Now that the Specification Document for the Student Registration System has been approved by the university registrar, we are ready to begin designing the database portion of the system. In Chapter 12, we will discuss the complete design process for an entire transaction processing application, including the final product of that process, the Design Document. Here we discuss the database portion of the design, of which the final product is the complete (compilable) set of CREATE statements that declare the database schema—tables, indices, domains, assertions, and so forth. We present the complete database design for the Student Registration System in Section 5.7.

The key issue in database design is to accurately model the appropriate aspects of a large enterprise as a relational database that can be efficiently accessed and updated by a large number of concurrently executing transactions in (perhaps) a number of distinct applications, some of which might also include decision support queries. As in other engineering disciplines, the design process can be facilitated if it is performed according to some specific methodology and can be evaluated according to some objective criteria.

In this chapter, we present a popular methodology for designing relational databases, the entity-relationship (E-R) approach, introduced by [Chen 1976]. Database design will be revisited in Chapter 8, where we present the relational normalization theory, which provides objective criteria for evaluating alternative designs.

It should be noted that many of the mechanisms underlying both the E-R approach and the relational normalization theory have been captured in computer programs, thereby relieving the designer of some routine aspects of carrying out a design. Still, this job requires a good deal of creativity, technical expertise, experience, and understanding of fundamental database design principles.

5.1 CONCEPTUAL MODELING WITH THE E-R APPROACH

To quash one very common misconception, we emphasize that the E-R approach is not a relative, a derivative, or a generalization of the relational data model. In fact, it is not a data model at all but a design methodology, which can be applied (but
Chapter 5  ■  Database Design I: The Entity-Relationship Model

is not limited) to the relational model. The term “relationship” refers to one of the two main components of the methodology rather than to the relational data model.

The two main components of the E-R approach are the concepts of entity and relationship. Entities model the objects that are involved in an enterprise—for example, the students, professors, and courses in a university. Relationships model the connections among the entities—for example, professors teach courses. In addition, integrity constraints on the entities and relationships form an important part of an E-R specification, much as they do in the relational model. For example, a professor can teach only one course at a given time on a given day.

An entity-relationship (E-R) diagram (peek ahead at Figure 5.1, page 92, and Figure 5.3, page 96) is a graphical representation of the entities, relationships, and constraints that make up a given design. As in other visually oriented design methodologies, it provides a graphical summary of the design that is extremely useful to the designer, not only in validating the correctness of the design but also in discussing it with colleagues and in explaining it to the programmers who will be using it. Unfortunately, there is no standard drawing convention for E-R diagrams, and hence there is a good deal of variation among database texts in many aspects of this approach.

After the enterprise has been modeled with E-R diagrams (in any of their flavors), there are straightforward ways of converting these diagrams into sets of CREATE TABLE statements. Unfortunately, this conversion process does not yield a unique schema, especially in the presence of constraints, because some constraints that can be indicated in the E-R diagrams have no direct counterparts in SQL. These and related issues will be discussed in due time.

The creative part of the E-R methodology is deciding what entities, relationships, and constraints to use in modeling the enterprise. The simple examples we can include in a text might make these decisions look easy, but in practice designers must combine a detailed understanding of the workings of the enterprise with a considerable amount of technical knowledge, judgment, and experience.

An important advantage of the methodology is that the designer can focus on complete and accurate modeling of the enterprise, without (initially) worrying about efficiently executing the required queries and updates against the final database. Later, when the E-R diagrams are to be converted to CREATE TABLE statements, the designer can add efficiency considerations to the final table designs using normalization theory (Chapter 8) and other techniques to be discussed later in this chapter and in Chapter 13.

5.2 ENTITIES AND ENTITY TYPES

The first step in the E-R approach is to select the entities that will be used to model the enterprise. An entity is quite similar to an object, (see Appendix Section A.1.1) except that an entity does not have methods. It might be a concrete object in the real world, such as John Doe, the Cadillac parked at 123 Main Street, or the Empire State Building, or it might be an abstract object, such as the Citibank account 123456789, the database course CS305, or the Computer Science Department at SUNY Stony Brook.
Similar entities are aggregated into **entity types**. For instance, John Doe, Mary Doe, Joe Blow, and Ann White might be aggregated into the entity type **Person** based on the fact that these entities represent humans. John Doe and Joe Blow might also belong to the entity type **Student** because in our sample database of Chapter 4 these objects presumably represented students. Similarly, Mary Doe and Ann White might be classified as members of the entity type **Professor**.

Other examples of entity types include

- CS305, MGT315, and EE101, entities of type **Course**
- Alf and E.T., entities of type **SpaceAlien**
- CIA, FBI, and IRS, entities of type **GovernmentAgency**

**Attributes.** As with relations (and objects), entities are described using attributes. Every attribute of an entity specifies a particular property of that entity. For instance, the **Name** attribute of a **Person** entity normally specifies a string of characters that denotes the real-world name of the person represented by that database entity. Similarly, the **Age** attribute specifies the number of times the Earth had circled around the Sun since the moment that real-world person was born.

All of our examples of entity types have been semantic in nature; that is, the entity type consists of a semantically related set of entities. For example, it is usually pointless to classify people, cars, and paper clips in one entity type because they have little in common in a typical enterprise modeling. More useful is classifying semantically similar entities in one entity type, since such entities are likely to have useful common attributes that describe them. For example, in any enterprise people have many common attributes, such as **Name**, **Age**, and **Address**. Classification into entity types allows us to associate these attributes with the entity type instead of with the entities themselves.

Of course, different entity types have different sets of attributes. For instance, the **PaperClip** entity type might have attributes **Size** and **Price**, while **Course** might have attributes **CrsName**, **CrsCode**, **Credits**, and **Description**.

**Domains.** As in the relational model, the domain of an attribute specifies the set from which its value can be drawn. Unlike the relational model, however, E-R attributes can be **set-valued**. This means that the value of an attribute can be a set of values from the corresponding domain rather than a single value. For example, an entity type **Person** might have set-valued attributes **ChildrenNames** and **Hobbies**.

The inability to express set-valued attributes conveniently was one of the major criticisms of the relational data model that motivated the development of the object-oriented data model. However, the use of set-valued attributes in the E-R model is just a matter of convenience. Relations (as defined in Chapter 4) can be used to model entities with set-valued attributes with some extra effort.

**Keys.** As with the relational model, it is useful to introduce the key constraints associated with entity types. A key constraint on an entity type, \( S \), is a set of attributes, \( A \), of \( S \) such that
1. No two entities in \( S \) have the same values for every attribute in \( A \) (for instance, two different \texttt{Company} entities cannot agree on both the \texttt{Name} and the \texttt{Address} attributes).

2. No proper subset of the attributes in \( A \) has property 1 (i.e., the set \( A \) is \textit{minimal} with respect to this property).

Clearly, this concept of entity keys is analogous to that of candidate keys in the relational model. One subtle point, though, is that attributes in the E-R approach can be set-valued and, in principle, such an attribute can be part of a key. However, in practice set-valued attributes that occur in keys are not very natural and often indicate poor design.

**Schema.** As with the relational model, we define the \textit{schema} of an entity type to consist of the name of that type, the collection of the attributes (with their associated domains and the indicator of whether each attribute is set-valued or single-valued), and the key constraints.

**E-R diagram representation.** Entity types are represented in E-R diagrams as rectangles, and their attributes are represented as ovals. Set-valued attributes are represented as double ovals. Figure 5.1 depicts one possible representation of the \texttt{Person} entity type. Note that in this picture \texttt{Hobbies} is specified as a set-valued attribute and \texttt{SSN} is underlined to indicate that it is a key.

**Representation in the relational model.** The correspondence between entities in the E-R model and relations in the relational model is straightforward. Each entity type is converted into a relation, and each of its attributes is converted into an attribute of the relation.

This simplicity might seem suspicious in view of the fact that entities can have set-valued attributes while relations cannot. How, then, can a set-valued attribute of an entity be turned into a single-valued attribute of the corresponding relation without violating the property of data atomicity (see Section 4.2) of the relational model?

The answer is that each entity that has a set-valued attribute is represented in the translation by a \textit{set} of tuples, one for each element in the attribute value. To illustrate, suppose that the entity type \texttt{Person} of Figure 5.1 is populated by the following entities:
Figure 5.2 Translation of entity type **PERSON** into a relation.

<table>
<thead>
<tr>
<th>Person</th>
<th>SSN</th>
<th>Name</th>
<th>Address</th>
<th>Hobby</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>111111111</td>
<td>John Doe</td>
<td>123 Main St.</td>
<td>Stamps</td>
</tr>
<tr>
<td>111111111</td>
<td>John Doe</td>
<td>123 Main St.</td>
<td>Coins</td>
<td></td>
</tr>
<tr>
<td>555666777</td>
<td>Mary Doe</td>
<td>7 Lake Dr.</td>
<td>Hiking</td>
<td></td>
</tr>
<tr>
<td>555666777</td>
<td>Mary Doe</td>
<td>7 Lake Dr.</td>
<td>Skating</td>
<td></td>
</tr>
<tr>
<td>987654321</td>
<td>Bart Simpson</td>
<td>Fox 5 TV</td>
<td>Acting</td>
<td></td>
</tr>
</tbody>
</table>

(111111111, John Doe, 123 Main St., {Stamps, Coins})
(555666777, Mary Doe, 7 Lake Dr., {Hiking, Skating})
(987654321, Bart Simpson, Fox 5 TV, {Acting})

In translation, we obtain the relation depicted in Figure 5.2.

The next question to ask is what keys of the relation are obtained by the above translation. If the entity type does not have set-valued attributes, the answer is simple: Each key (which is a set of attributes) of the entity type becomes a key of the corresponding relation schema. However, if one of the entity attributes is set-valued, determining the keys is a bit more involved. In the entity type **PERSON**, the attribute **SSN** is a key, because no two **PERSON** entities can have the same Social Security number and all hobbies are elements of a set that is the value of the **Hobby** attribute. However, in the **PERSON** relation (Figure 5.2) both John Doe and Mary Doe are represented by a pair of tuples and each Social Security number occurs twice. Therefore, **SSN** is not a key of that relation.

Clearly, the set-valued attribute **Hobby** is the trouble maker. To obtain a key of the relation in question, we must include the offending attribute. Thus, the key of the **PERSON** relation in Figure 5.2 is \{**SSN**, **Hobby**\}.

The following CREATE TABLE statement defines the schema for the **PERSON** relation.

```sql
CREATE TABLE PERSON (  
  SSN INTEGER,  
  Name CHAR(20),  
  Address CHAR(50),  
  Hobby CHAR(10),  
  PRIMARY KEY (SSN, Hobby) )
```

Even though we found a key in the end, a careful examination of the above table leaves us uneasy. Intuitively, it does not seem right that the **Hobby** attribute must be used to identify tuples in the **PERSON** relation. Furthermore, in the original entity type **PERSON**, any concrete value of **SSN** is known to uniquely identify the value of **Name** and **Address**. In the translation of Figure 5.2, we see that this property is still true, but it is not captured by the primary key constraint, which states that in
order to uniquely determine a tuple, we must specify the value of both SSN and Hobby. In contrast, in the entity type Person the value of Hobby is not required to determine the value of Name and Address. This important constraint has been lost in the translation!

The preceding example is the first indication that the E-R approach alone is not a sufficient tool for guaranteeing good relational design. Chapter 8 will provide a host of objective criteria that can help the database designer evaluate the relational schema obtained by converting E-R diagrams into relations. In particular, the problem with the relation in Figure 5.2 is that it is not in a certain “normal form” defined in Chapter 8. That chapter proceeds to develop algorithms that can automatically rectify the problem by splitting such offending relations into smaller relations that are in a desired normal form.

5.3 RELATIONSHIPS AND RELATIONSHIP TYPES

The E-R approach makes a sharp distinction between the entities themselves and the mechanism that relates them to each other. This mechanism is called relationships. Just as entities are classified into entity types, relationships that relate the same types of entities and that have the same meaning are grouped into relationship types.¹

For instance, Student entities are related to Program entities via relationships of type MajorsIn. Likewise, Professor entities are related to the departments they work for via relationships of type WorksIn.

Once again, we emphasize that the concept of relationship in the E-R approach is distinct from the concept of relation in the relational data model. Relationships are just one of the modeling tools in the arsenal of the E-R approach. When it comes to implementation, both entities and relationships are represented as relations (i.e., tables) in the DBMS.

Attributes and roles. Like entities, relationships can have attributes. For instance, the relationship MajorsIn might have an attribute Since, which indicates the date the student was admitted into the corresponding major. The WorksIn relationship might have the attribute Since to indicate the start date of employment.

Attributes do not provide a complete description of relationships. Consider the entity type Employee and the relationship ReportsTo, which relates employees to other employees. The first type of employee is the subordinate while the second is the boss. Thus, if we just say that ⟨John, Bill⟩ is a relationship of type ReportsTo, we still do not know who reports to whom.

Splitting the Employee entity type into Subordinate and Supervisor does not help, because ReportsTo might represent the entire chain of reporting in a corporate hierarchy, making some employees subordinates and supervisors at the same time.

¹ When there is a danger of confusion between relationships and entities on the one hand and relationship types and entity types on the other, we use the terms “relationship instance” and “entity instance” instead of “relationship” and “entity,” respectively.
The solution is to recognize that the various entity types participating in a relationship type play different roles in that relationship. For each such participating entity type, we define a role and give that role a name (for example, Subordinate). A role is similar to an attribute, but instead of specifying some property of the relationship it specifies in what way an entity type participates in the relationship. Both roles and attributes are part of the schema of the relationship type.

For example, the relationship type WorksIn has two roles, Professor and Department. The Professor role identifies the Professor entity involved in a WorksIn relationship, and the Department role identifies the corresponding Department entity in the relationship. Similarly, the relationship type MajorsIn has two roles: Student and Program.

When all of the entities involved in a relationship belong to distinct entity types (as in WorksIn and MajorsIn), it is not necessary to explicitly indicate the roles, because we can always adopt some convention, such as naming the roles after the corresponding entity types (which is typical in practice). However, such simplification is not possible when some of the entities involved are drawn from the same entity type, as is the case with the ReportsTo relationship. Here, we have to explicitly indicate the roles, for example, Subordinate and Supervisor. Figure 5.3 shows several examples of relationships, including those where roles must be named explicitly.

To summarize, a schema of a relationship type includes

- A list of attributes along with their corresponding domains. An attribute can be single-valued or set-valued.
- A list of roles along with their corresponding entity types. Unlike attributes, roles are always single-valued.
- A set of constraints, to be described later. (In Figure 5.3, constraints are represented as arrows, which will be explained later.)

The number of roles engaged in a relationship type is called the degree of the type.

We can now define the concept of a relationship more precisely. A relationship type \( R \) of degree \( n \) is defined by its attributes \( A_1, \ldots, A_k \) and roles \( R_1, \ldots, R_n \). The relationships populating \( R \) are defined to be tuples of the form

\[
(e_1, e_2, \ldots, e_n; a_1, a_2, \ldots, a_k)
\]

where \( e_1, \ldots, e_n \) are entities that are values of the roles \( R_1, \ldots, R_n \), respectively (these are the entities involved in the relationship) and \( a_1, a_2, \ldots, a_k \) are values of the attributes \( A_1, \ldots, A_k \), respectively. We assume that all of the values of the attributes in the relationship are in their respective domains as defined in the relationship type and all of the entities are of the correct entity types as defined in their respective roles.

\[\text{To avoid confusion, we use different fonts to distinguish entity types from the roles they play in various relationships.}\]
Since

Husband

Wife

ProductCustomer

Subordinate

Supervisor

Date

Price

PERSONEMPLOYEE

STUDENT PROGRAM

PARTPROJECT

SUPPLIER

PROFESSOR DEPARTMENT

MAJORSIN

WORKSIN

REPORTSTO

MARRIEDTO

Figure 5.3 E-R diagrams for several relationship types.

For instance, the relationship type MAJORSIN can have the schema

\[(\text{Student}, \text{Program}; \text{Since})\]

where Student and Program are roles and Since is an attribute. One instance in this relationship type might be

\[(\text{Homer Simpson}, \text{EE}; 1994)\]

This relationship states that the entity Homer Simpson is a student who is enrolled in the program represented by the entity EE and has been enrolled since 1994. The first two components in the tuple are entities, while the last one is a plain value drawn from the domain of years.

**E-R diagram representation.** In E-R diagrams, relationship types are represented as diamonds and roles are represented as edges that connect relationship types with the appropriate entity types. If a role must be named explicitly (because some of the entities involved in the relationship are drawn from the same entity type), the name is included in the diagram. Figure 5.3 shows the E-R diagram for several of the relationships we have been discussing (we omitted all entity attributes to reduce
clutter). The first three relationships in the figure are binary because they each relate two entity types. The last relationship is ternary because it relates three entity types. This last diagram also illustrates the point that sometimes the semantics of a diagram can be easier to convey if default role names (in this case Project and Part) are renamed (as Customer and Product, respectively).

**Keys.** The key of a relationship enables the designer to naturally express many constraints easily and uniformly. In the case of entity types, a key is just a set of attributes that uniquely identifies an entity. However, attributes alone do not fully characterize relationships. Roles must also be taken into account, so we define a **key of a relationship type**, R, to be a minimal set of roles and attributes of R whose values uniquely identify the relationship instances in that relationship type. In other words, let \( R_1, \ldots, R_k \) be a subset of the set of all roles of R and \( A_1, \ldots, A_s \) be a subset of the attributes of R. Then the set \( \{R_1, \ldots, R_k; A_1, \ldots, A_s\} \) is a key of R if the following holds.

1. **Uniqueness.** R does not have a pair of distinct relationship instances that have the same values for every role and attribute in \( \{R_1, \ldots, R_k; A_1, \ldots, A_s\} \).
2. **Minimality.** No subset of \( \{R_1, \ldots, R_k; A_1, \ldots, A_s\} \) has property 1.

In some cases, the key of a relationship takes a special form. Consider the relationship WorksIn between entities of type Professor and type Department. It is reasonable to assume that each department has several professors but that each professor works for a single department. In other words, WorksIn relates professors to departments in a many-to-one fashion. Because any given Professor entity can occur in at most one relationship of type WorksIn, the role Professor is a key of WorksIn. While there is no universally accepted representation for relationship keys in E-R diagrams, a relationship key that consists of just one role can be conveniently expressed by drawing this role as an arrow pointing in the direction of the relationship’s diamond. Observe that there can be several roles that form a key, and so an E-R diagram can have several arrows pointing toward the same diamond. For instance, in Figure 5.3 both \{Husband\} and \{Wife\} are keys of the relationship type MarriedTo, so each of these roles is represented as an arrow.

Keys that consist of more than one role or attribute are usually represented textually, next to the diamonds that represent the corresponding relationship type. When representing such keys, we first list the roles and then the attributes. For example, in the last diagram of Figure 5.3 one key could be \{Customer, Product; Date\} (Date is included in the key because price may vary depending on the date).

In many situations, however, the relationship key turns out to be the set of all roles and is unique. In such a case, we do not specify the key in the diagram.

**Representation in the relational model.** The following procedure can be used to map a relationship type R into a relation schema.

---

3 Most texts define a limited form of relationship key under the guise of the many-to-one cardinality constraint.
Attributes of the relation schema. The attributes of the relation schema are
1. The attributes of R itself
2. For each role in R, the primary key of the associated entity type

Note that we use the primary key of the entity type, not the primary key of the relation schema constructed out of that entity type, because the goal is to identify the entity involved in the relationship. Thus, for example, in the case of a role associated with the entity type PERSON we use SSN and omit Hobby.

Candidate keys of the relation schema. In most cases, the keys of the relation schema are obtained by direct translation from the keys of R itself. That is, if a role, R, of R belongs to the key of R then the attributes of the primary key, K, of the entity associated with R must belong to the candidate key of the relation schema derived from R.

A slight problem arises when R has set-valued attributes. In that case, we resort to an earlier trick that was used for converting entity keys into relation keys: All set-valued attributes must be included in the candidate key of the relation (see the PERSON entity-to-relation translation) whether or not they are included in the key of R. Note that roles are always single-valued, so this special treatment of set-valued attributes does not apply to the roles.

Foreign key constraints of the relation schema. Because, in the E-R model, a role always refers to some entity (which is mapped to a relation), roles translate into foreign key constraints. The foreign keys of the relation schema corresponding to R are constructed as follows.

For each role in R, the primary key of the associated entity type (which is among the attributes of the relation schema corresponding to R) becomes a foreign key that references the relation obtained from that entity type.

The only problem might be that the primary key of the entity type (e.g., SSN) need not be the primary key of the corresponding relation (e.g., \{SSN, Hobby\}), as we saw in the case of the PERSON entity type. Such problems are eliminated by the relational normalization theory of Chapter 8.

Figure 5.4 shows the CREATE TABLE commands that define the schemas corresponding to some of the relationships in Figure 5.3. Observe that, in the MARRIEDTo relation, we did not define the foreign key constraint because, although SSNhusband and SSNwife clearly reference the SSN attribute of the PERSON relation, SSN is not a candidate key for that relation, as explained earlier. The UNIQUE constraint in the schema guarantees that SSNwife is unique in any instance of the table and is hence a candidate key.

Note that when E-R diagrams are translated into tables, some of these tables describe entities and others describe relationships. Thus, the first E-R diagram of Figure 5.3 translates into three tables: one to describe the entity type of professors, one to describe the entity type of departments, and one to describe the relationship type that describes professors working for various departments.
5.4 ADVANCED FEATURES OF THE E-R APPROACH

5.4.1 Entity Type Hierarchies

Some entity sets might be closely related to others. For instance, every entity of the STUDENT entity set is also a member of the PERSON entity set. Therefore, all of the attributes of the PERSON set are applicable to such entities. Students can also have attributes that are not applicable to a typical PERSON entity (e.g., Major, StartDate, GPA). In this case, we say that the entity type STUDENT is a subtype of the entity type PERSON.

Formally, saying that an entity type \( R \) is a subtype of the entity type \( R' \) is the same as specifying an inter-entity constraint, which means that

1. Every entity instance in \( R \) is also an entity instance in \( R' \),
2. Every attribute in \( R' \) is also an attribute in \( R \).

One important consequence of this definition is that any key of a supertype is also a key of all of its subtypes.
Note that subtyping is not only a constraint but also a relationship. It can be represented as a relationship with roles Sub(type) and Super(type) and is often called the IsA relationship. Thus, the names of the roles in the IsA relationship are fixed, but the ranges of these roles depend on the entity types that are related by each particular IsA relationship.

For instance, in the IsA relationship type that relates Student and Person the range of Sub is Student and the range of Super is Person. A particular instance of this relationship could be (Homer Simpson, Homer Simpson), which states that Homer Simpson is a student and a person. Note that the two entities involved in IsA are always identical.

So what is so useful about the IsA relationship? The answer lies in the fact that subtype constraints introduce a classification hierarchy in the E-R model. For instance, Freshman is a subtype of Student, which in turn is a subtype of Person. This property is transitive, which means that Freshman is also a subtype of Person. The transitive property gives us a way to draw E-R diagrams in a more concise and readable manner. Because of property 2 of subtyping, every attribute of Person is also an attribute of Student and, by transitivity, is also an attribute of Freshman. This phenomenon is often expressed by saying that Student inherits attributes from Person and that Freshman inherits attributes from both Person and Student.

Note that the inherited attributes (SSN, Name, etc.) are not specified explicitly for the entity types Student and Freshman, and yet they are considered valid attributes because of the IsA relationship. In addition to the inherited attributes, Student and Freshman might have attributes of their own, which their corresponding supertypes might not have. Figure 5.5 illustrates this idea. As Student is a subtype of Person, this entity type inherits all of the attributes specified for Person, and so there is no need to repeat the attributes Name and D.O.B. (date of birth) for the Student type. Similarly, Freshman, Sophomore, and so forth, are subtypes of Student, and so we do not need to copy the attributes of Student and Person over to these subtypes. Transitivity has the potential of greatly simplifying an E-R diagram. The Employee branch of the IsA tree provides another example of attribute inheritance. The relationship Employee IsA Person represents the fact that every Employee entity has attributes Department and Salary. In addition, it states that every Employee entity is also a Person entity and, as such, also has the attributes Name, SSN, and so forth.

Note that each IsA triangle in Figure 5.5 represents several relationship types. For instance, the upper triangle represents the relationship types Student IsA Person and Employee IsA Person. Although this notation makes the representation of the IsA relationship different from the representation of other kinds of relationships, it is used because there are a number of constraints associated with entity type hierarchies that can be naturally represented using such notation.

For instance, the union of the entities that belong to the entity types Freshman, Sophomore, Junior, and Senior might be equal to the set of entities of type Student (for example, in a four-year college). This constraint, called the covering constraint, can be handily attached to the lower right IsA triangle. In addition,
Figure 5.5  Example of an IsA hierarchy.

these entity sets might always be disjoint (they are in most American universities), and such a disjointness constraint can also be attached to the lower right triangle.\footnote{There is no universally accepted way of representing covering and disjointness constraints in the E-R diagrams, so pick your own.}

**Entity type hierarchies and data fragmentation.** Our discussion of the IsA relationship focused on conceptual organization and attribute inheritance. However, these hierarchies are also a good way to approach the issue of physical data fragmentation (or data partitioning). The need for data fragmentation often arises in distributed environments, where multiple geographically diverse entities must access a common database. Banks are a typical example because they often have many branches in different cities.

The problem that arises in such distributed enterprises is that of network delay: Accessing a database in New York City from a bank branch in Buffalo can be prohibitive for frequently running transactions. However, the bulk of the data needed by a local bank branch is likely to be of mostly local interest, and it might be a good idea to distribute fragments of such information among databases maintained at the individual branches. We will discuss data fragmentation for distributed
Figure 5.6 Using IsA for data fragmentation.

 database in more detail in Chapter 18. Other uses of data fragmentation will be described in Section 24.3.1.

To see how data fragmentation can be addressed at the database design stage, consider an entity type, Customer, that represents the information about all customers of a bank. For each branch, we can create subtypes, such as NYC_Customer or Buffalo_Customer, that are related to Customer as described in Figure 5.6.

Observe that the constraints associated with type hierarchies provide considerable expressive power in specifying how data might be fragmented. For instance, Figure 5.6 could be interpreted as a requirement that the New York City and Buffalo data must be stored locally. It does not say that the New York City database and the Buffalo customer database must be disjoint, but this restriction can be specified using the disjointness constraint introduced earlier. In addition, we can add the covering constraint to specify that the combined customer information at the branches includes all customers.

Representing IsA hierarchies in the relational model. There are several ways to represent the IsA relationship using relational tables. One general way is to choose a candidate key for all entity types on a branch of the tree that represents the IsA hierarchy, add the key attributes to each entity type, and then convert the entities on that branch into relations, as discussed in Section 5.2. The choice of such a key is possible because, as mentioned earlier, a key of a supertype is also a key of a subtype.

For instance, in Figure 5.5 we can choose {SSN} as such a key and add the SSN attribute to each entity type. The subsequent conversion process will then yield the following relation schemas.

```
PERSON(SSN, Name, D.O.B.)
STUDENT(SSN, StartDate, GPA)
FRESHMAN(SSN)
SOPHOMORE(SSN, Major)
JUNIOR(SSN, Major)
SENIOR(SSN, Major, Advisor)
```
5.4 Advanced Features of the E-R Approach

Employee(SSN, Department, Salary)
Secretary(SSN)
Technician(SSN, Specialization)

In the presence of various constraints, such as disjointness and covering, there are more efficient ways to represent the same information. For instance, if all subentities of Student had the same set of attributes (i.e., if Senior did not have its private attribute Advisor), then instead of representing students using five relations we could have used just one with a special attribute Status whose range would be the set of constants \{Freshman, Sophomore, Junior, Senior\}.  

5.4.2 Participation Constraints

Consider a relationship type WorksIn between entities of types Professor and Department. As discussed earlier, each department has several professors but each professor works for a single department, so the role Professor is a key of WorksIn.

This key constraint ensures that no professor can occur in more than one relationship of type WorksIn. However, it does not guarantee that each professor occurs in some relationship of this type. In other words, the key constraint does not rule out professors who do not work for any department (and possibly get away without teaching any courses!). To close this loophole, the designer can use participation constraints.

Given an entity type, E, a relationship type, R, and a role, ρ, a participation constraint of E in R in role ρ states that for every entity instance e in E, there is a relationship r in R such that e participates in r in role ρ.

Clearly, requiring that the entity type Professor participate in the relationship type WorksIn in role Professor ensures that every professor works in some department.

For another example, we can try to ensure that every student takes at least one course. To this end, we can assume that there is a ternary relationship type, Transcript, which relates Student, Course, and Semester. Our goal can be achieved by imposing a participation constraint on the Student entity type.

In E-R diagrams, participation constraints are represented by a thick line connecting the participating entity with the corresponding relationship as shown in Figure 5.7. The thick arrow connecting Professor to WorksIn indicates both that each professor participates in at least one relationship (denoted by the thick line) and that each professor can participate in at most one relationship (denoted by the arrow). Hence, a one-to-one correspondence between Professor entities and relationships is implied.

Representation in the relational model. Conceptually, representing participation constraints using the relational model is easy. We have already seen, in Figure 5.4, page 99, the CREATE TABLE statement for the WorksIn relationship. So,

---

5 This transformation can be done even if Senior has its own attribute. However, in this case all students who have not reached the senior status will have their Advisor attribute set to NULL.
all it takes to enforce the participation constraint of Professors in WorksIn is to specify the inclusion dependency\(^6\) that states

\[
\text{Professor}(\text{Id}) \text{ references WorksIn}(\text{ProfId})
\]

Incidentally, this is also a foreign key constraint because ProfId is a key of WorksIn. Thus, we can specify the above participation constraint in SQL-92 by simply declaring the Id attribute of Professor as a foreign key constraint.

```sql
CREATE TABLE Professor
(
    Id INTEGER,
    Name CHAR(20),
    DeptId CHAR(4),
    PRIMARY KEY (Id),
    FOREIGN KEY (Id) REFERENCES WorksIn (ProfId) )
```

Note that this foreign key constraint does not rule out the possibility that Id can be a null, which means that in general a NOT NULL constraint for Id would be in order. However, in our case Id is declared as a primary key of Professor, so the NOT NULL constraint is redundant (at least, in SQL-92-compliant databases).

We could do an even better translation by noticing that Id is a key of Professor and also of WorksIn (indirectly, through the foreign key constraint). Thus, we can merge the attributes of WorksIn into the Professor relation (after deleting the attribute ProfId, as the same information is given by the attribute Id). This is possible because the common key of these tables guarantees that each Professor tuple has

\(^6\) Inclusion dependencies were discussed in Section 4.2.2.
exactly one corresponding \texttt{WorksIn} tuple, so no redundancy is created by concatenating such related tuples. This yields the table \texttt{ProfessorMergedWithWorksIn}:

\begin{verbatim}
CREATE TABLE ProfessorMergedWithWorksIn (
    Id INTEGER,
    Name CHAR(20),
    DeptId CHAR(4),
    Since DATE,
    PRIMARY KEY (Id)
    FOREIGN KEY DeptId REFERENCES Department
)
\end{verbatim}

Although conceptually representation of participation constraints in the relational model amounts to nothing more than specifying an inclusion dependency, the actual representation in SQL-92 is not always as simple as in the previous example. The reason is that not all inclusion dependencies are foreign key constraints (see Section 4.2.2), and their expression in SQL-92 requires heavy machinery, such as assertions or triggers, which can have a negative effect on performance.

An example of this situation is the constraint on the participation of \texttt{Student} in the \texttt{Transcript} relationship, depicted in Figure 5.7. The translation of \texttt{Transcript} to SQL-92 is

\begin{verbatim}
CREATE TABLE Transcript (
    StudId INTEGER,
    CrsCode CHAR(6),
    Semester CHAR(6),
    Grade CHAR(1),
    PRIMARY KEY (StudId, CrsCode, Semester),
    FOREIGN KEY (StudId) REFERENCES Student (Id),
    FOREIGN KEY (CrsCode) REFERENCES Course (CrsCode),
    FOREIGN KEY (Semester) REFERENCES Semesters (SemCode))
\end{verbatim}

As before, the foreign key constraints specified for the \texttt{Transcript} table do not guarantee that every student takes a course. To ensure that every student participates in some \texttt{Transcript} relationship, the \texttt{Student} relation must have an inclusion dependency of the form

\begin{verbatim}
Student(Id) references Transcript(StudId)
\end{verbatim}

However, since \texttt{StudId} is not a candidate key in \texttt{Transcript}, this inclusion dependency cannot be represented by foreign key constraints. In Chapter 4, we illustrated how inclusion dependencies can be defined using the \texttt{CREATE ASSERTION} statement (see (4.4) on page 75).

Unfortunately, verifying general assertions is often significantly more costly than verifying foreign key constraints, so the use of constraints such as (4.4) should be carefully weighed against the potential overhead. For instance, if it is determined that including such an assertion slows down crucial database operations, the designer might opt for checking the inclusion dependency as part of a separate, periodically run transaction rather than one run in real time.
A BROKERAGE FIRM EXAMPLE

We have been using the Student Registration System to illustrate most of the concepts of the E-R model. In this section, we use a different enterprise as an additional illustration of these concepts.

The Pie in the Sky Securities Corporation (PSSC) is a brokerage firm that buys and sells stocks for its clients. Thus, the main actors are brokers and clients. PSSC has offices in different cities, and each broker works in one of these offices. A broker can also be an office manager (for the office she works in).

Clients own accounts, and any account can have more than one owner. Each account is also managed by some broker. A client can have several accounts and a broker can manage several accounts, but a client cannot have more than one account in a given office.

The requirement is to design a database for maintaining the above information as well as information about the trades performed in each account.

The basic information about brokers and clients is depicted in Figure 5.8. More information is shown in Figure 5.9. Here we make additional assumptions that a broker can manage at most one office and that each office has at most one manager. Notice that we did not specify a participation constraint for Office in the relationship MANAGEDBy, so it is possible that an office might not have a manager (e.g., if the manager quits and the position remains vacant). Since each account can be maintained in exactly one office and by at most one broker, Figure 5.9 shows a participation constraint of entity Account in the relationship ISHANDLEDBy by the arrow leading from Account to ISHANDLEDBy. Thus, {Account} is a key of ISHANDLEDBy. Notice that the attributes that form a key of an entity are underlined and different keys are underlined differently. Thus, for instance, OFFICE has two keys: {Phone#} and {Address}.

However, after careful examination we can see that the diagram in Figure 5.9 has problems. For one, it requires every account to have a broker. This might or might
not be correct depending on company policy. For another, the requirement that a client cannot have two separate accounts in the same office is not represented in the diagram.

We might try to rectify these problems using the diagram depicted in Figure 5.10. Here we take a slightly different approach and introduce a ternary relation HasAccount with \{Client, Office\} as a key. In addition, we maintain one more relationship, HandledBy, which relates accounts and brokers but does not require that every account has a broker.

Unfortunately, even this diagram has problems. First, notice that the edge that connects Account and HasAccount does not have an arrow. Such an arrow would have made the role \{Account\} a key of the relationship HasAccount, which contradicts the requirement that an account can have multiple owners. However, our new design introduces a different problem: the constraint that each account is assigned to exactly one office is no longer represented in the diagram. The participation constraint of Account in HasAccount says that each account must be assigned to at least one office (and at least one customer), but nothing here says that such an office must be unique. Furthermore, we cannot solve this problem by adding an arrow to this participation constraint because, as noted earlier, this would make it impossible to have multiple owners for the same account.
Figure 5.10  Client/broker Information: second try.

There is one more problem with our new design (which, in fact, was also present in our original design in Figure 5.9). Suppose that we have the following relationships:

\[(\text{Client}_1, \text{Acct}_1, \text{Office}_1) \in \text{HasAccount}\]
\[(\text{Acct}_1, \text{Broker}_1) \in \text{HandledBy}\]
\[(\text{Broker}_1, \text{Office}_2) \in \text{WorksIn}\]

What is there to ensure that \text{Office}_1 and \text{Office}_2 are the same (i.e., \text{Account}_1's office is the same as that of the broker who manages \text{Account}_1)? This last problem is known as a navigation trap: Starting with a given entity, \text{Office}_1, and moving along the triangle formed by the three relationships \text{HasAccount}, \text{HandledBy}, and \text{WorksIn}, we might end up with a different entity, \text{Office}_2, of the same type. Navigation traps of this kind are particularly difficult to avoid in the E-R model because doing so requires the use of participation constraints in combination with so-called functional dependencies (see Section 8.3), but these constraints are
supported by the E-R model only in a very limited way: as keys and participation constraints.  

Note that we can avoid the navigation trap by removing the relationship HasAccount completely and reintroducing the Owns relationship between clients and accounts. However, this brings back the problem that the constraint that a client cannot have more than one account in any given office is no longer represented.

Even though we have not yet obtained a completely satisfactory design for this part of the database, we now turn our attention to the part that deals with stock trading. On a bigger canvas, Figures 5.10 and 5.11 would be connected through the entity Account.

7 To neutralize the navigation trap in this example, we need an inclusion dependency of a fairly complex form. Loosely speaking, we want to ensure that in every tuple \( t \in \text{HasAccount} \), the attribute Account refers to an account handled by some broker and the attribute Office refers to that broker’s office. After learning the basics of relational algebra in Chapter 6, you will see that such an inclusion dependency can be formally specified as follows: \( \pi_{\text{Office}, \text{Account}}(\text{HandledBy} \bowtie_{\text{WorksIn}}) \supseteq \pi_{\text{Office}, \text{Account}}(\text{HasAccount}) \), where \( \pi \) is the projection operator (that chops off all attributes except Office and Account) and \( \bowtie \) is the join operator (which “lines up” the tuples that correspond to the same broker in the relations HandledBy and WorksIn).
Trading information is specified using three entities, `ACCOUNT`, `STOCK`, and `TRANSACTION`. These entities are linked through the relationship `POSITION`, which relates stocks with the accounts they are held in, and the relationship `TRADE`, which represents the actual buying and selling of stocks. This information is depicted in Figure 5.11.

Notice that the role `Transaction` is a key of the relationship `Trade` and that transactions do not exist outside of `Trade` relationships. These constraints are expressed in Figure 5.11 using a thick arrow, which specifies the one-to-one correspondence between `TRANSACTION` entities and `TRADE` relationships.

### 5.6 LIMITATIONS OF THE E-R APPROACH

You have now seen two case studies of the use of the entity-relationship model for database design. If E-R design still does not seem completely clear to you, you are in good company. Although we have discussed several concepts that might provide general guidance in organizing enterprise data, applying these concepts in any concrete situation is still a mix of experience, art, and black magic. There is considerable freedom in deciding whether a particular datum should be an entity, a relationship, or an attribute. Furthermore, even after these issues are settled the various relationships that exist among entities can be expressed in different ways. This section discusses some of the dilemmas that often confront a database designer.

**Entity or attribute?** In Figure 5.7, semesters are represented as entities. However, we could as well make `TRANSCRIPT` into a binary (rather than ternary) relation and turn `SEMESTER` into one of its attributes. The obvious question is which representation is best (and in which case).

To some extent, the decision of whether a particular datum should be represented as an entity or an attribute is a matter of taste. Beyond that, the representation might depend on whether the datum has an internal structure of its own. If the datum has no internal data structure, keeping it as a separate entity makes the E-R diagram more complex and, more important, adds an extra relation to your database schema when you convert the diagram into the relational model. On the other hand, if the datum has attributes of its own, it is possible that these attributes cannot be represented if the datum itself is demoted to the status of an attribute.

For instance, in Figure 5.7 the entity type `SEMESTER` does not have its own attributes, so representing the semester information as an entity appears to be overkill. However, it is entirely possible that the Requirements Document might state that the following additional information must be available for each semester: `Start_date`, `End_date`, `Holidays`, `Enrollment`. In such a case, the semester information cannot be an attribute of the `TRANSCRIPT` relationship, as there would then be no place to specify the information about the key dates and the enrollment associated with semesters (e.g., the total enrollment in the university during a semester is not an attribute of any particular transcript relationship).
Entity or relationship? Consider once again the diagram in Figure 5.7, where we treat transcript records as relationships between Student, Course, and Semester entities. An alternative to this design is to represent transcript records as entities and use a new relationship type, Enrolled, to connect them. This alternative is represented in Figure 5.12. Here we incorporate some of the attributes for the entity Semester, as discussed earlier. We also add an extra attribute, Credits, to the relationship Enrolled. Clearly, the two diagrams represent the same information (except for the extra attributes added to Figure 5.12), but which one is better?

As with the “entity versus attribute” dilemma, the choice largely depends on your taste. However, a number of points are worth considering. For instance, it is a good idea to keep the total number of entities and relations as small as possible because it is directly related to the number of relations that will result when the E-R diagram is converted to the relational model. Generally, it is not too serious a problem if two relations are lumped together at this stage because the relational design theory presented in Chapter 8 is geared to identifying relation schemas that must be split and to providing algorithms for doing that. On the other hand, it is much harder to spot the opposite problem: needlessly splitting one relation into two or more.

Coming back to Figure 5.12, we notice that there is a participation constraint for the entity Transcript in the relationship type Enrolled. Moreover, the arrow leading from Transcript to Enrolled indicates that the Transcript role forms a key of the Enrolled relationship. Therefore, there is a one-to-one correspondence between the relationships of type Enrolled and the entities of type Transcript. This means that relationships of type Enrolled can be viewed as superfluous,

---

Figure 5.12 An alternative representation of the transcript information.
because Transcript entities can be used instead to relate the entities of types Student, Course, and Semester. All that is required (in order not to lose information) is to transfer the proper attributes of Enrolled to Transcript after converting the latter into a relationship.

As a result of this discussion, we have the following rule:

Consider a relationship type, $R$, that relates the entity types $E_1, \ldots, E_n$, and suppose that $E_i$ is attached to $R$ via a role that (by itself) forms a key of $R$, and that a participation constraint exists between $E_i$ and $R$. Then it might be possible to collapse $E_i$ and $R$ into a new relationship type that involves only the entity types $E_2, \ldots, E_n$.

Note that this rule is only an indication that $E_i$ can be collapsed into $R$, not a guarantee that this is possible. For instance, $E_i$ might be involved in some other relationship, $R'$. In that case, collapsing $E_i$ into $R$ leaves an edge that connects two relationship types, $R$ and $R'$, which is not allowed by the construction rules for E-R diagrams. Such is the situation of the Broker and Account entities in Figure 5.10: The above rule suggests that Broker can be collapsed into WorksIn and Account can be collapsed into HandledBy. However, both Broker and Account are involved in two different relationships, and each such collapse leaves us with a diagram where two relationships, WorksIn and HandledBy or HasAccount and HandledBy, are directly connected by an edge. On the other hand, the Transaction entity type in Figure 5.11 can be collapsed into the Trade relationship type.

**Information loss.** We have seen examples where the arity of a relationship might change by demoting an entity to an attribute or by collapsing an entity into a relationship. In all of these cases, however, the transformations obviously preserve the diagrams’ information content. Now we are going to discuss some typical situations where seemingly innocuous transformations cause information loss; that is, they lead to diagrams with subtly different information content.

Consider the Parts/Supplier/Project diagram of Figure 5.3. Some designers do not like ternary relationships, preferring to deal with multiple binary relationships instead. Such a decision might lead to the diagram shown in Figure 5.13.
Although superficially the new diagram seems equivalent to the original, there are several subtle differences. First, the new design introduces a navigation trap of the kind we saw in the stock trading example: It is possible that a supplier, Acme, sells “Screw” and that Acme has sold something to project “Screw Driving.” It is even possible that the screw driving project uses screws of the kind Acme sells. However, from the relationships represented in the diagram it is not possible to conclude that it was Acme who sold these screws to the project. All we can tell is that Acme might have done so.

The other problem with the new design is that the price attribute is now associated with the relationship Supplies. This implies that a supplier has a fixed price for each item regardless of the project to which that item is sold. In contrast, the original design in Figure 5.3, page 96, supports different pricing for different projects (e.g., based on quantity). Similarly, the new design allows only one transaction between any supplier and project on any given day because each transaction is represented as a triple \( p, s; d \), so there is no way to distinguish among different transactions between the same parties on the same day. The original design, on the other hand, allows several such deals, provided that different parts are involved.

Having realized the danger of introducing navigation traps, we might be inclined to use higher-degree relationships whenever possible. For instance, in Figure 5.10 we might want to try eliminating the navigation trap caused by the relationships HasAccount, WorksIn, and HandleBy by collapsing these three relationships into one. However, this transformation introduces more problems than it solves. For instance, if this transformation keeps the arrow that connects Broker and WorksIn, we unwittingly introduce the constraint that a broker can have at most one account and at most one client. If we do not keep this arrow, we lose the constraint that each broker is assigned to exactly one office. This transformation also makes it impossible to have brokers who have no accounts and accounts that have no brokers.

**E-R and object databases.** Although we will not discuss object databases until Chapter 16, we briefly mention here that some of the difficult issues involved in translating E-R diagrams into schemas become easier for object databases.

- In Section 5.2, we discussed the issues involved in representing entities with set-valued attributes in a relational database. The objects stored in an object database can have set-valued attributes, so the representation of such entities in the schema of the object database is considerably easier.

- In Section 5.4, we discussed the issues involved in representing the IsA relationship in a relational database. Object databases allow a direct representation of the IsA relationship within the schema, so, again, representation of such relationships is considerably easier.

From these examples, it should be apparent that not only is it generally easier to translate E-R diagrams into schemas for object databases than into schemas for
relational databases, but for many applications object databases allow a much more intuitive model of the enterprise than do relational databases.

## 5.7 CASE STUDY: A DESIGN FOR THE STUDENT REGISTRATION SYSTEM

In this section, we present a database design for the Student Registration System. This design might be included in the Design Document for the entire application, as will be discussed in Section 12.1.

The first step in the design process is to construct the E-R diagram shown in Figure 5.14, which is a model of the student registration enterprise as described in the Requirements Document (Section 3.2). Note from the figure that

- **STUDENT** is related to **CLASS** through **TRANSSCRIPT**, which signifies that a student is registered, is enrolled, or has completed a class in some semester.
FACULTY is related to CLASS through TEACHES, meaning that a faculty member teaches a class in some semester and every class is taught by exactly one faculty member.

COURSE is related to COURSE through the relationship REQUIRES; that is, a course can be a prerequisite for another course in some semester. The attribute EnforcedSince specifies the date when the prerequisite became in force.

CLASS is related to CLASSROOM through TAUGHTIN; that is, a class is taught in a classroom in some semester.

A class (i.e., a particular offering of a course) uses at most one textbook, because the attribute Textbook is not part of the key of the entity CLASS. In Section 8.12, we will remove this restriction and discuss the impact of this decision on the system design.

On the basis of this E-R diagram and the list of integrity constraints given in the Requirements Document in Section 3.2.4, the next step is to produce the schema shown in Figures 5.15 and 5.16. The translation was done in a straightforward manner using the techniques described so far in this chapter. Note that tables were not created to correspond to the TEACHES and TAUGHTIN relationships because both have only roles, no attributes, and both are many-to-one. Therefore, the information about these relationships can be stored in the CLASS table—the tuple corresponding to each class in the CLASS table contains the ClassroomId of the room where it is taught and the InstructorId of the faculty member who teaches it.

Note that some of the integrity constraints specified in the Requirements Document are checked within CREATE TABLE statements (specifically constraints 1, 3, 5, and 6 from Section 3.2.4). In the complete schema design, other integrity constraints are checked with separate CREATE ASSERTION statements and with one trigger. However, this part of the design requires SQL constructs that we have not yet discussed, and so we will complete the schema design in Section 12.6 after we cover these constructs.

We selected this particular design for inclusion in the book because it is straightforward and transparent. In practice, such a design might be a starting point for a number of enhancements whose goal is to capture more features and increase the efficiency of the final implementation.

Let us discuss a few possible enhancements and alternatives. Consider course dependencies. One obvious omission in our schema is the corequisite relationship and all of the constraints entailed by it. More subtly, university curricula change all the time: New courses are introduced, old courses are removed, and prerequisite dependencies between courses evolve in time. Thus, the REQUIRES relationship in Figure 5.14 might need two additional attributes, Start and End, to designate the period when the prerequisite relationship is effective. Even more interesting is the possibility that a particular prerequisite relationship might exist at different times. For instance, course A might be a prerequisite for course B between 1985 and 1990 and again between 1999 and the present. Modeling this situation is left to Exercise 5.10.
Figure 5.15  A schema for the Student Registration System—Part 1.

```sql
CREATE TABLE Student  
  (Id  CHAR(9) NOT NULL,  
   Name  CHAR(20) NOT NULL,  
   Password  CHAR(10) NOT NULL,  
   Address  CHAR(50),  
   PRIMARY KEY (Id) )

CREATE TABLE Faculty  
  (Id  CHAR(9) NOT NULL,  
   Name  CHAR(20) NOT NULL,  
   DeptId  CHAR(4) NOT NULL,  
   Password  CHAR(10) NOT NULL,  
   Address  CHAR(50),  
   PRIMARY KEY (Id) )

CREATE TABLE Course (  
  CrsCode  CHAR(6) NOT NULL,  
  DeptId  CHAR(4) NOT NULL,  
  CrsName  CHAR(20) NOT NULL,  
  CreditHours  INTEGER NOT NULL,  
  PRIMARY KEY (CrsCode),  
  UNIQUE (CrsCode),  
  PRIMARY KEY (CrsCode, DeptId) )

CREATE TABLE WhenOffered (  
  CrsCode  CHAR(6) NOT NULL,  
  Semester  CHAR(6) NOT NULL,  
  PRIMARY KEY (CrsCode, Semester),  
  CHECK (Semester IN ('Spring','Fall')) )

CREATE TABLE Classroom (  
  ClassroomId  CHAR(3) NOT NULL,  
  Seats  INTEGER NOT NULL,  
  PRIMARY KEY (ClassroomId) )
```

Another interesting enhancement can be to account for the possibility that certain highly popular courses might be restricted to certain majors only. In this situation, the E-R diagram and the corresponding tables have to include information about the student majors (a set-valued attribute!), and this has to be consistent with the majors allowed in the courses. This enhancement is left to Exercise 5.11.

Enhancements to express more complex requirements are one source of modifications to the proposed design. Another source is the vast range of possible alternative designs, which might have implications for the overall performance of the system. We discuss one such alternative and its implications.
Figure 5.16 A schema for the Student Registration System—Part 2.

CREATE TABLE Requires (
    CrsCode CHAR(6) NOT NULL,
    PrereqCrsCode CHAR(6) NOT NULL,
    EnforcedSince DATE NOT NULL,
    PRIMARY KEY (CrsCode, PrereqCrsCode),
    FOREIGN KEY (CrsCode) REFERENCES Course(CrsCode),
    FOREIGN KEY (PrereqCrsCode) REFERENCES Course(CrsCode) )

CREATE TABLE Class (
    CrsCode CHAR(6) NOT NULL,
    SectionNo INTEGER NOT NULL,
    Semester CHAR(6) NOT NULL,
    Year INTEGER NOT NULL,
    Textbook CHAR(50),
    ClassTime CHAR(5),
    Enrollment INTEGER,
    MaxEnrollment INTEGER,
    ClassroomId CHAR(3),
    InstructorId CHAR(9),
    PRIMARY KEY (CrsCode,SectionNo,Semester,Year),
    CONSTRAINT TimeConflict UNIQUE (InstructorId,Semester,Year,ClassTime),
    CONSTRAINT ClassroomConflict UNIQUE (ClassroomId,Semester,Year,ClassTime),
    CONSTRAINT Enrollment CHECK (Enrollment <= MaxEnrollment AND Enrollment >= 0),
    FOREIGN KEY (CrsCode) REFERENCES Course(CrsCode),
    FOREIGN KEY (ClassroomId) REFERENCES Classroom(ClassroomId),
    FOREIGN KEY (CrsCode, Semester)
    REFERENCES WhenOffered(CrsCode, Semester),
    FOREIGN KEY (InstructorId) REFERENCES Faculty(Id) )

CREATE TABLE Transcript (
    StudId CHAR(9) NOT NULL,
    CrsCode CHAR(6) NOT NULL,
    SectionNo INTEGER NOT NULL,
    Semester CHAR(6) NOT NULL,
    Year INTEGER NOT NULL,
    Grade CHAR(1),
    PRIMARY KEY (StudId,CrsCode,SectionNo,Semester,Year),
    FOREIGN KEY (StudId) REFERENCES Student(Id),
    FOREIGN KEY (CrsCode,SectionNo,Semester,Year)
    REFERENCES Class(CrsCode,SectionNo,Semester,Year),
    CHECK (Grade IN ('A','B','C','D','F','I')) ,
    CHECK (Semester IN ('Spring','Fall')) )
Consider the attribute SemestersOffered of entity Course. Because it is a set-valued attribute, we translate it using a separate table, WhenOffered. We chose this particular design because we saw that it leads to a particularly easy way to express the constraint that the semester when any particular class is taught must be one of the semesters when that course is offered. For instance, it should not be possible for course CS305 to be offered only in spring semesters but for a certain class of this course to be taught in fall 2000. However, this should be allowed if CS305 is offered in both spring and fall semesters.

In our design, this requirement is expressed simply as a foreign key constraint in relation Class:

\[
\text{FOREIGN KEY (CrsCode, Semester) REFERENCES WhenOffered(CrsCode, Semester)}
\]

This constraint says that if a class of course with code crscode is offered during a semester, sem, then \( \langle \text{crscode, sem} \rangle \) should be a tuple in the relation WhenOffered; that is, sem must be one of the allowed semesters for the course.

Despite the simplicity of this design, one might feel that creating a separate relation for such a trivial piece of information is unacceptable overhead. Such a separate relation requires an extra operation for certain queries, additional storage requirements, and the like.\(^9\) An alternative is to define a new SQL domain (introduced in Chapter 4), which contains just three values:

\[
\text{CREATE DOMAIN Semesters CHAR(6) CHECK (VALUE IN ('Spring', 'Fall', 'Both'))}
\]

The set-valued attribute SemestersOffered of the entity Course is now single-valued, but it ranges over the domain Semesters. The advantage is that the translation into the relational model is more straightforward and there is no need for the extra relation WhenOffered. However, it is now more difficult to specify the constraint that a class can be taught only in the semesters when the corresponding course is offered. Details are left to Exercise 5.12.

Finally, let us consider the possible alternatives for representing the current and the next semesters, as is required by some transactions specified in the Requirements Document in Section 3.2. In fact, our design has no obvious way to represent this information. One simple way to tell which semester is current or next is to create a separate relation to store this information. However, this would not be a good design because every reference to the current or to the next semester would require a relatively expensive database query. The right way to do this type of thing is to use the function \text{CURRENT_DATE} provided by SQL and the function \text{EXTRACT} to extract particular fields from that date. For instance,

\[
\text{EXTRACT(YEAR FROM CURRENT_DATE)}
\]

\(^9\)In our particular case, none of these disadvantages seems to apply because, in all likelihood, the relation WhenOffered will be mostly used to verify the above foreign key constraint and having a separate relation for course-semester pairs is a very efficient way to check this constraint.
EXTRACT(MONTHS FROM CURRENT_DATE)

returns the numeric values of the current year and month. This should be sufficient to determine whether any given semester is current or next.

5.8 BIBLIOGRAPHIC NOTES

The entity-relationship approach was introduced in [Chen 1976]. Since then it has received considerable attention and various extensions have been proposed (see, for example, research papers in [Spaccapietra 1987]). Conceptual design using the E-R model has also been advanced significantly. The reader is referred to [Teorey 1992, Batini et al. 1992, Thalheim 1992] for comprehensive coverage.

More recently, a new design methodology called Unified Modeling Language (UML) was developed and has been gaining popularity [Booch et al. 1999]. UML borrows many ideas from the E-R model, but extends it in the direction of object-oriented modeling. In particular, it provides means to model not only the structure of the data but also the behavioral aspects of programs and how complex applications are to be deployed in complex computing environments. A succinct introduction to UML can be found in [Fowler and Scott 1999].

A number of tools exist to help the database designer with E-R and UML modeling. These tools guide the user through the process of specifying the diagrams, attributes, constraints, and so forth. When all is done, they map the conceptual model into relational tables. Such tools include ERwin from Computer Associates, ER/Studio from Embarcadero Technologies, and Rational Rose from Rational Software. In addition, DBMS vendors provide their own design tools, such as Oracle Designer from Oracle Corporation and PowerDesigner from Sybase.

5.9 EXERCISES

5.1 Suppose that you decide to convert IsA hierarchies into the relational model by adding a new attribute (such as Status in the case of STUDENT entities, as described in Section 5.4.1 on page 103). What kind of problems exist if subentities are not disjoint (e.g., if a secretary can also be a technician)? What problems exist if the covering constraint does not hold (e.g., if some employees are not classified as either secretary or technician)?

5.2 Construct your own example of an E-R diagram whose direct translation into the relational model has an anomaly similar to that of the PERSON entity (see the discussion regarding Figures 5.1 and 5.2).

5.3 Represent the IsA hierarchy in Figure 5.5 in the relational model. For each IsA relationship discuss your choice of the representation technique (i.e., either the technique where each entity in the hierarchy is represented by a separate relation or the one where the entity and its subentities are in the same relation and are distinguished using a new attribute). Discuss the circumstances
when an alternative representation (to the one you have chosen) would be better.

5.4 Translate the brokerage example of Section 5.5 into an SQL-92 schema. Use the necessary SQL-92 machinery to express all of the constraints specified in the E-R model.

5.5 Identify the navigation traps present in the diagram of Figure 5.9.

5.6 Consider the following database schema:

- **Supplier**
  - (SName, ItemName, Price)
  - supplier SName sells item ItemName at Price

- **Customer**
  - (CName, Address)
  - customer CName lives at Address.

- **Order**
  - (CName, SName, ItemName, Qty)
  - customer CName has ordered Qty of item ItemName from supplier SName.

- **Item**
  - (ItemName, Description)
  - information about items.

a. Draw the E-R diagram from which the above schema might have been derived. Specify the keys.

b. Suppose now that you want to add the following constraint to this diagram: Every item is supplied by some supplier. Modify the diagram to accommodate this constraint. Also show how this new diagram can be translated back to the relational model.

5.7 Use the E-R approach to model the operations of your local community library. The library has books, CDs, tapes, and so forth, which are lent to library patrons. The latter have accounts, addresses, and so forth. If a loaned item is overdue, it accumulates penalty. However, some patrons are minors, so they must have sponsoring patrons who are responsible for paying penalties (or replacing a book in case of a loss).

5.8 Design the E-R diagram for a real estate brokerage firm. The firm keeps track of the houses for sale, and it has customers looking to buy a house. A house for sale can be listed with this firm or with a different one. Being “listed” with a firm means that the house owner has a contract with an agent who works for that firm. Each house on the market has price, address, owner, and list of characteristics, such as the number of bedrooms, bathrooms, type of heating, appliances, size of garage, and the like. This list can be different for different houses, and some attributes can be present for some houses but missing for others. Likewise, each customer has preferences that are expressed in the same terms (the number of bedrooms, bathrooms, etc.). Apart from these preferences customers specify the price range of houses they are interested in.

5.9 A supermarket chain is interested in building a decision support system with which they can analyze the sales of different products in different supermarkets at different times. Each supermarket is in a city, which is in a state, which is in a region. Time can be measured in days, months, quarters,
and years. Products have names and categories (produce, canned goods, etc.). Design an E-R diagram for this application.

5.10 Modify the E-R diagram for the Student Registration Schema in Figure 5.14 to include corequisite and prerequisite relationships that exist over multiple periods of time. Each period begins in a certain semester and year and ends in a certain semester and year, or it continues into the present. Modify the translation into the relational model appropriately.

5.11 Modify the E-R diagram for the Student Registration System in Figure 5.14 to include information about the student majors and the majors allowed in courses. A student can have several majors (which are codes of the various programs in the university, such as CSE, ISE, MUS, ECO). A course can also have several admissible majors, or the list of admissible majors can be empty. In the latter case, anyone is admitted into the course. Express the constraint that says that a course with restrictions on majors can have only those students who hold one of the allowed majors.

Alas, in full generality this constraint can be expressed only as an SQL assertion (introduced in Section 4.3) that uses features to be discussed in Section 6.2. However, it is possible to express this constraint under the following simplifying assumption: When a student registers for a course, she must state the major for which the course is going to be taken.

Modify the schema to reflect this simplifying assumption and then express the aforesaid integrity constraint.

5.12 Make the necessary modifications to the schema of the Student Registration System to reflect the design that uses the SQL domain Semesters, as discussed at the end of Section 5.7. Express the constraint that a class can be taught only during the semesters in which the corresponding course is offered. For instance, if the value of the attribute SemestersOffered for the course CS305 is Both, then the corresponding classes can be taught in the spring and the fall semesters. However, if the value of that attribute is Spring then these classes can be taught only in the spring.

5.13 Design an E-R model for the following enterprise. Various organizations make business deals with various other organizations. (For simplicity, let us assume that there are only two parties to each deal.) When negotiating (and signing) a deal, each organization is represented by a lawyer. The same organization can have deals with many other organizations and it might use different lawyers in each case. Lawyers and organizations have various attributes, like address and name. They also have their own unique attributes, such as specialization and fee in case of a lawyer and budget in case of an organization.

Show how information loss can occur if a relationship of degree higher than two is split into binary relationship. Discuss assumption under which such a split does not lead to a loss of information.