16 USING RELATIONAL DATABASES

After studying Chapter 16, you should be able to:

- Understand relational database fundamentals
- Create databases and table descriptions
- Identify primary keys
- Understand database structure notation
- Understand the principles of adding, deleting, updating, and sorting records within a table
- Write queries
- Understand relationships between tables and functional dependence between columns
- Recognize poor table design
- Understand anomalies, normal forms, and the normalization process
- Understand the performance and security issues connected to database administration

UNDERSTANDING RELATIONAL DATABASE FUNDAMENTALS

When you store data items for use within computer systems, they are often stored in what is known as a data hierarchy, where the smallest usable unit of data is the character, often a letter or number. Characters are grouped together to form fields, such as `firstName`, `lastName`, and `socialSecurityNumber`. Related fields are often grouped together to form records—groups of fields that go together because they represent attributes of some entity, such as an employee, a customer, an inventory item, or a bank account. Files are composed of related records; for example, a file might contain a record for each employee in a company or each account at a bank.

TIP You first learned about the data hierarchy in Chapter 1 of this book. The terms *character*, *field*, *record*, and *file* were defined there, and you have been using these terms throughout this book.

Most organizations store many files that contain the data they need to operate their businesses; for example, businesses often need to maintain files containing data about employees, customers, inventory items, and orders. Many organizations use database software to organize the information in these files. A **database** holds a group of files that an organization needs to support its applications. In a database, the files often are called **tables** because you can arrange their contents in rows and columns. Real-life examples of database-like tables abound. For example, consider the listings in a telephone book. Each listing in a city directory might contain four columns, as shown in Figure 16-1—last name, first name, street address, and phone number. Although your local phone directory might not store its data in the rigid columnar format shown in the figure, it could. You can see that each column represents a field and that each row represents one record. You can picture a table within a database in the same way.

FIGURE 16-1: A TELEPHONE BOOK TABLE

Last name	First name	Address	Phone
Abbott	William	123 Oak Lane	490-8920
Ackerman	Kimberly	467 Elm Drive	787-2781
Adams	Stanley	8120 Pine Street	787-0129
Adams	Violet	347 Oak Lane	490-8912
Adams	William	12 Second Street	490-3667

TIP One record or row is also sometimes called an **entity**; however, many definitions of "entity" exist in database texts. One column (field) can also be called an **attribute**.

Figure 16-1 includes five records, each representing a unique person. It is relatively easy to scan this short list of names to find a person's phone number; of course, telephone books contain many more records. Some telephone book users, such as telemarketers or even the phone company, might prefer to look up a number in a book in which the records are organized in telephone-number order. Others, such as door-to-door salespeople, might prefer a telephone book in which the records are organized in street-address order. Most people, however, prefer a telephone book in

which the records are organized as shown, in alphabetical order by last name. It is most convenient for different users when computerized databases can sort records in various orders based on the contents of different columns.

Unless you are reading a telephone book for a very small town, a last name alone often is not sufficient to identify a person. In the example in Figure 16-1, three people have the last name of Adams. For these records, you need to examine the first name before you can determine the correct phone number. In a large city, many people might have the same first and last names; in that case, you might also need to examine the street address to identify a person. As with the telephone book, in most computerized database tables, it is important to have a way to uniquely identify each record, even if it means using multiple columns. A value that uniquely identifies a record is called a **primary key**, or a **key** for short. Key fields often are defined as a single table column, but as with the telephone book, keys can be constructed from multiple columns; a key constructed from multiple columns is a **compound key**.

TIP You learn more about key fields and compound keys later in this chapter. Compound keys also are known as **composite keys**.

Telephone books are republished periodically because changes have occurred—new people have moved into the city and become telephone customers, and others have left, canceled service, or changed phone numbers. With computerized database tables, you also need to add, delete, and modify records, although usually far more frequently than phone books are published.

Telephone books often contain thousands of records. Computerized database tables also frequently contain thousands of records, or rows, and each row might contain entries in dozens of columns. Handling and organizing all the data contained in an organization's tables requires sophisticated software. **Database management software** is a set of programs that allows users to:

- Create table descriptions.
- Identify keys.
- Add, delete, and update records within a table.
- Arrange records within a table so they are sorted by different fields.
- Write questions that select specific records from a table for viewing.
- Write questions that combine information from multiple tables. This is possible because the database management software establishes and maintains relationships between the columns in the tables. A group of database tables from which you can make these connections is a **relational database**.
- Create reports that allow users to easily interpret your data, and create forms that allow users to view and enter data using an easy-to-manage interactive screen.
- Keep data secure by employing sophisticated security measures.

If you have used different word-processing or spreadsheet programs, you know that each version works a little differently, although each carries out the same types of tasks. Like other computer programs, each database management software package operates differently; however, with each, you need to perform the same types of tasks.

CREATING DATABASES AND TABLE DESCRIPTIONS

Creating a useful database requires a lot of planning and analysis. You must decide what data will be stored, how that data will be divided between tables, and how the tables will interrelate. Before you create any tables, you must create the database itself. With most database software packages, creating the database that will hold the tables requires nothing more than providing a name for the database and indicating the physical location, perhaps a hard disk drive, where the database will be stored. When you save a table, it is conventional to provide it with a name that begins with the prefix "tbl"—for example, `tblCustomers`. Your databases often become filled with a variety of objects—tables, forms that users can use for data entry, reports that organize the data for viewing, queries that select subsets of data for viewing, and so on. Using naming conventions, such as beginning each table name with a prefix that identifies it as a table, helps you to keep track of the various objects in your system.

TIP

Many database management programs suggest that you use a generic name such as Table1 when you save a table description. Usually, a more descriptive name is more useful to you as you continue to create objects.

Before you can enter any data into a database table, you must design the table. At minimum, this involves two tasks:

- You must decide what columns your table needs, and provide names for them.
- You must provide a data type for each column.

For example, assume you are designing a customer database table. Figure 16-2 shows some column names and data types you might use.

FIGURE 16-2: CUSTOMER TABLE DESCRIPTION

Column	Data type
customerID	text
lastName	text
firstName	text
streetAddress	text
balanceOwed	numeric

TIP

A table description closely resembles the record descriptions you have used with data files throughout this book.

TIP

It is important to think carefully about the original design of a database. After the database has been created and data has been entered, it could be difficult and time-consuming to make changes.

The table description in Figure 16-2 uses just two data types—text and numeric. Text columns can hold any type of characters—letters or digits. Numeric columns can hold numbers only. Depending on the database management software you use, you might have many more sophisticated data types at your disposal. For example, some database software divides the numeric data type into several subcategories such as integer (whole number only) values and double-precision numbers (numbers that contain decimals). Other options might include special categories for currency numbers (representing dollars and cents), dates, and Boolean columns (representing true or false). At the least, all database software recognizes the distinction between text and numeric data.

TIP

You have been aware of the distinction that computers make between character and numeric data throughout this book. Because of the way computers handle data, every type of software observes this distinction. Throughout this book, the term "char" has been used to describe text fields. The term "text" is used in this chapter only because it is the term that popular database packages use.

TIP

Unassigned variables within computer programs might be empty (containing a null value), or might contain unknown or garbage values. Similarly, columns in database tables might also contain null or unknown values. When a field in a database contains a null value, it does not mean that the field holds a 0 or a space; it means that no data has been entered for the field at all. Although "null" and "empty" are used synonymously by many database developers, the terms have slightly different meanings to Visual Basic programmers.

The table description in Figure 16-2 uses one-word column names and camel casing, in the same way that variable names have been defined throughout this book. Many database software packages do not require that data column names be single words without embedded spaces, but many database table designers prefer single-word names because they resemble variable names in programs. In addition, when you write programs that access a database table, the single-word field names can be used "as is," without special syntax to indicate the names that represent a single field. As a further advantage, when you use a single word to label each database column, it is easier to understand whether just one column is being referenced, or several.

The `customerID` column in Figure 16-2 is defined as a text field or text column. If `customerID` numbers are composed entirely of digits, this column could also be defined as numeric. However, many database designers feel that columns should be defined as numeric only if they need to be—that is, only if they might be used in arithmetic calculations. The description in Figure 16-2 follows this convention by declaring `customerID` to be a text column.

Many database management software packages allow you to add a narrative description of each data column to a table. This allows you to make comments that become part of the table. These comments do not affect the way the table operates; they simply serve as documentation for those who are reading a table description. For example, you might want to make a note that `customerID` should consist of five digits, or that `balanceOwed` should not exceed a given limit. Some software allows you to specify that values for a certain column are required—the user cannot create a record without providing data for these columns. In addition, you might be able to indicate value limits for a column—high and low numbers between which the column contents must fall.

IDENTIFYING PRIMARY KEYS

In most tables you create for a database, you want to identify a column, or possibly a combination of columns, as the table's key column or field, also called the primary key. The primary key in a table is the column that makes each record different from all others. For example, in the customer table in Figure 16-2, the logical choice for a primary key is the `customerID` column—each customer record that is entered into the customer table has a unique value in this column. Many customers might have the same first name or last name (or both), and multiple customers also might have the same street address or balance due. However, each customer possesses a unique ID number.

Other typical examples of primary keys include:

- A student ID number in a table that contains college student information
- A part number in a table that contains inventory items
- A Social Security number in a table that contains employee information

In each of these examples, the primary key uniquely identifies the row. For example, each student has a unique ID number assigned by the college. Other columns in a student table would not be adequate keys—many students have the same last name, first name, hometown, or major.

TIP It is no coincidence that each of the preceding examples of a key is a number, such as a student ID number or item number. Usually, assigning a number to each row in a table is the simplest and most efficient method of obtaining a useful key. However, it is possible that a table's key could be a text field.

The primary key is important for several reasons:

- You can configure your database software to prevent multiple records from containing the same value in this column, thus avoiding data-entry errors.
- You can sort your records in this order before displaying or printing them.
- You use this column when setting up relationships between this table and others that will become part of the same database.
- In addition, you need to understand the concept of the primary key when you normalize a database—a concept you will learn more about later in this chapter.

TIP In some database software packages, such as Microsoft Access, you indicate a primary key simply by selecting a column name and clicking a button that is labeled with a key icon.

In some tables, when no identifying number has been assigned to the rows, more than one column is required to construct a primary key. A multicolumn key is a compound key. For example, consider Figure 16-3, which might be used by a residence hall administrator to store data about students living on a university campus. Each room in a building has a number and two students, each assigned to either bed A or bed B.

FIGURE 16-3: TABLE CONTAINING RESIDENCE HALL STUDENT RECORDS

hall	room	bed	lastName	firstName	major
Adams	101	A	Fredricks	Madison	Chemistry
Adams	101	B	Garza	Lupe	Psychology
Adams	102	A	Liu	Jennifer	CIS
Adams	102	B	Smith	Crystal	CIS
Browning	101	A	Patel	Sarita	CIS
Browning	101	B	Smith	Margaret	Biology
Browning	102	A	Jefferson	Martha	Psychology
Browning	102	B	Bartlett	Donna	Spanish
Churchill	101	A	Wong	Cheryl	CIS
Churchill	101	B	Smith	Madison	Chemistry
Churchill	102	A	Patel	Jennifer	Psychology
Churchill	102	B	Jones	Elizabeth	CIS

In Figure 16-3, no single column can serve as a primary key. Many students live in the same residence hall, and the same room numbers exist in the different residence halls. In addition, some students have the same last name, first name, or major. It is even possible that two students with the same first name, last name, or major are assigned to the same room. In this case, the best primary key is a multicolumn key that combines residence hall, room number, and bed number (`hall`, `room`, and `bed`). "Adams 101 A" identifies a single room and student, as does "Churchill 102 B".

TIP

A primary key should be **immutable**, meaning that a value does not change during normal operation. In other words, in Figure 16-3, "Adams 102 A" will always pertain to a fixed location, even though the resident or her major might change. Of course, the school might choose to change the name of a residence hall—for example, to honor a benefactor—but that action would fall outside the range of "normal operation." (In object-oriented programming, a class is immutable if it contains no methods that allow changes to its attributes after construction.)

TIP

Sometimes, there are several columns that could serve as the key. For example, if an employee record contains both a company-assigned employee ID and a Social Security number, then both columns are **candidate keys**. After you choose a primary key from among candidate keys, the remaining candidate keys become **alternate keys**.

TIP

Even if there were only one student named Smith, for example, or only one Psychology major in the table in Figure 16-3, those fields still would not be good primary key candidates because of the potential for future Smiths and Psychology majors within the database. Analyzing existing data is not a foolproof way to select a good key; you must also consider likely future data.

TIP

As an alternative to selecting three columns to create the compound key for the table in Figure 16-3, many database designers prefer to simply add a new column containing a bed location ID number that would uniquely identify each row. Many database designers feel that a primary key should be short to minimize the amount of storage required for it in all the tables that refer to it.

Usually, after you have identified the necessary fields and their data types, and identified the primary key, you are ready to save your table description and begin to enter data.

UNDERSTANDING DATABASE STRUCTURE NOTATION

A shorthand way to describe a table is to use the table name followed by parentheses containing all the field names, with the primary key underlined. Thus, when a table is named `tblStudents` and contains columns named `idNumber`, `lastName`, `firstName`, and `gradePointAverage`, and `idNumber` is the key, you can reference the table using the following notation:

```
tblStudents(idNumber, lastName, firstName, gradePointAverage)
```

Although this shorthand notation does not provide you with information about data types or range limits on values, it does provide you with a quick overview of the structure of a table.

TIP

Some database designers insert an asterisk after the key instead of underlining it.

TIP

The key does not have to be the first attribute listed in a table reference, but frequently it is.

ADDING, DELETING, AND UPDATING RECORDS WITHIN TABLES

Entering data into an already created table is not difficult, but it requires a good deal of time and accurate typing. Depending on the application, the contents of the tables might be entered over the course of many months or years by any number of data-entry personnel. Entering data of the wrong type is not allowed by most database software. In addition, you might have set up your table to prevent duplicate data in specific fields, or to prevent data entry outside of specified bounds in other fields. With some database software, you type data into rows representing each record, and columns representing each field in each record, much as you would enter data into a spreadsheet. With other software, you can create on-screen forms to make data entry more user-friendly. Some software does not allow you to enter a partial record; that is, you might not be allowed to leave any fields blank.

TIP

Computer professionals use the acronym GIGO, which stands for "garbage in, garbage out." It means that if you enter invalid input data into an application, the output results will be worthless. You first learned this expression in Chapter 10.

Deleting records from and modifying records within a database table are also relatively easy tasks. In most organizations, most of the important data is in a constant state of change. Maintaining the data records so they are up to date is a vital part of any database management system.

TIP

In many database systems, some "deleted" records are not physically removed. Instead, they are just marked as deleted so they will not be used to process active records. For example, a company might want to retain data about former employees, but not process them with current personnel reports. On the other hand, an employee record that was entered by mistake would be permanently removed from the database.

SORTING THE RECORDS IN A TABLE

Database management software generally allows you to sort a table based on any column, letting you view your data in the way that is most useful to you. For example, you might want to view inventory items in alphabetical order, or from the most to the least expensive. You also can sort by multiple columns—for example, you might sort employees by first name within last name (so that Aaron Black is listed before Andrea Black), or by department within first name within last name (so that Aaron Black in Department 1 is listed before another Aaron Black in Department 6).

TIP

When performing sorts on multiple fields, the software sorts first by a primary sort—for example, last name. After all those with the same primary sort key are grouped, the software sorts by the secondary key—for example, first name.

After rows are sorted, they usually can be grouped. For example, you might want to sort customers by their zip code, or employees by the department in which they work; in addition, you might want counts or subtotals at the end of each group. Database software provides the means to create displays in the formats that suit your present information needs.

TIP

When a database program includes counts or totals at the end of each sorted group, it is creating a control break report. You learned about control break reports in Chapter 7.

CREATING QUERIES

Data tables often contain hundreds or thousands of rows; making sense out of that much information is a daunting task. Frequently, you want to cull subsets of data from a table you have created. For example, you might want to view only those customers with an address in a specific state, only those inventory items whose quantity in stock has fallen below the normal reorder point, or only those employees who participate in an insurance plan. Besides limiting records, you might also want to limit the columns that you view. For example, student records might contain dozens of fields, but a school administrator might only be interested in looking at names and grade point averages. The questions that cause the database software to extract the appropriate records from a table and specify the fields to be viewed are called queries; a **query** is simply a question asked using the syntax that the database software can understand.

Depending on the software you use, you might create a query by filling in blanks (a process called **query by example**) or by writing statements similar to those in many programming languages. The most common language that database administrators use to access data in their tables is **Structured Query Language**, or **SQL**. The basic form of the SQL command that retrieves selected records from a table is **SELECT-FROM-WHERE**. The `SELECT–FROM–WHERE` SQL statement:

- Selects the columns you want to view
- From a specific table
- Where one or more conditions are met

TIP "SQL" frequently is pronounced "sequel"; however, several SQL product Web sites insist that the official pronunciation is "S-Q-L." Similarly, some people pronounce GUI as "gooey" and others insist that it should be "G-U-I." In general, a preferred pronunciation evolves in an organization. The TLA, or three-letter abbreviation, is the most popular type of abbreviation in technical terminology.

For example, suppose a customer table named `tblCustomer` contains data about your business customers and that the structure of the table is `tblCustomer(custId, lastName, state)`. Then, a statement such as:

```
SELECT custId, lastName FROM tblCustomer WHERE state = "WI"
```

would display a new table containing two columns—`custId` and `lastName`—and only as many rows as needed to hold those customers whose state column contains "WI". Besides using = to mean "equal to," you can use the comparison conditions > (greater than), < (less than), >= (greater than or equal to), and <= (less than or equal to). As you have already learned from working with programming variables throughout this book, text field values are always contained within quotes, whereas numeric values are not.

TIP Conventionally, SQL keywords such as SELECT appear in all uppercase; this book follows that convention.

TIP In database management systems, a particular way of looking at a database is sometimes called a **view**. Typically, a view arranges records in some order and makes only certain fields visible. The different views provided by database software are virtual; that is, they do not affect the physical organization of the database.

To select all fields for each record in a table, you can use the asterisk as a wildcard; a wildcard is a symbol that means "any" or "all." For example, `SELECT * from tblCustomer WHERE state = "WI"` would select all columns for every customer whose state is "WI", not just specifically named columns. To select all customers from a table, you can omit the `WHERE` clause in a `SELECT-FROM-WHERE` statement. In other words, `SELECT * FROM tblCustomer` selects all columns for all customers.

You learned about making selections in computer programs much earlier in this book, and you have probably noticed that `SELECT-FROM-WHERE` statements serve the same purpose as programming decisions. As with decision statements in programs, when using SQL, you can create compound conditions using `AND` or `OR` operators. In addition, you can precede any condition with a `NOT` operator to achieve a negative result. In summary, Figure 16-4 shows a database table named `tblInventory` with the following structure: `tblInventory(itemNumber, description, quantityInStock, price)`. The table contains five records. Figure 16-5 lists several typical SQL `SELECT` statements you might use with `tblInventory`, and explains each.

FIGURE 16-4: THE `tblInventory` TABLE

itemNumber	description	quantityInStock	price
144	Pkg 12 party plates	250	$14.99
231	Helium balloons	180	$2.50
267	Paper streamers	68	$1.89
312	Disposable tablecloth	20	$6.99
383	Pkg 20 napkins	315	$2.39

FIGURE 16-5: SAMPLE SQL STATEMENTS AND EXPLANATIONS

SQL statement	Explanation
`SELECT itemNumber, price FROM tblInventory`	Shows only the item number and price for all five records.
`SELECT * FROM tblInventory WHERE price > 5.00`	Shows all fields from only those records where price is over $5.00—items 144 and 312.
`SELECT itemNumber FROM tblInventory WHERE quantityInStock > 200 AND price > 10.00`	Shows item number 144—the only record that has a quantity greater than 200 as well as a price greater than $10.00.
`SELECT description, price FROM tblInventory WHERE description = "Pkg 20 napkins" OR itemNumber < 200`	Shows the description and price fields for the package of 12 party plates and the package of 20 napkins. Each selected record must satisfy only one of the two criteria.
`SELECT itemNumber FROM tblInventory WHERE NOT price < 14.00`	Shows the item number for the only record where the price is not less than $14.00—item 144.

UNDERSTANDING TABLE RELATIONSHIPS

Most database applications require many tables, and these applications also require that the tables be related. The connection between two tables is a **relationship**, and the database containing the relationships is called a relational database. Connecting two tables based on the values in a common column is called a **join operation**, or more simply, a **join**; the column on which they are connected is the **join column**. A virtual, or imaginary, table that is displayed as the result of the query takes some of its data from each joined table. For example, in Figure 16-6, the `customerNumber` column is the join column that could produce a virtual image when a user makes a query. When a user asks to see the

name of a customer associated with a specific order number, or a list of all the names of customers who have ordered a specific item, then a joined table is produced. The three types of relationships that can exist between tables are:

- One-to-many
- Many-to-many
- One-to-one

FIGURE 16-6: SAMPLE CUSTOMERS AND ORDERS

tblCustomers

customerNumber	customerName
214	Kowalski
215	Jackson
216	Lopez
217	Thompson
218	Vitale

tblOrders

orderNumber	customerNumber	orderQuantity	orderItem	orderDate
10467	215	2	HP203	10/15/2007
10468	218	1	JK109	10/15/2007
10469	215	4	HP203	10/16/2007
10470	216	12	ML318	10/16/2007
10471	214	4	JK109	10/16/2007
10472	215	1	HP203	10/16/2007
10473	217	10	JK109	10/17/2007

UNDERSTANDING ONE-TO-MANY RELATIONSHIPS

A **one-to-many relationship** is one in which one row in a table can be related to many rows in another table. It is the most common type of relationship between tables. Consider the following tables:

```
tblCustomers(customerNumber, customerName)
tblOrders(orderNumber, customerNumber, orderQuantity, orderItem, orderDate)
```

The `tblCustomers` table contains one row for each customer, and `customerNumber` is the primary key. The `tblOrders` table contains one row for each order, and each order is assigned an `orderNumber`, which is the primary key in this table.

In most businesses, a single customer can place many orders. For example, in the sample data in Figure 16-6, customer 215 has placed three orders. One row in the `tblCustomers` table can correspond to, and can be related to, many rows in the `tblOrders` table. This means there is a one-to-many relationship between the two tables `tblCustomers` and `tblOrders`. The "one" table (`tblCustomers`) is the **base table** in this relationship, and the "many" table (`tblOrders`) is the **related table**.

When two tables are related in a one-to-many relationship, the relationship occurs based on the values in one or more columns in the tables. In this example, the column, or attribute, that links the two tables together is the `customerNumber` attribute. In the `tblCustomers` table, `customerNumber` is the primary key, but in the `tblOrders` table, `customerNumber` is not a key—it is a **non-key attribute**. When a column that is not a key in a

table contains an attribute that is a key in a related table, the column is called a **foreign key**. When a base table is linked to a related table in a one-to-many relationship, it is always the primary key of the base table that is related to the foreign key in the related table. In this example, `customerNumber` in the `tblOrders` table is a foreign key.

TIP A key in a base table and the foreign key in the related table do not need to have the same name; they only need to contain the same type of data. Some database management software programs automatically create a relationship for you if the columns in two tables you select have the same name and data type. However, if this is not the case (for example, if the column is named `customerNumber` in one table and `custID` in another), you can explicitly instruct the software to create the relationship.

UNDERSTANDING MANY-TO-MANY RELATIONSHIPS

Another example of a one-to-many relationship is depicted with the following tables:

```
tblItems(itemNumber, itemName, itemPurchaseDate, itemPurchasePrice,
   itemCategoryId)
tblCategories(categoryId, categoryName, categoryInsuredAmount)
```

Assume you are creating these tables to keep track of all the items in your household for insurance purposes. You want to store data about items such as your sofa, stereo, refrigerator, and so on. The `tblItems` table contains the name, purchase date, and purchase price of each item. In addition, this table contains the ID number of the item category (Appliance, Jewelry, Antique, and so on) to which the item belongs. You need the category of each item because your insurance policy has specific coverage limits for different types of property. For example, with many insurance policies, antiques might have a different coverage limit than appliances, or jewelry might have a different limit than furniture. Sample data for these tables is shown in Figure 16-7.

The primary key of the `tblItems` table is `itemNumber`, a unique identifying number that you have assigned to each item that you own. (You might even prepare labels with these numbers and stick a label on each item in an inconspicuous place.) The `tblCategories` table contains the category names and the maximum insured amounts for the specific categories. For example, one row in this table may have a `categoryName` of "Jewelry" and a `categoryInsuredAmount` of $15,000. The primary key for the `tblCategories` table is `categoryId`, which is simply a uniquely assigned value for each property category.

The two tables in Figure 16-7 have a one-to-many relationship. Which is the "one" table and which is the "many" table? Or, asked in another way, which is the base table and which is the related table? You have probably determined that the `tblCategories` table is the base table (the "one" table) because one category can describe many items that you own. Therefore, the `tblItems` table is the related table (the "many" table); that is, there are many items that fall into each category. The two tables are linked with the `categoryId` attribute, which is the primary key in the base table (`tblCategories`) and a foreign key in the related table (`tblItems`).

FIGURE 16-7: SAMPLE ITEMS AND CATEGORIES: A ONE-TO-MANY RELATIONSHIP

tblItems

itemNumber	itemName	itemPurchaseDate	itemPurchasePrice	itemCategoryId
1	Sofa	1/13/2001	$6,500	5
2	Stereo	2/10/2003	$1,200	6
3	Refrigerator	5/12/2003	$750	1
4	Diamond ring	2/12/2004	$42,000	2
5	TV	7/11/2004	$285	6
6	Rectangular pine coffee table	4/21/2005	$300	5
7	Round pine end table	4/21/2005	$200	5

tblCategories

categoryId	categoryName	categoryInsuredAmount
1	Appliance	$30,000
2	Jewelry	$15,000
3	Antique	$10,000
4	Clothing	$25,000
5	Furniture	$5,000
6	Electronics	$2,500
7	Miscellaneous	$5,000

In the tables in Figure 16-7, one row in the `tblCategories` table relates to multiple items you own. The opposite is not true—that is, one item in the `tblItems` table cannot relate to multiple categories in the `tblCategories` table. The row in the `tblItems` table that describes the "rectangular pine coffee table" relates to one specific category in the `tblCategories` table—the Furniture category. However, what if you own a rectangular pine coffee table that has a built-in DVD player, or a diamond ring that is an antique, or a stereo that could also be worn as a hat on a rainy day? Even though this last example is humorous, it does bring up an important consideration.

The structure of the tables shown in Figure 16-7 and the relationship between those tables are designed to support a particular application—keeping track of possessions for insurance purposes. If you acquired a sofa with a built-in CD player and speakers, what would you do? For guidance, you probably would call your insurance agent. If the agent said, "Well, for insurance purposes that item is considered a piece of furniture," then the existing table structures and relationships are adequate.

However, if the insurance agent said, "Well, actually a sofa with a CD player is considered a special type of hybrid item, and that category of property has a specific maximum insured amount," then you could simply create a new row in the `tblCategories` table to describe this special hybrid category—perhaps Electronic Furniture. This new category would acquire a category number, and then you could associate the CD-sofa to the new category using the foreign key in the `tblItems` table.

However, what if your insurance agent said, "You know, that's a good question. We've never had that come up before—a sofa with a CD player. What we would probably do if you filed a claim because the sofa was damaged is to take a look at it to try to determine whether the sofa is mostly a piece of furniture or mostly a piece of electronics." This answer presents a problem to your database. You may want to categorize your new sofa as both a furniture item *and* an electronic item. The existing table structures, with their one-to-many relationship, would not support this because the current design limits any specific item to one and only one category. When you insert a row into the `tblItems` table to describe the new CD-sofa, you can assign the Furniture code to the foreign key `itemCategory`, or you can assign the Electronics code, but not both.

If you want to assign the new CD-sofa to both categories (Furniture and Electronics), you have to change the design of the table structures and relationships, because there is no longer a one-to-many relationship between the two tables. Now, there is a **many-to-many relationship**—one in which multiple rows in each table can correspond to multiple rows in the other. That is, in this example, one row in the `tblCategories` table (for example, Furniture) can relate to many rows in the `tblItems` table (for example, sofa and coffee table), *and* one row in the `tblItems` table (for example, the sofa with the built-in CD player) can relate to multiple rows in the `tblCategories` table.

The `tblItems` table contains a foreign key named `itemCategoryId`. If you want to change the application so that one specific row in the `tblItems` table can link to many rows (and, therefore, many `categoryIds`) in the `tblCategories` table, you cannot continue to maintain the foreign key `itemCategoryId` in the `tblItems` table, because one item may be assigned to many categories. You could change the structure of the `tblItems` table so that you can assign multiple `itemCategoryIds` to one specific row in that table, but as you will learn later in this chapter, that approach leads to many problems using the data. Therefore, it is not an option.

The simplest way to support a many-to-many relationship between the `tblItems` and `tblCategories` tables is to remove the `itemCategoryId` attribute (what was once the foreign key) from the `tblItems` table, producing:

```
tblItems(itemNumber, itemName, itemPurchaseDate, itemPurchasePrice)
```

The `tblCategories` table structure remains the same:

```
tblCategories(categoryId, categoryName, categoryInsuredAmount)
```

With just the preceding two tables, there is no way to know that any specific row(s) in the `tblItems` table link(s) to any specific row(s) in the `tblCategories` table, so you create a new table called `tblItemsCategories` that contains the primary keys from the two tables that you want to link in a many-to-many relationship. This table is depicted as:

```
tblItemsCategories(itemNumber, categoryId)
```

Notice that this new table contains a compound primary key—both `itemNumber` and `categoryId` are underlined. The `itemNumber` value of 1 might be associated with many `categoryId`s. Therefore, `itemNumber` alone cannot be the primary key because the same value may occur in many rows. Similarly, a `categoryId` might relate to many different `itemNumber`s; this would disallow using just the `categoryId` as the primary key.

However, a combination of the two attributes `itemNumber` and `categoryId` results in a unique primary key value for each row of the `tblItemsCategories` table.

The purpose of all this is to create a many-to-many relationship between the `tblItems` and `tblCategories` tables. The `tblItemsCategories` table contains two attributes; together, these attributes are the primary key. In addition, each of these attributes separately is a foreign key to one of the two original tables. The `itemNumber` attribute in the `tblItemsCategories` table is a foreign key that links to the primary key of the `tblItems` table. The `categoryId` attribute in the `tblItemsCategories` table links to the primary key of the `tblCategories` table. Now, there is a one-to-many relationship between the `tblItems` table (the "one," or base table) and the `tblItemsCategories` table (the "many," or related table) and a one-to-many relationship between the `tblCategories` table (the "one," or base table) and the `tblItemsCategories` table (the "many," or related table). This, in effect, implies a many-to-many relationship between the two base tables (`tblItems` and `tblCategories`).

Figure 16-8 shows the new tables holding a few items. The sofa (`itemNumber` 1) in the `tblItems` table is associated with the Furniture category (`categoryId` 5) in the `tblCategories` table because the first row of the `tblItemsCategories` table contains a 1 and a 5. Similarly, the stereo (`itemNumber` 2) in the `tblItems` table is associated with the Electronics category (`categoryId` 6) in the `tblCategories` table because in the `tblItemsCategories` table there is a row containing the values 2, 6.

FIGURE 16-8: SAMPLE ITEMS, CATEGORIES, AND ITEM CATEGORIES: A MANY-TO-MANY RELATIONSHIP

tblItems

itemNumber	itemName	itemPurchaseDate	itemPurchasePrice
1	Sofa	1/13/2001	$6,500
2	Stereo	2/10/2003	$1,200
3	Sofa with CD player	5/24/2005	$8,500
4	Table with DVD player	6/24/2005	$12,000
5	Granpa's pocket watch	12/24/1927	$100

tblItemsCategories

itemNumber	categoryId
1	5
2	6
3	5
3	6
4	5
4	6
5	2
5	3

tblCategories

categoryId	categoryName	categoryInsuredAmount
1	Appliance	$30,000
2	Jewelry	$15,000
3	Antique	$10,000
4	Clothing	$25,000
5	Furniture	$5,000
6	Electronics	$2,500
7	Miscellaneous	$5,000

The fancy sofa with the built-in CD player (`itemNumber` 3 in the `tblItems` table) occurs in two rows in the `tblItemsCategories` table, once with a `categoryId` of 5 (Furniture) and once with a `categoryId` of 6 (Electronics). Similarly, the table with the DVD player and Grandpa's pocket watch both belong to multiple categories. It is the `tblItemsCategories` table, then, that allows the establishment of a many-to-many relationship between the two base tables, `tblItems` and `tblCategories`.

UNDERSTANDING ONE-TO-ONE RELATIONSHIPS

In a **one-to-one relationship**, a row in one table corresponds to exactly one row in another table. This type of relationship is easy to understand, but is the least frequently encountered. When one row in a table corresponds to a row in another table, the columns could be combined into a single table. A common reason you create a one-to-one relationship is security. For example, Figure 16-9 shows two tables, `tblEmployees` and `tblSalaries`. Each employee in the `tblEmployees` table has exactly one salary in the `tblSalaries` table. The salaries could have been added to the `tblEmployees` table as an additional column; the salaries are separate only because you want some clerical workers to be allowed to view only names, addresses, and other nonsensitive data, so you give them permission to access only the `tblEmployees` table. Others who work in payroll or administration can create queries that allow them to view joined tables that include the salary information.

FIGURE 16-9: EMPLOYEES AND SALARIES TABLES: A ONE-TO-ONE RELATIONSHIP

tblEmployees

empId	empLast	empFirst	empDept	empHireDate
101	Parker	Laura	3	4/07/1998
102	Walters	David	4	1/19/1999
103	Shannon	Ewa	3	2/28/2003

tblSalaries

empId	empSalary
101	$42,500
102	$28,800
103	$36,000

TIP Another reason to create tables with one-to-one relationships is to avoid lots of empty columns, or **nulls**, if a certain subset of columns is applicable only to specific types of rows in the main table.

TIP You learn more about security issues later in this chapter.

RECOGNIZING POOR TABLE DESIGN

As you create database tables that will hold the data an organization needs, you will encounter many occasions when the table design, or structure, is inadequate to support the needs of the application. In other words, even if a table contains all the attributes required by a specific application, the structural design of the table may make the application cumbersome to use (you will see examples of this later) and prone to data errors.

For example, assume that you have been hired by an Internet-based college to design a database to keep track of its students. After meeting with the college administrators, you determine that you need to know the following information:

- Students' names
- Students' addresses
- Students' cities
- Students' states
- Students' zip codes
- ID numbers for classes in which students are enrolled
- Titles for classes in which students are enrolled

TIP Of course, in a real-life example you could probably think of many other data requirements for the college, in addition to those listed here. The number of attributes is small here for simplicity.

Figure 16-10 contains the `Students` table. Assume that because the Internet-based college is new, only three students have already enrolled. Besides the columns you identified as being necessary, notice the addition of the `studentId` attribute. Given the earlier discussions, you probably recognize that this is the best choice to use as a primary key, because many students can have the same names and even the same addresses. Although the table in Figure 16-10 contains a column for each of the data requirements decided upon with the college administration, the table is poorly designed and will create many problems for the users of the database.

FIGURE 16-10: `Students` TABLE BEFORE NORMALIZATION PROCESS

studentId	name	address	city	state	zip	class	classTitle
1	Rodriguez	123 Oak	Schaumburg	IL	60193	CIS101	Computer Literacy
						PHI150	Ethics
						BIO200	Genetics
2	Jones	234 Elm	Wild Rose	WI	54984	CHM100	Chemistry
						MTH200	Calculus
3	Mason	456 Pine	Dubuque	IA	52004	HIS202	World History

What if a college administrator wanted to view a list of courses offered by the Internet-based college? Can you answer that question by reviewing the table? Well, you can see six courses listed for the three students, so you can assume that at least six courses are offered. But, is it possible that there is also a Psychology course, or a class whose code is CIS102? You can't determine this from the table because no students have enrolled in those classes. Wouldn't it be nice to know all the classes that are offered by your institution, regardless of whether any students have enrolled in them?

Consider another potential problem: What if student Mason withdraws from the school, and, therefore, his row is deleted from the table? You would lose some valuable information that really has nothing to do specifically with student

Mason, but that is very important for running the college. For instance, if Mason's row is deleted from the table, you no longer know, from the remaining data in the table, whether the college offers any History classes, because Mason was the only student enrolled in the HIS202 class.

Why is it so important to discuss the deficiencies of the existing table structure? You have probably heard the saying, "Pay me now or pay me later." This is especially true as it relates to table design. If you do not take the time to ensure well-designed table structures when you are initially designing your database, then you (or the users of your database) will surely spend lots of time later fixing data errors, typing the same information multiple times, and being frustrated by the inability to cull important subsets of information from the database. If you were really hired to create this database and this table structure was your solution to the college's needs, then it is unlikely you would be hired for future database projects.

UNDERSTANDING ANOMALIES, NORMAL FORMS, AND THE NORMALIZATION PROCESS

Database management programs can maintain all the relationships you need. As you add records to, delete records from, and modify records within your database tables, the software keeps track of all the relationships you have established, so that you can view any needed joins any time you want. The software, however, can only maintain useful relationships if you have planned ahead to create a set of tables that supports all the applications you will need. The process of designing and creating a set of database tables that satisfies the users' needs and avoids many potential problems is **normalization**.

The normalization process helps you reduce data redundancies and anomalies. **Data redundancy** is the unnecessary repetition of data. An **anomaly** is an irregularity in a database's design that causes problems and inconveniences. Three common types of anomalies are:

- Update anomalies
- Delete anomalies
- Insert anomalies

If you look ahead to the college database table in Figure 16-11, you will see an example of an **update anomaly**, or a problem that occurs when the data in a table needs to be altered. Because the table contains redundant data, if student Rodriguez moves to a new residence, you have to change the values stored as address, city, state, and zip in more than one location. Of course, this table example is small; imagine if additional data were stored about Rodriguez, such as birth date, e-mail address, major field of study, and previous schools attended.

The database table in Figure 16-10 contains a **delete anomaly**, or a problem that occurs when a row is deleted. If student Jones withdraws from the college, and his entries are deleted from the table, important data regarding the classes CHM100 and MTH200 are lost.

With an **insert anomaly**, problems occur when new rows are added to a table. In the table in Figure 16-10, if a new student named Ramone has enrolled in the college, but has not yet registered for any specific classes, then you can't insert a complete row for student Ramone; the only way to do so would be to "invent" at least one phony class for him.

It would certainly be valuable to the college to be able to maintain data on all enrolled students, regardless of whether those students have registered for specific classes—for example, the college might want to send catalogs and registration information to these students.

TIP In some databases, you might be able to enter an incomplete row for a student.

When you normalize a database table, you walk through a series of steps that allows you to remove redundancies and anomalies. The normalization process involves altering a table so that it satisfies one or more of three **normal forms**, or sets of rules for constructing a well-designed database. The three normal forms are:

- **First normal form**, also known as **1NF**, in which you eliminate repeating groups
- **Second normal form**, also known as **2NF**, in which you eliminate partial key dependencies
- **Third normal form**, also known as **3NF**, in which you eliminate transitive dependencies

Each normal form is structurally better than the one preceding it. In any well-designed database, you almost always want to convert all tables to 3NF.

TIP In a 1970 paper titled "A Relational Model of Data for Large Shared Data Banks," Dr. E.F. Codd listed seven normal forms. For business applications, 3NF is usually sufficient, and so only 1NF through 3NF are discussed in this chapter.

FIRST NORMAL FORM

A table that contains repeating groups is **unnormalized**. A **repeating group** is a subset of rows in a database table that all depend on the same key. A table in 1NF contains no repeating groups of data.

The table in Figure 16-10 violates this 1NF rule. The `class` and `classTitle` attributes repeat multiple times for some of the students. For example, student Rodriguez is taking three classes; her `class` attribute contains a repeating group. To remedy this situation, and to transform the table to 1NF, you simply repeat the rows for each repeating group of data. Figure 16-11 contains the revised table.

FIGURE 16-11: `Students` TABLE IN 1NF

studentId	name	address	city	state	zip	class	classTitle
1	Rodriguez	123 Oak	Schaumburg	IL	60193	CIS101	Computer Literacy
1	Rodriguez	123 Oak	Schaumburg	IL	60193	PHI150	Ethics
1	Rodriguez	123 Oak	Schaumburg	IL	60193	BIO200	Genetics
2	Jones	234 Elm	Wild Rose	WI	54984	CHM100	Chemistry
2	Jones	234 Elm	Wild Rose	WI	54984	MTH200	Calculus
3	Mason	456 Pine	Dubuque	IA	52004	HIS202	World History

The repeating groups have been eliminated from the table in Figure 16-11. However, as you look at the table, you will notice a problem—the primary key, `studentId`, is no longer unique for each row in the table. For example, the table

in Figure 16-11 now contains three rows in which `studentId` equals 1. You can fix this problem, and create a primary key, by simply adding the `class` attribute to the primary key, creating a compound key. (Other problems still exist, as you will see later in this chapter.) The table's key then becomes a combination of `studentId` and `class`. By knowing the `studentId` *and* `class`, you can identify one, and only one, row in the table—for example, a combination of `studentId` 1 and `class` BIO200 identifies a single row. Using the notation discussed earlier in this chapter, the table in Figure 16-11 can be described as:

```
tblStudents(studentId, name, address, city, state, zip, class, classTitle)
```

Both the `studentId` and `class` attributes are underlined, showing that they are both part of the key.

TIP When you combine two columns to create a compound key, you are **concatenating the columns**.

The table in Figure 16-11 is now in 1NF because there are no repeating groups and the primary key attributes are defined. Satisfying the "no repeating groups" condition is also called making the columns **atomic attributes**; that is, making them as small as possible, containing an undividable piece of data. In 1NF, all values for an intersection of a row and column must be atomic. Recall the table in Figure 16-10 in which the class attribute for `studentId` 1 (Rodriguez) contained three entries: CIS101, PHI150, and BIO200. This violated the 1NF atomicity rule because these three classes represented a set of values rather than one specific value. The table in Figure 16-11 does not repeat this problem because, for each row in the table, the class attribute contains one and only one value. The same is true for the other attributes that were part of the repeating group.

TIP Database developers also refer to operations or transactions as **atomic transactions** when they appear to execute completely or not at all.

Now, think back to the earlier discussion about why we want to normalize tables in the first place. Look at Figure 16-11. Are there still redundancies? Are there still anomalies? Yes to both questions. Recall that you want to have your tables in 3NF before actually defining them to the database. Currently, the table in Figure 16-11 is only in 1NF.

In Figure 16-11, notice that Student 1, Rodriguez, is taking three classes. If you were the college employee who was responsible for typing the data into this table, would you want to type this student's name, address, city, state, and zip code for each of the three classes Rodriguez is taking? It is very probable that you may, for one of her classes, type her name as "Rodrigues" instead of "Rodriguez." Or, you might misspell the city of "Schaumburg" as "Schamburg" for one of Rodriguez's classes. A college administrator looking at the table might not know whether Rodriguez's correct city of residence is Schaumburg or Schamburg. If you queried the database to select or count the number of classes being taken by students residing in "Schaumburg," one of Rodriguez's classes would be missed.

TIP Misspelling the student name "Rodriguez" is an example of a data integrity error. You learn more about this type of error later in this chapter.

Consider the student Jones, who is taking two classes. If Jones changes his residence, how many times will you need to retype his new address, state, city, and zip code? What if Jones is taking six classes?

SECOND NORMAL FORM

To improve the design of the table and bring the table in Figure 16-11 to 2NF, you need to eliminate all **partial key dependencies**; that is, no column should depend on only part of the key. Restated, this means that for a table to be in 2NF, it must be in 1NF and all non-key attributes must be dependent on the entire primary key.

In the table in Figure 16-11, the key is a combination of `studentId` and `class`. Consider the `name` attribute. Does the `name` "Rodriguez" depend on the entire primary key? In other words, do you need to know that the `studentId` is 1 *and* that the `class` is CIS101 to determine that the `name` is "Rodriguez"? No, it is sufficient to know that the `studentId` is 1 to know that the `name` is "Rodriguez." Therefore, the `name` attribute is only partially dependent on the primary key, and so the table violates 2NF. The same is true for the other attributes of `address`, `city`, `state`, and `zip`. If you know, for example, that `studentId` is 3, then you also know that the student's `city` is "Dubuque"; you do not need to know any class codes.

Similarly, examine the `classTitle` attribute in the first row in the table in Figure 16-11. This attribute has a value of "Computer Literacy". In this case, you do not need to know both the `studentId` and the `class` to predict the `classTitle` "Computer Literacy". Rather, just the `class` attribute, which is only part of the compound key, is required. Looked at in another way, class "PHI150" will always have the associated `classTitle` "Ethics", regardless of the particular students who are taking that class. So, `classTitle` represents a partial key dependency.

You bring a table into 2NF by eliminating the partial key dependencies. To accomplish this, you create multiple tables so that each non-key attribute of each table is dependent on the *entire* primary key for the specific table within which the attribute occurs. If the resulting tables are still in 1NF and there are no partial key dependencies, then those tables will also be in 2NF.

Figure 16-12 contains three tables: `tblStudents`, `tblClasses`, and `tblStudentClasses`. To create the `tblStudents` table, you simply take those attributes from the original table that depend on the `studentId` attribute, and group them into a new table; name, address, city, state, and zip code all can be determined by the `studentId` alone. The primary key to the `tblStudents` table is `studentId`. Similarly, you can create the `tblClasses` table by simply grouping the attributes from the 1NF table that depend on the `class` attribute. In this application, only one attribute from the original table, the `classTitle` attribute, depends on the `class` attribute. The first two Figure 16-12 tables can be notated as:

```
tblStudents(studentId, name, address, city, state, zip)
tblClasses(class, classTitle)
```

FIGURE 16-12: `Students` TABLE IN 2NF

tblStudents

studentId	name	address	city	state	zip
1	Rodriguez	123 Oak	Schaumburg	IL	60193
2	Jones	234 Elm	Wild Rose	WI	54984
3	Mason	456 Pine	Dubuque	IA	52004

tblClasses

class	classTitle
CIS101	Computer Literacy
PHI150	Ethics
BIO200	Genetics
CHM100	Chemistry
MTH200	Calculus
HIS202	World History

tblStudentClasses

studentId	class
1	CIS101
1	PHI150
1	BIO200
2	CHM100
2	MTH200
3	HIS202

The `tblStudents` and `tblClasses` tables contain all the attributes from the original table. Remember the prior redundancies and anomalies. Several improvements have occurred:

- You have eliminated the update anomalies. The name "Rodriguez" occurs just once in the `tblStudents` table. The same is true for Rodriguez's address, city, state, and zip code. The original table contained three rows for student Rodriguez. By eliminating the redundancies, you have fewer anomalies. If Rodriguez changes her residence, you only need to update one row in the `tblStudents` table.
- You have eliminated the insert anomalies. With the new configuration, you can insert a complete row into the `tblStudents` table even if the student has not yet enrolled in any classes. Similarly, you can add a complete row for a new class offering to the `tblClasses` table even though no students are currently taking the class.
- You have eliminated the delete anomalies. Recall from the original table that student Mason was the only student taking HIS202. This caused a delete anomaly because the HIS202 class would disappear if student Mason was removed. Now, if you delete Mason from the `tblStudents` table in Figure 16-12, the HIS202 class remains in the `tblClasses` list.

If you create the first two tables shown in Figure 16-12, you have eliminated many of the problems associated with the original version. However, if you have those two tables alone, you have lost some important information that you originally had while at 1NF—specifically, which students are taking which classes or which classes are being taken by which students. When breaking up a table into multiple tables, you need to consider the type of relationship among the resulting tables—you are designing a *relational* database, after all.

You know that the Internet-based college application requires that you keep track of which students are taking which classes. This implies a relationship between the `tblStudents` and `tblClasses` tables. Your job is to determine what type of relationship exists between the two tables. Recall from earlier in the chapter that the two most common types of relationships are one-to-many and many-to-many. This specific application requires that one specific student can enroll in many different classes, and that one specific class can be taken by many different students. Therefore, there is a many-to-many relationship between the tables `tblStudents` and `tblClasses`.

As you learned in the earlier example of categorizing insured items, you create a many-to-many relationship between two tables by creating a third table that contains the primary keys from the two tables that you want to relate. In this case, you create the `tblStudentClasses` table in Figure 16-12 as:

```
tblStudentClasses(studentId, class)
```

If you examine the rows in the `tblStudentClasses` table, you can see that the student with `studentId` 1, Rodriguez, is enrolled in three classes; `studentId` 2, Jones, is taking two classes; and `studentId` 3, Mason, is enrolled in only one class. Finally, the table requirements for the Internet-based college have been fulfilled.

Or have they? Earlier, you saw the many redundancies and anomalies that were eliminated by structuring the tables into 2NF, and it is certainly true that the 2NF table structures result in a much "better" database than the 1NF structures. But look again at the `tblStudents` table in Figure 16-12. What if, as the college expands, you need to add 50 new students to this table, and all of the new students reside in Schaumburg, IL? If you were the data-entry person, would you want to type the city of "Schaumburg", the state of "IL", and the zip code of "60193" 50 times? This data is redundant, and you can improve the design of the tables to eliminate this redundancy.

THIRD NORMAL FORM

3NF requires that a table be in 2NF and that it have no transitive dependencies. A **transitive dependency** occurs when the value of a non-key attribute determines, or predicts, the value of another non-key attribute. Clearly, the `studentId` attribute of the `tblStudents` table in Figure 16-12 is a determinant—if you know a particular `studentId` value, you can also know that student's `name`, `address`, `city`, `state`, and `zip`. But this is not considered a transitive dependency because the `studentId` attribute is the primary key for the `tblStudents` table, and, after all, the primary key's job is to determine the values of the other attributes in the row.

There is a problem, however, if a non-key attribute determines another non-key attribute. In the Figure 16-12 `tblStudents` table, there are five non-key attributes: `name`, `address`, `city`, `state`, and `zip`.

The name is a non-key attribute. If you know the value of `name` is "Rodriguez", do you also know the one specific address where Rodriguez resides? In other words, is this a transitive dependency? No, it isn't. Even though only one student is named "Rodriguez" now, there may be many more in the future. So, though it may be tempting to consider that the `name` attribute is a determinant of `address`, it isn't. Looked at another way, if your boss said, "Look at the `tblStudents` table and tell me Jones' address," you wouldn't be able to do so if you had 10 students named "Jones".

The `address` attribute is a non-key attribute. Does it predict anything? If you know the value of `address` is "20 N. Main Street", can you, for instance, determine the name of the student who is associated with that address? No, because in the

future, you might have many students who live at "20 N. Main Street," but they might live in different cities, or you might have two students who live at the same address in the same city. Therefore, `address` does not cause a transitive dependency.

Similarly, the `city` and `state` attributes are not keys, but they also are not determinants because knowing their values alone is not sufficient to predict another non-key attribute value. You might argue that if you know a city's name, you know the state, but many states contain cities named, for example, Union or Springfield.

But what about the non-key attribute `zip`? If you know, for example, that the zip code is 60193, can you determine the value of any other non-key attributes? Yes, a zip code of 60193 indicates that the `city` is Schaumburg and the `state` is IL. This is the "culprit" that is causing the redundancies with regard to the `city` and `state` attributes. The attribute `zip` is a determinant because it determines `city` and `state`; therefore, the `tblStudents` table contains a transitive dependency and is not in 3NF.

To convert the `tblStudents` table to 3NF, simply remove the attributes that depend on, or are **functionally dependent** on, the `zip` attribute. For example, if attribute `zip` determines attribute `city`, then attribute `city` is considered to be functionally dependent on attribute `zip`. So, as Figure 16-13 shows, the new `tblStudents` table is defined as:

```
tblStudents(studentId, name, address, zip)
```

FIGURE 16-13: THE COMPLETE `Students` DATABASE

tblStudents

studentId	name	address	zip
1	Rodriguez	123 Oak	60193
2	Jones	234 Elm	54984
3	Mason	456 Pine	52004

tblZips

zip	city	state
60193	Schaumburg	IL
54984	Wild Rose	WI
52004	Dubuque	IA

tblClasses

class	classTitle
CIS101	Computer Literacy
PHI150	Ethics
BIO200	Genetics
CHM100	Chemistry
MTH200	Calculus
HIS202	World History

tblStudentClasses

studentId	class
1	CIS101
1	PHI150
1	BIO200
2	CHM100
2	MTH200
3	HIS202

TIP A functionally dependent relationship is sometimes written using an arrow that extends from the depended-upon attribute to the dependent attribute—for example, `zip` ⟶ `city`.

Figure 16-13 also shows the `tblZips` table, which is defined as:

```
tblZips(zip, city, state)
```

The new tblZips table is related to the tblStudents table by the zip attribute. Using the two tables together, you can determine, for example, that studentId 3, Mason, in the tblStudents table resides in the city of Dubuque and the state of IA, attributes stored in the tblZips table. When you encounter a table with a functional dependence, you almost always can reduce data redundancy by creating two tables, as in Figure 16-13. With the new configuration, a data-entry operator must still type a zip code for each student, but the drudgery of typing and the possibility of introducing data-entry errors in city and state names for each student is eliminated.

Is the students-to-zip-codes relationship a one-to-many relationship, a many-to-many relationship, or a one-to-one relationship? You know that one row in the tblZips table can relate to many rows in the tblStudents table—that is, many students can reside in zip code 60193. However, the opposite is not true—one row in the tblStudents table (a particular student) cannot relate to many rows in the tblZips table, because a particular student can only reside in one zip code. Therefore, there is a one-to-many relationship between the base table, tblZips, and the related table tblStudents. The link to the relationship is the zip attribute, which is a primary key in the tblZips table and a foreign key in the tblStudents table.

This was a lot of work, but it was worth it. The tables are in 3NF, and the redundancies and anomalies that would have contributed to an unwieldy, error-prone, inefficient database design have been eliminated.

Recall that the definition of 3NF is 2NF plus no transitive dependencies. What if you were considering changing the structure of the tblStudents table by adding an attribute to hold the students' Social Security numbers (ssn)? If you know a specific ssn value, you also know a particular student name, address, and so on; in other words, a specific value for ssn determines one and only one row in the tblStudents table. No two students have the same Social Security number (ruling out identity theft, of course). However, studentId is the primary key; ssn is a non-key determinant, which, by definition, seems to violate the requirements of 3NF. However, if you add ssn to the tblStudents table, the table is still in 3NF because a determinant is allowed in 3NF if the determinant is also a candidate key. Recall that a candidate key is an attribute that could qualify as the primary key but has not been used as the primary key. In the example concerning the zip attribute of the tblStudents table (Figure 16-11), zip was a determinant of the city and state attributes. Therefore, the tblStudents table was not in 3NF because many rows in the tblStudents table can have the same value for zip, meaning zip is not a candidate key. The situation with the ssn column is different because ssn could be used as a primary key for the tblStudents table.

TIP

In general, you try to create a database in the highest normal form. However, when data items are stored in multiple tables, it takes longer to access related information than when it is all stored in a single table. So, sometimes, for performance, you might **denormalize** a table, or reduce it to a lower normal form, by placing some repeated information back into the table. Deciding on the best form in which to store a body of data is a sophisticated art.

In summary:

- A table is in first normal form when there are no repeating groups.
- A table is in second normal form if it is in first normal form and no non-key column depends on just part of the primary key.
- A table is in third normal form if it is in second normal form and the only determinants are candidate keys.

TIP

Not every table starts out denormalized. For example, a table might already be in third normal form when you first encounter it. On the other hand, a table might not be normalized, but after you put it in 1NF, you may find that it also satisfies the requirements for 2NF and 3NF.

DATABASE PERFORMANCE AND SECURITY ISSUES

Frequently, a company's database is its most valuable resource. If buildings, equipment, or inventory items are damaged or destroyed, they can be rebuilt or re-created. However, the information contained in a database is often irreplaceable. A company that has spent years building valuable customer profiles cannot re-create them at the drop of a hat; a company that loses billing or shipment information might not simply lose the current orders—it might also lose the affected customers forever as they defect to competitors who can serve them better. Keeping an organization's data secure is often the most economically valuable responsibility in the company.

You can study entire books to learn all the details involved in data security. The major issues include:

- Providing data integrity
- Recovering lost data
- Avoiding concurrent update problems
- Providing authentication and permissions
- Providing encryption

PROVIDING DATA INTEGRITY

Database software provides the means to ensure that data integrity is enforced; a database has **data integrity** when it follows a set of rules that makes the data accurate and consistent. For example, you might indicate that a quantity in an inventory record can never be negative, or that a price can never be higher than a predetermined value. In addition, you can enforce integrity between tables; for example, you might prohibit entering an insurance plan code for an employee if the insurance plan code is not one of the types offered by the organization.

RECOVERING LOST DATA

An organization's data can be destroyed in many ways—legitimate users can make mistakes, hackers or other malicious users can enter invalid data, and hardware problems can wipe out records or entire databases. **Recovery** is the process of returning the database to a correct form that existed before an error occurred.

Periodically making a backup copy of a database and keeping a record of every transaction together provide one of the simplest approaches to recovery. When an error occurs, you can replace the database with an error-free version that was saved at the last backup. Usually, there have also been changes to the database, called transactions, since the last backup; if so, you must then reapply those transactions.

TIP

Many organizations keep a copy of their data off-site (sometimes hundreds or thousands of miles away) so that if a disaster such as a fire or flood destroys data, the remotely stored copy can serve as a backup.

AVOIDING CONCURRENT UPDATE PROBLEMS

Large databases are accessible by many users at a time. The database is stored on a central computer, and users work at terminals in diverse locations. For example, several order takers might be able to update customer and inventory tables concurrently. A **concurrent update problem** occurs when two database users need to make changes to the same record at the same time. Suppose two order processors take a phone order for item number 101 in an inventory file. Each gets a copy of the quantity in stock—for example, 25—loaded into the memory of her terminal. Each accepts her customer's order and subtracts 1 from inventory. Now, in each local terminal, the quantity is 24. One order gets written to the central database, then the other, and the final inventory is 24, not 23 as it should be.

Several approaches can be used to avoid this problem. With one approach, a lock can be placed on one record the moment it is accessed. A **lock** is a mechanism that prevents changes to a database for a period of time. While one order taker makes a change, the other cannot access the record. Potentially, a customer on the phone with the second order taker could be inconvenienced while the first order taker maintains the lock, but the data in the inventory table would remain accurate.

TIP A **persistent lock** is a long-term database lock required when users want to maintain a consistent view of their data while making modifications over a long transaction.

Another approach to preventing the concurrent update problem is to not allow the users to update the original database at all, but to have them store transactions, which then can be applied to the database all at once, or in a **batch**, at a later time—perhaps once or twice a day or after business hours. The problem with this approach is that as soon as the first transaction occurs and until the batch processing takes place, the original database is out of date. For example, if several order takers place orders for the same item, the item might actually be out of stock. However, none of the order takers will realize the item is unavailable because the database will not reflect the orders until it is updated with the current batch of transactions.

PROVIDING AUTHENTICATION AND PERMISSIONS

Most database software can authenticate that those who are attempting to access an organization's data are legitimate users. **Authentication techniques** include storing and verifying passwords or even using physical characteristics, such as fingerprints or voice recognition, before users can view data. When a user is authenticated, the user typically receives authorization to all or part of the database. The **permissions** assigned to a user indicate which parts of the database the user can view, and which parts he or she can change or delete. For example, an order taker might not be allowed to view or update personnel data, whereas a clerk in the personnel office might not be allowed to alter inventory data.

PROVIDING ENCRYPTION

Database software can be used to encrypt data. **Encryption** is the process of coding data into a format that human beings cannot read. If unauthorized users gain access to database files, the data will be in a coded format that is useless to them. Only authorized users see the data in a readable format.

CHAPTER SUMMARY

- A database holds a group of files that an organization needs to support its applications. In a database, the files often are called tables because you can arrange their contents in rows and columns. A value that uniquely identifies a record is called a primary key, a key field, or a key for short. Database management software is a set of programs that allows users to create table descriptions; identify keys; add records to, delete records from, and update records within a table; arrange records so they are sorted by different fields; write questions that select specific records from a table for viewing; write questions that combine information from multiple tables; create reports and forms; and keep data secure by employing sophisticated security measures.

- Creating a useful database requires a lot of planning and analysis. You must decide what data will be stored, how that data will be divided between tables, and how the tables will interrelate.

- In most tables you create for a database, you want to identify a column, or possibly a combination of columns, as the table's key column or field, also called the primary key. The primary key is important because you can configure your software to prevent multiple records from containing the same value in this column, thus avoiding data-entry errors. In addition, you can sort your records in primary key order before displaying or printing them, and you need to use this column when setting up relationships between the table and others that will become part of the same database.

- A shorthand way to describe a table is to use the table name followed by parentheses containing all the field names, with the primary key underlined.

- Entering data into an already created table requires a good deal of time and accurate typing. Depending on the application, the contents of the tables might be entered over the course of many months or years by any number of data-entry personnel. Deleting records from and modifying records within a database table are relatively easy tasks. In most organizations, most of the important data is in a constant state of change.

- Database management software generally allows you to sort a table based on any column, letting you view your data in the way that is most useful to you. After rows are sorted, they usually can be grouped.

- Frequently, you want to cull subsets of data from a table you have created. The questions that cause the database software to extract the appropriate records from a table and specify the fields to be viewed are called queries. Depending on the software you use, you might create a query by filling in blanks, a process called query by example, or by writing statements similar to those in many programming languages. The most common language that database administrators use to access data in their tables is Structured Query Language, or SQL.
- Most database applications require many tables, and these applications also require that the tables be related. The three types of relationships are one-to-many, many-to-many, and one-to-one.
- As you create database tables that will hold the data an organization needs, you will encounter many situations in which the table design, or structure, is inadequate to support the needs of the application.
- Normalization is the process of designing and creating a set of database tables that satisfies the users' needs and avoids many potential problems. The normalization process helps you reduce data redundancies, update anomalies, delete anomalies, and insert anomalies. The normalization process involves altering a table so that it satisfies one or more of three normal forms, or rules, for constructing a well-designed database. The three normal forms are first normal form, also known as 1NF, in which you eliminate repeating groups; second normal form, also known as 2NF, in which you eliminate partial key dependencies; and third normal form, also known as 3NF, in which you eliminate transitive dependencies.
- Frequently, a company's database is its most valuable resource. Major security issues include providing data integrity, recovering lost data, avoiding concurrent update problems, providing authentication and permissions, and providing encryption.

KEY TERMS

A **database** holds a group of files, or tables, that an organization needs to support its applications.

A database **table** contains data in rows and columns.

An **entity** is one record or row in a database table.

An **attribute** is one field or column in a database table.

A **primary key**, or **key** for short, is a field or column that uniquely identifies a record.

A **compound key**, also known as a **composite key**, is a key constructed from multiple columns.

Database management software is a set of programs that allows users to create table descriptions; identify key fields; add records to, delete records from, and update records within a table; arrange records so they are sorted by different fields; write questions that select specific records from a table for viewing; write questions that combine information from multiple tables; create reports and forms; and keep data secure by employing sophisticated security measures.

A **relational database** contains a group of tables from which you can make connections to produce virtual tables.

Immutable means not changing during normal operation.

Candidate keys are columns or attributes that could serve as a primary key in a table.

After you choose a primary key from among candidate keys, the remaining candidate keys become **alternate keys**.

A **query** is a question asked using syntax that the database software can understand. Its purpose is often to display a subset of data.

Query by example is the process of creating a query by filling in blanks.

Structured Query Language, or **SQL**, is a commonly used language for accessing data in database tables.

The **SELECT-FROM-WHERE** SQL statement is the command that selects the fields you want to view from a specific table where one or more conditions are met.

A **view** is a particular way of looking at a database.

A **relationship** is a connection between two tables.

A **join operation**, or a **join**, connects two tables based on the values in a common column.

A **join column** is the column on which two tables are connected.

A **one-to-many relationship** is one in which one row in a table can be related to many rows in another table. It is the most common type of relationship among tables.

The **base table** in a one-to-many relationship is the “one” table.

The **related table** in a one-to-many relationship is the “many” table.

A **non-key attribute** is any column in a table that is not a key.

A **foreign key** is a column that is not a key in a table, but contains an attribute that is a key in a related table.

A **many-to-many relationship** is one in which multiple rows in each of two tables can correspond to multiple rows in the other.

In a **one-to-one relationship**, a row in one table corresponds to exactly one row in another table.

In a database, empty columns are **nulls**.

Normalization is the process of designing and creating a set of database tables that satisfies the users’ needs and avoids redundancies and anomalies.

Data redundancy is the unnecessary repetition of data.

An **anomaly** is an irregularity in a database's design that causes problems and inconveniences.

An **update anomaly** is a problem that occurs when the data in a table needs to be altered; the result is repeated data.

A **delete anomaly** is a problem that occurs when a row in a table is deleted; the result is loss of related data.

An **insert anomaly** is a problem that occurs when new rows are added to a table; the result is incomplete rows.

Normal forms are rules for constructing a well-designed database.

First normal form, also known as **1NF**, is the normalization form in which you eliminate repeating groups.

Second normal form, also known as **2NF**, is the normalization form in which you eliminate partial key dependencies.

Third normal form, also known as **3NF**, is the normalization form in which you eliminate transitive dependencies.

An **unnormalized** table contains repeating groups.

A **repeating group** is a subset of rows in a database table that all depend on the same key.

To **concatenate columns** is to combine columns to produce a compound key.

Atomic attributes or columns are as small as possible so as to contain an undividable piece of data.

Atomic transactions appear to execute completely or not at all.

A **partial key dependency** occurs when a column in a table depends on only part of the table's key.

A **transitive dependency** occurs when the value of a non-key attribute determines, or predicts, the value of another non-key attribute.

An attribute is **functionally dependent** on another if it can be determined by the other attribute.

You might **denormalize** a table, or place it in a lower normal form, by placing some repeated information back into it.

A database has **data integrity** when it follows a set of rules that makes the data accurate and consistent.

Recovery is the process of returning the database to a correct form that existed before an error occurred.

A **concurrent update problem** occurs when two database users need to make changes to the same record at the same time.

A **lock** is a mechanism that prevents changes to a database for a period of time.

A **persistent lock** is a long-term database lock required when users want to maintain a consistent view of their data while making modifications over a long transaction.

A **batch** is a group of transactions applied all at once.

Authentication techniques include storing and verifying passwords or even using physical characteristics, such as fingerprints or voice recognition, before users can view data.

The **permissions** assigned to a user indicate which parts of the database the user can view, and which parts he or she can change or delete.

Encryption is the process of coding data into a format that human beings cannot read.

REVIEW QUESTIONS

1. **A field or column that uniquely identifies a row in a database table is a(n) _____.**
   a. variable
   b. identifier
   c. principal
   d. key

2. **Which of the following is *not* a feature of most database management software?**
   a. sorting records in a table
   b. creating reports
   c. preventing poorly designed tables
   d. relating tables

3. **Before you can enter any data into a database table, you must do all of the following except _____.**
   a. determine the attributes the table will hold
   b. provide names for each attribute
   c. provide data types for each attribute
   d. determine maximum and minimum values for each attribute

4. **Which of the following is the best key for a table containing a landlord's rental properties?**
   a. `numberOfBedrooms`
   b. `amountOfMonthlyRent`
   c. `streetAddress`
   d. `tenantLastName`

5. **A table's notation is:** `tblClients(socialSecNum, lastName, firstName, clientNumber, balanceDue)`. **You know that** _____.
   a. the primary key is `socialSecNum`
   b. the primary key is `clientNumber`
   c. there are four candidate keys
   d. there is at least one numeric attribute

6. **You can extract subsets of data from database tables using a(n)** _____.
   a. query
   b. sort
   c. investigation
   d. subroutine

7. **A database table has the structure** `tblPhoneOrders(orderNum, custName, custPhoneNum, itemOrdered, quantity)`. **Which SQL statement could be used to extract all attributes for orders for item AB3333?**
   a. `SELECT * FROM tblPhoneOrders WHERE itemOrdered = "AB3333"`
   b. `SELECT tblPhoneOrders WHERE itemOrdered = "AB3333"`
   c. `SELECT itemOrdered FROM tblPhoneOrders WHERE = "AB3333"`
   d. Two of these are correct.

8. **Connecting two database tables based on the value of a column (producing a virtual view of a new table) is a** _____ **operation.**
   a. merge
   b. concatenate
   c. join
   d. met

9. **Heartland Medical Clinic maintains a database to keep track of patients. One table can be described as:** `tblPatients(patientId, name, address, primaryPhysicianCode)`. **Another table contains physician codes along with other physician data; it is described as** `tblPhysicians(physicianCode, name, officeNumber, phoneNumber, daysOfWeekInOffice)`. **In this example, the relationship is** _____.
   a. one-to-one
   b. one-to-many
   c. many-to-many
   d. impossible to determine

10. Edgerton Insurance Agency sells life, home, health, and auto insurance policies. The agency maintains a database containing a table that holds customer data—each customer's name, address, and types of policies purchased. For example, customer Michael Robertson holds life and auto policies. Another table contains information on each type of policy the agency sells—coverage limits, term, and so on. In this example, the relationship is _____.

a. one-to-one
b. one-to-many
c. many-to-many
d. impossible to determine

11. Kratz Computer Repair maintains a database that contains a table that holds job information about each repair job the company agrees to perform. The jobs table is described as: `tblJobs(jobId, dateStarted, customerId, technicianId, feeCharged)`. Each job has a unique ID number that serves as a key to this table. The `customerId` and `technicianId` columns in the table each link to other tables where customer information, such as name, address, and phone number, and technician information, such as name, office extension, and hourly rate, are stored. When the `tblJobs` and `tblCustomers` tables are joined, which is the base table?

a. `tblJobs`
b. `tblCustomers`
c. `tblTechnicians`
d. a combination of two tables

12. When a column that is not a key in a table contains an attribute that is a key in a related table, the column is called a _____.

a. foreign key
b. merge column
c. internal key
d. primary column

13. The most common reason to construct a one-to-one relationship between two tables is _____.

a. to save money
b. to save time
c. for security purposes
d. so that neither table is considered "inferior"

14. The process of designing and creating a set of database tables that satisfies the users' needs and avoids many potential problems is _____.

a. purification
b. normalization
c. standardization
d. structuring

15. The unnecessary repetition of data is called data _____.

a. amplification
b. echoing
c. redundancy
d. mining

16. Problems with database design are caused by irregularities known as _____.

a. glitches
b. anomalies
c. bugs
d. abnormalities

17. When you place a table into first normal form, you have eliminated _____.

a. transitive dependencies
b. partial key dependencies
c. repeating groups
d. all of the above

18. When you place a table into third normal form, you have eliminated _____.

a. transitive dependencies
b. partial key dependencies
c. repeating groups
d. all of the above

19. If a table contains no repeating groups, but a column depends on part of the table's key, the table is in _____ normal form.

a. first
b. second
c. third
d. fourth

20. Which of the following is not a database security issue?

a. providing data integrity
b. recovering lost data
c. providing normalization
d. providing encryption

FIND THE BUGS

1. Create tables as needed so the following employee table is in 3NF.

empID	lastName	firstName	dept	floor	supervisor	payRate
123	Henderson	Robert	HR	1	Rollings	11.00
124	Barker	Anne	MKTG	2	Jenkins	23.50
145	Lee	Benjamin	MFG	3	Liu	15.00
157	Davis	Robert	MFG	3	Liu	14.75
178	Nance	Cody	MKTG	2	Jenkins	24.00
184	Rice	Paula	HR	1	Rollings	12.45
189	Lee	Anne	MFG	3	Liu	15.55
243	Saunders	Marcie	MKTG	2	Jenkins	25.75
256	Freize	Michael	MFG	3	Liu	15.00

2. Suppose you have started a collection of old records. You want to store them in a database so you can select records by title, artist, or condition of the recording. Create tables as needed so the following record collection table is in 3NF.

idNum	title	artists	condition
11	Ebony and Ivory	Paul McCartney Stevie Wonder	Good
12	Yesterday	Paul McCartney John Lennon	Excellent
13	Just a Gigolo	Louis Prima	Fair
14	I've Got You Under My Skin	Peggy Lee	Fair
15	I've Got You Under My Skin	Louis Prima Keely Smith	Excellent

EXERCISES

1. **The Lucky Dog Grooming Parlor maintains data about each of its clients in a table named `tblClients`. Attributes include each dog's name, breed, and owner's name, all of which are text attributes. The only numeric attributes are an ID number assigned to each dog and the balance due on services. The table structure is `tblClients(dogID, name, breed, owner, balanceDue)`. Write the SQL statement that would select each of the following:**

   a. names and owners of all Great Danes
   b. owners of all dogs with balance due over $100
   c. all attributes of dogs named "Fluffy"
   d. all attributes of poodles whose balance is no greater than $50

2. **Consider the following table with the structure `tblRecipes(recipeName, timeToPrepare, ingredients)`. If necessary, redesign the table so it satisfies each of the following:**

   a. 1NF
   b. 2NF
   c. 3NF

recipeName	timeToPrepare	ingredients
Baked lasagna	1 hour	1 pound lasagna noodles ½ pound ground beef 16 ounces tomato sauce ½ pound ricotta cheese ½ pound parmesan cheese 1 onion
Fruit salad	10 minutes	1 apple 1 banana 1 bunch grapes 1 pint blueberries
Marinara sauce	30 minutes	16 ounces tomato sauce ¼ pound parmesan cheese 1 onion

3. Consider the following table with the structure `tblFriends(lastName, firstName, address, birthday, phoneNumbers, emailAddresses)`**. If necessary, redesign the table so it satisfies each of the following:**

a. 1NF
b. 2NF
c. 3NF

lastName	firstName	address	birthday	phoneNumbers	emailAddresses
Gordon	Alicia	34 Second St.	3/16	222-4343 349-0012	agordon@mail.com
Washington	Edward	12 Main St.	12/12	222-7121	ewash@mail.com coolguy@earth.com
Davis	Olivia	55 Birch Ave.	10/3	222-9012 333-8788 834-0112	olivia@abc.com

4. **You have created the following table to keep track of your DVD collection. The structure is `tblDVDs(movie, year, stars)`. If necessary, redesign the table so it satisfies each of the following:**

   a. 1NF
   b. 2NF
   c. 3NF

movie	year	stars
Jerry McGuire	1996	Tom Cruise Renee Zellweger
Chicago	2002	Renee Zellweger Catherine Zeta-Jones Richard Gere
Risky Business	1983	Tom Cruise Rebecca DeMornay

5. **The Midtown Ladies Auxiliary is sponsoring a scholarship for local high-school students. They have constructed a table with the structure `tblScholarshipApplicants(appId, lastName, hsAttended, hsAddress, gpa, honors, clubsActivities)`. The `hsAttended` and `hsAddress` attributes represent high school attended and its street address, respectively. The `gpa` attribute is a grade point average. The `honors` attribute holds awards received, and the `clubsActivities` attribute holds the names of clubs and activities in which the student participated. If necessary, redesign the table so it satisfies each of the following:**

   a. 1NF
   b. 2NF
   c. 3NF

appId	lastName	hsAttended	hsAddress	gpa	honors	clubsActivities
1	Wong	Central	1500 Main	3.8	Citizenship award Class officer Soccer MVP	Future teachers Model airplane Newspaper
2	Jefferson	Central	1500 Main	4.0	Valedictorian Citizenship award Homecoming court Football MVP	Pep Yearbook
3	Mitchell	Highland	200 Airport	3.6	Class officer Homecoming court	Pep Future teachers
4	O'Malley	St. Joseph	300 Fourth	4.0	Valedictorian	Pep Chess
5	Abel	Central	1500 Main	3.7	Citizenship award Class officer	Yearbook

6. Assume you want to create a database to store information about your music collection. You want to be able to query the database for each of the following attributes:

- ☐ A particular title (for example, *Tapestry* or Beethoven's Fifth Symphony)
- ☐ Artist (for example, Carole King or the Chicago Symphony Orchestra)
- ☐ Format of the recording (for example, CD or tape)
- ☐ Style of music (for example, rock or classical)
- ☐ Year recorded
- ☐ Year acquired as part of your collection
- ☐ Recording company
- ☐ Address of the recording company

Design the tables you would need so they are all in third normal form. Create at least five sample data records for each table you create.

7. **Design a group of database tables for the St. Charles Riding Academy. The Academy teaches students to ride by starting them on horses that have been ranked as to their manageability, using a numeric score from 1 to 4. The data you need to store includes the following attributes:**

   - ☐ Student's last name
   - ☐ Student's first name
   - ☐ Student's address
   - ☐ Student's age
   - ☐ Student's emergency contact information—name and phone number
   - ☐ Student's riding level—1, 2, 3, or 4
   - ☐ Each horse's name
   - ☐ Horse's age
   - ☐ Horse's color
   - ☐ Horse's manageability level—1, 2, 3, or 4
   - ☐ Horse's veterinarian's name
   - ☐ Horse's veterinarian's phone number

   Design the tables you would need so they are all in third normal form. Create at least five sample data records for each table you create.

DETECTIVE WORK

1. **What is data mining? Is it a good or bad thing?**
2. **How many free databases can you locate on the Web? What types of data do they offer?**
3. **What organization uses the world's most heavily used database system?**

UP FOR DISCUSSION

1. In this chapter, a phone book was mentioned as an example of a database you use frequently. Name some other examples.

2. Suppose you have authority to browse your company's database. The company keeps information on each employee's past jobs, health insurance claims, and any criminal record. Also suppose that there is an employee at the company whom you want to ask out on a date. Should you use the database to obtain information about the person? If so, are there any limits on the data you should use? If not, should you be allowed to pay a private detective to discover similar data?

3. The FBI's National Crime Information Center (NCIC) is a computerized database of criminal justice information (for example, data on criminal histories, fugitives, stolen property, and missing persons). It is available to federal, state, and local law enforcement and other criminal justice agencies 24 hours a day, 365 days a year. It is almost inevitable that such large systems will contain some inaccuracies. Various studies have indicated that perhaps less than half the records in this database are complete, accurate, and unambiguous. Do you approve of this system or object to it? Would you change your mind if there were no inaccuracies? Is there a level of inaccuracy you would find acceptable to realize the benefits such a system provides?

4. What type of data might be useful to a community in the wake of a natural disaster? Who should pay for the expense of gathering, storing, and maintaining this data?