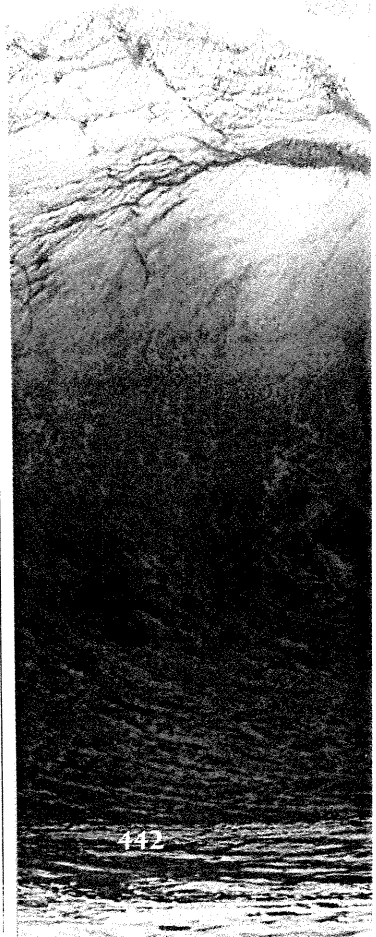
chapter

A Table with a View

Introduction to Database Concepts

Learning Objectives

- Use XML to describe the metadata for a table of information, and classify the uses of the tags as identification, affinity, or collection
- Explain the differences between everyday tables and database tables
- Explain how the concepts of entities and attributes are used to design a database table
- Use the six database operations: `Select`, `Project`, `Union`, `Difference`, `Product`, and `Join`
- Describe the differences between physical and logical databases
- Express a query using Query By Example



Computers are useless. They only give answers.

–PABLO PICASSO

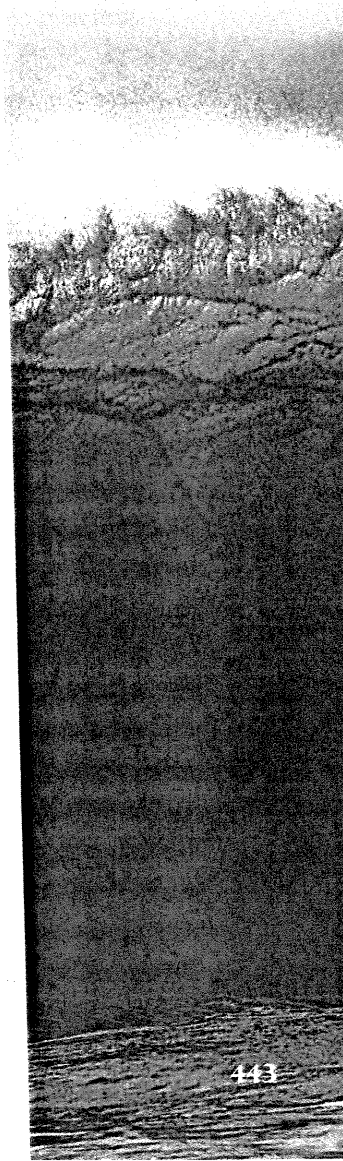
Now that we have all this useful information, it would be nice to do something with it. (Actually, it can be emotionally fulfilling just to get the information. This is usually only true, however, if you have the social life of a kumquat.)

–UNIX PROGRAMMER'S MANUAL

WE HAVE seen the benefits of using spreadsheets to organize lists of information. By arranging similar information into columns and using a separate row for each new list item, we can easily sort data, use formulas to summarize and compute values, get help from the computer to set up series, and so forth. Spreadsheets are very powerful, but with databases it's possible to apply even greater degrees of organization and receive even more help from the computer.

The key idea is to supply metadata describing the properties of the collected information. Recall that metadata is simply information describing (the properties of) other information. We applied the idea of specifying metadata in Chapter 8, when we used tags—the metadata—to describe the content of the *Oxford English Dictionary*, enabling the computer to help us search for words and definitions. Some databases use tags for metadata, others use different kinds of metadata, but the same principles apply: Knowing the structure and properties of the data, the computer can help us retrieve, organize, and manage it.

In this chapter we distinguish between the everyday concept of a table and a relational database table. Next, we explain how to set up the metadata for collections of information to create a database. The principles are straightforward and intuitive. We will make the metadata tangible by using a notation called XML. After introducing basic table concepts, we present the five fundamental operations on tables and the `Join` operation. The concepts of physical database and logical database are connected by the concept of queries, and we illustrate how to build a user's logical view from physical tables. Finally, the convenience of Query By Example is illustrated using simple examples.



Differences Between Tables and Databases

When we think of databases, we often think of tables of information. For example, your iTunes or similar application records the title, artist, running time, and so on in addition to the actual MP3 data (the music). Your favorite song is a row in that table. Another example is your state's database of automobile registrations recording the owner's name and address, the vehicle identification number (VIN), the license plate number, and such. Your car is a row in the registration database table. And as a last example, the U.S. Central Intelligence Agency (CIA) keeps an interesting database called the World Factbook; see `https://www.cia.gov/library/publications/the-world-factbook/index.html`. The demographic table records the country name, population, life expectancy, and so on. The U.S. is a row in the demography table.

Comparing Tables

To see the difference between these database tables and other forms of tables, such as spreadsheets and HTML tables, consider the row for Canada in the CIA's demographic database. This row is displayed as

Canada	**32805041**	**1.61**	**5**	**80.1**

in a table with column headings such as Country, Population, and Birthrate. In the file it is represented as

```
<demogData>
  <country>Canada</country>
  <population>32805041</population>
  <fertility>1.61</fertility>
  <infant>5</infant>
  <lifeExpct>80.1</lifeExpct>
</demogData>
```

where the tags identify the population, fertility or birthrate, infant mortality (per 1,000 live births), and life expectancy. That is, we are shown a row of data as it appears in any other table, but inside the computer it has a tag identifying each of the data fields.

How does this data appear in other table forms? In a spreadsheet, the following is the row for Canada.

36	Cameroon	16988132	4.47	65	50.89
37	Canada	32805041	1.61	5	80.1
38	Cape Verde	418224	3.48	48	70.45

The entries for Canada are the same, but the software knows the values only by position, not by their meaning. So, if a cell is inserted at the beginning, causing all of the data to shift right one position,

36	Cameroon	16988132	4.47	65	50.89	
37	Canada	32805041	1.61	5	80.1	
38	Cape Verde	418224	3.48	48	70.45	

the identity of the information is lost. Spreadsheets rely on position to keep the integrity of their data; the information is not known by its `<country>` tag, but rather as `A37`.

HTML tables are possibly even worse. The usual Web page presentation of the data for Canada is represented in HTML as

```
<tr>
  <td>Canada</td>
  <td>32805041</td>
  <td>1.61</td>
  <td>5</td>
  <td>80.1</td>
</tr>
```

where we recall that `<tr>` is a table row tag and `<td>` is a table data tag. These tags simply identify Canada's data as table entries with no unique identity at all; that is, the same kind of `<td>` tags surround all of the different forms of data. HTML is concerned only with how to display the data, not with its meaning.

The Database's Advantage

The metadata is the key advantage of databases over other systems recording data as tables. Here's why. Suppose we want to know the life expectancy of Canadians. Database software can search for the `<country>` tag surrounding Canada. When it's found, the `<country>` tag will be one of several tags surrounded by `<demogData>` tags. These constitute the entry for Canada in the database. The software can then look for the `<lifeExpct>` tag among those tags and report the data that they surround as the data for Canada. The computer knew which data to return based on the availability of the metadata.

The tags for the CIA database just discussed fulfill two of the most important roles in defining metadata.

- **Identify the type of data:** Each different type of value is given a unique tag.
- **Define the affinity of the data:** Tags enclose all data that is logically related.

The `<country>`, `<population>`, and similar tags have the role of identification because they label the content. The `<demogData>` tag has the role of implementing affinity because it keeps an entry's data together. There are other properties of data that metadata must record, as you will see throughout this chapter, but these are perhaps the most fundamental.

XML: A Language for Metadata Tags

To emphasize the importance of metadata and to prepare for our own applications of database technology, let's take a moment to discuss the basics of XML. XML stands for the Extensible Markup Language, and like the Hypertext Markup

Language (HTML), it is basically a tagging scheme, making it rather intuitive. The tagging scheme used for the *Oxford English Dictionary* (*OED*) in Chapter 8 was a precursor to XML, and the demographic data of the last section was written in XML.

What makes XML easy and intuitive is that there are no standard tags to learn. *We think up the tags we need!* Computer scientists call this a *self-describing language*, because whatever we create becomes the language (tags) to structure the data. There are a couple of rules—for example, always match tags—but basically anything goes. Perhaps XML is the world's easiest-to-learn "foreign" language.

The same people who coordinate the Web—the World Wide Web Consortium (W3C)—developed XML. As a result, it works very well with browsers and other Web-based applications. So, it comes as no surprise that just as HTML must be written with a text editor rather than a word processor to avoid unintentionally including the word processor's tags, we must also write XML in a simple text editor for the same reason. Use the same editor that you used in Chapter 4 to practice writing Web pages: for Mac users that might be TextEdit or TextWrangler; for Windows users it might be Notepad or Notepad++.

fit TIP

Use A Text Editor for XML. Like HTML, XML should be written using a text editor like Notepad++ or TextWrangler rather than a word processor like Word or Word Perfect. Text editors give you only the text you see, but word processors include their own tags and other information that could confuse XML.

An Example from Tahiti

Let's use XML to define tags to specify the metadata for a small data collection. Given the following size data (area in km^2) for Tahiti and its neighboring islands in the Windward Islands archipelago of the South Pacific,

Tahiti	1048
Moorea	130
Maiao	9.5
Mehetia	2.3
Tetiaroa	12.8

we want to add the metadata, that is, identify which data is an island name and which is the area. As usual, the tag and its companion closing tag surround the data. We choose `<iName>` and `<area>` as the tags and write:

```
<iName>Tahiti</iName>      <area>1048</area>
<iName>Moorea</iName>      <area>130</area>
<iName>Maiao</iName>       <area>9.5</area>
<iName>Mehetia</iName>     <area>2.3</area>
<iName>Tetiaroa</iName>    <area>12.8</area>
```

These tags are used in the identification role. Notice that we chose `<iName>` rather than, say, `<island name>`. This is because XML tag names cannot contain spaces. But because both uppercase and lowercase are allowed—XML is case sensitive—

we capitalize the "N" to make the tag more readable. All XML rules are shown later in this section in Table 16.1.

Though we have labeled each item with a tag describing what it is, we're not done describing the data. We need tags describing what sort of thing the name specifies and the area measures. That's an island, of course. So we enclose each entry with an `<island>` tag, as in

```
<island><iName>Tahiti</iName>   <area>1048</area></island>
<island><iName>Moorea</iName>   <area>130</area></island>
<island><iName>Maiao</iName>    <area>9.5</area></island>
<island><iName>Mehetia</iName>  <area>2.3</area></island>
<island><iName>Tetiaroa</iName> <area>12.8</area></island>
```

The `<island>` tag serves in the affinity role to keep the two facts together; that is, Tahiti is grouped with its area and it is separated from Moorea and its area.

We're nearly done. The islands are not just randomly dispersed around the ocean. They are part of an archipelago, the proper name for a group of islands. So, we naturally invent one more tag, `<archipelago>`, and surround all of the islands with it. The result is shown in Figure 16.1.

```
<?xml version = "1.0" encoding="ISO-8859-1" ?>
<archipelago>
<island><iName>Tahiti</iName>   <area>1048</area></island>
<island><iName>Moorea</iName>   <area>130</area></island>
<island><iName>Maiao</iName>    <area>9.5</area></island>
<island><iName>Mehetia</iName>  <area>2.3</area></island>
<island><iName>Tetiaroa</iName> <area>12.8</area></island>
</archipelago>
```

Figure 16.1 XML file encoding data for the Windward Islands database. The first line states that the file contains XML tags.

Notice that in Figure 16.1 an additional line has been added at the beginning of the file. This line, which uses the unusual form of associating question marks (?) within the brackets, identifies the file as containing XML data representations. (It also states that the file's characters are the standard ASCII set used in the U.S.; see Chapter 8.) This first line is required and must be the first line of any XML file. By identifying the file as XML, hundreds of software applications can understand what it contains. In this way the effort to tag all of the information can be repaid by using the data with those applications.

fit TIP

Start Off Right with XML. XML files must be identified as such, and so they are required to begin with the text

```
<?xml version = "1.0" encoding="ISO-8859-1" ?>
```

(or other encoding) as their first line and without leading spaces. The file should be ASCII text, and the file extension should be `.xml`.

try it Write an XML metadata coding for the following collection of data from the Galápagos archipelago.

Island	Area	Elevation
Isabela	4588	1707
Fernandina	642	1494
Tower	14	76
Santa Cruz	986	846

For the items of the same type as the data from the Windward archipelago, use the same tags; for the elevation, the highest point on the island, think up a new tag.

Answer: Using a tag name different from `<elev>` for the elevation is possible, but otherwise this is the one solution apart from spacing

```
<archipelago>
    <island> <iName>Isabela</iName>
        <area>4588</area><elev>1707</elev> </island>
   <island> <iName>Fernandina</iName>
        <area>642</area> <elev>1494</elev> </island>
   <island> <iName>Tower</iName>
        <area>14</area>  <elev>76</elev>    </island>
   <island> <iName>Santa Cruz</iName>
        <area>986</area> <elev>846</elev>  </island>
</archipelago>
```

Expanding the Use of XML

Given the XML encoding of two archipelagos—the Windward Islands and the Galápagos Islands—it seems reasonable to combine the encodings.

To create a database of the two archipelagos, we place them in a file, one after the other. This might seem odd because the Windward Islands have only two data values—name and area—while the Galápagos Islands have three—name, area, and elevation. But this is okay. Both archipelago encodings use the same tags for the common information, which is the key issue to consider when combining them. Extra data is allowed and, in fact, we might want to gather the elevation data for the Windward Islands.

With the two archipelagos combined into one database, we want to include the name of each to tell them apart easily. Of course, this means adding another tag for the name. We could use `<name>`, which is different from the `<iName>` tag used before. But in the same way that we added "`i`" to remind ourselves that it is an island name, it is probably wise to use the same idea to create a more specific tag name. Let's adopt the tag `<a_name>`. Notice the use of underscore, which is an allowed punctuation symbol for XML. We will place the name inside the `<archipelago>` tag, since it is data about the archipelago.

Finally, we have two archipelagos and we need to group them together by surrounding them with tags; these tags will serve as the root element of our XML database. A **root element** is the tag that encloses all of the content of the XML file. In Figure 16.1 the `<archipelago>` tag was the root element, but now with two archipelagos in the file, we need a new tag to enclose them. They are both geographic features of our planet, so we will use `<geo_feature>` as the tag that surrounds both archipelagos. The final result of our revisions is shown in Figure 16.2.

Notice that the text in the file has been indented to make it more readable. Like HTML, XML doesn't care about white space—spaces, tabs, and new lines—when they are between tags. This allows us to format XML files to simplify working with them, but the indenting is only for our use.

Attributes in XML

Recall that HTML tags can have attributes to give additional information, such as `bgcolor` in `<body bgcolor="blue">`. Our invented tags of XML can also have

```
<?xml version = "1.0"
      encoding="ISO-8859-1" ?>
<geo_feature>
   <archipelago>
      <a_name>Windward Islands
      </a_name>
      <island>
         <iName>Tahiti</iName>
         <area>1048</area>
      </island>
      <island>
         <iName>Moorea</iName>
         <area>130</area>
      </island>
      <island>
         <iName>Maiao</iName>
         <area>9.5</area>
      </island>
      <island>
         <iName>Mehetia</iName>
         <area>2.3</area>
      </island>
      <island>
         <iName>Tetiaroa</iName>
         <area>12.8</area>
      </island>
   </archipelago>
   <archipelago>
      <a_name>Galapagos Islands
      </a_name>
      <island>
         <iName>Isabella</iName>
         <area>4588</area>
         <elevation>1707</elevation>
      </island>
      <island>
         <iName>Fernandina</iName>
         <area>642</area>
         <elevation>1494</elevation>
      </island>
      <island>
         <iName>Tower</iName>
         <area>14</area>
         <elevation>76</elevation>
      </island>
      <island>
         <iName>Santa Cruz</iName>
         <area>986</area>
         <elevation>846</elevation>
      </island>
   </archipelago>
</geo_feature>
```

Figure 16.2 XML file for the Geographic Features database. XML ignores white space, so the text in the file has been indented for easier reading.

attributes. They have a similar form, and must always be set inside the simple quotation marks—that is, the straight quotes, not the curly "smart" quotes. Tag attribute values can be enclosed either in paired single or double quotes. If the content of the tag attribute requires quotes or an apostrophe (the single quote), then enclose the attribute value in the other form of quotes. So, we might have

```
<entry warnIfNone="Ain't there!">The user entered this
data.</entry>
```

for one instance, and

```
<entry warnIfNone='I say, "Please Enter"'>The data is
from a user.</entry>
```

for another.

Understanding how to write tag attributes is easy enough. Even the rules for using quotes are straightforward. But, we want to use them wisely, which requires some thought.

The best advice about attributes is to use them for additional metadata, not for actual content. So, although we could have written

```
<archipelago name="Galapagos">
```

we chose not to because the name of an archipelago is content. A better use is to give an alternate form of the data, as in

```
<a_name accents="Gal&aacute;pagos">Galapagos</a_name>
```

which records that the second "a" in Galápagos is accented. The name of the islands is still given using a normal tag, but specifying accent marks separately simplifies searching and display options.

Effective Design with XML Tags

XML is a very flexible way to encode metadata. As we have described the archipelagos, we have used a few basic guidelines to decide how to use the tags. To emphasize these rules, let's review our thinking in creating metadata tags for the archipelago data, encapsulating it into three encoding rules.

Identification Rule: Label Data with Tags Consistently. *You can choose whatever tag names you wish to name data, but once you've decided on a tag for a particular kind of data, you must always surround it with that tag.*

Notice that one of the advantages of enclosing data with tags is that it keeps the data together. For example, the island of Santa Cruz in the Galápagos is a two-word name, but we don't have to treat it any differently than the one-word island names since the tags keep the two words together.

You may think that because we can choose our own tag names, it might be difficult to combine databases written by two different people—without planning ahead, they will probably choose different tags. Luckily, such differences are easily resolved: Because the tags are used consistently, it is possible to edit a file using *Find/Replace* to change the tag names. (There are other, more sophisticated ways to

make them consistent, too.) For example, if your friend, who gathered the archipelago data for the Northern Hemisphere, used `<Name>` for the archipelago name rather than `<a_name>`, use *Find* to locate `<Name>` and *Replace* to substitute `<a_name>`. Of course, searching for `Name` alone and replacing it with `a_name` does not work since it would match and ruin the `<iName>` tags. (If such cases do get in the way, use the Placeholder Technique described in Chapter 2.)

Affinity Rule: Group Related Data. *Enclose in a pair of tags all tagged data referring to the same entity. Grouping it keeps it all together, but the idea is much more fundamental: Grouping makes an association of the tagged data items as being related to each other, properties of the same thing.*

We applied this rule when we grouped the island name and area data inside `<island>` tags. We did this because both items referred to the same thing, the island. This is an important association, because the area data is not just area data about some random place on the earth; it is the area data for a specific place that is named Tahiti. This is an extremely important result from the simple act of enclosing data in tags.

When we added elevation data as an additional feature of islands, we included it inside the `<island>` tags for the same reason. As the elevation data shows, it is not necessary for every instance of an object to have data for the same set of characteristics.

Collection Rule: Group Related Instances. *When you have several instances of the same kind of data, enclose them in tags; again, it keeps them together and implies that they are related by being instances of the same type.*

When we had a group of five islands from the same area of the ocean, we grouped them inside an `<archipelago>` tag, and when we had a group of two archipelagos, we grouped them inside a `<geo_feature>` tag. We also added the names to the archipelagos using `<a_name>`, because as a collection they also have this additional property that we want to record.

Notice that the Affinity Rule and Collection Rule are different. The Affinity Rule groups together the data for a single thing—an island. Typically, in this case the tags of the data values will all be different reflecting the different properties of the thing. The Collection Rule groups together the data of several instances of the same thing. Typically, in this case the tags—in our case `<islands>`—will be the same. The first association is among properties of an object, the second is among the objects themselves, which we also call *entities*. Notice that being grouped by the Collection Rule doesn't preclude being an object; the islands grouped together form a larger object, the archipelago, and so it has properties, too, such as a name.

The XML Tree

The rules for producing XML encodings of information produce hierarchical descriptions that can be thought of as trees. (We interpreted hierarchies as trees in Chapter 5.) See Figure 16.3 for the tree structure of the encoding of Figure 16.2. The hierarchy is a consequence of how the tags enclose one another and the data.

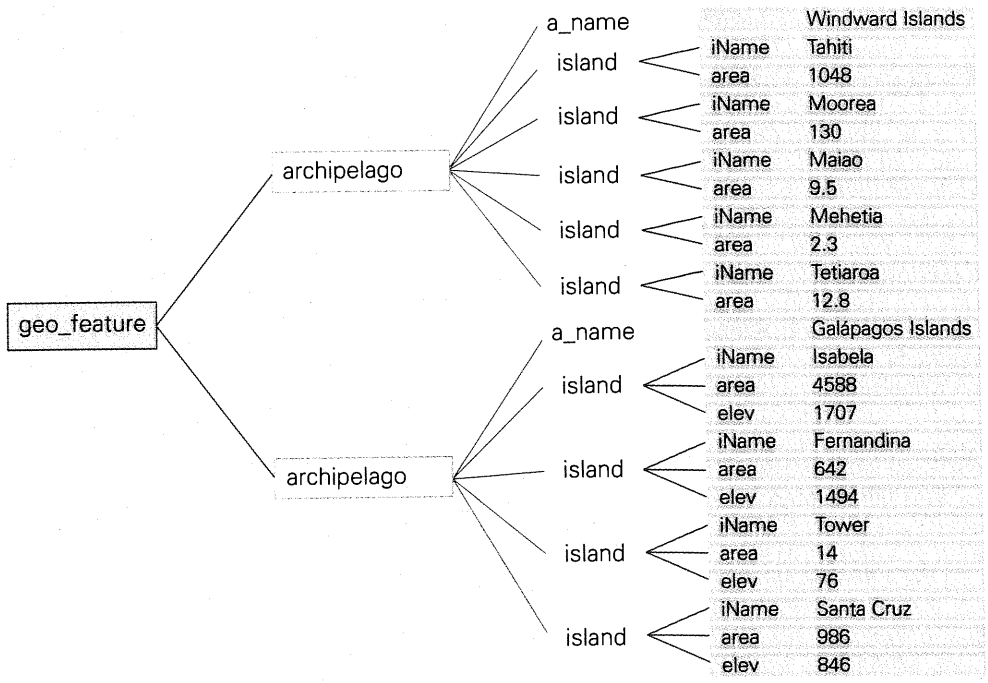


Figure 16.3 The XML displayed as a tree. The encoding from Figure 16.2 is shown with the root element (`geo_feature`) to the left and the leaves (content) shown to the right.

Table 16.1 Rules for writing XML.

Required first line	`<?xml version="1.0" encoding="ISO-8859-1"?>` must appear on the first line, starting in the first position.
First tag	The first tag encountered is the *root* element, and it must enclose all of the file's content; it appears on the second or possibly third line.
Closing tags	All tags must be closed.
Element naming	Observe these rules: • Names can contain letters, numbers, and underscore characters. • Names must not start with a number or punctuation character. • Names must not start with the letters xml (or XML, or Xml, etc.). • Names cannot contain spaces.
Case sensitivity	Tags and attributes are case sensitive.
Proper nesting	All tags must be well-nested.
Attribute quoting	All attribute values must be quoted; paired single quotes (apostrophes) or paired double quotes are okay; use "dumb" quotes only; choose 'opposite' quotes to enclose quoted values.
White space	White space is preserved and converted to a single space.
Comments	XML comments have the form `<!-- This is a comment. -->`.

Tables and Entities

You have seen how you can record metadata about a collection of data values using XML tags. For the moment, let's set aside the topics of tagging and XML, and focus directly on table database systems generally. We want you to understand the concepts of database organization and the desirable properties embodied in the metadata, not simply the way to encode that structure with tags. We'll return to tagging later in the next chapter, but for now, think of tables pure and simple.

The kind of database approach we will discuss is known as a relational database. **Relational databases** describe the relationships among the different kinds of data—the sort of ideas embodied in Affinity and Collection Rules—allowing the software to answer queries about them. Although every relational database can be described by XML, it is not true that anything described by XML is a relational database. It may seem that relational databases are limited, but their power is enormous.

*fit*BYTE

A Bright Idea. Though many people contributed to the creation of relational databases, E. F. Codd of IBM is widely credited with the original concept. He received the Association of Computing Machinery's Turing Award, the field's Nobel Prize, for the idea.

Entities

What *do* we want in database tables? Entities. "Entity" is about as vague as "thing" and "stuff," but the inventors of databases didn't want to limit the kinds of information that can be stored. An **entity** is anything that can be identified by a fixed number of its characteristics, called **attributes**; the attributes have names and values, and the values are the data that is stored in the table. (Unfortunately, *attributes* is an overused word in computing; in relational databases, think of an attribute as a "column of a table," where the "attribute names" are the column headings and the "attribute values" are the entries. We use the term *tag attributes* when we mean the attributes of XML.)

To relate entities and attributes to the metadata discussion earlier in this chapter, think of the attribute's name as the tag used in the Identity role, and the attribute values as the content enclosed in the tags. An entity is a group of attributes collected together by a tag used in the Affinity role. When describing affinity, we noted that the tagged data that we were grouping together all applied to one object, which was why it made sense to enclose it in tags. That object is the entity—the thing that the data applies to. Think of the tag used in affinity as the entity's name, and the tags that we allow within it as its attributes. So, an "island" is an entity, and its attributes include "name," "area," and "elevation"; see Figure 16.4. An "archipelago" is also an entity.

Island		
Name	*Area*	*Elevation*
Isabela	4588	1707
Fernandina	642	1494
Tower	14	76
Santa Cruz	986	846

Figure 16.4 A table instance for the island entity.

So, an entity defines a table. The name of the entity is the name of the table, and each of its possible attributes is assigned a column with the column heading being the attribute name. The values in the columns are the attributes' values, and the rows are the entity instances. We say **entity instances** for a row because a specific set of values for the attributes of an entity—that is, the content of the row—define one particular object, an instance of the entity. So, "name" and "area" are attributes of "island" generally, but "Tahiti" and "1048" define a specific island; a row with those values is an instance of the "island" entity. Any table containing specific rows is said to be a **table instance**.

In addition to having a name, attributes also have a **data type**, such as number, text, image, and so on. (We haven't been concerned about data types so far.) The data type defines the form of the information that can be stored in a field. By specifying the data type, database software can prevent us from accidentally storing bad information in a table. To connect the data type to the tagging discussed earlier, think of the type as a tag attribute, as in `<name type="text">` or `<area type="number">`, though database software uses other forms of metadata to record the data type.

fit BYTE

For the Record. Because databases are so important and long-studied, the concepts are known by several terms. The technical term for a row is a **tuple** (short u) from words like quintuple, sextuple, and septuple. Rows are often called **records**, a holdover from computing's punch-card days. Attributes are also known as **fields** and **columns**; an attribute's data type is sometimes referred to as its **format**. Tables are technically known as **relations**.

Properties of Entities

One curious property of a relational database table is that it can be empty. That is, the table has no rows. (Visualize the idea by deleting the last four rows of the table in Figure 16.4.) It seems odd, but it makes sense. Once we agree that an entity is anything defined by a specific set of attributes, then in principle a table exists with a name and column headings. When we specify entity instances, we'll have rows. So, among the instances of any table is the "empty instance."

`Instances Are Unordered.` Each distinct table is a different table instance. Two table instances will have a different set of rows. And, tables with the same rows, but reordered—that is, the same rows are listed in different sequence, say one sorted and the other unsorted—are the same table instance. Thus, the order of the rows doesn't matter in databases. We need to list them in some order, of course, but any order will do.

The attributes (columns) are also considered to be unordered, though we must list them in some order. Since the attributes have a name—think of the column heading or the tag—they do not have to be tracked by position.

Notice that the columns are unordered and the rows are unordered, but that doesn't mean that data in the table can go anywhere. Columns stay as columns, because they embody a kind of data being stored, and the items in a row stay as a row, because they are descriptive of an individual entity. The freedom to move the data is limited to exchanging entire rows or exchanging entire columns.

`Uniqueness.` There are few limits on what an entity can be. Things that can be identified or distinguished from each other based on a fixed set of attributes qualify as entities, which covers almost everything. Amoebas are not entities, because they have no characteristics that allow us to tell them apart. (Perhaps amoebas can tell each other apart, and if we could figure out how, then the characteristics on which they differ could be their attributes, allowing them to become entities.) Of course, one-celled animals are entities.

Because entities can be distinguished by their attributes, they are unique. Accordingly, in a database table no two rows can be the same. Unique instances is usually what we intend when we set up a database. For example, the database containing information about registered students at a college intends for each row—corresponding to a student—to be unique since the students are. When we set up the database, we ensure that we store information that uniquely identifies each student—such as name, birth date, parents' names, and permanent address.

In cases where the entities are unique but it is difficult to process the information, we might select an alternate encoding. For example, killer whales can be distinguished by the arrangement of their black-and-white markings. Even though images can be stored in a database, it is difficult to compare two images to determine if they show the same whale. So, we assign names to the killer whales, which are easy to manipulate, letting a human do the recognition and assign the name.

Notice that the two rows can have the same value for some attributes, just not all attributes.

`Keys.` The fact that no two rows in a database table are identical motivates us to ask which attributes distinguish them. In most cases, there will be several possibilities. Single attributes might be sufficient, like island name, or pairs of attributes like island name and archipelago name may be needed if certain island

names, like Santa Maria, are common. Or we may have to consider three or more attributes taken together to ensure uniqueness. Any set of attributes for which all entities are different is called a **candidate key**. Because database tables usually have several candidate keys, we choose one and call it the primary key. The **primary key** is the one that the database system will use to decide uniqueness.

Notice that candidate keys qualify only if they distinguish among all entities forever, not just those that are in the table at the moment, that is, the given instance. For example, all currently registered college students might have different names (first, middle, and last taken together), making the name attribute unique for the current class. But, as we know, there are many people with identical names, and so that triple is not an actual candidate key.

If no combination of attributes qualifies as a candidate key, then a unique ID must be assigned to each entity. That's why your school issues students IDs: Some other student might match you on all of the attributes that the school records in its database, but because the school doesn't want to worry about the possibility of two distinct students matching on its key, it issues an ID number to guarantee that one attribute distinguishes each student.

`Atomic Data.` In addition to requiring a description of each attribute's type of data—for example, number, text, or date—databases also treat the information as **atomic**, that is, not decomposable into any smaller parts. So, for example, an address value

```
1234 Sesame Street
```

is treated in a database table as a single sequence of ASCII characters; the street number and the street name cannot be separated. This is why forms—both paper and Web—have separate fields for street, city, state, and postal code: Most uses of address information must manipulate the city, state, and postal code information independently, which means the data must be assigned to separate fields.

The "only atomic data" rule is usually relaxed for certain types of data, such as dates, time, and currency. Strictly speaking, a date value `01/01/1970` must be treated as a single unity; any use of the date that refers to the month alone has to store the date as three attributes: day, month, and year. But database software usually bends the rules, allowing us to specify the format of the data attribute, say `dd/mm/yyyy`, which allows the program to understand how the field decomposes. This format saves us the trouble of manipulating three attributes.

Database Schemes

Though tags may be a precise way to specify the structure of a table, it is a cumbersome way to define a table. Accordingly, database systems specify a table as a **database scheme** or **database schema**. The scheme is a collection of table definitions that gives the name of the table, lists the attributes and their data types, and identifies the primary key. Each database system has specific requirements for how a scheme is presented, so there are no universal rules. We use an informal

approach in which the attributes are given by their name, a data type, and a comment describing the meaning of the field. Figure 16.5 shows a database scheme for the `Island` table.

```
Island
  iName          Text      Island Name
  area           Number    Area in square kilometers
  elevation      Number    Highest point on the island
Primary Key: iName
```

Figure 16.5 Database table definition for an `Island` table.

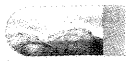
XML Trees and Entities

As mentioned earlier, relational database tables and XML trees are not the same. A full explanation of the differences is for database experts, but basically relational databases are more restrictive than XML trees; the limits make them more powerful and allow them to do more for us, as you'll soon see. For us, the main difference concerns the Collection Rule: When entity instances are grouped, all entities within the tag must have the same structure, because that structure defines the attributes that make up a row.

For example, the `Island` table for the Galápagos in Figure 16.4, which is a legal relational database table, can be encoded in XML as shown in the answer to the *Try It!* on page 448. So, the relational formulation and the XML formulation are the same. But, when we added the `<a_name>` tags inside of the `<archipelago>` tags, we violated the relational requirement that all entities have the same structure: The `<a_name>` was not an `<island>` entity. Including the `<a_name>` tag made sense for XML, but not for the relational model. So, they are related but not identical.

Database Tables Recap

Summarizing the important points of the last few sections, tables in databases are not simply an arrangement of text, but rather they have a structure that is specified by metadata. The structure of a database table is separate from its content. A table structures a set of entities—things that we can tell apart by their attributes—by naming the attributes and giving their data types. The entities of the table are represented as rows. We understand that rows and columns are unordered in databases, though when they are listed they have to be listed in some order. Tables and fields should have names that describe their contents, the fields must be atomic (i.e., indivisible), and one or more attributes define the primary key (i.e., field(s) with the property of having a different value for every row in any table instance ever).



Operations on Tables

A database is a collection of database tables. The main use of a database is to look up information. Users specify what they want to know and the database software finds it. For example, imagine a database containing Olympic records. There might be a table of participants for each Olympics, including attributes of name, country, and event; there might be a table of the medal winners for each Olympics, including attributes for the medal, the winner's name, the winner's country, and perhaps the score, distance, time, or other measure of the achievement. The database has many tables, but if we want to know how many marathon medalists have come from African countries, there is no table to look in—the table of medalists probably includes all winners in all sports, not just marathon winners from African countries. The data is in the database, but it's not stored in a single table where we, or the computer, can look it up. What we need to do is describe the information we want in such a way that the computer can figure out how to find it for us.

Database operations allow us to ask questions of a database in a way that lets the software find the answer for us. For example, we will ask for the number of African marathon winners by asking:

Put together the medalists for all of the Olympic Games (the operation will be called union*), find the rows of medalists who won in the marathon (the operation will be called* select*), and pick out those who come from African countries (the operation will be called* join*). Count the resulting rows, which is the answer we want.*

This example illustrates two important points. First, we can perform operations on tables to produce tables. It's analogous to familiar operations on numbers: Operations like addition combine two numbers and produce another number; operations like union combine two tables and produce another table. Second, the questions we ask of a database are answered with a whole table. If the question has a single answer—who won the marathon in 2000?—then the table instance answering the question will have only a single row. Generally there will be several answers forming the table. Of course, if there is no answer, the table will be empty.

In this section we illustrate the idea of combining tables to produce new tables. For this example, we imagine a table of the countries of the world as might be used by a travel agency. Its structure and sample entries are shown in Figure 16.6. Using that table, `Nations`, we'll investigate the five fundamental operations that can be performed on tables: `Select`, `Project`, `Union`, `Difference`, and `Product`.

Select Operation

The `Select` operation takes rows from one table to create a new table. Generally we specify the `Select` operation by giving the (single) table from which rows are to be selected and the test for selection. We use the syntax:

`Select` *Test* `From` *Table*

Nations

Name	text	*Common rather than official name*
Domain	text	*Internet top-level domain name*
Capital	text	*Nation's capital*
Latitude	number	*Approx. latitude of capital*
N_S	Boolean	*Latitude is N(orth) or S(outh)*
Longitude	number	*Approx. longitude of capital*
E_W	Boolean	*Longitude is E(ast) or W(est)*
Interest	text	*A short description of the country*

Primary Key: Name

Name	Dom	Capital	Lat	NS	Lon	EW	Interest
Ireland	IE	Dublin	52	N	7	W	History
Israel	IR	Jerusalem	32	N	35	E	History
Italy	IT	Rome	42	N	12	E	Art
Jamaica	JM	Kingston	18	N	77	W	Beach
Japan	JP	Tokyo	35	N	143	E	Kabuki

Figure 16.6 The `Nations` table definition and sample entries.

The *Test* is to be applied to each row of the given table to decide if it should be included in the new result table. The *Test* is a short formula that tests attribute values. It is written using attribute names, constants like numbers or letter strings, and the relational operators $<$, $\leq$, $\neq$, $=$, $\geq$, $>$. The **relational operators** test whether the attribute value has a particular relationship, for example, `Interest = 'Beach'` or `Latitude < 45`. If the *Test* is true, the row is included in the new table; otherwise, it is ignored. Notice that the information used to create the new table is a copy, so the original table is not changed by `Select` (or any of the other table-building operations discussed here).

To use the `Nations` table to create a table of countries with beaches, we write a `Select` command to remove all rows for countries that have `Beach` as their `Interest` attribute. The operation is

```
Select Interest = 'Beach' From Nations
```

This gives us a new table, shown in part in Figure 16.7. Notice that the information in the last column is constant because the *Test* required the word "Beach" for that field for all selected rows.

The *Test* can be more than a test of a single value. For example, we can use the logical operations `AND` and `OR` in the way they were used to search in Chapters 5 and 6. So, for example, to find countries whose capitals are at least 60° north latitude, we write

```
Select Latitude ≥ 60 AND N_S = 'N' From Nations
```

which should produce a four-row table created from the `Nations` table's rows for Greenland, Iceland, Norway, and Finland.

Name	Dom	Capital	Lat	NS	Lon	EW	Interest
Australia	AU	Canberra	37	S	148	E	Beach
Bahamas	BS	Nassau	25	N	78	W	Beach
Barbados	BB	Bridgetown	13	N	59	W	Beach
Belize	BZ	Belmopan	17	N	89	W	Beach
Bermuda	BM	Hamilton	32	N	64	W	Beach

Figure 16.7 Part of the table created by selecting countries with a Test for Interest equal to Beach.

Project Operation

If we can pick out rows of a table (using `Select`), we should be able to pick out columns too. `Project` (pronounced *pro-JECT*) is the operation that builds a new table from the columns of an existing table. We only need to specify the name of a table and the columns (field names) from it to be included in the new table. The syntax is

`Project` *Field_List* **`From`** *Table*

For example, to create a new table from the `Nations` table without the `capital` and position information—that is, to keep the other three columns—write

```
Project Name, Domain, Interest From Nations
```

The new table will have as many rows as the `Nation` table, but just three columns. Figure 16.8 shows part of that table.

Name	Dom	Word
Nauru	NR	Beach
Nepal	NP	Mountains
Netherlands	NL	Canals
New Caledonia	NC	Beach
New Zealand	NZ	Adventure

Figure 16.8 Sample entries for a `Project` operation on `Nations`.

`Project` does not *always* result in a table with the same number of rows as the original table. When the new table includes a key from the old table (e.g., `Name`), the key makes each row distinct, so the new table will include fields from all rows of the original table. But if some of the new table's rows are the same—which can't happen if key columns are included, but can if there is no key among the chosen columns—they will be merged together into a single row. The rows have to be merged because of the rule that the rows of any table must always be distinct. If rows in one table are merged, the two tables will, of course, have different numbers of rows. So, for example, to list the `Interest` descriptions that travel agents

use to summarize countries, we create a new table of only the last column of `Nations`.

```
Project Interest From Nations
```

This produces a one-column table with a row for each descriptive word: `Beach` appears once, `Mountains` appears once, and so on. Thus the table has as many rows as unique words, and because of merging, it does not have as many rows as `Nations`.

We often use `Select` and `Project` operations together to "trim" base tables to keep only some of the rows and some of the columns. To illustrate, we define a table of the countries with northern capitals, called `Northern`, and define it with the command

```
At60OrAbove = (Select Latitude ≥ 60 AND N_S = 'N' From Nations)
```

which is the table we created earlier. To throw away everything except the name, domain, and latitude to produce `Northern`, we write

```
Northern = (Project Name, Domain, Latitude From At60OrAbove)
```

as shown in Figure 16.9.

Name	Dom	Lat
Finland	FI	61
Greenland	GL	72
Iceland	IS	65
Norway	NO	60

Figure 16.9 `Northern`, the table of countries with northern capitals.

Another way to achieve the same result is to combine the two operations:

```
Project Name, Domain, Latitude From
   (Select Latitude ≥ 60 AND N_S = 'N' From Nations)
```

First a temporary table is created with the four countries, just as before. Then the desired columns are selected. It might be a slightly more efficient solution if we don't need the `At60OrAbove` table for any other purpose, but generally either solution is fine.

Union Operation

Besides picking out rows and columns of a table, another operation on tables is to combine two tables. This only makes sense if they have the same set of attributes, of course. The operation is known as `Union`, and is written as though it were addition:

Table1 + *Table2*

The plus sign (+) can be read "combined with." So, if the table of countries with capitals at least 45° south latitude are named `At45OrBelow` with the command

```
At45OrBelow = (Select Latitude ≥ 45 AND N_S = 'S' From Nations)
```

then we can define places with their capitals closest to the poles using the union operation. Call the result `ExtremeGovt` and define it by

```
ExtremeGovt = At60OrAbove + At45OrBelow
```

The result is shown in Figure 16.10. This table could also have been created with a complex `Select` command.

Name	Dom	Capital	Lat	NS	Lon	EW	Interest
Falkland Is	FK	Stanley	51	S	58	W	Nature
Finland	FI	Helsinki	61	N	26	E	Nature
Greenland	GL	Nuuk	72	N	40	W	Nature
Iceland	IS	Reykjavik	65	N	18	W	Geysers
Norway	NO	Oslo	60	N	10	E	Vikings

Figure 16.10 The `ExtremeGovt` table created with `Union`.

`Union` can be used to combine separate tables, say, `Nations` with `Canada_Provinces`. (`Canada_Provinces` gives the same data about the provinces as `Nations` does about countries, except the `Domain` field is `CA` for all rows.) For example, had the `At60OrAbove` table been defined by

```
Select Latitude ≥ 60 AND N_S = 'N'
    From (Nations + Canada_Provinces)
```

then the Yukon would be included because its capital, Whitehorse, is north of 60°.

Difference Operation

The opposite of combining two tables with `Union` is to remove from one table the rows also listed in a second table. The operation is known as `Difference` and it is written with the syntax

Table1 – *Table2*

The operation can be read, "remove from *Table1* any rows also in *Table2*." Like `Union`, `Difference` only makes sense when the table's fields are the same. For example,

```
Nations – At60OrAbove
```

produces a table without those countries with northern capitals—that is, without Finland, Greenland, Iceland, and Norway. Interestingly, this command works just as well if `At60OrAbove` had included Canadian provinces like the Yukon. That is,

in a Difference command, the items "subtracted away" do not have to exist in the original table.

Product Operation

Adding and subtracting tables is easy. What is multiplying tables like? The Product operation on tables, which is written as

Table1 × *Table2*

creates a supertable. The table has the columns from *both* tables. So, if the first table has five attributes and the second table has six attributes, the Product table has eleven attributes. The rows of the new table are created by *appending* or concatenating each row of the second table to each row of the first table—that is, putting the rows together. The result is the "product" of the rows of each table.

For example, if the first table is Nations with 230 rows, and the second table has 4 rows, there will be 230 × 4 = 920 rows, because each row of the Nations table is appended with each row of the second table to produce a row of the result.

To illustrate, suppose you have a table of your traveling companions, as described in Figure 16.11(a), containing the information shown in Figure 16.11(b).

```
Travelers
  Friend    Text  A Traveling Companion
  Homeland  Text  Friend's Home Country

Primary Key: Friend
```

(a)

Friend	Homeland
Isabela	Argentina
Brian	South Africa
Wen	China
Clare	Canada

(b)

Figure 16.11 (a) The definition of the Travelers table, and (b) its values.

Then the Product operation

```
Super = Nations × Travelers
```

creates a new table with ten fields—eight fields from Nations and two fields from Travelers—a total of 920 rows. Some of the rows of the new table are shown in Figure 16.12. For each country, there is a row for each of your friends.

The Product operation may seem a little odd at first because its all-combinations approach merges information that may not "belong together." And it's true. But most often, Product is used to create a supertable that contains both useful and useless rows, and then it is "trimmed down" using Select, Project, and Difference to contain only the intended information. This is a powerful approach that we will use repeatedly in later sections.

To illustrate, suppose your traveling companions volunteer to tutor students preparing for the National Geographic Society's Geography Bee. Each friend agrees

Name	Dom	Capital	Lat	NS	Log	EW	Interest	Friend	Homeland
Cyprus	CY	Nicosia	35	N	32	E	History	Clare	Canada
Czech Rep.	CZ	Prague	51	N	15	E	Pilsner	Isabella	Argentina
Czech Rep.	CZ	Prague	51	N	15	E	Pilsner	Brian	South Africa
Czech Rep.	CZ	Prague	51	N	15	E	Pilsner	Wen	China
Czech Rep.	CZ	Prague	51	N	15	E	Pilsner	Clare	Canada
Denmark	DK	Copenhagen	55	N	12	E	History	Isabella	Argentina

Figure 16.12 Some rows from the supertable that is the product of `Nations` and `Travelers`. For each row in `Nations` and each row in `Travelers`, there is a row in the product table that combines them.

to tutor students "on their part of the world," that is, in the quarter of the planet from which they come. So, Isabella, who comes from Argentina in the southern and western hemispheres, agrees to tutor students on the geography of that part of the world, and so on. Then you can produce a master list of who's responsible for each country. We'll call it the `Master` table. It is produced by these commands:

```
Super  = Nations × Travelers
Assign = (Select N_S = 'S' AND E_W = 'W'
              AND Friend = 'Isabella' From Super)
         + (Select N_S = 'S' AND E_W = 'E'
              AND Friend = 'Brian' From Super)
         + (Select N_S = 'N' AND E_W = 'E'
              AND Friend = 'Wen' From Super)
         + (Select N_S = 'N' AND E_W = 'W'
              AND Friend = 'Clare' From Super)
Master = Project Name, Friend From Assign
```

Notice that we have used `Product` (×), `Union` (+), `Select`, and `Project`.

How do these commands work? The `Super` table is the product table discussed earlier with a row for each nation paired with each friend (see Figure 16.12). Then the `Assign` table is created by the `Union` operation (+) that combines four tables, each created by a `Select` operation from `Super`. The first `Select` keeps only those countries from `Super` with Isabella's name that are also in the southern and western hemispheres. The second `Select` keeps only those countries from `Super` with Brian's name that are in the southern and eastern hemispheres. The same kind of operations are used for Wen and Clare. The resulting `Assign` table has 230 rows—the same as the original `Nations` table—with one of your friends' names assigned to each country.

We know that all of the countries are in the `Assign` table because every country is in one of the four hemisphere pairs, and in `Super` there is a row for each country for each friend. When the right combination "comes up," the country will be chosen by one of the four `Select`s. In addition, `Assign` has the property that each person is given countries in "their" part of the world. (Wen has been assigned the greatest amount of work!) Finally, we throw away all of the location information to create our `Master` list, keeping only the names of the countries and the friends

responsible for tutoring students about that geography. Part of the result is shown in Figure 16.13.

Name	Friend
Chad	Wen
Chile	Isabella
China	Wen
Christmas Is.	Clare
Cocos Is.	Brian

Figure 16.13 A portion of the `Master` table of your friends' assignments.

We have introduced five basic operations on tables. They are straightforward and simple. It is surprising, therefore, that these five are the only operations needed to create any table in a relational database. In practice, we will rarely use the operations directly, because they are incorporated into database software. When we want to create tables from other tables—an idea that is now quite natural—we will hardly be aware that we're using these operations.

*fit*BYTE

Quotient Intelligence. There is a `Divide` operation on tables, but it's complicated and rather bizarre. Because it doesn't give us any new capabilities, we will leave it to the experts.

Join Operation

Another powerful and useful operation for creating database tables is `Join`. Indeed, it is so useful that although `Join` can be defined from the five primitive database operations of the last section, it is usually provided as a separate operator.

Join Defined

`Join` combines two tables, like the `Product` operation does, but it doesn't necessarily produce all pairings. If the two tables each have fields with a common data type, the new table produced by `Join` combines only the rows from the given tables that match on the fields, not all pairings of rows, as does `Product`. We write the `Join` operation as follows:

Table1 ⋈ *Table2* **On** *Match*

The unusual "bow tie" symbol suggests a special form of `Product` in which the two tables "match up." *Match* is a comparison test involving fields from each table, which when true for a row from each table produces a result row that is their concatenation. To refer to attributes in each table, we use the notation *Table.Field*, as in `Master.Name`.

Join Applied

To show how `Join` works, recall the `Northern` table (Figure 16.9) and the `Master` table of your friends' assignments (Figure 16.13). The `Join`

```
Master ⋈ Northern On Master.Name = Northern.Name
```

pairs all rows where the country name matches the home country of a friend. Which rows are those? This is how to find out. Beginning with the first row of the `Master` table (shown here):

Name	Friend
Afghanistan	Wen
Albania	Wen
...	...

the Afghanistan row does not have the same `Name` field as any of the four countries of `Northern`, so it is not part of the result. Nor does the `Name` in the second row of `Master` (Albania) appear as a `Name` field in `Northern`. Indeed, only four rows of `Master` have the same `Name` field as rows in `Northern`: Finland, Greenland, Iceland, and Norway. We combine those four rows with their corresponding rows in `Northern` to produce the four-row result shown in Figure 16.14. As you see, `Join` associates the information from the rows of two tables in a sensible way. Thus, `Join` is used to create new associations of information in the database.

There are at least two ways to think about the `Join` operation. One way is to see it as a "lookup" operation on tables. That is, for each row in one table, locate a row (or rows) in the other table with the same value in the common field; if found, combine the two; if not, look up the next row. That's how we explained it in the last paragraph. Another way is to see it as a `Product` operation forming all pairs of the two tables, and then eliminating all rows that don't match in the common fields. Both ideas accurately describe the result, and the computer probably uses still another approach to produce the `Join` table.

`Join`, as described, is called a natural join because the natural meaning of "to match" is for the fields to be equal. But as is typical of IT, it is also possible to join using any relational operator ($<$, $\leq$, $\neq$, $=$, $\geq$, $>$), not just = to compare fields. Unnatural or not, a `Join` where `T1.fieldID < T2.fieldID` can be handy.

African Marathon Runners

To complete the task we discussed earlier of finding out how many African marathon winners there have been in the history of the Olympics, we assume there are tables `Medalists1896`, `Medalists1900`, . . ., `Medalists2004`, and that there is a table, `Africa`, of African nation names, which includes colonial names like Rhodesia and modern names like Zimbabwe.

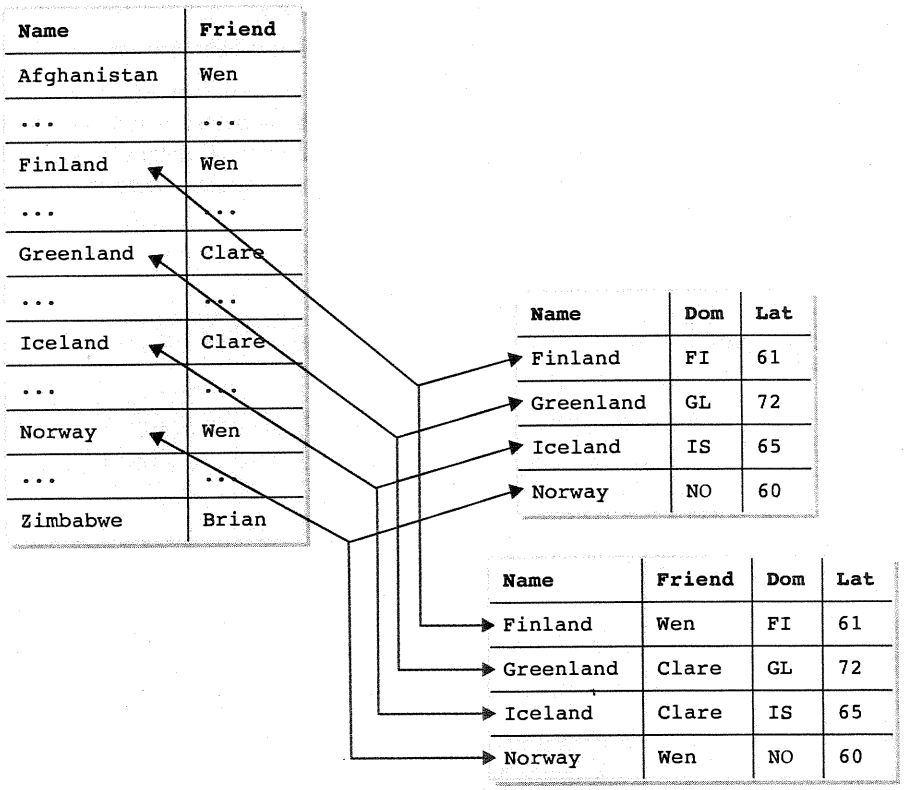


Figure 16.14 The Join operation: Master ⋈ Northern.

Assuming these tables, we write

```
All_Medalists = Medalists1896 + Medalists1900 +
                ... + Medalists2004
```

which is a lot of typing. The `All_Medalists` table contains the names, medal, event, and country of everyone who won in the Olympics. Next, we pick out the marathon winners with

```
Distance26 = Select medal='gold' AND event='marathon'
             From All_Medalists
```

The `Distance26` table contains all runners who received a gold medal in the Olympic marathon event. Next, eliminate everyone but the African winners with

```
Africa_marathon = Distance26 ⋈ Africa
                  On Distance26.country = Africa.name
```

producing a table of African winners. Counting the rows produces the result. Database software provides a function for counting the number of rows in a table, which is applied in the present case as

```
count(Africa_marathon)
```

Using the operators, we specified a set of tables that allowed us to find our solution. We will refine this skill after the next section.

*fit*BYTE

You Can Look It Up. Database systems such as Microsoft's Access, MySQL (pronounced my S-Q-L), and commercial database systems such as Oracle give users the ability to answer database queries using the five primitive operations and `Join`.

Structure of a Database

You have learned that by using the five primitive operations and `Join` we can create tables from tables to answer questions from a database. But usually these operations are used in a slightly different way. We don't usually ask a single question and quit. Rather, we want to arrange the information of a database in a way that users see a relevant-to-their-needs view of the data that they will use continually. Figure 16.15 shows a schematic of this idea.

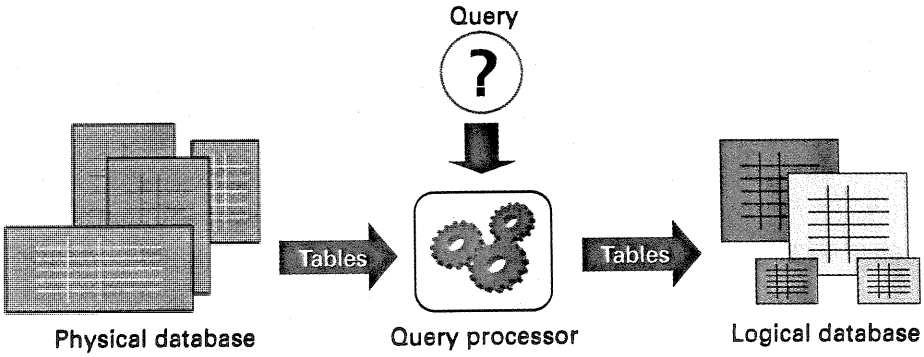


Figure 16.15 Structure of a database system. The physical database is the permanent repository of the data; the logical database, or view of the database, is the form of the database the users see. The transformation is implemented by the query processor, and is based on queries that define the logical database tables from the physical database tables.

In Figure 16.15 you see that there are two forms of tables. The physical database, stored on the disk drives of the computer system, is the permanent repository of the database. The logical database, also known as the *view of the database*, is created for users on-the-fly and is customized for their needs. Why do we use this two-level solution? The answer requires that we look a little closer at the two groups of tables.

Physical and Logical Databases

The point of the two-level system is to separate the management of the data, which is typically done at the physical database level, from the presentation of the data, which typically involves many different versions for many different users.

`Physical Database.` The physical database is designed by database administrators so that the data is fast to access. More importantly, the physical database is set up to avoid redundancy, that is, duplicating information. It seems obvious that data should not be stored repeatedly because it will waste space, but disk space is *extremely* cheap, implying that that isn't the reason to avoid redundancy. Rather, if data is stored in multiple places in the physical database, there is a chance—possibly, a good chance—that when it's changed in one place, it will not be changed in every other place where it is stored. This causes the data to be *inconsistent*.

For example, if your school stores your home address, and your major department also stores a separate copy of your address, then when you notify the school of your new residence, both addresses should be changed. But, with multiple copies, that might not occur. If the database contains two different addresses for you, then the school has no idea which address is correct; perfectly good information gets turned into garbage because it is inconsistent. For this reason, database administrators make sure that there is only one copy of each piece of data. That is, data is not stored redundantly.

It might seem risky to keep only one copy of the data: What happens if it accidentally gets deleted or the disk crashes? Database administrators worry about this problem all the time, and have a process of making backup copies of the database, which they store in a safe place, *never to be used*. That is, until the data is accidentally deleted or the disk crashes—in other words—when the other copy is gone. There is still only one copy.

Avoiding redundancy is obviously good, but keeping one copy seems to ignore the fact that multiple users need the information. The administration needs to send tuition bills, the dean needs to send notification that you "made the list," and the Sports Center needs to send you the picture of your photo finish; they all need your address. Where do they get their copy? That's where the logical database comes in.

`Logical Database.` The logical database shows users the view of the information that they need and want. It doesn't exist permanently, but is created for them fresh every time they look at it. This solves the problem of getting everyone a copy of the address. It's retrieved from the one copy stored in the physical database, and provided to the users as needed, fresh every time. Creating a new copy each time is essential, because if it were to be created once and then stored on the user's computer, then there would be two copies of the information again—the copy in the physical database and the one in the logical database—making the data redundantly stored. So, it never stays on the user's computer; it's always recreated. As a result, when you notify the administration that you moved in the morning, the dean can send you a congratulatory letter in the afternoon and have your correct address.

The other advantage of creating specialized versions of the database for each user is that different users want to see different information. For example, the Sports Center needs to record a student's locker number, but no other unit on campus

cares. Similarly, the fact that a student is on academic probation is information that most users of the school's database don't need to know, and it should not be included in their view. In principle, each user wants a different view of the database.

Queries. Queries are the key to making this two-level organization work. Each user group, say the Dean's Office, needs a version of the database created for them. For each user table a query is formulated. A **query** is a specification using the five operations and `Join` that define a table from other tables. Think of the query as being written as described in the previous sections, but it is actually written in the standard database language SQL, short for Structured Query Language. Then, when the dean clicks on the table of Spring Term Grades, the database system runs the query that defines that table, creating it and displaying it to the dean. It probably doesn't exist in that form in the physical database, but `Select`, `Project`, and the other operations can define how to create it from the data that is physically stored. On the next day, when the dean opens the table of Spring Term Grades again, a new copy will be created, which means that the grade change made the previous afternoon by some physics professor (and stored in the physical database) will be visible to the dean.

It all seems pretty complicated, but it is not. Indeed, in the next section you will see that it is all rather straightforward.

Defining Physical Tables

In this section we define two tables to be used for illustration purposes, focusing on the roles of keys and relationships.

Database Schemes. Recall that the metadata specification of a database's tables is given by a database schema, or database scheme. Interactive software can help us define a database schema, but, as we saw earlier, declaring an entity's structure is easy enough to do without software. The database schema is important because it describes the database design. When we want to analyze a database design, we look at its schema.

To illustrate the basics of the two-level approach, imagine a college having a set of tables in its database schema, two of which are `Student` and `Home_Base`.

```
Student
Student_ID      Number     Eight digits
First_Name      Text       Single name, capitalized
Middle_Name     Text       All other names, but family
Last_Name       Text       Family name
Birthdate       Date
Grade_Point     Number     0 <= GPA <= 4
Major           Text       None, or degree granting unit
On_Probation    Boolean    0 is 'no'; 1 is 'yes'

Primary Key: Student_ID
```

```
Home_Base
Student_ID      Number     Eight digits
Street          Text       All address info before city
City            Text       No abbreviations like NYC
State           Text       Or province, canton, prefecture ...
Country         Text       Standard postal abbreviations OK
Postal_Code     Text       Full postal code

Primary Key: Student_ID
```

Figure 16.16 shows the preceding table definitions as they appear in the Microsoft Access database system. Notice that they are different forms of the same thing.

Connecting Database Tables by Relationships

The `Student` entity records the information basic to the person's identity and associates a student with his or her `Student_ID`. This is the college's master record of each student. Part of each student's information is where he or she lives.

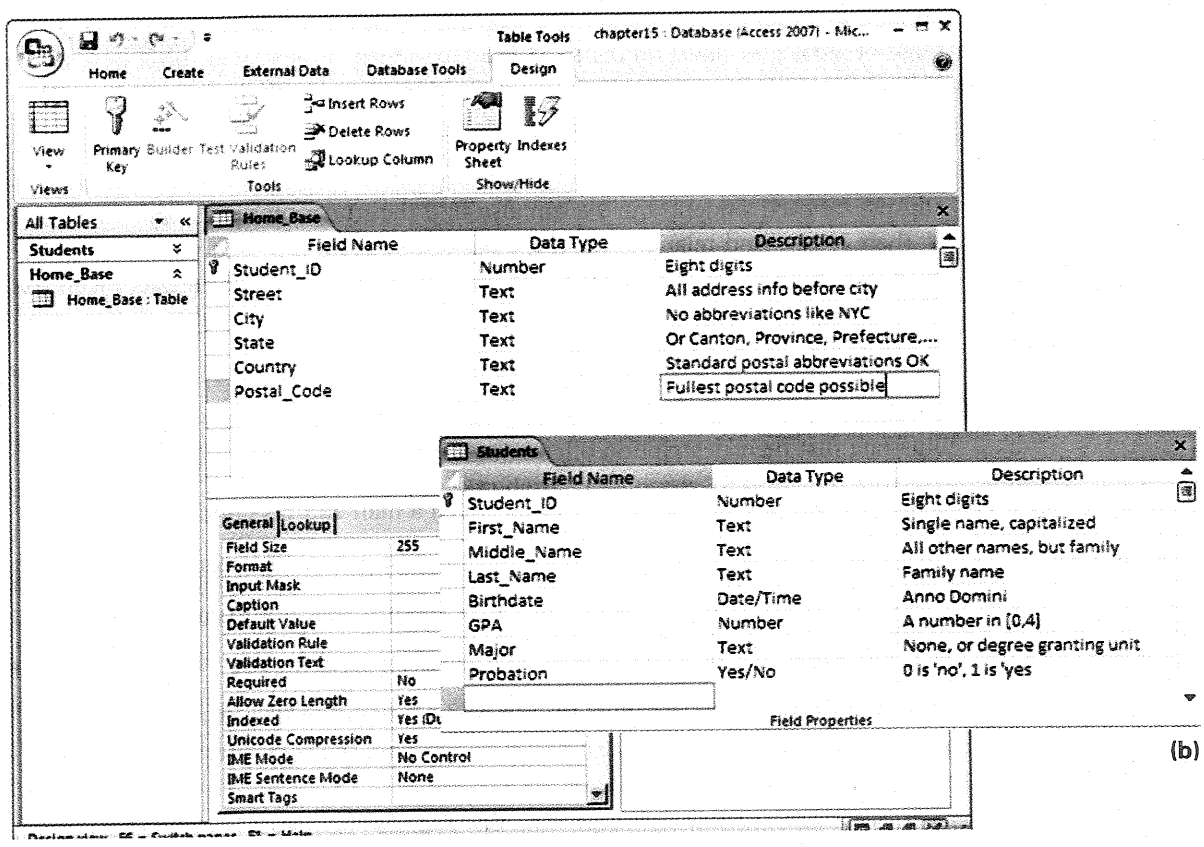


Figure 16.16 Table declarations from Microsoft Access 2007: (a) `Home_Base` table declaration shown in the design view; and (b) students table declaration. Notice that the key is specified by the tiny key next to `Student_ID` in the first column.

Though we could put addresses in the `Student` table, we decide not to. This is because other campus units will want to access the address information, but they shouldn't have access to all of the information (especially the sensitive information) about each student. The addresses are stored in a different table, the `Home_Base` table, which can have a lower security rating. Though these two tables are separate, they are not independent. The `Student_ID` connects each row in `Student` with his or her address in `Home_Base`. We say that there is a relationship between the two entities.

`The Idea of Relationships.` A **relationship** is a correspondence between rows of one table and the rows of another table. Relationships are part of the metadata of a database, and because they are critical to building the logical database from the physical database, we give them names and characterize their properties.

The relationship between `Student` and `Home_Base`—that is, for each row in `Student` there is a single row in `Home_Base` (found by the `Student_ID`)—will be called *Lives_At*. Setting up the tables in this way is largely equivalent to storing the address in `Student`, but not all relationships are so close. This one is especially close because it is based on the `Student_ID`, which is the key for both tables. (Recall that keys are unique, meaning no two rows can have the same value.) The *Lives_At* relationship is said to be one-to-one.

Because we used the key `Student_ID` in both tables, we not only can find the address for each student, but we can also find the student for each address. That is, there is a second relationship in the opposite direction, which we can call *Home_Of*, meaning that the home base entry is the address of the student who has that ID. Like *Lives_At*, *Home_Of* is a one-to-one relationship, because each row in `Home_Base` corresponds to a single row in `Student`.

`Relationship Examples.` Familiar relationships that we encounter every day illustrate that their description often ends with a preposition.

- *Father_Of*, the relationship between a man and his child
- *Daughter_Of*, the relationship between a girl and her parent
- *Employed_By*, the relationships between people and companies
- *Stars_In*, the relationships between actors and movies

Names of database relationships should be meaningful to help people working with the database, but like all names in computing, the computer doesn't know whether the name makes sense or not.

`Relationships in Practice.` Database software systems need to know what relationships exist among the tables if they are to help us create the logical databases. The systems allow us to define relationships among tables. The details are specific to each system, of course, but the example of *Lives_At* and *Home_Of* are shown in Figure 16.17 as they would appear in Microsoft Access.

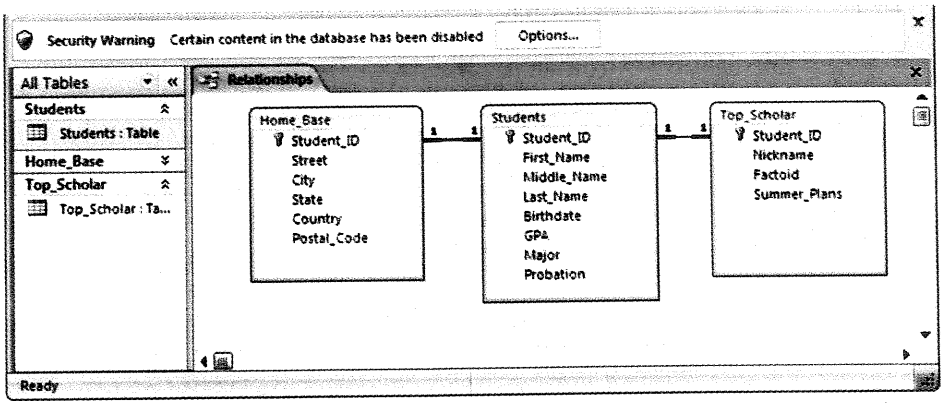


Figure 16.17 The *Relationships* window from the Microsoft Access database system; the 1-to-1 *Lives_At* and *Home_Of* relationships are shown between `Home_Base` and `Students`.

Defining Logical Tables

The school's administration probably thinks there is a single master list recording all of the data for each student. Because that's what they want to see, it's part of the administration's logical view of the database. So, we create it for them from the physical database.

`Construction Using Join.` The relationships between the `Student` and `Home_Base` tables allow us to construct a single table, `Master_List`, which contains the combined information from both tables. How? Using the natural `Join` operation, described earlier in this chapter. Recall that the natural `Join` creates a table out of two other tables by joining rows that match—it's an equality test—on specified fields. Thus, we write

```
Master_List = Student ⋈ Home_Base
              On Student.Student_ID = Home_Base.Student_ID
```

where the match is on the common field of `Student_ID`. Fields of the resulting table are shown in Figure 16.18. We don't lose anything by storing the basic student information in one table and the addresses in another, because with this simple command, we can create a table that recombines the information just as if it were stored in a single table.

The important idea here is that although we chose to store the information in two tables, we never lost the association of the information because we kept the `Student_ID` with the addresses. The relationship *Lives_At* lets us connect each student with his or her address by the `Student_ID`. The approach gives us the flexibility to arrange tables so as to avoid problems of redundancy—though we haven't demonstrated that benefit yet—while keeping track of important information, like where a person lives.

```
Student_ID
First_Name
Middle_Name
Last_Name
Birthdate
On_Probation
Street_Address
City
State
Country
Postal_Code
```

Figure 16.18 Attributes of the `Master_List` table. Being created from `Student` and `Home_Base` allows `Master_List` to inherit its data types and key (`Student_ID`) from the component tables.

`Practical Construction Using QBE.` Though it wasn't difficult to write the natural `Join` query in the last section to create the `Master_List` table, database systems can make it even easier for us. A technique developed at IBM in the 1970s, called **Query By Example (QBE)**, is available to us in the Microsoft Access system. Basically, the software gives us a template of a table, and we fill in what we want in the fields. That is, we give an example of what we want in the table, referencing fields from other tables that have already been defined. The software then figures out a query that creates the table from the sample table. It couldn't be easier! Figure 16.19 shows the QBE query window that will create `Master_List`.

The database software automatically creates the query needed for `Master_List`. What query did it create? We can ask and find out what it generated; see Figure 16.20. The query is expressed in SQL, the standard database query language. If we could read SQL—it's actually not too hard—we'd see that this query is the query we created for `Master_List`.

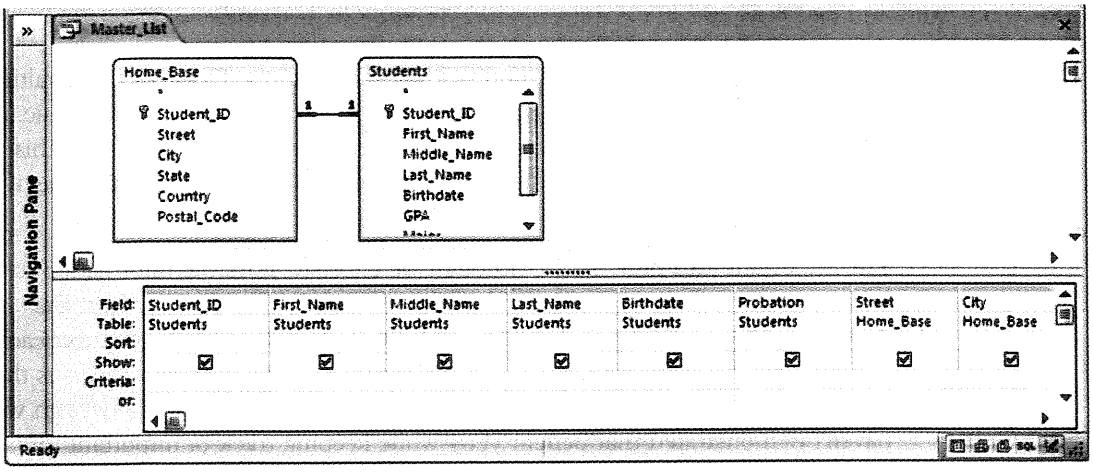


Figure 16.19 The Query By Example definition of the `Master_List` table from MS Access.

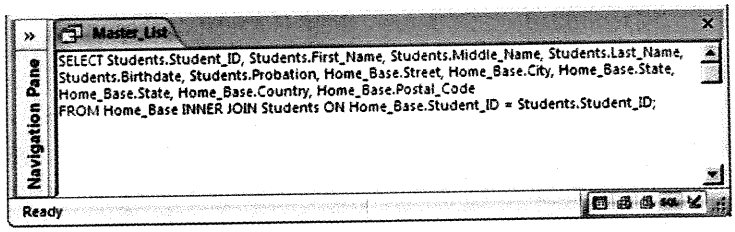


Figure 16.20 SQL query created from the Query By Example data in Figure 16.19.

The Dean's View

Because the school administrators probably want to see the entire student record, there isn't much advantage to breaking the files into smaller tables in the physical design, but it does make sense for others who only need to see parts of the database in their view. To illustrate one more logical database, we create a view for the dean.

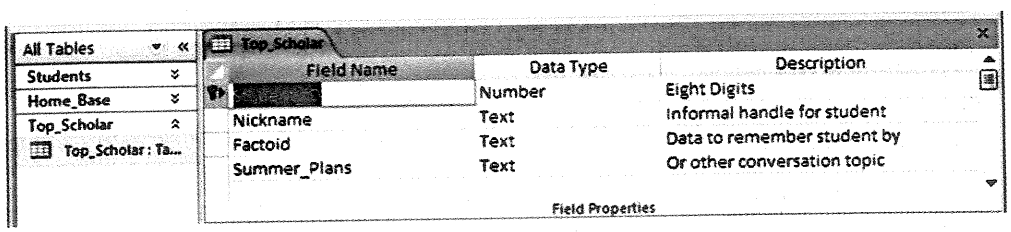
Storing the Dean's Data. We imagine that the database administrators have set up a special table with the dean's record of the students in the college. The table definition is shown in Figure 16.21. The `Top_Scholar` is basically information of interest only to the dean.

The table has a one-to-one relationship with the `Home_Base` table, based on the `Student_ID` attribute, just as `Student` does: For each scholar, there is an address in `Home_Base`. Therefore, there is a relationship between the `Top_Scholar` and the `Home_Base` tables, which we'll call *Resides_At*. The relationship gives the dean access to the student's hometown, which is something the dean wants to be

```
Top_Scholar:
    Student_ID        Number        Eight digits
    Nickname          Text          Informal handle for student
    Factoid           Text          Data to remember student by
    Summer_Plans      Text          Or other conversation topic

Primary Key: Student_ID
```

(a)



(b)

Figure 16.21 The `Top_Scholar` definition: (a) informal form, (b) in MS Access.

reminded of. Of course, there is a relationship in the opposite direction, too, from `Home_Base` to `Top_Scholar`.

`Student_ID` also connects `Top_Scholar` to `Student`. This is lucky, because otherwise the dean doesn't know a student's legal name, only the nickname. All of this data can be combined in tables for the dean's office using natural `Join` operations like we did in the last section, but the dean doesn't want to see all that information.

`Creating a Dean's View.` Imagine a table, known as the Dean's View, containing information specific to the dean's unique needs. For example, because the dean is not the person who sends letters to top students telling them they made the "Dean's List," the Dean's View doesn't need the students' full home addresses. (Someone else in the dean's office will need them.) The students' hometowns are enough information for the dean to make small talk at parties in honor of the good students. So the Dean's View will include information selected from the physical tables, as shown in Figure 16.22.

Deans_View

Name	Source Table	
Nickname	Top_Scholar	Used by the dean to seem "chummy"
First_Name	Student	Name information required because
Last_Name	Student	the dean forgets the person's actual name, being so chummy
Birthdate	Student	Is student of "drinking age"?
City	Home_Base	Hometown (given by city, state) is
State	Home_Base	important for small talk, but full address not needed by dean
Major	Student	Indicates what the student's doing in college besides hanging out
GPA	Student	How's student doing grade-wise
Factoid	Top_Scholar	Data to remember student by
Summer_Plans	Top_Scholar	Or other conversation topic

Figure 16.22 The Dean's View fields showing their source in physical database tables.

Notice that the dean doesn't even want to see the student ID. We use it to create the Dean's View, but it doesn't have to be part of what the dean looks at in the database view.

`Join Three Tables into One.` The first step in creating a query for the Dean's View is to note that it contains information from three tables: `Top_Scholar`, the table actually storing the data the dean wants kept; `Student`, the college's permanent record of the student; and `Home_Base`, the college's current address list. The information for each student must be associated to create the `Deans_View` table, and the `Join` operation is the key to doing it. The expression

```
Dean_Data_Collect = ((Top_Scholar ⋈ (Student ⋈ Home_Base
                    On Student.Student_ID=Home_Base.Student_ID)
                    On Student.Student_ID=Top_Scholar.Student_ID)
```

makes a table that has a row for each student in the dean's `Top_Scholar` table, but it also has all of the information from all three tables for that student. The association of each student's row in each table is accomplished by matching on the `Student_ID` attribute.

Trim the Table. The resulting table contains too much information, of course, because it has all the columns from the three tables. The dean doesn't want to see so much information. So, the second step is to retrieve only the columns the dean wants to see.

The `Project` operation retrieves columns:

```
Deans_View =
   Project Nickname, First_Name, Last_Name, Birthdate,
           City, State, Major, GPA, Factoid, Summer_Plans
   From Dean_Data_Collect
```

In English, the query says, "Save the `Nickname` column, `First_Name` column, and so forth, from the table, `Dean_Data_Collect`, that is formed by joining—that is, associating on `Student_ID`—the three tables `Top_Scholar`, `Student`, and `Home_Base`." This is precisely what the dean wants. The query defines the `Deans_View` table. Although the dean probably thinks the table exists physically, it is created fresh every time it's needed.

The join-then-trim strategy used to create the Dean's View is a standard approach to creating logical tables: a supertable is formed by joining several physical tables. These are then trimmed down to keep only the information of interest to the user. The `Deans_View` query used `Project`, but `Select` and `Difference` are also frequently used.

Software Creates Dean's View. If we add `Top_Scholar` to the Access database schema given in Figure 16.16, and include the one-to-one relationship between it and the other tables based on the `Student_ID`, as shown in Figure 16.17, then we can use Query By Example again to define the Dean's View, saving ourselves the effort of working out our own query, though writing SQL directly wouldn't be that difficult. Figure 16.23 shows the QBE window from Microsoft Access that defines the Dean's View from the three tables.

For the record, the SQL query that Access produced for us based on our example from Figure 16.23 is shown in Figure 16.24. It is the identical query we developed ourselves, expressed in SQL syntax. (Notice that SQL uses the word "Select" where we used "Project"; the concepts are the same, but the term is different between the theory of relational databases and the SQL language. This naming inconsistency is an annoying feature of the study of databases.)

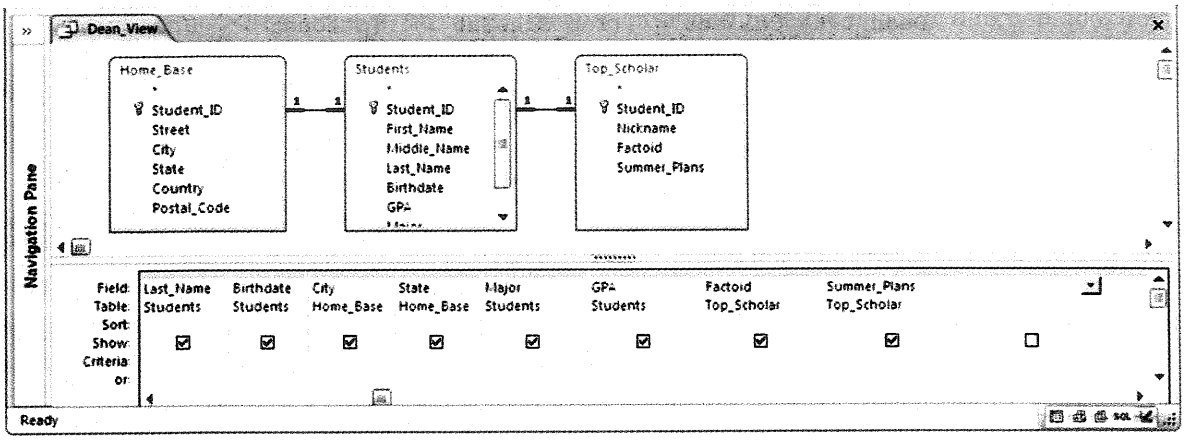


Figure 16.23 The Query By Example definition of the Dean's View table as expressed in Microsoft Access 2007.

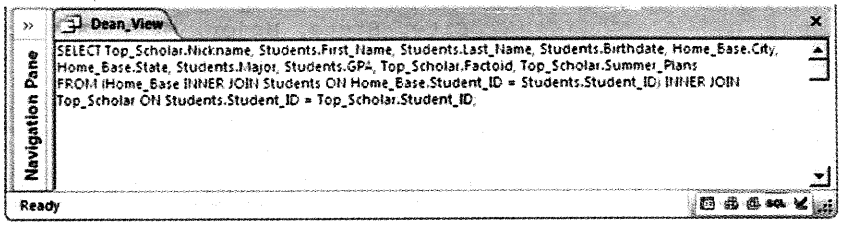


Figure 16.24 SQL query created for the Dean's View by the Query By Example data in Figure 16.22.

SUMMARY

In this chapter we followed a path from XML tagging through to the construction of logical views using QBE. You learned a lot, including:

- XML tags are an effective way to record metadata in a file.
- Metadata is used to identify values, can capture the affinity among values of the same entity, and can collect together a group of entity instances.
- Database tables have names and fields that describe the attributes of the entity contained in the table.
- The data that quantitatively records each property has a specific data type and is atomic.
- There are five fundamental operations on tables: `Select`, `Project`, `Union`, `Difference`, and `Product`. These operations are the only ones you need to create new tables from other database tables.
- `Join` is an especially useful operation that associates information from separate tables in new ways, based on matching fields.

- Relationships are the key to associating fields of the physical database.
- The physical database resides on the disk drive; it avoids storing data redundantly and is optimized for speed.
- The main approach for creating logical views from physical data is the join-and-trim technique.
- There is a direct connection between the theoretical ideas of database tables and the software of database systems.

Review Questions

Multiple Choice

1. If you know the structure and properties of data you can
   a. retrieve it
   b. organize it
   c. manage it
   d. all of the above
2. An important task when defining metadata is to
   a. identify the type of data
   b. normalize the data
   c. define the affinity of the data
   d. more than one of the above
3. Which of the following is an invalid XML tag?
   a. `<address>`
   b. `<stud ID>`
   c. `<cellPhone>`
   d. `<SSN>`
4. Which of the following is a valid XML tag?
   a. `<active?>`
   b. `<grad-date>`
   c. `<zip code>`
   d. `<DOB>`
5. The first tag in an XML document is known as a(n)
   a. metatag
   b. tree
   c. root element
   d. entity
6. An XML comment looks like
   a. `<!--Updated 09-26-07-->`
   b. `<! Updated 09-26-07 !>`
   c. `<" Updated 09-26-07 ">`
   d. `</ Updated 09-26-07>`

7. In database terminology, a set of entities refers to
   a. field
   b. column
   c. table
   d. information
8. The kind of information stored in a field in a database is described by the
   a. tuple
   b. field name
   c. data type
   d. record
9. A `Project` operation will
   a. return a table with as many rows as the original tables
   b. return only unique rows and merge duplicate rows
   c. automatically sort the list in alphabetical order by the first field
   d. all of the above
10. The *Test* in a `Select` command is used to
    a. add rows to an existing table
    b. remove rows from an existing table
    c. include rows in a new table
    d. describe rows in any table
11. Databases store data just once
    a. in order to avoid data redundancy
    b. because data storage is expensive
    c. because data access is slow
    d. all of the above

Short Answer

1. _____ is information describing other information.
2. XML is _____, that is, the tags create the structure of the data.
3. XML should be edited with a _____.
4. XML attributes must be enclosed in _____.
5. A(n) _____ is a group of related items in an XML document.
6. The rules for XML encodings are a hierarchical description called _____.
7. _____ describe the relationships among the different kinds of data.
8. A(n) _____ is used to ensure that all entities in a database are unique.
9. Data that cannot be decomposed into smaller parts is considered _____.
10. A(n) _____ is a collection of table definitions that give the name of the table, list of the attributes and their data types, and identifies the primary key.

11. A(n) _____ is a specification using the five operations and `join` that define a table from other tables.
12. A(n) _____ between two tables means that there is a corresponding row in one table for every row in the other table.

Exercises

1. Use XML to define your class schedule.
2. Create a list of IDs you have that could be considered primary keys in a database.
3. For the following, either indicate that the field is atomic or divide the field to make the result atomic.

Field	Contents
Phone	(212) 555-1212
Name	Maria Murray
Class	CSE 100
City	Seattle, WA
DOB	September 26, 1948

4. Take your class schedule from Exercise 1 and define it as a database table.
5. Define the attribute names, data types, and optional comments needed to create a table that could be used as a datebook.
6. Write an operation to display the Name and Interest from the `Nations` table for those countries with Beach, and store it in a table called `Vacation`.
7. Create tables that might exist with your student information on campus. Include such areas as Registrar, Bursar, Library, Financial Aid, Food Service, Residence Halls, Parking, and so forth.
8. Take a look at one of your monthly bills, such as the cable bill, phone bill or utility bill. What fields are used and what is their structure?
9. Using a text editor, create your own XML file containing CD or DVD information. Open the file in a browser.
10. Create a table with information from your driver's license.