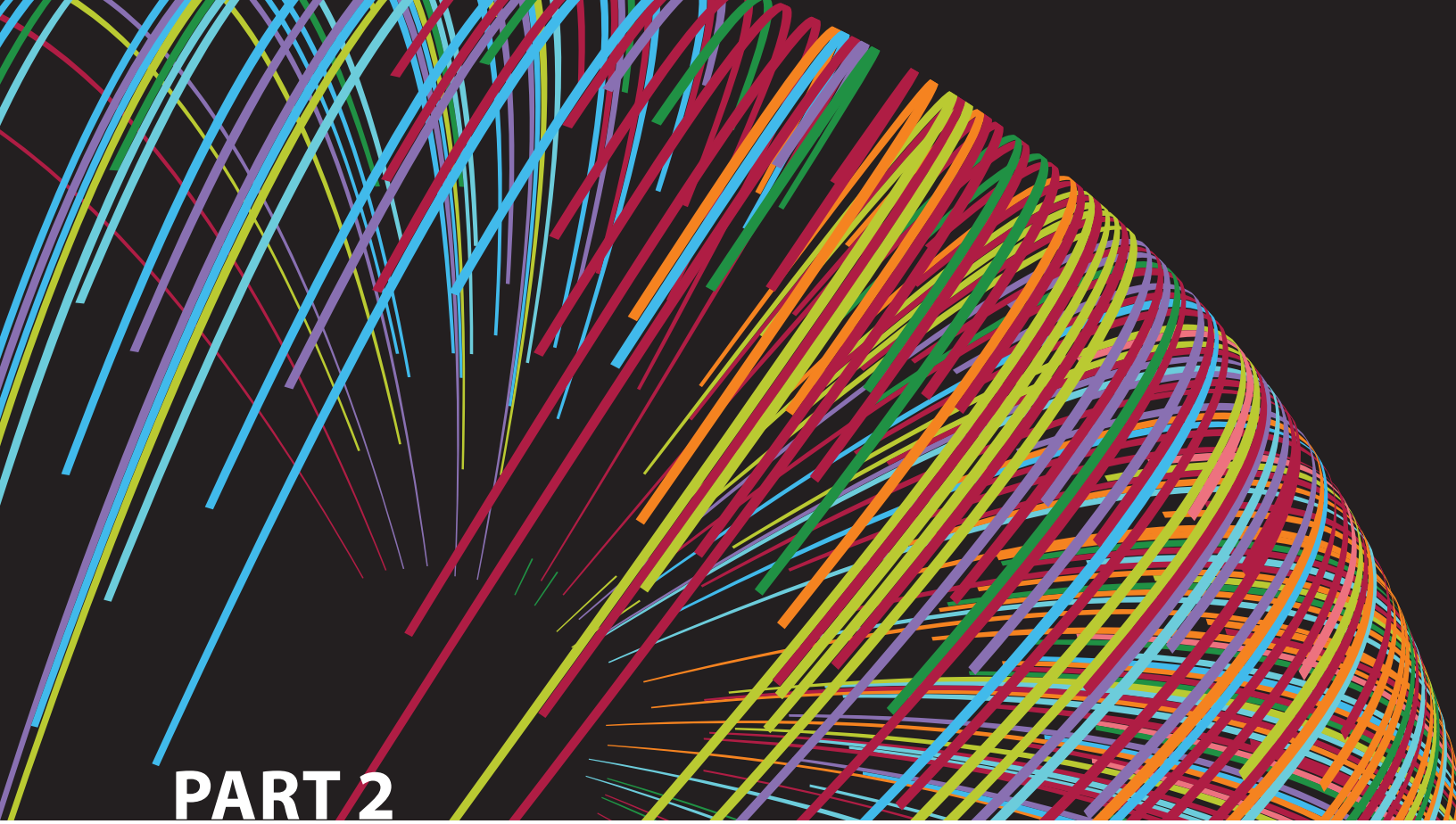


PART 2
Design Concepts

Chapter 3

The Relational Database Model

After completing this chapter, you will be able to:

- Describe the relational database model's logical structure
- Identify the relational model's basic components and explain the structure, contents, and characteristics of a relational table
- Use relational database operators to manipulate relational table contents
- Explain the purpose and components of the data dictionary and system catalog
- Identify appropriate entities and then the relationships among the entities in the relational database model
- Describe how data redundancy is handled in the relational database model
- Explain the purpose of indexing in a relational database

Preview

In this chapter, you will learn about the relational model's logical structure and more about how entity relationship diagrams (ERDs) can be used to design a relational database. You will also learn how the relational database's basic data components fit into a logical construct known as a table, and how tables within a database can be related to one another.

After learning about tables, their components, and their relationships, you will be introduced to basic table design concepts and the characteristics of well-designed and poorly designed tables. These concepts will become your gateway to the next few chapters.

Data Files and Available Formats

	MS Access	Oracle	MS SQL	My SQL		MS Access	Oracle	MS SQL	My SQL
CH03_CollegeTry	✓	✓	✓	✓	CH03_AviaCo	✓	✓	✓	✓
CH03_CollegeTry2	✓	✓	✓	✓	CH03_BeneCo	✓	✓	✓	✓
CH03_InsureCo	✓	✓	✓	✓	CH03_CollegeQue	✓	✓	✓	✓
CH03_Museum	✓	✓	✓	✓	CH03_NoComp	✓	✓	✓	✓
CH03_SaleCo	✓	✓	✓	✓	CH03_StoreCo	✓	✓	✓	✓
CH03_TinyCollege	✓	✓	✓	✓	CH03_Theater	✓	✓	✓	✓
CH03_Relational_DB	✓	✓	✓	✓	CH03_TransCo	✓	✓	✓	✓
					CH03_VendingCo	✓	✓	✓	✓

Data Files Available on cengagebrain.com

Note

The relational model, introduced by E. F. Codd in 1970, is based on predicate logic and set theory. **Predicate logic**, used extensively in mathematics, provides a framework in which an assertion (statement of fact) can be verified as either true or false. For example, suppose that a student with a student ID of 12345678 is named Melissa Sanduski. This assertion can easily be demonstrated to be true or false. **Set theory** is a mathematical science that deals with sets, or groups of things, and is used as the basis for data manipulation in the relational model. For example, assume that set A contains three numbers: 16, 24, and 77. This set is represented as A(16, 24, 77). Furthermore, set B contains four numbers, 44, 77, 90, and 11, and so is represented as B(44, 77, 90, 11). Given this information, you can conclude that the intersection of A and B yields a result set with a single number, 77. This result can be expressed as A ∩ B = 77. In other words, A and B share a common value, 77.

Based on these concepts, the relational model has three well-defined components:

1. A logical data structure represented by relations (see Sections 3-1, 3-2, and 3-5)
2. A set of integrity rules to enforce that the data is consistent and remains consistent over time (see Sections 3-3, 3-6, 3-7, and 3-8)
3. A set of operations that defines how data is manipulated (see Section 3-4)

3-1 A Logical View of Data

In Chapter 1, Database Systems, you learned that a database stores and manages both data and metadata. You also learned that the DBMS manages and controls access to the data and the database structure. Such an arrangement—placing the DBMS between the application and the database—eliminates most of the file system's inherent limitations. The result of such flexibility, however, is a far more complex physical structure. In fact, the database structures required by both the hierarchical and network database models often become complicated enough to diminish efficient database design. The relational data model changed all of that by allowing the designer to focus on the logical representation of the data and its relationships, rather than on the physical storage details. To use an automotive analogy, the relational database uses an automatic transmission to relieve you of the need to manipulate clutch pedals and gearshifts. In short, the relational model enables you to view data *logically* rather than *physically*.

The practical significance of taking the logical view is that it serves as a reminder of the simple file concept of data storage. Although the use of a table, quite unlike that of a file, has the advantages of structural and data independence, a table does resemble a file from a conceptual point of view. Because you can think of related records as being stored in independent tables, the relational database model is much easier to understand than the hierarchical and network models. Logical simplicity tends to yield simple and effective database design methodologies.

Because the table plays such a prominent role in the relational model, it deserves a closer look. Therefore, our discussion begins by exploring the details of table structure and contents.

predicate logic
Used extensively in mathematics to provide a framework in which an assertion (statement of fact) can be verified as either true or false.

set theory
A part of mathematical science that deals with sets, or groups of things, and is used as the basis for data manipulation in the relational model.

3-1a Tables and Their Characteristics

The logical view of the relational database is facilitated by the creation of data relationships based on a logical construct known as a relation. Because a relation is a mathematical construct, end users find it much easier to think of a relation as a table. A *table* is perceived as a two-dimensional structure composed of rows and columns. A table is also

called a *relation* because the relational model's creator, E. F. Codd, used the two terms as synonyms. You can think of a table as a *persistent* representation of a logical relation—that is, a relation whose contents can be permanently saved for future use. As far as the table's user is concerned, a table contains *a group of related entity occurrences*—that is, an entity set. For example, a STUDENT table contains a collection of entity occurrences, each representing a student. For that reason, the terms *entity set* and *table* are often used interchangeably.

Note

The word *relation*, also known as a *dataset* in Microsoft Access, is based on the mathematical set theory from which Codd derived his model. Because the relational model uses attribute values to establish relationships among tables, many database users incorrectly assume that the term *relation* refers to such relationships. Many then incorrectly conclude that only the relational model permits the use of relationships.

You will discover that the table view of data makes it easy to spot and define entity relationships, thereby greatly simplifying the task of database design. The characteristics of a relational table are summarized in Table 3.1.

TABLE 3.1

CHARACTERISTICS OF A RELATIONAL TABLE

1	A table is perceived as a two-dimensional structure composed of rows and columns.
2	Each table row (**tuple**) represents a single entity occurrence within the entity set.
3	Each table column represents an attribute, and each column has a distinct name.
4	Each intersection of a row and column represents a single data value.
5	All values in a column must conform to the same data format.
6	Each column has a specific range of values known as the **attribute domain**.
7	The order of the rows and columns is immaterial to the DBMS.
8	Each table must have an attribute or combination of attributes that uniquely identifies each row.

Note

Relational database terminology is very precise. Unfortunately, file system terminology sometimes creeps into the database environment. Thus, rows are sometimes referred to as *records*, and columns are sometimes labeled as *fields*. Occasionally, tables are labeled *files*. Technically speaking, this substitution of terms is not always appropriate. The database table is a logical concept rather than a physical concept, and the terms *file*, *record*, and *field* describe physical concepts. Nevertheless, as long as you recognize that the table is actually a logical concept rather than a physical construct, you may think of table rows as records and table columns as fields. In fact, many database software vendors still use this familiar file system terminology.

tuple
In the relational model, a table row.

attribute domain
In data modeling, the construct used to organize and describe an attribute's set of possible values.

The database table shown in Figure 3.1 illustrates the characteristics listed in Table 3.1.

FIGURE 3.1 STUDENT TABLE ATTRIBUTE VALUES

Table name: STUDENT **Database name: Ch03_TinyCollege**

STU_NUM	STU_LNAME	STU_FNAME	STU_INIT	STU_DOB	STU_HRS	STU_CLASS	STU_GPA	STU_TRANSFER	DEPT_CODE	STU_PHONE	PROF_NUM
321452	Bowser	William	C	12-Feb-1985	42	So	2.84	No	BIOL	2134	205
324257	Smithson	Anne	K	15-Nov-1991	81	Jr	3.27	Yes	CIS	2256	222
324258	Brewer	Juliette		23-Aug-1979	36	So	2.26	Yes	ACCT	2256	228
324269	Oblonski	Walter	H	16-Sep-1986	66	Jr	3.09	No	CIS	2114	222
324273	Smith	John	D	30-Dec-1968	102	Sr	2.11	Yes	ENGL	2231	199
324274	Katinga	Raphael	P	21-Oct-1989	114	Sr	3.15	No	ACCT	2267	228
324291	Robertson	Gerald	T	08-Apr-1983	120	Sr	3.87	No	EDU	2267	311
324299	Smith	John	B	30-Nov-1996	15	Fr	2.92	No	ACCT	2315	230

STU_NUM = **Student number**
STU_LNAME = **Student last name**
STU_FNAME = **Student first name**
STU_INIT = **Student middle initial**
STU_DOB = **Student date of birth**
STU_HRS = **Credit hours earned**
STU_CLASS = **Student classification**
STU_GPA = **Grade point average**
STU_TRANSFER = **Student transferred from another institution**
DEPT_CODE = **Department code**
STU_PHONE = **4-digit campus phone extension**
PROF_NUM = **Number of the professor who is the student's advisor**

Using the STUDENT table shown in Figure 3.1, you can draw the following conclusions corresponding to the points in Table 3.1:

1. The STUDENT table is perceived to be a two-dimensional structure composed of 8 rows (tuples) and 12 columns (attributes).
2. Each row in the STUDENT table describes a single entity occurrence within the entity set. (The entity set is represented by the STUDENT table.) For example, row 4 in Figure 3.1 describes a student named Walter H. Oblonski. Given the table contents, the STUDENT entity set includes eight distinct entities (rows) or students.
3. Each column represents an attribute, and each column has a distinct name.
4. All of the values in a column match the attribute's characteristics. For example, the grade point average (STU_GPA) column contains only STU_GPA entries for each of the table rows. Data must be classified according to its format and function. Although various DBMSs can support different data types, most support at least the following:
   a. *Numeric*. You can use numeric data to perform meaningful arithmetic procedures. For example, in Figure 3.1, STU_HRS and STU_GPA are numeric attributes.
   b. *Character*. Character data, also known as text data or string data, can contain any character or symbol not intended for mathematical manipulation. In Figure 3.1, STU_CLASS and STU_PHONE are examples of character attributes.
   c. *Date*. Date attributes contain calendar dates stored in a special format known as the Julian date format. In Figure 3.1, STU_DOB is a date attribute.
   d. Logical. Logical data can only have true or false (yes or no) values. In Figure 3.1, the STU_TRANSFER attribute uses a logical data format.
5. The column's range of permissible values is known as its domain. Because the STU_GPA values are limited to the range 0–4, inclusive, the domain is [0,4].
6. The order of rows and columns is immaterial to the user.

Online Content

The databases used to illustrate the material in this chapter (see the Data Files list at the beginning of the chapter) are available at *www.cengagebrain.com*. The database names match the database names shown in the figures.

7. Each table must have a primary key. In general terms, the **primary key (PK)** is an attribute or combination of attributes that uniquely identifies any given row. In this case, STU_NUM (the student number) is the primary key. Using the data in Figure 3.1, observe that a student's last name (STU_LNAME) would not be a good primary key because several students have the last name of Smith. Even the combination of the last name and first name (STU_FNAME) would not be an appropriate primary key because more than one student is named John Smith.

primary key (PK)
In the relational model, an identifier composed of one or more attributes that uniquely identifies a row. Also, a candidate key selected as a unique entity identifier. See also *key*.

key
One or more attributes that determine other attributes. See also *candidate key*, *foreign key*, *primary key (PK)*, *secondary key*, and *superkey*.

determination
The role of a key. In the context of a database table, the statement "A determines B" indicates that knowing the value of attribute A means that the value of attribute B can be looked up.

functional dependence
Within a relation R, an attribute B is functionally dependent on an attribute A if and only if a given value of attribute A determines exactly one value of attribute B. The relationship "B is dependent on A" is equivalent to "A determines B" and is written as A → B.

determinant
Any attribute in a specific row whose value directly determines other values in that row. See also *Boyce-Codd normal form (BCNF)*.

dependent
An attribute whose value is determined by another attribute.

3-2 Keys

In the relational model, keys are important because they are used to ensure that each row in a table is uniquely identifiable. They are also used to establish relationships among tables and to ensure the integrity of the data. A **key** consists of one or more attributes that determine other attributes. For example, an invoice number identifies all of the invoice attributes, such as the invoice date and the customer name.

One type of key, the primary key, has already been introduced. Given the structure of the STUDENT table shown in Figure 3.1, defining and describing the primary key seem simple enough. However, because the primary key plays such an important role in the relational environment, you will examine the primary key's properties more carefully. In this section, you also will become acquainted with superkeys, candidate keys, and secondary keys.

3-2a Dependencies

The role of a key is based on the concept of determination. **Determination** is the state in which knowing the value of one attribute makes it possible to determine the value of another. The idea of determination is not unique to the database environment. You are familiar with the formula revenue − cost = profit. This is a form of determination, because if you are given the *revenue* and the *cost*, you can determine the *profit*. Given *profit* and *revenue*, you can determine the *cost*. Given any two values, you can determine the third. Determination in a database environment, however, is not normally based on a formula but on the relationships among the attributes.

If you consider what the attributes of the STUDENT table in Figure 3.1 actually represent, you will see a relationship among the attributes. If you are given a value for STU_NUM, then you can determine the value for STU_LNAME because one and only one value of STU_LNAME is associated with any given value of STU_NUM. A specific terminology and notation is used to describe relationships based on determination. The relationship is called **functional dependence**, which means that the value of one or more attributes determines the value of one or more other attributes. The standard notation for representing the relationship between STU_NUM and STU_LNAME is as follows:

STU_NUM → STU_LNAME

In this functional dependency, the attribute whose value determines another is called the **determinant** or the key. The attribute whose value is determined by the other attribute is called the **dependent**. Using this terminology, it would be correct to say that STU_NUM is the determinant and STU_LNAME is the dependent. STU_NUM functionally determines STU_LNAME, and STU_LNAME is functionally dependent on STU_NUM. As stated earlier, functional dependence can involve a determinant that comprises more than one attribute and multiple dependent attributes. Refer to the STUDENT table for the following example:

STU_NUM → (STU_LNAME, STU_FNAME, STU_GPA)

and

(STU_FNAME, STU_LNAME, STU_INIT, STU_PHONE) → (STU_DOB, STU_HRS, STU_GPA)

Determinants made of more than one attribute require special consideration. It is possible to have a functional dependency in which the determinant contains attributes that are not necessary for the relationship. Consider the following two functional dependencies:

STU_NUM → STU_GPA

(STU_NUM, STU_LNAME) → STU_GPA

In the second functional dependency, the determinant includes STU_LNAME, but this attribute is not necessary for the relationship. The functional dependency is valid because given a pair of values for STU_NUM and STU_LNAME, only one value would occur for STU_GPA. A more specific term, **full functional dependence**, is used to refer to functional dependencies in which the entire collection of attributes in the determinant is necessary for the relationship. Therefore, the dependency shown in the preceding example is a functional dependency, but not a full functional dependency.

3-2b Types of Keys

Recall that a key is an attribute or group of attributes that can determine the values of other attributes. Therefore, keys are determinants in functional dependencies. Several different types of keys are used in the relational model, and you need to be familiar with them.

A **composite key** is a key that is composed of more than one attribute. An attribute that is a part of a key is called a **key attribute**. For example,

STU_NUM → STU_GPA

(STU_LNAME, STU_FNAME, STU_INIT, STU_PHONE) → STU_HRS

In the first functional dependency, STU_NUM is an example of a key composed of only one key attribute. In the second functional dependency, (STU_LNAME, STU_FNAME, STU_INIT, STU_PHONE) is a composite key composed of four key attributes.

A **superkey** is a key that can uniquely identify any row in the table. In other words, a superkey functionally determines every attribute in the row. In the STUDENT table, STU_NUM is a superkey, as are the composite keys (STU_NUM, STU_LNAME), (STU_NUM, STU_LNAME, STU_INIT) and (STU_LNAME, STU_FNAME, STU_INIT, STU_PHONE). In fact, because STU_NUM alone is a superkey, any composite key that has STU_NUM as a key attribute will also be a superkey. Be careful, however, because not all keys are superkeys. For example, Gigantic State University determines its student classification based on hours completed, as shown in Table 3.2.

Therefore, you can write STU_HRS → STU_CLASS.

However, the specific number of hours is not dependent on the classification. It is quite possible to find a junior with 62 completed hours or one with 84 completed hours. In other words, the classification (STU_CLASS) does not determine one and only one value for completed hours (STU_HRS).

full functional dependence
A condition in which an attribute is functionally dependent on a composite key but not on any subset of the key.

composite key
A multiple-attribute key.

key attribute
An attribute that is part of a primary key. See also *prime attribute*.

superkey
An attribute or attributes that uniquely identify each entity in a table. See *key*.

TABLE 3.2

STUDENT CLASSIFICATION

HOURS COMPLETED	CLASSIFICATION
Less than 30	Fr
30–59	So
60–89	Jr
90 or more	Sr

One specific type of superkey is called a candidate key. A **candidate key** is a minimal superkey—that is, a superkey without any unnecessary attributes. A candidate key is based on a full functional dependency. For example, STU_NUM would be a candidate key, as would (STU_LNAME, STU_FNAME, STU_INIT, STU_PHONE). On the other hand, (STU_NUM, STU_LNAME) is a superkey, but it is not a candidate key because STU_LNAME could be removed and the key would still be a superkey. A table can have many different candidate keys. If the STUDENT table also included the students' Social Security numbers as STU_SSN, then it would appear to be a candidate key. Candidate keys are called *candidates* because they are the eligible options from which the designer will choose when selecting the primary key. The primary key is the candidate key chosen to be the primary means by which the rows of the table are uniquely identified.

Entity integrity is the condition in which each row (entity instance) in the table has its own unique identity. To ensure entity integrity, the primary key has two requirements: (1) all of the values in the primary key must be unique and (2) no key attribute in the primary key can contain a null.

Note

A null is no value at all. It does *not* mean a zero or a space. A null is created when you press the Enter key or the Tab key to move to the next entry without making an entry of any kind. Pressing the Spacebar creates a blank (or a space).

Null values are problematic in the relational model. A **null** is the absence of any data value, and it is never allowed in any part of the primary key. From a theoretical perspective, it can be argued that a table that contains a null is not properly a relational table at all. From a practical perspective, however, some nulls cannot be reasonably avoided. For example, not all students have a middle initial. As a general rule, nulls should be avoided as much as reasonably possible. In fact, an abundance of nulls is often a sign of a poor design. Also, nulls should be avoided in the database because their meaning is not always identifiable. For example, a null could represent any of the following:

- An unknown attribute value
- A known, but missing, attribute value
- A "not applicable" condition

Depending on the sophistication of the application development software, nulls can create problems when functions such as COUNT, AVERAGE, and SUM are used. In addition, nulls can create logical problems when relational tables are linked.

In addition to its role in providing a unique identity to each row in the table, the primary key may play an additional role in the controlled redundancy that allows

candidate key
A minimal superkey; that is, a key that does not contain a subset of attributes that is itself a superkey. See *key*.

entity integrity
The property of a relational table that guarantees each entity has a unique value in a primary key and that the key has no null values.

null
The absence of an attribute value. Note that a null is not a blank.

the relational model to work. Recall from Chapter 2, Data Models, that a hallmark of the relational model is that relationships between tables are implemented through common attributes as a form of controlled redundancy. For example, Figure 3.2 shows PRODUCT and VENDOR tables that are linked through a common attribute, VEND_CODE. VEND_CODE is referred to as a foreign key in the PRODUCT table. A **foreign key (FK)** is the primary key of one table that has been placed into another table to create a common attribute. In Figure 3.2, the primary key of VENDOR, VEND_CODE, was placed in the PRODUCT table; therefore, VEND_CODE is a foreign key in PRODUCT. One advantage of using a proper naming convention for table attributes is that you can identify foreign keys more easily. For example, because the STUDENT table in Figure 3.1 used a proper naming convention, you can identify two foreign keys in the table (DEPT_CODE and PROF_NUM) that imply the existence of two other tables in the database (DEPARTMENT and PROFESSOR) related to STUDENT.

FIGURE 3.2 AN EXAMPLE OF A SIMPLE RELATIONAL DATABASE

Table name: PRODUCT **Database name: Ch03_SaleCo**
Primary key: PROD_CODE
Foreign key: VEND_CODE

PROD_CODE	PROD_DESCRIPT	PROD_PRICE	PROD_ON_HAND	VEND_CODE
001278-AB	Claw hammer	12.95	23	232
123-21UUY	Houselite chain saw, 16-in. bar	189.99	4	235
QER-34256	Sledge hammer, 16-lb. head	18.63	6	231
SRE-657UG	Rat-tail file	2.99	15	232
ZZX/3245Q	Steel tape, 12-ft. length	6.79	8	235

link

Table name: VENDOR
Primary key: VEND_CODE
Foreign key: none

VEND_CODE	VEND_CONTACT	VEND_AREACODE	VEND_PHONE
230	Shelly K. Smithson	608	555-1234
231	James Johnson	615	123-4536
232	Annelise Crystall	608	224-2134
233	Candice Wallace	904	342-6567
234	Arthur Jones	615	123-3324
235	Henry Ortozo	615	899-3425

Just as the primary key has a role in ensuring the integrity of the database, so does the foreign key. Foreign keys are used to ensure **referential integrity**, the condition in which every reference to an entity instance by another entity instance is valid. In other words, every foreign key entry must either be null or a valid value in the primary key of the related table. Note that the PRODUCT table has referential integrity because every entry in VEND_CODE in the PRODUCT table is either null or a valid value in VEND_CODE in the VENDOR table. Every vendor referred to by a row in the PRODUCT table is a valid vendor.

Finally, a **secondary key** is defined as a key that is used strictly for data retrieval purposes. Suppose that customer data is stored in a CUSTOMER table in which the customer number is the primary key. Do you think that most customers will remember their numbers? Data retrieval for a customer is easier when the customer's last name and phone number are used. In that case, the primary key is the customer number; the secondary key is the combination of the customer's last name and phone number. Keep in mind that a secondary key does not necessarily yield a unique outcome. For example, a customer's last name and home telephone number could easily yield several matches in which one family lives together and shares a phone line. A less efficient secondary key would be the combination of the last name and zip code; this could yield dozens of matches, which could then be combed for a specific match.

foreign key (FK)
An attribute or attributes in one table whose values must match the primary key in another table or whose values must be null. See *key*.

referential integrity
A condition by which a dependent table's foreign key must have either a null entry or a matching entry in the related table.

secondary key
A key used strictly for data retrieval purposes. For example, customers are not likely to know their customer number (primary key), but the combination of last name, first name, middle initial, and telephone number will probably match the appropriate table row. See also *key*.

A secondary key's effectiveness in narrowing down a search depends on how restrictive the key is. For instance, although the secondary key CUS_CITY is legitimate from a database point of view, the attribute values *New York* or *Sydney* are not likely to produce a usable return unless you want to examine millions of possible matches. (Of course, CUS_CITY is a better secondary key than CUS_COUNTRY.)

Table 3.3 summarizes the various relational database table keys.

TABLE 3.3

RELATIONAL DATABASE KEYS

KEY TYPE	DEFINITION
Superkey	An attribute or combination of attributes that uniquely identifies each row in a table
Candidate key	A minimal (irreducible) superkey; a superkey that does not contain a subset of attributes that is itself a superkey
Primary key	A candidate key selected to uniquely identify all other attribute values in any given row; cannot contain null entries
Foreign key	An attribute or combination of attributes in one table whose values must either match the primary key in another table or be null
Secondary key	An attribute or combination of attributes used strictly for data retrieval purposes

3-3 Integrity Rules

Relational database integrity rules are very important to good database design. Relational database management systems (RDBMSs) enforce integrity rules automatically, but it is much safer to make sure your application design conforms to the entity and referential integrity rules mentioned in this chapter. Those rules are summarized in Table 3.4.

TABLE 3.4

INTEGRITY RULES

ENTITY INTEGRITY	DESCRIPTION
Requirement	All primary key entries are unique, and no part of a primary key may be null.
Purpose	Each row will have a unique identity, and foreign key values can properly reference primary key values.
Example	No invoice can have a duplicate number, nor can it be null; in short, all invoices are uniquely identified by their invoice number.
REFERENTIAL INTEGRITY	**DESCRIPTION**
Requirement	A foreign key may have either a null entry, as long as it is not a part of its table's primary key, or an entry that matches the primary key value in a table to which it is related (every non-null foreign key value *must* reference an *existing* primary key value).
Purpose	It is possible for an attribute *not* to have a corresponding value, but it will be impossible to have an invalid entry; the enforcement of the referential integrity rule makes it impossible to delete a row in one table whose primary key has mandatory matching foreign key values in another table.
Example	A customer might not yet have an assigned sales representative (number), but it will be impossible to have an invalid sales representative (number).

The integrity rules summarized in Table 3.4 are illustrated in Figure 3.3.

FIGURE 3.3 AN ILLUSTRATION OF INTEGRITY RULES

Table name: CUSTOMER **Database name: Ch03_InsureCo**
Primary key: CUS_CODE
Foreign key: AGENT_CODE

CUS_CODE	CUS_LNAME	CUS_FNAME	CUS_INITIAL	CUS_RENEW_DATE	AGENT_CODE
10010	Ramas	Alfred	A	05-Apr-2018	502
10011	Dunne	Leona	K	16-Jun-2018	501
10012	Smith	Kathy	W	29-Jan-2019	502
10013	Olowski	Paul	F	14-Oct-2018	
10014	Orlando	Myron		28-Dec-2018	501
10015	O'Brian	Amy	B	22-Sep-2018	503
10016	Brown	James	G	25-Mar-2019	502
10017	Williams	George		17-Jul-2018	503
10018	Farriss	Anne	G	03-Dec-2018	501
10019	Smith	Olette	K	14-Mar-2019	503

Table name: AGENT (only five selected fields are shown)
Primary key: AGENT_CODE
Foreign key: none

AGENT_CODE	AGENT_AREACODE	AGENT_PHONE	AGENT_LNAME	AGENT_YTD_SLS
501	713	228-1249	Alby	132736.75
502	615	882-1244	Hahn	138967.35
503	615	123-5589	Okon	127093.45

Note the following features of Figure 3.3.

- *Entity integrity*. The CUSTOMER table's primary key is CUS_CODE. The CUSTOMER primary key column has no null entries, and all entries are unique. Similarly, the AGENT table's primary key is AGENT_CODE, and this primary key column is also free of null entries.
- *Referential integrity*. The CUSTOMER table contains a foreign key, AGENT_CODE, that links entries in the CUSTOMER table to the AGENT table. The CUS_CODE row identified by the (primary key) number 10013 contains a null entry in its AGENT_CODE foreign key because Paul F. Olowski does not yet have a sales representative assigned to him. The remaining AGENT_CODE entries in the CUSTOMER table all match the AGENT_CODE entries in the AGENT table.

To avoid nulls, some designers use special codes, known as **flags**, to indicate the absence of some value. Using Figure 3.3 as an example, the code –99 could be used as the AGENT_CODE entry in the fourth row of the CUSTOMER table to indicate that customer Paul Olowski does not yet have an agent assigned to him. If such a flag is used, the AGENT table must contain a dummy row with an AGENT_CODE value of –99. Thus, the AGENT table's first record might contain the values shown in Table 3.5.

flags
Special codes implemented by designers to trigger a required response, alert end users to specified conditions, or encode values. Flags may be used to prevent nulls by bringing attention to the absence of a value in a table.

TABLE 3.5
A DUMMY VARIABLE VALUE USED AS A FLAG

AGENT_CODE	AGENT_AREACODE	AGENT_PHONE	AGENT_LNAME	AGENT_YTD_SLS
–99	000	000–0000	None	$0.00

Chapter 4, Entity Relationship (ER) Modeling, discusses several ways to handle nulls.

Other integrity rules that can be enforced in the relational model are the NOT NULL and UNIQUE constraints. The NOT NULL constraint can be placed on a column to ensure that every row in the table has a value for that column. The UNIQUE constraint is a restriction placed on a column to ensure that no duplicate values exist for that column.

3-4 Relational Algebra

The data in relational tables is of limited value unless the data can be manipulated to generate useful information. This section describes the basic data manipulation capabilities of the relational model. **Relational algebra** defines the theoretical way of manipulating table contents using relational operators. In Chapter 7, Introduction to Structured Query Language (SQL), and Chapter 8, Advanced SQL, you will learn how SQL commands can be used to accomplish relational algebra operations.

Note

The degree of relational completeness can be defined by the extent to which relational algebra is supported. To be considered minimally relational, the DBMS must support the key relational operators SELECT, PROJECT, and JOIN.

3-4a Formal Definitions and Terminology

Recall that the relational model is actually based on mathematical principles, and manipulating the data in the database can be described in mathematical terms. The good news is that, as database professionals, we do not have to write mathematical formulas to work with our data. Data is manipulated by database developers and programmers using powerful languages like SQL that hide the underlying math. However, understanding the underlying principles can give you a good feeling for the types of operations that can be performed, and it can help you to understand how to write your queries more efficiently and effectively.

One advantage of using formal mathematical representations of operations is that mathematical statements are unambiguous. These statements are very specific, and they require that database designers be specific in the language used to explain them. As previously explained, it is common to use the terms *relation* and *table* interchangeably. However, since the mathematical terms need to be precise, we will use the more specific term relation when discussing the formal definitions of the various relational algebra operators.

Before considering the specific relational algebra operators, it is necessary to formalize our understanding of a table.

relational algebra
A set of mathematical principles that form the basis for manipulating relational table contents; the eight main functions are SELECT, PROJECT, JOIN, INTERSECT, UNION, DIFFERENCE, PRODUCT, and DIVIDE.

relvar
Short for relation variable, a variable that holds a relation. A relvar is a container (variable) for holding relation data, not the relation itself.

One important aspect of using the specific term *relation* is that it acknowledges the distinction between the relation and the relation variable, or *relvar*, for short. A relation is the data that we see in our tables. A **relvar** is a variable that holds a relation. For example, imagine you were writing a program and created a variable named *qty* for holding integer data. The variable *qty* is not an integer itself; it is a container for holding integers. Similarly, when you create a table, the table structure holds the table data. The structure is properly called a relvar, and the data in the structure would be a relation. The relvar is a container (variable) for holding relation data, not the relation itself. The data in the table is a relation.

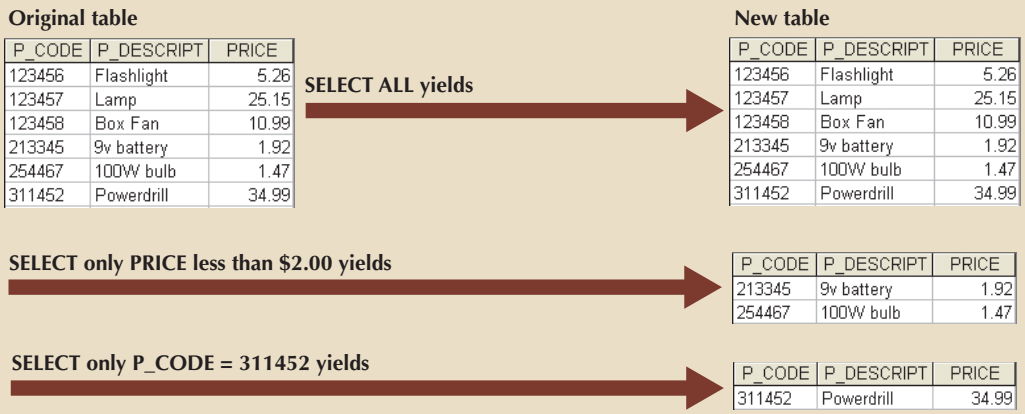
A relvar has two parts: the heading and the body. The relvar heading contains the names of the attributes, while the relvar body contains the relation. To conveniently maintain this distinction in formulas, an unspecified relation is often assigned a lowercase letter (e.g., "r"), while the relvar is assigned an uppercase letter (e.g., "R"). We could then say that *r* is a relation of type *R*, or *r(R)*.

3-4b Relational Set Operators

The relational operators have the property of **closure**; that is, the use of relational algebra operators on existing relations (tables) produces new relations. Numerous operators have been defined. Some operators are fundamental, while others are convenient but can be derived using the fundamental operators. In this section, the focus will be on the SELECT (or RESTRICT), PROJECT, UNION, INTERSECT, DIFFERENCE, PRODUCT, JOIN, and DIVIDE operators.

Select (Restrict) **SELECT**, also known as **RESTRICT**, is referred to as a unary operator because it only uses one table as input. It yields values for all rows found in the table that satisfy a given condition. SELECT can be used to list all of the rows, or it can yield only rows that match a specified criterion. In other words, SELECT yields a horizontal subset of a table. SELECT will not limit the attributes returned so all attributes of the table will be included in the result. The effect of a SELECT operation is shown in Figure 3.4.

FIGURE 3.4 SELECT



Note

Formally, SELECT is denoted by the lowercase Greek letter sigma (σ). Sigma is followed by the condition to be evaluated (called a predicate) as a subscript, and then the relation is listed in parentheses. For example, to SELECT all of the rows in the CUSTOMER table that have the value "10010" in the CUS_CODE attribute, you would write the following:

$$\sigma_{cus\_code = 10010} (customer)$$

Project **PROJECT** yields all values for selected attributes. It is also a unary operator, accepting only one table as input. PROJECT will return only the attributes requested, in the order in which they are requested. In other words, PROJECT yields a vertical subset of a table. PROJECT will not limit the rows returned, so all rows of the specified attributes will be included in the result. The effect of a PROJECT operation is shown in Figure 3.5.

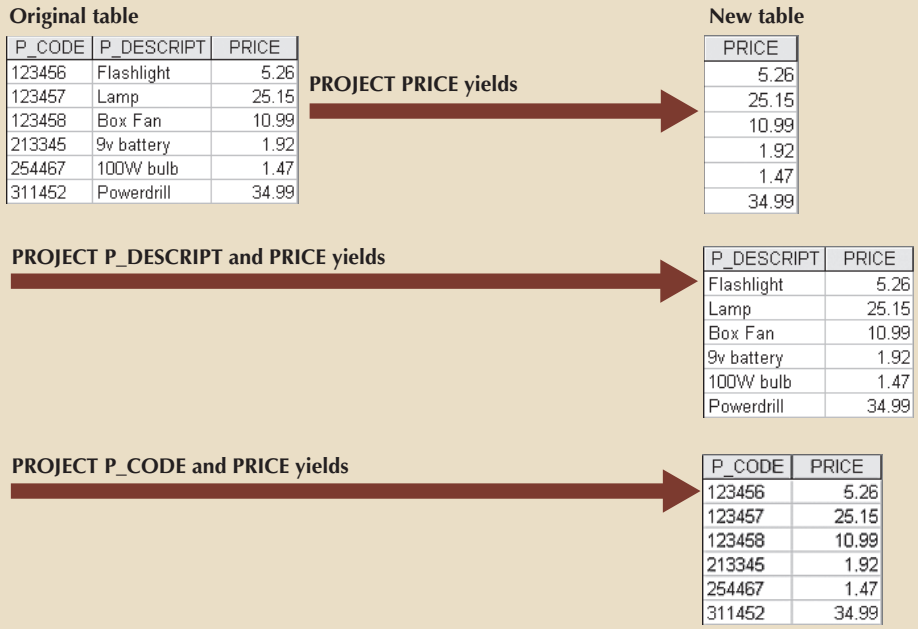
closure
A property of relational operators that permits the use of relational algebra operators on existing tables (relations) to produce new relations.

SELECT
In relational algebra, an operator used to select a subset of rows. Also known as *RESTRICT*.

RESTRICT
See *SELECT*.

PROJECT
In relational algebra, an operator used to select a subset of columns.

FIGURE 3.5 PROJECT



UNION
In relational algebra, an operator used to merge (append) two tables into a new table, dropping the duplicate rows. The tables must be *union-compatible*.

union-compatible
Two or more tables that have the same number of columns and the corresponding columns have compatible domains.

Note

Formally, PROJECT is denoted by the Greek letter pi (π). Some sources use the uppercase letter, and other sources use the lowercase letter. Codd used the lowercase π in his original article on the relational model, and that is what we use here. Pi is followed by the list of attributes to be returned as subscripts and then the relation listed in parentheses. For example, to PROJECT the CUS_FNAME and CUS_LNAME attributes in the CUSTOMER table, you would write the following:

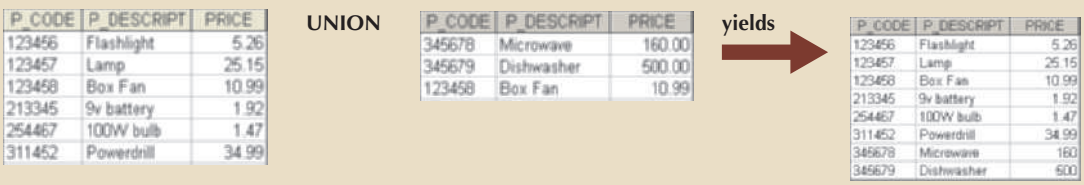
$$\pi_{cus\_fname,\ cus\_lname}\ (customer)$$

Since relational operators have the property of closure, that is, they accept relations as input and produce relations as output, it is possible to combine operators. For example, you can combine the two previous operators to find the first and last name of the customer with customer code 10010:

$$\pi_{cus\_fname,\ cus\_lname}\ (\sigma_{cus\_code\ =\ 10010}\ (customer))$$

Union **UNION** combines all rows from two tables, excluding duplicate rows. To be used in the UNION, the tables must have the same attribute characteristics; in other words, the columns and domains must be compatible. When two or more tables share the same number of columns, and when their corresponding columns share the same or compatible domains, they are said to be **union-compatible**. The effect of a UNION operation is shown in Figure 3.6.

FIGURE 3.6 UNION



Note

UNION is denoted by the symbol ∪. If the relations SUPPLIER and VENDOR are union-compatible, then a UNION between them would be denoted as follows:

supplier ∪ vendor

It is rather unusual to find two relations that are union-compatible in a database. Typically, PROJECT operators are applied to relations to produce results that are union-compatible. For example, assume the SUPPLIER and VENDOR tables are not union-compatible. If you wish to produce a listing of all vendor and supplier names, then you can PROJECT the names from each table and then perform a UNION with them.

$\pi_{supplier\_name}$ (supplier) ∪ π_{vendor_name} (vendor)

Intersect **INTERSECT** yields only the rows that appear in both tables. As with UNION, the tables must be union-compatible to yield valid results. For example, you cannot use INTERSECT if one of the attributes is numeric and one is character-based. For the rows to be considered the same in both tables and appear in the result of the INTERSECT, the entire rows must be exact duplicates. The effect of an INTERSECT operation is shown in Figure 3.7.

FIGURE 3.7 INTERSECT

STU_FNAME	STU_LNAME
George	Jones
Jane	Smith
Peter	Robinson
Franklin	Johnson
Martin	Lopez

INTERSECT

EMP_FNAME	EMP_LNAME
Franklin	Lopez
William	Turner
Franklin	Johnson
Susan	Rogers

yields

STU_FNAME	STU_LNAME
Franklin	Johnson

Note

INTERSECT is denoted by the symbol ∩. If the relations SUPPLIER and VENDOR are union-compatible, then an INTERSECT between them would be denoted as follows:

supplier ∩ vendor

Just as with the UNION operator, it is unusual to find two relations that are union-compatible in a database, so PROJECT operators are applied to relations to produce results that can be manipulated with an INTERSECT operator. For example, again assume the SUPPLIER and VENDOR tables are not union-compatible. If you wish to produce a listing of any vendor and supplier names that are the same in both tables, then you can PROJECT the names from each table and then perform an INTERSECT with them.

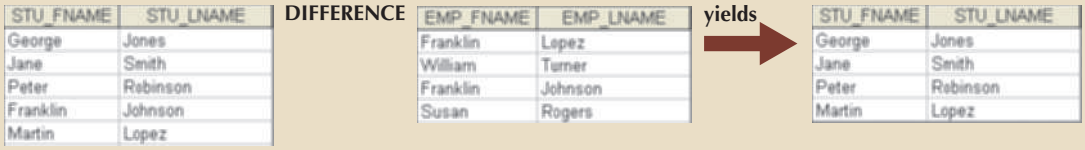
$\pi_{supplier\_name}$ (supplier) ∩ π_{vendor_name} (vendor)

INTERSECT
In relational algebra, an operator used to yield only the rows that are common to two union-compatible tables.

DIFFERENCE
In relational algebra, an operator used to yield all rows from one table that are not found in another union-compatible table.

Difference **DIFFERENCE** yields all rows in one table that are not found in the other table; that is, it subtracts one table from the other. As with UNION, the tables must be union-compatible to yield valid results. The effect of a DIFFERENCE operation is shown in Figure 3.8. However, note that subtracting the first table from the second table is not the same as subtracting the second table from the first table.

FIGURE 3.8 DIFFERENCE



STU_FNAME	STU_LNAME
George	Jones
Jane	Smith
Peter	Robinson
Franklin	Johnson
Martin	Lopez

DIFFERENCE

EMP_FNAME	EMP_LNAME
Franklin	Lopez
William	Turner
Franklin	Johnson
Susan	Rogers

yields

STU_FNAME	STU_LNAME
George	Jones
Jane	Smith
Peter	Robinson
Martin	Lopez

Note

DIFFERENCE is denoted by the minus symbol –. If the relations SUPPLIER and VENDOR are union-compatible, then a DIFFERENCE of SUPPLIER minus VENDOR would be written as follows:

supplier – vendor

Assuming the SUPPLIER and VENDOR tables are not union-compatible, producing a list of any supplier names that do not appear as vendor names, then you can use a DIFFERENCE operator.

$\pi_{supplier\_name}(supplier) - \pi_{vendor\_name}(vendor)$

Product **PRODUCT** yields all possible pairs of rows from two tables—also known as the Cartesian product. Therefore, if one table has 6 rows and the other table has 3 rows, the PRODUCT yields a list composed of 6 × 3 = 18 rows. The effect of a PRODUCT operation is shown in Figure 3.9.

FIGURE 3.9 PRODUCT

P_CODE	P_DESCRIPT	PRICE
123456	Flashlight	5.26
123457	Lamp	25.15
123458	Box Fan	10.99
213345	9v battery	1.92
254467	100W bulb	1.47
311452	Powerdrill	34.99

PRODUCT

STORE	AISLE	SHELF
23	W	5
24	K	9
25	Z	6

yields

P_CODE	P_DESCRIPT	PRICE	STORE	AISLE	SHELF
123456	Flashlight	5.26	23	W	5
123456	Flashlight	5.26	24	K	9
123456	Flashlight	5.26	25	Z	6
123457	Lamp	25.15	23	W	5
123457	Lamp	25.15	24	K	9
123457	Lamp	25.15	25	Z	6
123458	Box Fan	10.99	23	W	5
123458	Box Fan	10.99	24	K	9
123458	Box Fan	10.99	25	Z	6
213345	9v battery	1.92	23	W	5
213345	9v battery	1.92	24	K	9
213345	9v battery	1.92	25	Z	6
311452	Powerdrill	34.99	23	W	5
311452	Powerdrill	34.99	24	K	9
311452	Powerdrill	34.99	25	Z	6
254467	100W bulb	1.47	23	W	5
254467	100W bulb	1.47	24	K	9
254467	100W bulb	1.47	25	Z	6

Note

PRODUCT is denoted by the multiplication symbol ×. The PRODUCT of the CUSTOMER and AGENT relations would be written as follows:

customer × agent

A Cartesian product produces a set of sequences in which every member of one set is paired with every member of another set. In terms of relations, this means that every tuple in one relation is paired with every tuple in the second relation.

PRODUCT
In relational algebra, an operator used to yield all possible pairs of rows from two tables. Also known as the *Cartesian product*.

Join **JOIN** allows information to be intelligently combined from two or more tables. JOIN is the real power behind the relational database, allowing the use of independent tables linked by common attributes. The CUSTOMER and AGENT tables shown in Figure 3.10 will be used to illustrate several types of joins.

FIGURE 3.10 TWO TABLES THAT WILL BE USED IN JOIN ILLUSTRATIONS

Table name: CUSTOMER

CUS_CODE	CUS_LNAME	CUS_ZIP	AGENT_CODE
1132445	Walker	32145	231
1217782	Adares	32145	125
1312243	Rakowski	34129	167
1321242	Rodriguez	37134	125
1542311	Smithson	37134	421
1657399	Vanloo	32145	231

Table name: AGENT

AGENT_CODE	AGENT_PHONE
125	6152439887
167	6153426778
231	6152431124
333	9041234445

A **natural join** links tables by selecting only the rows with common values in their common attribute(s). A natural join is the result of a three-stage process:

1. First, a PRODUCT of the tables is created, yielding the results shown in Figure 3.11.

FIGURE 3.11 NATURAL JOIN, STEP 1: PRODUCT

CUS_CODE	CUS_LNAME	CUS_ZIP	CUSTOMER.AGENT_CODE	AGENT.AGENT_CODE	AGENT_PHONE
1132445	Walker	32145	231	125	6152439887
1132445	Walker	32145	231	167	6153426778
1132445	Walker	32145	231	231	6152431124
1132445	Walker	32145	231	333	9041234445
1217782	Adares	32145	125	125	6152439887
1217782	Adares	32145	125	167	6153426778
1217782	Adares	32145	125	231	6152431124
1217782	Adares	32145	125	333	9041234445
1312243	Rakowski	34129	167	125	6152439887
1312243	Rakowski	34129	167	167	6153426778
1312243	Rakowski	34129	167	231	6152431124
1312243	Rakowski	34129	167	333	9041234445
1321242	Rodriguez	37134	125	125	6152439887
1321242	Rodriguez	37134	125	167	6153426778
1321242	Rodriguez	37134	125	231	6152431124
1321242	Rodriguez	37134	125	333	9041234445
1542311	Smithson	37134	421	125	6152439887
1542311	Smithson	37134	421	167	6153426778
1542311	Smithson	37134	421	231	6152431124
1542311	Smithson	37134	421	333	9041234445
1657399	Vanloo	32145	231	125	6152439887
1657399	Vanloo	32145	231	167	6153426778
1657399	Vanloo	32145	231	231	6152431124
1657399	Vanloo	32145	231	333	9041234445

2. Second, a SELECT is performed on the output of Step 1 to yield only the rows for which the AGENT_CODE values are equal. The common columns are referred to as the **join columns**. Step 2 yields the results shown in Figure 3.12.

JOIN
In relational algebra, a type of operator used to yield rows from two tables based on criteria. There are many types of joins, such as natural join, theta join, equijoin, and outer join.

natural join
A relational operation that yields a new table composed of only the rows with common values in their common attribute(s).

join columns
Columns that are used in the criteria of join operations. The join columns generally share similar values.

FIGURE 3.12 NATURAL JOIN, STEP 2: SELECT

CUS_CODE	CUS_LNAME	CUS_ZIP	CUSTOMER.AGENT_CODE	AGENT.AGENT_CODE	AGENT_PHONE
1217782	Adares	32145	125	125	6152439887
1321242	Rodriguez	37134	125	125	6152439887
1312243	Rakowski	34129	167	167	6153426778
1132445	Walker	32145	231	231	6152431124
1657399	Vanloo	32145	231	231	6152431124

3. A PROJECT is performed on the results of Step 2 to yield a single copy of each attribute, thereby eliminating duplicate columns. Step 3 yields the output shown in Figure 3.13.

FIGURE 3.13 NATURAL JOIN, STEP 3: PROJECT

CUS_CODE	CUS_LNAME	CUS_ZIP	AGENT_CODE	AGENT_PHONE
1217782	Adares	32145	125	6152439887
1321242	Rodriguez	37134	125	6152439887
1312243	Rakowski	34129	167	6153426778
1132445	Walker	32145	231	6152431124
1657399	Vanloo	32145	231	6152431124

The final outcome of a natural join yields a table that does not include unmatched pairs and provides only the copies of the matches.

Note a few crucial features of the natural join operation:

- If no match is made between the table rows, the new table does not include the unmatched row. In that case, neither AGENT_CODE 421 nor the customer whose last name is Smithson is included. Smithson's AGENT_CODE 421 does not match any entry in the AGENT table.
- The column on which the join was made—that is, AGENT_CODE—occurs only once in the new table.
- If the same AGENT_CODE were to occur several times in the AGENT table, a customer would be listed for each match. For example, if the AGENT_CODE 167 occurred three times in the AGENT table, the customer named Rakowski would also occur three times in the resulting table because Rakowski is associated with AGENT_CODE 167. (Of course, a good AGENT table cannot yield such a result because it would contain unique primary key values.)

Note

Natural join is normally just referred to as JOIN in formal treatments. JOIN is denoted by the symbol ⋈. The JOIN of the CUSTOMER and AGENT relations would be written as follows:

customer ⋈ agent

Notice that the JOIN of two relations returns all of the attributes of both relations, except only one copy of the common attribute is returned. Formally, this is described as a UNION of the relvar headings. Therefore, the JOIN of the relations (c ⋈ a) includes the UNION of the relvars (C ∪ A). Also note that, as described above, JOIN is not a fundamental relational algebra operator. It can be derived from other operators as follows:

$$\pi_{\text{cus\_code, cus\_lname, cus\_fname, cus\_initial, cus\_renew\_date, agent\_code, agent\_areacode, agent\_phone, agent\_lname, agent\_ytd\_sls}} (\sigma_{\text{customer.agent\_code = agent.agent\_code}} (\text{customer} \times \text{agent}))$$

Another form of join, known as an **equijoin**, links tables on the basis of an equality condition that compares specified columns of each table. The outcome of the equijoin does not eliminate duplicate columns, and the condition or criterion used to join the tables must be explicitly defined. In fact, the result of an equijoin looks just like the outcome shown in Figure 3.12 for Step 2 of a natural join. The equijoin takes its name from the equality comparison operator (=) used in the condition. If any other comparison operator is used, the join is called a **theta join**.

Note

In formal terms, theta join is considered an extension of natural join. Theta join is denoted by adding a theta subscript after the JOIN symbol: $\bowtie_{\theta}$. Equijoin is then a special type of theta join.

Each of the preceding joins is often classified as an inner join. An **inner join** only returns matched records from the tables that are being joined. In an **outer join**, the matched pairs would be retained, and any unmatched values in the other table would be left null. It is an easy mistake to think that an outer join is the opposite of an inner join. However, it is more accurate to think of an outer join as an "inner join plus." The outer join still returns all of the matched records that the inner join returns, plus it returns the unmatched records from one of the tables. More specifically, if an outer join is produced for tables CUSTOMER and AGENT, two scenarios are possible:

- A **left outer join** yields all of the rows in the CUSTOMER table, including those that do not have a matching value in the AGENT table. An example of such a join is shown in Figure 3.14.

FIGURE 3.14 LEFT OUTER JOIN

CUS_CODE	CUS_LNAME	CUS_ZIP	CUSTOMER.AGENT_CODE	AGENT.AGENT_CODE	AGENT_PHONE
1217782	Adares	32145	125	125	6152439887
1321242	Rodriguez	37134	125	125	6152439887
1312243	Rakowski	34129	167	167	6153426778
1132445	Walker	32145	231	231	6152431124
1657399	Vanloo	32145	231	231	6152431124
1542311	Smithson	37134	421		

- A **right outer join** yields all of the rows in the AGENT table, including those that do not have matching values in the CUSTOMER table. An example of such a join is shown in Figure 3.15.

FIGURE 3.15 RIGHT OUTER JOIN

CUS_CODE	CUS_LNAME	CUS_ZIP	CUSTOMER.AGENT_CODE	AGENT.AGENT_CODE	AGENT_PHONE
1217782	Adares	32145	125	125	6152439887
1321242	Rodriguez	37134	125	125	6152439887
1312243	Rakowski	34129	167	167	6153426778
1132445	Walker	32145	231	231	6152431124
1657399	Vanloo	32145	231	231	6152431124
				333	9041234445

Outer joins are especially useful when you are trying to determine what values in related tables cause referential integrity problems. Such problems are created when foreign key values do not match the primary key values in the related table(s). In fact, if you are asked to convert large spreadsheets or other "nondatabase" data into relational database tables,

equijoin
A join operator that links tables based on an equality condition that compares specified columns of the tables.

theta join
A join operator that links tables using an inequality comparison operator (<, >, <=, >=) in the join condition.

inner join
A join operation in which only rows that meet a given criterion are selected. The criterion can be an equality condition (natural join or equijoin) or an inequality condition (theta join). The most commonly used type of join.

outer join
A join operation that produces a table in which all unmatched pairs are retained; unmatched values in the related table are left null.

left outer join
A join operation that yields all the rows in the left table, including those that have no matching values in the other table.

right outer join
A join operation that yields all of the rows in the right table, including the ones with no matching values in the other table.

you will discover that the outer joins save you vast amounts of time and uncounted headaches when you encounter referential integrity errors after the conversions.

You may wonder why the outer joins are labeled "left" and "right." The labels refer to the order in which the tables are listed in the SQL command. Chapter 7 explores such joins in more detail.

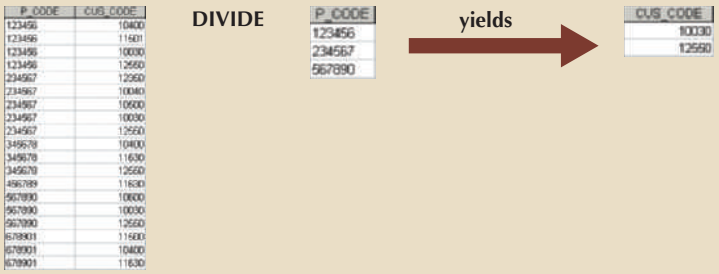
Note

Outer join is also an extension of JOIN. Outer joins are the application of JOIN, DIFFERENCE, UNION, and PRODUCT. A JOIN returns the matched tuples, DIFFERENCE finds the tuples in one table that have values in the common attribute that do not appear in the common attribute of the other relation, these unmatched tuples are combined with NULL values through a PRODUCT, and then a UNION combines these results into a single relation. Clearly, a defined outer join is a great simplification! Left and right outer joins are denoted by the symbols ⋈ and ⋈, respectively.

Divide The **DIVIDE** operator is used to answer questions about one set of data being associated with all values of data in another set of data. The DIVIDE operation uses one double-column table (Table 1) as the dividend and one single-column table (Table 2) as the divisor. For example, Figure 3.16 shows a list of customers and the products purchased in Table 1 on the left. Table 2 in the center contains a set of products that are of interest to the users. A DIVIDE operation can be used to determine which customers, if any, purchased every product shown in Table 2. In the figure, the dividend contains the P_CODE and CUS_CODE columns. The divisor contains the P_CODE column. The tables must have a common column—in this case, the P_CODE column. The output of the DIVIDE operation on the right is a single column that contains all values from the second column of the dividend (CUS_CODE) that are associated with every row in the divisor.

Using the example shown in Figure 3.16, note the following:

FIGURE 3.16 DIVIDE



DIVIDE
In relational algebra, an operator that answers queries about one set of data being associated with all values of data in another set of data.

- Table 1 is "divided" by Table 2 to produce Table 3. Tables 1 and 2 both contain the P_CODE column but do not share the CUS_CODE column.
- To be included in the resulting Table 3, a value in the unshared column (CUS_CODE) must be associated with every value in Table 2.
- The only customers associated with all of products 123456, 234567, and 567890 are customers 10030 and 12550.

Note

The DIVIDE operator is denoted by the division symbol ÷. Given two relations, R and S, the DIVISION of them would be written as r ÷ s.

3-5 The Data Dictionary and the System Catalog

The **data dictionary** provides a detailed description of all tables in the database created by the user and designer. Thus, the data dictionary contains at least all of the attribute names and characteristics for each table in the system. In short, the data dictionary contains metadata—data about data. Using the small database presented in Figure 3.3, you might picture its data dictionary as shown in Table 3.6.

Note

The data dictionary in Table 3.6 is an example of the *human* view of the entities, attributes, and relationships. The purpose of this data dictionary is to ensure that all members of database design and implementation teams use the same table and attribute names and characteristics. The DBMS's internally stored data dictionary contains additional information about relationship types, entity and referential integrity checks and enforcement, and index types and components. This additional information is generated during the database implementation stage.

The data dictionary is sometimes described as "the database designer's database" because it records the design decisions about tables and their structures.

Like the data dictionary, the system catalog contains metadata. The **system catalog** can be described as a detailed system data dictionary that describes all objects within the database, including data about table names, table's creator and creation date, number of columns in each table, data type corresponding to each column, index filenames, index creators, authorized users, and access privileges. Because the system catalog contains all required data dictionary information, the terms *system catalog* and *data dictionary* are often used interchangeably. In fact, current relational database software generally provides only a system catalog, from which the designer's data dictionary information may be derived. The system catalog is actually a system-created database whose tables store the user/designer-created database characteristics and contents. Therefore, the system catalog tables can be queried just like any user/designer-created table.

In effect, the system catalog automatically produces database documentation. As new tables are added to the database, that documentation also allows the RDBMS to check for and eliminate homonyms and synonyms. In general terms, **homonyms** are similar-sounding words with different meanings, such as *boar* and *bore*, or a word with different meanings, such as *fair* (which means "just" in some contexts and "festival" in others). In a database context, the word *homonym* indicates the use of the same name to label different attributes. For example, you might use C_NAME to label a customer name attribute in a CUSTOMER table and use C_NAME to label a consultant name attribute in a CONSULTANT table. To lessen confusion, you should avoid database homonyms; the data dictionary is very useful in this regard.

data dictionary
A DBMS component that stores metadata—data about data. Thus, the data dictionary contains the data definition as well as their characteristics and relationships. A data dictionary may also include data that are external to the DBMS. Also known as an *information resource dictionary*. See also *active data dictionary*, *metadata*, and *passive data dictionary*.

system catalog
A detailed system data dictionary that describes all objects in a database.

homonym
The use of the same name to label different attributes. Homonyms generally should be avoided. See also *synonym*.

TABLE 3.6

A SAMPLE DATA DICTIONARY

TABLE NAME	ATTRIBUTE NAME	CONTENTS	TYPE	FORMAT	RANGE	REQUIRED	PK OR FK	FK REFERENCED TABLE
CUSTOMER	CUS_CODE	Customer account code	CHAR(5)	99999	10000–99999	Y	PK	
	CUS_LNAME	Customer last name	VARCHAR(20)	Xxxxxxxx		Y		
	CUS_FNAME	Customer first name	VARCHAR(20)	Xxxxxxxx		Y		
	CUS_INITIAL	Customer initial	CHAR(1)	X				
	CUS_RENEW_DATE	Customer insurance renewal date	DATE	dd-mmm-yyyy				
	AGENT_CODE	Agent code	CHAR(3)	999			FK	AGENT
AGENT	AGENT_CODE	Agent code	CHAR(3)	999		Y	PK	
	AGENT_AREACODE	Agent area code	CHAR(3)	999		Y		
	AGENT_PHONE	Agent telephone number	CHAR(8)	999–9999		Y		
	AGENT_LNAME	Agent last name	VARCHAR(20)	Xxxxxxxx		Y		
	AGENT_YTD_SLS	Agent year-to-date sales	NUMBER(9,2)	9,999,999.99				

FK	= Foreign key
PK	= Primary key
CHAR	= Fixed character length data (1 – 255 characters)
VARCHAR	= Variable character length data (1 – 2,000 characters)
NUMBER	= Numeric data. NUMBER (9,2) is used to specify numbers with up to nine digits, including two digits to the right of the decimal place. Some RDBMS permit the use of a MONEY or CURRENCY data type.

Note

Telephone area codes are always composed of digits 0–9, but because area codes are not used arithmetically, they are most efficiently stored as character data. Also, the area codes are always composed of three digits. Therefore, the area code data type is defined as CHAR(3). On the other hand, names do not conform to a standard length. Therefore, the customer first names are defined as VARCHAR(20), indicating that up to 20 characters may be used to store the names. Character data are shown as left-aligned.

In a database context, a **synonym** is the opposite of a homonym and indicates the use of different names to describe the same attribute. For example, *car* and *auto* refer to the same object. Synonyms must be avoided whenever possible.

3-6 Relationships within the Relational Database

You already know that relationships are classified as one-to-one (1:1), one-to-many (1:M), and many-to-many (M:N or M:M). This section explores those relationships further to help you apply them properly when you start developing database designs. This section focuses on the following points:

- The 1:M relationship is the relational modeling ideal. Therefore, this relationship type should be the norm in any relational database design.
- The 1:1 relationship should be rare in any relational database design.
- M:N relationships cannot be implemented as such in the relational model. Later in this section, you will see how any M:N relationship can be changed into two 1:M relationships.

3-6a The 1:M Relationship

The 1:M relationship is the norm for relational databases. To see how such a relationship is modeled and implemented, consider the PAINTER and PAINTING example shown in Figure 3.17.

FIGURE 3.17 THE 1:M RELATIONSHIP BETWEEN PAINTER AND PAINTING

Compare the data model in Figure 3.17 with its implementation in Figure 3.18. As you examine the PAINTER and PAINTING table contents in Figure 3.18, note the following features:

- Each painting was created by one and only one painter, but each painter could have created many paintings. Note that painter 123 (Georgette P. Ross) has three works stored in the PAINTING table.
- There is only one row in the PAINTER table for any given row in the PAINTING table, but there may be many rows in the PAINTING table for any given row in the PAINTER table.

synonym
The use of different names to identify the same object, such as an entity, an attribute, or a relationship; synonyms should generally be avoided. See also *homonym*.

Note

The one-to-many (1:M) relationship is easily implemented in the relational model by putting the *primary key of the "1" side in the table of the "many" side as a foreign key*.

FIGURE 3.18 THE IMPLEMENTED 1:M RELATIONSHIP BETWEEN PAINTER AND PAINTING

Table name: PAINTER
Primary key: PAINTER_NUM
Foreign key: none

Database name: Ch03_Museum

PAINTER_NUM	PAINTER_LNAME	PAINTER_FNAME	PAINTER_INITIAL
123	Ross	Georgette	P
126	Itero	Julio	G

Table name: PAINTING
Primary key: PAINTING_NUM
Foreign key: PAINTER_NUM

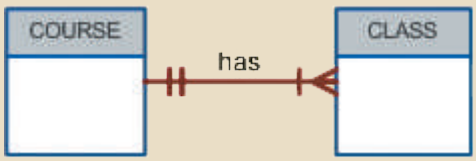
PAINTING_NUM	PAINTING_TITLE	PAINTER_NUM
1338	Dawn Thunder	123
1339	Vanilla Roses To Nowhere	123
1340	Tired Flounders	126
1341	Hasty Exit	123
1342	Plastic Paradise	126

The 1:M relationship is found in any database environment. Students in a typical college or university will discover that each COURSE can generate many CLASSes but that each CLASS refers to only one COURSE. For example, an Accounting II course might yield two classes: one offered on Monday, Wednesday, and Friday (MWF) from 10:00 a.m. to 10:50 a.m. and one offered on Thursday (Th) from 6:00 p.m. to 8:40 p.m. Therefore, the 1:M relationship between COURSE and CLASS might be described this way:

- Each COURSE can have many CLASSes, but each CLASS references only one COURSE.
- There will be only one row in the COURSE table for any given row in the CLASS table, but there can be many rows in the CLASS table for any given row in the COURSE table.

Figure 3.19 maps the entity relationship model (ERM) for the 1:M relationship between COURSE and CLASS.

FIGURE 3.19 THE 1:M RELATIONSHIP BETWEEN COURSE AND CLASS



The 1:M relationship between COURSE and CLASS is further illustrated in Figure 3.20.

Using Figure 3.20, take a minute to review some important terminology. Note that CLASS_CODE in the CLASS table uniquely identifies each row. Therefore, CLASS_CODE has been chosen to be the primary key. However, the combination CRS_CODE and CLASS_SECTION will also uniquely identify each row in the class table. In other words, the *composite key* composed of CRS_CODE and CLASS_SECTION is a candidate key. Any *candidate key* must have the not-null and unique constraints enforced. (You will see how this is done when you learn SQL in Chapter 8.)

FIGURE 3.20 THE IMPLEMENTED 1:M RELATIONSHIP BETWEEN COURSE AND CLASS

Table name: COURSE
Primary key: CRS_CODE
Foreign key: none

Database name: Ch03_TinyCollege

CRS_CODE	DEPT_CODE	CRS_DESCRIPTION	CRS_CREDIT
ACCT-211	ACCT	Accounting I	3
ACCT-212	ACCT	Accounting II	3
CIS-220	CIS	Intro. to Microcomputing	3
CIS-420	CIS	Database Design and Implementation	4
QM-261	CIS	Intro. to Statistics	3
QM-362	CIS	Statistical Applications	4

Table name: CLASS
Primary key: CLASS_CODE
Foreign key: CRS_CODE

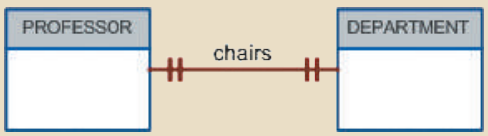
CLASS_CODE	CRS_CODE	CLASS_SECTION	CLASS_TIME	CLASS_ROOM	PROF_NUM
10012	ACCT-211	1	MWF 8:00-8:50 a.m.	BUS311	105
10013	ACCT-211	2	MWF 9:00-9:50 a.m.	BUS200	105
10014	ACCT-211	3	TTh 2:30-3:45 p.m.	BUS252	342
10015	ACCT-212	1	MWF 10:00-10:50 a.m.	BUS311	301
10016	ACCT-212	2	Th 6:00-8:40 p.m.	BUS252	301
10017	CIS-220	1	MWF 9:00-9:50 a.m.	KLR209	228
10018	CIS-220	2	MWF 9:00-9:50 a.m.	KLR211	114
10019	CIS-220	3	MWF 10:00-10:50 a.m.	KLR209	228
10020	CIS-420	1	W 6:00-8:40 p.m.	KLR209	162
10021	QM-261	1	MWF 8:00-8:50 a.m.	KLR200	114
10022	QM-261	2	TTh 1:00-2:15 p.m.	KLR200	114
10023	QM-362	1	MWF 11:00-11:50 a.m.	KLR200	162
10024	QM-362	2	TTh 2:30-3:45 p.m.	KLR200	162

For example, note in Figure 3.18 that the PAINTER table's primary key, PAINTER_NUM, is included in the PAINTING table as a foreign key. Similarly, in Figure 3.20, the COURSE table's primary key, CRS_CODE, is included in the CLASS table as a foreign key.

3-6b The 1:1 Relationship

As the 1:1 label implies, one entity in a 1:1 relationship can be related to only one other entity, and vice versa. For example, one department chair—a professor—can chair only one department, and one department can have only one department chair. The entities PROFESSOR and DEPARTMENT thus exhibit a 1:1 relationship. (You might argue that not all professors chair a department and professors cannot be *required* to chair a department. That is, the relationship between the two entities is optional. However, at this stage of the discussion, you should focus your attention on the basic 1:1 relationship. (Optional relationships will be addressed in Chapter 4.) The basic 1:1 relationship is modeled in Figure 3.21, and its implementation is shown in Figure 3.22.

FIGURE 3.21 THE 1:1 RELATIONSHIP BETWEEN PROFESSOR AND DEPARTMENT



As you examine the tables in Figure 3.22, note several important features:

- Each professor is a Tiny College employee. Therefore, the professor identification is through the EMP_NUM. (However, note that not all employees are professors—there's another optional relationship.)
- The 1:1 "PROFESSOR chairs DEPARTMENT" relationship is implemented by having the EMP_NUM foreign key in the DEPARTMENT table. Note that the 1:1 relationship is treated as a special case of the 1:M relationship in which the "many" side is restricted to a single occurrence. In this case, DEPARTMENT contains the EMP_NUM as a foreign key to indicate that it is the *department* that has a chair.

FIGURE 3.22 THE IMPLEMENTED 1:1 RELATIONSHIP BETWEEN PROFESSOR AND DEPARTMENT

Table name: PROFESSOR
Primary key: EMP_NUM
Foreign key: DEPT_CODE

Database name: Ch03_TinyCollege

EMP_NUM	DEPT_CODE	PROF_OFFICE	PROF_EXTENSION	PROF_HIGH_DEGREE
103	HIST	DRE 156	6783	Ph.D.
104	ENG	DRE 102	5561	MA
105	ACCT	KLR 229D	8665	Ph.D.
106	MKT/MGT	KLR 126	3899	Ph.D.
110	BIOL	AAK 160	3412	Ph.D.
114	ACCT	KLR 211	4436	Ph.D.
155	MATH	AAK 201	4440	Ph.D.
160	ENG	DRE 102	2248	Ph.D.
162	CIS	KLR 203E	2359	Ph.D.
191	MKT/MGT	KLR 409B	4016	DBA
195	PSYCH	AAK 297	3550	Ph.D.
209	CIS	KLR 333	3421	Ph.D.
228	CIS	KLR 300	3000	Ph.D.
297	MATH	AAK 194	1145	Ph.D.
299	ECON/FIN	KLR 284	2851	Ph.D.
301	ACCT	KLR 244	4683	Ph.D.
335	ENG	DRE 208	2000	Ph.D.
342	SOC	BBG 208	5514	Ph.D.
387	BIOL	AAK 230	8665	Ph.D.
401	HIST	DRE 156	6783	MA
425	ECON/FIN	KLR 284	2851	MBA
435	ART	BBG 185	2278	Ph.D.

The 1:M DEPARTMENT employs PROFESSOR relationship is implemented through the placement of the DEPT_CODE foreign key in the PROFESSOR table.

The 1:1 PROFESSOR chairs DEPARTMENT relationship is implemented through the placement of the EMP_NUM foreign key in the DEPARTMENT table.

Table name: DEPARTMENT
Primary key: DEPT_CODE
Foreign key: EMP_NUM

DEPT_CODE	DEPT_NAME	SCHOOL_CODE	EMP_NUM	DEPT_ADDRESS	DEPT_EXTENSION
ACCT	Accounting	BUS	114	KLR 211, Box 52	3119
ART	Fine Arts	A&SCI	435	BBG 185, Box 128	2278
BIOL	Biology	A&SCI	387	AAK 230, Box 415	4117
CIS	Computer Info. Systems	BUS	209	KLR 333, Box 56	3245
ECON/FIN	Economics/Finance	BUS	299	KLR 284, Box 63	3126
ENG	English	A&SCI	160	DRE 102, Box 223	1004
HIST	History	A&SCI	103	DRE 156, Box 284	1867
MATH	Mathematics	A&SCI	297	AAK 194, Box 422	4234
MKT/MGT	Marketing/Management	BUS	106	KLR 126, Box 55	3342
PSYCH	Psychology	A&SCI	195	AAK 297, Box 438	4110
SOC	Sociology	A&SCI	342	BBG 208, Box 132	2008

- Also note that the PROFESSOR table contains the DEPT_CODE foreign key to implement the 1:M "DEPARTMENT employs PROFESSOR" relationship. This is a good example of how two entities can participate in two (or even more) relationships simultaneously.

The preceding "PROFESSOR chairs DEPARTMENT" example illustrates a proper 1:1 relationship. *In fact, the use of a 1:1 relationship ensures that two entity sets are not placed in the same table when they should not be.* However, the existence of a 1:1 relationship sometimes means that the entity components were not defined properly. It could indicate that the two entities actually belong in the same table!

Although 1:1 relationships should be rare, certain conditions absolutely require their use. In Chapter 5, Advanced Data Modeling, you will explore a concept called a generalization hierarchy, which is a powerful tool for improving database designs under specific conditions to avoid a proliferation of nulls. One characteristic of generalization hierarchies is that they are implemented as 1:1 relationships.

Online Content

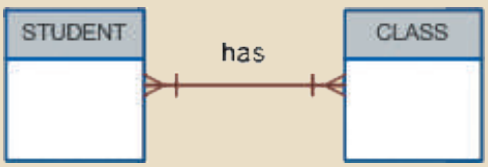
If you open the Ch03_TinyCollege database at *www.cengagebrain.com*, you will see that the STUDENT and CLASS entities still use PROF_NUM as their foreign key. PROF_NUM and EMP_NUM are labels for the same attribute, which is an example of the use of synonyms—that is, different names for the same attribute. These synonyms will be eliminated in future chapters as the Tiny College database continues to be improved.

3-6c The M:N Relationship

A many-to-many (M:N) relationship is not supported directly in the relational environment. However, M:N relationships can be implemented by creating a new entity in 1:M relationships with the original entities.

To explore the M:N relationship, consider a typical college environment. The ER model in Figure 3.23 shows this M:N relationship.

FIGURE 3.23 THE ERM'S M:N RELATIONSHIP BETWEEN STUDENT AND CLASS



Note the features of the ERM in Figure 3.23.

- Each CLASS can have many STUDENTs, and each STUDENT can take many CLASSes.
- There can be many rows in the CLASS table for any given row in the STUDENT table, and there can be many rows in the STUDENT table for any given row in the CLASS table.

To examine the M:N relationship more closely, imagine a small college with two students, each of whom takes three classes. Table 3.7 shows the enrollment data for the two students.

Online Content

If you look at the Ch03_AviaCo database at *www.cengagebrain.com*, you will see the implementation of the 1:1 PILOT to EMPLOYEE relationship. This relationship is based on a generalization hierarchy, which you will learn about in Chapter 5.

TABLE 3.7

SAMPLE STUDENT ENROLLMENT DATA

STUDENT'S LAST NAME	SELECTED CLASSES
Bowser	Accounting 1, ACCT-211, code 10014 Intro to Microcomputing, CIS-220, code 10018 Intro to Statistics, QM-261, code 10021
Smithson	Accounting 1, ACCT-211, code 10014 Intro to Microcomputing, CIS-220, code 10018 Intro to Statistics, QM-261, code 10021

Given such a data relationship and the sample data in Table 3.7, you could wrongly assume that you could implement this M:N relationship simply by adding a foreign key in the "many" side of the relationship that points to the primary key of the related table, as shown in Figure 3.24.

FIGURE 3.24 THE WRONG IMPLEMENTATION OF THE M:N RELATIONSHIP BETWEEN STUDENT AND CLASS

Table name: STUDENT
Primary key: STU_NUM
Foreign key: none

Database name: Ch03_CollegeTry

STU_NUM	STU_LNAME	CLASS_CODE
321452	Bowser	10014
321452	Bowser	10018
321452	Bowser	10021
324257	Smithson	10014
324257	Smithson	10018
324257	Smithson	10021

Table name: CLASS
Primary key: CLASS_CODE
Foreign key: STU_NUM

CLASS_CODE	STU_NUM	CRS_CODE	CLASS_SECTION	CLASS_TIME	CLASS_ROOM	PROF_NUM
10014	321452	ACCT-211	3	TTh 2:30-3:45 p.m.	BUS252	342
10014	324257	ACCT-211	3	TTh 2:30-3:45 p.m.	BUS252	342
10018	321452	CIS-220	2	MWF 9:00-9:50 a.m.	KLR211	114
10018	324257	CIS-220	2	MWF 9:00-9:50 a.m.	KLR211	114
10021	321452	QM-261	1	MWF 8:00-8:50 a.m.	KLR200	114
10021	324257	QM-261	1	MWF 8:00-8:50 a.m.	KLR200	114

However, the M:N relationship should *not* be implemented as shown in Figure 3.24 for two good reasons:

- The tables create many redundancies. For example, note that the STU_NUM values occur many times in the STUDENT table. In a real-world situation, additional student attributes such as address, classification, major, and home phone would also be contained in the STUDENT table, and each of those attribute values would be repeated in each of the records shown here. Similarly, the CLASS table contains much duplication: each student taking the class generates a CLASS record. The problem would be even worse if the CLASS table included such attributes as credit hours and course description. Those redundancies lead to the anomalies discussed in Chapter 1.
- Given the structure and contents of the two tables, the relational operations become very complex and are likely to lead to system efficiency errors and output errors.

composite entity
An entity designed to transform an M:N relationship into two 1:M relationships. The composite entity's primary key comprises at least the primary keys of the entities that it connects. Also known as a *bridge entity or associative entity*. See also *linking table*.

bridge entity
See *composite entity*.

associative entity
See *composite entity*.

Fortunately, the problems inherent in the M:N relationship can easily be avoided by creating a **composite entity** (also referred to as a **bridge entity** or an **associative entity**). Because such a table is used to link the tables that were originally related in an M:N relationship, the composite entity structure includes—as foreign keys—*at least* the primary keys of the tables that are to be linked. The database designer has two main options when defining a composite table's primary key: use the combination of those foreign keys or create a new primary key.

Remember that each entity in the ERM is represented by a table. Therefore, you can create the composite ENROLL table shown in Figure 3.25 to link the tables CLASS and STUDENT. In this example, the ENROLL table's primary key is the combination of its foreign keys CLASS_CODE and STU_NUM. However, the designer could have decided to create a single-attribute new primary key such as ENROLL_LINE, using a different

line value to identify each ENROLL table row uniquely. (Microsoft Access users might use the Autonumber data type to generate such line values automatically.)

FIGURE 3.25 CONVERTING THE M:N RELATIONSHIP INTO TWO 1:M RELATIONSHIPS

Table name: STUDENT
Primary key: STU_NUM
Foreign key: none

Database name: Ch03_CollegeTry2

STU_NUM	STU_LNAME
321452	Bowser
324257	Smithson

Table name: ENROLL
Primary key: CLASS_CODE + STU_NUM
Foreign key: CLASS_CODE, STU_NUM

CLASS_CODE	STU_NUM	ENROLL_GRADE
10014	321452	C
10014	324257	B
10018	321452	A
10018	324257	B
10021	321452	C
10021	324257	C

Table name: CLASS
Primary key: CLASS_CODE
Foreign key: CRS_CODE

CLASS_CODE	CRS_CODE	CLASS_SECTION	CLASS_TIME	CLASS_ROOM	PROF_NUM
10014	ACCT-211	3	TTh 2:30-3:45 p.m.	BUS252	342
10018	CIS-220	2	MWF 9:00-9:50 a.m.	KLR211	114
10021	QM-261	1	MWF 8:00-8:50 a.m.	KLR200	114

Because the ENROLL table in Figure 3.25 links two tables, STUDENT and CLASS, it is also called a **linking table**. In other words, a linking table is the implementation of a composite entity.

Note

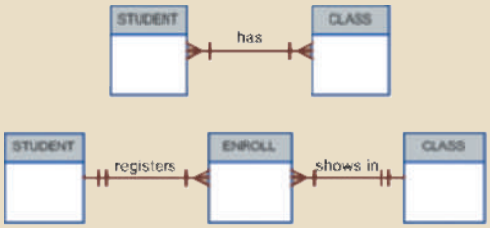
In addition to the linking attributes, the composite ENROLL table can also contain such relevant attributes as the grade earned in the course. In fact, a composite table can contain any number of attributes that the designer wants to track. Keep in mind that the composite entity, *although implemented as an actual table*, is *conceptually* a logical entity that was created as a means to an end: to eliminate the potential for multiple redundancies in the original M:N relationship.

The ENROLL table shown in Figure 3.25 yields the required M:N to 1:M conversion. Observe that the composite entity represented by the ENROLL table must contain at least the primary keys of the CLASS and STUDENT tables (CLASS_CODE and STU_NUM, respectively) for which it serves as a connector. Also note that the STUDENT and CLASS tables now contain only one row per entity. The ENROLL table contains multiple occurrences of the foreign key values, but those controlled redundancies are incapable of producing anomalies as long as referential integrity is enforced. Additional attributes may be assigned as needed. In this case, ENROLL_GRADE is selected to satisfy a reporting requirement. Also note that ENROLL_GRADE is fully dependent on the composite primary key. Naturally, the conversion is reflected in the ERM, too. The revised relationship is shown in Figure 3.26.

As you examine Figure 3.26, note that the composite entity named ENROLL represents the linking table between STUDENT and CLASS.

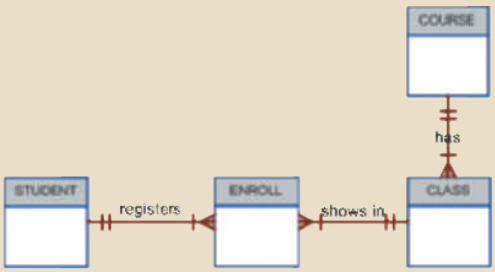
linking table
In the relational model, a table that implements an M:M relationship. See also *composite entity*.

FIGURE 3.26 CHANGING THE M:N RELATIONSHIPS TO TWO 1:M RELATIONSHIPS



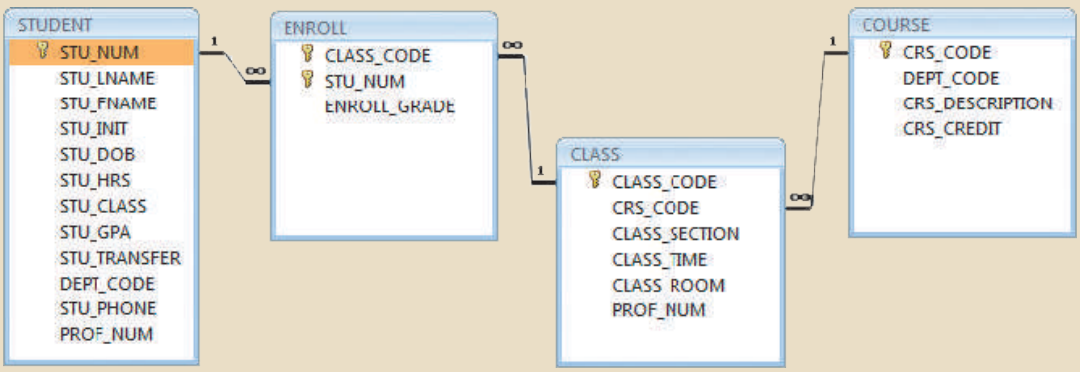
The 1:M relationship between COURSE and CLASS was first illustrated in Figure 3.19 and Figure 3.20. You can increase the amount of available information even as you control the database's redundancies. Thus, Figure 3.27 shows the expanded ERM, including the 1:M relationship between COURSE and CLASS shown in Figure 3.19. Note that the model can handle multiple sections of a CLASS while controlling redundancies by making sure that all of the COURSE data common to each CLASS are kept in the COURSE table.

FIGURE 3.27 THE EXPANDED ER MODEL



The relational diagram that corresponds to the ERM in Figure 3.27 is shown in Figure 3.28.

FIGURE 3.28 THE RELATIONAL DIAGRAM FOR THE CH03_TINYCOLLEGE DATABASE



The ERM will be examined in greater detail in Chapter 4 to show you how it is used to design more complex databases. The ERM will also be used as the basis for developing and implementing a realistic database design of a university computer lab in Appendixes B and C. These appendixes are available at *www.cengagebrain.com*.

3-7 Data Redundancy Revisited

In Chapter 1, you learned that data redundancy leads to data anomalies, which can destroy the effectiveness of the database. You also learned that the relational database makes it possible to control data redundancies by using common attributes that are shared by tables, called foreign keys.

The proper use of foreign keys is crucial to controlling data redundancy, although they do not totally eliminate the problem because the foreign key values can be repeated many times. However, the proper use of foreign keys *minimizes* data redundancies and the chances that destructive data anomalies will develop.

Note

The real test of redundancy is *not* how many copies of a given attribute are stored, *but whether the elimination of an attribute will eliminate information*. Therefore, if you delete an attribute and the original information can still be generated through relational algebra, the inclusion of that attribute would be redundant. Given that view of redundancy, proper foreign keys are clearly not redundant in spite of their multiple occurrences in a table. However, even when you use this less restrictive view of redundancy, keep in mind that *controlled* redundancies are often designed as part of the system to ensure transaction speed and/or information requirements.

You will learn in Chapter 4 that database designers must reconcile three often contradictory requirements: design elegance, processing speed, and information requirements. Also, you will learn in Chapter 13, Business Intelligence and Data Warehouses, that proper data warehousing design requires carefully defined and controlled data redundancies to function properly. Regardless of how you describe data redundancies, the potential for damage is limited by proper implementation and careful control.

As important as it is to control data redundancy, sometimes the level of data redundancy must actually be increased to make the database serve crucial information purposes. You will learn about such redundancies in Chapter 13. Also, data redundancies sometimes *seem* to exist to preserve the historical accuracy of the data. For example, consider a small invoicing system. The system includes the CUSTOMER, who may buy one or more PRODUCTs, thus generating an INVOICE. Because a customer may buy more than one product at a time, an invoice may contain several invoice LINEs, each providing details about the purchased product. The PRODUCT table should contain the product price to provide a consistent pricing input for each product that appears on the invoice. The tables that are part of such a system are shown in Figure 3.29. The system's relational diagram is shown in Figure 3.30.

As you examine the tables and relationships in the two figures, note that you can keep track of typical sales information. For example, by tracing the relationships among the four tables, you discover that customer 10014 (Myron Orlando) bought two items on March 8, 2018, that were written to invoice number 1001: one Houselite chain saw with a 16-inch bar and three rat-tail files. In other words, trace the CUS_CODE number 10014 in the CUSTOMER table to the matching CUS_CODE value in the INVOICE table. Next, trace the INV_NUMBER 1001 to the first two rows in the LINE table. Finally, match the two PROD_CODE values in LINE with the PROD_CODE values in PRODUCT. Application software will be used to write the correct bill by multiplying each invoice line item's LINE_UNITS by its LINE_PRICE, adding the results, and applying appropriate taxes. Later, other application software might use the same technique to write sales reports that track and compare sales by week, month, or year.

FIGURE 3.29 A SMALL INVOICING SYSTEM

Table name: CUSTOMER
Primary key: CUS_CODE
Foreign key: none

Database name: Ch03_SaleCo

CUS_CODE	CUS_LNAME	CUS_FNAME	CUS_INITIAL	CUS_AREACODE	CUS_PHONE
10010	Ramas	Alfred	A	615	844-2573
10011	Dunne	Leona	K	713	894-1238
10012	Smith	Kathy	W	615	894-2285
10013	Olowski	Paul	F	615	894-2180
10014	Orlando	Myron		615	222-1672
10015	O'Brian	Amy	B	713	442-3381
10016	Brown	James	G	615	297-1228
10017	Williams	George		615	290-2556
10018	Farriss	Anne	G	713	382-7185
10019	Smith	Olette	K	615	297-3809

Table name: INVOICE
Primary key: INV_NUMBER
Foreign key: CUS_CODE

INV_NUMBER	CUS_CODE	INV_DATE
1001	10014	08-Mar-18
1002	10011	08-Mar-18
1003	10012	08-Mar-18
1004	10011	09-Mar-18

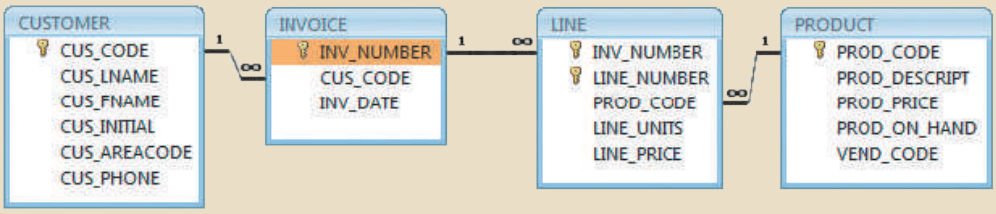
Table name: LINE
Primary key: INV_NUMBER + LINE_NUMBER
Foreign key: INV_NUMBER, PROD_CODE

INV_NUMBER	LINE_NUMBER	PROD_CODE	LINE_UNITS	LINE_PRICE
1001	1	123-21UUY	1	189.99
1001	2	SRE-657UO	3	2.99
1002	1	QER-34256	2	18.63
1003	1	ZZX/3245Q	1	6.79
1003	2	SRE-657UO	1	2.99
1003	3	001278-AB	1	12.95
1004	1	001278-AB	1	12.95
1004	2	SRE-657UO	2	2.99

Table name: PRODUCT
Primary key: PROD_CODE
Foreign key: none

PROD_CODE	PROD_DESCRIPT	PROD_PRICE	PROD_ON_HAND	VEND_CODE
001278-AB	Claw hammer	12.95	23	232
123-21UUY	Houselite chain saw, 16-in. bar	189.99	4	235
QER-34256	Sledge hammer, 16-lb. head	18.63	6	231
SRE-657UO	Rat-tail file	2.99	15	232
ZZX/3245Q	Steel tape, 12-ft. length	6.79	8	235

FIGURE 3.30 THE RELATIONAL DIAGRAM FOR THE INVOICING SYSTEM



As you examine the sales transactions in Figure 3.29, you might reasonably suppose that the product price billed to the customer is derived from the PRODUCT table because the product data is stored there. *But why does that same product price occur again in the LINE table? Is that not a data redundancy?* It certainly appears to be, but this time, the apparent redundancy is crucial to the system's success. Copying the product price from the PRODUCT table to the LINE table maintains the *historical accuracy of the transactions*. Suppose, for instance, that you fail to write the LINE_PRICE in the LINE table and that you use the PROD_PRICE from the PRODUCT table to calculate the sales revenue. Now suppose that the PRODUCT table's PROD_PRICE changes, as prices frequently do. This price change will be properly reflected in all subsequent sales revenue calculations. However, the calculations of past sales revenues will also reflect the new product price, which was not in effect when the transaction took

place! As a result, the revenue calculations for all past transactions will be incorrect, thus eliminating the possibility of making proper sales comparisons over time. On the other hand, if the price data is copied from the PRODUCT table and stored with the transaction in the LINE table, that price will always accurately reflect the transaction that took place *at that time*. You will discover that such planned "redundancies" are common in good database design.

Finally, you might wonder why the LINE_NUMBER attribute was used in the LINE table in Figure 3.29. Wouldn't the combination of INV_NUMBER and PROD_CODE be a sufficient composite primary key—and, therefore, isn't the LINE_NUMBER redundant? Yes, it is, but this redundancy is common practice on invoicing software that typically generates such line numbers automatically. In this case, the redundancy is not necessary, but given its automatic generation, the redundancy is not a source of anomalies. The inclusion of LINE_NUMBER also adds another benefit: the order of the retrieved invoicing data will always match the order in which the data was entered. If product codes are used as part of the primary key, indexing will arrange those product codes as soon as the invoice is completed and the data is stored. You can imagine the potential confusion when a customer calls and says, "The second item on my invoice has an incorrect price," and you are looking at an invoice whose lines show a different order from those on the customer's copy!

3-8 Indexes

Suppose you want to locate a book in a library. Does it make sense to look through every book until you find the one you want? Of course not; you use the library's catalog, which is indexed by title, topic, and author. The index (in either a manual or computer library catalog) points you to the book's location, making retrieval a quick and simple matter. An **index** is an orderly arrangement used to logically access rows in a table.

Or, suppose you want to find a topic in this book, such as *ER model*. Does it make sense to read through every page until you stumble across the topic? Of course not; it is much simpler to go to the book's index, look up the phrase *ER model*, and read the references that point you to the appropriate page(s). In each case, an index is used to locate a needed item quickly.

Indexes in the relational database environment work like the indexes described in the preceding paragraphs. From a conceptual point of view, an index is composed of an index key and a set of pointers. The **index key** is, in effect, the index's reference point. More formally, an index is an ordered arrangement of keys and pointers. Each key points to the location of the data identified by the key.

For example, suppose you want to look up all of the paintings created by a given painter in the Ch03_Museum database in Figure 3.18. Without an index, you must read each row in the PAINTING table and see if the PAINTER_NUM matches the requested painter. However, if you index the PAINTER table and use the index key PAINTER_NUM, you merely need to look up the appropriate PAINTER_NUM in the index and find the matching pointers. Conceptually speaking, the index would resemble the presentation in Figure 3.31.

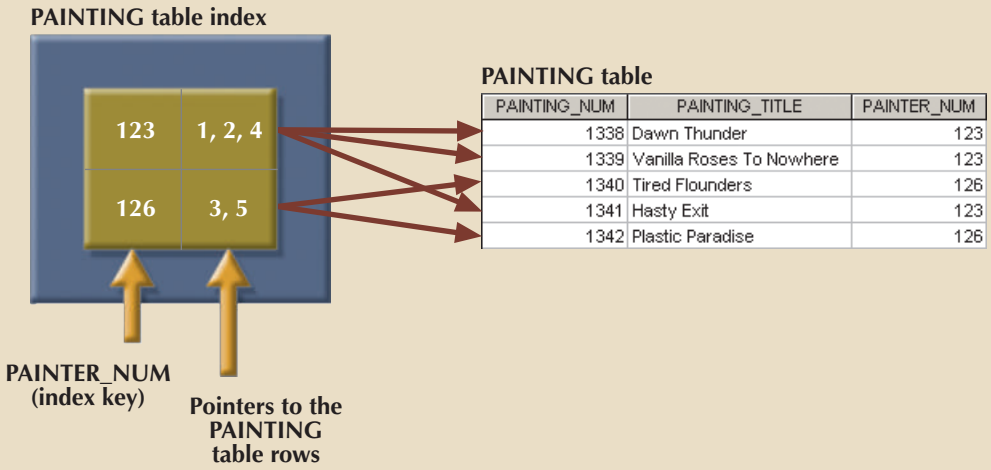
As you examine Figure 3.31, note that the first PAINTER_NUM index key value (123) is found in records 1, 2, and 4 of the PAINTING table. The second PAINTER_NUM index key value (126) is found in records 3 and 5 of the PAINTING table.

DBMSs use indexes for many different purposes. You just learned that an index can be used to retrieve data more efficiently, but indexes can also be used by a DBMS to retrieve data ordered by a specific attribute or attributes. For example, creating an index

index
An ordered array of index key values and row ID values (pointers). Indexes are generally used to speed up and facilitate data retrieval. Also known as an *index key*.

index key
See *index*.

FIGURE 3.31 COMPONENTS OF AN INDEX



on a customer's last name will allow you to retrieve the customer data alphabetically by the customer's last name. Also, an index key can be composed of one or more attributes. For example, in Figure 3.29, you can create an index on VEND_CODE and PROD_CODE to retrieve all rows in the PRODUCT table ordered by vendor and, within vendor, ordered by product.

Indexes play an important role in DBMSs for the implementation of primary keys. When you define a table's primary key, the DBMS automatically creates a unique index on the primary key column(s) you declared. For example, in Figure 3.29, when you declare CUS_CODE to be the primary key of the CUSTOMER table, the DBMS automatically creates a unique index on that attribute. In a **unique index**, as its name implies, the index key can have only one pointer value (row) associated with it. (The index in Figure 3.31 is not a unique index because the PAINTER_NUM has multiple pointer values associated with it. For example, painter number 123 points to three rows—1, 2, and 4—in the PAINTING table.)

A table can have many indexes, but each index is associated with only one table. The index key can have multiple attributes (a composite index). Creating an index is easy. You will learn in Chapter 8 that a simple SQL command produces any required index.

3-9 Codd's Relational Database Rules

In 1985, Dr. E. F. Codd published a list of 12 rules to define a relational database system.[1] He published the list out of concern that many vendors were marketing products as "relational" even though those products did not meet minimum relational standards. Dr. Codd's list, shown in Table 3.8, is a frame of reference for what a truly relational database should be. Bear in mind that even the dominant database vendors do not fully support all 12 rules.

unique index
An index in which the index key can have only one associated pointer value (row).

[1] Codd, E., "Is Your DBMS Really Relational?" and "Does Your DBMS Run by the Rules?" *Computerworld*, October 14 and 21, 1985.

TABLE 13.8

DR. CODD'S 12 RELATIONAL DATABASE RULES

RULE	RULE NAME	DESCRIPTION
1	Information	All information in a relational database must be logically represented as column values in rows within tables.
2	Guaranteed access	Every value in a table is guaranteed to be accessible through a combination of table name, primary key value, and column name.
3	Systematic treatment of nulls	Nulls must be represented and treated in a systematic way, independent of data type.
4	Dynamic online catalog based on the relational model	The metadata must be stored and managed as ordinary data—that is, in tables within the database; such data must be available to authorized users using the standard database relational language.
5	Comprehensive data sublanguage	The relational database may support many languages; however, it must support one well-defined, declarative language as well as data definition, view definition, data manipulation (interactive and by program), integrity constraints, authorization, and transaction management (begin, commit, and rollback).
6	View updating	Any view that is theoretically updatable must be updatable through the system.
7	High-level insert, update, and delete	The database must support set-level inserts, updates, and deletes.
8	Physical data independence	Application programs and ad hoc facilities are logically unaffected when physical access methods or storage structures are changed.
9	Logical data independence	Application programs and ad hoc facilities are logically unaffected when changes are made to the table structures that preserve the original table values (changing order of columns or inserting columns).
10	Integrity independence	All relational integrity constraints must be definable in the relational language and stored in the system catalog, not at the application level.
11	Distribution independence	The end users and application programs are unaware of and unaffected by the data location (distributed vs. local databases).
12	Nonsubversion	If the system supports low-level access to the data, users must not be allowed to bypass the integrity rules of the database.
13	Rule zero	All preceding rules are based on the notion that to be considered relational, a database must use its relational facilities exclusively for management.

Summary

- Tables are the basic building blocks of a relational database. A grouping of related entities, known as an entity set, is stored in a table. Conceptually speaking, the relational table is composed of intersecting rows (tuples) and columns. Each row represents a single entity, and each column represents the characteristics (attributes) of the entities.
- Keys are central to the use of relational tables. Keys define functional dependencies; that is, other attributes are dependent on the key and can therefore be found if the key value is known. A key can be classified as a superkey, a candidate key, a primary key, a secondary key, or a foreign key.
- Each table row must have a primary key. The primary key is an attribute or combination of attributes that uniquely identifies all remaining attributes found in any given row. Because a primary key must be unique, no null values are allowed if entity integrity is to be maintained.
- Although tables are independent, they can be linked by common attributes. Thus, the primary key of one table can appear as the foreign key in another table to which it is linked. Referential integrity dictates that the foreign key must contain values that match the primary key in the related table or must contain nulls.
- The relational model supports several relational algebra functions, including SELECT, PROJECT, JOIN, INTERSECT, UNION, DIFFERENCE, PRODUCT, and DIVIDE. Understanding the basic mathematical forms of these functions gives a broader understanding of the data manipulation options.
- A relational database performs much of the data manipulation work behind the scenes. For example, when you create a database, the RDBMS automatically produces a structure to house a data dictionary for your database. Each time you create a new table within the database, the RDBMS updates the data dictionary, thereby providing the database documentation.
- Once you know the basics of relational databases, you can concentrate on design. Good design begins by identifying appropriate entities and their attributes and then the relationships among the entities. Those relationships (1:1, 1:M, and M:N) can be represented using ERDs. The use of ERDs allows you to create and evaluate simple logical design. The 1:M relationship is most easily incorporated in a good design; just make sure that the primary key of the "1" is included in the table of the "many."

Key Terms

associative entity	full functional dependence	PRODUCT
attribute domain	functional dependence	PROJECT
bridge entity	homonym	referential integrity
candidate key	index	relational algebra
closure	index key	relvar
composite entity	inner join	RESTRICT
composite key	INTERSECT	right outer join
data dictionary	JOIN	secondary key
dependent	join column	SELECT
determinant	key	set theory
determination	key attribute	superkey
DIFFERENCE	left outer join	synonym
DIVIDE	linking table	system catalog
domain	natural join	theta join
entity integrity	null	tuple
equijoin	outer join	UNION
flags	predicate logic	union-compatible
foreign key (FK)	primary key (PK)	unique index

Review Questions

1. What is the difference between a database and a table?
2. What does it mean to say that a database displays both entity integrity and referential integrity?
3. Why are entity integrity and referential integrity important in a database?
4. What are the requirements that two relations must satisfy to be considered union-compatible?
5. Which relational algebra operators can be applied to a pair of tables that are not union-compatible?
6. Explain why the data dictionary is sometimes called "the database designer's database."
7. A database user manually notes that "The file contains two hundred records, each record containing nine fields." Use appropriate relational database terminology to "translate" that statement.

Use Figure Q3.8 to answer Questions 8–12.

8. Using the STUDENT and PROFESSOR tables, illustrate the difference between a natural join, an equijoin, and an outer join.
9. Create the table that would result from π_{stu_code} (student).

Online Content

All of the databases used in the questions and problems are available at *www.cengagebrain.com*. The database names match the database names shown in the figures.

10. Create the table that would result from $\pi_{stu\_code,\ dept\_code}$ (student ⋈ professor).
11. Create the basic ERD for the database shown in Figure Q3.8.
12. Create the relational diagram for the database shown in Figure Q3.8.

FIGURE Q3.8 THE CH03_COLLEGEQUE DATABASE TABLES

Database name: Ch03_CollegeQue

Table name: STUDENT

STU_CODE	PROF_CODE
100278	
128569	2
512272	4
531235	2
531268	
553427	1

Table name: PROFESSOR

PROF_CODE	DEPT_CODE
1	2
2	6
3	6
4	4

Use Figure Q3.13 to answer Questions 13–17.

FIGURE Q3.13 THE CH03_VENDINGCO DATABASE TABLES

Database name: Ch03_VendingCo

Table name: BOOTH

BOOTH_PRODUCT	BOOTH_PRICE
Chips	1.5
Cola	1.25
Energy Drink	2

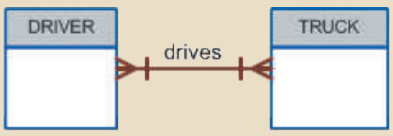
Table name: MACHINE

MACHINE_PRODUCT	MACHINE_PRICE
Chips	1.25
Chocolate Bar	1
Energy Drink	2

13. Write the relational algebra formula to apply a UNION relational operator to the tables shown in Figure Q3.13.
14. Create the table that results from applying a UNION relational operator to the tables shown in Figure Q3.13.
15. Write the relational algebra formula to apply an INTERSECT relational operator to the tables shown in Figure Q3.13.
16. Create the table that results from applying an INTERSECT relational operator to the tables shown in Figure Q3.13.
17. Using the tables in Figure Q3.13, create the table that results from MACHINE DIFFERENCE BOOTH.

Use Figure Q3.18 to answer Question 18.

FIGURE Q3.18 THE CROW'S FOOT ERD FOR DRIVER AND TRUCK



During some time interval, a DRIVER can drive many TRUCKs and any TRUCK can be driven by many DRIVERs

18. Suppose you have the ERD shown in Figure Q3.18. How would you convert this model into an ERM that displays only 1:M relationships? (Make sure you create the revised ERD.)
19. What are homonyms and synonyms, and why should they be avoided in database design?
20. How would you implement a l:M relationship in a database composed of two tables? Give an example.

Use Figure Q3.21 to answer Question 21.

FIGURE Q3.21 THE CH03_NOCOMP DATABASE EMPLOYEE TABLE

Table name: EMPLOYEE **Database name: Ch03_NoComp**

EMP_NUM	EMP_LNAME	EMP_INITIAL	EMP_FNAME	DEPT_CODE	JOB_CODE
11234	Friedman	K	Robert	MKTG	12
11238	Olanski	D	Delbert	MKTG	12
11241	Fontein		Juliette	INFS	5
11242	Cruazona	J	Maria	ENG	9
11245	Smithson	B	Bernard	INFS	6
11248	Washington	G	Oleta	ENGR	8
11256	McBride		Randall	ENGR	8
11257	Kachinn	D	Melanie	MKTG	14
11258	Smith	W	William	MKTG	14
11260	Ratula	A	Katrina	INFS	5

21. Identify and describe the components of the table shown in Figure Q3.21, using correct terminology. Use your knowledge of naming conventions to identify the table's probable foreign key(s).

Use the database shown in Figure Q3.22 to answer Questions 22–27.

FIGURE Q3.22 THE CH03_THEATER DATABASE TABLES

Database name: Ch03_Theater

Table name: DIRECTOR

DIR_NUM	DIR_LNAME	DIR_DOB
100	Broadway	12-Jan-65
101	Hollywoody	18-Nov-53
102	Goofy	21-Jun-62

Table name: PLAY

PLAY_CODE	PLAY_NAME	DIR_NUM
1001	Cat On a Cold, Bare Roof	102
1002	Hold the Mayo, Pass the Bread	101
1003	I Never Promised You Coffee	102
1004	Silly Putty Goes To Washington	100
1005	See No Sound, Hear No Sight	101
1006	Starstruck in Biloxi	102
1007	Stranger In Parrot Ice	101

22. Identify the primary keys.
23. Identify the foreign keys.
24. Create the ERM.
25. Create the relational diagram to show the relationship between DIRECTOR and PLAY.
26. Suppose you wanted quick lookup capability to get a listing of all plays directed by a given director. Which table would be the basis for the INDEX table, and what would be the index key?
27. What would be the conceptual view of the INDEX table described in Question 26? Depict the contents of the conceptual INDEX table.

Problems

FIGURE P3.1 THE CH03_STORECO DATABASE TABLES

Table name: EMPLOYEE **Database name: Ch03_StoreCo**

EMP_CODE	EMP_TITLE	EMP_LNAME	EMP_FNAME	EMP_INITIAL	EMP_DOB	STORE_CODE
1	Mr.	Williamson	John	W	21-May-84	3
2	Ms.	Ratula	Nancy		09-Feb-89	2
3	Ms.	Greenboro	Lottie	R	02-Oct-81	4
4	Mrs.	Rumpersfro	Jennie	S	01-Jun-91	5
5	Mr.	Smith	Robert	L	23-Nov-79	3
6	Mr.	Renselaer	Cary	A	25-Dec-85	1
7	Mr.	Ogallo	Roberto	S	31-Jul-82	3
8	Ms.	Johnsson	Elizabeth	I	10-Sep-88	1
9	Mr.	Eindsmar	Jack	W	19-Apr-75	2
10	Mrs.	Jones	Rose	R	06-Mar-86	4
11	Mr.	Broderick	Tom		21-Oct-92	3
12	Mr.	Washington	Alan	Y	08-Sep-94	2
13	Mr.	Smith	Peter	N	25-Aug-84	3
14	Ms.	Smith	Sherry	H	25-May-86	4
15	Mr.	Olenko	Howard	U	24-May-84	5
16	Mr.	Archialo	Barry	V	03-Sep-80	5
17	Ms.	Grimaldo	Jeanine	K	12-Nov-90	4
18	Mr.	Rosenberg	Andrew	D	24-Jan-91	4
19	Mr.	Rosten	Peter	F	03-Oct-88	4
20	Mr.	Mckee	Robert	S	06-Mar-90	1
21	Ms.	Baumann	Jennifer	A	11-Dec-94	3

Table name: STORE

STORE_CODE	STORE_NAME	STORE_YTD_SALES	REGION_CODE	EMP_CODE
1	Access Junction	1003455.76	2	8
2	Database Corner	1421987.39	2	12
3	Tuple Charge	986783.22	1	7
4	Attribute Alley	944568.56	2	3
5	Primary Key Point	2930098.45	1	15

Table name: REGION

REGION_CODE	REGION_DESCRIPT
1	East
2	West

Use the database shown in Figure P3.1 to answer Problems 1–9.

1. For each table, identify the primary key and the foreign key(s). If a table does not have a foreign key, write *None*.
2. Do the tables exhibit entity integrity? Answer yes or no, and then explain your answer.
3. Do the tables exhibit referential integrity? Answer yes or no, and then explain your answer. Write *NA* (Not Applicable) if the table does not have a foreign key.
4. Describe the type(s) of relationship(s) between STORE and REGION.
5. Create the ERD to show the relationship between STORE and REGION.
6. Create the relational diagram to show the relationship between STORE and REGION.
7. Describe the type(s) of relationship(s) between EMPLOYEE and STORE. (*Hint*: Each store employs many employees, one of whom manages the store.)

8. Create the ERD to show the relationships among EMPLOYEE, STORE, and REGION.
9. Create the relational diagram to show the relationships among EMPLOYEE, STORE, and REGION.

FIGURE P3.10 THE CH03_BENECO DATABASE TABLES

Database name: Ch03_BeneCo

Table name: EMPLOYEE

EMP_CODE	EMP_LNAME	JOB_CODE
14	Rudell	2
15	McDade	1
16	Ruellardo	1
17	Smith	3
20	Smith	2

Table name: BENEFIT

EMP_CODE	PLAN_CODE
15	2
15	3
16	1
17	1
17	3
17	4
20	3

Table name: JOB

JOB_CODE	JOB_DESCRIPTION
1	Clerical
2	Technical
3	Managerial

Table name: PLAN

PLAN_CODE	PLAN_DESCRIPTION
1	Term life
2	Stock purchase
3	Long-term disability
4	Dental

Use the database shown in Figure P3.10 to work Problems 10–16. Note that the database is composed of four tables that reflect these relationships:

- An EMPLOYEE has only one JOB_CODE, but a JOB_CODE can be held by many EMPLOYEEs.
- An EMPLOYEE can participate in many PLANs, and any PLAN can be assigned to many EMPLOYEEs.

Note also that the M:N relationship has been broken down into two 1:M relationships for which the BENEFIT table serves as the composite or bridge entity.

10. For each table in the database, identify the primary key and the foreign key(s). If a table does not have a foreign key, write *None*.
11. Create the ERD to show the relationship between EMPLOYEE and JOB.
12. Create the relational diagram to show the relationship between EMPLOYEE and JOB.
13. Do the tables exhibit entity integrity? Answer yes or no, and then explain your answer.
14. Do the tables exhibit referential integrity? Answer yes or no, and then explain your answer. Write *NA* (Not Applicable) if the table does not have a foreign key.
15. Create the ERD to show the relationships among EMPLOYEE, BENEFIT, JOB, and PLAN.
16. Create the relational diagram to show the relationships among EMPLOYEE, BENEFIT, JOB, and PLAN.

FIGURE P3.17 THE CH03_TRANSCO DATABASE TABLES

Table name: TRUCK **Database name: Ch03_TransCo**
Primary key: TRUCK_NUM
Foreign key: BASE_CODE, TYPE_CODE

TRUCK_NUM	BASE_CODE	TYPE_CODE	TRUCK_MILES	TRUCK_SERIAL_NUM
1001	501	1	32123.5	AA-322-12212-W11
1002	502	1	76984.3	AC-342-22134-Q23
1003	501	2	12346.6	AC-445-78656-Z99
1004		1	2894.3	WQ-112-23144-T34
1005	503	2	45673.1	FR-998-32245-W12
1006	501	2	193245.7	AD-456-00845-R45
1007	502	3	32012.3	AA-341-96573-Z84
1008	502	3	44213.6	DR-559-22189-D33
1009	503	2	10932.9	DE-887-98456-E94

Table name: BASE
Primary key: BASE_CODE
Foreign key: none

BASE_CODE	BASE_CITY	BASE_STATE	BASE_AREA_CODE	BASE_PHONE	BASE_MANAGER
501	Murfreesboro	TN	615	123-4567	Andrea D. Gallager
502	Lexington	KY	568	234-5678	George H. Delarosa
503	Cape Girardeau	MO	456	345-6789	Maria J. Talindo
504	Dalton	GA	901	456-7890	Peter F. McAvee

Table name: TYPE
Primary key: TYPE_CODE
Foreign key: none

TYPE_CODE	TYPE_DESCRIPTION
1	Single box, double-axle
2	Single box, single-axle
3	Tandem trailer, single-axle

Use the database shown in Figure P3.17 to answer Problems 17–23.

17. For each table, identify the primary key and the foreign key(s). If a table does not have a foreign key, write *None*.
18. Do the tables exhibit entity integrity? Answer yes or no, and then explain your answer.
19. Do the tables exhibit referential integrity? Answer yes or no, and then explain your answer. Write *NA* (Not Applicable) if the table does not have a foreign key.
20. Identify the TRUCK table's candidate key(s).
21. For each table, identify a superkey and a secondary key.
22. Create the ERD for this database.
23. Create the relational diagram for this database.

FIGURE P3.24 THE CH03_AVIACO DATABASE TABLES

Table name: CHARTER **Database name: Ch03_AviaCo**

CHAR_TRIP	CHAR_DATE	CHAR_PILOT	CHAR_COPILOT	AC_NUMBER	CHAR_DESTINATION	CHAR_DISTANCE	CHAR_HOURS_FLOWN	CHAR_HOURS_WAIT	CHAR_FUEL_GALLONS	CHAR_OIL_QTS	CUS_CODE
10001	05-Feb-18	104		2289L	ATL	936.0	5.1	2.2	354.1	1	10011
10002	05-Feb-18	101		2778V	BNA	328.0	1.8	0.0	72.6	0	10016
10003	05-Feb-18	105	109	4278Y	GNV	1574.0	7.8	0.0	339.8	2	10014
10004	06-Feb-18	106		1484P	STL	472.0	2.9	4.9	97.2	1	10019
10005	06-Feb-18	101		2289L	ATL	1023.0	5.7	3.5	397.7	2	10011
10006	06-Feb-18	109		4278Y	STL	472.0	2.5	5.2	117.1	0	10017
10007	06-Feb-18	104	105	2778V	GNV	1574.0	7.9	0.0	348.4	2	10012
10008	07-Feb-18	106		1484P	TYS	644.0	4.1	0.0	140.6	1	10014
10009	07-Feb-18	105		2289L	GNV	1574.0	6.6	23.4	459.9	0	10017
10010	07-Feb-18	109		4278Y	ATL	998.0	6.2	3.2	279.7	0	10016
10011	07-Feb-18	101	104	1484P	BNA	352.0	1.9	5.3	99.4	1	10012
10012	08-Feb-18	101		2778V	MOB	864.0	4.8	4.2	215.1	0	10010
10013	08-Feb-18	105		4278Y	TYS	644.0	3.9	4.5	174.3	1	10011
10014	09-Feb-18	106		4278Y	ATL	936.0	6.1	2.1	382.6	0	10017
10015	09-Feb-18	104	101	2289L	GNV	1645.0	6.7	0.0	459.5	2	10016
10016	09-Feb-18	109	105	2778V	MGV	312.0	1.5	0.0	67.2	0	10011
10017	10-Feb-18	101		1484P	STL	508.0	3.1	0.0	185.5	0	10014
10018	10-Feb-18	105	104	4278Y	TYS	644.0	3.8	4.5	167.4	0	10017

The destinations are indicated by standard three-letter airport codes. For example,
STL = St. Louis, MO ATL = Atlanta, GA BNA = Nashville, TN

Table name: AIRCRAFT

AC_NUMBER	MOD_CODE	AC_TTAF	AC_TTEL	AC_TTER
1484P	PA23-250	1833.1	1833.1	101.8
2289L	C-90A	4243.8	768.9	1123.4
2778V	PA31-350	7992.9	1513.1	789.5
4278Y	PA31-350	2147.3	622.1	243.2

AC-TTAF = Aircraft total time, airframe (hours)
AC-TTEL = Total time, left engine (hours)
AC_TTER = Total time, right engine (hours)

In a fully developed system, such attribute values would be updated by application software when the CHARTER table entries were posted.

Table name: MODEL

MOD_CODE	MOD_MANUFACTURER	MOD_NAME	MOD_SEATS	MOD_CHG_MILE
B200	Beechcraft	Super KingAir	10	1.93
C-90A	Beechcraft	KingAir	8	2.67
PA23-250	Piper	Aztec	6	1.93
PA31-350	Piper	Navajo Chieftain	10	2.35

Customers are charged per round-trip mile, using the MOD_CHG_MILE rate. The MOD_SEATS column lists the total number of seats in the airplane, including the pilot and copilot seats. Therefore, a PA31-350 trip that is flown by a pilot and a copilot has eight passenger seats available.

Use the database shown in Figure P3.24 to answer Problems 24–31. AviaCo is an aircraft charter company that supplies on-demand charter flight services using a fleet of four aircraft. Aircraft are identified by a unique registration number. Therefore, the aircraft registration number is an appropriate primary key for the AIRCRAFT table.

FIGURE P3.24 THE CH03_AVIACO DATABASE TABLES (CONTINUED)

Table name: PILOT **Database name: Ch03_AviaCo**

EMP_NUM	PIL_LICENSE	PIL_RATINGS	PIL_MED_TYPE	PIL_MED_DATE	PIL_PT135_DATE
101	ATP	ATP/SEL/MEL/Instr/CFII	1	20-Jan-18	11-Jan-18
104	ATP	ATP/SEL/MEL/Instr	1	18-Dec-17	17-Jan-18
105	COM	COMM/SEL/MEL/Instr/CFI	2	05-Jan-18	02-Jan-18
106	COM	COMM/SEL/MEL/Instr	2	10-Dec-17	02-Feb-18
109	COM	ATP/SEL/MEL/SES/Instr/CFII	1	22-Jan-18	15-Jan-18

The pilot licenses shown in the PILOT table include the ATP = Airline Transport Pilot and COM = Commercial Pilot. Businesses that operate "on demand" air services are governed by Part 135 of the Federal Air Regulations (FARs) that are enforced by the Federal Aviation Administration (FAA). Such businesses are known as "Part 135 operators." Part 135 operations require that pilots successfully complete flight proficiency checks each six months. The "Part 135" flight proficiency check date is recorded in PIL_PT135_DATE. To fly commercially, pilots must have at least a commercial license and a 2nd class medical certificate (PIL_MED_TYPE = 2.)

The PIL_RATINGS include
SEL = Single Engine, Land
SES = Single Engine (Sea)
CFI = Certified Flight Instructor
MEL = Multi-engine Land
Instr. = Instrument
CFII = Certified Flight Instructor, Instrument

Table name: EMPLOYEE

EMP_NUM	EMP_TITLE	EMP_LNAME	EMP_FNAME	EMP_INITIAL	EMP_DOB	EMP_HIRE_DATE
100	Mr.	Kolmycz	George	D	15-Jun-62	15-Mar-08
101	Ms.	Lewis	Rhonda	G	19-Mar-85	25-Apr-06
102	Mr.	Vandam	Rhett		14-Nov-78	18-May-13
103	Ms.	Jones	Anne	M	11-May-94	26-Jul-17
104	Mr.	Lange	John	P	12-Jul-91	20-Aug-10
105	Mr.	Williams	Robert	D	14-Mar-95	19-Jun-17
106	Mrs.	Duzak	Jeanine	K	12-Feb-88	13-Mar-18
107	Mr.	Diante	Jorge	D	01-May-95	02-Jul-16
108	Mr.	Wiesenbach	Paul	R	14-Feb-86	03-Jun-13
109	Ms.	Travis	Elizabeth	K	18-Jun-81	14-Feb-16
110	Mrs.	Genkazi	Leighla	W	19-May-90	29-Jun-10

Table name: CUSTOMER

CUS_CODE	CUS_LNAME	CUS_FNAME	CUS_INITIAL	CUS_AREACODE	CUS_PHONE	CUS_BALANCE
10010	Ramas	Alfred	A	615	844-2573	0.00
10011	Dunne	Leona	K	713	894-1238	0.00
10012	Smith	Kathy	W	615	894-2285	896.54
10013	Olowski	Paul	F	615	894-2180	1285.19
10014	Orlando	Myron		615	222-1672	673.21
10015	O'Brian	Amy	B	713	442-3381	1014.56
10016	Brown	James	G	615	297-1228	0.00
10017	Williams	George		615	290-2556	0.00
10018	Farriss	Anne	G	713	382-7185	0.00
10019	Smith	Olette	K	615	297-3809	453.98

The nulls in the CHARTER table's CHAR_COPILOT column indicate that a copilot is not required for some charter trips or for some aircraft. Federal Aviation Administration (FAA) rules require a copilot on jet aircraft and on aircraft that have a gross take-off weight over 12,500 pounds. None of the aircraft in the AIRCRAFT table are governed by this requirement; however, some customers may require the presence of a copilot for insurance reasons. All charter trips are recorded in the CHARTER table.

Note

Earlier in the chapter, you were instructed to avoid homonyms and synonyms. In this problem, both the pilot and the copilot are listed in the PILOT table, but EMP_NUM cannot be used for both in the CHARTER table. Therefore, the synonyms CHAR_PILOT and CHAR_COPILOT were used in the CHARTER table.

Although the solution works in this case, it is very restrictive, and it generates nulls when a copilot is not required. Worse, such nulls proliferate as crew requirements change. For example, if the AviaCo charter company grows and starts using larger aircraft, crew requirements may increase to include flight engineers and load masters. The CHARTER table would then have to be modified to include the additional crew assignments; such attributes as CHAR_FLT_ENGINEER and CHAR_LOADMASTER would have to be added to the CHARTER table. Given this change, each time a smaller aircraft flew a charter trip without the number of crew members required in larger aircraft, the missing crew members would yield additional nulls in the CHARTER table.

You will have a chance to correct those design shortcomings in Problem 27. The problem illustrates two important points:

1. Don't use synonyms. If your design requires the use of synonyms, revise the design!
2. To the greatest possible extent, design the database to accommodate growth without requiring structural changes in the database tables. Plan ahead and try to anticipate the effects of change on the database.

24. For each table, identify each of the following when possible:
    a. The primary key
    b. A superkey
    c. A candidate key
    d. The foreign key(s)
    e. A secondary key
25. Create the ERD. (*Hint*: Look at the table contents. You will discover that an AIRCRAFT can fly many CHARTER trips but that each CHARTER trip is flown by one AIRCRAFT, that a MODEL references many AIRCRAFT but that each AIRCRAFT references a single MODEL, and so on.)
26. Create the relational diagram.
27. Modify the ERD you created in Problem 25 to eliminate the problems created by the use of synonyms. (*Hint*: Modify the CHARTER table structure by eliminating the CHAR_PILOT and CHAR_COPILOT attributes; then create a composite table named CREW to link the CHARTER and EMPLOYEE tables. Some crew members, such as flight attendants, may not be pilots. That's why the EMPLOYEE table enters into this relationship.)
28. Create the relational diagram for the design you revised in Problem 27.

You want to see data on charters flown by either Robert Williams (employee number 105) or Elizabeth Travis (employee number 109) as pilot or copilot, but not charters flown by both of them. Complete Problems 29–31 to find this information.

29. Create the table that would result from applying the SELECT and PROJECT relational operators to the CHARTER table to return only the CHAR_TRIP, CHAR_PILOT, and CHAR_COPILOT attributes for charters flown by either employee 105 or employee 109.
30. Create the table that would result from applying the SELECT and PROJECT relational operators to the CHARTER table to return only the CHAR_TRIP, CHAR_PILOT, and CHAR_COPILOT attributes for charters flown by both employee 105 and employee 109.
31. Create the table that would result from applying a DIFFERENCE relational operator of your result from Problem 29 to your result from Problem 30.