains a candidate key for R
  then
    i := i + 1;
    R_i := any candidate key for R;
/* Optionally, remove redundant relations */
repeat
    if any schema R_j is contained in another schema R_k
      then
        /* Delete R_j */
        R_j := R_i;
        i := i - 1;
until no more R_j s can be deleted
return (R_1, R_2, ..., R_i)
```

Figure 7.12 Dependency-preserving, lossless decomposition into 3NF.

The resultant set of schemas can contain redundant schemas, with one schema R_k containing all the attributes of another schema R_j. For example, R_2 above contains all the attributes from R_1. The algorithm deletes all such schemas that are contained in another schema. Any dependencies that could be tested on an R_j that is deleted can also be tested on the corresponding relation R_k, and the decomposition is lossless even if R_j is deleted.

Now let us consider again the schema of the *class* relation of Section 7.5.1.2 and apply the **3NF decomposition algorithm**. The set of functional dependencies we listed there happen to be a canonical cover. As a result, the algorithm gives us the same three schemas *course, classroom*, and *section*.

The preceding example illustrates an interesting property of the 3NF algorithm. Sometimes, the result is not only in 3NF, but also in BCNF. This suggests an alternative method of generating a BCNF design. First use the 3NF algorithm. Then, for any schema in the 3NF design that is not in BCNF, decompose using the BCNF algorithm. If the result is not dependency-preserving, revert to the 3NF design.

7.5.3 Correctness of the 3NF Algorithm

The 3NF algorithm ensures the preservation of dependencies by explicitly building a schema for each dependency in a canonical cover. It ensures that the decomposition is a

lossless decomposition by guaranteeing that at least one schema contains a candidate key for the schema being decomposed. Practice Exercise 7.16 provides some insight into the proof that this suffices to guarantee a lossless decomposition.

This algorithm is also called the **3NF synthesis algorithm**, since it takes a set of dependencies and adds one schema at a time, instead of decomposing the initial schema repeatedly. The result is not uniquely defined, since a set of functional dependencies can have more than one canonical cover. The algorithm may decompose a relation even if it is already in 3NF; however, the decomposition is still guaranteed to be in 3NF.

To see that the algorithm produces a 3NF design, consider a schema R_i in the decomposition. Recall that when we test for 3NF it suffices to consider functional dependencies whose right-hand side consists of a single attribute. Therefore, to see that R_i is in 3NF you must convince yourself that any functional dependency $\gamma \rightarrow B$ that holds on R_i satisfies the definition of 3NF. Assume that the dependency that generated R_i in the synthesis algorithm is $\alpha \rightarrow \beta$. B must be in α or β, since B is in R_i and $\alpha \rightarrow \beta$ generated R_i. Let us consider the three possible cases:

- B is in both α and β. In this case, the dependency $\alpha \rightarrow \beta$ would not have been in F_c since B would be extraneous in β. Thus, this case cannot hold.
- B is in β but not α. Consider two cases:
  - γ is a superkey. The second condition of 3NF is satisfied.
  - γ is not a superkey. Then α must contain some attribute not in γ. Now, since $\gamma \rightarrow B$ is in F^+, it must be derivable from F_c by using the attribute closure algorithm on γ. The derivation could not have used $\alpha \rightarrow \beta$, because if it had been used, α must be contained in the attribute closure of γ, which is not possible, since we assumed γ is not a superkey. Now, using $\alpha \rightarrow (\beta - \{B\})$ and $\gamma \rightarrow B$, we can derive $\alpha \rightarrow B$ (since $\gamma \subseteq \alpha\beta$, and γ cannot contain B because $\gamma \rightarrow B$ is nontrivial). This would imply that B is extraneous in the right-hand side of $\alpha \rightarrow \beta$, which is not possible since $\alpha \rightarrow \beta$ is in the canonical cover F_c. Thus, if B is in β, then γ must be a superkey, and the second condition of 3NF must be satisfied.
- B is in α but not β.
  Since α is a candidate key, the third alternative in the definition of 3NF is satisfied.

Interestingly, the algorithm we described for decomposition into 3NF can be implemented in polynomial time, even though testing a given schema to see if it satisfies 3NF is NP-hard (which means that it is very unlikely that a polynomial-time algorithm will ever be invented for this task).

7.6 Decomposition Using Multivalued Dependencies

Some relation schemas, even though they are in BCNF, do not seem to be sufficiently normalized, in the sense that they still suffer from the problem of repetition of information. Consider a variation of the university organization where an instructor may be associated with multiple departments, and we have a relation:

$$inst\ (ID, dept\_name, name, street, city)$$

The astute reader will recognize this schema as a non-BCNF schema because of the functional dependency

$$ID \rightarrow name, street, city$$

and because *ID* is not a key for *inst*.

Further assume that an instructor may have several addresses (say, a winter home and a summer home). Then, we no longer wish to enforce the functional dependency "$ID \rightarrow street, city$", though, we still want to enforce "$ID \rightarrow name$" (i.e., the university is not dealing with instructors who operate under multiple aliases!). Following the BCNF decomposition algorithm, we obtain two schemas:

$$r_1\ (ID, name)$$
$$r_2\ (ID, dept\_name, street, city)$$

Both of these are in BCNF (recall that an instructor can be associated with multiple departments and a department may have several instructors, and therefore, neither "$ID \rightarrow dept\_name$" nor "$dept\_name \rightarrow ID$" hold).

Despite r_2 being in BCNF, there is redundancy. We repeat the address information of each residence of an instructor once for each department with which the instructor is associated. We could solve this problem by decomposing r_2 further into:

$$r_{21}\ (dept\_name, ID)$$
$$r_{22}\ (ID, street, city)$$

but there is no constraint that leads us to do this.

To deal with this problem, we must define a new form of constraint, called a *multivalued dependency*. As we did for functional dependencies, we shall use multivalued dependencies to define a normal form for relation schemas. This normal form, called **fourth normal form** (4NF), is more restrictive than BCNF. We shall see that every 4NF schema is also in BCNF but there are BCNF schemas that are not in 4NF.

	α	β	$R - \alpha - \beta$
t_1	$a_1 \ldots a_i$	$a_{i+1} \ldots a_j$	$a_{j+1} \ldots a_n$
t_2	$a_1 \ldots a_i$	$b_{i+1} \ldots b_j$	$b_{j+1} \ldots b_n$
t_3	$a_1 \ldots a_i$	$a_{i+1} \ldots a_j$	$b_{j+1} \ldots b_n$
t_4	$a_1 \ldots a_i$	$b_{i+1} \ldots b_j$	$a_{j+1} \ldots a_n$

Figure 7.13 Tabular representation of $\alpha \twoheadrightarrow \beta$.

7.6.1 Multivalued Dependencies

Functional dependencies rule out certain tuples from being in a relation. If $A \rightarrow B$, then we cannot have two tuples with the same A value but different B values. Multivalued dependencies, on the other hand, do not rule out the existence of certain tuples. Instead, they *require* that other tuples of a certain form be present in the relation. For this reason, functional dependencies sometimes are referred to as **equality-generating dependencies**, and multivalued dependencies are referred to as **tuple-generating dependencies**.

Let $r(R)$ be a relation schema and let $\alpha \subseteq R$ and $\beta \subseteq R$. The **multivalued dependency**

$$\alpha \twoheadrightarrow \beta$$

holds on R if, in any legal instance of relation $r(R)$, for all pairs of tuples t_1 and t_2 in r such that $t_1[\alpha] = t_2[\alpha]$, there exist tuples t_3 and t_4 in r such that

$$\begin{aligned} t_1[\alpha] &= t_2[\alpha] = t_3[\alpha] = t_4[\alpha] \\ t_3[\beta] &= t_1[\beta] \\ t_3[R - \beta] &= t_2[R - \beta] \\ t_4[\beta] &= t_2[\beta] \\ t_4[R - \beta] &= t_1[R - \beta] \end{aligned}$$

This definition is less complicated than it appears to be. Figure 7.13 gives a tabular picture of t_1, t_2, t_3, and t_4. Intuitively, the multivalued dependency $\alpha \twoheadrightarrow \beta$ says that the relationship between α and β is independent of the relationship between α and $R - \beta$. If the multivalued dependency $\alpha \twoheadrightarrow \beta$ is satisfied by all relations on schema R, then $\alpha \twoheadrightarrow \beta$ is a *trivial* multivalued dependency on schema R. Thus, $\alpha \twoheadrightarrow \beta$ is trivial if $\beta \subseteq \alpha$ or $\beta \cup \alpha = R$. This can be seen by looking at Figure 7.13 and considering the two special cases $\beta \subseteq \alpha$ and $\beta \cup \alpha = R$. In each case, the table reduces to just two columns and we see that t_1 and t_2 are able to serve in the roles of t_3 and t_4.

To illustrate the difference between functional and multivalued dependencies, we consider the schema r_2 again, and an example relation on that schema is shown in Figure 7.14. We must repeat the department name once for each address that an instructor has, and we must repeat the address for each department with which an instructor is associated. This repetition is unnecessary, since the relationship between an instructor

ID	*dept_name*	*street*	*city*
22222	Physics	North	Rye
22222	Physics	Main	Manchester
12121	Finance	Lake	Horseneck

Figure 7.14 An example of redundancy in a relation on a BCNF schema.

and his address is independent of the relationship between that instructor and a department. If an instructor with *ID* 22222 is associated with the Physics department, we want that department to be associated with all of that instructor's addresses. Thus, the relation of Figure 7.15 is illegal. To make this relation legal, we need to add the tuples (Physics, 22222, Main, Manchester) and (Math, 22222, North, Rye) to the relation of Figure 7.15.

Comparing the preceding example with our definition of multivalued dependency, we see that we want the multivalued dependency:

$$ID \twoheadrightarrow street, city$$

to hold. (The multivalued dependency $ID \twoheadrightarrow dept\_name$ will do as well. We shall soon see that they are equivalent.)

As with functional dependencies, we shall use multivalued dependencies in two ways:

1. To test relations to determine whether they are legal under a given set of functional and multivalued dependencies.
2. To specify constraints on the set of legal relations; we shall thus concern ourselves with *only* those relations that satisfy a given set of functional and multivalued dependencies.

Note that, if a relation r fails to satisfy a given multivalued dependency, we can construct a relation r' that *does* satisfy the multivalued dependency by adding tuples to r.

Let D denote a set of functional and multivalued dependencies. The closure D^+ of D is the set of all functional and multivalued dependencies logically implied by D. As we did for functional dependencies, we can compute D^+ from D, using the formal definitions of functional dependencies and multivalued dependencies. We can manage

ID	*dept_name*	*street*	*city*
22222	Physics	North	Rye
22222	Math	Main	Manchester

Figure 7.15 An illegal r_2 relation.

with such reasoning for very simple multivalued dependencies. Luckily, multivalued dependencies that occur in practice appear to be quite simple. For complex dependencies, it is better to reason about sets of dependencies by using a system of inference rules.

From the definition of multivalued dependency, we can derive the following rules for $\alpha, \beta \subseteq R$:

- If $\alpha \rightarrow \beta$, then $\alpha \twoheadrightarrow \beta$. In other words, every functional dependency is also a multivalued dependency.
- If $\alpha \twoheadrightarrow \beta$, then $\alpha \twoheadrightarrow R - \alpha - \beta$

Section 28.1.1 outlines a system of inference rules for multivalued dependencies.

7.6.2 Fourth Normal Form

Consider again our example of the BCNF schema:

$$r_2 \ (ID, dept\_name, street, city)$$

in which the multivalued dependency $ID \twoheadrightarrow street, city$ holds. We saw in the opening paragraphs of Section 7.6 that, although this schema is in BCNF, the design is not ideal, since we must repeat an instructor's address information for each department. We shall see that we can use the given multivalued dependency to improve the database design by decomposing this schema into a **fourth normal form** decomposition.

A relation schema R is in **fourth normal form** (**4NF**) with respect to a set D of functional and multivalued dependencies if, for all multivalued dependencies in D^+ of the form $\alpha \twoheadrightarrow \beta$, where $\alpha \subseteq R$ and $\beta \subseteq R$, at least one of the following holds:

- $\alpha \twoheadrightarrow \beta$ is a trivial multivalued dependency.
- α is a superkey for R.

A database design is in 4NF if each member of the set of relation schemas that constitutes the design is in 4NF.

Note that the definition of 4NF differs from the definition of BCNF in only the use of multivalued dependencies. Every 4NF schema is in BCNF. To see this fact, we note that, if a schema R is not in BCNF, then there is a nontrivial functional dependency $\alpha \rightarrow \beta$ holding on R, where α is not a superkey. Since $\alpha \rightarrow \beta$ implies $\alpha \twoheadrightarrow \beta$, R cannot be in 4NF.

Let R be a relation schema, and let $R_1, R_2, \ldots, R_n$ be a decomposition of R. To check if each relation schema R_i in the decomposition is in 4NF, we need to find what multivalued dependencies hold on each R_i. Recall that, for a set F of functional dependencies, the restriction F_i of F to R_i is all functional dependencies in F^+ that include *only* attributes of R_i. Now consider a set D of both functional and multivalued dependencies. The **restriction** of D to R_i is the set D_i consisting of:

1. All functional dependencies in D^+ that include only attributes of R_i.
2. All multivalued dependencies of the form:

$$\alpha \twoheadrightarrow \beta \cap R_i$$

where $\alpha \subseteq R_i$ and $\alpha \twoheadrightarrow \beta$ is in D^+.

7.6.3 4NF Decomposition

The analogy between 4NF and BCNF applies to the algorithm for decomposing a schema into 4NF. Figure 7.16 shows the 4NF decomposition algorithm. It is identical to the BCNF decomposition algorithm of Figure 7.11, except that it uses multivalued dependencies and uses the restriction of D^+ to R_i.

If we apply the algorithm of Figure 7.16 to (*ID*, *dept_name*, *street*, *city*), we find that *ID*$\twoheadrightarrow$ *dept_name* is a nontrivial multivalued dependency, and *ID* is not a superkey for the schema. Following the algorithm, we replace it with two schemas:

(*ID*, *dept_name*)
(*ID*, *street*, *city*)

This pair of schemas, which is in 4NF, eliminates the redundancy we encountered earlier.

As was the case when we were dealing solely with functional dependencies, we are interested in decompositions that are lossless and that preserve dependencies. The following fact about multivalued dependencies and losslessness shows that the algorithm of Figure 7.16 generates only lossless decompositions:

---

result := {*R*};
done := false;
compute D^+; Given schema R_i, let D_i denote the restriction of D^+ to R_i
while (**not** *done*) **do**
  if (there is a schema R_i in *result* that is not in 4NF w.r.t. D_i)
    then begin
      let $\alpha \twoheadrightarrow \beta$ be a nontrivial multivalued dependency that holds
      on R_i such that $\alpha \rightarrow R_i$ is not in D_i, and $\alpha \cap \beta = \emptyset$;
      result := (*result* − R_i) ∪ (R_i − β) ∪ (α, β);
    end
  else *done* := true;

Figure 7.16 4NF decomposition algorithm.

- Let $r(R)$ be a relation schema, and let D be a set of functional and multivalued dependencies on R. Let $r_1(R_1)$ and $r_2(R_2)$ form a decomposition of R. This decomposition of R is lossless if and only if at least one of the following multivalued dependencies is in D^+:

$$R_1 \cap R_2 \twoheadrightarrow R_1$$
$$R_1 \cap R_2 \twoheadrightarrow R_2$$

Recall that we stated in Section 7.2.3 that, if $R_1 \cap R_2 \rightarrow R_1$ or $R_1 \cap R_2 \rightarrow R_2$, then $r_1(R_1)$ and $r_2(R_2)$ forms a lossless decomposition of $r(R)$. The preceding fact about multivalued dependencies is a more general statement about losslessness. It says that, for *every* lossless decomposition of $r(R)$ into two schemas $r_1(R_1)$ and $r_2(R_2)$, one of the two dependencies $R_1 \cap R_2 \twoheadrightarrow R_1$ or $R_1 \cap R_2 \twoheadrightarrow R_2$ must hold. To see that this is true, we need to show first that if at least one of these dependencies holds, then $\Pi_{R_1}(r) \bowtie \Pi_{R_2}(r) = r$ and next we need to show that if $\Pi_{R_1}(r) \bowtie \Pi_{R_2}(r) = r$ then $r(R)$ must satisfy at least one of these dependencies. See the Further Reading section for references to a full proof.

The issue of dependency preservation when we decompose a relation schema becomes more complicated in the presence of multivalued dependencies. Section 28.1.2 pursues this topic.

A further complication arises from the fact that it is possible for a multivalued dependency to hold only on a proper subset of the given schema, with no way to express that multivalued dependency on that given schema. Such a multivalued dependency may appear as the result of a decomposition. Fortunately, such cases, called **embedded multivalued dependencies**, are rare. See the Further Reading section for details.

7.7 More Normal Forms

The fourth normal form is by no means the "ultimate" normal form. As we saw earlier, multivalued dependencies help us understand and eliminate some forms of repetition of information that cannot be understood in terms of functional dependencies. There are types of constraints called **join dependencies** that generalize multivalued dependencies and lead to another normal form called **project-join normal form (PJNF)**. PJNF is called **fifth normal form** in some books. There is a class of even more general constraints that leads to a normal form called **domain-key normal form (DKNF)**.

A practical problem with the use of these generalized constraints is that they are not only hard to reason with, but there is also no set of sound and complete inference rules for reasoning about the constraints. Hence PJNF and DKNF are used quite rarely. Chapter 28 provides more details about these normal forms.

Conspicuous by its absence from our discussion of normal forms is **second normal form** (2NF). We have not discussed it because it is of historical interest only. We simply

define it and let you experiment with it in Practice Exercise 7.19. First normal form deals with a different issue than the normal forms we have seen so far. It is discussed in the next section.

7.8 Atomic Domains and First Normal Form

The E-R model allows entity sets and relationship sets to have attributes that have some degree of substructure. Specifically, it allows multivalued attributes such as *phone_number* in Figure 6.8 and composite attributes (such as an attribute *address* with component attributes *street*, *city*, and *state*). When we create tables from E-R designs that contain these types of attributes, we eliminate this substructure. For composite attributes, we let each component be an attribute in its own right. For multivalued attributes, we create one tuple for each item in a multivalued set.

In the relational model, we formalize this idea that attributes do not have any substructure. A domain is **atomic** if elements of the domain are considered to be indivisible units. We say that a relation schema R is in **first normal form** (**1NF**) if the domains of all attributes of R are atomic.

A set of names is an example of a non-atomic value. For example, if the schema of a relation *employee* included an attribute *children* whose domain elements are sets of names, the schema would not be in first normal form.

Composite attributes, such as an attribute *address* with component attributes *street* and *city* also have non-atomic domains.

Integers are assumed to be atomic, so the set of integers is an atomic domain; however, the set of all sets of integers is a non-atomic domain. The distinction is that we do not normally consider integers to have subparts, but we consider sets of integers to have subparts—namely, the integers making up the set. But the important issue is not what the domain itself is, but rather how we use domain elements in our database. The domain of all integers would be non-atomic if we considered each integer to be an ordered list of digits.

As a practical illustration of this point, consider an organization that assigns employees identification numbers of the following form: The first two letters specify the department and the remaining four digits are a unique number within the department for the employee. Examples of such numbers would be "CS001" and "EE1127". Such identification numbers can be divided into smaller units and are therefore non-atomic. If a relation schema had an attribute whose domain consists of identification numbers encoded as above, the schema would not be in first normal form.

When such identification numbers are used, the department of an employee can be found by writing code that breaks up the structure of an identification number. Doing so requires extra programming, and information gets encoded in the application program rather than in the database. Further problems arise if such identification numbers are used as primary keys: When an employee changes departments, the employee's identification number must be changed everywhere it occurs, which can be a difficult task, or the code that interprets the number would give a wrong result.

From this discussion, it may appear that our use of course identifiers such as "CS-101", where "CS" indicates the Computer Science department, means that the domain of course identifiers is not atomic. Such a domain is not atomic as far as humans using the system are concerned. However, the database application still treats the domain as atomic, as long as it does not attempt to split the identifier and interpret parts of the identifier as a department abbreviation. The *course* schema stores the department name as a separate attribute, and the database application can use this attribute value to find the department of a course, instead of interpreting particular characters of the course identifier. Thus, our university schema can be considered to be in first normal form.

The use of set-valued attributes can lead to designs with redundant storage of data, which in turn can result in inconsistencies. For instance, instead of having the relationship between instructors and sections being represented as a separate relation *teaches*, a database designer may be tempted to store a set of course section identifiers with each instructor and a set of instructor identifiers with each section. (The primary keys of *section* and *instructor* are used as identifiers.) Whenever data pertaining to which instructor teaches which section is changed, the update has to be performed at two places: in the set of instructors for the section, and in the set of sections for the instructor. Failure to perform both updates can leave the database in an inconsistent state. Keeping only one of these sets would avoid repeated information; however keeping only one of these would complicate some queries, and it is unclear which of the two it would be better to retain.

Some types of non-atomic values can be useful, although they should be used with care. For example, composite-valued attributes are often useful, and set-valued attributes are also useful in many cases, which is why both are supported in the E-R model. In many domains where entities have a complex structure, forcing a first normal form representation represents an unnecessary burden on the application programmer, who has to write code to convert data into atomic form. There is also the runtime overhead of converting data back and forth from the atomic form. Support for non-atomic values can thus be very useful in such domains. In fact, modern database systems do support many types of non-atomic values, as we shall see in Chapter 29 restrict ourselves to relations in first normal form, and thus all domains are atomic.

7.9 Database-Design Process

So far we have looked at detailed issues about normal forms and normalization. In this section, we study how normalization fits into the overall database-design process.

Earlier in the chapter starting in Section 7.1.1, we assumed that a relation schema $r(R)$ is given, and we proceeded to normalize it. There are several ways in which we could have come up with the schema $r(R)$:

1. $r(R)$ could have been generated in converting an E-R diagram to a set of relation schemas.

2. $r(R)$ could have been a single relation schema containing *all* attributes that are of interest. The normalization process then breaks up $r(R)$ into smaller schemas.
3. $r(R)$ could have been the result of an ad hoc design of relations that we then test to verify that it satisfies a desired normal form.

In the rest of this section, we examine the implications of these approaches. We also examine some practical issues in database design, including denormalization for performance and examples of bad design that are not detected by normalization.

7.9.1 E-R Model and Normalization

When we define an E-R diagram carefully, identifying all entity sets correctly, the relation schemas generated from the E-R diagram should not need much further normalization. However, there can be functional dependencies among attributes of an entity set. For instance, suppose an *instructor* entity set had attributes *dept_name* and *dept_address*, and there is a functional dependency *dept_name* → *dept_address*. We would then need to normalize the relation generated from *instructor*.

Most examples of such dependencies arise out of poor E-R diagram design. In the preceding example, if we had designed the E-R diagram correctly, we would have created a *department* entity set with attribute *dept_address* and a relationship set between *instructor* and *department*. Similarly, a relationship set involving more than two entity sets may result in a schema that may not be in a desirable normal form. Since most relationship sets are binary, such cases are relatively rare. (In fact, some E-R-diagram variants actually make it difficult or impossible to specify nonbinary relationship sets.)

Functional dependencies can help us detect poor E-R design. If the generated relation schemas are not in desired normal form, the problem can be fixed in the E-R diagram. That is, normalization can be done formally as part of data modeling. Alternatively, normalization can be left to the designer's intuition during E-R modeling, and it can be done formally on the relation schemas generated from the E-R model.

A careful reader will have noted that in order for us to illustrate a need for multivalued dependencies and fourth normal form, we had to begin with schemas that were not derived from our E-R design. Indeed, the process of creating an E-R design tends to generate 4NF designs. If a multivalued dependency holds and is not implied by the corresponding functional dependency, it usually arises from one of the following sources:

- A many-to-many relationship set.
- A multivalued attribute of an entity set.

For a many-to-many relationship set, each related entity set has its own schema, and there is an additional schema for the relationship set. For a multivalued attribute, a separate schema is created consisting of that attribute and the primary key of the entity set (as in the case of the *phone_number* attribute of the entity set *instructor*).

The universal-relation approach to relational database design starts with an assumption that there is one single relation schema containing all attributes of interest. This single schema defines how users and applications interact with the database.

7.9.2 Naming of Attributes and Relationships

A desirable feature of a database design is the **unique-role assumption**, which means that each attribute name has a unique meaning in the database. This prevents us from using the same attribute to mean different things in different schemas. For example, we might otherwise consider using the attribute *number* for phone number in the *instructor* schema and for room number in the *classroom* schema. The join of a relation on schema *instructor* with one on *classroom* is meaningless. While users and application developers can work carefully to ensure use of the right *number* in each circumstance, having a different attribute name for phone number and for room number serves to reduce user errors.

While it is a good idea to keep names for incompatible attributes distinct, if attributes of different relations have the same meaning, it may be a good idea to use the same attribute name. For this reason we used the same attribute name "*name*" for both the *instructor* and the *student* entity sets. If this was not the case (i.e., if we used different naming conventions for the instructor and student names), then if we wished to generalize these entity sets by creating a *person* entity set, we would have to rename the attribute. Thus, even if we did not currently have a generalization of *student* and *instructor*, if we foresee such a possibility, it is best to use the same name in both entity sets (and relations).

Although technically, the order of attribute names in a schema does not matter, it is a convention to list primary-key attributes first. This makes reading default output (as from **select** *) easier.

In large database schemas, relationship sets (and schemas derived therefrom) are often named via a concatenation of the names of related entity sets, perhaps with an intervening hyphen or underscore. We have used a few such names, for example, *inst_sec* and *student_sec*. We used the names *teaches* and *takes* instead of using the longer concatenated names. This was acceptable since it is not hard for you to remember the associated entity sets for a few relationship sets. We cannot always create relationship-set names by simple concatenation; for example, a manager or works-for relationship between employees would not make much sense if it were called *employee_employee*! Similarly, if there are multiple relationship sets possible between a pair of entity sets, the relationship-set names must include extra parts to identify the relationship set.

Different organizations have different conventions for naming entity sets. For example, we may call an entity set of students *student* or *students*. We have chosen to use the singular form in our database designs. Using either singular or plural is acceptable, as long as the convention is used consistently across all entity sets.

As schemas grow larger, with increasing numbers of relationship sets, using consistent naming of attributes, relationships, and entities makes life much easier for the database designer and application programmers.

7.9.3 Denormalization for Performance

Occasionally database designers choose a schema that has redundant information; that is, it is not normalized. They use the redundancy to improve performance for specific applications. The penalty paid for not using a normalized schema is the extra work (in terms of coding time and execution time) to keep redundant data consistent.

For instance, suppose all course prerequisites have to be displayed along with the course information, every time a course is accessed. In our normalized schema, this requires a join of *course* with *prereq*.

One alternative to computing the join on the fly is to store a relation containing all the attributes of *course* and *prereq*. This makes displaying the "full" course information faster. However, the information for a course is repeated for every course prerequisite, and all copies must be updated by the application, whenever a course prerequisite is added or dropped. The process of taking a normalized schema and making it nonnormalized is called **denormalization**, and designers use it to tune the performance of systems to support time-critical operations.

A better alternative, supported by many database systems today, is to use the normalized schema and additionally store the join of *course* and *prereq* as a materialized view. (Recall that a materialized view is a view whose result is stored in the database and brought up to date when the relations used in the view are updated.) Like denormalization, using materialized views does have space and time overhead; however, it has the advantage that keeping the view up to date is the job of the database system, not the application programmer.

7.9.4 Other Design Issues

There are some aspects of database design that are not addressed by normalization and can thus lead to bad database design. Data pertaining to time or to ranges of time have several such issues. We give examples here; obviously, such designs should be avoided.

Consider a university database, where we want to store the total number of instructors in each department in different years. A relation *total_inst*(*dept_name*, *year*, *size*) could be used to store the desired information. The only functional dependency on this relation is *dept_name*, *year*→ *size*, and the relation is in BCNF.

An alternative design is to use multiple relations, each storing the size information for a different year. Let us say the years of interest are 2017, 2018, and 2019; we would then have relations of the form *total_inst_2017*, *total_inst_2018*, *total_inst_2019*, all of which are on the schema (*dept_name*, *size*). The only functional dependency here on each relation would be *dept_name* → *size*, so these relations are also in BCNF.

However, this alternative design is clearly a bad idea—we would have to create a new relation every year, and we would also have to write new queries every year, to take

each new relation into account. Queries would also be more complicated since they may have to refer to many relations.

Yet another way of representing the same data is to have a single relation *dept_year*(*dept_name*, *total_inst_2017*, *total_inst_2018*, *total_inst_2019*). Here the only functional dependencies are from *dept_name* to the other attributes, and again the relation is in BCNF. This design is also a bad idea since it has problems similar to the previous design—namely, we would have to modify the relation schema and write new queries every year. Queries would also be more complicated, since they may have to refer to many attributes.

Representations such as those in the *dept_year* relation, with one column for each value of an attribute, are called **crosstabs**; they are widely used in spreadsheets and reports and in data analysis tools. While such representations are useful for display to users, for the reasons just given, they are not desirable in a database design. SQL includes features to convert data from a normal relational representation to a crosstab, for display, as we discussed in Section 11.3.1.

7.10 Modeling Temporal Data

Suppose we retain data in our university organization showing not only the address of each instructor, but also all former addresses of which the university is aware. We may then ask queries, such as "Find all instructors who lived in Princeton in 1981." In this case, we may have multiple addresses for instructors. Each address has an associated start and end date, indicating when the instructor was resident at that address. A special value for the end date, for example, null, or a value well into the future, such as 9999-12-31, can be used to indicate that the instructor is still resident at that address.

In general, **temporal data** are data that have an associated time interval during which they are **valid**.[10]

Modeling temporal data is a challenging problem for several reasons. For example, suppose we have an *instructor* entity set with which we wish to associate a time-varying address. To add temporal information to an address, we would then have to create a multivalued attribute, each of whose values is a composite value containing an address and a time interval. In addition to time-varying attribute values, entities may themselves have an associated valid time. For example, a student entity may have a valid time from the date the student entered the university to the date the student graduated (or left the university). Relationships too may have associated valid times. For example, the *prereq* relationship may record when a course became a prerequisite for another course. We would thus have to add valid time intervals to attribute values, entity sets, and relationship sets. Adding such detail to an E-R diagram makes it very difficult to create and to comprehend. There have been several proposals to extend the E-R notation to

[10]There are other models of temporal data that distinguish between **valid time** and **transaction time**, the latter recording when a fact was recorded in the database. We ignore such details for simplicity.

course_id	*title*	*dept_name*	*credits*	*start*	*end*
BIO-101	Intro. to Biology	Biology	4	1985-01-01	9999-12-31
CS-201	Intro. to C	Comp. Sci.	4	1985-01-01	1999-01-01
CS-201	Intro. to Java	Comp. Sci.	4	1999-01-01	2010-01-01
CS-201	Intro. to Python	Comp. Sci.	4	2010-01-01	9999-12-31

Figure 7.17 A temporal version of the *course* relation

specify in a simple manner that an attribute value or relationship is time varying, but there are no accepted standards.

In practice, database designers fall back to simpler approaches to designing temporal databases. One commonly used approach is to design the entire database (including E-R design and relational design) ignoring temporal changes. After this, the designer studies the various relations and decides which relations require temporal variation to be tracked.

The next step is to add valid time information to each such relation by adding start and end time as attributes. For example, consider the *course* relation. The title of the course may change over time, which can be handled by adding a valid time range; the resultant schema would be:

$$course\ (course\_id, title, dept\_name, credits, start, end)$$

An instance of the relation is shown in Figure 7.17. Each tuple has a valid interval associated with it. Note that as per the SQL:2011 standard, the interval is **closed** on the left-hand side, that is, the tuple is valid at time *start*, but is **open** on the right-hand side, that is, the tuple is valid until just before time *end*, but is invalid at time *end*. This allows a tuple to have the same start time as the end time of another tuple, without overlapping. In general, left and right endpoints that are closed are denoted by [and], while left and right endpoints that are open are denoted by (and). Intervals in SQL:2011 are of the form [*start*, *end*), that is they are closed on the left and open on the right, Note that 9999-12-31 is the highest possible date as per the SQL standard.

It can be seen in Figure 7.17 that the title of the course CS-201 has changed several times. Suppose that on 2020-01-01 the title of the course is updated again to, say, "Intro. to Scala". Then, the *end* attribute value of the tuple with title "Intro. to Python" would be updated to 2020-01-01, and a new tuple (CS-201, Intro. to Scala, Comp. Sci., 4, 2020-01-01, 9999-12-31) would be added to the relation.

When we track data values across time, functional dependencies that we assumed to hold, such as:

$$course\_id \rightarrow title, dept\_name, credits$$

may no longer hold. The following constraint (expressed in English) would hold instead: "A course *course_id* has only one *title* and *dept_name* value at any given time *t*."

Functional dependencies that hold at a particular point in time are called temporal functional dependencies. We use the term **snapshot** of data to mean the value of the data at a particular point in time. Thus, a snapshot of *course* data gives the values of all attributes, such as title and department, of all courses at a particular point in time. Formally, a **temporal functional dependency** $\alpha \overset{\tau}{\rightarrow} \beta$ holds on a relation schema $r(R)$ if, for all legal instances of $r(R)$, all snapshots of r satisfy the functional dependency $\alpha \rightarrow \beta$.

The original primary key for a temporal relation would no longer uniquely identify a tuple. We could try to fix the problem by adding start and end time attributes to the primary key, ensuring no two tuples have the same primary key value. However, this solution is not correct, since it is possible to store data with overlapping valid time intervals, which would not be caught by merely adding the start and end time attributes to the primary-key constraint. Instead, the temporal version of the primary key constraint must ensure that if any two tuples have the same primary key values, their valid time intervals do not overlap. Formally, if $r.A$ is a **temporal primary key** of relation r, then whenever two tuples t_1 and t_2 in r are such that $t_1.A = t_2.A$, their valid time intervals of t_1 and t_2 must not overlap.

Foreign-key constraints are also more complicated when the referenced relation is a temporal relation. A temporal foreign key should ensure that not only does each tuple in the referencing relation, say r, have a matching tuple in the referenced relation, say s, but also their time intervals are accounted for. It is not required that there be a matching tuple in s with exactly the same time interval, nor even that a single tuple in s has a time interval containing the time interval of the r tuple. Instead, we allow the time interval of the r tuple to be covered by one or more s tuples. Formally, a **temporal foreign-key** constraint from $r.A$ to $s.B$ ensures the following: for each tuple t in r, with valid time interval (l, u), there is a subset s_t of one or more tuples in s such that each tuple $s_i \in s_t$ has $s_i.B = t.A$, and further the union of the temporal intervals of all the s_i contains (l, u).

A record in a student's transcript should refer to the course title at the time when the student took the course. Thus, the referencing relation must also record time information, to identify a particular record from the *course* relation. In our university schema, *takes.course_id* is a foreign key referencing *course*. The *year* and *semester* values of a *takes* tuple could be mapped to a representative date, such as the start date of the semester; the resulting date value could be used to identify a tuple in the temporal version of the *course* relation whose valid time interval contains the specified date. Alternatively, a *takes* tuple may be associated with a valid time interval from the start date of the semester until the end date of the semester, and *course* tuples with a matching *course_id* and an overlapping valid time may be retrieved; as long as *course* tuples are not updated during a semester, there would be only one such record.

Instead of adding temporal information to each relation, some database designers create for each relation a corresponding *history* relation that stores the history of updates to the tuples. For example, a designer may leave the *course* relation unchanged,

but create a relation *course_history* containing all the attributes of *course*, with an additional *timestamp* attribute indicating when a record was added to the *course_history* table. However, such a scheme has limitations, such as an inability to associate a *takes* record with the correct course title.

The SQL:2011 standard added support for temporal data. In particular, it allows existing attributes to be declared to specify a valid time interval for a tuple. For example, for the extended *course* relation we saw above, we could declare

period for *validtime* (*start, end*)

to specify that the tuple is valid in the interval specified by the *start* and *end* (which are otherwise ordinary attributes).

Temporal primary keys can be declared in SQL:2011, as illustrated below, using the extended *course* schema:

primary key (*course_id*, *validtime* **without overlaps**)

SQL:2011 also supports temporal foreign-key constraints that allow a **period** to be specified along with the referencing relation attributes, as well as with the referenced relation attributes. Most databases, with the exception of IBM DB2, Teradata, and possibly a few others, do not support temporal primary-key constraints. To the best of our knowledge, no database system currently supports temporal foreign-key constraints (Teradata allows them to be specified, but at least as of 2018, does not enforce them).

Some databases that do not directly support temporal primary-key constraints allow workarounds to enforce such constraints. For example, although PostgreSQL does not support temporal primary-key constraints natively, such constraints can be enforced using the `exclude` constraint feature supported by PostgreSQL. For example, consider the *course* relation, whose primary key is *course_id*. In PostgreSQL, we can add an attribute *validtime*, of type `tsrange`; the `tsrange` data type of PostgreSQL stores a timestamp range with a start and end timestamp. PostgreSQL supports an && operator on a pair of ranges, which returns true if two ranges overlap and false otherwise. The temporal primary key can be enforced by adding the following `exclude` constraint (a type of constraint supported by PostgreSQL) to the *course* relation as follows:

exclude (*course_id* **with** =, *validtime* **with** &&)

The above constraint ensures that if two *course* tuples have the same *course_id* value, then their *validtime* intervals do not overlap.

Relational algebra operations, such as select, project, or join, can be extended to take temporal relations as inputs and generate temporal relations as outputs. Selection and projection operations on temporal relations output tuples whose valid time intervals are the same as that of their corresponding input tuples. A **temporal join** is slightly different: the valid time of a tuple in the join result is defined as the intersection of the valid times of the tuples from which it is derived. If the valid times do not intersect, the tuple is discarded from the result. To the best of our knowledge, no database supports temporal joins natively, although they can be expressed by SQL queries that explicitly

handle the temporal conditions. Predicates, such as *overlaps, contains, before, and after* and operations such as *intersection* and *difference* on pairs of intervals are supported by several database systems.

7.11 Summary

- We showed pitfalls in database design and how to design a database schema systematically in a way that avoids those pitfalls. The pitfalls included repeated information and inability to represent some information.
- Chapter 6 showed the development of a relational database design from an E-R design and when schemas may be combined safely.
- Functional dependencies are consistency constraints that are used to define two widely used normal forms, Boyce – Codd normal form (BCNF) and third normal form (3NF).
- If the decomposition is dependency preserving, all functional dependencies can be inferred logically by considering only those dependencies that apply to one relation. This permits the validity of an update to be tested without the need to compute a join of relations in the decomposition.
- A canonical cover is a set of functional dependencies equivalent to a given set of functional dependencies, that is minimized in a specific manner to eliminate extraneous attributes.
- The algorithm for decomposing relations into BCNF ensures a lossless decomposition. There are relation schemas with a given set of functional dependencies for which there is no dependency-preserving BCNF decomposition.
- A canonical cover is used to decompose a relation schema into 3NF, which is a small relaxation of the BCNF condition. This algorithm produces designs that are both lossless and dependency-preserving. Relations in 3NF may have some redundancy, but that is deemed an acceptable trade-off in cases where there is no dependency-preserving decomposition into BCNF.
- Multivalued dependencies specify certain constraints that cannot be specified with functional dependencies alone. Fourth normal form (4NF) is defined using the concept of multivalued dependencies. Section 28.1.1 gives details on reasoning about multivalued dependencies.
- Other normal forms exist, including PJNF and DKNF, which eliminate more subtle forms of redundancy. However, these are hard to work with and are rarely used. Chapter 28 gives details on these normal forms. Second normal form is of only historical interest since it provides no benefit over 3NF.
- Relational designs typically are based on simple atomic domains for each attribute. This is called first normal form.

- Time plays an important role in database systems. Databases are models of the real world. Whereas most databases model the state of the real world at a point in time (at the current time), temporal databases model the states of the real world across time.
- There are possible database designs that are bad despite being lossless, dependency-preserving, and in an appropriate normal form. We showed examples of some such designs to illustrate that functional-dependency-based normalization, though highly important, is not the only aspect of good relational design.
- In order for a database to store not only current data but also historical data, the database must also store for each such tuple the time period for which the tuple is or was valid. It then becomes necessary to define temporal functional dependencies to represent the idea that the functional dependency holds at any point in time but not over the entire relation. Similarly, the join operation needs to be modified so as to appropriately join only tuples with overlapping time intervals.
- In reviewing the issues in this chapter, note that the reason we could define rigorous approaches to relational database design is that the relational data model rests on a firm mathematical foundation. That is one of the primary advantages of the relational model compared with the other data models that we have studied.

Review Terms

- Decomposition
  - Lossy decompositions
  - Lossless decompositions
- Normalization
- Functional dependencies
- Legal instance
- Superkey
- R satisfies F
- Functional dependency
  - Holds
  - Trivial
  - Trivial
- Closure of a set of functional dependencies
- Dependency preserving
- Third normal form
- Transitive dependencies
- Logically implied
- Axioms
- Armstrong's axioms
- Sound
- Complete
- Functionally determined
- Extraneous attributes
- Canonical cover
- Restriction of F to R_i
- Dependency-preserving decomposition
- Boyce – Codd normal form (BCNF)
- BCNF decomposition algorithm

- Third normal form (3NF)
- 3NF decomposition algorithm
- 3NF synthesis algorithm
- Multivalued dependency
  - Equality-generating dependencies
  - Tuple-generating dependencies
  - Embedded multivalued dependencies
- Closure
- Fourth normal form (4NF)
- Restriction of D to R_i
- Fifth normal form
- Domain-key normal form (DKNF)
- Atomic domains
- First normal form (1NF)
- Unique-role assumption
- Denormalization
- Crosstabs
- Temporal data
- Snapshot
- Temporal functional dependency
- Temporal primary key
- Temporal foreign-key
- Temporal join

Practice Exercises

7.1 Suppose that we decompose the schema $R = (A, B, C, D, E)$ into

$$(A, B, C)$$
$$(A, D, E).$$

Show that this decomposition is a lossless decomposition if the following set F of functional dependencies holds:

$$A \rightarrow BC$$
$$CD \rightarrow E$$
$$B \rightarrow D$$
$$E \rightarrow A$$

7.2 List all nontrivial functional dependencies satisfied by the relation of Figure 7.18.

A	B	C
a_1	b_1	c_1
a_1	b_1	c_2
a_2	b_1	c_1
a_2	b_1	c_3

Figure 7.18 Relation of Exercise 7.2.

7.3 Explain how functional dependencies can be used to indicate the following:

- A one-to-one relationship set exists between entity sets *student* and *instructor*.
- A many-to-one relationship set exists between entity sets *student* and *instructor*.

7.4 Use Armstrong's axioms to prove the soundness of the union rule. (*Hint*: Use the augmentation rule to show that, if $\alpha \rightarrow \beta$, then $\alpha \rightarrow \alpha\beta$. Apply the augmentation rule again, using $\alpha \rightarrow \gamma$, and then apply the transitivity rule.)

7.5 Use Armstrong's axioms to prove the soundness of the pseudotransitivity rule.

7.6 Compute the closure of the following set F of functional dependencies for relation schema $R = (A, B, C, D, E)$.

$$A \rightarrow BC$$
$$CD \rightarrow E$$
$$B \rightarrow D$$
$$E \rightarrow A$$

List the candidate keys for R.

7.7 Using the functional dependencies of Exercise 7.6, compute the canonical cover F_c.

7.8 Consider the algorithm in Figure 7.19 to compute α^+. Show that this algorithm is more efficient than the one presented in Figure 7.8 (Section 7.4.2) and that it computes α^+ correctly.

7.9 Given the database schema $R(A, B, C)$, and a relation r on the schema R, write an SQL query to test whether the functional dependency $B \rightarrow C$ holds on relation r. Also write an SQL assertion that enforces the functional dependency. Assume that no null values are present. (Although part of the SQL standard, such assertions are not supported by any database implementation currently.)

7.10 Our discussion of lossless decomposition implicitly assumed that attributes on the left-hand side of a functional dependency cannot take on null values. What could go wrong on decomposition, if this property is violated?

7.11 In the BCNF decomposition algorithm, suppose you use a functional dependency $\alpha \rightarrow \beta$ to decompose a relation schema $r(\alpha, \beta, \gamma)$ into $r_1(\alpha, \beta)$ and $r_2(\alpha, \gamma)$.

a. What primary and foreign-key constraint do you expect to hold on the decomposed relations?

b. Give an example of an inconsistency that can arise due to an erroneous update, if the foreign-key constraint were not enforced on the decomposed relations above.

```
result := ∅;
/* fdcount is an array whose ith element contains the number
   of attributes on the left side of the ith FD that are
   not yet known to be in α+ */
for i := 1 to |F| do
   begin
      let β → γ denote the ith FD;
      fdcount [i] := |β|;
   end
/* appears is an array with one entry for each attribute. The
   entry for attribute A is a list of integers. Each integer
   i on the list indicates that A appears on the left side
   of the ith FD */
for each attribute A do
   begin
      appears [A] := NIL;
      for i := 1 to |F| do
         begin
            let β → γ denote the ith FD;
            if A ∈ β then add i to appears [A];
         end
   end
addin (α);
return (result);

procedure addin (α);
for each attribute A in α do
   begin
      if A ∉ result then
         begin
            result := result ∪ {A};
            for each element i of appears[A] do
               begin
                  fdcount [i] := fdcount [i] − 1;
                  if fdcount [i] := 0 then
                     begin
                        let β → γ denote the ith FD;
                        addin (γ);
                     end
               end
         end
   end
```

Figure 7.19 An algorithm to compute α^+.

c. When a relation schema is decomposed into 3NF using the algorithm in Section 7.5.2, what primary and foreign-key dependencies would you expect to hold on the decomposed schema?

7.12 Let $R_1, R_2, \ldots, R_n$ be a decomposition of schema U. Let $u(U)$ be a relation, and let $r_i = \Pi_{R_i}(u)$. Show that

$$u \subseteq r_1 \bowtie r_2 \bowtie \cdots \bowtie r_n$$

7.13 Show that the decomposition in Exercise 7.1 is not a dependency-preserving decomposition.

7.14 Show that there can be more than one canonical cover for a given set of functional dependencies, using the following set of dependencies:

$$X \rightarrow YZ,\ Y \rightarrow XZ, \text{ and } Z \rightarrow XY.$$

7.15 The algorithm to generate a canonical cover only removes one extraneous attribute at a time. Use the functional dependencies from Exercise 7.14 to show what can go wrong if two attributes inferred to be extraneous are deleted at once.

7.16 Show that it is possible to ensure that a dependency-preserving decomposition into 3NF is a lossless decomposition by guaranteeing that at least one schema contains a candidate key for the schema being decomposed. (*Hint*: Show that the join of all the projections onto the schemas of the decomposition cannot have more tuples than the original relation.)

7.17 Give an example of a relation schema R' and set F' of functional dependencies such that there are at least three distinct lossless decompositions of R' into BCNF.

7.18 Let a **prime** attribute be one that appears in at least one candidate key. Let α and β be sets of attributes such that $\alpha \rightarrow \beta$ holds, but $\beta \rightarrow \alpha$ does not hold. Let A be an attribute that is not in α, is not in β, and for which $\beta \rightarrow A$ holds. We say that A is **transitively dependent** on α. We can restate the definition of 3NF as follows: A relation schema R is in 3NF with respect to a set F of functional dependencies if there are no nonprime attributes A in R for which A is transitively dependent on a key for R. Show that this new definition is equivalent to the original one.

7.19 A functional dependency $\alpha \rightarrow \beta$ is called a **partial dependency** if there is a proper subset γ of α such that $\gamma \rightarrow \beta$; we say that β is *partially dependent* on α. A relation schema R is in **second normal form** (**2NF**) if each attribute A in R meets one of the following criteria:

- It appears in a candidate key.

- It is not partially dependent on a candidate key.

Show that every 3NF schema is in 2NF. (*Hint*: Show that every partial dependency is a transitive dependency.)

7.20 Give an example of a relation schema R and a set of dependencies such that R is in BCNF but is not in 4NF.

Exercises

7.21 Give a lossless decomposition into BCNF of schema R of Exercise 7.1.

7.22 Give a lossless, dependency-preserving decomposition into 3NF of schema R of Exercise 7.1.

7.23 Explain what is meant by *repetition of information* and *inability to represent information*. Explain why each of these properties may indicate a bad relational-database design.

7.24 Why are certain functional dependencies called *trivial* functional dependencies?

7.25 Use the definition of functional dependency to argue that each of Armstrong's axioms (reflexivity, augmentation, and transitivity) is sound.

7.26 Consider the following proposed rule for functional dependencies: If $\alpha \rightarrow \beta$ and $\gamma \rightarrow \beta$, then $\alpha \rightarrow \gamma$. Prove that this rule is *not* sound by showing a relation r that satisfies $\alpha \rightarrow \beta$ and $\gamma \rightarrow \beta$, but does not satisfy $\alpha \rightarrow \gamma$.

7.27 Use Armstrong's axioms to prove the soundness of the decomposition rule.

7.28 Using the functional dependencies of Exercise 7.6, compute B^+.

7.29 Show that the following decomposition of the schema R of Exercise 7.1 is not a lossless decomposition:

$$(A, B, C)$$
$$(C, D, E).$$

Hint: Give an example of a relation $r(R)$ such that $\Pi_{A,B,C}(r) \bowtie \Pi_{C,D,E}(r) \neq r$

7.30 Consider the following set F of functional dependencies on the relation schema (A, B, C, D, E, G):

$$A \rightarrow BCD$$
$$BC \rightarrow DE$$
$$B \rightarrow D$$
$$D \rightarrow A$$

a. Compute B^+.

b. Prove (using Armstrong's axioms) that AG is a superkey.

c. Compute a canonical cover for this set of functional dependencies F; give each step of your derivation with an explanation.

d. Give a 3NF decomposition of the given schema based on a canonical cover.

e. Give a BCNF decomposition of the given schema using the original set F of functional dependencies.

7.31 Consider the schema $R = (A, B, C, D, E, G)$ and the set F of functional dependencies:

$$AB \rightarrow CD$$
$$B \rightarrow D$$
$$DE \rightarrow B$$
$$DEG \rightarrow AB$$
$$AC \rightarrow DE$$

R is not in BCNF for many reasons, one of which arises from the functional dependency $AB \rightarrow CD$. Explain why $AB \rightarrow CD$ shows that R is not in BCNF and then use the BCNF decomposition algorithm starting with $AB \rightarrow CD$ to generate a BCNF decomposition of R. Once that is done, determine whether your result is or is not dependency preserving, and explain your reasoning.

7.32 Consider the schema $R = (A, B, C, D, E, G)$ and the set F of functional dependencies:

$$A \rightarrow BC$$
$$BD \rightarrow E$$
$$CD \rightarrow AB$$

a. Find a nontrivial functional dependency containing no extraneous attributes that is logically implied by the above three dependencies and explain how you found it.

b. Use the BCNF decomposition algorithm to find a BCNF decomposition of R. Start with $A \rightarrow BC$. Explain your steps.

c. For your decomposition, state whether it is lossless and explain why.

d. For your decomposition, state whether it is dependency preserving and explain why.

7.33 Consider the schema $R = (A, B, C, D, E, G)$ and the set F of functional dependencies:

$$AB \rightarrow CD$$
$$ADE \rightarrow GDE$$
$$B \rightarrow GC$$
$$G \rightarrow DE$$

Use the 3NF decomposition algorithm to generate a 3NF decomposition of R, and show your work. This means:

a. A list of all candidate keys

b. A canonical cover for F, along with an explanation of the steps you took to generate it

c. The remaining steps of the algorithm, with explanation

d. The final decomposition

7.34 Consider the schema $R = (A, B, C, D, E, G, H)$ and the set F of functional dependencies:

$$AB \rightarrow CD$$
$$D \rightarrow C$$
$$DE \rightarrow B$$
$$DEH \rightarrow AB$$
$$AC \rightarrow DC$$

Use the 3NF decomposition algorithm to generate a 3NF decomposition of R, and show your work. This means:

a. A list of all candidate keys

b. A canonical cover for F

c. The steps of the algorithm, with explanation

d. The final decomposition

7.35 Although the BCNF algorithm ensures that the resulting decomposition is lossless, it is possible to have a schema and a decomposition that was not generated by the algorithm, that is in BCNF, and is not lossless. Give an example of such a schema and its decomposition.

7.36 Show that every schema consisting of exactly two attributes must be in BCNF regardless of the given set F of functional dependencies.

7.37 List the three design goals for relational databases, and explain why each is desirable.

7.38 In designing a relational database, why might we choose a non-BCNF design?

7.39 Given the three goals of relational database design, is there any reason to design a database schema that is in 2NF, but is in no higher-order normal form? (See Exercise 7.19 for the definition of 2NF.)

7.40 Given a relational schema $r(A, B, C, D)$, does $A \twoheadrightarrow BC$ logically imply $A \twoheadrightarrow B$ and $A \twoheadrightarrow C$? If yes prove it, or else give a counter example.

7.41 Explain why 4NF is a normal form more desirable than BCNF.

7.42 Normalize the following schema, with given constraints, to 4NF.

books(*accessionno*, *isbn*, *title*, *author*, *publisher*)
users(*userid*, *name*, *deptid*, *deptname*)
accessionno $\rightarrow$ *isbn*
isbn $\rightarrow$ *title*
isbn $\rightarrow$ *publisher*
isbn $\twoheadrightarrow$ *author*
userid $\rightarrow$ *name*
userid $\rightarrow$ *deptid*
deptid $\rightarrow$ *deptname*

7.43 Although SQL does not support functional dependency constraints, if the database system supports constraints on materialized views, and materialized views are maintained immediately, it is possible to enforce functional dependency constraints in SQL. Given a relation $r(A, B, C)$, explain how constraints on materialized views can be used to enforce the functional dependency $B \rightarrow C$.

7.44 Given two relations $r(A, B, validtime)$ and $s(B, C, validtime)$, where *validtime* denotes the valid time interval, write an SQL query to compute the temporal natural join of the two relations. You can use the && operator to check if two intervals overlap and the $*$ operator to compute the intersection of two intervals.

Further Reading

The first discussion of relational database design theory appeared in an early paper by [Codd (1970)]. In that paper, Codd also introduced functional dependencies and first, second, and third normal forms.

Armstrong's axioms were introduced in [Armstrong (1974)]. BCNF was introduced in [Codd (1972)]. [Maier (1983)] is a classic textbook that provides detailed coverage of normalization and the theory of functional and multivalued dependencies.

Bibliography

[Armstrong (1974)] W. W. Armstrong, "Dependency Structures of Data Base Relationships", In *Proc. of the 1974 IFIP Congress* (1974), pages 580–583.

[Codd (1970)] E. F. Codd, "A Relational Model for Large Shared Data Banks", *Communications of the ACM*, Volume 13, Number 6 (1970), pages 377–387.

[Codd (1972)] E. F. Codd. "Further Normalization of the Data Base Relational Model", In *[Rustin (1972)]*, pages 33–64 (1972).

[Maier (1983)] D. Maier, *The Theory of Relational Databases*, Computer Science Press (1983).

[Rustin (1972)] R. Rustin, *Data Base Systems*, Prentice Hall (1972).

Credits