

# Data Queries

## Chapter Outline

- Introduction, 187
- Two-Minute Chapter, 188
- Three Tasks of a Query Language, 189
- SQL SELECT Overview, 190
- Four Questions to Retrieve Data, 191
  - What Output Do You Want to See?*, 191
  - What Do You Already Know?*, 192
  - What Tables Are Involved?*, 192
  - How Are the Tables Joined?*, 193
- Sally's Pet Store, 195
- Vendor Differences, 196
- Query Basics, 196
  - Single Tables*, 197
  - Introduction to SQL*, 198
  - Sorting the Output*, 200
  - Distinct*, 200
  - Criteria*, 201
  - Pattern Matching*, 202
  - Boolean Algebra*, 204
  - DeMorgan's Law*, 206
  - Useful WHERE Clauses*, 208
- Computations, 209
  - Basic Arithmetic Operators*, 209
  - Aggregation*, 210
  - Functions*, 212
- Subtotals and GROUP BY, 214
  - Conditions on Totals (HAVING)*, 216
  - WHERE versus HAVING*, 216
  - The Best and the Worst*, 217
- Multiple Tables, 218
  - Joining Tables*, 219
  - Identifying Columns in Different Tables*, 220
  - Joining Many Tables*, 220
  - Hints on Joining Tables*, 222
  - Table Alias*, 223
  - Create View*, 224
- Newer Searches and Patterns, 226
  - XQuery*, 227
  - Regular Expressions (RegEx) Patterns*, 233
- Summary, 239
- Key Terms, 240
- Review Questions, 240
- Exercises, 241
- Web Site References, 247
- Additional Reading, 247
- Appendix: SQL Commands, 248

## What You Will Learn in This Chapter

- Why do you need a query language?
- What are the main tasks of a query language?
- What business questions can be answered with the basic SQL SELECT command?
- What is the basic structure of a query?
- What tables and columns are used in the Pet Store?
- How do you write queries for a specific DBMS?
- How do you create a basic query?
- What types of computations can be performed in SQL?
- How do you compute subtotals?
- How do you use multiple tables in a query?
- How do you search XML and complex text strings?

## A Developer's View

**Miranda:** Wow that was hard work! I sure hope normalization gets easier the next time.

**Ariel:** At least now you have a good database. What's next? Are you ready to start building the application?

**Miranda:** Not quite yet. I told my uncle that I had some sample data. He already started asking me business questions; for example, Which products were backordered most often? and Which employees sold the most items last month? I think I

need to know how to answer some of those questions before I try to build an application.

**Ariel:** Can't you just look through the data and find the answer?

**Miranda:** Maybe, but that would take forever. Instead, I'll use a query system that will do most of the work for me. I just have to figure out how to phrase the business questions as a correct query.

### Getting Started

Building basic SQL queries requires you to address four questions. (1) What output (columns and calculations) do you want to see? (2) What do you know or what constraints are given? (3) What tables are involved? (4) How are the tables joined? A powerful feature of SQL is the ease of computing subtotals with the GROUP BY statement. Learn the SELECT statement and use it as a fill-in-the-blanks model:

```
SELECT
FROM
INNER JOIN
WHERE
GROUP BY
HAVING
ORDER BY
```

## Introduction

**Why do you need a query language?** Why not just ask your question in a natural language like English? Natural language processors have improved, and several companies have attempted to connect them to databases. Similarly, speech recognition is improving. Eventually, computers may be able to answer ad hoc questions using a natural language. However, even if an excellent natural language processor existed, it still would be better to use a specialized query language. The main reason for the problem is communication. If you ask a question of a database, a computer, or even another person, you can get an answer. The catch is, did the computer give you the answer to the question you asked? In other words, you have to know that the machine (or other person) interpreted the question in exactly the way you wanted. The problem with any natural language is that it can be ambiguous. If there is any doubt in the interpretation, you run the risk of receiving an answer that might appear reasonable, but is not the answer to the question you meant to ask.

A query system is more structured than a natural language so there is less room for misinterpretation. Query systems are also becoming more standardized, so that developers and users can learn one language and use it on a variety of different systems. **SQL** is the standard database query language. The standard is established through the ISO (International Organization of Standards) and it is updated every few years. Most database management systems implement most of the SQL 2003 standard. The draft SQL 2006 standard adopted as SQL 2008 provides definitions for several important programming concepts and for XML, but most DBMS vendors have continued to use their existing, proprietary definitions. Consequently, although these standards are accepted by most vendors, there is still room for variations in the SQL syntax, so queries written for one database system will not always work on another system. SQL 2011 added a few new elements to the features added in 2008. Plus it initiated new definitions for handling time elements.

Most database systems also provide a **query by example (QBE)** method or query builder to help beginners create SQL queries. These visually oriented tools generally let users select items from lists, and handle the syntax details to make it easier to create ad hoc queries. Although the QBE designs are easy to use and save time by minimizing typing, you must still learn to use the SQL commands. Many times, you will have to enter SQL into programming code or copy and edit SQL statements.

As you work on queries, you should also think about the overall database design. Chapter 3 shows how normalization is used to split data into tables that can be stored efficiently. Queries are the other side of that problem: They are used to put the tables back together to answer ad hoc questions and produce reports.

The first two sections of this chapter provide an overview of queries. The sections beginning with the Pet Store provide a more detailed explanation of how to build queries, starting with a single table and basic criteria issues. Overall, this chapter covers the basic features of the SELECT statement. The focus is on learning the key SELECT clauses and on translating business questions into SQL queries. Chapter 5 covers more complex business questions that utilize some of the more complex and more powerful features of SQL.

## Two-Minute Chapter

---

Relational databases were initially created to store large amounts of data efficiently. Putting data into separate tables reduces duplication and simplifies adding new rows of data. For example, new sales can be added without interfering with existing sales or even added by multiple people at the same time. But, you might be wondering, how is it possible to retrieve this data? For example, the Customer data was split from the Sales data and stored in separate tables, linked by CustomerID. Sure, a person could retrieve a row from the Sales table, get the CustomerID value and then go look up the matching data in the Customer table, but that seems painful.

The SQL query language was created to answer these questions. The foundations of the SELECT command are covered in this chapter. One strength of SQL is that it is a declarative language instead of procedural. In a typically procedural programming language, you would have to write code to open a table, and loop through all of the rows to find the ones you want. With SQL, you simply tell (declare to) the DBMS what you want to see. You specify (1) the columns you want displayed, (2) the conditions you want to apply, (3) the tables involved, and

(4) how the tables are connected. Retrieving Sales and Customers in March is a simple as:

```
SELECT Customer.CID, Lastname, SaleDate
FROM Customer
INNER JOIN Sale ON Customer.CID = Sale.CID
WHERE (SaleDate BETWEEN '3/1/2013' AND '3/31/2013');
```

SQL can also handle basic calculations such as price \* quantity. More importantly, it can compute totals with a simple function: Sum(price\*quantity). Totals are computed across all of the specified rows. But SQL also computes subtotals for any level of grouping using the GROUP BY statement. A critical goal of this chapter is to be able to read business questions and write the matching SQL statement—particularly for computing subtotals. For instance, to find the best employee for the month of March, use:

```
SELECT Employee.EID, Lastname, Sum(Price*Quantity) As
TotalSales
FROM Employee
INNER JOIN Sale ON Employees.EID = Sales.EID
INNER JOIN SaleItem ON Sales.SaleID = SaleItems.SaleID
WHERE (SaleDate BETWEEN '3/1/2013' AND '3/31/2013')
GROUP BY Employee.EID, Lastname
ORDER BY Sum(Price*Quantity) DESC;
```

Query editors can be used to drag-and-drop tables and columns to create the JOINS and enter conditions. But you should learn the basic elements of the SELECT command so you can type them by hand when necessary.

### Three Tasks of a Query Language

---

**What are the main tasks of a query language?** To create databases and build applications, you need to perform three basic sets of tasks: (1) define the database, (2) change the data, and (3) retrieve data. Some systems use formal terms to describe these categories. Commands grouped as **data definition language (DDL)** are used to define the data tables and other features of the database. The common DDL commands include: ALTER, CREATE, and DROP. Commands used to modify the data are classified as **data manipulation language (DML)**. Common DML commands are: DELETE, INSERT, and UPDATE. Some systems include data retrieval within the DML group, but the SELECT command is complex enough to require its own discussion. The appendix to this chapter lists the syntax of the various SQL commands. Virtually all tasks can be performed by issuing a DDL, DML, or query command. This chapter focuses on the SELECT command. The DML and DDL commands will be covered in more detail in Chapter 5.

The SELECT command is used to retrieve data: It is the most complex SQL command, with several different options. The main objective of the SELECT command is to retrieve specified columns of data for rows that meet some criteria. Database management systems are driven by query systems. Many query systems support a graphical interface which makes it easier to create queries by reducing typing and through visualizing the relationships among tables. But, ultimately you should learn the text versions of the SQL commands.

What output do you want to see?	SELECT columns
What tables are involved?	FROM table
How are the tables joined?	INNER JOIN table
What do you already know (or what constraints are given)?	WHERE conditions

**Figure 4.1**

Four questions to create a query. Every query is built by asking these four questions. The SELECT... FROM ... INNER JOIN ... WHERE ... syntax is the SQL form to creating a query.

## SQL SELECT Overview

**What business questions can be answered with the basic SQL SELECT command?** For the most part, SQL is a declarative language, which is unlike traditional programming procedural languages. You simply have to tell the DBMS what you want and it determines how to get that data. You do not have to write loops or conditional statements. The SELECT command is the primary method of retrieving data from tables. This chapter focuses on its basic elements: (1) choosing columns and making basic calculations, (2) selecting rows of data based on given information, (3) joining related tables, (4) sorting the results, and (5) computing subtotals. Many business questions rely on these basic tools.

At the simplest level, business questions just need to retrieve data that matches some basic conditions. Questions such as: List customers who made a purchase in a specified month or bought a specific product; or Find employees who sold items to a specific customer. Building a query to answer these basic questions just involves identifying the tables that hold the desired data, selecting the desired columns to display, and entering the specified filter conditions, and perhaps sorting the results. It is critical that you learn to build these simple queries correctly.

As a small step up, the SELECT statement can also perform simple computations. Business problems often need to multiply values in two columns, such as Price\*Quantity, or to subtract two values. The SELECT statement handles basic arithmetic as well as common mathematical and string functions. These calculations operate on data held in one row and follow standard rules for mathematical operations.

The most important capability of SQL in basic business questions is to compute subtotals. A surprising number of business questions involve subtotals. For instance: Which customer spent the most money last month? Which employee sold the most items last week? Which product category is the best seller? What are total sales by month? SQL easily handles subtotals with the GROUP BY clause. Simply list the column that contains the items to break (group) on, then include an aggregate function (usually Sum, Avg, Count) as a computation. The DBMS will find each unique value in the GROUP BY column (such as each employee), then compute the subtotal indicated in the Sum function for each item. Sorting the results makes it easy to find the highest or lowest value. To convert a business question into SQL, you often identify the items to be summed (or counted) and then determine which columns hold the grouping values. Examine the business question and look for words such as “by” or “for each.”

## Four Questions to Retrieve Data

**What is the basic structure of a query?** Every attempt to retrieve data from a relational DBMS requires answering the four basic questions listed in Figure 4.1. The difference among query systems is how you fill in those answers. The figure also shows the matching SQL clauses for answering the questions.

Notice that in some easy situations you will not have to answer all four questions. Many easy questions involve only one table, so you will not have to worry about joining tables. As another example, you might want the total sales for the entire company, as opposed to the total sales for a particular employee, so there may not be any constraints.

The SELECT statements can be used as a fill-in-the-blanks type of form. Start the query by writing down or typing those key words on the left side of the page. Then fill in the items to the right of the keywords. It is often easiest to begin by writing down the output that you want to see on the SELECT statement, followed by the constraints on the WHERE clause. Then you can identify all of the tables needed and use the relationship diagram to see how the tables are joined.

### What Output Do You Want to See?

As an initial step, you can think of a query as a way to filter data—both in terms of columns you want to see and limiting the rows based on various conditions. You could just retrieve every column from a table, but it gets hard to wade through columns that are not important to the business question. So you need to tell the DBMS which columns you want to see. More importantly, you first have to visualize your output before you can write the rest of the query. In general, a query system answers your query by displaying rows of data from various columns. You

**Figure 4.2**

Simple example of a column filter. Use the SELECT clause to choose only the columns or calculations you want to see.

EmployeeID	LastName	Phone
1	Reeves	402-146-7714
2	Gibson	919-245-0526
3	Reasoner	413-414-8275
4	Hopkins	412-524-0814
5	James	407-026-6653
6	Eaton	906-446-7957
7	Farris	615-891-5545
8	Carpenter	212-545-8897
9	O'Conner	203-180-0146
10	Shields	304-607-9081
11	Smith	80-333-9872

```
SELECT EmployeeID, LastName, Phone
FROM Employee
```

EmployeeID	LastName	Phone	EmployeeLevel
4	Hopins	412-524-9814	3
5	James	407-026-6653	3
7	Farris	615-891-5545	3

```
SELECT EmployeeID, LastName, phone, EmployeeLevel
FROM Employee
WHERE EmployeeLevel=3;
```

Figure 4.3

Simple example of a row filter. The WHERE clause limits the rows to be displayed based on multiple conditions. Connecting conditions with an “OR” statement returns more rows in the results because each row could meet on many conditions.

can also ask the DBMS to perform some basic computations, so you also need to identify any calculations and totals you need.

You generally answer this question by selecting columns of data from the various tables stored in the database. Of course, you need to know the names of all of the columns to answer this question. Generally, the hardest part in answering this question is to wade through the list of tables and identify the columns you really want to see. The problem is more difficult when the database has hundreds of tables and thousands of columns. Queries are easier to build if you have a copy of the class diagram that lists the tables, their columns, and the relationships that join the tables.

Figure 4.2 shows a simple example of a SELECT statement using the Employee table in the Pet Store database. The SELECT clause specifies the columns or calculations you want to see. You can think of it as a column filter—choosing a subset of the columns available in the tables. If you want to see all of the columns, you can simply use SELECT \* FROM Employee to show all the columns.

### What Do You Already Know?

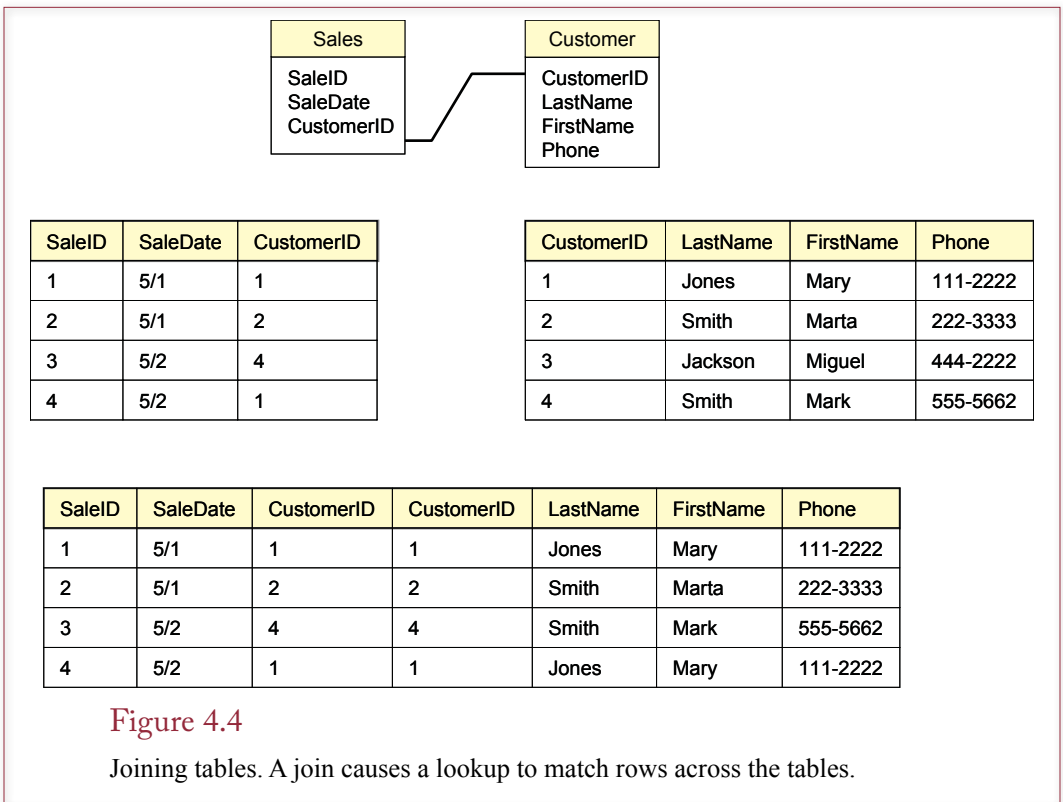
In most situations you want to restrict your search based on various criteria. For instance, you might be interested in sales on a particular date or sales from only one department. The search conditions must be converted into a standard Boolean notation (phrases connected with AND or OR). The most important part of this step is to write down all the conditions to help you understand the purpose of the query.

Figure 4.3 shows that you can think of the WHERE clause as a filtering statement. Rows are displayed in the results only if the data within the row matches the conditions in the WHERE clause. If multiple conditions are connected with an “OR” term, the query generally returns more rows because the data can match any one of the conditions. Connecting the conditions with an “AND” term reduces the number of rows because each row must match all of the conditions.

### What Tables Are Involved?

With only a few tables, this question is easy. With hundreds of tables, it could take a while to determine exactly which ones you need. A good data dictionary with synonyms and comments will make it easier for you (and users) to determine





exactly which tables you need for the query. It is also critical that tables be given names that accurately reflect their content and purpose.

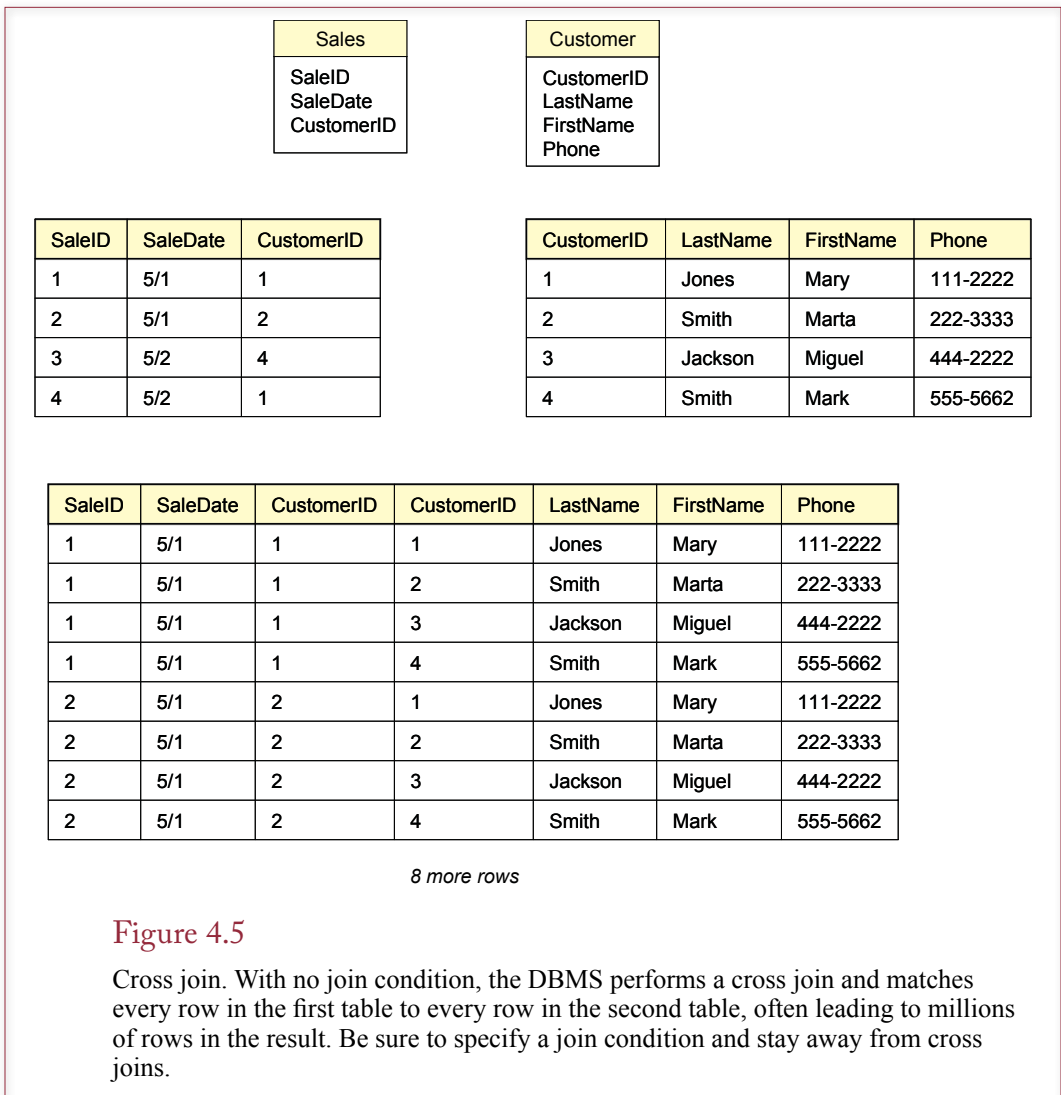
One hint in choosing tables is to start with the tables containing the columns listed in the first two questions (output and criteria). Next decide whether other tables might be needed to serve as intermediaries to connect these tables.

### How Are the Tables Joined?

This question relates to the issues in data normalization and is the heart of a relational database. Tables are connected by data in similar columns. For instance, as shown in Figure 4.4, a Sales table has a CustomerID column. Corresponding data is stored in the Customer table, which also has a CustomerID column. In many cases matching columns in the tables will have the same name (e.g., CustomerID) and this question is easy to answer. The join performs a matching or lookup for the rows. You can think of the result as one giant table and use any of the columns from any of the joined tables. Note that columns are not required to have the same name, so you sometimes have to think a little more carefully. For example, an Order table might have a column for SalesPerson, which is designed to match the EmployeeID key in an Employee table.

Joining tables is usually straightforward as long as your database design is sound. In fact, most QBE systems will automatically use the design to join any tables you add. However, two problems can arise in practice: (1) You should verify that all tables are joined, and (2) Double-check any tables with multiple join conditions.





Technically, it is legal to use tables without adding a join condition. However, when no join condition is explicitly specified, the DBMS creates a **cross join** or Cartesian product between the tables. A cross join matches every row in the first table to every other row in the second table. For example, if both tables have 10 rows, the resulting cross join yields  $10 \times 10 = 100$  rows of data. If the tables each have 1,000 rows, the resulting join has one million rows! A cross join will seriously degrade performance on any DBMS, so be sure to specify a join condition for every table. The one exception is that it is sometimes used to join a single-row result with every row in a second table. With only one row in a table, the cross join is reasonably fast. Figure 4.5 shows the results of a cross join using two small tables.

Sometimes table designs have multiple relationship connections between tables. For example, the Pet Store database joins Customer to City and City to Employee. A query system that automatically adds relationship joins will bring along

every connection. But, you rarely want to use all of the joins at the same time. The key is to remember that a join represents a restrictive condition. In the Pet Store case, if you include the two joins from the Customer, City, and Employee tables, you would be saying that you only want to see customers who live in the same city as an employee.

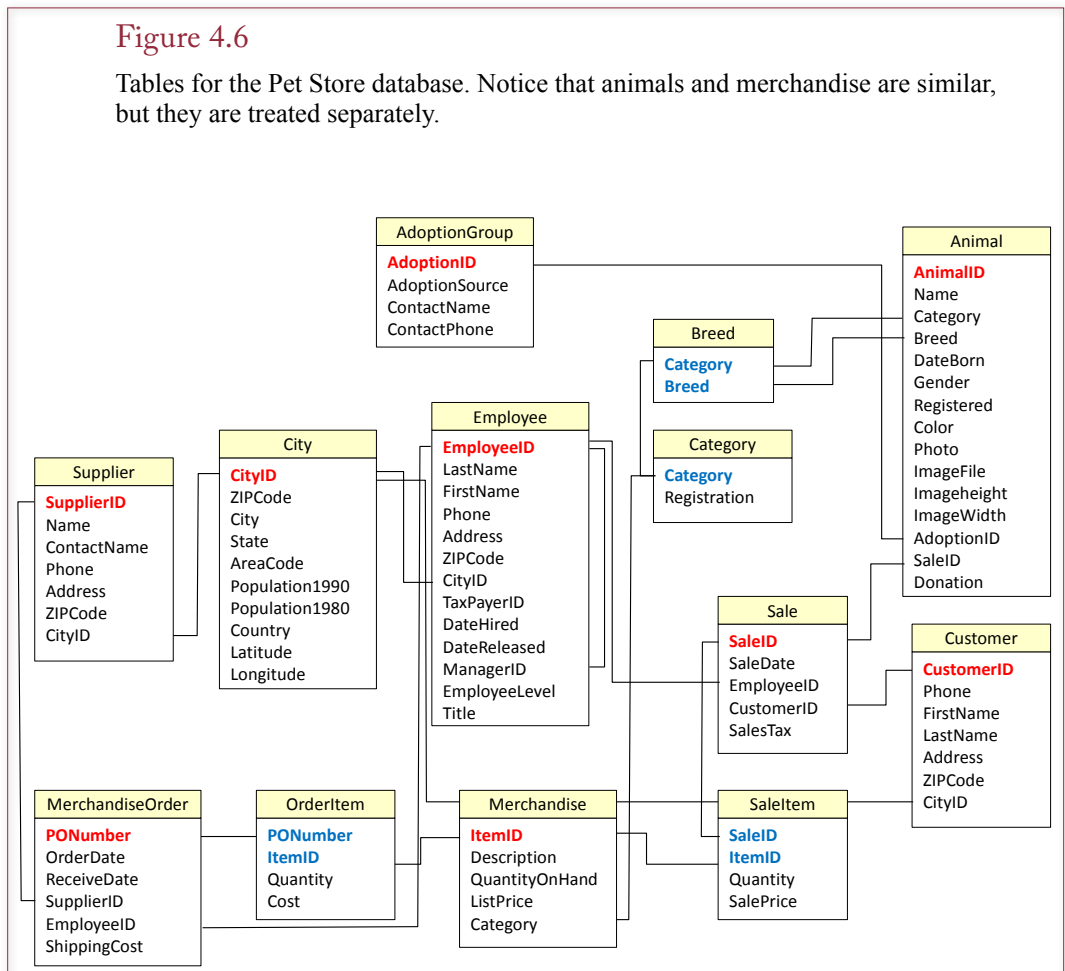
## Sally's Pet Store

**What tables and columns are used in the Pet Store?** The initial Pet Store database has been built, and some basic historical data has been transferred from Sally's old files. When you show your work to Sally, she becomes very excited. She immediately starts asking questions about her business, and wants to see how the database can answer them.

The examples in this chapter are derived from the Pet Store database. The tables and relationships for this case are shown in Figure 4.6. After reading each section, you should work through the queries on your own. You should also solve the exercises at the end of the chapter. Queries always look easy when the answers are printed in the book. To learn to write queries, you must sit down and struggle through the process of answering the four basic questions.

Figure 4.6

Tables for the Pet Store database. Notice that animals and merchandise are similar, but they are treated separately.



Chapter 3 notes that data normalization results in a business model of the organization. The list of tables gives a picture of how the firm operates. Notice that the Pet Store treats merchandise differently than it treats animals. For example, each animal is listed separately on a sale, but customers can purchase multiple copies of merchandise items (e.g., bags of cat food). The reason for the split is that each animal is unique and can be adopted only once. Also, you need to keep additional information about the animals that does not apply to general merchandise.

When you begin to work with an existing database, the first thing you need to do is familiarize yourself with the tables and columns. You should look through some of the main tables to become familiar with the type and amount of data stored in each table. Make sure you understand the terminology and examine the underlying assumptions. For example, in the Pet Store case, an animal might be registered with a breeding agency, but it can be registered with only one agency. If it is not registered, the Registered column is **NULL** (or missing) for that animal. This first step is easier when you work for a specific company, since you should already be familiar with the firm's operations and the terms that it uses for various objects.

## Vendor Differences

---

**How do you write queries for a specific DBMS?** The SQL standards present a classic example of software development trade-offs. New releases of the standards provide useful features, but vendors face the need to maintain compatibility with a large installed base of applications and users. Consequently, substantial differences exist across database products. These differences are even more pronounced when you look at the graphical interfaces.

Whenever possible, you should use the newer standards because the queries are easier to read. However, it is likely that you will encounter queries written in the older syntax, so you should also learn how to read these older versions. The one catch is that each DBMS vendor had its own proprietary syntax. It is impossible to cover all of the variations in this book. The details of the syntax and the basic steps for writing and testing queries within a DBMS are explained in the accompanying workbooks. Each workbook explores the same issues using a single DBMS. At a minimum, you should read through and work the examples in one workbook. If you have time, it is instructive to compare the techniques of several vendors.

## Query Basics

---

**How do you create a basic query?** The basic goal is to convert a business question into a database query. It is best to begin with relatively easy queries. This chapter first presents queries that involve a single table to show the basics of creating a query. Then it covers details on constraints, followed by a discussion on computations and aggregations. Groups and subtotals are then explained. Finally, the chapter discusses how to select data from several tables at the same time.

Figure 4.7 presents several business questions that might arise at the Pet Store. Most of the questions are relatively easy to answer. In fact, if there are not too many rows in the Animal table, you could probably find the answers by hand-searching the table. Actually, you might want to work some of the initial questions by hand to help you understand what the query system is doing.

- Which animals were born after August 1?
- List the animals by category and breed.
- List the categories of animals that are in the Animal list.
- Which dogs have a donation value greater than \$250?
- Which cats have black in their color?
- List cats excluding those that are registered or have red in their color.
- List all dogs who are male and registered or who were born before 01-June-2013 and have white in their color.
- What is the extended value (price \* quantity) for sale items on sale 24?
- What is the average donation value for animals?
- What is the total value of order number 22?
- How many animals were adopted in each category?
- How many animals were adopted in each category with total adoptions of more than 10?
- How many animals born after June 1 were adopted in each category with total adoptions more than 10?
- List the CustomerID of everyone who bought or adopted something between April 1, 2013 and May 31, 2013.
- List the names of everyone who bought or adopted something between April 1, 2013 and May 31, 2013.
- List the name and phone number of anyone who adopted a registered white cat between two given dates.

Figure 4.7

Sample questions for the Pet Store. Most of these are easier since they involve only one table. They represent typical questions that a manager or customer might ask.

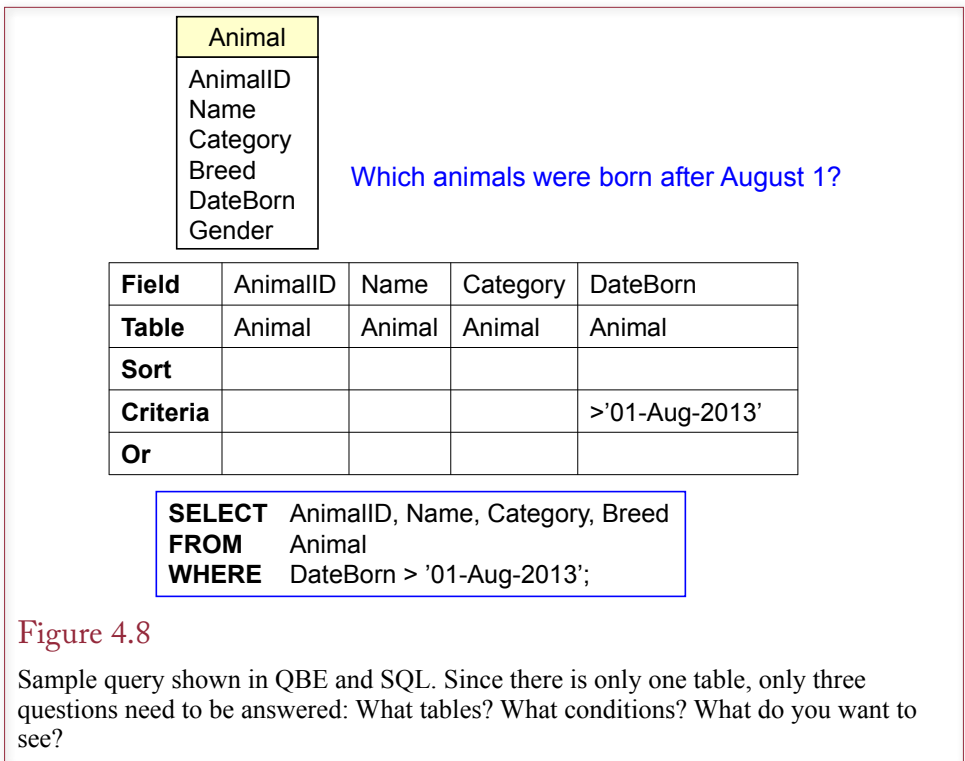
The foundation of queries is that you want to see only some of the columns from a table and that you want to restrict the output to a set of rows that match some criteria. For example, in the first query (animals with yellow color), you might want to see the AnimalID, Category, Breed, and their Color. Instead of listing every animal in the table, you want to restrict the list to just those with a yellow color.

### Single Tables

The first query to consider is: *Which animals were born after August 1?* Figure 4.8 shows a QBE approach and the SQL. The two methods utilize the same underlying structure. The QBE approach saves some typing, but eventually you need to be able to write the SQL statements. If you write down the SQL keywords, you can fill in the blanks—similar to the way you fill in the QBE grid.

First consider answering this question with a QBE system. The QBE system will ask you to choose the tables involved. This question involves only one table: Animal. You know that because all of the data you want to see and the constraint are based on columns in the Animal table. With the table displayed, you can now choose which columns you want to see in the output. The business question is a little vague, so select AnimalID, Name, Category, and DateBorn.

The next step is to enter the criteria that you already know. In this example, you are looking for animals born after a specific date. On the QBE grid, enter the condition `>'01-Aug-2013'` on the Criteria row under the DateBorn column. There is one catch: Different DBMSs use different syntax for the way you enter the date. Most of them will accept the date format and the single quotes shown here. For



Microsoft Access, do not include any quotation marks. The Access QBE interface will automatically add # marks instead. It is a good idea to run the query now. Check the DateBorn result to ensure that the query was entered correctly.

The four basic questions are answered by filling out blanks on the QBE grid. (1) The output to be displayed is placed as a field on the grid. (2) The constraints are entered as criteria or conditions under the appropriate fields. (3) The tables involved are displayed at the top (and often under each field name). (4) The table joins are shown as connecting lines among the tables. The one drawback to QBE systems is that you have to answer the most difficult question first: Identifying the tables involved. The QBE system uses the table list to provide a list of the columns you can choose. Keep in mind that you can always add more tables as you work on the problem.

### Introduction to SQL

SQL is a powerful query language. However, unlike QBE, you generally have to type in the entire statement. Most systems enable you to switch back and forth between QBE and SQL, which saves some typing. Perhaps the greatest strength of SQL is that it is a standard that most vendors of DBMS software support. Hence, once you learn the base language, you will be able to create queries on all of the major systems in use today. Note that some people pronounce SQL as “sequel,” arguing that it descended from a vendor’s early DBMS called quel. Also, “Sequel” is easier to say than “ess-cue-el.” But with the introduction of CQL for Cassandra (see Chapter 13) it will be safer to just say SQL.

The most commonly used command in SQL is the SELECT statement, which is used to retrieve data from tables. A simple version of the command is shown

SELECT	columns	What do you want to see?
FROM	tables	What tables are involved?
JOIN	conditions	How are the tables joined?
WHERE	criteria	What are the constraints?

Figure 4.9

The basic SQL SELECT command matches the four questions you need to create a query. The uppercase letters are used in this text to highlight the SQL keywords. They can also be typed in lowercase.

in Figure 4.9, which contains the four basic parts: **SELECT**, **FROM**, **JOIN**, and **WHERE**. These parts match the basic questions needed by every query. In the example in Figure 4.8, notice the similarity between the QBE and SQL approaches. The four basic questions are answered by entering items after each of the four main keywords. When you write SQL statements, it is best to write down the keywords and then fill in the blanks. You can start by listing the columns you want to see as output, then write the constraints in the WHERE clause. By looking at the columns you used, it is straightforward to identify the tables involved. You can use the class diagram to understand how the tables are joined.

Figure 4.10

The ORDER BY clause sorts the output rows. The default is to sort in ascending order, adding the keyword DESC after a column name results in a descending sort. When columns like Category contain duplicate data, use a second column (e.g., Breed) to sort the rows within each category.

Animal
AnimalID
Name
Category
Breed
DateBorn
Gender

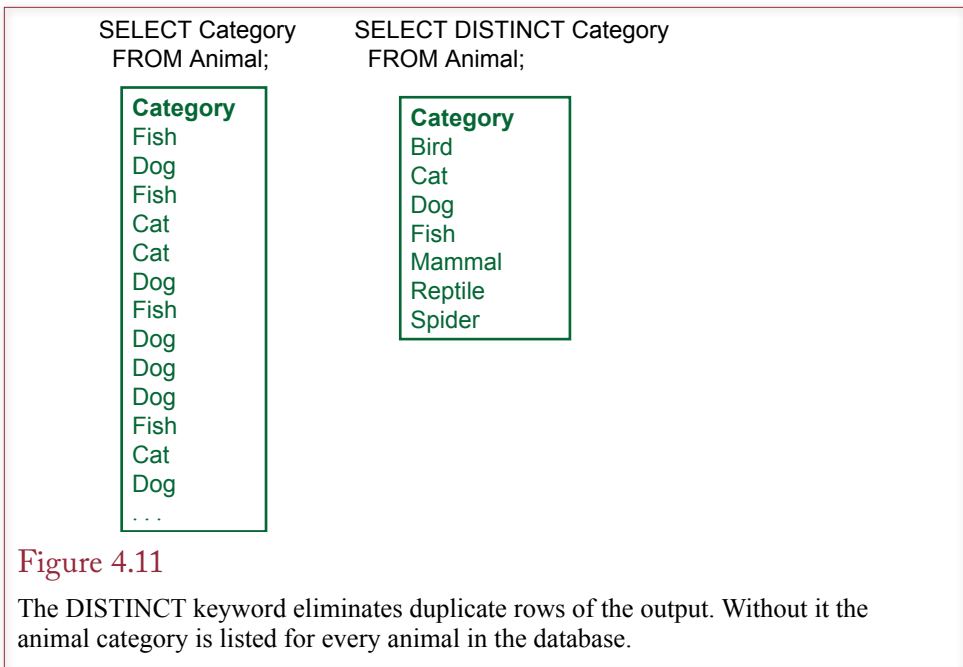
Field	Name	Category	Breed
Table	Animal	Animal	Animal
Sort		Ascending	Ascending
Criteria			
Or			

Name	Category	Breed
Cathy	Bird	African Grey
	Bird	Canary
Debbie	Bird	Cockatiel
	Bird	Cockatiel
Terry	Bird	Lovebird
	Bird	Other
Charles	Bird	Parakeet
Curtis	Bird	Parakeet
Ruby	Bird	Parakeet
Sandy	Bird	Parrot
Hoyt	Bird	Parrot
	Bird	Parrot

```

SELECT Name, Category, Breed
FROM Animal
ORDER BY Category, Breed;

```



## Sorting the Output

Database systems treat tables as collections of data. For efficiency the DBMS is free to store the table data in any manner or any order that it chooses. Yet in most cases you will want to display the results of a query in a particular order. The SQL **ORDER BY** clause is an easy and fast means to display the output in any order you choose. As shown in Figure 4.10, simply list the columns you want to sort. The default is ascending (A to Z or low to high with numbers). Add the phrase **DESC** (for descending) after a column to sort from high to low. In QBE you select the sort order on the QBE grid.

In some cases you will want to sort columns that do not contain unique data. For example, the rows in Figure 4.10 are sorted by Category. In these situations you would want to add a second sort column. In the example, rows for each category (e.g., Bird) are sorted on the Breed column. The column listed first is sorted first. In the example, all birds are listed first, and birds are then sorted by Breed. To change this sort sequence in QBE, you have to move the entire column on the QBE grid so that Category is to the left of Breed.

## Distinct

The **SELECT** statement has an option that is useful in some queries. The **DISTINCT** keyword tells the DBMS to display only rows that are unique. For example, the query in Figure 4.11 (*SELECT Category FROM Animal*) would return a long list of animal types (Bird, Cat, Dog, etc.). In fact, it would return the category for every animal in the table—obviously; there are many cats and dogs. To prevent the duplicates from being displayed, use the **SELECT DISTINCT** phrase.

Note that the **DISTINCT** keyword applies to the entire row. If there are any differences in a row, it will be displayed. For example, the query *SELECT DISTINCT Category, Breed FROM Animal* will return more than the seven rows shown in



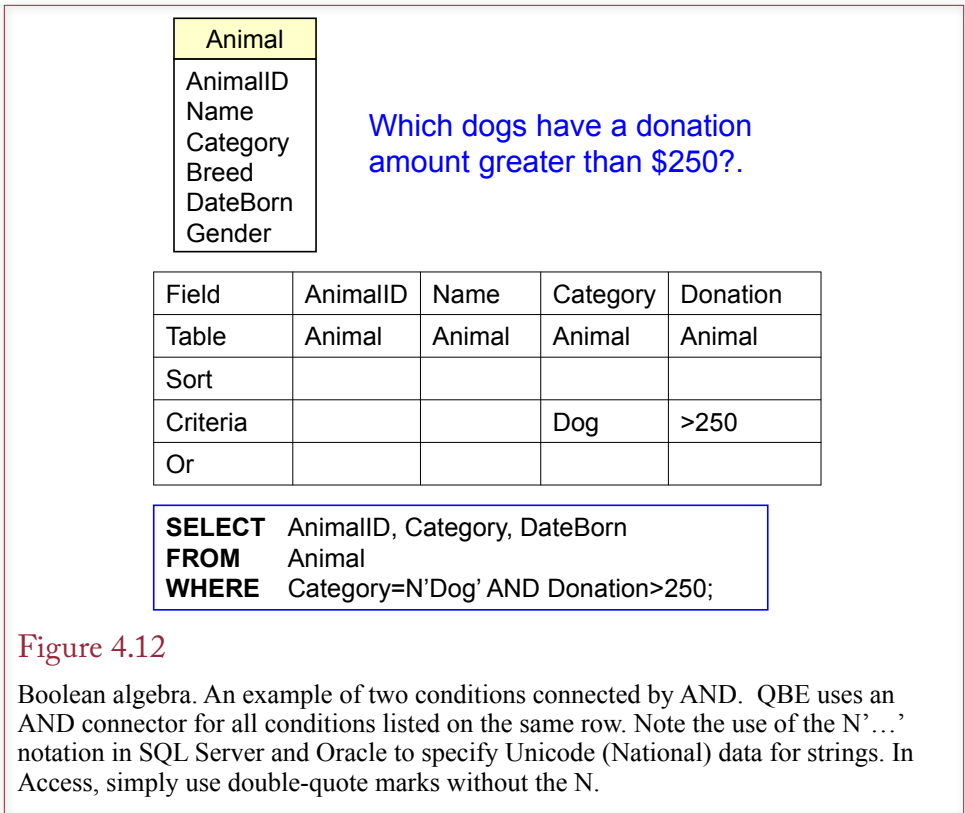


Figure 4.11 because each category can have many breeds. That is, each category/breed combination will be listed only once, such as Dog/Retriever. Microsoft Access supports the **DISTINCT** keyword, but you have to enter it in the SQL statement.

### Criteria

In most questions, identifying the output columns and the tables is straightforward. If there are hundreds of tables, it might take a while to decide exactly which tables and columns you want, but it is just an issue of perseverance. On the other hand, identifying constraints and specifying them correctly can be more challenging. More importantly if you make a mistake on a constraint, you will still get a result. The problem is that it will not be the answer to the question you asked—and it is often difficult to see that you made a mistake.

The primary concept of constraints is based on **Boolean algebra**, which you learned in mathematics. In practice, the term simply means that various conditions are connected with **AND** and **OR** clauses. Sometimes you will also use a **NOT** statement, which negates or reverses the truth of the statement that follows it. For example, **NOT (Category = N'Dog')** means you are interested in all animals except dogs.

Consider the example in Figure 4.12. The first step is to note that two conditions define the business question: dog and donation. The second step is to recognize that both of these conditions need to be true at the same time, so they are connected by **AND**. As the database system examines each row, it evaluates both

clauses. If any one clause is false, the row is skipped. Notice the use of N'Dog' statement which converts the entered text to Unicode (national) format. Because the data table formats were defined as Unicode, it is best to enter this specification whenever you write a query. If the text in the query processor uses English, the conversion is usually automatic and you could skip the N prefix. However, it is always safer to specify the conversion explicitly.

Notice that the SQL statement is straightforward—just write the two conditions and connect them with an AND clause. The QBE is a little trickier. With QBE, every condition listed on the same criteria row is connected with an AND clause. Conditions on different criteria rows are joined with an OR clause. You have to be careful creating (and reading) QBE statements, particularly when there are many different criteria rows.

## Pattern Matching

Databases are designed to handle many different types of data, including numbers, dates, and text. The standard comparison operators (<, >, =, and so on) work well for numbers, dates, and simple text values. However, larger text fields often require more powerful comparisons. The SQL standard provides the **LIKE** command to handle simple pattern matching tasks. The LIKE command uses two special characters to create a pattern that is compared to each selected row of text. In standard SQL, the percent sign (%) in a pattern matches any character or characters (including none). The underscore ( \_ ) matches exactly one character. Before exploring patterns, note that Microsoft Access uses an asterisk (\*) and question mark (?) instead. Access does provide the option to use the standard percent sign and underscore characters, but almost no one activates that option.

You construct a pattern by using the percent or underscore characters. Generally, you want to search for a specific word or phrase. Consider the request from a customer who wants a black cat. If you look at the Color column of the Animal table, you will see that can contain multiple colors for any animal. If you think about animals for a minute, it is clear that an animal can have multiple colors. Technically, this choice means that the Color column is probably not atomic; and you could have specified a completely new table that lists each color on a separate line. However, color definitions are somewhat subjective, and it is more complicated to enter data and write queries when multiple tables are involved. Consequently, the database is a little more usable by listing the colors in a single column. But, now you have to search it. If you search using the equals sign (say, WHERE Color=N'Black'), you will see only animals that are completely black. Perhaps the customer is willing to settle for a cat that has a few white spots, which might have been entered as Black/White; and will not show up in the simple equality search.

The answer is to construct a pattern search that will list a cat that has the word Black anywhere in the Color column. Figure 4.13 shows the syntax of the query. The key is the phrase: Color LIKE N'%Black%'. Placing a percent sign at the start of the pattern means that any characters can appear before the word Black. Placing a percent sign at the end of the pattern means that any characters can appear after the word Black. Consequently, if the word Black appears anywhere in the color list, the LIKE condition will be true. Note that the simple color "Black" will also be matched because the percent sign matches no characters. If you leave off the first percent sign (Color LIKE N'Black%'), the condition would be true only if the Color column begins with the word Black (followed by anything else).

Animal
AnimalID
Name
Category
Breed
DateBorn
Gender

Which cats have black in their color?

Field	AnimalID	Name	Category	Color
Table	Animal	Animal	Animal	Animal
Sort				
Criteria			'Cat'	LIKE '%Black%'
Or				

```

SELECT AnimalID, Name, Category, Color
FROM Animal
WHERE Category='Cat' AND Color LIKE '%Black%';

```

**Figure 4.13**

Pattern matching. The percent sign matches any characters, so if the word Black appears anywhere in the Color column the LIKE condition is true.

You can construct more complex conditions using pattern matching, but you should test these patterns carefully. For instance, you could search a Comment column for two words using: `Comment LIKE N'%friendly%children%'`. This pattern will match any row that has a comment containing both of the words (friendly and children). There can be other words in front of, behind, or between the two words, but they must appear in the order listed.

You can also use the single character matching tool (underscore) to create a pattern. This tool is useful in certain situations. It is most useful when you have a text column that is created with a particular format. For instance, most automobile license plates follow a given pattern (such as AAA-999). If a policeman gets a partial license plate number, he could search for matches. For instance, `License LIKE N'XQ_-12_'`, would search for plates where the third character and third number are not known. Keep in mind that the single-character pattern will only match a character that exists. In the example, if a license number has three letters but only two numbers, the pattern will never match it because the pattern requires a third number. In business, the single-character pattern is useful for searching product codes that contain a fixed format. For instance, a department store might identify products by a three-character department code, a two-character color code, a two-digit size code, and a five-digit item code: `DDDCC11-12345`. If you wanted to find all blue (BL) items of size 9 (09), you could use: `ItemCode LIKE N'___BL09-____'`. Note that spaces were added in the text to show the number of underscores, but you would need to enter the underscores into the query without any intervening spaces.

a	b	a AND b	a OR b
T	T	T	T
T	F	F	T
F	T	F	T
F	F	F	F

Figure 4.14

A truth table shows the difference between AND and OR. Both clauses must be true when connected by AND. Only one clause needs to be true when clauses are connected by OR.

## Boolean Algebra

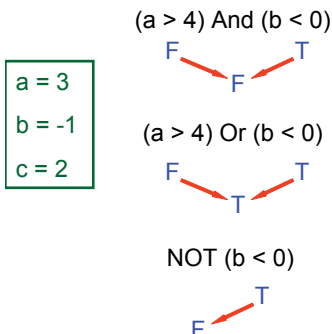
One of the most important aspects of a query is the choice of rows that you want to see. Most tables contain a huge number of rows, and you want to see only the few that meet a business condition. Some conditions are straightforward. For example, you might want to examine only dogs. Other criteria are complex and involve several conditions. For instance, a customer might want a list of all yellow dogs born after June 1, 2013, or registered black labs. Conditions are evaluated according to Boolean algebra, which is a standard set of rules for evaluating conditions. You are probably already familiar with the rules from basic algebra courses; however, it pays to be careful.

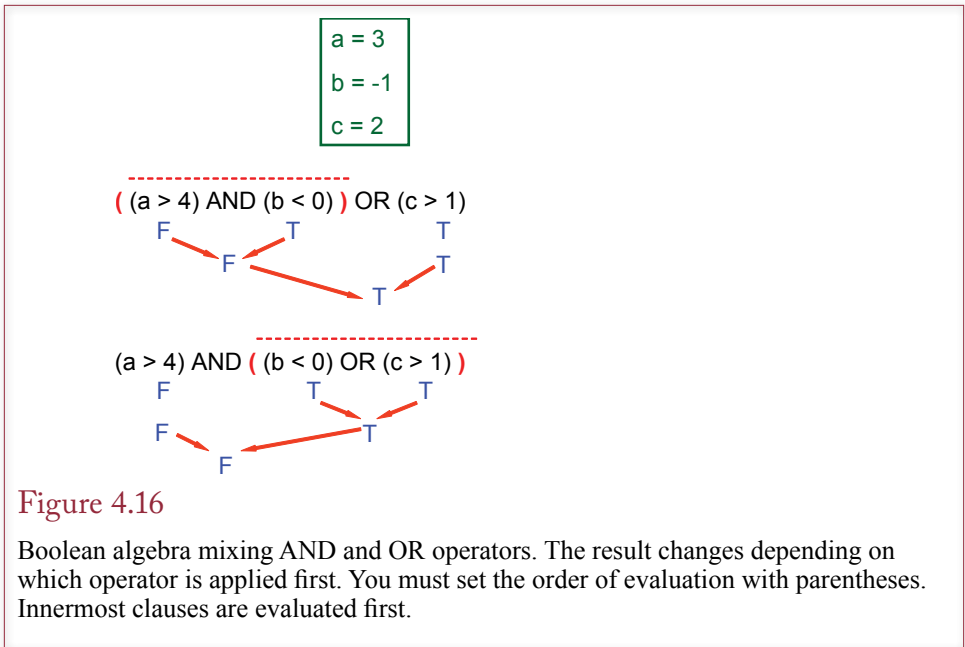
The DBMS uses Boolean algebra to evaluate conditions that consist of multiple clauses. The clauses are connected by these operators: AND, OR, NOT. Each individual clause is evaluated as true or false, and then the operators are applied to evaluate the truth value of the overall criterion. Figure 4.14 shows how the primary operators (AND, OR) work. The DBMS examines each row of data and evaluates the Boolean condition. The row is displayed only if the condition is true.

A condition consisting of two clauses connected by AND can be true only if both of the clauses (a And b) are true. A statement that consists of two clauses connected by OR is true as long as at least one of the two conditions is true. Consider the examples shown in Figure 4.15. The first condition is false because it

Figure 4.15

Boolean algebra examples. Evaluate each clause separately. Then evaluate the connector. The NOT operator reverses the truth value.





asks for both clauses to be true, and the first one is false ( $a < 4$ ). The second example is true because it requires only that one of the two clauses be true. Consider an example from the Pet Store. If a customer asks to see a list of yellow dogs, he or she wants a list of animals where the category is Dog AND the color is yellow.

As shown in Figure 4.16, conditions that are more complex can be created by adding additional clauses. A complication arises when the overall condition contains both AND connectors and OR connectors. In this situation the resulting truth value depends on the order in which the clauses are evaluated. You should always use parentheses to specify the desired order. Innermost parentheses are evaluated first. In the example at the top of Figure 4.16, the AND operation is performed before the OR operation, giving a result of true. In the bottom example, the OR connector is evaluated first, leading to an evaluation of false.

If you do not use parentheses, the operators are evaluated from left to right. This result may not be what you intended, yet the DBMS will still provide a response. To be safe, you should build complex conditions one clause at a time. Check the resulting selection each time to be sure you get what you wanted. To find the data matching the conditions in Figure 4.16, you would first enter the ( $a > 4$ ) clause and display all of the values. Then you would add the ( $b < 0$ ) clause and display the results. Finally, you would add the parentheses and then the ( $c > 1$ ) clause.

No matter how careful you are with Boolean algebra there is always room for error. The problem is that natural languages such as English are ambiguous. For example, consider the request by a customer who wants to see a list of “All dogs that are yellow or white and born after June 1.” This statement can be interpreted two ways:

1. (dogs AND yellow) OR (white AND born after June 1).
2. (dogs) AND (yellow OR white) AND (born after June 1).

Animal
AnimalID
Name
Category
Breed
DateBorn
Gender

Customer: "I want to look at a cat, but I don't want any cats that are registered or that have red in their color."

Field	AnimalID	Category	Registered	Color
Table	Animal	Animal	Animal	Animal
Sort				
Criteria		'Cat'	Is Null	Not Like '%Red%'
Or				

```

SELECT AnimalID, Category, Registered, Color
FROM Animal
WHERE (Category='Cat') AND
      NOT ((Registered is NOT NULL)
           OR (Color LIKE '%Red%')).

```

**Figure 4.17**

Sample problem with negation. Customer knows what he or she does not want. SQL can use NOT, but you should use DeMorgan's law to negate the Registered and Color statements.

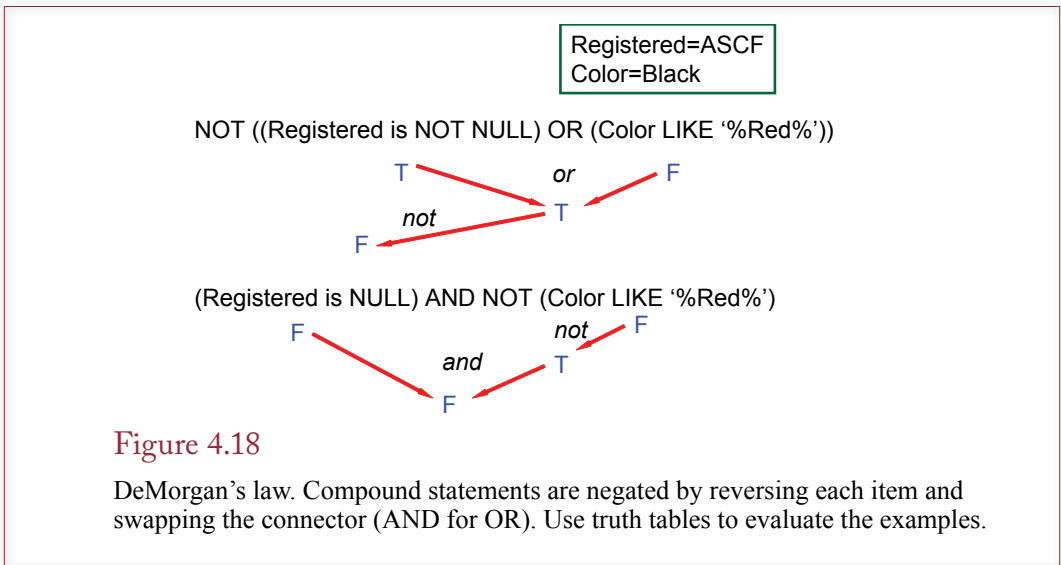
These two requests are significantly different. The first interpretation returns all yellow dogs, even if they are older. The second interpretation requests only young dogs, and they must be yellow or white. Most people do not use parentheses when they speak—although pauses help indicate the desired interpretation. A good designer (or salesperson) will ask the customer for clarification.

### DeMorgan's Law

Designing queries is an exercise in logic. A useful technique for simplifying complex queries was created by a logician named Augustus DeMorgan. Consider the Pet Store example displayed in Figure 4.17. A customer might come in and say, "I want to look at a cat, but I don't want any cats that are registered or that have red in their color." Even in SQL, the condition for this query is a little confusing: `(Category = 'Cat') AND NOT ((Registered is NOT NULL) OR (Color LIKE '%Red%'))`. The negation (NOT) operator makes it harder to understand the condition. It is even more difficult to create the QBE version of the statement.

The solution lies with **DeMorgan's law**, which explains how to negate conditions when two clauses are connected with an AND or an OR. DeMorgan's law states that to negate a condition with an AND or an OR connector, you negate each of the two clauses and switch the connector. An AND becomes an OR, and vice versa. Figure 4.17 shows how to handle the negative condition for the Pet Store customer. Each condition is negated (NOT NULL becomes NULL, and red becomes NOT red). Then the connector is changed from OR to AND. Figure 4.18 shows that the final truth value stays the same when the statement is evaluated both ways.

The advantage of the new version of the condition is that it is a little easier to understand and much easier to use in QBE. In QBE you enter the individual clauses for Registration and Color. Placing them on the same line connects them

**Figure 4.19**

Boolean criteria—mixing AND and OR. Notice the use of parentheses in SQL to ensure the clauses are interpreted in the right order. Also note that QBE required duplicating the condition for “Dog” in both rows.

Animal
AnimalID
Name
Category
Breed
DateBorn
Gender

List all dogs who are male and registered or who were born before 6/1/2013 and have white in their color.

Field	AnimalID	Category	Gender	Registered	DateBorn	Color
Table	Animal	Animal	Animal	Animal	Animal	Animal
Sort						
Criteria		'Dog'	'Male'	Is Not Null		
Or		'Dog'			< '01-Jun-2013'	Like '%White%'

```

SELECT AnimalID, Category, Gender, Registered, DateBorn, Color
FROM Animal
WHERE (( Category=N'Dog') AND
      ( ( (Gender=N'Male') AND (Registered Is Not Null) ) OR
        ( (DateBorn<'01-Jun-2013') AND (Color Like N'%White%') ) ) );

```



with AND. In natural language the new version is expressed as follows: A cat that is not registered and is not red. In practice DeMorgan's law is useful to simplify complex statements. However, you should always test your work by using sample data to evaluate the truth tables.

Criteria can become more complex when you mix clauses with AND and OR in the same query. Consider the question in Figure 4.19 to list all dogs who are male and registered or who were born before June 1 and have white in their color.

First, note that there is some ambiguity in the English statement about how to group the two clauses. Figure 4.20 shows the two possibilities. The use of the second who helps to clarify the split, but the only way to be absolutely certain is to use either parentheses or more words.

The SQL version of the query is straightforward—just be sure to use parentheses to indicate the priority for evaluating each phrase. Innermost clauses are always evaluated first. A useful trick in proofreading queries is to use a sample row and mark T or F above each condition. Next, combine the marks based on the parentheses and connectors (AND, OR). Then read the statement in English and see whether you arrive at the same result.

With QBE you list clauses joined by AND on the same row, which is equivalent to putting them inside one set of parentheses. Separate clauses connected by OR are placed on a new row. To interpret the query, look at each criteria row separately. If all of the conditions on one line are true, then the row is determined to be a match. A data row needs to match only one of the separate criteria lines (not all of them).

A second hint for building complex queries is to test just part of the criteria at one time—particularly with QBE. In this example, you would first write and test a query for male and registered. Then add the other conditions and check the results at each step. Although this process takes longer than just leaping to the final query, it helps to ensure that you get the correct answer. For complex queries it is always wise to examine the SQL WHERE clause to make sure the parentheses are correct.

### Useful WHERE Clauses

Most database systems provide the comparison operators displayed in Figure 4.21. Standard numeric data can be compared with equality and inequality operators. Text comparisons are usually made with the LIKE operator for pattern matching. For all text criteria, you need to know if the system uses case-sensitive comparisons. By default, Microsoft Access and SQL Server are not case-sensitive, so you can type the pattern or condition using any case. On the other hand, Oracle

#### Figure 4.20

Ambiguity in natural languages means the sentence could be interpreted either way. However, version (1) is the most common interpretation.

List all dogs who are male and registered or who were born before 6/1/2007 and have white in their color.

- 1: (male and registered) or (born before June 1 and white)
- 2: (male) and (registered or born before June 1) and (white)

Comparisons	Examples
Operators	<, =, >, <>, >=, BETWEEN, LIKE, IN
Numbers	AccountBalance > 200
Text	
Simple	Name > 'Jones'
Pattern match one	License LIKE 'A__82_'
Pattern match any	Name LIKE 'J%'
Dates	SaleDate BETWEEN '15-Aug-2013' AND '31-Aug-2013'
Missing Data	City IS NULL
Negation	Name IS NOT NULL
Sets	Category IN ('Cat', 'Dog', 'Hamster')

Figure 4.21

Common comparisons used in the WHERE clause. The BETWEEN clause is useful for dates but can be used for any type of data.

is case-sensitive by default so you have to be careful to type the case correctly. If you do not know which case was used, you can use the UPPER function to convert to upper case and then write the pattern using capital letters.

The **BETWEEN** clause is a useful way to handle common date conditions. The clause (SaleDate BETWEEN '15-Aug-2013' AND '31-Aug-2013') is equivalent to (SaleDate >= '15-Aug-2013' AND SaleDate <= '31-Aug-2013'). The date syntax shown here can be used on most database systems. Some systems allow you to use shorter formats, but on others, you will have to specify a conversion format. These conversion functions are not standard. For example, Access can read almost any common date format if you surround the date by pound signs (#) instead of quotes. Oracle often requires the TO\_DATE conversion function, such as SaleDate >= TO\_DATE('8/15/13', 'mm/dd/yy'). Be sure that you test all date conversions carefully, especially when you first start working with a new DBMS.

Another useful condition is to test for missing data with the NULL comparison. Two common forms are IS NULL and IS NOT NULL. Be careful—the statement (City = NULL) will not work with most systems, because NULL is not really a value. You must use (City IS NULL) instead. Unfortunately, conditions with the equality sign are not flagged as errors. The query will run—it just will never match anything.

## Computations

**What types of computations can be performed in SQL?** For the most part you would use a spreadsheet or write separate programs for serious computations. However, queries can be used for two types of computations: aggregations and simple arithmetic on a row-by-row basis. Sometimes the two types of calculations are combined. Consider the row-by-row computations first.

### Basic Arithmetic Operators

SQL and QBE can both be used to perform basic computations on each row of data. This technique can be used to automate basic tasks and to reduce the amount

```
SaleItem(SaleID, ItemID, SalePrice, Quantity)
```

```
Select SaleID, ItemID, SalePrice, Quantity,
SalePrice*Quantity As Extended
From SaleItem;
```

SaleID	ItemID	Price	Quantity	Extended
24	25	2.70	3	8.10
24	26	5.40	2	10.80
24	27	31.50	1	31.50

Figure 4.22

Computations. Basic computations (+ - \* /) can be performed on numeric data in a query. The new display column should be given a meaningful name.

of data storage. Consider a common order or sales form. As Figure 4.22 shows, the basic tables would include a list of items purchased: SaleItem(SaleID, ItemID, SalePrice, Quantity). In most situations you would need to multiply SalePrice by Quantity to get the total value for each item ordered. Because this computation is well defined (without any unusual conditions), there is no point in storing the result—it can be recomputed whenever it is needed. Simply build a query and add one more column. The new column uses elementary algebra and lists a name: SalePrice\*Quantity AS Extended. Remember that the computations are performed for each row in the query.

Most systems provide additional mathematical functions. For example, basic mathematical functions such as absolute value, logarithms, and trigonometric functions are usually available. Although these functions provide extended capabilities, always remember that they can operate only on data stored in one row of a table or query at a time.

### Aggregation

Databases for business often require the computation of totals and subtotals. Hence, query systems provide functions for **aggregation** of data. The common functions listed in Figure 4.23 can operate across several rows of data and return one value. The most commonly used functions are Sum and Avg, which are similar to those available in spreadsheets.

With SQL, the functions are simply added as part of the SELECT statement. With QBE, the functions are generally listed on a separate Total line. With Microsoft Access, you first have to click the summation ( $\Sigma$ ) button on the toolbar to add the Total line to the QBE grid. In both SQL and QBE, you should provide a meaningful name for the new column.

The Count function is useful in many situations, but make sure you understand the difference between Sum and Count. Sum totals the values in a numeric column. Count simply counts the number of rows. If you supply a column name to the Count function, you should use a primary key column or an asterisk (\*).

The difficulty with the Count function lies in knowing when to use it. You must first understand the English question. For example, the question *How many employees does the Pet Store have?* would use the Count function: SELECT Count(\*) From Employee. The question *How many units of Item 9764 have been sold?* requires the Sum function: SELECT Sum(Quantity) FROM OrderItem. The

Animal
AnimalID
Name
Category
Donation

Field	SalePrice
Table	SaleAnimal
Total	<b>Avg</b>
Sort	
Criteria	
Or	

```
SELECT Avg(Donation) AS AvgOfDonation
FROM Animal;
```

Sum  
Avg  
Min  
Max  
Count  
StDev or  
StdDev  
Var

Figure 4.23

Aggregation functions. Sample query in QBE and SQL to answer: What is the average sale price for all animals? Note that with Microsoft Access you have to click the summation button on the toolbar ( $\Sigma$ ) to display the Total line on the QBE grid.

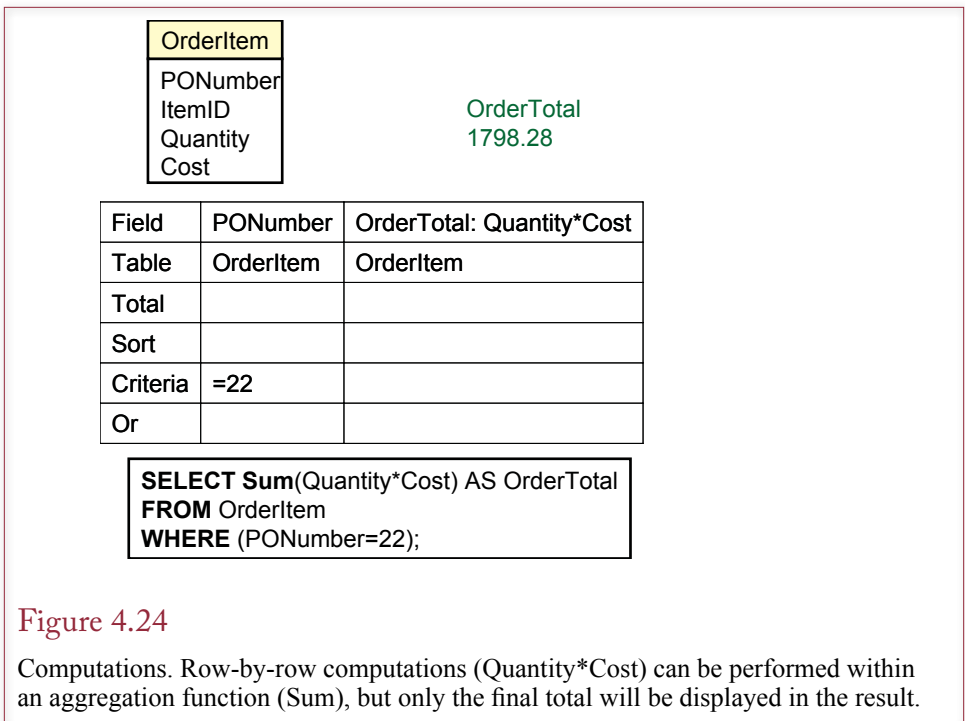
difference is that there can be only one employee per row in the Employee table, whereas a customer can buy multiple quantities of an item at one time. Also keep in mind that Sum can be used only on a column of numeric data (e.g., Quantity).

In many cases you will want to combine the **row-by-row calculations** with an aggregate function. The example in Figure 4.24 asks for the total value of a particular order. To get total value, the database must first calculate Quantity \* Cost for each row and then get the total of that column. The example also shows that it is common to specify a condition (WHERE) to limit the rows used for the total. In this example, you want the total for just one order.

There is one important restriction to remember with aggregation. You cannot display detail lines (row by row) at the same time you display totals. In the order example you can see either the detail computations (Figure 4.22) or the total value (Figure 4.24). In most cases it is simple enough to run two queries. However, if you want to see the detail and the totals at the same time, you need to create a report. Some of the most recent SQL standard extensions include provisions for displaying totals and details, but it is almost always easier to create a report.

Note that you can compute several aggregate functions at the same time. For example, you can display the Sum, Average, and Count at the same time: `SELECT Sum(Quantity), Avg(Quantity), Count(Quantity) From OrderItem`. In fact, if you need all three values, you should compute them at one time. Consider what happens if you have a table with a million rows of data. If you write three separate queries, the DBMS has to make three passes through the data. By combining the computations in one query, you cut the total query time to one-third. With huge tables or complex systems, these minor changes in a query can make the difference between a successful application and one that takes days to run.

Sometimes when using the Count function, you will also want to include the DISTINCT operator. For example, `SELECT COUNT (DISTINCT Category)`



*FROM Animal* will count the number of different categories and ignore duplicates. Although the command is part of the SQL standard, some systems (notably Access) do not support the use of the **DISTINCT** clause within the Count statement. To obtain the same results in Access, you would first build the query with the **DISTINCT** keyword. Save the query and then create a new query that computes the Count on the saved query.

## Functions

The **SELECT** command also supports functions that perform calculations on the data. These calculations include numeric forms such as the trigonometric functions, string function such as concatenating two strings, date arithmetic functions, and formatting functions to control the display of the data. Unfortunately, these functions are not standardized, so each DBMS vendor has different function names and different capabilities. Nonetheless, you should learn how to perform certain standard tasks in whichever DBMS you are using. Figure 4.25 lists some of the common functions you might need. Even if you are learning only one DBMS right now, you should keep this table handy in case you need to convert a query from one system to another.

String operations are relatively useful. Concatenation is one of the more powerful functions, because it enables you to combine data from multiple columns into a single display field. It is particularly useful when you want to combine a person's last and first names. Other common string functions convert the data to all lowercase or all uppercase characters. The length function counts the number of characters in the string column. A substring function is used to return a selected portion of a string. For example, you might choose to display only the first 20 characters of a long title.

Task	Access	SQL Server	Oracle
Strings			
Concatenation	FName & " " & LName	FName + ' ' + LName	Fname    ' '    LName
Length	Len(LName)	Length(LName)	LENGTH(LName)
Upper case	UCase(LName)	Upper(LName)	UPPER(LName)
Lower case	LCase(LName)	Lower(LName)	LOWER(LName)
Partial string	MID(LName,2,3)	Substring(LName,2,3)	SUBSTR(LName,2,3)
Dates			
Today	Date( ), Time( ), Now( )	GetDate( )	SYSDATE
Month	Month(myDate)	DateName(month, myDate)	TRUNC(myDate, 'mm')
Day	Day(myDate)	DatePart(day, myDate)	TRUNC(myDate, 'dd')
Year	Year(myDate)	DatePart(year, myDate)	TRUNC(myDate, 'yyyy')
Date arithmetic	DateAdd DateDiff	DateAdd DateDif	ADD_MONTHS MONTHS_BETWEEN LAST_DAY
Formatting	Format(item, format)	Str(item, length, decimal) Cast, Convert	TO_CHAR(item, format) TO_DATE(item, format)
Numbers			
Math functions	Cos, Sin, Tan, Sqrt	Cos, Sin, Tan, Sqrt	COS, SIN, TAN, SQRT
Exponentiation	2 ^ 3	Power(2, 3)	POWER(2, 3)
Aggregation	Min, Max, Sum, Count,	Min, Max, Sum, Count,	MIN, MAX, SUM, COUNT,
Statistics	Avg, StDev, Var	Avg, StDev, Var, LinRegSlope, Correlation	REGR, CORR

**Figure 4.25**

Differences in SQL functions. This table shows some of the differences that are commonly encountered when working with these database systems. Queries are often used to perform basic computations, but the syntax for handling these computations depends on the specific DBMS.

The powerful date functions are often used in business applications. Date columns can be subtracted to obtain the number of days between two dates. Additional functions exist to get the current date and time or to extract the month, day, or year parts of a date column. Date arithmetic functions can be used to add (or subtract) months, weeks, or years to a date. One issue you have to be careful with is entering date values into a query. Most systems are sensitive to the fact that world regions have different standards for entering and displaying dates. For example, 5/1/2013 is the first day in May in the United States, but it is the fifth day in January in Europe. To make sure that the DBMS understands exactly how you want a date interpreted, you might have to use a conversion function and specify the date format. Additional formatting functions can be used for other types of data, such as setting a fixed number of decimal points or displaying a currency sign.

A DBMS might have dozens of numeric functions, but you will rarely use more than a handful. Most systems have the common trigonometric functions (e.g., sine and cosine), as well as the ability to raise a number to a power. Most also provide some limited statistical calculations such as the average and standard deviation, and occasionally correlation or regression computations. You will have to consult the DBMS documentation for availability and details on additional functions. However, keep in mind that you can always write your own functions and use them in queries just as easily as the built-in functions.

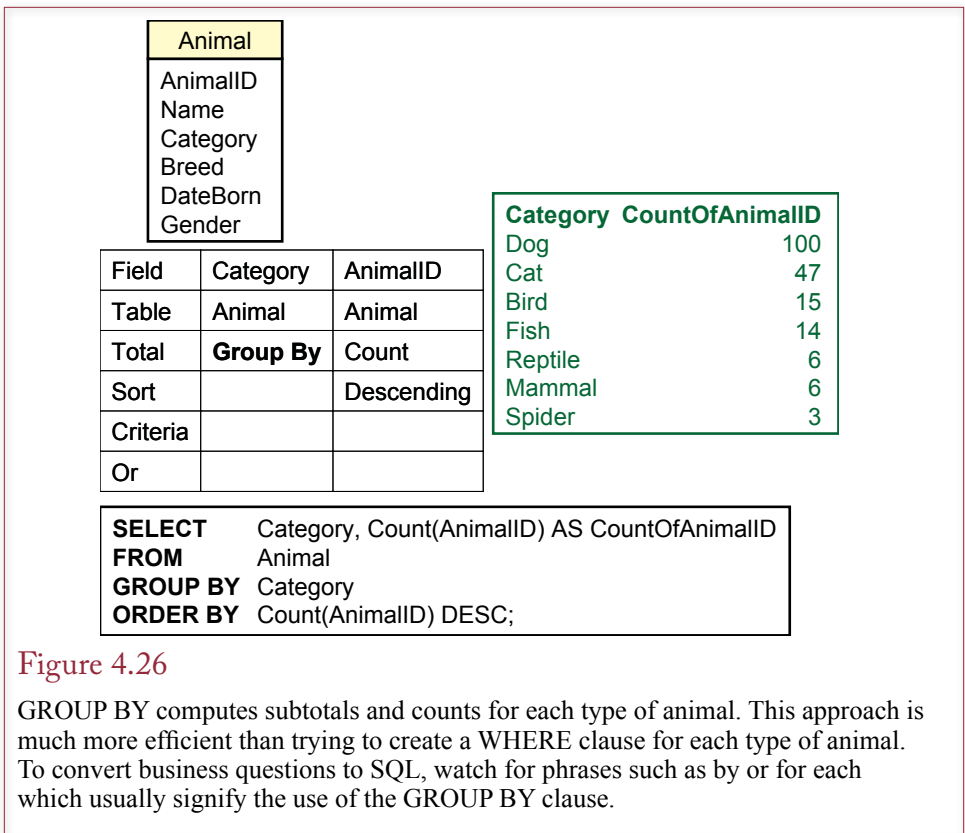


Figure 4.26

GROUP BY computes subtotals and counts for each type of animal. This approach is much more efficient than trying to create a WHERE clause for each type of animal. To convert business questions to SQL, watch for phrases such as by or for each which usually signify the use of the GROUP BY clause.

## Subtotals and GROUP BY

**How do you compute subtotals?** To look at totals for only a few categories, you can use the Sum function with a WHERE clause. For example you might ask *How many cats are in the animal list?* The query is straightforward: SELECT Count (AnimalID) FROM Animal Where (Category = N'Cat'). This technique will work, and you will get the correct answer. You could then go back and edit the query to get the count for dogs or any other category of animal. However, eventually you will get tired of changing the query. Also, what if you do not know all the categories?

Consider the more general query: Count the number of animals in each category. As shown in Figure 4.26, this type of query is best solved with the GROUP BY clause. This technique is available in both QBE and SQL. The SQL syntax is straightforward: just add the clause GROUP BY Category. The **GROUP BY** statement can be used only with one of the aggregate functions (Sum, Avg, Count, and so on). With the GROUP BY statement, the DBMS looks at all the data, finds the unique items in the group, and then performs the aggregate function for each item in the group.

By default, the output will generally be sorted by the group items. However, for business questions, it is common to sort (ORDER BY) based on the computation. The Pet Store example is sorted by the Count, listing the animals with the highest count first. Be careful about adding multiple columns to the GROUP BY clause. The subtotals will be computed for each distinct item in the entire GROUP BY



clause. If you include additional columns (e.g., Category and Breed), you might end up with a more detailed breakdown than you wanted.

Microsoft added a useful feature that can be used in conjunction with the ORDER BY statement. Sometimes a query will return thousands of lines of output. Although the rows are sorted, you might want to examine only the first few rows. For example, you might want to list your 10 best salespeople or the top 10 percent of your customers. When you have sorted the results, you can easily limit the output displayed by including the **TOP** statement; for example, SELECT TOP 10 SalesPerson, SUM(Sales) FROM Sales GROUP BY SalesPerson ORDER BY SUM(Sales) DESC. This query will compute total sales for each salesperson and display a list sorted in descending order. However, only the first 10 rows of the output will be displayed. Of course, you could choose any value instead of 10. You can also enter a percentage value (e.g., TOP 5 PERCENT), which will cut the list off after 5 percent of the rows have been displayed. These commands are useful when a manager wants to see the “best” of something and skip the rest of the rows. Oracle does not support the TOP condition, but you can use the internal row numbers to accomplish the same task. The command syntax relies on subqueries covered in the next chapter, but you might want to reduce your output rows, so an example is given here:

```
SELECT * FROM (SELECT ... FROM ...) WHERE ROWNUM <= 10;
```

Figure 4.27

Limiting the output with a HAVING clause. The GROUP BY clause with the Count function provides a count of the number of animals in each category. The HAVING clause restricts the output to only those categories having more than 10 animals.

Animal
AnimalID
Name
Category
Breed
DateBorn
Gender

Category	CountOfAnimalID
Dog	100
Cat	47
Bird	15
Fish	14

Field	Category	AnimalID
Table	Animal	Animal
Total	Group By	Count
Sort		Descending
Criteria		>10
Or		

```
SELECT Category, Count(AnimalID) AS CountOfAnimalID
FROM Animal
GROUP BY Category
HAVING Count(AnimalID) > 10
ORDER BY Count(AnimalID) DESC;
```

The 2011 SQL standard expanded a clause to do the same thing but the syntax is slightly different and it might take a while to be fully supported. The new clause is the `FETCH` statement added to the end of the `SELECT` clause. For example, the salesperson query would be written as:

```
SELECT SalesPerson, SUM(Sales) FROM Sales GROUP BY
SalesPerson ORDER BY SUM(Sales) DESC
FETCH FIRST 10 ROWS WITH TIES
```

### Conditions on Totals (`HAVING`)

The `GROUP BY` clause is powerful and provides useful information for making decisions. In cases involving many groups, you might want to restrict the output list, particularly when some of the groups are relatively minor. The Pet Store has categories for reptiles and spiders, but they are usually special-order items. In analyzing sales the managers might prefer to focus on the top-selling categories.

One way to reduce the amount of data displayed is to add the `HAVING` clause. The `HAVING` clause is a condition that applies to the `GROUP BY` output. In the example presented in Figure 4.27, the managers want to skip any animal category that has fewer than 10 animals. Notice that the SQL statement simply adds one line. The same condition can be added to the criteria grid in the QBE query. The `HAVING` clause is powerful and works much like a `WHERE` statement. Just be sure that the conditions you impose apply to the computations indicated by the `GROUP BY` clause. The `HAVING` clause is a possible substitute in Oracle which lacks the `TOP` statement. You can sort a set of subtotals and cut off the list to display only values above a certain limit.

### `WHERE` versus `HAVING`

When you first learn QBE and SQL, `WHERE` and `HAVING` look very similar, and choosing the proper clause can be confusing. Yet it is crucial that you understand the difference. If you make a mistake, the DBMS will give you an answer, but it will not be the answer to the question you want.

The key is that the `WHERE` statement applies to every single detail row in the original table. The `HAVING` statement applies only to the subtotal output from a `GROUP BY` query. To add to the confusion, you can even combine `WHERE` and `HAVING` clauses in a single query—because you might want to look at only some rows of data and then limit the display on the subtotals.

Consider the question in Figure 4.28 that counts the animals born after June 1, 2013, in each Category, but lists only the Category if there are more than 10 of these animals. The structure of the query is similar to the example in Figure 4.25. The difference in the SQL statement is the addition of the `WHERE` clause (`DateBorn > #6/1/2013#`). This clause is applied to every row of the original data to decide whether it should be included in the computation. Compare the count for dogs in Figure 4.26 (30) with the count in Figure 4.25 (100). Only 30 dogs were born after June 1, 2013. The `HAVING` clause then limits the display to only those categories with more than 10 animals.

The query is processed by first examining each row to decide whether it meets the `WHERE` condition. If so, the Category is examined and the Count is increased for that category. After processing each row in the table, the totals are examined to see whether they meet the `HAVING` condition. Only the acceptable rows are displayed. The same query in QBE is a bit more confusing. Both of the conditions are listed in the criteria grid. However, look closely at the Total row, and you will

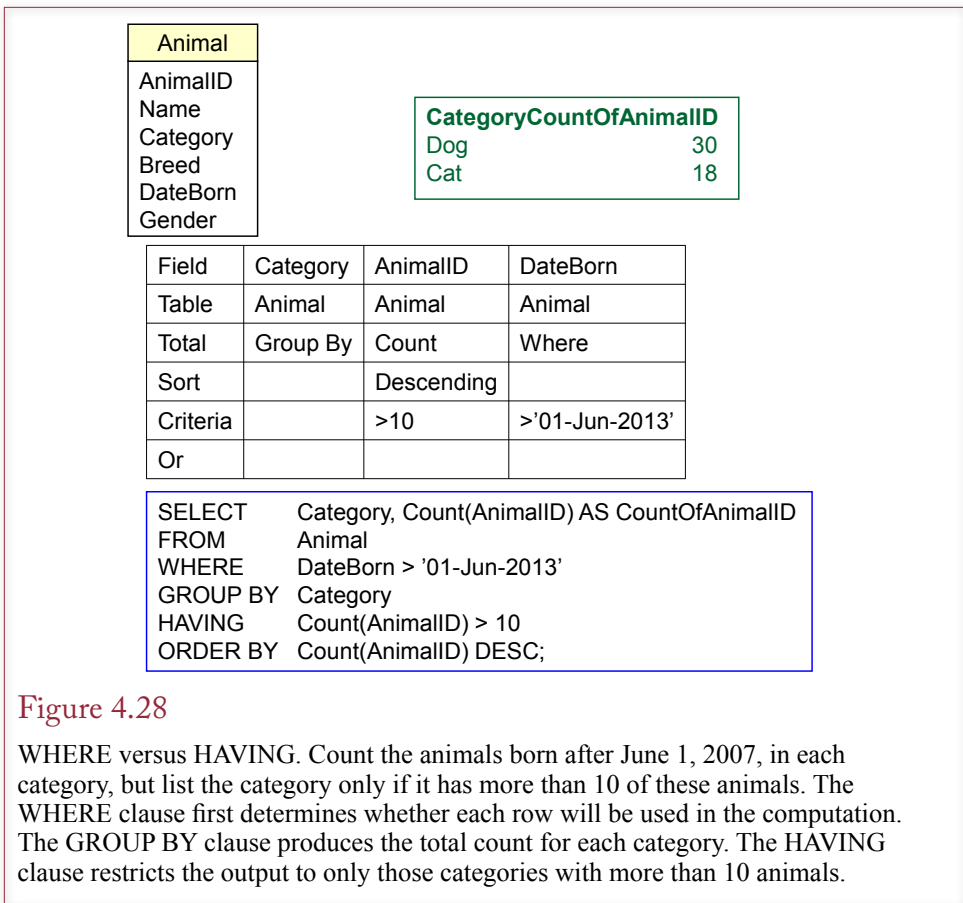


Figure 4.28

WHERE versus HAVING. Count the animals born after June 1, 2007, in each category, but list the category only if it has more than 10 of these animals. The WHERE clause first determines whether each row will be used in the computation. The GROUP BY clause produces the total count for each category. The HAVING clause restricts the output to only those categories with more than 10 animals.

see a Where entry for the DateBorn column. This entry is required to differentiate between a HAVING and a WHERE condition. To be safe, you should always look at the SQL statement to make sure your query was interpreted correctly.

### The Best and the Worst

Think about the business question, *Which product is our best seller?* How would you build a SQL statement to answer that question? To begin, you have to decide if “best” is measured in quantity or revenue (price times quantity). For now, simply use quantity. A common temptation is to write a query similar to: `SELECT Max(Quantity) FROM SaleItem`. This query will run. It will return the individual sale that had the highest sale quantity, but it will not sum the quantities. A step closer might be: `SELECT ItemID, Max(Sum(Quantity)) FROM SaleItem GROUP BY ItemID`. But this query will not run because the database cannot compute the maximum until after it has computed the sum. So, the best answer is to use: `SELECT ItemID, Sum(Quantity) FROM SaleItem GROUP BY ItemID ORDER BY Sum(Quantity) DESC`. This query will compute the total quantities purchased for each item and display the result in descending order—the best-sellers will be at the top of the list.

Note that this query displays more than the simple “best” answer. It displays all of the totals. The advantage to this approach is that it shows other rows that might

be close to the “best” entry, which is information that might be valuable to the decision maker. The one drawback to this approach is that it returns the complete list of items sold. Generally, most businesspeople will want to see more than just the top or bottom item, so it is not a serious drawback—unless the list is too long. In that case, you can use the TOP or HAVING command to reduce the length of the list.

## Multiple Tables

**How do you use multiple tables in a query?** All the examples so far have used a single table—to keep the discussion centered on the specific topics. In practice, however, you often need to combine data from several tables. In fact, the strength of a DBMS is its ability to combine data from multiple tables.

Chapter 3 shows how business forms and reports are dissected into related tables. Although the normalization process makes data storage more efficient and avoids common problems, ultimately, to answer the business question, you need to recombine the data from the tables. For example, the Sale table contains just the CustomerID to identify the specific customer. Most people would prefer to see the customer name and other attributes. This additional data is stored in the Customer table—along with the CustomerID. The objective is to take the CustomerID from the Sale table and look up the matching data in the Customer table.

Figure 4.29

List the CustomerID of everyone who bought something between April 1, 2013 and May 31, 2013. Most people would prefer to see the name and address of the customer—those attributes are in the Customer table.

Sale			CustomerID
SaleID			6
SaleDate			8
EmployeeID			14
CustomerID			19
SalesTax			22
Field	CustomerID	SaleDate	24
Table	Sale	Sale	28
Sort	Ascending		36
Criteria		Between '01-Apr-2013' And '31-May-2013'	37
Or			38
			39
			42
			50
			57
			58
			63
			74
			80
			90

SELECT DISTINCT CustomerID
FROM Sale
WHERE (SaleDate Between '01-Apr-2013'
And '31-May-2013')
ORDER BY CustomerID;

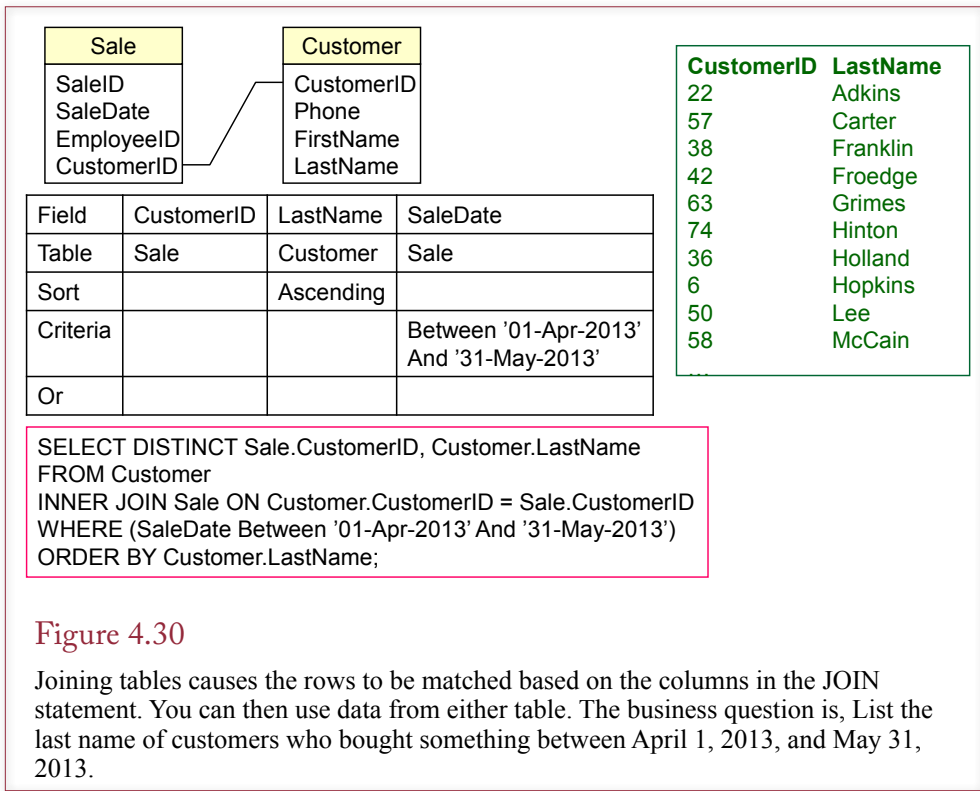


Figure 4.30

Joining tables causes the rows to be matched based on the columns in the JOIN statement. You can then use data from either table. The business question is, List the last name of customers who bought something between April 1, 2013, and May 31, 2013.

## Joining Tables

With modern query languages, combining data from multiple tables is straightforward. You simply specify which tables are involved and how the tables are connected. QBE is particularly easy to use for this process. To understand the process, first consider the business question posed in Figure 4.29: list the CustomerID of everyone who bought something between 4/1/2013 and 5/31/2013. Because some customers might have made purchases on several days, the DISTINCT clause can be used to delete the duplicate listings.

Most managers would prefer to see the customer name instead of CustomerID. However, the name is stored in the Customer table because it would be a waste of space to copy all of the attributes to every table that referred to the customer. If you had these tables only as printed reports, you would have to take the CustomerID from the sale reports and find the matching row in the Customer table to get the customer name. Of course, it would be time-consuming to do the matching by hand. The query system can do it easily.

As illustrated in Figure 4.30, the QBE approach is somewhat easier than the SQL syntax. However, the concept is the same. First, identify the two tables involved (Sale and Customer). In QBE, you select the tables from a list, and they are displayed at the top of the form. In SQL, you enter the table names on the FROM line. Second, you tell the DBMS which columns are matched in each table. In this case you match CustomerID in the Sale table to the CustomerID in the Customer table. Most of the time the column names will be the same, but they could be different.

```
SQL 92/Current Syntax
FROM Table1
INNER JOIN Table2
  ON Table1.Column = Table2.Column

SQL 89/Old Syntax
FROM Table1, Table2
WHERE Table1.Column = Table2.Column

Informal Syntax for Notes
FROM Table1, Table2
JOIN Column
```

Figure 4.31

SQL 92 and SQL 89 syntax to join tables. The informal syntax cannot be used with a DBMS, but it is easier to read when you need to combine many tables.

In SQL tables are connected with the JOIN statement. This statement was changed with the introduction of SQL 92—however, you will encounter many older queries that still use the older SQL 89 syntax. With SQL 89 the JOIN condition is part of the WHERE clause. Most vendors have converted to the SQL 92 syntax, so this text will rely on that format. As Chapter 5 shows, the SQL 92 syntax is much easier to understand when you need to change the join configuration.

The syntax for a JOIN is displayed in Figure 4.31. An informal syntax similar to SQL 89 is also shown. The DBMS will not accept statements using the informal syntax, but when the query uses many tables, it is easier to jot down the informal syntax first and then add the details needed for the proper syntax. Note that with both QBE and SQL, you must specify the tables involved and which columns contain matching data.

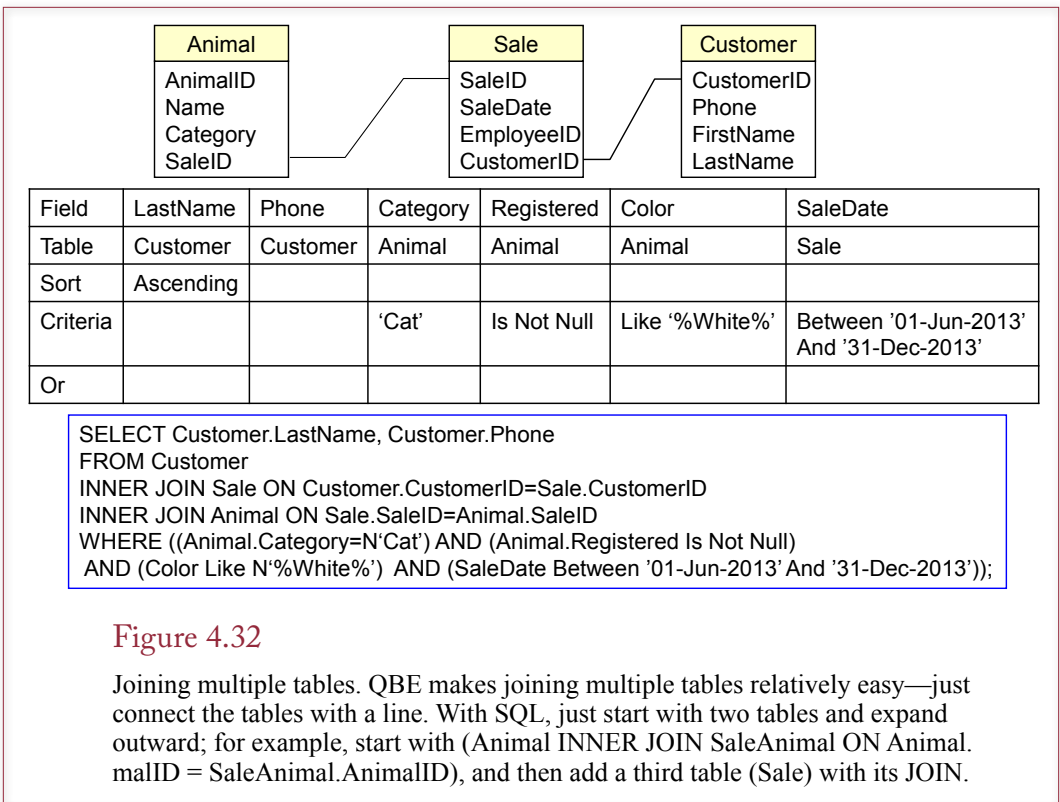
### Identifying Columns in Different Tables

Examine how the columns are specified in the SQL JOIN statement. Because the column CustomerID is used in both tables, it would not make sense to write CustomerID = CustomerID. The DBMS would not know what you meant. To keep track of which column you want, you must also specify the name of the table: Sale.CustomerID. Actually, you can use this syntax anytime you refer to a column. You are required to use the full table.column name only when the same column name is used in more than one table.

### Joining Many Tables

A query can use data from several different tables. The process is similar regardless of the number of tables. Each table you want to add must be joined to one other table through a data column. If you cannot find a common column, either the normalization is wrong or you need to find a third table that contains links to both tables.

Consider the example in Figure 4.32: List the name and phone number of anyone who adopted a registered white cat between two given dates. An important step is to identify the tables needed. For large problems involving several tables, it is best to first list the columns you want to see as output and the ones involved in the constraints. In the example, the name and phone number you want to see are



**Figure 4.32**

Joining multiple tables. QBE makes joining multiple tables relatively easy—just connect the tables with a line. With SQL, just start with two tables and expand outward; for example, start with (Animal INNER JOIN SaleAnimal ON Animal.malID = SaleAnimal.AnimalID), and then add a third table (Sale) with its JOIN.

in the Customer table. The Registration status, Color, and Category (Cat) are all in the Animal table. The SaleDate is in the Sale table. The Animal table connects to the Sale table through the SaleID column in the Animal table.

When the database contains a large number of tables, complex queries can be challenging to build. You need to be familiar with the tables to determine which tables contain the columns you want to see. For large databases, an entity-relationship diagram (ERD) or a class diagram can show how the tables are connected. Chapter 3 explains how Access sets referential integrity for foreign key relationships. Access uses the relationships to automatically add the JOINS to QBE when you choose a table. You can also use the class diagram to help users build queries.

When you first see it, the SQL 92 syntax for joining more than two tables can look confusing. In practice, it is best not to memorize the syntax. When you are first learning SQL, understanding the concept of the JOIN is far more important than worrying about syntax. Figure 4.33 shows the syntax needed to join three tables. To read it or to create a similar statement, start with the first table and JOIN it to a second table with the corresponding ON condition. Then JOIN the next table with a matching ON statement. Just be sure that the new table can be joined to one of the existing tables. Unfortunately, this syntax will not work in Microsoft Access, which requires the addition of parentheses. Figure 4.33 also shows an easier syntax that is faster to write when you are first developing a query or when you are in a hurry—perhaps on an exam. It is similar to the older SQL 89 syntax (but not exactly correct) where you list all the tables in the FROM clause and then join them in the WHERE statement.



```
SQL 92 Syntax for Three Tables
FROM Table1
  INNER JOIN Table2 ON Table1.ColA = Table2.ColA
  INNER JOIN Table3 ON Table2.ColB = Table3.ColB

Easier notation, But Not Correct Syntax
FROM Table1, Table2, Table3
JOIN      ColA   ColB
```

Figure 4.33

Joining multiple tables. With SQL 92 syntax, first join two tables within parentheses and then add a table and its JOIN condition. When you want to focus on the tables being joined, use the easier notation—just remember that it must be converted to SQL 92 syntax for the computer to understand it.

### Hints on Joining Tables

Joining tables is closely related to data normalization. Normalization splits data into tables that can be stored and searched more efficiently. Queries and SQL are the reverse operation: JOINS are used to recombine the data from the tables. If the normalization is incorrect, it might not be possible to join the tables. As you build queries, double-check your normalization to make sure it is correct. Students often have trouble with JOINS, so this section provides some hints to help you understand the potential problems.

Remember that any time you use multiple tables, you must join them together. Most database query systems will accept a query even if the tables are not joined. They will even give you a result. Unfortunately, the result is usually meaningless. The joined tables also create a huge query result. Without any constraints most query systems will produce a cross join, where every row in one table is paired with every row in the other table.

Where possible, you should double-check the answer to a complex query. Use sample data and individual test cases in which you can compute the answer by hand. You should also build a complex query in stages. Start with one or two tables and check the intermediate results to see if they make sense. Then add new tables and additional constraints. Add the summary calculations last (e.g., Sum, Avg). It's hard to look at one number (total) and decide whether it is correct. Instead, look at an intermediate listing and make sure it includes all of the rows you want; then add the computations.

Columns used in a JOIN are often key columns, but you can join tables on any column. Similarly, joined columns may have different names. For example, you might join an Employee.EmployeeID column to a Sale.SalesPerson column. The only technical constraint is that the columns must contain the same type of data (domain). In some cases, you can minimize this limitation by using a function to convert the data. For example, you might use `Left(ZipCode,5) = ZipCode5` to reduce a nine-digit ZipCode string to five digits. Just make sure that it makes sense to match the data in the two columns. For instance, joining tables on `Animal.AnimalID = Employee.EmployeeID` would be meaningless. The DBMS would actually accept the JOIN (if both ID values are integers), but the JOIN does not make any sense because an Employee can never be an Animal (except in science-fiction movies).

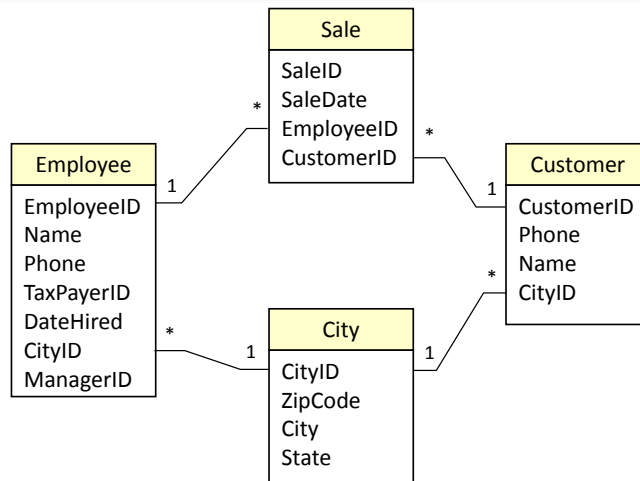


Figure 4.34

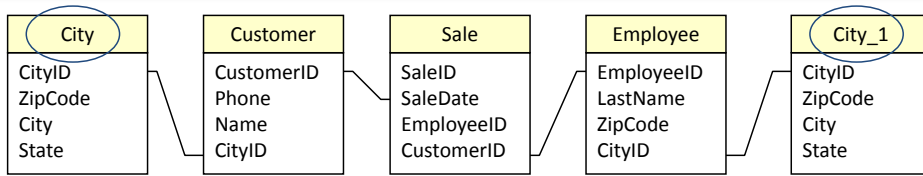
A query with these four tables with four JOINS would return only rows where the Employee had the same CityID as the Customer. If you need only the Customer city, just delete the JOIN between Employee and CityID. If you want both cities, add a second copy of the City table as a fifth table.

Avoid multiple ties between tables. This problem often arises in Access when you have predefined relationships between tables. Access QBE automatically uses those relationships to join tables in a query. If you select the four tables shown in Figure 4.34 and leave all four JOINS, you will not get the answer you want. The four JOINS will return Sales only where the Employee placing the order has the same CityID as the Customer! If you only need the City for the Customer, the solution is to delete the JOIN between Employee and City. In general, if your query uses four tables, you should have three JOINS (one less than the number of tables).

Sometimes it is helpful to remember that a JOIN condition also works as a row filter. The standard join will only return rows from a table that match those in the first table. For example, `Sale.CustomerID = Customer.CustomerID` will return customer data but only if those customers have already participated in a sale.

### Table Alias

Consider the preceding Employee/Customer/City example in more detail. What if you really want to display the City for the Customer and the City for the Employee? Of course, you want to allow the cities to be different. The answer involves a little-known trick in SQL: just add the City table twice. The second “copy” will have a different name (e.g., `City_1`). You give a table a new name (**alias**) within the FROM clause: `FROM City AS City_1`. As shown in Figure 4.35, the City table is joined to the Customer. The `City_1` table is joined to the Employee table. Now the query will perform two separate JOINS to the same table—simply because it has a different name.



```

SELECT Customer.CustomerID, Customer.CityID, City.City, Sale.EmployeeID,
Employee.LastName, Employee.CityID, City_1.City
FROM (City INNER JOIN (Customer INNER JOIN (Employee INNER JOIN Sale ON
Employee.EmployeeID = Sale.EmployeeID) ON Customer.CustomerID =
Sale.CustomerID) ON City.CityID = Customer.CityID) INNER JOIN City AS City_1 ON
Employee.CityID = City_1.CityID;
  
```

CID	Customer.CityID	City.City	EID	LastName	Employee.CityID	City_1.City
15	11013	Galveston	1	Reeves	11060	Lackland AFB
53	11559	Beaver Dam	2	Gibson	9146	Roanoke Rapids
38	11701	Laramie	3	Reasoner	8313	Springfield
66	7935	Danville	8	Carpenter	10592	Philadelphia
5	9175	Fargo	3	Reasoner	8313	Springfield

Figure 4.35

Table alias. The City table is used twice. The second time, it is given the alias City\_1 and treated as a separate table. Hence, different cities can be retrieved for Customer and for Employee.

## Create View

Any query that you build can be saved as a **view**. Microsoft simply refers to them as saved queries, but SQL and Oracle call them views. In either case the DBMS analyzes and stores the SQL statement so that it can be run later. If a query needs to be run many times, you should save it as a view so that the DBMS has to analyze it only once. Figure 4.36 shows the basic SQL syntax for creating a view. You start with any SELECT statement and add the line (CREATE VIEW ...).

The most powerful feature of a view is that it can be used within another query. Views are useful for queries that you have to run many times. You can also create views to handle complex questions. Users can then create new, simpler queries based on the views. In the example in Figure 4.36, you would create a view (Kit-

Figure 4.36

Views. Views are saved queries that can be run at any time. They improve performance because they have to be entered only once, and the DBMS has to analyze them only once.

```

CREATE VIEW Kittens AS
SELECT *
FROM Animal
WHERE (Category = 'Cat' AND (Today-DateBorn < 180));
  
```

```
SELECT Avg(ListPrice)
FROM Kittens
WHERE (Color LIKE '%Black%');
```

Figure 4.37

Queries based on views. Views can be used within other queries.

tens) that displays data for Cats born within the last 180 days. As shown in Figure 4.37, users could search the Kittens view based on other criteria such as color.

As long as you want to use a view only to display data, the technique is straightforward. However, if you want a view that will be used to change data, you must be careful. Depending on how you create the view, you might not be able to update some of the data columns in the view. The example shown in Figure 4.38 is an updatable view. The purpose is to add new data for ordering items. The user enters the OrderID and the ItemID. The corresponding description of that Item is automatically retrieved from the Item table.

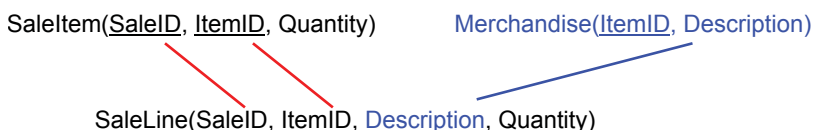
Figure 4.39 illustrates the problem that can arise if you are hasty in choosing the columns in a view. Here the OrderLine view uses the ItemID value from the Item table (instead of from the OrderItem table). Now you will not be able to add new data to the OrderLine view. To understand why, consider what happens when you try to change the ItemID from 57 to 32. If it works at all, the new value is stored in the Item table, which simply changes the ItemID of cat food from 57 to 32.

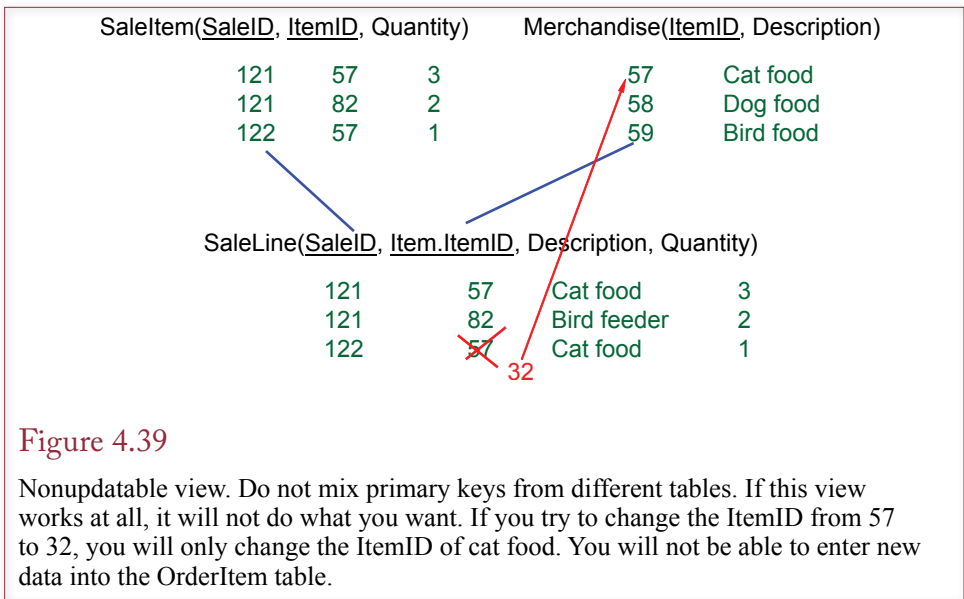
To ensure that a view can be updated, the view should be designed to change data in only one table. The rest of the data is included simply for display—such as verifying that the user entered the correct ItemID. You should never include primary key columns from more than one table. Also, to remain updatable, a view cannot use the DISTINCT keyword or contain a GROUP BY or HAVING clause.

Views have many uses in a database. They are particularly useful in helping business managers work with the database. A database administrator (DBA) or MIS worker can create views for the business managers, who see the section of the database expressed only in the views. Hence, you can hide the view's complexity and size. Most important, you can hide the JOINS needed to build the view, so managers can work with simple constraints. By keeping the view updatable, managers never need to use the underlying raw tables.

Figure 4.38

Updatable view. The OrderLine view is designed to change data in only one table (OrderItem). The Description from the Item table is used for display to help the user verify that the ItemID was entered correctly.





**Figure 4.39**

Nonupdatable view. Do not mix primary keys from different tables. If this view works at all, it will not do what you want. If you try to change the ItemID from 57 to 32, you will only change the ItemID of cat food. You will not be able to enter new data into the OrderItem table.

Note that some database systems place restrictions on commands allowed within a view. For example, older Oracle and newer SQL Server systems do not allow you to use the ORDER BY clause in a saved view. The reason for this restriction was to enable the system to provide better performance by optimizing the query. To sort a result, you had to add the ORDER BY statement to a new query that called the saved view. Finally, no matter how careful you are at constructing a view with a JOIN statement, the DBMS might still refuse to consider it updateable. When the DBMS accepts it, updateable views can save some time later when building forms. But, at other times you have to give up and go with simpler forms.

## Newer Searches and Patterns

**How do you search XML and complex text strings?** Over time, companies have found the need to store complex data in databases. Although most DBMSs can store new and different types of data, it also becomes important to retrieve that data. The standard WHERE conditions apply only to the basic data types (simple numbers and text). A few types of complex data have become important and common enough that vendors have adopted standard methods to search these new data types. The two most common types of data are: XML hierarchies and long text.

XML is stored as tagged data, and an entry is commonly organized as a hierarchy. A parent node can have multiple child nodes. For instance, an <Order> tag can have multiple <Item> tags to indicate which items are being ordered. Developers and users need a common method to search an XML tag, including the ability to drill down and list elements within the hierarchy. XQuery was developed as a standard to perform these searches. Today, the SQL 2006 standard and most of the big DBMSs support the XML data type and the use of XQuery to search XML data.

Note: This section covers XQuery and RegEx searches for XML data and text strings. It could be skipped or covered later. Be sure you understand basic SQL commands before dealing with this material.

```
<shipment>
<ShipID>1573</ShipID>
<ShipDate>15-May-2010</ShipDate>
<Items>
  <Item>
    <ItemID>15</ItemID>
    <Description>Leash</Description>
    <Quantity>20</Quantity>
    <Price>8.95</Price>
  </Item>
  <Item>
    <ItemID>32</ItemID>
    <Description>Collar</Description>
    <Quantity>25</Quantity>
    <Price>14.50</Price>
  </Item>
</Items>
</shipment>
```

Figure 4.40

Sample XML data. Assume vendors send a shipping invoice file when they send products. The sample data shows a single shipment that contains a ShipID and ShipDate. The repeating section contains a list of items and the quantity that were shipped.

The basic pattern matching provided by the SQL standard is somewhat simplistic. With only two search symbols (all text or one character), it is relatively easy to use. But it is not very powerful. Programmers have long had a powerful string search tool called regular expressions. Technically, regular expressions were added to the SQL 1999 standard, but only recently have vendors added it as a feature.

## XQuery

**XQuery** is a standardized method for retrieving values from an XML string. XML uses tags to mark each item and the designer can create almost any terms for the tags. But the XML string has to be well-formed and can be validated against a schema that specifies the data model. An XML data model is essentially a hierarchical definition of the data and repeating elements that can be stored in the XML string. Figure 4.40 shows a simple XML file with sample data. When vendors ship products to Sally's Pet Store, they are asked to send this XML file that contains the shipping invoice data. The vendor provides a ShipID and the ShipDate to reference their data in case questions arise later. The repeating Items section contains a list of the items that were shipped along with the quantity and price paid. This example is intentionally kept simple. A real-world invoice could have many levels and options. Note that all XML tags are case-sensitive.

For illustration, Figure 4.41 shows a simple version of the **XML schema** for the sample data. The schema is the data definition and it can be used to create and to validate XML data files. Note how the hierarchical form is defined through the nested elements. In this case, the reference to Items is listed within the main shipment. Also, note that each data point is defined by an element and the ele-

```

<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema" >
  <xsd:element name="shipment">
    <xsd:complexType>
      <xsd:element name="ShipID" minOccurs="0" type="xsd:int" />
      <xsd:element name="ShipDate" minOccurs="0" type="xsd:date" />
      <xsd:sequence>
        <xsd:element ref="Items" minOccurs="0" maxOccurs="unbounded" />
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
  <xsd:element name="Items">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="ItemID" minOccurs="0" type="xsd:int" />
        <xsd:element name="Description" minOccurs="0" />
        <xsd:element name="Quantity" minOccurs="0" type="xsd:int" />
        <xsd:element name="Price" minOccurs="0" type="xsd:double" />
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
</xsd:schema>

```

Figure 4.41

Simple XML schema for sample data. Note the hierarchical definition through the nested elements.

ment specifies the type of data. The default data type is text. XML and schemas support considerably more complex data specifications, but it is best to start with the simple formats. One way to create an XML data file and a schema definition is to export a table from Microsoft Access as an XML file. This approach adds some overhead to define a few Office/Access features; but it is a relatively painless method to create an XML schema.

### Storing XML Data

Unfortunately, Microsoft Access tables do not directly support the XML data type and XQuery. You will need to use SQL Server or Oracle to work with the remaining examples. A few standalone tools found on the Web can also be used to learn and experiment with XQuery.

SQL Server (and Oracle) can handle XML both as data stored within a table and as a data variable within the programming language. The underlying concepts are the same, but since this chapter focuses on tables and queries, the examples here use XML data placed into a table. Figure 4.42 shows the CREATE TABLE and INSERT commands to create a new ShippingInvoice table and insert the sample data. Notice that the entire XML string is inserted into a single cell (row/column) of the ShippingInvoice table. That is, the table contains one row to represent the single invoice and all of the XML data goes into the XML Contents column.

### Retrieving XML Data with XQuery

XML data is basically a large string. You can retrieve the entire XML string with a relatively standard SELECT statement, simply by specifying the desired row.

```

CREATE TABLE ShippingInvoice (
    ShippingID int IDENTITY(1,1) NOT NULL,
    InvoiceDate date NULL,
    OrderID int NULL,
    Contents xml NULL,
    CONSTRAINT pk_ShippingInvoice PRIMARY KEY (ShippingID)
)
GO
INSERT INTO ShippingInvoice (InvoiceDate, OrderID, Contents)
VALUES ('19-May-2013', 12, '
<shipment>
<ShipID>1573</ShipID>
<ShipDate>15-May-2013</ShipDate>
<Items>
<Item><ItemID>15</ItemID><Description>Leash</Description>
  <Quantity>20</Quantity><Price>8.95</Price></Item>
<Item><ItemID>32</ItemID><Description>Collar</Description>
  <Quantity>25</Quantity><Price>14.50</Price></Item>
</Items>
</shipment>
');

```

Figure 4.42

SQL Server table with XML data and INSERT command.

Figure 4.43

Simple XQuery example. The SQL SELECT statement specifies the desired row and column that contains the XML data. The `.query('shipment/Items')` term is a new element that builds the XML query.

```

SELECT Contents.query('shipment/Items') As ItemList
FROM ShippingInvoice
WHERE ShippingID=1;
<Items>
  <Item>
    <ItemID>15</ItemID>
    <Description>Leash</Description>
    <Quantity>20</Quantity>
    <Price>8.95</Price>
  </Item>
  <Item>
    <ItemID>32</ItemID>
    <Description>Collar</Description>
    <Quantity>25</Quantity>
    <Price>14.50</Price>
  </Item>
</Items>

```



However, you will often need to use XQuery to extract particular elements from the XML string. Figure 4.43 shows one of the simplest XQuery examples. The SQL SELECT statement is used to specify the desired row (ShippingID=1) and the column (Contents) of the XML document. The .query('shipment/Items') is the new term that activates XQuery. XQuery supports several relatively complex capabilities for searching an XML document. Only a few commonly-used examples are given here. Once you understand the basic structure of XML and XQuery, you can study the reference documents to create more complex queries. However, keep in mind that instead of building a hugely complex XQuery, it is often better to extract all of the XML data and store it directly in relational tables. Then you can use the power and speed of SQL to retrieve the data.

The simplest form of an XQuery is to retrieve a segment of the XML document. The clearest way to do that is to simply specify the desired segment from the top down. In the sample data, “shipment” is the root node and “Items” is the repeating section. Hence, the query ‘shipment/Items’ returns the entire entry under the “Items” node.

What if you want to retrieve data based on a specific value stored within a node? For instance, you want to retrieve all of the information for ItemID 15 in the sample data. XQuery has a couple of methods for retrieving this data. Figure 4.44 shows the simplest approach: /shipment/Items/Item[ItemID=15]. Specify the hierarchical structure and then enter the desired condition in square brackets [ ].

Another useful trick is to return just a single element within a node. In the prior example, instead of returning the entire entry for ItemID 15, it might be useful to retrieve only the Quantity shipped. Figure 4.45 shows the syntax for specifying a single node. Simply add the node name after the brackets: /shipment/Items/Item[ItemID=15]/Quantity.

Complex search conditions are available, including a “contains” function to search text elements for specific values. Elements can also include attributes, such as <Price currency=”USD”>8.95</Price>, which uses a currency attribute to specify the monetary units. (USD is the standard symbol for U.S. Dollar). To search for this particular attribute, you could use a query of the form: /shipment/

Figure 4.44

XQuery to retrieve entry based on value stored within an element. Find entry for ItemID 15.

```
SELECT Contents.query('
/shipment/Items/Item[ItemID=15]
') As ItemList
FROM ShippingInvoice
WHERE ShippingID=1;

<Item>
  <ItemID>15</ItemID>
  <Description>Leash</Description>
  <Quantity>20</Quantity>
  <Price>8.95</Price>
</Item>
```

```

SELECT Contents.query('
  /shipment/Items/Item[ItemID=15]/Quantity
') As ItemList
FROM ShippingInvoice
WHERE ShippingID=1;

<Quantity>20</Quantity>

```

**Figure 4.45**

Extract a single element from the found node by adding the element name (`/Quantity`) after the brackets.

`Items/Item/Price[@currency="USD"]`; which will return only those items where the price currency is directly specified as USD.

Several other search options are available, but they use a somewhat cryptic annotation mechanism. XQuery also supports a relatively flexible search language that is often easier to read. It is abbreviated as FLWOR: for, let, where, order by, return. Figure 4.46 shows an example of the “for” loop that examines all of the items, searching for an entry with ItemID equal to 15. Note the use of an internal variable (`$item`) that is used to temporarily hold the values of each node being examined. The query also uses the data function to extract the value and return the simple number (20) without its XML tags. This data function could have been used in the prior examples as well. The point of the “for” loop is that it returns all of the nodes that meet the specified criteria, so it can return multiple “rows” of data—although all of the “rows” are stored within the single XML document. The return statement also supports an if/then/else construct so you can return modified results based on a conditional test.

### *XQuery in Oracle*

Oracle support for XML and XQuery is similar to the examples in this section, but the syntax is slightly different. First, the data type needed is `XMLType`. Also, note that “contents” is a reserved word in Oracle, so the column name in the `ShippingInvoice` table should be changed to `xContents`. If you create a sequence and an insert trigger for the table, the existing `INSERT` command will work. However, it

**Figure 4.46**

Using FLWOR commands to search nodes and return a single value. Also uses the data function to return the value without the XML tags.

```

SELECT Contents.query('
  for $item in /shipment/Items/Item
  where $item/ItemID=15
  return data($item/Quantity)
') As ItemList
FROM ShippingInvoice
WHERE ShippingID=1;

20

```

is probably easier to just modify the INSERT command to include the ShippingID and specify the ShippingID value as 1. So, the setup commands are:

```
CREATE TABLE ShippingInvoice (
    ShippingID number NOT NULL,
    InvoiceDate date,
    OrderID number,
    xContents XMLType,
    CONSTRAINT pk_ShippingInvoice PRIMARY KEY (ShippingID)
);
INSERT INTO ShippingInvoice (ShippingID, InvoiceDate,
OrderID, xContents)
VALUES (1, '19-May-2013', 12, '
<shipment>
<ShipID>1573</ShipID>
<ShipDate>15-May-2013</ShipDate>
<Items>
  <Item><ItemID>15</ItemID><Description>Leash</
Description>
  <Quantity>20</Quantity><Price>8.95</Price></Item>
  <Item><ItemID>32</ItemID><Description>Collar</
Description>
  <Quantity>25</Quantity><Price>14.50</Price></Item>
</Items>
</shipment>
');
```

The syntax for calling XQuery is also somewhat different. However, the XQuery functions are mostly standardized. The basic command matching Figure 4.43 shows the difference using the extract function:

```
SELECT extract(xContents, '
  /shipment/Items
') As ListResult
FROM ShippingInvoice
WHERE ShippingID=1;
```

Once you understand the syntax, the method of using XQuery is the same as the other examples. But always be sure to test everything.

### Summary

XQuery is a powerful tool for searching XML data trees. However, keep in mind that all searches through XML are based on string values and they are rarely (if ever) indexed. Consequently, XQuery searches can be quite slow. Additionally, you have to be cautious and test all of your queries to ensure they are retrieving exactly the requested data and not missing anything. If XML data needs to be searched often, it is better to extract the individual elements and put them into standard relational tables. Then use SQL to perform the searches. XQuery could be used to perform the data extraction. Remember that XQuery works on data in one row and one cell at a time.

Ultimately, a designer must make the decision of whether to extract XML data into relational tables or to leave it stored as a single XML document. If the data is rarely used except for occasional searches, it can probably be left as an XML document. However, it will still be necessary to have someone around who can

1. Start Visual Studio.
2. Open a New Project: Visual C#, Database, SQL Server: Visual C# SQL CLR Database Project.
3. Choose the Pet Store database.
4. Right-click the project name, Add, User Defined Function: RegexMatch.cs.
5. If using VS 2010, right-click project name, Properties. Change .NET version from 4.0 down to 3.5 (not the client).
6. Modify or replace the function code (in the next figure).
7. Build then Build, Deploy.
8. In SQL Server Management Studio, open the database and enable CLR functions.

```
EXEC sp_configure 'show advanced options' , '1';  
reconfigure;  
EXEC sp_configure 'clr enabled' , '1' ;  
reconfigure;  
EXEC sp_configure 'show advanced options' , '0';  
reconfigure;
```

Figure 4.47

Steps to create a CLR project in SQL Server and enable CLR functions in the database.

write and test XQuery code for those times when a manager does need to search the data. As shown in the examples, it is possible to prebuild SQL views that contain XQuery searches. These views can be saved and run later. The tools include the ability to reference SQL parameter variables within the XQuery so these views can be controlled through other code. However, because of the complexity and tricky nature of XML queries, avoid giving users the ability to create their own XQuery searches.

### Regular Expressions (RegEx) Patterns

Increasingly, applications are being built that contain unstructured text data. For instance, a database might hold open-ended comments entered by workers or customers; or a database might be built to hold boilerplate paragraphs for use in contracts or negotiations. Think in terms of the open content on the Web, but it is data stored in internal databases. Now think about how people will want to search this data. Many times, they will want to enter keywords or phrases or even more complex conditions to find matches to sophisticated patterns. If the data consists of HTML or PDF pages stored on an internal Web server, it is possible to purchase commercial search engines to help index and find pages. But how are you going to create searches for text stored in a relational database? Basic SQL pattern matching was discussed in an earlier section of this chapter. It consists of two wildcard characters (% and \_ in the standard) that match any characters or any single character. This basic approach is not going to be enough. To address these issues, SQL 1999 added support for **regular expressions**, usually abbreviated RegEx. It has taken vendors a few years, but the big systems now support regular expression searches.

Regular expressions were created many years ago and were heavily used by programmers—particularly on UNIX-based systems. Today, most programming languages support them, and the big DBMSs also support their usage for matching text values. Microsoft Access does not support them for table searches; however, any code written in one of the Microsoft languages (Visual Basic, C#, C++ and so on) has regular expression processing which can be applied to the rows retrieved from queries.

Some systems, such as MySQL, support regular expressions directly as part of the query. For example, Oracle defines functions for `Regexp_Count`, `Regexp_Instr`, `Regexp_Like`, and `Regexp_Replace`. The `Regexp_Like` function is similar to an extended `Like` command used in the `WHERE` clause. SQL Server is more complicated to set up, but SQL server also has a simpler option that extends the standard `LIKE` command.

Regular expressions have powerful options supporting many complex types of searches. On the simple side, a pattern can search for a single word. More complex patterns can be created to see if an entered string is an e-mail address. Patterns can be written to search for repeating characters or phrases.

### *SQL Server Setup*

Beginning with SQL Server 2005, Microsoft added the ability to create user-defined **common-language runtime (CLR)** functions in SQL Server. CLR functions are written in a Visual Studio language, such as C#, compiled and installed into the database so that they can be used as functions within SQL. The process takes a few steps to set up the function, but once the function is installed, it is used

Figure 4.48

Microsoft C# function to create the `RegexMatch` function for use in SQL Server.

```
using System;
using System.Data;
using System.Data.SqlClient;
using System.Data.SqlTypes;
using System.Text.RegularExpressions;
using Microsoft.SqlServer.Server;

public partial class UserDefinedFunctions
{
    public static readonly RegexOptions Options =
        RegexOptions.IgnorePatternWhitespace |
        RegexOptions.Singleline;

    [Microsoft.SqlServer.Server.SqlFunction]
    public static SqlBoolean RegexMatch(
        SqlChars input, SqlString pattern)
    {
        Regex regex = new Regex( pattern.Value, Options );
        return regex.IsMatch( new string( input.Value ) );
    }
};
```

much like the functions in the other database systems. Figure 4.47 outlines the steps needed to create a CLR function and enable it to work within the database.

Figure 4.48 shows the entire C# code needed to create the `RegexMatch` function. Visual Studio generates much of the code automatically, but it needs to be edited. It might be easier to use copy-and-paste to replace the entire function.

Use the Build option to compile the function and correct any errors. Use the Build | Publish menu option and Visual Studio will compile the function and install it in the database. From that point, you can use the new function (`RegexMatch`) to examine any string or table column for patterns.

### *RegEx Patterns*

Regular expressions are powerful search tools, but they can be cryptic and hard to follow. This section presents only the basic concepts to get started. Once you are comfortable with these tools, you can use tutorials and reference documents on the Web to learn more detailed techniques if you need them. For example, `RegEx` also supports search and replace for patterns, but this option is not covered here.

The `regex` function uses two string parameters: (1) The text to be searched, and (2) A regex pattern. In a database context, the text to be searched can be a column from a table and the function can be written into the `SELECT` clause or the `WHERE` clause. For instance, consider searching the `Merchandise` table in the `Pet Store` database. Figure 4.49 illustrates using the `RegEx` function to search for any description containing the word “Small.” A plain string is the simplest search pattern. Note that patterns are case-sensitive by default. The simple pattern will match a row if the pattern exists exactly as written anywhere within the column data.

One useful tool of `RegEx` is the ability to specify a range of characters using square brackets. For example, as shown in Figure 4.50, the pattern `[a-z]` would match a single letter between “a” and “z” and only in lower-case form. The hyphen is a range indicator but it is also possible to specify individual characters. For example, the pattern `[AEIOU]` would match any one of the vowels in the list, or `[123]` would match one of the three digits.

The Microsoft SQL Server `LIKE` command also supports the square brackets—even without implementing the regular expression function. Hence, if all you need is a simple pattern to check for individual characters or ranges of characters, you

**Figure 4.49**

A simple text pattern search using `RegEx` in the `LIKE` clause. By default, `RegEx` comparisons are case-sensitive. Entering a simple string will try to match that pattern anywhere within the `Description` column.

```
SELECT *
FROM Merchandise
WHERE dbo.RegexMatch(Description, N'Small') <> 0;
```

1	Dog Kennel-Small	11	45.00	Dog
5	Cat Bed-Small	36	25.00	Cat
32	Collar-Dog-Small	47	12.00	Dog

[a-z]	Match one lower-case letter.
[AEIOU]	Match one of the vowels.
[123]	Match one of the numbers.
[^0-9]	Match a character not a digit.
<pre>SELECT * FROM Customer WHERE dbo.RegexMatch(LastName, N'H[ai]l') &lt;&gt; 0;</pre>	
78 (505) 646-2748 Elaine Hall ...	

Figure 4.50

Groups of characters using square brackets. Will match if a single character matches one of the pattern characters. The example would find customers with a last name of Hill or Hall. The caret (^) negates the pattern.

can use the square brackets directly within the LIKE clause. For example, consider the clause:

```
WHERE LastName LIKE N'Sm[iy]th'
```

This clause will match either Smith or Smyth because the brackets accept either the “i” or the “y” character.

RegEx has several special characters—many of which handle common ranges that are useful for various comparisons. Figure 4.51 shows the most commonly-used characters. Note the case-sensitive characters. Upper-case letters generally mean the negation or reverse of the lower-case symbol, such as “d” for digits and “D” for anything except digits. The caret (^) and dollar sign (\$) are useful because they anchor the string comparison to the start or end of the input text. Without these, the pattern is always tested at any point within the text string. For instance,

Figure 4.51

Special characters. Many of them match commonly-used ranges such as digits or alphanumeric characters.

.	(dot)	Match any single character.
\n		Match newline character.
\t		Match the tab character.
\d		Match a digit [0-9].
\D		Match a non-digit [^0-9].
\w		Match an alphanumeric character.
\W		Match a non-alphanumeric character.
\s		Match a whitespace character.
\S		Match a non-whitespace character.
\		Escape special characters, such as \.
^		Match the beginning of the input.
\$		Match the end of the input.

a simple pattern “One” would be tested and could appear at any point in the search text. However, the pattern “^One” will only match if the word “One” appears exactly at the start of the input.

So far, the patterns apply to a single character or word at a time. In many cases, it is useful to allow a digit or character to repeat. RegEx has several ways to quantify the number of characters. Three special characters are useful: \* ? +. The asterisk (\*) matches any number of what falls before it—from zero to infinity. The question mark (?) matches zero or one of the pattern preceding it. The plus sign (+) matches one or more of the pattern before it. Figure 4.52 summarizes these differences and provides an example of the difference between the asterisk and the plus sign. You should run the two queries and check the results. Almost all of the merchandise rows will appear when using the asterisk because the “n” is optional and most rows contain the letter “e”.

These special characters cover cases of zero, one, and infinite repetition. In some cases, you want the ability to specify exactly how many times a pattern should repeat. The RegEx pattern for that is to put the number in curly braces, such as: \d{3}. The letter \d specifies a numeric digit and the {3} repetition annotation says exactly three digits must exist. Figure 4.53 shows an example of using the fixed repetition to test a U.S. Social Security number. This example comes from Microsoft’s MSDN article. U.S. Social Security numbers consist of nine digits, commonly written in three groups separated by hyphens, such as 123-45-6789. The RegEx pattern is straightforward. For example, the term \d{3} tests for the presence of exactly three digits. In the full pattern, note the use of the start (^) and end (\$) markers to prevent the introduction of extraneous characters.

RegEx patterns can be much more complex. One useful feature is the ability to group characters together using parentheses (). Figure 4.54 shows some straightforward examples of creating groups. Anything placed in parentheses is treated as a group, so the pattern (ab)+ searches for at least one occurrence of the two

**Figure 4.52**

Repetition Characters. The characters apply to the pattern immediately preceding the character. Notice that the asterisk is less useful than it appears because the zero means the pattern does not have to exist.

- \* Match zero or infinite of the pattern before the asterisk.
- ? Match zero or one of the pattern before.
- + Match one or more of the pattern before.

```
SELECT *
FROM Merchandise
WHERE dbo.RegexMatch(Description, N'n*e')<>0;
>>>> Match any description that contains the letter “e”.
(Because the * means the n is optional.
```

```
SELECT *
FROM Merchandise
WHERE dbo.RegexMatch(Description, N'n+e')<>0;
>>>> Match any description that contains the letter “n” followed by any
characters and then the letter “e”.
(Because the + means the n is required.
```



{#} such as {3} Exact number of repetitions.

```
SELECT dbo.RegexMatch(N'123-45-6789', N'^\d{3}-\d{2}-\d{4}$')
```

Pattern to test a U.S. Social Security Number. The string must begin with 3 digits, then a hyphen, then 2 digits, another hyphen, and end with exactly 4 digits.

### Figure 4.53

Exact repetition. Enter the exact number of repetitions in curly braces. A useful method when a data element must have an exact number of digits, spaces, or characters.

characters “ab” together. The figure also shows that grouping is more powerful when the “Or” connector ( | ) is added. The pattern (aa|bb)+ searches for at least one occurrence of either “aa” or “bb” (or both). Be careful to note the difference between square brackets and parentheses. Brackets represent a single character to be matched while parentheses require the entire term to be matched. In the example, [ab] would match either the letter a or b, so the string “acb” would be matched as true because it contains an “a” (and a “b”). However the pattern (ab) does not match the string “acb” because the parentheses require an exact match of the entire term and the pattern “ab” does not exist in the string “acb”.

### Summary

This section is merely an introduction to regular expressions. Several other features exist, and you can find many tutorials and reference works on the Web. However, before trying to learn, and memorize, the many features of expressions you need to practice the simpler versions so that you completely understand these features. Regular expressions are often combined into long, complex pattern strings. These patterns can be difficult to read and debug. They are even harder when someone else has written the pattern. Whenever you create regular expression patterns, be sure you document them and explain the objectives. Better yet, you should always create sample test cases before you try to write a RegEx pattern. Create or find real-world sample data that includes several cases that you do and do not want to match. As you build the pattern, you can test it in sections against the sample data. More importantly, if anyone modifies the patterns later, they can be re-tested against the data to ensure they are still correct.

Be cautious when creating regular expressions—particularly if they are intended for use at restricting data entry. For example, it is tempting to create a pattern to force people to enter telephone numbers in a specific manner. But, what hap-

### Figure 4.54

Grouping patterns with parentheses and using the Or connector: |.

(ab)+ matches ab, abab, tab, but not acb.

(aa|bb)+ matches aa, bbcc, bbddaa, but not abab.

pens when someone needs to enter a phone number that does not match the pattern? For example, international phone numbers require more digits (international code), and generally do not follow the same pattern as U.S. phone numbers. Similarly, be cautious writing patterns for e-mail addresses. Data formats and usages change over time.

Regular expressions are powerful tools, but they carry a price. Because of their complexity, they are difficult to optimize and rarely used with indexes. In most cases, the query processor needs to retrieve every single row, apply the pattern, and then decide whether the row is included in the results. This process can be time consuming if the query retrieves millions, billions, or trillions of rows of data. When users truly need to search everything, the delay is probably acceptable. However, it is best to try and write a query without using regular expressions—particularly for queries with multiple tables and JOINS.

## Summary

---

The key to creating a query is to answer four questions: (1) What output do you want to see? (2) What constraints do you know? (3) What tables are involved? (4) How are the tables joined? The essence of creating a query is to use these four questions to get the logic correct. The WHERE clause is a common source of errors. Be sure that you understand the objectives of the query. Be careful when combining OR and AND statements and use DeMorgan's law to simplify the conditions.

Always test your queries. The best method to build complex queries is to start with a simpler query and add tables. Then add conditions one at a time and check the output to see whether it is correct. Finally, enter the computations and GROUP BY clauses. When performing computations, be sure that you understand the difference between Sum and Count. Remember that Count simply counts the number of rows. Sum produces the total of the values in the specified column.

Joining tables is straightforward. Generally the best approach is to use QBE to specify the columns that link the tables and then check the syntax of the SQL command. Remember that JOIN columns can have different names. Also remember that you need to add a third (or fourth) table to link two tables with no columns in common. Keep the class diagram handy to help you determine which tables to use and how they are linked to each other.

### **A Developer's View**

As Miranda noted, SQL and QBE are much easier than writing programs to retrieve data. However, you must still be careful. The most dangerous aspect of queries is that you may get a result that is not really an answer to the business question. To minimize this risk, build queries in pieces and check the results at each step. Be particularly careful to add aggregation and GROUP BY clauses last, so that you can see whether the WHERE clause was entered correctly. If you name your columns carefully, it is easier to see how tables should be joined. However, columns do not need the same names to be joined. For your class project, you should identify some common business questions and write queries for them.

## Key Terms

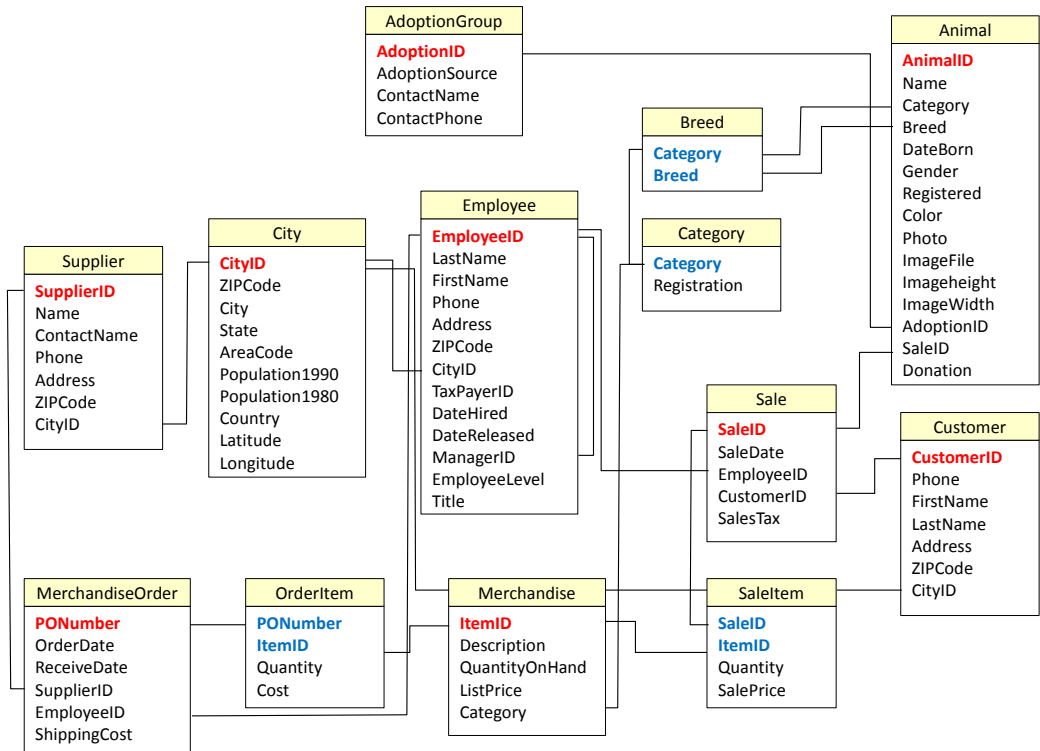
---

aggregation	JOIN
alias	LIKE
BETWEEN	NOT
Boolean algebra	NULL
common-language runtime (CLR)	ORDER BY
cross join	query by example (QBE)
data definition language (DDL)	regular expression (RegEx)
data manipulation language (DML)	row-by-row calculations
DeMorgan's law	SELECT
DESC	SQL
DISTINCT	TOP
FETCH	view
FROM	WHERE
GROUP BY	XML schema
HAVING	XQuery

## Review Questions

---

1. What are the three basic tasks of a query language?
- ✓ 2. What are the four questions used to create a query?
3. What is the basic structure of the SQL SELECT command?
4. What is the purpose of the DISTINCT operator?
5. Why is it important to use parentheses in complex (Boolean) WHERE clauses?
6. How is pattern matching used to select rows of data?
7. What is DeMorgan's law, and how does it simplify conditions?
8. How do you compute subtotals using SQL?
- ✓ 9. How do the basic SQL arithmetic operators (+, -, etc.) differ from the aggregation (SUM, etc.) commands?
10. What basic aggregation functions are available in the SELECT command?
- ✓ 11. What is the difference between Count and Sum? Give an example of how each would be used.
12. What is the difference between the WHERE and HAVING clauses? Give an example of how each would be used.
13. What is the SQL syntax for joining two tables?
14. How do you identify a column when the same name appears in more than one table?
15. What is XQuery and when would you use it?
16. What are regular expressions? What are their strengths and weaknesses?



## Exercises



### Sally's Pet Store



1. Which employee still working for the store was hired the most recently?
2. What is the largest quantity of items ever ordered/purchased by the store at one time?

3. List all cats with no black in their coloring.
4. List customers from Tennessee (TN) who bought cat merchandise.



5. List employees who participated in adoptions of female dogs in March.
6. List customers who bought a dog kennel in March.
7. List the name and contact information for suppliers in Nebraska (NE).
8. List the items sold in May with no duplicates.
9. List the name and phone number of each customer who adopted an animal in February.



10. List the adoption groups with phone number who placed cats in October.
11. List all of the employees who are managed by Katy Reasoner.
12. What is the largest value of sale ever made?

13. Which adoption group has placed the most animals?



14. Which day of the week (Sun/Mon/...) has the highest total sales value? Hint: In Access use the function Format(date, "ddd") to obtain the day of the week.

15. Are male dogs more likely to be registered than the females?

16. What was the most popular item sold in May (by quantity)?



17. By total value, which supplier has the highest sales for the year?

18. Does the store have more money tied up in inventory (quantity on hand) for dog items or cat items?

19. On average by supplier, how long does it take to receive an order from suppliers?

20. By value, which category of items were sold the most in the second quarter of the year?

21. Which employee sold the most quantity of items in March?

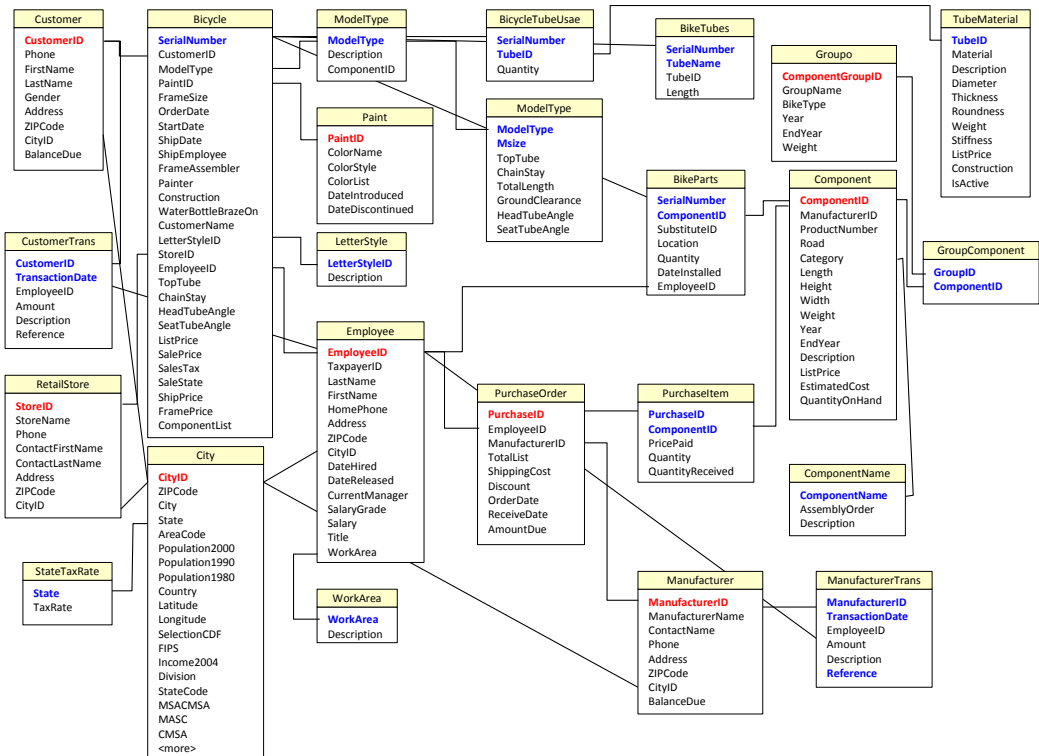


22. By count of state, where are most of the customers located?

23. Which category of items had the highest sales value in May?

24. From which supplier did the store purchase the most cat merchandise by value?






25. Which sale had the highest total discount (ListPrice – SalePrice)\*Quantity?

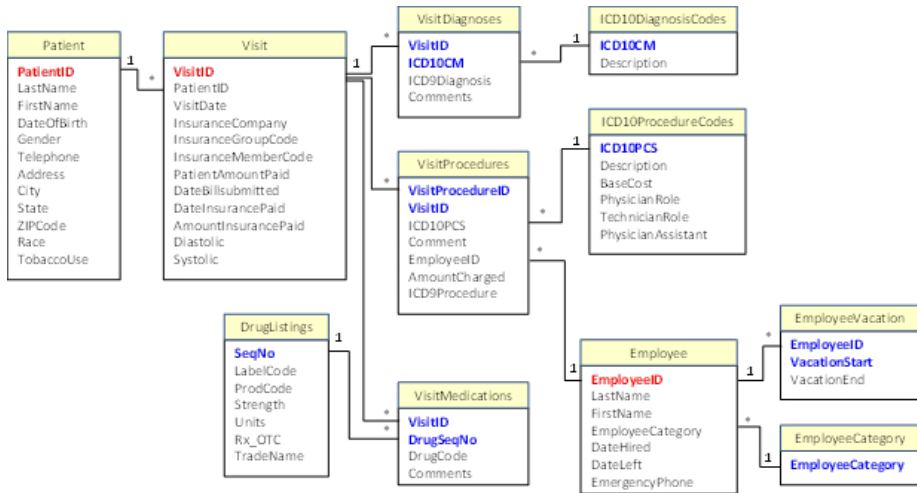


## Rolling Thunder Bicycles

Write the SQL statements that will answer questions 26 through 50 based on the tables in the Rolling Thunder database.

26. List customers (name, phone) who bought race bikes in 2012 with a frame size greater than 60 cm.
27. List the component product number and weight that are in the SRAM Red 2012 groupo.
28. Which full suspension bikes sold in 2012 were equipped with SRAM (manufacturer) cranks?
29. List the employees who sold race bikes with a sale price of more than \$9000 in 2010.
30. List the retail stores (ID > 2) that participated in selling hybrid bikes in 2012.
31. List the phone number of all women who purchased race bikes with white in the color in 2012.
32. For future correlation analysis, list the sale price, most recent population and per capita income for the city where it was purchased in 2013.
33. List all of the employees who placed purchase orders with Shimano (manufacturer) in 2012 with a total list value over 120,000.


34. Looking for tall riders, list all customers who purchased road bikes in 2012 with a frame size greater than or equal to 62 cm for men and 60 cm for women.
-  35. Find the greatest percentage discount (Discount/TotalList) received on a purchase order placed in 2013.
36. Compute the total price of components installed on each Road bike sold in 2013 (without using the ComponentList value in the Bicycle table).
37. Compute the average price paid for Campy Record 11 cranks purchased in 2013.
38. What was the most popular letter style in 2012 on all Mountain bikes?
-  39. How many Race bikes were sold to women in each state in 2013?
40. On average, not counting frames, what is the most expensive (list price) category of component carried for the 2012 component year?
41. In 2012, which model type carried the highest average sale price?
42. For the years 2010-2013 (inclusive), did men or women pay higher average prices for Road bikes?
-  43. Compute the total value and count of sales by month for 2010 through 2013. Hint: In Access, use the format string “yyyy-mm”, or Year(...)\*100+Month(...) for others.
-  44. Which customer purchased the most number of Road and Race bikes combined (for all dates)?
45. Show how the average weight of rear derailleurs for Road bikes has changed over time.
46. What is the total tax value collected and owed to each state for bikes sold in 2013?
-  47. Which employee sold the most bikes by value in November 2012?
48. What is the average percentage of shipping cost per total list by manufacturer for orders placed in 2013?
49. For 2013, did the average percent discount on bike prices vary significantly based on model type?
50. Which manufacturer made the most popular crank installed on Race bikes in 2013?




### Corner Med

51. Which patient paid the most amount of money directly for a single visit in April 2013?
52. List all of the drugs prescribed on June 24, 2013.
53. List the employees who treated patients on July 4, 2013; without duplicates.
- ✓ 54. List the female patients who do not use tobacco and were between 50 and 55 at the time of the visit. Hint: Use the DateDiff function to compute the age.
55. List all of the procedures and amounts charged for patients treated by Dr. Johnson on May 15.
- ✓ 56. List any patients who were older than 60 and had a systolic pressure below 120 (but not zero).
57. List all of the ICD10 diagnostic codes that refer to the esophagus.
58. List the visits where the patient paid nothing up front and the insurance company took longer than 90 days to pay the bill (DateInsurancePaid – DateBillSubmitted).
59. List all of the physicians who prescribed the drug Ambien in October.
- ✓ 60. List all of the patients (name and phone) seen by Dr. Sanchez in February.
61. What was the most commonly used insurance company in March?
62. What was the highest total amount billed for a single visit in August?
63. What were the total amounts charged for each week in the year?
64. Compare the total number of visits by day of the week for the year.
- ✓ 65. Based on the first seven letters of the trade name, what was the most commonly prescribed drug in July?
66. For December, which group performed more total procedures: Physicians or everyone else?





67. Which employee took the most total vacation time for the year?
68. Which patient was prescribed the most number of drugs (count) in September?
-  69. Which 2-level ICD10 treatment procedure was most common in January – March?
70. What was the total amount charged to each insurance company for the month of November?

### XQuery

71. Create the ShippingInvoice table and add the sample data. Run the sample queries from the figures and verify the results.
-  72. Create an XQuery that retrieves the items with a quantity of more than 20.
73. Create an XQuery that retrieves the price for any description equal to “Collar.”
74. Create an XQuery that retrieves the Item Descriptions shipped where ShippingID=1 and ItemID is 15. Use the text() function to return just the value without the tags.
75. Write an XQuery using the “for” statement that returns all of the ItemIDs in ascending order from the first shipment. Hint: The result should be: `<ItemID>15</ItemID><ItemID>32</ItemID>`.

### Regular Expressions

76. If you are using SQL Server, write the code to create and deploy the Visual Studio C# RegExMatch function and enable the CLR code in the Pet Store database. In all cases, create and test the RegEx expressions from the figures in the chapter.
-  77. Create a regular expression pattern that retrieves all Merchandise items that refer to dry food which might be written as in either order (dry...food or food...dry).
78. Create a regular expression pattern that retrieves all Merchandise items with a weight of 10 pounds. Do not include 100 pound items and do not assume there is a space after the 10.
79. Get a list of customers who have a street address that contains less than 4 digits. Hint: The number must appear at the start of the Address and look up the options for the { } repeating specification.
-  80. Check the breed entries to list all of the Terriers, but also check for misspellings that use only a single “r” (Terier).
81. Using the CornerMed database and RegEx, list all of the Descriptions in the ICD10 procedures that include the words (Left, Artery, and Endoscopic) in any order. Hint: Search for RegEx lookahead examples.
82. Using the CornerMed database and RegEx, list all of the patients with a last name of McCarthy; which might be spelled with or without a space between the “C”s and might have only one C.

## Web Site References

---

<a href="http://www.jcc.com/sql.htm">http://www.jcc.com/sql.htm</a>	Blog on SQL Standards
<a href="http://jtc1sc32.org/">http://jtc1sc32.org/</a> <a href="http://www.wiscorp.com/SQLStandards.html">http://www.wiscorp.com/SQLStandards.html</a>	Standards documents. Free versions of some drafts.
<a href="http://www.sqlmag.com">http://www.sqlmag.com</a>	Magazine with SQL emphasis.
<a href="http://www.sqlteam.com">http://www.sqlteam.com</a>	SQL hints and comments.
<a href="http://www.sqlcourse.com">http://www.sqlcourse.com</a>	Online SQL notes.
<a href="http://www.w3.org/TR/xquery/">http://www.w3.org/TR/xquery/</a>	XQuery reference
<a href="http://docs.oracle.com/cd/E13214_01/wli/docs92/xref/xqlangxml.html">http://docs.oracle.com/cd/E13214_01/wli/docs92/xref/xqlangxml.html</a>	Oracle XML documentation
<a href="http://www.aivosto.com/vbtips/regex.html">http://www.aivosto.com/vbtips/regex.html</a>	One regex tutorial
<a href="http://msdn.microsoft.com/en-us/magazine/cc163473.aspx">http://msdn.microsoft.com/en-us/magazine/cc163473.aspx</a>	Microsoft article with the CLR regex function.

## Additional Reading

---

Gulutzan, P. and T. Pelzer, *SQL-99 Complete, Really*, Gilroy, CA: CMP Books, 2000. [In depth presentation of the SQL-99/SQL3 standard.]

Melton, J. and A. R. Simon. *SQL 1999: Understanding Relational Language Components*, 2002. San Mateo: Morgan Kaufmann Publishers, 1993. [An in-depth presentation of SQL 1999, by those who played a leading role in developing the standard.]

## Appendix: SQL Syntax

### SQL Commands

**Alter Table**

```
ALTER TABLE table
    ADD COLUMN column datatype (size)
    DROP COLUMN column
```

**Commit Work**

```
COMMIT WORK
```

**Create Index**

```
CREATE [UNIQUE] INDEX index
ON table (column1, column2, ...)
WITH {PRIMARY | DISALLOW NULL | IGNORE NULL}
```

**Create Table**

```
CREATE TABLE table
(
    column1 datatype (size) [NOT NULL] [index1],
    column2 datatype (size) [NOT NULL] [index2],
    ... ,
    CONSTRAINT pkname PRIMARY KEY (column, ...),
    CONSTRAINT fkname FOREIGN KEY (column)
        REFERENCES existing_table (key_column)
        ON DELETE CASCADE
)
```

**Create Trigger**

```
CREATE TRIGGER triggername { BEFORE | AFTER }
    {DELETE | INSERT | UPDATE}
ON table { FOR EACH ROW }
    { program code block}
```

**Create View**

```
CREATE VIEW viewname AS
SELECT ...
```

**Delete**

```
DELETE
FROM table
WHERE condition
```

**Drop Index**

```
DROP INDEX index ON table
```

**Drop Table**

```
DROP TABLE table name
```

**Drop Trigger**

```
DROP TRIGGER trigger name
```

**Drop View**

```
DROP VIEW view name
```

**Insert**

```
INSERT INTO table (column1, column2, ...)  
VALUES (value1, value2, ...)
```

**Insert (copy multiple rows)**

```
INSERT INTO newtable (column1, column2, ...)  
SELECT ...
```

**Grant**

```
GRANT privilege  
ON object  
TO user | PUBLIC
```

**Revoke**

```
REVOKE privilege  
ON object  
FROM user | PUBLI
```

**Privileges for Grant and Revoke**

```
ALL, ALTER, DELETE, INDEX,  
INSERT, SELECT, UPDATE
```

**Rollback**

```
ROLLBACK WORK  
TO savepoint
```

**SavePoint**

```
SAVEPOINT savepoint
```

**Select**

```
SELECT DISTINCT table.column {AS alias}, ...  
FROM table/view  
INNER JOIN table/view ON T1.ColA = T2.ColB  
WHERE (condition)  
GROUP BY column  
HAVING (group condition)  
ORDER BY table.column  
{ UNION, INTERSECT, EXCEPT, ... }
```

**Select Into**

```
SELECT column1, column2, ...  
INTO newtable  
FROM tables  
WHERE condition
```

**Update**

```
UPDATE table  
SET column1 = value1, column2 = value2, ...  
WHERE condition
```