Chapter 14

Using Relational Databases

- 14-11c Review Questions
- 14-11d Exercises

Chapter Introduction

In this chapter, you will learn about:

- Relational database fundamentals
- Creating databases and table descriptions
- Primary keys
- Database structure notation
- Working with records within a table
- Creating queries
- Relationships between tables
- Poor table design
- Anomalies, normal forms, and normalization
- Database performance and security issues

14-1 Understanding Relational Database Fundamentals

In Chapter 7, you learned that when you store data items for use within computer systems, they are often stored in a data hierarchy that is organized as follows:

- **Character** (A letter, number, or special symbol such as *A*, *7*, or *$*.) are the smallest usable units of data—for example, a letter, digit, or punctuation mark is a character. When characters are stored in a computer, they are created from smaller pieces called bits, which represent computer circuitry, but human users seldom care about bits; characters have meaning to users.

- **Fields** (A single data item, such as `lastName`, `streetAddress`, or `annualSalary`.) are formed from groups of characters and represent a piece of information, such as `firstName`, `lastName`, or `socialSecurityNumber`.

- **Records** (are formed from groups of related fields. The fields go together because they represent attributes of an entity, such as an employee, a customer, an inventory item, or a bank account.) are formed from groups of related fields. The fields go together because they represent attributes of some entity, such as an employee, a customer, an inventory item, or a bank account.

- **Files** (A group of records that go together for some logical reason.) are composed of associated records; for example, a file might contain a record for each employee in a company or each account at a bank.

Most organizations store many files that contain the data they need to operate their businesses; for example, businesses often need to maintain files of data about employees, customers, inventory items, and orders. Many organizations use a database to organize and coordinate the information in these files. A **database** (A logical container that holds a group of files, often called tables, that together serve the information needs of an organization.) holds a group of files that an organization needs to support its applications. In a database, the files often are called **tables** (A database file that contains data in rows and columns; also, a term sometimes used by mathematicians to describe a twodimensional array.) because you can arrange their contents in rows and columns. Real-life examples of these tables abound. For example, consider the listings in a telephone book. Each listing might contain four columns, as shown in Figure 14-1—last name, first name, street address, and phone number. 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 14-1

A Telephone Book Table

Arrays (stored in memory) and tables (stored in databases) are similar in that both contain rows and columns. In an array, each element must have the same data type. The same is not true for tables stored in databases.

Sometimes, one record or row is also called an **entity** (One record or row in a database table.) ; however, many definitions of *entity* exist in database texts. For example, some developers refer to a table as an entity. One column (field) can also be called an **attribute** (One field or column in a database table or a characteristic that defines an object as part of a class.) .

Figure 14-1 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. The records are in alphabetical order by last name. Some users might prefer alternate orders. For example, telemarketers or phone company employees might prefer to have records organized in telephone-number order, and door-to-door salespeople might prefer street-address order. Using a computerized database is convenient because the stored data can easily be sorted and displayed to fit each user's needs.

Unless you are using 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 14-1, 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, most computerized database tables need to have a way to identify each record uniquely, even if it means using multiple columns. A value that uniquely identifies a record is called a **primary key** (A field or column that uniquely identifies a record or database object.) , or a **key** (A field or column that uniquely identifies a record.) 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; such a key is a **compound key** (In a database, a key constructed from multiple columns.) , or **composite key** (In a database, a key constructed from multiple columns.) .

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

Computerized database tables frequently contain thousands of records, or rows, and each row might contain entries in dozens of columns. Handling and organizing all the data in an organization's tables requires sophisticated software. **Database management software** (A set of programs that allows users to create and manage data.) 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** (A group of tables from which connections can be made to produce virtual tables.) .
- Create reports for interpreting data, and create forms for viewing and entering data using interactive screens.
- Keep data secure by employing sophisticated security measures.

Each database management software package operates differently; however, with each, you perform the same types of tasks.

Two Truths & A Lie

Understanding Relational Database Fundamentals

1. Files are composed of associated records, and records are composed of fields.

   T F

2. In a database, files often are called tables because you can arrange their contents in rows and columns.

   T F

3. Key fields always are defined as a single table column.

   T F

14-2 Creating Databases and Table Descriptions

Creating a useful database requires 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 naming it 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 a name that begins with the prefix *tbl*—for example, `tblCustomers`. Your databases often become filled with a variety of objects—tables, forms 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 keep track of the objects in your system.

When you save a table description, many database management programs suggest a default, generic table name such as `Table1`. Usually, a more descriptive name is more useful as you create objects.

Before you can enter data into a database table, you must design the table. 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.

At minimum, designing a table requires that you choose columns for it and then provide names and data types for each column. A table description closely resembles the list of variables that you have used with every program throughout this book. For example, assume you are designing a customer database table. Figure 14-2 shows some column names and data types you might use.

Figure 14-2

Customer Table Description

The table description in Figure 14-2 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 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 values (whole number only) and double-precision numbers (which 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.

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

Unassigned variables within computer programs might be empty (containing a null value), or they 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 some professionals, such as Visual Basic programmers.

The table description in Figure 14-2 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 allow multiple-word column names with 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. Also, when you use a single word to label each database column, it is easier to understand whether one column or several are being referenced.

The `customerID` column in Figure 14-2 defined as text. 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 necessary—that is, only if they might be used in arithmetic calculations. The description in Figure 14-2 follows this convention by declaring `customerID` as text.

 Watch the video *Creating Databases*.

Many database management software packages allow you to add a narrative description of each data column to a table. These comments become part of the table, but do not affect the way it 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 range limits for a column—high and low values between which the column contents must fall. In some database programs, the comments fields are called *memos*.

Two Truths & A Lie

Creating Databases and Table Descriptions

1. When you save a table, it is conventional to provide a name that begins with the prefix *table*.

   T F

2. Designing a table involves deciding what columns your table needs, providing names for them, and providing a data type for each column.

   T F

3. Many database table designers prefer single-word column names because they resemble variable names in programs, they can be easily used in programs, and they make it easier to understand whether one column or several are being referenced.

   T F

14-3 Identifying Primary Keys

In most tables you create for a database, you want to identify a column or a combination of columns as the table's key column or field (or 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 14-2, the logical choice for a primary key is the `customerID` column—each record 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 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 state abbreviation in a table that contains sales information for each state in the United States

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 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. Often, keys are numbers. Usually, assigning a number to each row in a table is the simplest and most efficient method of obtaining a useful key. However, a table's key can be a text field, as in the state abbreviation example.

Sometimes, several columns 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** (Columns or attributes that could serve as a primary key in a table.) . After you choose a primary key from candidate keys, the remaining candidate keys become **alternate keys** (In a database, the remaining candidate keys after a primary key is chosen.) . (Many database developers would object to using a Social Security number as a primary key because of privacy issues.)

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 the primary key column when setting up relationships between this table and others that will become part of the same database.
- 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.

In some tables, when no identifying number has been assigned to the rows, a primary key must be constructed from multiple columns, making it a compound key. For example, consider Figure 14-3, 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 14-3

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 14-3, 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”. A primary key should be **immutable** (Not changing during normal operation.) , meaning that a value does not change during normal operation. In other words, in Figure 14-3, “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.”

As an alternative to selecting three columns to create the compound key in Figure 14-3, many database designers would prefer that the college uniquely number every bed on campus and add a new column to the database for the ID number. Many database

designers feel that a primary key should be short to minimize the amount of required storage in all the tables that refer to it.

Even if only one student was named Smith, for example, or only one Psychology major was listed in Figure 14-3, 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.

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

Two Truths & A Lie

Identifying Primary Keys

1. The primary key in a table is the record that has different data in its columns from all other records.

   T F

2. A multicolumn key is needed when no single column in a table contains unique data for a record.

   T F

3. You usually enter database data after all the fields and keys have been determined.

   T F

14-4 Understanding Database Structure Notation

A shorthand way to describe a table is to use the table name followed by parentheses that contain all the field names, with the primary key underlined. (Some database designers insert an asterisk after the key instead of underlining it.) 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 information about data types or range limits on values, it does provide a quick overview of the table's structure. The key does not have to be the first attribute listed in a table reference, but frequently it is.

Two Truths & A Lie

Understanding Database Structure Notation

1. A shorthand way to describe a table is to use the table name followed by parentheses that contain all the field names.

   T F

2. Typically, when you describe a table using database structure notation, the primary key is underlined.

   T F

3. Database structure notation provides information about column names, their data types, and their range limits.

   T F

Chapter 14: Using Relational Databases: 14-5 Working with Records within Tables
Book Title: Programming Logic and Design
Printed By: Ronald Suchy (rsuchy@mayfieldschools.org)

14-5 Working with Records within Tables

The records in most databases are continuously changing. Personnel must frequently add, delete, update, and sort database records.

Entering data into an existing table is not difficult, but it can require a good deal of time and accurate typing. Depending on the application, the contents of the tables might be entered over the course of months or years by many data-entry personnel. Entering data of the wrong type is not allowed by most database applications. In addition, you might set up your table to prevent duplicate data in specific fields, or to prevent data entry outside of specified bounds in certain 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 fields blank.

In some organizations, data values are not entered manually but are scanned from an original source, greatly reducing the chances for error. For example, purchases can be scanned at the point of sale in a retail store, and patient wristbands and medicines can be scanned by healthcare workers in a hospital.

Deleting and modifying records in a database table are also relatively easy tasks. Products are discontinued, customers change addresses, and so on. Keeping data records up to date is a vital part of any database management system.

In many database systems, some “deleted” records are not physically removed. Instead, they are just marked as deleted so that 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.

Database management software generally allows you to sort a table based on any column, letting you view the 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). When sorting records on multiple fields, the software first uses a primary sort—for example, by last name. After all records 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 allows you to create displays in the formats that suit your needs.

Two Truths & A Lie

Working with Records within Tables

1. Depending on the application, the contents of tables in a database system might be entered over the course of months or years by many data-entry personnel.

   T F

2. In most organizations, much of the important data is permanent.

   T F

3. Database management software generally allows you to sort and group data, letting you view the data in the way that is most useful to you.

   T F

14-6 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 view subsets of data from a table you have created. For example, you might want to examine 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 you use to extract the appropriate records from a table and specify the fields to be viewed are called queries; a **query** (A question used to access values in a database; its purpose is often to display a subset of data.) is a question using syntax that the database software can understand.

Depending on the software, you might create a query by filling in blanks (a process called **query by example** (The process of creating a database query by filling in blanks.)) 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** (A commonly used language for accessing data in database tables.) , or **SQL** (A commonly used language for accessing data in database tables.) . The basic form of the SQL statement that retrieves selected records from a table is **SELECT-FROM-WHERE** (An SQL statement that selects fields to view from a table where one or more conditions are met.) . This statement:

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

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 that a customer table named `tblCustomer` contains data about your business customers and that the structure of the table is as follows:

Then, a statement such as the following 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 operators > (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 contained within quotation marks, but numeric values are not.

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

In database management systems, a particular way of looking at a database by selecting specific fields and records, or placing records in a selected order, is sometimes called a **view** (A way of looking at a database.) . The different views provided by database software are virtual; they do not affect the physical organization or contents of the database.

To create a view that contains all fields for each record in a table, you can use the asterisk as a wildcard; a **wildcard** (A symbol that means *any* or *all*.) is a symbol that means "any" or "all." For example, the following statement would select all columns for every customer whose state is "WI", not just specifically named columns:

```
SELECT * FROM tblCustomer WHERE state = "WI"
```

To select all customers from a table, you can omit the `WHERE` clause in a `SELECT-FROM-WHERE` statement. The following statement selects all columns for all customers:

```
SELECT * FROM tblCustomer
```

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, SQL allows you to 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 14-4 shows a database table named `tblInventory` with the following structure:

Figure 14-4

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

Creating Queries

The table contains five records. Figure Figure 14-5 lists several typical SQL `SELECT` statements you might use with `tblInventory` and explains each.

Figure 14-5

Sample SQL Statements and Explanations

Two Truths & A Lie

Creating Queries

1. A query is a question you use to extract appropriate fields and records froma table.

   T F

2. The most common language that database administrators use to access data in their tables is Structured Query Language, or SQL.

   T F

3. The basic form of the SQL command that retrieves selected records from a table is `RETRIEVE-FROM-SELECTION`.

   T F

14-7 Understanding Relationships between Tables

Most database applications require many tables, and require that the tables be related. The connection between two tables is a **relationship** (A connection between two tables in a database.) , and the database containing the relationships is a *relational database*. Connecting two tables based on the values in a common column is called a **join operation** (The operation that connects two tables based on the values in a common column.) , or more simply, a **join** (The operation that connects two tables based on the values in a common column.) ; the column on which they are connected is the **join column** (The column on which two tables are connected in a database.) . The table displayed as the result of the query provides a virtual view—it uses data from each joined table without disrupting the contents of the originals. For example, in Figure 14-6, the `customerNumber` column is the join column that could produce a virtual view when a user asks to see the name of a customer associated with a specific order number. Three types of relationships can exist between tables:

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

- One-to-one

Figure 14-6

Sample Customers and Orders

14-7a Understanding One-To-Many Relationships

In a **one-to-many relationship** (The relationship 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.) , 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:

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. In the sample data in Figure 14-6, customer 215 has placed three orders. One row in the `tblCustomers` table can correspond to, and 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** (The "one" table in a one-to-many relationship in a database.) in this relationship, and the "many" table (`tblOrders`) is the **related table** (The "many" table in a one-to-many relationship in a database.) .

When two tables have a one-to-many relationship, it is 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 **nonkey attribute** (Any column in a database table that is not a key.) . 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 (A column that is not a key in a table but contains an attribute that is a key in a related table.) . When a base table is linked to a related table in a one-to-many relationship, the primary key of the base table is always related to the foreign key in the related table. In the example in Figure 14-6, `customerNumber` in the `tblOrders` table is a foreign key.

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 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.

14-7b Understanding Many-To-Many Relationships

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

Assume that you are creating these tables to keep track of all the items in your household for insurance purposes. You want to store data about your sofa, stereo, refrigerator, and so on. The `tblItems` table contains the item number, name, purchase date, and purchase price of each item. In addition, this table contains the ID number of the category (Appliance, Jewelry, Antique, and so on) to which the item belongs. You need these categories 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 14-7.

Figure 14-7

Sample Items and Categories: A One-To-Many Relationship

The primary key of the `tblItems` table is `itemNumber`, a unique identifying number that you have assigned to each item 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 has a `categoryName` of "Jewelry" and a `categoryInsuredAmount` of $15,000. The primary key for the `tblCategories` table is `categoryId`, a uniquely assigned value for each property category.

The two tables in Figure 14-7 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 `tblCategories` is the base table (the "one" table) because one category can describe many items that you own. Therefore, `tblItems` is the related table (the "many" table); that is, many items fall into each category. The two tables are linked with the category ID attribute, which is the primary key in the base table (`tblCategories`) and a foreign key in the related table (`tblItems`).

In Figure 14-7, one row in the `tblCategories` table relates to multiple items you own. The opposite is not true—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 refrigerator, or a diamond ring that is an antique?

The structure of the tables shown in Figure 14-7 and the relationship between those tables are designed to keep track of possessions for insurance purposes. If you needed help

categorizing your sofa with a built-in DVD player, you might call your insurance agent. If the agent says that the item is considered a piece of furniture for insurance purposes, then the existing table structures and relationships are adequate. If the agent says the sofa is considered a special type of hybrid item that has a specific maximum insured amount, you could 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 you could associate the DVD-sofa with the new category using the foreign key in the `tblItems` table.

However, if your insurance agent didn't know whether to categorize the sofa as furniture or electronics, the item would present 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 only one category. When you insert a row into the `tblItems` table to describe the new DVD-sofa, you can assign the Furniture code to the foreign key `itemCategoryId`, or you can assign the Electronics code, but not both.

If you want to assign the new DVD-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** (A relationship in which multiple rows in a database table can correspond to multiple rows in another table.) —one in which multiple rows in each table can correspond to multiple rows in the other. 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 DVD 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 `categoryId`s) 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 `itemCategoryId`s 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:

The `tblCategories` table structure remains the same:

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

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 `itemNumbers`; this would disallow using just the `categoryId` as the primary key. However, combining the two attributes `itemNumber` and `categoryId` results in a unique primary key value for each row of the `tblItemsCategories` table.

A table such as `tblItemsCategories` that contains common fields from multiple tables is known by several terms, including *junction table, bridge table, join table, map table, cross-reference table, linking table, many-to-many resolver*, and *association table*. Junction tables can also contain additional fields.

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 14-8 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 the `tblItemsCategories` table has a row containing the values 2, 6.

Figure 14-8

Sample Items, Categories, and Item Categories: A Many-To-Many Relationship

The fancy sofa with the built-in DVD 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 coffee table with the built-in refrigerator (a piece of furniture that is an appliance) and Grandpa's pocket watch (an antique piece of jewelry) both belong to multiple categories. The `tblItemsCategories` table, then, allows the establishment of a many-to-many relationship between the two base tables, `tblItems` and `tblCategories`.

14-7c Understanding One-To-One Relationships

In a **one-to-one relationship** (The relationship in which a row in one table corresponds to exactly one row in another table.) , 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 14-9 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 another column; the salaries are separate because you want some clerks 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 and view joined tables that include the salary information.

Figure 14-9

Employees and Salaries Tables: a One-To-One Relationship

Another reason to create tables with one-to-one relationships is to avoid extensive empty columns, or **nulls** (Empty columns in a database.) , if a subset of columns is applicable only to specific types of rows in the main table.

Two Truths & A Lie

Understanding Relationships Between Tables

1. In a one-to-many relationship, one row in a table can be related to many rows in another table; this is the most common type of relationship between tables.

   T F

2. In a many-to-many relationship, multiple rows in a table each correspond to a single row in many different tables.

   T F

3. 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.

   T F

14-8 Recognizing Poor Table Design

As you create database tables to hold the data used by an organization, you will often find the table design, or structure, 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 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

In a real-life example, you could think of many other data requirements for the college. The number of attributes is small in this example for simplicity.

Figure 14-10 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 for a primary key, because many students can have the same names and even the same addresses. Although the table in Figure 14-10 contains a column for each data requirement from the preceding list, the table is poorly designed and will create many problems.

Figure 14-10

`Students` Table Before Normalization

What if a college administrator wanted to view a list of courses offered by the Internet-based college? You can see six courses listed for the three students, so you can assume that at least six courses are offered. But, could there also be a Psychology course, or a class whose code is CIS102? You can't tell from the table because no students have enrolled in those classes. It would be good to know all the classes 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 his row is deleted from the table? You would lose some valuable information that has nothing to do with student Mason, but is important for running the college. For instance, if Mason's row is deleted, 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 a class with the HIS prefix (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 for table design. If you do not take the time to ensure well-designed table structures during the initial database design, users will spend plenty 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 had created this table structure as a solution to the college's needs, you probably would not be hired for future database projects.

Two Truths & A Lie

Recognizing Poor Table Design

1. The structural design of a table is excellent when the table contains all the attributes required by a specific application.

   T F

2. In a poorly designed database, you might risk losing important data when specific records are deleted.

   T F

3. If you do not take the time to ensure well-designed table structures

during the initial database design, users will spend plenty 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.

T F

Chapter 14: Using Relational Databases: 14-9 Understanding Anomalies, Normal Forms, and Normalization
Book Title: Programming Logic and Design
Printed By: Ronald Suchy (rsuchy@mayfieldschools.org)

14-9 Understanding Anomalies, Normal Forms, and Normalization

Database management programs can maintain all the relationships you need. As you add, delete, 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 database tables that satisfies the users' needs, supports all the applications you will need, and avoids potential problems. This process is called **normalization** (The process of designing and creating a set of database tables that satisfies the users' needs and avoids redundancies and anomalies.) .

The normalization process helps you reduce data redundancies and anomalies. **Data redundancy** (The unnecessary repetition of data.) is the unnecessary repetition of data. An **anomaly** (An irregularity in a database's design that causes problems and inconveniences.) 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 14-11, you will see an example of an **update anomaly** (A problem that occurs when the data in a database table needs to be altered; the result is repeated data.) , 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 address, city, state, and zip values 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 14-10 contains a **delete anomaly** (A problem that occurs when a row is deleted from a database table; the result is loss of related data.) , 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** (A problem that occurs in a database when new rows are added to a table; the result is incomplete rows.) , problems occur when new rows are added to a table. In the table in Figure 14-10, if a new student named Reed has enrolled in the college but has not yet registered for specific classes, then you can't insert a complete row for student Reed; the only way to do so would be to "invent" at least one phony class for him. (Some database software allows incomplete rows.) It would be valuable to the college to be able to maintain data on all enrolled students, regardless of whether they have registered for specific classes—for example, the college might want to send catalogs and registration information to these students.

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

- **First normal form** (The normalization form in which repeating groups are eliminated from a database.) , also known as **1NF** (The normalization form in which repeating groups are eliminated from a database.) , in which you eliminate repeating groups
- **Second normal form** (The normalization form in which partial key dependencies are eliminated from a database.) , or **2NF** (The normalization form in which partial key dependencies are eliminated from a database.) , in which you eliminate partial key dependencies
- **Third normal form** (The normalization form in which transitive dependencies are eliminated from a database.) , or **3NF** (The normalization form in which transitive dependencies are eliminated from a database.) , 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.

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.

14-9a First Normal Form

A table that contains repeating groups is **unnormalized** (Describes a database table that contains repeating groups.) . A **repeating group** (A subset of rows in a database table that all depend on the same key.) 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 14-10 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 14-11 contains the revised table.

Figure 14-11

Students Table in 1NF

The repeating groups have been eliminated from the table in Figure 14-11. However, there is still a problem—the primary key, `studentId`, is no longer unique for each row in the table. For example, the table now contains three rows in which `studentId` equals 1. You can fix this problem and create a primary key simply by 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 14-11 can be described as:

Both the `studentId` and `class` attributes are underlined, showing that they are both part of the key. When you combine two columns to create a compound key, you are **concatenating the columns** (To concatenate columns is to combine columns to produce a compound key.) .

The table in Figure 14-11 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** (In a database, describes columns that are as

small as possible so that they contain undividable pieces of data.) —making them as small as possible to contain 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 14-10, 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 14-11 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.

Database developers also refer to series of operations or transactions as **atomic transactions** (A series of transactions that execute completely or not at all, avoiding partial completion of a task.) when they execute completely or not at all. Making actions atomic guarantees that no actions will be only partially completed, which could cause more problems than if the tasks were not started at all.

Think back to the earlier discussion about why we normalize tables in the first place. Does Figure 14-11 still have redundancies? Are there still anomalies? The answer is *Yes* to both questions. Recall that you want to have the tables in 3NF before actually defining them to the database. Currently, the table in Figure 14-11 is only in 1NF.

In Figure 14-11, notice that Student 1, Rodriguez, is taking three classes. If you were responsible for typing 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? For one of her classes, you might mistype her name as "Rodrigues." Or, you might misspell the city of Schaumburg as "Schamburg" for one of Rodriguez's classes. A college administrator might look at the table and not know the correct spelling for the name or city, and if the administrator 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.

Misspellings are examples of data integrity errors. 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?

14-9b Second Normal Form

To improve the design of the table in Figure 14-11 and bring the table to 2NF, you need to eliminate all **partial key dependencies** (A partial key dependency occurs when a column in a table depends on only part of the table's key.) ; that is, no column should depend on only part of the key. For a table to be in 2NF, it must be in 1NF and all nonkey attributes must be dependent on the entire primary key.

In the table in Figure 14-11, 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 of the table in Figure 14-11. 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. Also, 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 nonkey 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 14-12 contains three tables: `tblStudents`, `tblClasses`, and `tblStudentClasses`. To create the `tblStudents` table, you simply take the attributes from the original table that depend on the `studentId` attribute and group them into a new table—`name`, `address`, `city`, `state`, and `zip` all can be determined by the `studentId` alone. The primary key to the `tblStudents` table is `studentId`. Similarly, you can create the `tblClasses` table simply by 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 tables in Figure 14-12 can be notated as:

Figure 14-12

`Students` Table in 2NF

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 to the `tblClasses` table for a new class offering even though no students are currently taking the class.

- You have eliminated the delete anomalies. Recall from the original table that Mason was the only student taking HIS202. This caused a delete anomaly because the HIS202 class would disappear if Mason was removed. Now, if you delete Mason from the `tblStudents` table in Figure 14-12, the HIS202 class remains in the `tblClasses` list.

If you create the first two tables shown in Figure 14-12, 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 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, a many-to-many relationship exists 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 14-12 as:

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 the 2NF table structures certainly result in a much better database than the 1NF structures. But look again at the `tblStudents` table in Figure 14-12. Asa the college expands, what if 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.

14-9c Third Normal Form

3NF requires that a table be in 2NF and that it have no transitive dependencies. A **transitive dependency** (A database condition in which the value of a nonkey attribute determines or predicts the value of another nonkey attribute.) occurs when the value of a nonkey attribute determines, or predicts, the value of another nonkey attribute. Clearly, the `studentId` attribute of the `tblStudents` table in Figure 14-12 is a determinant—if you know a particular `studentId` value, you can also know that student's `name`, `address`, `city`, `state`, and `zip`. However, 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.

A problem arises, however, if a nonkey attribute determines another nonkey attribute. The `tblStudents` table in Figure 14-12 has five nonkey attributes: `name`, `address`, `city`, `state`, and `zip`.

The name is a nonkey 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 more in the future. So, though it may be tempting to consider that the `name` attribute is a determinant of `address`, it isn't. If your boss said, "Look at the `tblStudents` table and tell me Jones's address," you couldn't if you had 10 students named Jones.

The `address` attribute is a nonkey attribute. Does it predict anything? If you know that the value of `address` is "20 N. Main Street", can you determine which student is associated with that address? No, because in the future, many students might live at 20 N. Main Street, but they might live in different cities, or two students might 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 nonkey attribute value. You might argue that if you know a city's name, you know the state, but many states contain cities named Union or Springfield, for example.

What about the nonkey attribute `zip`? If you know that the zip code is 60193, can you determine the value of any other nonkey 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 in 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 are determined by, or are **functionally dependent** (describes an attribute's relationship to another if it can be determined by the other attribute.) 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 14-13 shows, the new `tblStudents` table is defined as:

Figure 14-13

The Complete `Students` Database

A functionally dependent relationship is sometimes written using an arrow that extends from the dependedupon attribute to the dependent attribute—for example, `zip` → `city`.

Figure 14-13 also shows the `tblZips` table, which is defined as:

The new `tblZips` table is related to the `tblStudents` table by the `zip` attribute. Using the two tables together, you can determine 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 14-13. With the new configuration, a data-entry operator must still type a zip code for each student, but you have eliminated redundancy and the possibility of introducing data-entry errors in city and state names.

Is the student-to-zip-code relationship a one-to-many, many-to-many, or 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 should have the same Social Security number. However, `studentId` is the primary key; `ssn` is a nonkey 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 (see Figure 14-11), `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 could have the same value for `zip`, meaning `zip` was 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.

 Watch the video *Normalization*.

Although Social Security numbers are often considered unique, many organizations refuse to use them as unique identifiers for several reasons. Millions of people use the same number as another person because of identity theft or mistakes. At one point, more than 5000 people were using the same number from an advertisement by a wallet manufacturer. Also, some people can have multiple numbers in cases of domestic violence or federal witness protection.

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, for performance, you sometimes might **denormalize** (To place a database table in a lower normal form by repeating information.) 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 nonkey 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.

Not every table starts as 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.

Two Truths & A Lie

Understanding Anomalies, Normal Forms, and Normalization

1. Normalization helps you reduce data redundancies and anomalies.

   T F

2. Data redundancy is the unnecessary repetition of data.

   T F

3. First normal form is structurally better than third normal form.

   T F

14-10 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—the affected customers might defect to competitors who can serve them better. Keeping data secure is often a company's most economically crucial responsibility.

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

14-10a Providing Data Integrity

Database software provides the means to ensure that data integrity is enforced; a database has **data integrity** (Describes a database that follows a set of rules to make its data accurate and consistent.) when it follows a set of rules that makes the data accurate and consistent. For example, you might specify 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 code is not one of the types offered by the organization.

14-10b 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** (The process of returning a database to a correct form that existed before an error occurred.) 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

are two 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, changes to the database, called transactions, have occurred since the last backup; if so, you must then reapply those transactions. 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.

14-10c 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 clerks might be able to update customer and inventory tables concurrently. A **concurrent update problem** (A problem that can occur when two database users revise the same record at the same time.) occurs when two database users need to modify the same record at the same time. Suppose that two order clerks take a phone order for item number 101 in an inventory file. Each sees the quantity in stock—for example, 25—on her terminal. Each accepts the 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** (A mechanism that prevents changes to a database for a period of time.) is a mechanism that prevents changes to a database for a period of time. (A long-term lock is called a **persistent lock** (A long-term database lock required when users want to maintain a consistent view of their data while making modifications over a long transaction.) .) While one order clerk makes a change, the other cannot access the record. Potentially, a customer on the phone with the second order clerk could be inconvenienced while the first clerk maintains the lock, but the data in the inventory table would remain accurate.

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

14-10d Providing Authentication and Permissions

Most database software can authenticate that people who try to access an organization's data are legitimate users. **Authentication techniques** (Security techniques that include storing and verifying passwords and using physical characteristics, such as fingerprints or voice recognition, before users can be authorized to view data.) include storing and verifying passwords or even using physical characteristics, such as fingerprints or voice recognition, before users can view data. After being authenticated, the user typically receives authorization to all or part of the database. The **permissions** (Attributes assigned to a user to indicate which parts of a database the user can view, change, or delete.) assigned to a user indicate which parts of the database the user can view, modify, or delete. For example, an order clerk 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.

14-10e Providing Encryption

Database software can be used to encrypt data. **Encryption** (The process of coding data into a format that human beings cannot read.) 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.

Two Truths & A Lie

Database Performance and Security Issues

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

   T F

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

   T F

3. A concurrent update problem occurs when two database users need to modify the same record at the same time.

   T F

14-11 Chapter Review

14-11a 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 and manage data.
- Creating a useful database requires 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 combination of columns as the table's key column or field (or 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. 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 that contain all the field names, with the primary key underlined.
- Entering data into an existing 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 months or years by many data-entry personnel. Deleting and modifying records within a database table are relatively easy tasks. In most organizations, much 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 the data in the way that is most useful to you. After rows are sorted, they usually can be grouped.
- Frequently, you want to view subsets of data from a table you have created. The questions you use to extract the appropriate records from a table and specify the fields to be viewed are called queries. Depending on the software, 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 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 to hold the data an organization needs, you will often find the table design, or structure, 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 potential problems. Normalization helps you reduce data redundancies, update anomalies, delete anomalies, and insert anomalies. Normalization 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 (2NF), in which you eliminate partial key dependencies; and third normal form (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.

Chapter 14: Using Relational Databases: 14-11b Key Terms
Book Title: Programming Logic and Design
Printed By: Ronald Suchy (rsuchy@mayfieldschools.org)

Chapter Review

14-11b Key Terms

Character (A letter, number, or special symbol such as *A*, *7*, or *$*.)

Fields (A single data item, such as `lastName`, `streetAddress`, or `annualSalary`.)

Records (are formed from groups of related fields. The fields go together because they represent attributes of an entity, such as an employee, a customer, an inventory item, or a bank account.)

Files (A group of records that go together for some logical reason.)

database (A logical container that holds a group of files, often called tables, that together serve the information needs of an organization.)

tables (A database file that contains data in rows and columns; also, a term sometimes used by mathematicians to describe a twodimensional array.)

entity (One record or row in a database table.)

attribute (One field or column in a database table or a characteristic that defines an object as part of a class.)

primary key (A field or column that uniquely identifies a record or database object.)

key (A field or column that uniquely identifies a record.)

compound key (In a database, a key constructed from multiple columns.)

composite key (In a database, a key constructed from multiple columns.)

Database management software (A set of programs that allows users to create and manage data.)

relational database (A group of tables from which connections can be made to produce virtual tables.)

candidate keys (Columns or attributes that could serve as a primary key in a table.)

alternate keys (In a database, the remaining candidate keys after a primary key is chosen.)

immutable (Not changing during normal operation.)

query (A question used to access values in a database; its purpose is often to display a subset of data.)

query by example (The process of creating a database query by filling in blanks.)

Structured Query Language (A commonly used language for accessing data in database tables.)

SQL (A commonly used language for accessing data in database tables.)

SELECT-FROM-WHERE (An SQL statement that selects fields to view from a table where one or more conditions are met.)

view (A way of looking at a database.)

wildcard (A symbol that means *any* or *all*.)

relationship (A connection between two tables in a database.)

join operation (The operation that connects two tables based on the values in a common column.)

join (The operation that connects two tables based on the values in a common column.)

join column (The column on which two tables are connected in a database.)

one-to-many relationship (The relationship 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.)

base table (The "one" table in a one-to-many relationship in a database.)

related table (The "many" table in a one-to-many relationship in a database.)

nonkey attribute (Any column in a database table that is not a key.)

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

many-to-many relationship (A relationship in which multiple rows in a database table can correspond to multiple rows in another table.)

one-to-one relationship (The relationship in which a row in one table corresponds to exactly one row in another table.)

nulls (Empty columns in a database.)

normalization (The process of designing and creating a set of database tables that satisfies the users' needs and avoids redundancies and anomalies.)

Data redundancy (The unnecessary repetition of data.)

anomaly (An irregularity in a database's design that causes problems and inconveniences.)

update anomaly (A problem that occurs when the data in a database table needs to be altered; the result is repeated data.)

delete anomaly (A problem that occurs when a row is deleted from a database table; the result is loss of related data.)

insert anomaly (A problem that occurs in a database when new rows are added to a table; the result is incomplete rows.)

normal forms (Rules for constructing a well-designed database.)

First normal form (The normalization form in which repeating groups are eliminated from a database.)

1NF (The normalization form in which repeating groups are eliminated from a database.)

Second normal form (The normalization form in which partial key dependencies are eliminated from a database.)

2NF (The normalization form in which partial key dependencies are eliminated from a database.)

Third normal form (The normalization form in which transitive dependencies are eliminated from a database.)

3NF (The normalization form in which transitive dependencies are eliminated from a database.)

unnormalized (Describes a database table that contains repeating groups.)

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

concatenating the columns (To concatenate columns is to combine columns to produce a compound key.)

atomic attributes (In a database, describes columns that are as small as possible so that they contain undividable pieces of data.)

atomic transactions (A series of transactions that execute completely or not at all, avoiding partial completion of a task.)

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

transitive dependency (A database condition in which the value of a nonkey attribute determines or predicts the value of another nonkey attribute.)

Functionally dependent (describes an attribute's relationship to another if it can be determined by the other attribute.)

denormalize (To place a database table in a lower normal form by repeating information.)

data integrity (Describes a database that follows a set of rules to make its data accurate and consistent.)

Recovery (The process of returning a database to a correct form that existed before an error occurred.)

concurrent update problem (A problem that can occur when two database users revise the same record at the same time.)

lock (A mechanism that prevents changes to a database for a period of time.)

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

batch (A group of transactions applied all at once.)

Authentication techniques (Security techniques that include storing and verifying passwords and using physical characteristics, such as fingerprints or voice recognition, before users can be authorized to view data.)

permissions (Attributes assigned to a user to indicate which parts of a database the user

can view, change, or delete.)

Encryption (The process of coding data into a format that human beings cannot read.)

Chapter 14: Using Relational Databases: 14-11c Review Questions
Book Title: Programming Logic and Design
Printed By: Ronald Suchy (rsuchy@mayfieldschools.org)

Chapter Review

14-11c 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 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 the above 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 policy data—each record contains the policy number, the customer's name and address, and the type of policy purchased. For example, customer Michael Robertson is referenced in two records because he 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 of 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 of customer information, such as name, address, and phone number, and technician information, such as name, office extension, and hourly rate. 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 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

Chapter Review

14-11d 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 balances due of more than $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 1NF, 2NF, and 3NF.

3. Consider the following table with the structure `tblFriends(lastName, firstName, address, birthday, phoneNumbers, emailAddresses)`. If necessary, redesign the table so it satisfies 1NF, 2NF, and 3NF.

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 1NF, 2NF, and 3NF.

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 the 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 1NF, 2NF, and 3NF.

6. Assume that 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 MP3 file)
    - 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 collection of database tables for the Springfield Town Council. The council is made up of representatives from each of the town's 15 precincts. The data you need to store includes the following attributes:

   - Precinct number
   - Precinct population
   - Council representative's last name
   - Council representative's first name
   - Council representative's phone number
   - Council representative's political party
   - Political party chairperson's name
   - Political party headquarters address

   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.

Find the Bugs

8. Your downloadable student files for Chapter 14 include DEBUG14-01.doc, DEBUG14-02.doc, and DEBUG14-03.doc. Each file starts with some comments that describe the problem. Following the comments, each file contains a table that is not in 3NF. Create tables as needed to put the data in 3NF.

Game Zone

9. Massively Multiplayer Online Role-Playing Games (MMORPG) are online

computer role-playing games in which a large number of players interact with one another in a virtual world. Players assume the role of a fictional character and control that character's actions. MMORPGs are distinguished from smaller RPGs by the number of players and by the game's persistent world, usually hosted by the game's publisher, which continues to exist and evolve while the player is away from the game. Design the database you would use to host an MMORPG, including at least three tables.

Up for Discussion

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

11. Suppose that 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 you want to ask a coworker 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?

12. The FBI's National Crime Information Center (NCIC) is a computerized database of criminal justice information, including data on criminal histories, fugitives, stolen property, and missing persons. Such large systems almost inevitably contain 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 of such a system?

13. 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?