

Queries

An important step in building applications is creating queries to retrieve exactly the data that you want. Queries are used to answer business questions and serve as the foundation for forms and reports.

Chapter 4 shows you how to use two basic query systems: SQL and QBE. SQL has the advantage of being a standard that is supported by many database management systems. Once you learn it, you will be able to work with many different systems.

Chapter 5 shows some of the powerful aspects of SQL queries. In particular, it examines the use of subqueries to answer difficult business questions. It also shows that SQL is a complete database language that can be used to define new databases and tables. SQL is also a powerful tool to manipulate data.

Chapter 4. Data Queries

Chapter 5. Advanced Queries and Subqueries

Data Queries

Chapter Outline

Overview
Introduction
Three Tasks of a Query Language
Four Questions to Retrieve Data
 What Output Do You Want to See?
 What Do You Already Know?
 What Tables Are Involved?
 How Are the Tables Joined?
Sally's Pet Store
Vendor Differences
Query Basics
 Single Tables
 Introduction to SQL
 Sorting the Output
 Distinct
 Criteria
 Boolean Algebra
 DeMorgan's Law
 Useful WHERE Clauses
Computations
 Basic Arithmetic Operators
 Aggregation
 Functions
Subtotals and GROUP BY
 Conditions on Totals (HAVING)
 WHERE versus HAVING
 The Best and the Worst
Multiple Tables
 Joining Tables
 Identifying Columns in Different Tables
 Joining Many Tables
 Hints on Joining Tables
 Table Alias
 Create View

Summary
Key Words
Review Questions
Exercises
Website References
Additional Reading
Appendix: SQL Syntax
 ALTER TABLE
 COMMIT WORK
 CREATE INDEX
 CREATE TABLE
 CREATE TRIGGER
 CREATE VIEW
 DELETE
 DROP
 INSERT
 GRANT
 REVOKE
 ROLLBACK
 SELECT
 SELECT INTO
 UPDATE

What You Will Learn in This Chapter

- How do you get answers to business questions from a database? ●●●
- Why do you need to learn a special query language to use a database system? ●●●
- What are the four questions that you need to answer to create a query? ●●●
- How can you perform computations with a query? ●●●
- How can you compare results by groups? ●●●
- Why and how do you join tables in a query? ●●●

Overview

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.

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 would still 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 at least some of the SQL-99 standard, but a few still use the SQL-92 version. The ISO working group is developing a new standard, but it will probably be a few years before it becomes implemented. 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.

Most database systems also provide a **query by example (QBE)** method 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.

Three 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. 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. Virtually all tasks can be performed by issuing a DDL, DML, or query command. Modern systems simplify some of the database administration tasks (such as creating a table) by providing a graphical interface. The interface actually builds the appropriate CREATE TABLE command.

Four Questions to Retrieve Data

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. You need to remember these four questions, but do not worry about the specific order. When you first learn to create queries, you should write down these four questions each time you construct a query. With easy problems, you can almost automatically fill in

FIGURE 4.1

Four questions to create a query. Every query is built by asking these four questions.

- What output do you want to see?
- What do you already know (or what constraints are given)?
- What tables are involved?
- How are the tables joined?

answers to these questions. With more complex problems, you might fill in partial answers and switch between questions until you completely understand the query.

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 (question 3). 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 (question 2).

What Output Do You Want to See?

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.

The query system can generate aggregations, such as totals and averages. Similarly, the computer can perform basic arithmetic operations (add, subtract, multiply, and divide) on numeric data.

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.

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, an Order 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. However, the 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.

Sally's Pet Store



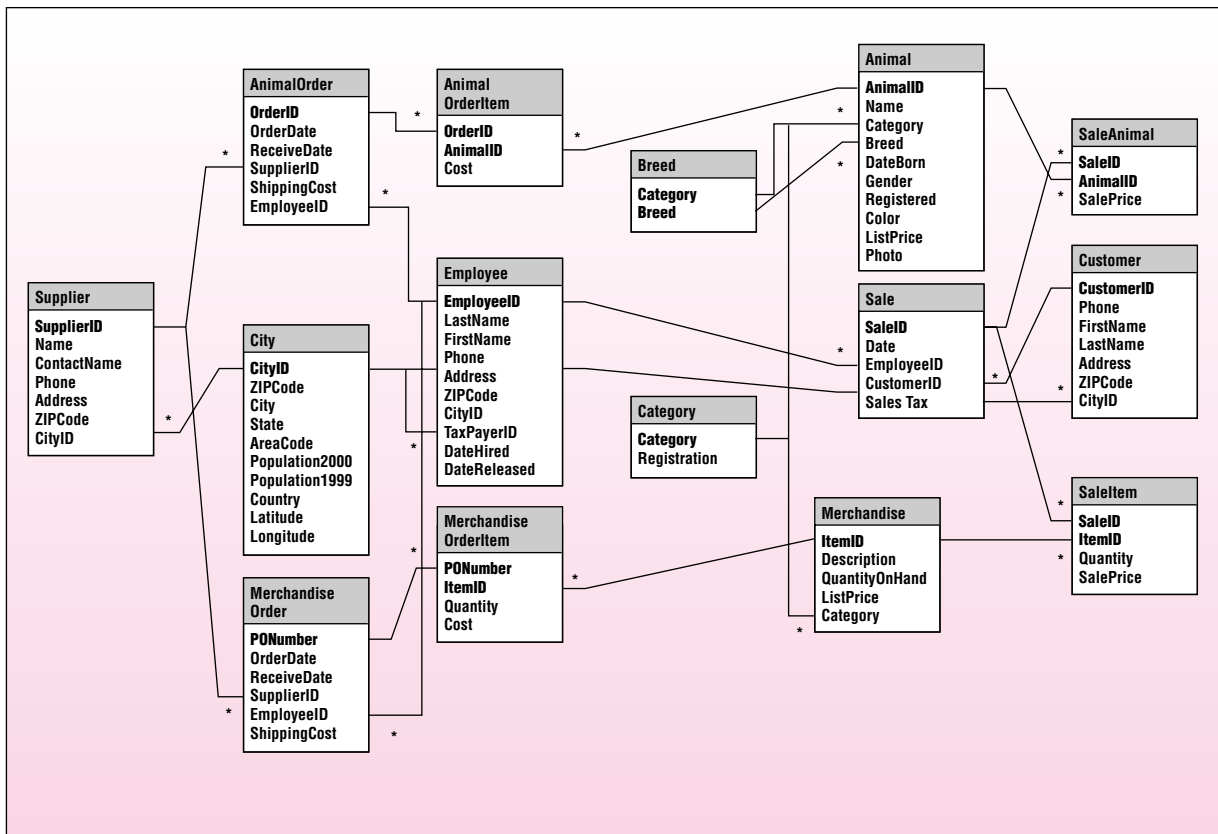
The initial Pet Store database has been built, and some basic historical data has been transferred from Sally's old files. When you show your work to Sally, she becomes very excited. She immediately starts asking questions about her business and wants to see how the database can answer them.

The examples in this chapter are derived from the Pet Store database. The tables and relationships for this case are shown in Figure 4.2. 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.

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 a little differently than it treats

FIGURE 4.2

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



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 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 also 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

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, there are substantial differences in the database products. These differences are even more pronounced when you look at the graphical interfaces. To remain up to date, the presentation in this chapter (and the next one) will follow the most recent SQL standards. The current versions of most systems now support many of the SQL standards. For example, Oracle 9i supports the more modern JOIN syntax for multiple tables.

Query Basics

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 4.3 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.

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, List all animals with yellow in their color. Note that an animal could have many colors. The designer of this database has chosen to store all the colors in one column of the database. So an animal's

FIGURE 4.3

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.

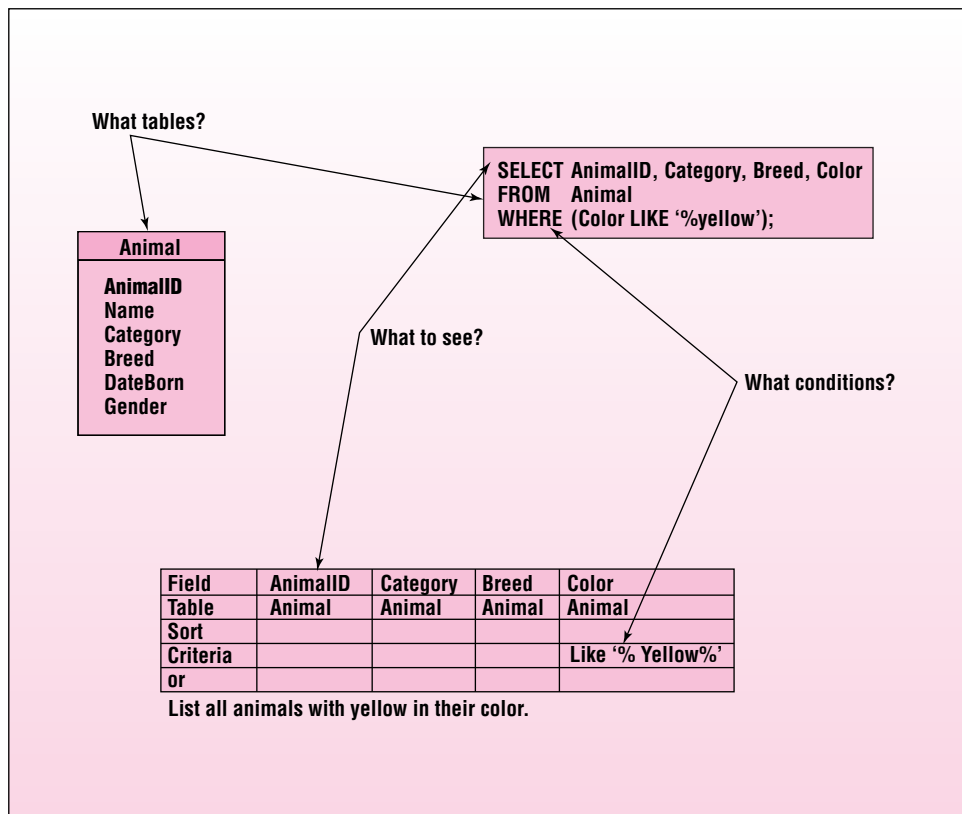
- List all animals with yellow in their color.
- List all dogs with yellow in their color born after 6/1/04.
- List all merchandise for cats with a list price greater than \$10.
- List all dogs who are male and registered or who were born before 6/1/04 and have white in their color.
- What is the average sale price of all animals?
- What is the total cost paid for all animals?
- List the top 10 customers and the total amount they spent.
- How many cats are in the animal list?
- Count the number of animals in each category.
- List the CustomerID of everyone who bought something between 4/1/04 and 5/31/04.
- List the first name and phone of every customer who bought something between 4/1/04 and 5/31/04.
- List the last name and phone of anyone who bought a registered white cat between 6/1/04 and 12/31/04.
- Which employee has sold the most items?

colors could be described as “yellow, white, brown.” Presumably, the primary color is listed first, but there is no mechanism to force the data to be entered that way.

First consider answering this question with a QBE system, as shown in Figure 4.4. The QBE system asks you to choose the tables involved. This

FIGURE 4.4

Sample query shown in QBE and SQL. Since there is only one table, only three questions need to be answered: What tables? What do you want to see? What conditions?



situation involves only one table: Animal. Note that all the output columns come from the Animal table. Similarly, the Color column in the criteria is also 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, Category, Breed, and the Color.

The next step is to enter the criteria that you already know. In this example, you are looking for animals with yellow in their color. On the QBE screen enter the condition “yellow” under the Color column. However, there is one catch: The Color column generally contains more than one word. For some animals, yellow might show up as the second or third color in the list. To match the animal regardless of where the word *yellow* is located, you need to use the LIKE pattern-matching function. By entering the condition LIKE ‘%yellow%’, you are asking the query system to match the word *yellow* anywhere in the list (with any number of characters before or after the word). It is a good idea to run the query now. Check the Color column to make sure the word *yellow* appears somewhere in the list.

Introduction to SQL

SQL is a powerful query language. However, unlike QBE, you generally have to type in the entire statement. Some systems like Microsoft Access 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. 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.”

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 in Figure 4.5, 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 4.4, notice the similarity between the QBE and SQL approaches.

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 4.6, 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 4.6 are sorted by Category. In these situations you would want to add a second sort column. In the example, rows for each

FIGURE 4.5

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.

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 4.6

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.

<table border="1" style="border-collapse: collapse; width: 100%;"> <tr><td style="text-align: center;">Animal</td></tr> <tr><td>AnimalID</td></tr> <tr><td>Name</td></tr> <tr><td>Category</td></tr> <tr><td>Breed</td></tr> <tr><td>DateBorn</td></tr> <tr><td>Gender</td></tr> </table>	Animal	AnimalID	Name	Category	Breed	DateBorn	Gender	<pre>SELECT Name, Category, Breed FROM Animal ORDER BY Category, Breed</pre>	<table border="1" style="border-collapse: collapse; width: 100%;"> <tr><td>Name</td><td>Category</td><td>Breed</td></tr> <tr><td>Cathy</td><td>Bird</td><td>African Grey</td></tr> <tr><td></td><td>Bird</td><td>Canary</td></tr> <tr><td>Debbie</td><td>Bird</td><td>Cockatiel</td></tr> <tr><td></td><td>Bird</td><td>Cockatiel</td></tr> <tr><td>Terry</td><td>Bird</td><td>Lovebird</td></tr> <tr><td></td><td>Bird</td><td>Other</td></tr> <tr><td>Charles</td><td>Bird</td><td>Parakeet</td></tr> <tr><td>Curtis</td><td>Bird</td><td>Parakeet</td></tr> <tr><td>Ruby</td><td>Bird</td><td>Parakeet</td></tr> <tr><td>Sandy</td><td>Bird</td><td>Parrot</td></tr> <tr><td>Hoyt</td><td>Bird</td><td>Parrot</td></tr> <tr><td></td><td>Bird</td><td>Parrot</td></tr> </table>	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
Animal																																																
AnimalID																																																
Name																																																
Category																																																
Breed																																																
DateBorn																																																
Gender																																																
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																																														
<table border="1" style="border-collapse: collapse; width: 100%;"> <tr><td>Field</td><td>Name</td><td>Category</td><td>Breed</td></tr> <tr><td>Table</td><td>Animal</td><td>Animal</td><td>Animal</td></tr> <tr><td>Sort</td><td></td><td>Ascending</td><td>Ascending</td></tr> <tr><td>Criteria</td><td></td><td></td><td></td></tr> <tr><td>Or</td><td></td><td></td><td></td></tr> </table>	Field	Name	Category	Breed	Table	Animal	Animal	Animal	Sort		Ascending	Ascending	Criteria				Or																															
Field	Name	Category	Breed																																													
Table	Animal	Animal	Animal																																													
Sort		Ascending	Ascending																																													
Criteria																																																
Or																																																

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 4.7 (*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*

FIGURE 4.7

The DISTINCT keyword eliminates duplicate rows of the output. Without it the animal category is listed for every animal in the database.

<pre>SELECT Category FROM Animal;</pre> <table border="1" style="border-collapse: collapse; width: 100%;"> <tr><td>Category</td></tr> <tr><td>Fish</td></tr> <tr><td>Dog</td></tr> <tr><td>Fish</td></tr> <tr><td>Cat</td></tr> <tr><td>Cat</td></tr> <tr><td>Dog</td></tr> <tr><td>Fish</td></tr> <tr><td>Dog</td></tr> <tr><td>Dog</td></tr> <tr><td>Dog</td></tr> <tr><td>Fish</td></tr> <tr><td>Cat</td></tr> <tr><td>Dog</td></tr> <tr><td>...</td></tr> </table>	Category	Fish	Dog	Fish	Cat	Cat	Dog	Fish	Dog	Dog	Dog	Fish	Cat	Dog	...	<pre>SELECT DISTINCT Category FROM Animal;</pre> <table border="1" style="border-collapse: collapse; width: 100%;"> <tr><td>Category</td></tr> <tr><td>Bird</td></tr> <tr><td>Cat</td></tr> <tr><td>Dog</td></tr> <tr><td>Fish</td></tr> <tr><td>Mammal</td></tr> <tr><td>Reptile</td></tr> <tr><td>Spider</td></tr> </table>	Category	Bird	Cat	Dog	Fish	Mammal	Reptile	Spider
Category																								
Fish																								
Dog																								
Fish																								
Cat																								
Cat																								
Dog																								
Fish																								
Dog																								
Dog																								
Dog																								
Fish																								
Cat																								
Dog																								
...																								
Category																								
Bird																								
Cat																								
Dog																								
Fish																								
Mammal																								
Reptile																								
Spider																								

DISTINCT Category, Breed FROM Animal will return more than the seven rows shown in Figure 4.7 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 an “answer.” 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 = ‘Dog’)** means you are interested in all animals except dogs.

Consider the example in Figure 4.8. The first step is to note that three conditions define the business question: dog, yellow, and date of birth. The second step is to recognize that all three 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 all three clauses. If any one clause is false, the row is skipped.

Notice that the SQL statement is straightforward—just write the criteria. 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.

FIGURE 4.8

Boolean algebra. An example of three conditions connected by **AND**. Notice the # signs surrounding the date. They are a convention used by Microsoft Access to help it recognize a date. They are particularly useful if you want to enter a text date (e.g., June 1, 2004).

	Animal			
	AnimalID Name Category Breed DateBorn Gender	SELECT AnimalID, Category, DateBorn FROM Animal WHERE ((Category = 'Dog') AND (Color Like '%Yellow%') AND (DateBorn > '01-Jun-2004'));		

Field	AnimalID	Category	DateBorn	Color
Table	Animal	Animal	Animal	Animal
Sort				
Criteria	'Dog'		>'01-Jun-2004'	Like '%Yellow%'
Or				

List all dogs with yellow in their color born after 6/1/04.

FIGURE 4.9

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.

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

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, 2004, 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 4.9 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 4.10. The first condition is false because it asks for both clauses to be true, and the first one is false. 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.

FIGURE 4.10

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

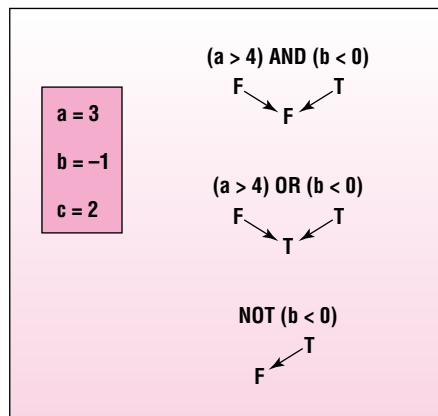
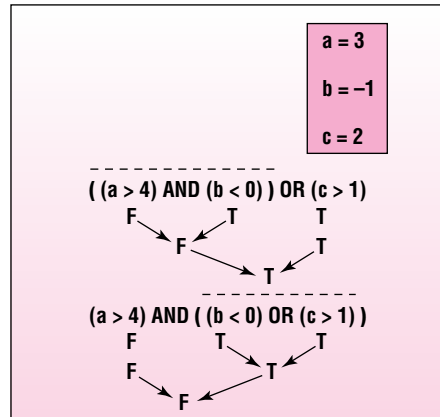


FIGURE 4.11

Boolean algebra mixing AND and OR operators. The result changes depending on which operator is applied first. You must set the order of evaluation with parentheses. Innermost clauses are evaluated first.



As shown in Figure 4.11, 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 4.11, 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 4.11, 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.” There are two interpretations of this statement:

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

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 4.12. A customer might

FIGURE 4.12

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.

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."

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

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

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 4.13 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 4.13 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 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.

FIGURE 4.15

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 6/1/2004 and have white in their color.

- 1: (male and registered) or (born after 6/1/2004 and white)
- 2: (male) and (registered or born after 6/1/2004) and (white)

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 4.16. Standard numeric data can be compared with equality and inequality operators. Text comparisons are usually made with the **LIKE** operator for pattern matching. The SQL standard, supported by Oracle and SQL Server, uses the percent sign (%) to match any number of characters and the underscore (_) to match exactly one character. Microsoft Access uses an asterisk (*) and a question mark (?) instead. The single-character match (?) is particularly useful for searches involving defined text strings like product numbers. For example, product numbers might be defined as DDDCCC9999 where the first three characters represent the department, the

FIGURE 4.16

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

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-2004' AND '31-Aug-2004'
Missing data	City IS NULL
Negation	Name IS NOT NULL
Sets	Category IN ('Cat', 'Dog', 'Hamster')

next three the product category, and the last four digits are a unique number. Then to find all products that refer to the Dog category you could use the WHERE condition: `ProductID LIKE "???dog???"`. Most systems also provide an option that controls whether text comparisons are sensitive to upper- and lowercase. Many people prefer to ignore case, since it is easier to type words without worrying about case.

The **BETWEEN** clause is not required, but it saves some typing and makes some conditions a little clearer. The clause `(SaleDate BETWEEN '15-Aug-2004' AND '31-Aug-2004')` is equivalent to `(SaleDate >= '15-Aug-2004' AND SaleDate <= '31-Aug-2004')`. 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('8/15/04','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.

Computations

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 of data storage. Consider a common order or sales form. As Figure 4.17 shows, the basic tables would include a list of items purchased: `OrderItem(OrderID, ItemID, Price, Quantity)`. In most situations you would need to multiply Price 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: `Price * Quantity AS Extended`. Remember that the computations are performed for each row in the query.

Some systems provide additional mathematical functions. For example, basic mathematical functions such as absolute value, logarithms, and trigonometric functions might be 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.

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 4.18 can operate across several rows of

FIGURE 4.17

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

```
OrderItem(OrderID, ItemID, Price, Quantity)

Select OrderID, ItemID, Price, Quantity,
Price*Quantity As Extended
From OrderItem;
```

OrderID	ItemID	Price	Quantity	Extended
151	9764	19.50	2	39.00
151	7653	8.35	3	25.05
151	8673	6.89	2	13.78

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 (Σ) 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. You can supply an argument to the Count function, but it rarely makes a difference—generally you just use Count(*). 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 Sally 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`

FIGURE 4.18

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 (Σ) to display the Total line on the QBE grid.

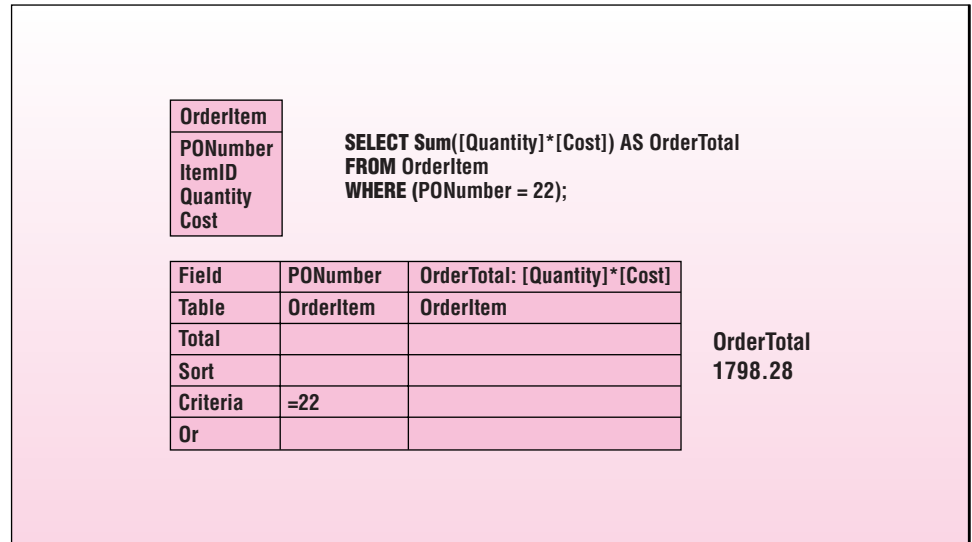
```
SELECT Avg(SalePrice) AS AvgOfSalePrice
FROM SaleAnimal;
```

SaleAnimal		Sum Avg Min Max Count StDev or StdDev Var
SaleID		
AnimalID		
SalePrice		

Field	SalePrice
Table	SaleAnimal
Total	Avg
Sort	
Criteria	
Or	

FIGURE 4.19

Computations. Row-by-row computations (Quantity * Cost) can be performed within an aggregation function (Sum), but only the final total will be displayed in the result.



Sum(Quantity) FROM OrderItem. The 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 4.19 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 4.17) or the total value (Figure 4.19). 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 as described in Chapter 6.

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 shown in Figure 4.7. 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 4.20 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

FIGURE 4.20

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.

Task	Access	SQL Server	Oracle
Strings			
Concatenation	FName & " " & LName	FName + ' ' + LName	FName ' ' LName
Length	Len(LName)	Length(LName)	LENGTH(LName)
Uppercase	UCase(LName)	Upper(LName)	UPPER(LName)
Lowercase	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 DateDiff	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, LinReqSlope, Correlation	AVG, STDDEV, VARIANCE, REGR, CORR

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.

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, whereas 5/1/2004 is the first day in May in the United States, 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.

Subtotals and GROUP BY

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 = "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 4.21, 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.

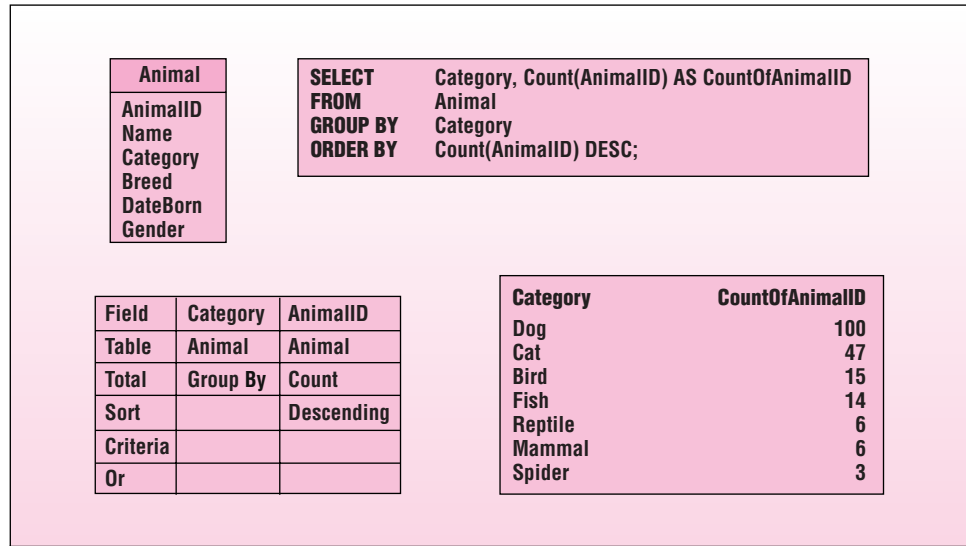
By default, the output will generally be sorted by the group items. However, for business questions, it is common to sort (ORDER BY) based on the computation. The Pet Store example is sorted by the Count, listing the animals with the highest count first.

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. So 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

FIGURE 4.21**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.



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. Note that Oracle does not support the TOP condition.

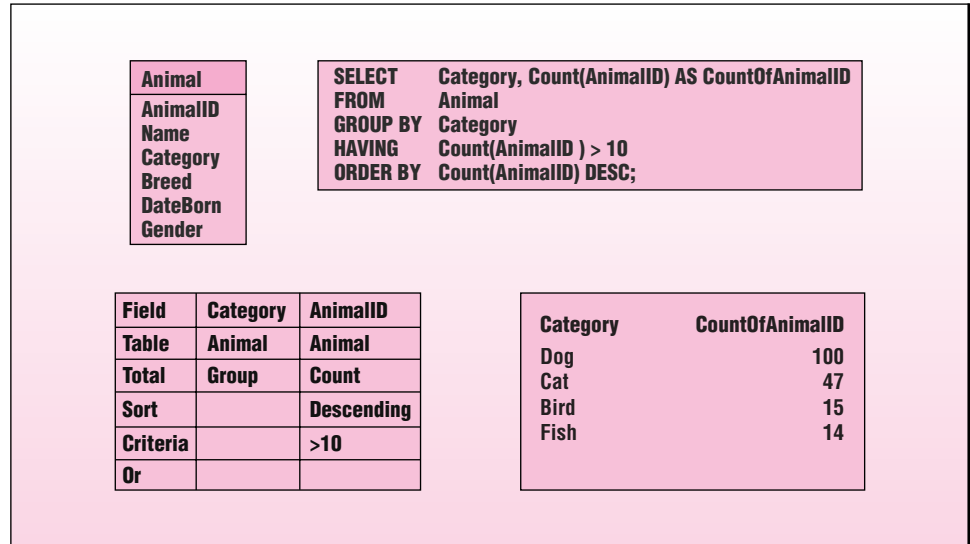
Conditions on Totals (HAVING)

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 4.22, 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.

FIGURE 4.22

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.



WHERE versus HAVING

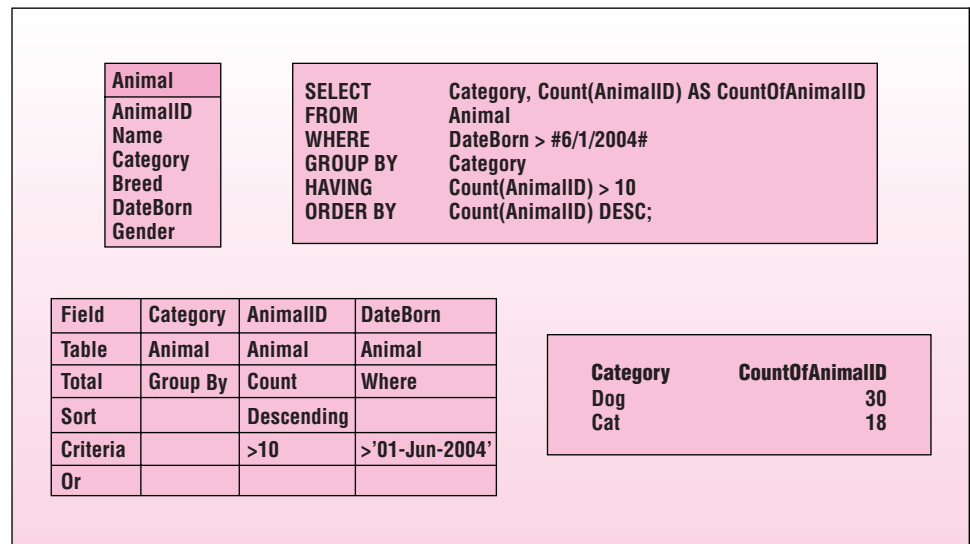
When you first learn QBE and SQL, WHERE and HAVING look very similar, and choosing the proper clause can be confusing. Yet it is crucial that you understand the difference. If you make a mistake, the DBMS will give you an answer, but it will not be the answer to the question you want.

The key is that the WHERE statement applies to every single 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 4.23 that counts the animals born after June 1, 2004, in each Category, but lists only the Category if there are more

FIGURE 4.23

WHERE versus HAVING. Count the animals born after June 1, 2001, 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.



than 10 of these animals. The structure of the query is similar to the example in Figure 4.22. The difference in the SQL statement is the addition of the WHERE clause (`DateBorn >#6/1/2004#`). 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 4.23 (30) with the count in Figure 4.22 (100). Only 30 dogs were born after June 1, 2004. 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 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.

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.

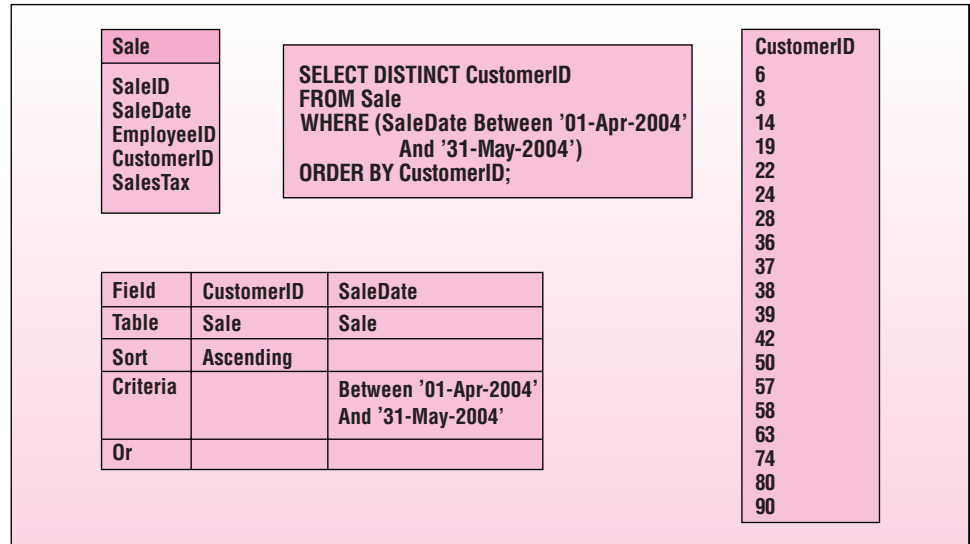
Multiple Tables

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

FIGURE 4.24

List the CustomerID of everyone who bought something between April 1, 2004, and May 31, 2004. Most people would prefer to see the name and address of the customer—those attributes are in the Customer table.



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.

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 4.24: List the CustomerID of everyone who bought something between 4/1/2004 and 5/31/2004. Because some customers might have made purchases on several days, the DISTINCT clause can be used to delete the duplicate listings.

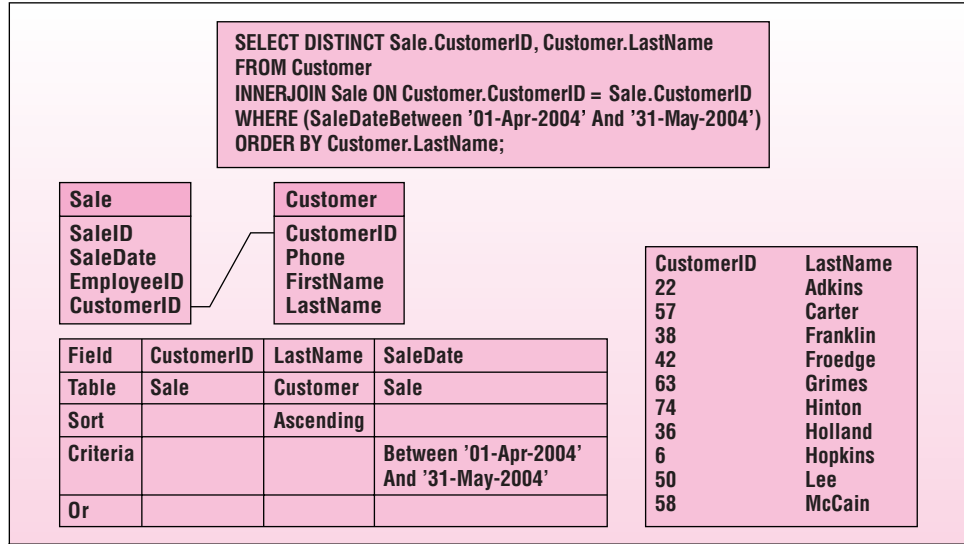
Most managers would prefer to see the customer name instead of CustomerID. However, the name is stored in the Customer table because it would be a waste of space to copy all of the attributes to every table that referred to the customer. If you had these tables only as printed reports, you would have to take the CustomerID from the sales report 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 4.25, 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.

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

FIGURE 4.25

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, 2004, and May 31, 2004.



JOIN condition is part of the WHERE clause. Most vendors are converting 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 4.26. 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 write 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

FIGURE 4.26

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.

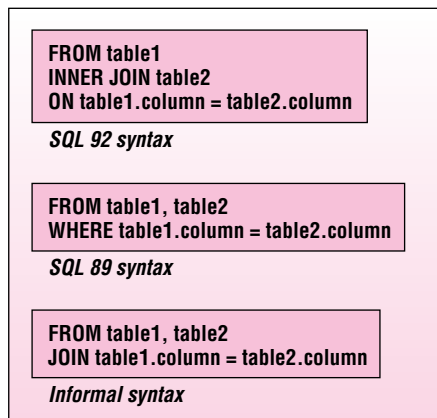


FIGURE 4.28

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.

SQL 92 syntax to join three tables:

```
FROM Table1
INNER JOIN (Table2 INNER JOIN Table3 ON Table2.ColA = Table3.ColA)
ON Table1.ColB = Table2.ColB
```

Easier notation, but not correct syntax:

```
FROM Table1, Table2, Table3
JOIN Table1.ColB = Table2.ColB
Table2.ColA = Table3.ColA
```

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. If the database is built in Access, be sure that you pre-define relationships when you create the tables. 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 ERD 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 4.28 shows the syntax needed to join three tables. To read it or to create a similar statement, start with the innermost JOIN (in the parentheses). Then add a table with the corresponding ON condition. If you need additional tables, continue adding parentheses and ON statements, working out from the center. Just be sure that the new table can be joined to one of the tables inside the parentheses. Figure 4.28 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 a midterm 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.

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. Interestingly, many 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. 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. In algebra, a Cross JOIN is known as a Cartesian product of two sets. If the tables have m and n rows each, the resulting query will have $m * n$ rows!

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).

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 4.29 and leave all four JOINS, you will not get the answer you want. The four JOINS will return AnimalOrders only where the Employee placing the order has the same CityID as the Supplier! If you only need the City for the Supplier, 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).

FIGURE 4.29

A query with these four tables with four JOINS would return only rows where the Employee had the same CityID as the Supplier. If you need only the supplier 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.

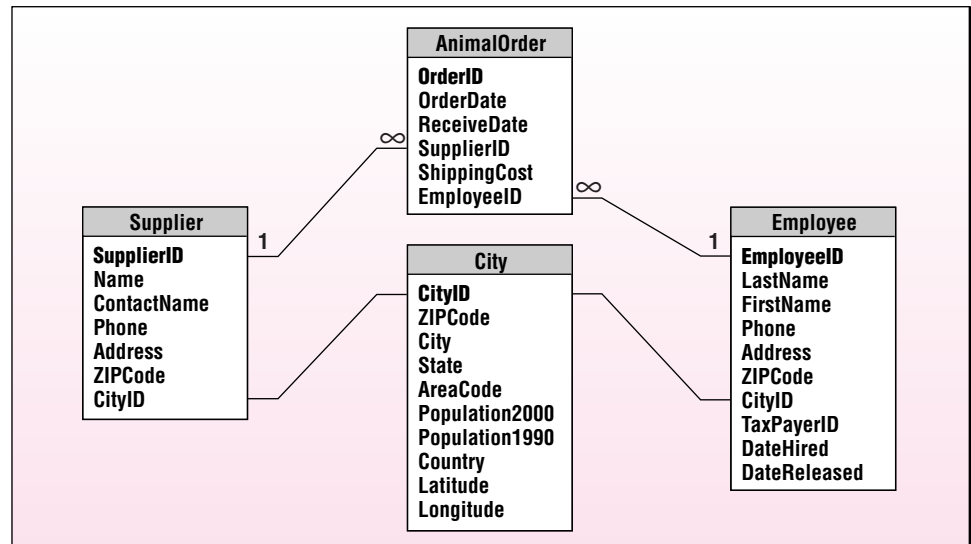


FIGURE 4.30

Table alias. The City table is used twice. The second time, it is given the alias City2 and treated as a separate table. Hence, different cities can be retrieved for Supplier and for Employee.

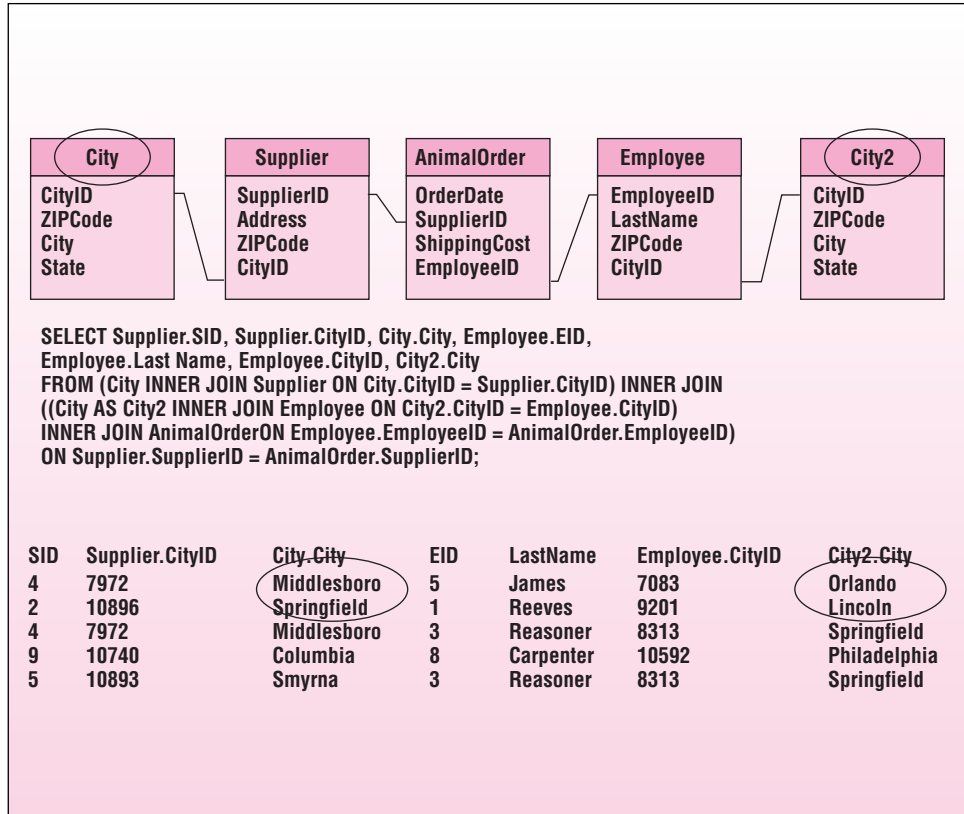


Table Alias

Consider the preceding example in more detail. What if you really want to display the City for the Supplier 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., City2). You give a table a new name (alias) within the FROM clause: FROM City AS City2. As shown in Figure 4.30, the City table is joined to the Supplier. The City2 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.

Create View

FIGURE 4.31

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.

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 4.31 shows the basic SQL

```

CREATE VIEW Kittens AS
SELECT *
FROM Animal
WHERE (Category = 'Cat' AND (TodayDateBorn < 180));

```

FIGURE 4.32

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

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

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 4.31, you would create a view (Kittens) that displays data for Cats born within the last 180 days. As shown in Figure 4.32, 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 4.33 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 4.34 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.

FIGURE 4.33

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.

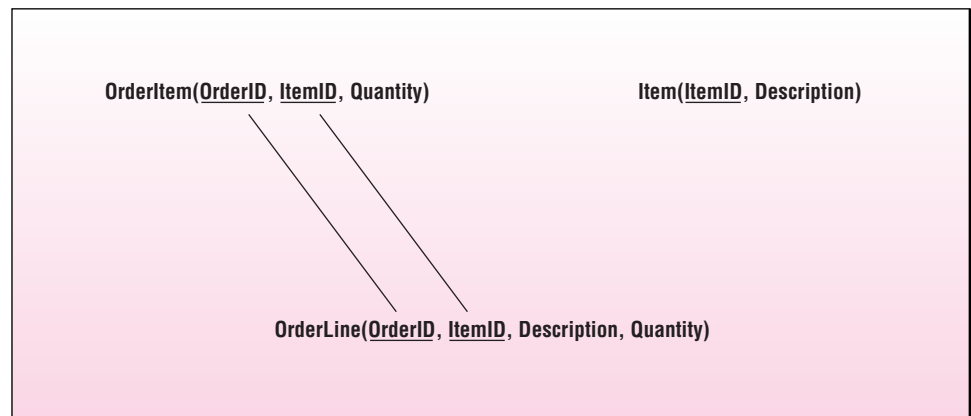
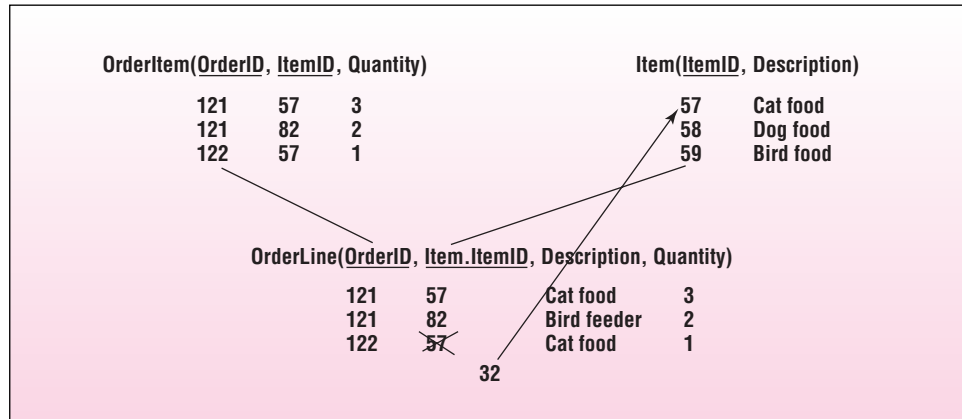


FIGURE 4.34

Nonupdateable 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.



Views have many uses in a database. They are particularly useful in helping business managers work with the database. A database administrator (DBA) or IS 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.

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.

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 Words

aggregation, ●●●	DESC, ●●●	query by example (QBE), ●●●
alias, ●●●	DISTINCT, ●●●	row-by-row calculations, ●●●
BETWEEN, ●●●	FROM, ●●●	SELECT, ●●●
Boolean algebra, ●●●	GROUP BY, ●●●	SQL, ●●●
Cross JOIN, ●●●	HAVING, ●●●	TOP, ●●●
data definition language (DDL), ●●●	JOIN, ●●●	view, ●●●
data manipulation language (DML), ●●●	LIKE, ●●●	WHERE, ●●●
DeMorgan's law, ●●●	NOT, ●●●	
	NULL, ●●●	
	ORDER BY, ●●●	

Review Questions

1. What are the four questions used to create a query?
2. What is the basic structure of the SQL SELECT command?
3. What is the purpose of the DISTINCT operator?
4. Why is it important to use parentheses in complex (Boolean) WHERE clauses?
5. What is DeMorgan's law, and how does it simplify conditions?
6. What is the difference between the ORDER BY and GROUP BY commands?
7. How do the basic SQL arithmetic operators (+, −, etc.) differ from the aggregation (SUM, etc.) commands?
8. What basic aggregation functions are available in the SELECT command?
9. What is the difference between Count and Sum? Give an example of how each would be used.
10. What is the difference between the WHERE and HAVING clauses? Give an example of how each would be used.
11. What is the SQL syntax for joining two tables?

Exercises

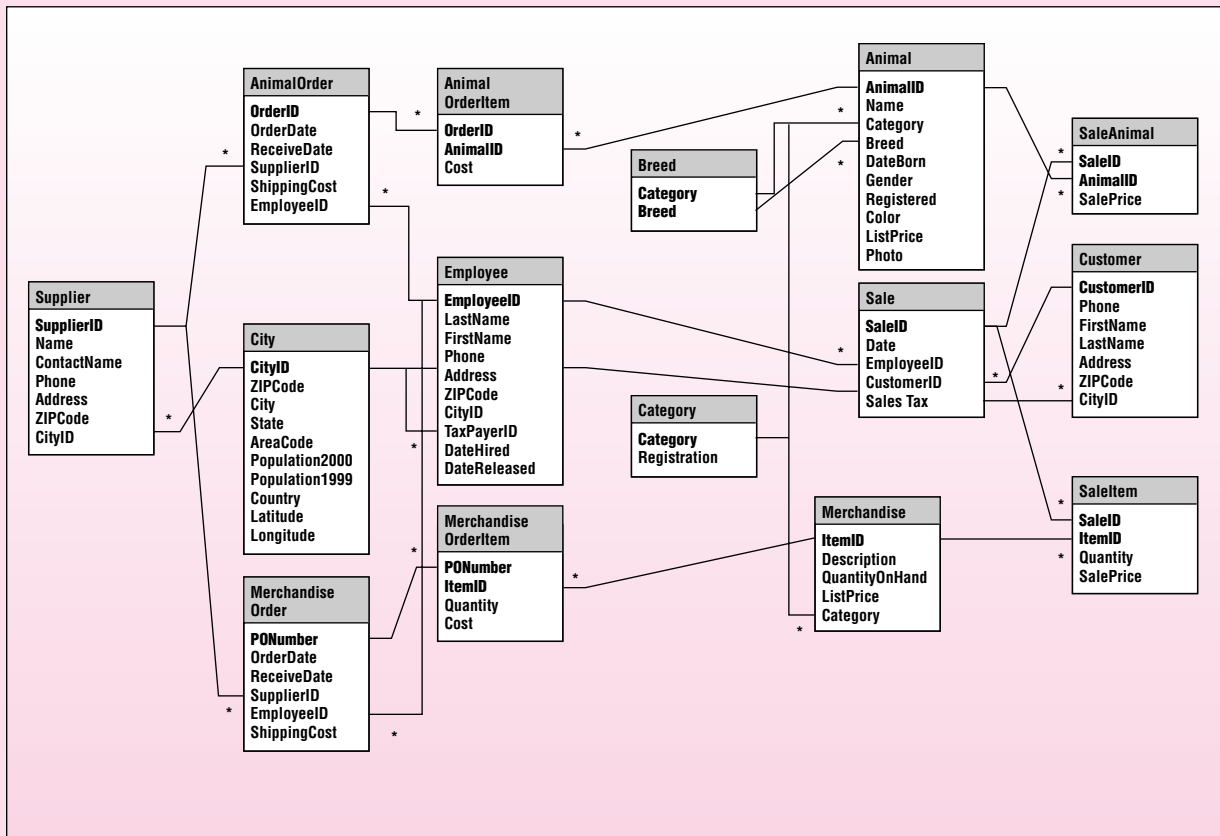


Sally's Pet Store

Write the SQL statements that will answer questions 1 through 25 based on the tables in the Pet Store database.

1. List the cats born in May.
2. List the customers who shopped at the store the first seven days in June.
3. What was the most expensive item sold in July?
4. Which suppliers sent orders that took more than 10 days to arrive?
5. List the items that have fewer than 10 units in stock.
6. List the cats that are brown and cost less than \$300, or any brown female animal priced less than \$150.
7. What is the total value of animals sold in December?

8. How many cats were sold in October?
9. Which employees ordered merchandise from suppliers in Tennessee in April?
10. List the suppliers in Nebraska who sold birds to Sally's Pet Store.
11. Which employee placed the most expensive order in August?
12. Which state holds most of Sally's customers?
13. To what state did the Pet Store sell the most merchandise (by value) in July?
14. List the employees who report to Reasoner.
15. List the customers who have purchased cats from Gibson.
16. Did the store sell more cats or dogs in the first quarter?
17. Did the store sell more female or male animals in the fourth quarter?
18. List the registered dogs sold in January with white in their color.
19. Which supplier sold the Pet Store the most dog merchandise in the first quarter?
20. Which employees sold the most dog merchandise in July and August?
21. Which cats were preordered, meaning they were sold before they were ordered?
22. Which animals sold for less than their cost?
23. Which cats sold for at least 50 percent more than their cost?
24. For each merchandise supplier, what is the average shipping cost?
25. During the third quarter, which items have been ordered the most times in more than 5-unit quantities?

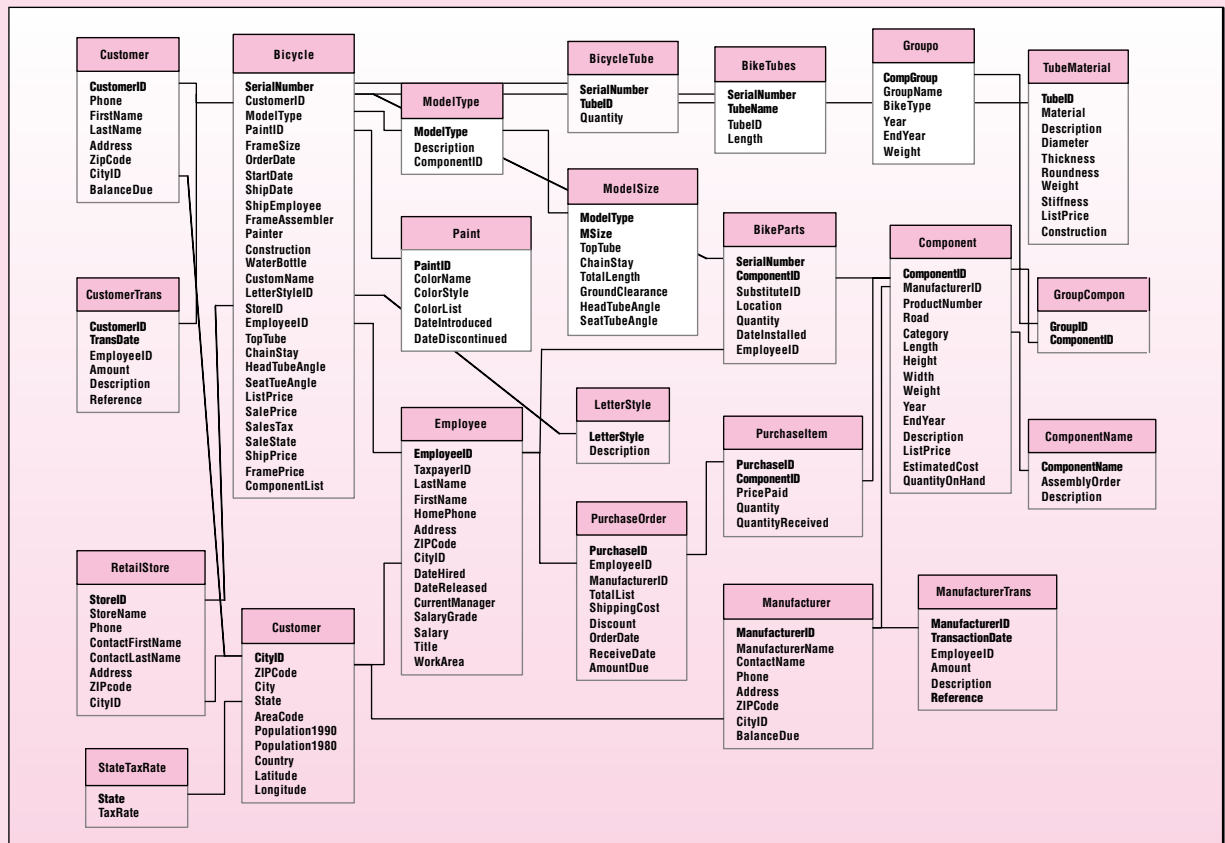




Rolling Thunder Bicycles

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

26. List the customers from California who bought red mountain bikes in September 2003.
27. List the employees who sold racing bikes shipped to Wisconsin without the help of a retail store in 2001.
28. List all of the (distinct) rear derailleurs installed on road bikes sold in Florida in 2002.
29. Who bought the largest (frame size) full-suspension mountain bike sold in Georgia in 2004?
30. Which manufacturer gave Rolling Thunder the largest discount on an order in 2003?
31. Which customer received the greatest percentage discount on a racing bike purchased since January 1, 2000?
32. Which employee has installed the most headsets in mountain bikes?
33. Which non-American company has received the most orders?
34. What is the most expensive road bike component Rolling Thunder stocks that has a quantity on hand greater than 200 units?
35. Which inventory item represents the most money sitting on the shelf based on estimated cost?



36. What is the greatest number of components installed in one day by one employee?
37. What was the most popular letter style on racing bikes in 2003?
38. Which customer spent the most money with the company, and how many bicycles did that person buy in 2002?
39. Have the sales of mountain bikes (full suspension or hard tail) increased or decreased from 2000 to 2004 (by count not by value)?
40. Which component did the company spend the most money on in 2003?
41. Which employee painted the most red racing bikes in May 2003?
42. Which California bike shop helped sell the most bikes (by value) in 2003?
43. What is the total weight of the components on bicycle 11356?
44. What is the total list price of all items in the 2002 Campy Record group?
45. In 2003, were more race bikes built from carbon or titanium (based on the down tube)?
46. What is the average price paid for the 2001 Shimano XTR rear derailleurs?
47. What is the average top tube length for a 54-cm (frame size) road bike built in 1999?
48. On average, which has the higher list price: road tires or mountain bike tires?
49. In May 2003, which employees sold road bikes that they also painted?
50. In 2002, was the Old English letter style more popular with some paint jobs?

Website References

Site	Description
http://www.opengroup.org	Standards group including SQL.
http://www.jtc1sc32.org/sc32/jtc1sc32.nsf/Attachments	Standards documents, start at 742.
http://thebestweb.com/db	Consulting group with SQL hints and lots of links to other sites.
http://www.sqlmag.com	Magazine with SQL emphasis.
http://www.sqlteam.com	SQL hints and comments.
http://www.vb-bookmark.com/SqlTutorial.html	General SQL reference links.
http://msdn.microsoft.com	Microsoft SQL Server notes.

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*. San Mateo: Morgan Kaufmann Publishers, 2002. [An in-depth presentation of SQL 1999, by those who played a leading role in developing the standard.]

SQL Syntax

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)
)
```

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

```
DROP INDEX index ON table
DROP TABLE
DROP TRIGGER
DROP VIEW
```

Insert

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

Grant

```
GRANT privilege    privileges
ON object          ALL, ALTER, DELETE, INDEX,
TO user|PUBLIC    INSERT, SELECT, UPDATE
```

Revoke

```
REVOKE privilege    privileges
ON object          ALL, ALTER, DELETE, INDEX,
FROM user|PUBLIC    INSERT, SELECT, UPDATE
```

Rollback

```
SAVEPOINT savepoint
ROLLBACK WORK
    TO 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
```