

Advanced Queries and Subqueries

Chapter Outline

- Introduction, 252
- Two-Minute Chapter, 253
- Sally's Pet Store, 254
- Outer Joins (LEFT JOIN), 255
- Subqueries: IN and NOT IN, 258
- Subqueries, 261
 - Calculations or Simple Lookup*, 262
 - Calculations for Percentages*, 262
 - Subqueries and Sets of Data*, 264
 - Subquery with ANY, ALL, and EXISTS*, 266
- Correlated Subqueries, 268
- More Features and Tricks with SQL SELECT, 270
 - UNION, INTERSECT, EXCEPT*, 270
 - Multiple JOIN Columns*, 272
 - Reflexive Join*, 273
 - CASE Function*, 275
 - Inequality Joins*, 276
 - Exists and Crosstabs*, 277
 - SQL SELECT Summary*, 280
- SQL Data Definition Commands, 280
- SQL Data Manipulation Commands, 283
 - INSERT and DELETE*, 283
 - UPDATE*, 284
- Quality: Testing Queries, 285
- Summary, 287
- Key Terms, 288
- Review Questions, 289
- Exercises, 290
- Web Site References, 295
- Additional Reading, 295
- Appendix: Programming, 296
- Variable Scope, 297
- Computations, 298
- Standard Internal Functions, 300
- Input and Output, 300
- Conditions, 301
- Loops, 303
- Subroutines, 304
- Summary, 305

What You Will Learn in This Chapter

- How can SQL be used to answer more complex questions?
- Why are some business questions more difficult than others?
- What common uses for subqueries?
- How do you find something that did not happen?
- How do you include rows from tables in a join even if the rows do not match?
- What are correlated subqueries?
- What tricky problems arise and how do you handle them in SQL?
- What are the SQL data definition commands?
- What SQL commands alter the data stored in tables?
- How do you know if your query is correct?

A Developer's View

Ariel: Hi Miranda. You look happy.

Miranda: I am. This query system is great. I can see how it will help the managers. Once I get the application done, they can get answers to any questions they have. They won't have to call me for answers every day. Plus, I can really see how the query system relates to data normalization. With normalization I split the tables so the database could store them properly. Now the query system helps me rejoin them to answer my questions.

Ariel: Does that mean you're finally ready to create the application?

Miranda: Close, but I'm not quite ready. Yesterday my uncle asked me a question that I don't know how to

answer.

Ariel: Really, I thought you could do anything with SQL. What was the question?

Miranda: Something about customers who did not order anything last month. I tried several times to get it to work, but the answers I get just aren't right.

Ariel: It doesn't sound like a hard question.

Miranda: I know. I can get a list of customers and orders that were placed any time except last month. But every time I join the Customer table to the Order table, all I get are the customers who did place orders. I don't know how to find something that's not there.

Getting Started

SQL has several powerful capabilities, including subqueries (the ability to nest a query inside another one), and outer joins (returning all rows from one table in a join instead of ignoring unmatched data). You need to think of data and questions in terms of sets. To answer complex questions, break it into pieces and create a query to return the data set for each piece. Then combine the pieces using joins, subqueries, or set operations.

Introduction

How can SQL be used to answer more complex questions? Now that you understand the basics of the SQL SELECT statement as described in Chapter 4, it is time to study more complex questions. The basic SELECT statement you have learned is useful for returning filtered rows and columns of data and for computing subtotals. However, some business questions are more complex than those examples. For instance, how would you find items that were not sold? The database only stores things that did happen and note that when tables are joined, only the rows with matching data are returned. How can you get to the data that is not matched—that is, data in one table (Merchandise) but not in the other (SaleItem)? Also, what if you need to combine data from multiple queries? A classic example is percentages. To compute percentages within a group, you must first compute the group totals and then divide by the overall total. One of the most powerful features of the SQL SELECT command is known as a **subquery** or **nested query**.

This feature enables you to ask complex questions that entail retrieving different types of data or data from different sources.

SQL is also more than a query language. The language can be used to create tables, as well as insert, delete, and update data. It can be used to create the entire database (data definition language). SQL has powerful commands to alter the data (data manipulation language). SQL also has a couple of commands to set security conditions (data control language).

Two key points will help you learn how to use subqueries: (1) SQL was designed to work with sets of data—avoid thinking in terms of individual rows, and (2) you can split nested queries into their separate parts and deal with the parts individually. Sometimes it is helpful to write a query to answer part of a question and save it. This saved query or view can then be used in part of a second query.

The features of SQL covered in Chapter 4 are already quite powerful. Why do you need more features? Consider this common business question for Sally's Pet Store: Which merchandise items have not been sold? Think about how you might answer that question using the SQL you know to this point.

The first step might be to choose the tables: `Merchandise` and `SaleItem` appear to be likely choices. Second, select the columns as output: `ItemID` and `Description`. Third, specify a condition. Fourth, join the tables. These last two steps cause the most problems in this example. How do you specify that an item has not been sold? The big catch is that you have to be careful when examining data in the `SaleItem` table. Because the item has not been sold, the `SaleItem` table will not contain any entries for it. The `SaleItem` table records things that have happened. You are looking for something that has not happened.

Actually, the fourth step (joining the tables) causes even more problems. Say you wrote a query like this: `SELECT ItemID, Name FROM Merchandise INNER JOIN SaleItem ON (Merchandise.ItemID = SaleItem.ItemID)`. As soon as you write that `JOIN` condition, you eliminate all the items you want to see. The `JOIN` clause restricts the output—just like a `WHERE` clause would. In this example, you told the DBMS to return only those items that are listed in both the `Merchandise` and `SaleItem` tables. But only items that have been sold are listed in the `SaleItem` table, so this query can never tell you anything about items that have not been sold. The following sections describe two solutions to this problem: either fix the `JOIN` statement so that it is not as restrictive or use a subquery.

Two-Minute Chapter

Some business questions are harder to answer than they first appear. Chapter 4 showed how to create basic SQL queries—selecting columns and rows, making basic calculations, and computing aggregations such as averages and sums. Computing subtotals using the `GROUP BY` statement is an important part of many queries. This foundation is used again in this chapter, but with a few twists. From a SQL perspective, four primary elements are added in this chapter: (1) subqueries, where you can embed a second `SELECT` statement into another one to look up different data, (2) `LEFT JOINS`, which keep rows of data from a table even if no values are matched on the other side of the join, (3) inequality `JOINS` where values can be compared using conditions beyond a simple equals sign, and (4) data manipulation language commands that enable you to `INSERT`, `UPDATE`, and `DELETE` data, not just retrieve it.

Just looking at the SQL capabilities, it is not always clear why these new features are needed. But some business questions can be tricky. How do you find

- Which items have not been sold?
- Which items were not sold in July 2013?
- Which cat merchandise sold for more than the average sale price of cat merchandise?
- Compute the merchandise sales by category in terms of percentage of total sales.
- List all of the customers who bought something in March and who bought something in May. (Two tests on the same data!)
- List dog merchandise with a list price greater than the sale price of the cheapest cat product.
- Has one salesperson made all of the sales on a particular day?
- Use Not Exists to list customers who have not bought anything.
- Which merchandise has a list price greater than the average sale price of merchandise within that category?
- List all the managers and their direct reports.
- Convert age ranges into categories.
- Classify payments by number of days late.
- Which employees sold merchandise from every category?
- List customers who adopted dogs and also bought cat products.

Figure 5.1

Harder questions. Even though there are few constraints on the problems, these questions are more complex. To answer many of them, we need to use subqueries or outer joins.

something that did not happen? A database table only stores things that did happen, so you need a way to find items in one list that are not in a second list. For instance, find Employees who did not make a Sale in a specific month. This question can be answered with a NOT IN subquery, or using a LEFT JOIN to connect the tables. Some business questions require separate sets of data—such as listing customers who bought items in March and June. Those lists have to be defined with separate queries and then combined—either through subqueries or as two saved views. Similarly, subqueries are useful when you need to compute percentage values—such as the percentage of total monthly sales attributed to each employee. At almost any point in SQL where you need a new value (divide by total), you can add parentheses and write a new SELECT statement to retrieve that value.

SQL contains a full set of commands to CREATE and DROP tables, indexes, and other items. It also has commands to UPDATE, INSERT, or DELETE rows of data. When working with these commands it is best to think in terms of sets of data. Using the power of the WHERE command (including subqueries) you can modify specific collections of data with one command.

When working with complex SQL commands, it is critical to build queries in pieces and test each piece along the way. The scariest part of SQL is that in most cases, a SELECT statement will return values—but you need to be sure that the query was interpreted the way you intended and the values accurately answer the business question.

Sally's Pet Store

Why are some business questions more difficult than others? Figure 5.1 shows some more business questions that Sally needs to answer to manage her business. Again, think about how you might answer these questions using the basic SQL of

Chapter 4. At first glance they do not seem too difficult. However, even the easiest question—to identify cats that sold for more than the average price—is harder than it first appears.

The common feature of these questions is that they need to be answered in multiple steps. All of these questions require an additional tool: the subquery. Actually, you can also answer multi-step questions by writing and saving the first part as a view and then using the view in another query. However, you should generally try to use subqueries so the DBMS query optimizer can use the complete query to find the most efficient solution.

Outer Joins (LEFT JOIN)

How do you find something that did not happen? One question that commonly arises in business settings is illustrated in Figure 5.2 with the question: Which merchandise has not been sold? This question is deceptive. At first glance it looks like you could just join the Merchandise table to the SaleItem table. But then what? The standard INNER JOIN statement will display only that merchandise that appears in both the Merchandise and SaleItem tables. As soon as you enter the JOIN statement, you automatically restrict your list to only that merchandise that has been sold.

Figure 5.2

INNER JOIN is a filter. Rows that are not in both tables are ignored. Because SaleItem includes only merchandise that has been sold, INNER JOIN discards the very data that you want to see.

Which items have not been sold?

```
Try:
SELECT *
FROM Merchandise
INNER JOIN SaleItem
  ON Merchandise.ItemID = SaleItem.ItemID
```

But INNER JOIN is a filter that returns only rows that exist in both tables.

SaleItem		Merchandise	
SaleID	ItemID	ItemID	Description
4	1	1	Dog Kennel-Small
4	36	2	Dog Kennel-Medium
6	20	3	Dog Kennel-Large
6	21	4	Dog Kennel-Extra Large
7	5	5	Cat Bed-Small
7	19	6	Cat Bed-Medium
7	40	7	Dog Toy
8	11	8	Cat Toy
8	16	9	Dog Food-Dry-10 pound
8	36	10	Dog Food-Dry-25 pound
10	23	11	Dog Food-Dry-50 pound
10	25	12	Cat Food-Dry-5 pound
10	26	13	Cat Food-Dry-10 pound
10	27	14	Cat Food-Dry-25 pound
		15	Dog Food-Can-Regular

Which merchandise has not been sold?

```
SELECT Merchandise.ItemID, Merchandise.Description, SaleItem.
SaleID
FROM Merchandise
LEFT JOIN SaleItem ON Merchandise.ItemID = SaleItem.ItemID
WHERE SaleItem.SaleID Is Null;
```

ItemID	Description	SaleID
12	Cat Food-Dry-5 pound	
13	Cat Food-Dry-10 pound	

Figure 5.3

LEFT JOIN. The left outer join includes all rows from the Merchandise (left) table and any matching rows from the SaleItem table. If an item has not been sold, there will be no entry in the SaleItem table, so the corresponding entries will be NULL.

One way to solve this problem is to change the behavior of the JOIN command. SQL provides the **OUTER JOIN** specifically to include the data that would otherwise be ignored with the INNER JOIN. In particular, the OUTER JOIN describes what should happen when values in one table do not exist in the second table.

In joining two tables, you have to consider two basic situations: (1) A value might exist in the left table with no matching value in the right table, or (2) a value might exist in the right table with no matching value in the left table. Of course, it really does not matter which table is on the left or right. However, you have to be careful about not mixing them up after you list the tables.

The query in Figure 5.3 illustrates a typical **LEFT JOIN**. With a LEFT JOIN, all rows in the table on the left will be displayed in the results, regardless of what rows exist in the other table. If there is no matching value from the table on the

Figure 5.4

Partial results from the left outer join. Note the missing (Null) values for items that have not been sold. To list just a single SaleID, use GROUP BY and use the FIRST option to pick a single SaleID.

M.ItemID	Description	SA.ItemID	SaleID
1	Dog Kennel-Small	1	4
2	Dog Kennel-Medium	2	54
3	Dog Kennel-Large	3	17
4	Dog Kennel-Extra Large	4	18
5	Cat Bed-Small	5	7
6	Cat Bed-Medium	6	46
7	Dog Toy	7	64
8	Cat Toy	8	13
9	Dog Food-Dry-10 pound	9	48
10	Dog Feed-Dry-25 pound	10	60
11	Dog Food-Dry-50 pound	11	8
12	Cat Food-Dry-5 pound		
13	Cat Food-Dry-10 pound		

right, NULL values will be inserted into the output. Note how the LEFT JOIN resolves the problem of identifying items that have not been sold. Because the query will now list all Merchandise items, the rows where the SaleID is Null represent items that are not in the SaleItem table and have not been sold.

Figure 5.4 shows the sample data without the “Is Null” condition. The data has also been reduced using a GROUP BY and First statement to focus on the individual Merchandise items. Notice the two values with the missing or null values.

The **RIGHT JOIN** behaves similarly to the LEFT JOIN. The only difference is the order of the tables. If you want to use all the rows from the table on the right side, use a RIGHT JOIN. Why not just have a LEFT JOIN and simply rearrange the tables? Most of the time, that is exactly what you will do. However, if you have a query that joins several tables, it is sometimes easier to use a RIGHT JOIN instead of trying to rearrange the tables. And with visual tools such as the Microsoft Access query editor, the position of the displayed table does not have to match the SQL statement. In every case, the Left/Right applies to the way the SQL statement is written.

Another join is the full OUTER JOIN (**FULL JOIN**) that combines every row from the left table and every row from the right table. Where the rows do not match (from the ON condition), the join inserts NULL values into the appropriate columns. Many systems do not support the FULL or OUTER JOIN on both tables at the same time. If you encounter a question that requires both a left and right join, you can use a LEFT JOIN and a RIGHT JOIN against a full list of the ID values--which can be obtained using a saved UNION query.

Warning: Be careful with OUTER JOINS—particularly full joins. With two large tables that do not have much data in common, you end up with a very large result that is not very useful. Also be careful when using outer joins on more than two tables in one query. You get different results depending on the order in which you join the tables. Many times you will find it necessary to create a view with only two tables to create an outer join. You can then use that view in other queries to add more tables.

Finally, note that these examples rely on the SQL 92 syntax, which is fairly easy to read and understand. Unfortunately, you will most likely encounter some queries that use older, proprietary syntax for outer joins. Figure 5.5 shows the query using the syntax for SQL Server and Oracle. SQL Server uses *= to indi-

Figure 5.5

Older syntax for LEFT JOIN. Note the asterisk in SQL Server to indicate the LEFT side table. Note the plus-sign in Oracle and note that it is on opposite side from what you would expect.

```
SELECT *                (SQL Server)
FROM Merchandise, SaleItem
WHERE Merchandise.ItemID *= SaleItemID.ItemID
And SaleItem.SaleID Is Null

SELECT *                (Oracle)
FROM Merchandise, SaleItem
WHERE Merchandise.ItemID = SaleItemID.ItemID (+)
And SaleItem.SaleID Is Null
```

Find a Customer with first name of Tim, David, or Dale

```
SELECT *
FROM Customer
WHERE FirstName=N'Tim' Or FirstName=N'David' Or FirstName=N'Dale'

SELECT *
FROM Customer
WHRE FirstName IN (N'Tim', N'David', N'Dale')
```

Figure 5.6

IN function. The IN function compares a column to a set of values. The WHERE condition is true if the column/row matches any one of the entries.

cate a left join, where the asterisk can be interpreted as the “all rows” side of the join. Oracle uses a plus sign, and it confusingly puts it on the opposite side of the equals sign. Be careful when reading older queries to look for the asterisk or plus sign. The query results are quite different if you ignore these left join indicators. Fortunately, all of the major systems now accept the newer syntax, so you should convert older queries to the new syntax to improve readability.

Subqueries: IN and NOT IN

How is a subquery used for IN and NOT IN conditions? There is another way to answer the question of which items have not been sold. This new approach has considerable power and can be used for many types of questions. The main tool is the subquery, but for the problem of finding things that did not happen it is tied to a special WHERE condition known as the **IN** statement. So this section begins with a brief explanation of the IN function. The IN function defines a set of values. You can think of it as a shortcut way of combining several entries with an “Or” condition. For example, say you want to search for a Customer but you are not certain about his first name. You think it might be “Tim” or “David” or “Dale.” As shown in Figure 5.6, you could build a query using “Or” conditions: WHERE FirstName=”Tim” or FirstName=”David” or FirstName=”Dale”. However, the figure also shows an easier way to write the query using the IN function. Simply list all possible values separated by commas and enclose them in parentheses. The IN function essentially defines a set of possible matches. It can

Figure 5.7

Subquery to find data for an IN set of values. This subquery essentially functions as a JOIN condition. Matching ItemID in the Merchandise table to the ItemID in the SaleItem table.

List Merchandise based on ItemID that has been sold.

```
SELECT * FROM Merchandise
WHERE ItemID IN (1,2,3,4,5,6,7,8,9,10,11,14,15);

SELECT *
FROM Merchandise
WHERE ItemID IN
(SELECT ItemID FROM SaleItem);
```


be used in many situations, just be sure to match the data types with the search column. In this case, the set contains possible FirstName values.

Now consider a more relevant set of data shown in Figure 5.7, using a different question: List Merchandise where ItemID is one of 1,2,3,4,5,6,7,8,9,10,11,14,15. The list of items is a bit long, but the process is identical to that used for the names: *SELECT * FROM Merchandise WHERE ItemID IN (1,2,3,4,5,6,7,8,9,10,11,14,15)*. Using the raw numbers, this list is not particularly interesting. However, rewrite the query as shown in the second half of the figure. Instead of a fixed list of numbers, use a new query (*SELECT ItemID FROM SaleItem*) to retrieve a list of ItemID values. This subquery is embedded directly into the main query; however, note that it is surrounded by parentheses. Also, the subquery text is indented to make it easier to read. The parentheses are required, the indentation is not. This subquery performs the same role as an INNER JOIN statement. Rows from the Merchandise table will be returned only if the ItemID exists in the SaleItem table. Notice that with this formulation, only data from the top-most query (Merchandise) can be displayed. The subquery acts as a filter, but data from the subquery table cannot be displayed in the results.

Finally, as shown in Figure 5.8, it is possible to answer the original question: *List the merchandise that has not been sold*. Note that the previous version listed merchandise that was sold. That is, list the Merchandise items that are in the

Figure 5.8

NOT IN. The top-level query retrieves items from the complete list (Merchandise) and subtracts items that are in the second list (SaleItem). Leaving the results of items in the first list that are not in the second list—or things that did not happen.

List Merchandise that has not been sold.

```
SELECT *
FROM Merchandise
WHERE ItemID NOT IN
(SELECT ItemID FROM SaleItem);
```

Merchandise

ItemID	Description
1	Dog Kennel-Small
2	Dog Kennel-Medium
3	Dog Kennel-Large
4	Dog Kennel-Extra Large
5	Cat Bed-Small
6	Cat Bed-Medium
7	Dog Toy
8	Cat Toy
9	Dog Food-Dry-10 pound
10	Dog Food-Dry-25 pound
11	Dog Food-Dry-50 pound
12	Cat Food-Dry-5 pound
13	Cat Food-Dry-10 pound
14	Cat Food-Dry-25 pound
15	Dog Food-Can-Regular

Which merchandise was not sold in July 2013?

```
SELECT *
FROM Merchandise
WHERE ItemID NOT IN
  (SELECT ItemID
   FROM SaleItem
   INNER JOIN Sale ON Sale.SaleID=SaleItem.SaleID
   WHERE SaleDate BETWEEN
     '01-JUL-2013' AND '31-JUL-2013'
  );
```

Figure 5.9

Subquery with a Date condition. Subqueries can be relatively complex. They can even be nested several levels deep. Often, subqueries can be used to write a single complex query that would need to be broken into pieces if handled differently.

SaleItem table. To answer the main question, start with the main list (Merchandise) and subtract the items that were sold (SaleItem). The process is similar to the way you would answer the question by hand if you had only paper lists. You would begin with the main Merchandise list, go through the SaleItem list and cross off all of the entries that you found. The ones that remain are the Merchandise items that never appeared on the SaleItem list so they were not sold.

When would you use the NOT IN subquery versus the LEFT JOIN? Ultimately, there is no fixed rule—use whichever method you feel is easiest to answer the question correctly. There are often multiple ways to write complex queries. Initially, the most important aspect is that you build the query correctly to answer the question. But, is one method faster to process than the other? Possibly, but ultimately that answer is up to the specific DBMS you are using. The high-end query processors automatically optimize every query, sometimes rewriting it to make it more efficient. On the other hand, if you work with a lower-end DBMS, you might have to rewrite some queries yourself to make them faster—particularly if the query needs to be run multiple times on large datasets.

Consider one more example to point out some other difficulties in creating queries that search for things not in the database. Which merchandise was not sold in July 2013? The change is to add the date condition. First, look at the subquery ap-

Figure 5.10

LEFT JOIN. This query might not run, and if it does, it might not return the correct results. The problem is that the question requires filtering the data rows in the SaleItem table first and then performing the LEFT JOIN.

Which merchandise was not sold in July 2013?

```
SELECT Merchandise.*
FROM Sale
INNER JOIN (Merchandise
  LEFT JOIN SaleItem ON Merchandise.ItemID = SaleItem.ItemID
  ON Sale.SaleID = SaleItem.SaleID
  WHERE SaleDate BETWEEN '01-JUL-2013' AND '31-JUL-2013');
```

Which merchandise was not sold in July 2013?

```
CREATE VIEW JulyItems AS
SELECT Sale.SaleID, ItemID
FROM Sale
INNER JOIN SaleItem ON Sale.SaleID=SaleItem.SaleID
WHERE SaleDate BETWEEN '01-JUL-2013' AND '31-JUL-2013';

SELECT Merchandise.*
FROM Merchandise
LEFT JOIN JulyItems ON Merchandise.ItemID=JulyItems.ItemID
WHERE JulyItems.Sale Is Null;
```

Figure 5.11

Saved View. To ensure the proper sequencing, save the view that filters the list of sale items to July.

proach. Figure 5.9 shows how to answer the question with a subquery. Essentially, the approach is the same as before—with a more complex subquery. Simply add the Sale table to the subquery and add the date condition. The overall structure is the same. Running the query results in 27 rows or items that were not sold in July.

Now consider writing the same query using the LEFT JOIN approach. As shown in Figure 5.10, try building the query directly. Note that it requires three tables: Merchandise, Sale, and SaleItem. Sale and SaleItem are connected with an INNER JOIN and Merchandise with a LEFT JOIN. Because of these links, it is likely that this query will not run. Even if it does return results, they might not be the correct results. The problem is that the question requires that the data be extracted in a specific order—and SQL does not guarantee that processing is handled in a specific sequence. To work correctly, the query must first filter the rows in the Sale+SaleItem tables to just sales that took place in July. This result must then use an outer join with the Merchandise table.

If you want (or need) to use LEFT JOIN to answer the question, you should build the query in two steps. As shown in Figure 5.11, in step 1, create and save a view that retrieves the ItemID for merchandise sold in July. In step 2, LEFT JOIN the Merchandise table to the new view. The result should be the same 27 items found using the subquery. The key to this query is that the view is created to ensure that the rows for sales in July are extracted first and then the LEFT JOIN is applied to the Merchandise table.

Subqueries

What are the common uses for subqueries? The most difficult step in creating a query is determining the overall structure. Chapter 4 shows you how to use the four big questions to determine the structure of simple queries. But you need to recognize when subqueries are needed. If you fail to use a subquery, you are likely to end up with bad results, and waste considerable time in the process. This section presents the most common situations that require the use of subqueries. The main situations are: (1) Calculations or lookup comparisons, (2) matching sets of data, (3) existence checks, and (4) finding items that are not in a list. The last example was covered in the previous section.

Which cat merchandise sold for more than the average sale price of cat merchandise?

```
SELECT Merchandise.ItemID, Merchandise.Description, Merchandise.
Category, SaleItem.SalePrice
FROM Merchandise
INNER JOIN SaleItem ON Merchandise.ItemID = SaleItem.ItemID
WHERE Merchandise.Category=N'Cat' AND SaleItem.SalePrice > 9;
```

```
SELECT Merchandise.ItemID, Merchandise.Description, Merchandise.
Category, SaleItem.SalePrice
FROM Merchandise
INNER JOIN SaleItem ON Merchandise.ItemID = SaleItem.ItemID
WHERE Merchandise.Category=N'Cat' AND SaleItem.SalePrice >
(SELECT Avg(SaleItem.SalePrice) AS AvgOfSalePrice
FROM Merchandise
INNER JOIN SaleItem ON Merchandise.ItemID = SaleItem.ItemID
WHERE Merchandise.Category=N'Cat')
```

Figure 5.12

Subqueries for calculation. If you know the average price is 9, the query is straightforward. If you do not know the average price, you can use a subquery to compute it. The subquery is always written inside a separate set of parentheses. The subquery in parentheses replaces the 9 in the original query).

Calculations or Simple Lookup

Perhaps the easiest way to see the value of a subquery is to consider the relatively simple question: Which cat merchandise sold for more than the average price of cat merchandise? If you already know the average sale price of cat merchandise (say, \$9), the query is easy, as shown in the top half of Figure 5.12.

Chapter 4 showed that it is also straightforward to write a query to compute the average price of cat merchandise. If you do not know anything about subqueries, you could write the average value on a piece of paper and then enter it into the main query in place of the 9. However, with a subquery, you can go one step further: The result (average) from the query can be transferred directly to the original query. Simply replace the value (\$9) with the complete SELECT AVG query as shown in the lower half of Figure 5.12. In fact, anytime you want to insert a value or comparison, you can use a subquery instead. You can even go to several levels, so a subquery can contain another subquery and so on. The DBMS generally evaluates the innermost query first and passes the results back to the higher level.

Calculations for Percentages

Typically, subqueries for calculations arise in WHERE clauses similar to the prior example when you need to make a comparison. You can also add subqueries to the SELECT statement to retrieve a value for a calculation. For instance, you might issue a subquery to retrieve a tax rate that is multiplied times a total.

Another interesting business problem is the need to compute percentages. Figure 5.13 shows a typical question to compute the percentage of merchandise sales by category. The first step is to compute the total sales by category—which is a straightforward question from Chapter 4. That query contains the subtotal calcula-

Compute the merchandise sales by category in terms of percentage of total sales.

```
CREATE VIEW CategorySubtotals AS
SELECT Merchandise.Category, Sum([Quantity]*[SalePrice]) AS [Value]
FROM Merchandise
INNER JOIN SaleItem ON Merchandise.ItemID = SaleItem.ItemID
GROUP BY Merchandise.Category;
```

```
SELECT CategorySubtotals.Category, CategorySubtotals.Value,
[Value] /
(SELECT Sum(Value) FROM CategorySubtotals) AS Percentage
FROM CategorySubtotals;
```

Category	Value	Percentage
Bird	\$631.50	7.45063292035315E-02
Cat	\$1,293.30	0.152587546411603
Dog	\$4,863.49	0.573809638983505
Fish	\$1,597.50	0.188478006179955
Mammal	\$90.00	1.06184792214059E-02

Figure 5.13

To obtain percentages, first compute the group subtotals and save the view. Select the values from the saved view and use a subquery in the SELECT clause to divide by the total. Ultimately, format the new percentage column to make it readable.

tion: $\text{Sum}([\text{Quantity}] * [\text{SalePrice}])$. To compute the percentages, add a new column that uses the same subtotal and divides by the overall total. The trick is that the overall total is computed using a subquery: $\text{SELECT Sum}([\text{Quantity}] * [\text{SalePrice}])$ FROM SaleItem. So the entire calculation becomes:

```
SELECT ... Sum([Quantity]*[SalePrice]) /
(SELECT Sum([Quantity]*[SalePrice] FROM SaleItem)
GROUP BY Category...
```

Of course, most problems are even more complex and trying to jam everything into one query can lead to mistakes. So you might want to first create a saved query that computes totals by category and use a second query to compute percentages, which makes it easier to check the results. Once the subtotals have been computed and saved, the small addition to compute percentages is almost always the same.

You should realize by now that there are other ways to answer the original question. For example, keep the first view that computes the subtotals. Create a second view to compute the overall total. This second view will contain only one row as a result. Now build a third query that joins these two results. Simply do not enter a JOIN condition—let the DBMS build a cross-join so that the overall total is matched to every row of the first query. Figure 5.14 shows the new view and the query that performs the cross join and division. The results should match those with the subquery method.

Two useful practices you should follow when building subqueries are to indent the subquery to make it stand out so humans can read it and to test the subquery before inserting it into the main query. Fortunately, most modern database systems make it easy to create a subquery and then cut and paste the SQL into the main

Compute the merchandise sales by category in terms of percentage of total sales.

```
CREATE VIEW TotalItemSales AS
SELECT Sum(Value) AS MainTotal
FROM CategorySubtotals;

SELECT Category, Value, Value/MainTotal AS Percentage
FROM CategorySubtotals, TotalItemSales;
```

Figure 5.14

Percentages using a cross join. Create a view to compute the total. A third query uses a cross join to connect this single value to every row in the subtotal query and then divide to get the percentage.

query. Similarly, if you have problems getting a complex query to work, cut out the inner subqueries and test them separately. And always remember to enclose the subquery in parentheses.

The main drawback to subqueries is that they are difficult to read and understand. It is easy to make mistakes and it is difficult to read complex queries created by other developers. You should always document your work when creating complex queries. Whenever possible, use the SQL comment characters (--) to add notes to the query to explain its purpose and how it is supposed to work. Sometimes, it is better to store complex subqueries as views and use a final query to retrieve data from the carefully-named views.

The other trick you will quickly learn is that QBE grids are not very useful when designing subqueries. You almost always need to work with plain SQL statements. If you want to save some typing, you can use QBE to write the join statements, but eventually, you need to copy and paste the SQL text.

Subqueries and Sets of Data

A key to understanding SQL is to focus on sets of data. Complex queries generally can be broken down into multiple pieces, where each piece of the question refers to a set of data. Then you have to figure out how to combine those sets to answer the business question. So far you have seen two ways to combine sets of data: (1) By saving each piece and using a JOIN statement, or (2) Using a subquery, typically with an IN function. In effect, these two methods work the same way. Which one you choose depends on which is easiest or fastest to use. Keep in mind that subqueries enable you to put the entire SQL into a single query, which reduces the risk of someone accidentally deleting a supporting saved view—because no one knew what it was for.

To understand the issue of sets of data, think about an apparently simple question: List all of the customers who bought something in March and in May. As shown in Figure 5.15, a beginner might try to answer the question by creating a simple query with the WHERE clause: SaleDate Between 01-Mar And 31-Mar AND SaleDate Between 01-May and 31-May. What is wrong with this approach? Try it. The query will run, but you will not get any matches. Why not? Because the clause is asking the DBMS to return rows where the SaleDate is in March and in May, at the same time! It is not possible for a date to be in two months at the same time.

List all of the customers who bought something in March and who bought something in May.

```
SELECT Customer.CustomerID, Customer.Phone, Customer.
LastName, Sale.SaleDate
FROM Customer
INNER JOIN Sale ON Customer.CustomerID = Sale.CustomerID
WHERE Sale.SaleDate Between '01-MAR-2013' And '31-MAR-2013'
AND Sale.SaleDate Between '01-MAY-2013' And '31-MAY-2013';
```

Figure 5.15

The wrong approach. Why does this query always return no rows? Because it is checking the date on each row to see if it falls in March AND May. No date can be in two months at the same time.

The answer to the question lies in realizing that you need to get two separate lists of people: those who bought something in March and those who bought something in May. Then you combine the lists to identify the people in both sets. You can answer this question with a subquery, or you can create two separate views and join them. The subquery illustrates the set operations.

Figure 5.16 shows the subquery approach. The outermost (top) query retrieves customers who bought something in March, and the subquery retrieves ID numbers for customers who bought something in May. Either month could be tested first, but it is critical to recognize that you need two separate queries to create the two separate WHERE clauses. The IN operator performs the matching so that the final query displays only those customers who fall in both sets of data.

Figure 5.17 shows how to answer the same query with a JOIN statement on saved views. The views are used to retrieve the desired sets, and they highlight that the sets are separate. The final query uses the JOIN command to retrieve only the values that exist in both of the saved views (March and May).

Both approaches (subquery and saved views) provide the same answer and you generally get to choose which approach you want to use. The drawback to saving views is that you end up with a huge collection of views, and no one remembers

Figure 5.16

Combining two separate lists. The question requires you to create two separate lists and then compare the matching values. This query uses the IN statement to find the customers that appear in both lists.

List all of the customers who bought something in March and who bought something in May.

```
SELECT Customer.LastName, Customer.FirstName
FROM Customer INNER JOIN Sale ON Customer.CustomerID =
Sale.CustomerID
WHERE (SaleDate Between '01-MAR-2013' And '31-MAR-2013')
AND Customer.CustomerID IN
(SELECT CustomerID
FROM Sale
WHERE (SaleDate Between '01-MAY-2013' And '31-MAY-2013'));
```

List all of the customers who bought something in March and who bought something in May. (Saved views.)

```
CREATE VIEW MarchCustomers AS
SELECT CustomerID
FROM Sale
WHERE (SaleDate Between '01-MAR-2013' And '31-MAR-2013');

CREATE VIEW MayCustomers AS
SELECT CustomerID
FROM Sale
WHERE (SaleDate Between '01-MAY-2013' And '31-MAY-2013');

SELECT Customer.LastName, Customer.FirstName
FROM Customer
INNER JOIN MarchCustomers ON Customer.
CustomerID=MarchCustomers.CustomerID
INNER JOIN MayCustomers ON MarchCustomers.
CustomerID=MayCustomers.CustomerID;
```

Figure 5.17

Combining two separate lists with JOIN. You can save separate lists as views and use the JOIN command to retrieve only the values that match.

which views depend on other views. Some administrator could accidentally delete a view that is required by another query. On the other hand, the views could be re-used in multiple queries, which might save a developer time on a different project. The bottom line is that you need to know how to write the queries both ways, and choose the method that is best in each situation.

Subquery with ANY, ALL, and EXISTS

The **ANY** and **ALL** operators combine comparison of numbers with subsets. In the previous sections, the IN operator compared a value to a list of items in a set: however, the comparison was based on equality. The test item had to exactly match an entry in the list. The ANY and ALL operators work with a less than (<) or greater than (>) operator and compare the test value to a list of values.

Figure 5.18 illustrates the use of the ANY query. It is hard to find a solid business example that needs the ANY operator. In the example, it would be just as easy to use the subquery to find the minimum value (MIN function) in the list and then do the comparison. However, sometimes it is clearer to use the ANY operator.

The ALL operator behaves similarly, but the test value must be greater than all of the values in the list. In other words, the test value must exceed the largest value in the list. Hence, the ALL operator is much more restrictive.

The ALL operator can be a powerful tool—particularly when used with an equals (=) comparison. For instance, you might want to test whether one salesperson made all of the sales on a particular day. Figure 5.19 shows that the WHERE clause contains the statement: EmployeeID = ALL (SELECT EmployeeID FROM Sale WHERE SaleDate = '28-MAR'). The subquery returns a list of IDs for all employees who sold something on that date. The “= ALL” clause checks to see if all of the values are the same and match a single employee. This query is some-

List dog merchandise with a list price greater than the sale price of the cheapest cat product.

```
SELECT Merchandise.ItemID, Merchandise.Description,
Merchandise.Category, Merchandise.ListPrice
FROM Merchandise
WHERE Category=N'Dog'
AND ListPrice > ANY
(SELECT SalePrice
FROM Merchandise
INNER JOIN SaleItem ON Merchandise.ItemID=SaleItem.ItemID
WHERE Merchandise.Category=N'Cat')
;
```

Figure 5.18

Subquery with ANY and ALL. The example identifies any animal that sold for more than any of the prices of cats. Effectively, it returns values greater than the smallest entry in the subquery list.

what contrived, but it can be useful when you need to find a specific answer. The alternative in this situation is to count the number of sales by each employee on the specified date and visually check to see if there is more than one value. But, sometime you might want to find the exact answer using the ANY query.

Sometimes it is difficult to control the details returned from a subquery. Perhaps the data exists in a table created by someone else, such as a system table. In these cases, only the WHERE clause matters. Does the query return any rows that match the conditions? The **EXISTS** key word handles these situations. It is true if the subquery returns any rows of data—otherwise it is false. The specific columns returned are irrelevant. Figure 5.20 shows a simple example. In actual practice, the example would be better written with a JOIN statement, but it does illustrate how the EXISTS term works. The EXISTS term is useful when you need to see if rows are retrieved in a subquery but you do not want to match the actual values.

Figure 5.19

Subquery with All and equality test. The subquery returns a list of EmployeeID values who made sales on the specified date. The “= ALL” test checks to see if they are all the same value and returns the matching employee.

Has one salesperson made all of the sales on a particular day (Mar 28)?

```
SELECT Employee.EmployeeID, Employee.LastName
FROM Employee
WHERE EmployeeID = ALL
(SELECT EmployeeID
FROM Sale
WHERE SaleDate = '28-MAR-2013')
;
```

Use Not Exists to list customers who have not bought anything.

```
SELECT Customer.CustomerID, Customer.Phone, Customer.
LastName
FROM Customer
WHERE NOT EXISTS
(SELECT SaleID, SaleDate
FROM Sale WHERE Sale.CustomerID=Customer.CustomerID);
```

Figure 5.20

Subquery with Exists. If the only thing that matters is the WHERE clause, you can use the EXISTS phrase to test if rows are returned or not. It is also useful when the details of the subquery are difficult to change.

Correlated Subqueries

What are correlated subqueries? Recall the example in Figure 5.12 that asked: Which cat merchandise sold for more than the average sale price of cat merchandise? This example used a subquery to first find the average sale price of cat merchandise and then examined all sales of cat merchandises to display the ones that had higher prices. It is a reasonable business question to extend this idea to other categories of animals. Managers would like to identify all merchandise that was sold for a price greater than the average price of other merchandise within their respective categories (dog merchandise compared to other dog merchandise, fish compared to fish, and so on).

As shown in Figure 5.21, building this query is tricky. The merchandise category in the subquery has to match that in the outer query. This task is accomplished by setting the categories equal to each other. But, the Merchandise table is used in both queries, so the condition can only be written by assigning aliases to the Merchandise table in both queries. Here, it is renamed as Merchandise1 and

Figure 5.21

Correlated subquery. The condition in the subquery depends on values in the outermost query. In some query systems, this query could run slowly if large tables are involved.

Which merchandise has a list price greater than the average sale price of merchandise within that category?

```
SELECT Merchandise1.ItemID, Merchandise1.Description,
Merchandise1.Category, Merchandise1.ListPrice
FROM Merchandise AS Merchandise1
WHERE Merchandise1.ListPrice>
(
SELECT Avg(SaleItem.SalePrice) AS AvgOfSalePrice
FROM Merchandise As Merchandise2 INNER JOIN SaleItem ON
Merchandise2.ItemID = SaleItem.ItemID
WHERE Merchandise2.Category=Merchandise1.Category
);
```

Merchandise2, but any distinct names would work. This type of query is called a **correlated subquery**, because the subquery refers to data rows in the main query.

The query in Figure 5.21 will run. However, it might be inefficient. Performance depends on the query optimizer, but systems might have problems computing this query for large sets of data. Even on a fast computer, queries of this type have been known to run for several days without finishing. If the query is run as written, the calculation in the subquery must be recomputed for each entry in the main table. The problem is illustrated in Figure 5.22. Consider an inefficient DBMS that starts at the top row of the Merchandise table. When it sees the category is Dog, it computes the average sale price of dog merchandise (\$23.32). Then it moves to the next row and computes the average sale price for dogs again. In the worst case, the DBMS recomputes the average for every single row in the Merchandise table. Recomputing the average sale price for every single row in the main query is time-consuming. To compute an average, the DBMS must go through every row in the SaleItem table that has the same category of animal. Consider a relatively small query of 100,000 rows and five categories of animals. On average, there are 20,000 rows per category. To recompute the average each time, the DBMS will have to retrieve $100,000 * 20,000$ or 2,000,000,000 rows!

Unfortunately, you cannot just tell the manager that it is impossible to answer this important business question. Is there an efficient way to answer this question? Some query processors can automatically cache the averages. In other cases, you will have to do it yourself. The answer illustrates the power of SQL and highlights the importance of thinking about the problem before you try to write a query. The problem with the correlated subquery lies in the fact that it has to continually recompute the average for each category. Think about how you might solve this problem by hand. You would first make a table that listed the average for each category and then simply look up the appropriate value when you needed it. As shown in Figure 5.23, the same approach can be used with SQL. Just create the query for the averages using GROUP BY and save it. Then join it to the Merchandise table to do the comparison.

Figure 5.22

Potential problem with correlated subquery. The average is recomputed for every row in the main query. Every time the DBMS sees a dog product, it computes the average to be \$23.32. It is inefficient and slow to force the machine to recalculate the average each time.

MerchID Category ListPrice

1	Dog	\$45.00
2	Dog	\$65.00
3	Dog	\$85.00
4	Dog	\$110.00
5	Cat	\$25.00
6	Cat	\$35.00
7	Dog	\$4.00
8	Cat	\$3.00
9	Dog	\$7.50

Compute Avg: \$23.32

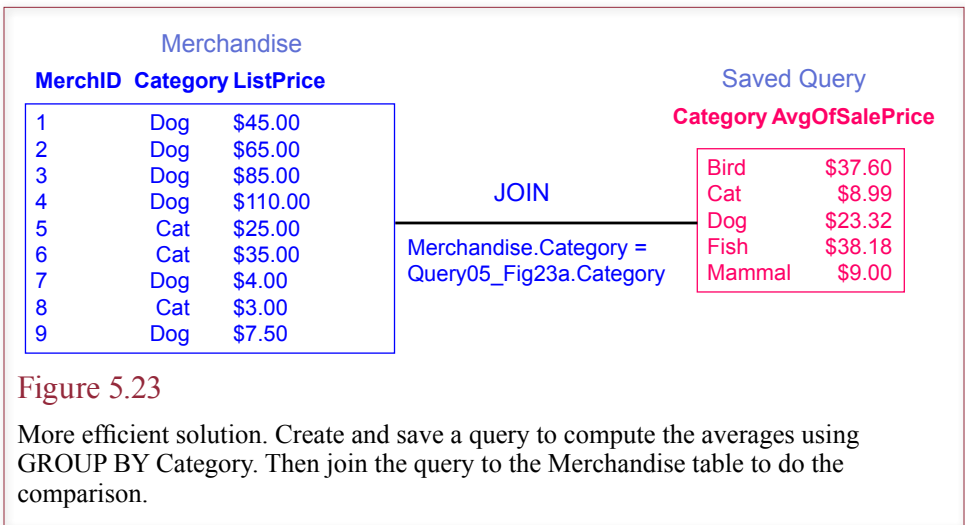
Compute Avg: \$23.32

Compute Avg: \$23.32

Compute Avg: \$23.32

Compute Avg: \$8.99

Recompute average
for every row in the
main query!



Today, you probably do not have to worry too much about the performance of correlated subqueries. The high-end DBMSs have good query optimizers that can recognize the problem and automatically find the solution to compute the values quickly and store them in a cache. However, some queries still require hand tuning. Also, you need to remember to look for different ways to approach queries. The solution in Figure 5.23 is much easier to read and verify that the answer is correct.

More Features and Tricks with SQL SELECT

What tricky problems arise and how do you handle them in SQL? As you may have noticed, the SQL SELECT command is powerful and has plenty of options. There are even more features and tricks that you should know about. Business questions can be difficult to answer. It helps to study different examples to gain a wider perspective on the problems and solutions you will encounter. One of the first big questions you will face is the need to combine rows from different tables. You also need to know how to handle several other complications, such as joining tables with multiple columns or inequality joins.

UNION, INTERSECT, EXCEPT

Codd originally conceived of tables as sets of data. The basic filtering aspects of the SELECT command perform some operations on these sets, but it is sometimes nice to be able to use more traditional set operators. Up to this point, the tables you have encountered have contained unique columns of data. The JOIN command links tables together so that a query can display and compare different columns of data from tables. Occasionally you will encounter a different type of problem where you need to combine rows of data from similar tables. The set operations, such as the **UNION** operator are designed to accomplish these tasks.

As an example, assume you work for a company that has offices in Los Angeles and New York. Each office maintains its own database. Each office has an Employee file that contains standard data about its employees. The offices are linked by a network, so you have access to both tables (call them EmployeeEast and EmployeeWest). But the corporate managers often want to search the entire Em-

```

SELECT EID, Name, Phone, Salary, 'East' As Office
FROM EmployeeEast
UNION
SELECT EID, Name, Phone, Salary, 'West' As Office
FROM EmployeeWest;

```

EID	Name	Phone	Salary	Office
352	Jones	3352	45,000	East
876	Inez	8736	47,000	East
372	Stoiko	7632	38,000	East
890	Smythe	9803	62,000	West
631	Kim	7736	73,000	West

Figure 5.24

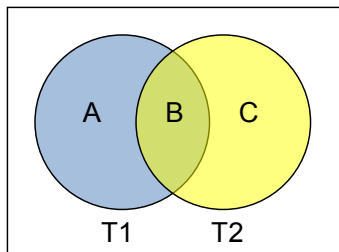
The UNION operator combines rows of data from two SELECT statements. The columns in both SELECT lines must match. The query is usually saved and used when managers need to search across both tables. Note the use of a new, constant column (Office) to track the source of the data.

ployee file—for example, to determine total employee salaries of the marketing department. One solution might be to run their basic query twice (once on each table) and then combine the results by hand.

As shown in Figure 5.24, the easier solution is to use the UNION operator to create a new query that combines the data from the two tables. All searches and operations performed on this new query will treat the two tables as one large table. By combining the tables with a view, each office can make changes to the original data on its system. Whenever managers need to search across the entire company,

Figure 5.25

Operators for combining rows from two tables. UNION selects all of the rows. INTERSECT retrieves only the rows that are in both tables. EXCEPT retrieves rows that exist in only one table.



T1 UNION T2	A + B + C
T1 INTERSECT T2	B
T1 EXCEPT T2	A

they use the saved query, which automatically examines the data from current versions of both tables.

The most important concept to remember when creating a UNION is that the data from both tables must match (e.g., EID to EID, Name to Name). Another useful trick is to insert a constant value in the SELECT statement. In this example the constant keeps track of which table held the original data. This value can also be used to balance out a SELECT statement if one of the queries will produce a column that is not available in the other query. To make sure both queries return the same number of columns, just insert a constant value in the query that does not contain the desired column. Make sure that it contains the same type of data that is stored in the other query (domains must match).

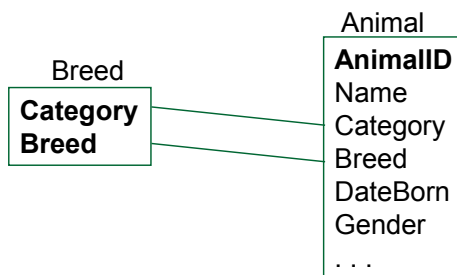
The UNION command combines matching rows of data from two tables. The basic version of the command automatically eliminates duplicate rows of data. If you want to keep all the rows—even the duplications, use the command UNION ALL. Two other options for combining rows are **EXCEPT** and **INTERSECT**. Figure 5.25 shows the difference between the three commands. They all apply to sets of rows and the Venn diagram shows that the tables might have some data in common (area B). The UNION operator returns all the rows that appear in either one of the tables, but rows appearing in both tables are only listed once. The INTERSECT operator returns the rows that appear in both tables (area B). The EXCEPT operator returns only rows that appear in the first table (area A). Notice that the result of the EXCEPT operator depends on which table is listed first. Microsoft Access supports only the UNION command. SQL Server (and other DBMSs) support all three. These set operators are another way to handle complex business questions, similar to the NOT IN problem of finding things in one set that are not in the second set. Just remember that there are often many ways to create a query.

Multiple JOIN Columns

Sometimes you will need to join tables based on data in more than one column. In the Pet Store example, each animal belongs to some category (Cat, Dog, Fish,

Figure 5.26

Multiple JOIN columns. The values in the tables are connected only when both the category and the breed match.



```

SELECT *
FROM Breed INNER JOIN Animal
ON Breed.Category = Animal.Category
AND Breed.Breed = Animal.Breed
  
```

```
SELECT Employee.EmployeeID, Employee.LastName, Employee.
ManagerID, E2.LastName
FROM Employee INNER JOIN Employee AS E2
ON Employee.ManagerID = E2.EmployeeID
```

EID	Name	Manager	Name
1	Reeves	11	Smith
2	Gibson	1	Reeves
3	Reasoner	1	Reeves

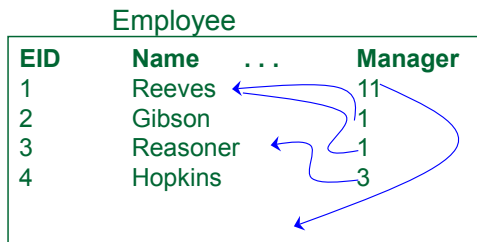


Figure 5.27

Reflexive JOIN to connect Employee table with itself. A manager is also an employee. Use a second copy of the Employee table (renamed to E2) to get the manager's name.

etc.). Each category of animal has different breeds. For example, a Cat might be a Manx, Maine Coon, or Persian; a Dog might be a Retriever, Labrador, or St. Bernard. A portion of the class diagram is reproduced in Figure 5.26. Notice the two lines connecting the Breed and Animal tables. This relationship ensures that only breeds listed in the Breed table can be entered for each type of Animal. A real store might want to include additional features in the Breed table (such as registration organization, breed description, or breed characteristics). The key point is that the tables must be connected by both the Category and the Breed.

In Microsoft Access QBE, the JOIN can be created by marking both columns and simultaneously dragging the two columns to the Animal table, but it is often easier to edit in SQL. The syntax for the SQL JOIN command is given in Figure 5.26. Simply expand the ON statement by listing both column connections. In this case, you want both sets of columns to be equal at the same time, so the statements are connected with an AND.

Reflexive Join

A **reflexive join** or **self-join** means simply that a table is joined to itself. One column in the table is used to match values in a second column in the same table. A common business example arises with an Employee table as illustrated in Figure 5.27. Employees typically have one manager. Hence the manager's ID can be stored in the row corresponding to each employee. The table would be Employee(EID, Name, Phone, . . . , Manager). The interesting feature is that a manager is also an employee, so the Manager column actually contains a value for EID. To get the corresponding name of the manager, you need to join the Employee table to itself.

List all the managers and their direct reports.

```

WITH DirectReports(EmployeeID, LastName, ManagerID, Title, Level)
AS
(
    --Root/anchor member (find employee with no manager)
    SELECT EmployeeID, LastName, ManagerID, Title, 0 As Level
    FROM Employee WHERE ManagerID=0    -- starting level
    UNION ALL
    -- Recursive members
    SELECT Employee.EmployeeID, Employee.LastName,
           Employee.ManagerID, Employee.Title, Level +1
    FROM Employee INNER JOIN DirectReports
    ON Employee.ManagerID = DirectReports.EmployeeID
)
-- Now execute the common table expression
SELECT ManagerID, EmployeeID, LastName, Title, Level
FROM DirectReports
ORDER BY Level, ManagerID, LastName

```

ManagerID	EmployeeID	LastName	Title	Level
0	11	Smith	Owner	0
11	1	Reeves	Manager	1
1	2	Gibson	Manager	2
1	3	Reasoner	Manager	2
2	6	Eaton	Animal Friend	3
2	7	Farris	Animal Friend	3
2	5	James	Animal Friend	3
2	9	O'Connor	Animal Friend	3
2	10	Shields	Animal Friend	3
3	8	Carpenter	Worker	3
3	4	Hopkins	Worker	3

Figure 5.28

Recursive query. The employee-manager relationship is a classic recursive example. The recursive query requires three steps: (1) Define the root level, (2) Define the recursion member that links to the higher level, and (3) Run the SELECT statement to execute the expression and sort the results.

The only trick with this operation is that you have to be careful with the ON condition. For instance, the following condition does not make sense: ON Employee.Manager = Employee.EID. The query would try to return employees who were their own managers, which is not likely to be what you wanted. Instead, you must use two instances of the Employee table and use an alias (say, E2) to rename the second copy. Then the correct ON condition becomes ON Employee.Manager = E2.EID. The key to self-joins is to make sure that the columns contain the same type of data and to create an alias for the second copy of the table.

SQL 1999 provides an even more powerful feature related to reflexive joins. Consider the employee example where you want to list all of the people who work for someone—not just the direct reports, but also the people who work for them, and the people who work for that group, and so on down the employee hierarchy tree. The standard provides the WITH RECURSIVE command that has several

options to search a data tree. Consider the pet store case with the partial Employee table: Employee(EmployeeID, LastName, Title, ManagerID). You want to start at the top with the CEO/owner and list all of the employees who report directly to a manager. For example, EmployeeID 1 (Reeves) is the only person who reports directly to Sally (EmployeeID=11), but two people (EmployeeID 2 and 3) report directly to Reeves. The actual syntax can be slightly different across systems. See the Workbooks for examples. Figure 5.28 shows the syntax used by SQL Server. The main difference with the standard is that the standard uses WITH RECURSIVE instead of just the WITH keyword. The main step is to define the common table expression to handle the recursion. You give a unique name (e.g., DirectReports) to the new expression and specify the columns that will be retrieved. The three main steps are: (1) Define the root starting point for the tree with a SELECT statement, (2) Define the recursive members with a second SELECT statement that links to the level above, and (3) Write the final SELECT statement to execute the recursive table and sort or group the results. In the example, root level is defined by choosing the owner who does not report to anyone (ManagerID=0). You might need to examine the data to know how to define the root level—it might be set by title, or by a Null value in some column. The second step is the one that does most of the work. You retrieve data from the Employee table, but the JOIN statement is the key. Notice that you join the Employee.ManagerID column to the higher-level DirectReports.EmployeeID table. The DirectReports table represents the parent level entry, and this SELECT statement will always have a similar JOIN condition. The third step is the easiest, because now you can treat the DirectReports entity as just another table. Open the pet store's Employee table and work through the results given here to see how the organization structure chart is created.

The Level column is also a useful trick. You define it with the root-level SELECT statement, and increment it with the recursive SELECT. It provides an easy way to specify the distance from the root. Picture the organizational chart with Smith at the top, followed by Reeves at Level 1, and Gibson and Reasoner at Level 2 because both report to Reeves. The recursive query is a powerful statement. Without it, you need to write substantial code to accomplish the same task.

Note that many lower-end systems (such as Microsoft Access) do not support recursive joins. In these cases, you will have to write programming code to iterate through each employee to build the tree. Also be cautious when building recursive queries—it is possible to accidentally create an infinite loop. You might want to set time limits on queries when testing recursive designs.

CASE Function

SQL 92 added the **CASE** function to simplify certain types of queries. However, many database systems have not yet implemented all the features of SQL 92. The CASE function evaluates a set of conditions and returns a single value. Similar to the Oracle decode function, the conditions can be simple (R=1) or complex.

Perhaps the managers want to classify the animals in Sally's Pet Store based on their age. Figure 5.29 shows the SQL statement that would create four categories based on different ages. Note the use of date arithmetic using today's date—Date()
)—and DateBorn. Whenever this query is executed, it will use the current day to assign each animal to the appropriate category. Of course, the next logical step is to run a GROUP BY query against this view to count the number of animals falling within each age category.

Convert age ranges into categories.

```
Select AnimalID,
      CASE
        WHEN Date()-DateBorn < 90 Then 'Baby'
        WHEN Date()-DateBorn >= 90
          AND Date()-DateBorn < 270 Then 'Young'
        WHEN Date()-DateBorn >= 270
          AND Date()-DateBorn < 365 Then 'Grown'
        ELSE 'Experienced'
      END
FROM Animal;
```

Figure 5.29

CASE function to convert DateBorn into age categories. Note the use of date arithmetic to generate descriptions that are always current.

Inequality Joins

A JOIN statement is actually just a condition. Most problems are straightforward and use a simple equality condition or **equi-join**. For example, the following statement joins the Customer and Order tables: FROM Customer INNER JOIN Order ON (Customer.CustomerID = Order.CustomerID).

SQL supports complex conditions including **inequality joins**, where the comparison is made with inequality operators (less than, greater than) instead of an equals sign. The generic name for any inequality or equality join is a theta join.

This type of join can be useful in some tricky situations. For example, consider a common business problem. You have a table for AccountsReceivable(TransactionID, CustomerID, Amount, DateDue). Managers would like to categorize the customer accounts and determine how many transactions are past due by 30, 90, and 120 or more days. This query can be built in a couple of ways. For instance, you could write three separate queries, or you could build a complex

Figure 5.30

Inequality join. Managers want to classify the AccountsReceivable (AR) data into three categories of overdue payments. First, store the business rules/categories in a new table. Then join the table to the AR data through inequality joins.

Classify payments by number of days late.

AR(TransactionID, CustomerID, Amount, DateDue)

LateCategory(Category, MinDays, MaxDays, Charge, ...)

Month	30	90	3%
Quarter	90	120	5%
Overdue	120	9999	10%

```
SELECT *
FROM AR INNER JOIN LateCategory
ON ((Date() - AR.DateDue) >= LateCategory.MinDays)
AND ((Date() - AR.DateDue) < LateCategory.MaxDays)
```

CASE statement. However, what happens if managers decide to change the business rules or add a new category? Then someone has to find your three queries and modify them. A more useful trick is to create a new table to hold the business rules or categories. In the example shown in Figure 5.30, create the table `LateCategory(Category, MinDays, MaxDays, Charge)`. This table defines the late categories based on the number of days past due. Now use inequality conditions to join the two tables. First, compute the number of days late using the current date (`Date() - AR.DateDue`). Finally, compare the number of days late to minimum and maximum values specified in the `LateCategory` table.

The ultimate value of this approach is that the business rules are now stored in a simple table (`LateCategory`). If managers want to change the conditions or add new criteria, they simply alter the data in the table. You can even build a form that makes it easy for managers to see the rules and quickly make the needed changes. With any other approach, a programmer needs to rewrite the code for the queries.

Exists and Crosstabs

Some queries need the `EXISTS` condition. Consider the business question: Which employees have sold merchandise in every category? The word *every* is the key here. Think about how you would answer that question if you did not have a computer. For each employee you would make a list of merchandise categories (Bird, Cat, Dog, etc.). Then you would go through the list of `ItemSales` and cross off each merchandise category sold by the employee. When finished, you would look at the employee list to see which people have every category crossed off (or an empty list). You will do the same thing using queries.

Remember, if this query returns any rows at all, then the selected employee has not sold every one of the categories. What you really want then is a list of employees for whom this query returns no rows of data. In other words, the rows from this query should `NOT EXIST`.

The next step is to examine the entire list of employees and see which ones do not retrieve any rows from the query in Figure 5.31. The final query is shown in Figure 5.32. Note that the specific `EmployeeID 5` has been replaced with the `EmployeeID` matching the value in the outer loop, which creates a correlated subquery. Unfortunately, you cannot avoid the correlated subquery in this type of

Figure 5.31

List the animal categories that have not been sold by `EmployeeID 5`. Use a basic `NOT IN` query.

List the Animal categories where merchandise has not been sold by an employee (#5).

```
SELECT Category
FROM Category
WHERE (Category <> N'Other') And Category NOT IN
(SELECT Merchandise.Category
FROM Merchandise INNER JOIN (Sale INNER JOIN SaleItem
ON Sale.SaleID = SaleItem.SaleID)
ON Merchandise.ItemID = SaleItem.ItemID
WHERE Sale.EmployeeID = 5)
```

Which employees have sold merchandise from every category?

```
SELECT Employee.EmployeeID, Employee.LastName
FROM Employee
WHERE Not Exists
(SELECT Category
FROM Category
WHERE (Category NOT IN (N'Other', N'Reptile', N'Spider')
And Category NOT IN
(SELECT Merchandise.Category
FROM Merchandise INNER JOIN (Sale INNER JOIN SaleItem
ON Sale.SaleID = SaleItem.SaleID)
ON Merchandise.ItemID = SaleItem.ItemID
WHERE Sale.EmployeeID = Employee.EmployeeID)
);
```

Figure 5.32

Example of NOT EXISTS clause. List the employees who have sold merchandise from every category (except “Other”).

problem. This query returns four employees who have sold every type of animal merchandise. Observe that categories for Other, Reptile, and Spider have been removed from the list because the shortened product list does not contain any items for these categories. Another way to handle this problem would be to select the Distinct Category from the Merchandise table instead of the Category table.

The type of query in Figure 5.32 is commonly used to answer questions that include some reference to “every” item. In some cases, a simpler solution is to

Figure 5.33

Using CASE to count items. The hard way to count items in each category. It works, but needs to be edited if categories are added.

Which employees have sold merchandise from every category?

```
SELECT Employee.EmployeeID, Employee.LastName,
Count(CASE Category WHEN 'Bird' THEN 1 END) As Bird,
Count(CASE Category WHEN 'Cat' THEN 1 END) As Cat,
Count(CASE Category WHEN 'Dog' THEN 1 END) As Dog,
Count(CASE Category WHEN 'Fish' THEN 1 END) As Fish,
Count(CASE Category WHEN 'Mammal' THEN 1 END) As Mammal,
Count(CASE Category WHEN 'Reptile' THEN 1 END) As Reptile,
Count(CASE Category WHEN 'Spider' THEN 1 END) As Spider
FROM Employee
INNER JOIN Sale ON Sale.EmployeeID=Employee.EmployeeID
INNER JOIN SaleAnimal ON Sale.SaleID=SaleAnimal.SaleID
INNER JOIN Animal ON Animal.AnimalID=SaleAnimal.AnimalID
GROUP BY Employee.EmployeeID, Employee.LastName
ORDER BY Employee.LastName;
```

just count the number of categories for each employee. One catch to this approach is that the DBMS must support the Count(DISTINCT) format. In general, these complex questions are probably better answered with multiple queries, or with tools provided by a data warehouse approach.

The query in Figure 5.32 is an interesting application of the EXISTS clause. However, there is an easier way to answer the question. You should build a crosstab or pivot query that counts the number of items sold by each employee and by each category. Notice that this question contains two “by each” statements. You could write a simple query that contains both of those variables (Employee and Category) in the GROUP BY section. However, most people find it easier to read the results if they are presented in a table, with one Group By variable (Employee) as the rows and the other (Category) as the columns. Then each cell can contain the count of the number of items sold for a specific employee in a given category.

Figure 5.33 shows the basic query. Microsoft Access has a simpler crosstab query, but with traditional SQL, you need to compute each column separately. Hence, you have to use the CASE function to select each category of animal—which means you have to know the categories ahead of time. Essentially, you compute each column separately by using a CASE statement to select only rows that match the group condition you want for the column.

Figure 5.34 shows the result of the crosstab query. It is relatively easy to see the types of animals sold by each employee. To answer the overall question of who sold items from each category, you simply look for a row with no zeros. With this sample data, four employees have sold at least one item from each category. Technically, this query contains more information that required to answer the question. However, additional data is often useful. If you write the EXISTS query to return exactly the information requested, it will return no names. Oftentimes, it is preferable to see that several other employees come close to meeting the conditions, instead of simply saying that no one meets them exactly.

Figure 5.34

Crosstab query. The columns are built using the CASE statement to select each specific category. The rows are formed by the GROUP BY clause. Note that Oracle uses the DECODE function instead of the CASE statement.

EID	LastName	Bird	Cat	Dog	Fish	Mammal
1	Reeves		4	15	6	
2	Gibson	1	25	24	9	2
3	Reasoner	2	9	26	5	2
4	Hopkins	3	21	33		
5	James	3	7	8	11	2
6	Eaton	1	2	8		1
7	Farris	1	4	24	1	1
8	Carpenter	3	1	11	5	
9	O'Connor		5	10	3	1
10	Shields	1		5		
11	Smith		1			

```

SELECT DISTINCT Table.Column {AS Alias}, ...
FROM Table/Query
INNER JOIN Table/Query ON T1.ColA = T2.ColB
WHERE (Condition)
GROUP BY Column
HAVING (Group Condition)
ORDER BY Table.Column
{UNION Second Select }

```

Figure 5.35

SQL SELECT options. Remember that WHERE statements can have subqueries.

SQL SELECT Summary

The SQL SELECT command is powerful and has many options. To help you remember the various options, they are presented in Figure 5.35. Each DBMS has a similar listing for the SELECT command, and you should consult the relevant Help system for details to see if there are implementation differences. Remember that the WHERE clause can have subqueries. Also remember that you can use the SELECT line to perform computations, both in-line and aggregations across the rows.

Most database systems are picky about the sequence of the various components of the SELECT statement. For example, the WHERE statement should come before the GROUP BY statement. Sometimes these errors can be hard to spot, so if you receive an enigmatic error message, verify that the segments are in the proper order. Figure 5.36 presents a mnemonic that may help you remember the proper sequence. Also, you should always build a query in pieces, so you can test each piece. For example, if you use a GROUP BY statement, first check the results without it to be sure that the proper rows are being selected.

SQL Data Definition Commands

What are the SQL data definition commands? Everything to this point has focused on only one aspect of a database: retrieving data. Clearly, you need to perform many more operations with a database. SQL was designed to handle all common operations. One set of commands is described in this section: data definition commands to create and modify the database and its tables. Note that the SQL commands can be cumbersome for these tasks. Hence, most modern database sys-

Figure 5.36

Mnemonic to help remember the proper sequence of the SELECT operators.

Someone	SELECT
From	FROM
Ireland	INNER JOIN
Will	WHERE
Grow	GROUP BY
Horseradish and	HAVING
Onions	ORDER BY

```
CREATE SCHEMA AUTHORIZATION DBName Password
CREATE TABLE TableName (Column Type, ...)
ALTER TABLE Table {Add, Column, Constraint, Drop}
DROP {Table TableName | Index IndexName ON TableName}
CREATE INDEX IndexName ON TableName (Column ASC/DESC)
```

Figure 5.37

Primary SQL data definition commands. In most cases you will avoid these commands and use a visual or menu-driven system to define and modify tables.

tems provide a visual or menu-driven system to assist with these tasks. The SQL commands are generally used when you need to automate some of these tasks and set up or make changes to a database from within a separate program.

The five most common data definition commands are listed in Figure 5.37. In building a new database, the first step is to **CREATE a SCHEMA**. A **schema** is a collection of tables. In some systems, the command is equivalent to creating a new database. In other systems, it simply defines a logical area where each user can store tables, which might or might not be in one physical database. The Authorization component describes the user and sets a password for security. Most DBMSs also have visually-oriented tools to perform these basic tasks. However, the SQL commands can be scripted and stored in a file that can be run whenever you need to recreate the database.

CREATE TABLE is one of the main SQL data definition commands. It is used to define a completely new table. The basic command lists the name of the table along with the names and data types for all of the columns. Figure 5.38 shows the format for the data definition commands. Additional options include the ability to assign default values with the **DEFAULT** command.

SQL 92 provides several standard data types, but system vendors do not yet implement all of them. SQL 92 also enables you to create your own data types with the **CREATE DOMAIN** command. For example, to ensure consistency you

Figure 5.38

The **CREATE TABLE** command defines a new table and all of the columns that it will contain. The **NOT NULL** command typically is used to identify the key column(s) for the table. The **ALTER TABLE** command enables you to add and delete entire columns from an existing table.

```
CREATE TABLE Customer
(
    CustomerID    INTEGER NOT NULL,
    LastName      NVARCHAR(10),
    ...
);

ALTER TABLE Customer
    DROP COLUMN ZIPCode;

ALTER TABLE Customer
    ADD COLUMN CellPhone NVARCHAR(15);
```

```

CREATE TABLE Order
  (OrderID      INTEGER NOT NULL,
   OrderDate    DATE,
   CustomerID   INTEGER,

   CONSTRAINT pkOrder PRIMARY KEY (OrderID),
   CONSTRAINT fkOrderCustomer FOREIGN KEY (CustomerID)
     REFERENCES Customer (CustomerID)
);

```

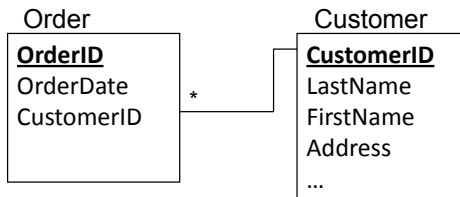


Figure 5.39

Identifying primary and foreign keys in SQL. Keys are defined as constraints that are enforced by the DBMS. The primary key constraint lists the columns that make up the primary key. The foreign key lists the column (CustomerID) in the current table (Order) that is linked to a column (CustomerID) in a second table (Customer).

could create a domain called DomAddress that consists of CHAR (35). Then any table that used an address column would refer to the DomAddress.

With SQL 92, you identify the primary key and foreign key relationships with constraints. SQL **constraints** are rules that are enforced by the database system. Figure 5.39 illustrates the syntax for defining both a primary key and a foreign key for an Order table. First, notice that each constraint is given a name (e.g., pkOrder). You can choose any name, but you should pick one that you will recognize later if problems arise. The primary key constraint simply lists the column or columns that make up the primary key. Note that each column in the primary key should also be marked as NOT NULL.

The foreign key constraint is easier to understand if you examine the relevant class diagram. Here you want to place orders only to customers who have data in the Customer table. That is, the CustomerID in the Order table must already exist in the Customer table. Hence, the constraint lists the column in the original Order table and then specifies a REFERENCE to the Customer table and the CustomerID.

The **ALTER TABLE** and **DROP TABLE** commands enable you to modify the structure of an existing table. Be careful with the DROP command, as it will remove the entire table from the database, including its data and structural definition. The ALTER TABLE command is less drastic. It can be used to ADD or DELETE columns from a table. Obviously, when you drop an entire column, all the data stored in that column will be deleted. Similarly, when you add a new column, it will contain NULL values for any existing rows.

You can use the CREATE INDEX and DROP INDEX commands to improve the performance of the database. Indexes can improve performance when re-

trieving data, but they can cause problems when many transactions take place in a short period of time. In general, these commands are issued once for a table. Typically, indexes are built for primary key columns. Most DBMSs automatically build those indexes.

Finally, as described in Chapter 4, the **CREATE VIEW** creates and saves a new query. The basic syntax is straightforward: *CREATE VIEW myview AS SELECT...* The command simply gives a name and saves any **SELECT** statement. Again, these commands are almost always easier to create and execute from a menu-driven interface. However, because you may have to create SQL data definition statements by hand sometime, so it is good to know how to do so.

SQL Data Manipulation Commands

What SQL commands alter the data stored in tables? A third set of SQL commands demonstrates the true power of SQL. The **SELECT** command retrieves data, whereas data manipulation commands are used to change the data within the tables. The basic commands and their syntax are displayed in Figure 5.40. These commands are used to insert data, delete rows, and update (change) the values of specific cells. Remember two points when using these commands: (1) They operate on sets of data at one time—avoid thinking in terms of individual rows, and (2) they utilize the power of the **SELECT** and **WHERE** statements you already know.

INSERT and DELETE

As you can tell from Figure 5.40, the **INSERT** command has two variations. The first version (**VALUES**) is used to insert one row of data at a time. Except for some programming implementations, it is not very interesting. Most database systems provide a visual or tabular data entry system that makes it easy to enter or edit single rows of data. Generally, you will build forms to make it easy for users to enter and edit single rows of data. These tools automatically build the single-row **INSERT** command. On most systems, the data will be inserted directly to the tables. In a few cases, you might have to write your own **INSERT** statement.

The second version of the **INSERT** command is particularly useful at copying data from one table into a second (target) table. Note that it accepts any **SE-**

Figure 5.40

Common SQL commands to add, delete, and change data within existing tables. The commands operate on entire sets of data, and they utilize the power of the **SELECT** and **WHERE** statements, including subqueries.

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

INSERT INTO target (column1, column2, ...)
    SELECT ... FROM ...

DELETE FROM table WHERE condition

UPDATE table
    SET Column1=Value1, Column2=Value2, ...
    WHERE condition
```

```
INSERT INTO OldCustomers
SELECT *
FROM Customer
WHERE CustomerID IN
(SELECT Sale.CustomerID
FROM Customer INNER JOIN Sale
ON Customer.CustomerID=Sale.CustomerID
GROUP BY Sale.CustomerID
HAVING Max(Sale.SaleDate) < '01-Jul-2013' );
```

Figure 5.41

INSERT command to copy older data rows. Note the use of the subquery to identify the rows to be copied.

LECT statement, including one with subqueries, making it far more powerful than it looks. For example, in the Pet Store database, you might decide to move older Customer data to a different computer. To move records for customers who have not purchased anything since the start of July, you would issue the INSERT command displayed in Figure 5.41. Notice that the subquery selects the customers based on the date they placed their latest sale. The INSERT command then copies the associated rows in the Customer table into an existing OldCustomers table.

The query in Figure 5.41 just copies the specified rows to a new table. The next step is to delete them from the main Customer table to save space and improve performance. The **DELETE** command performs this function easily. As Figure 5.42 illustrates, you simply replace the first two rows of the query (INSERT and SELECT) with DELETE. Be careful not to alter the subquery. You can use the cut-and-paste feature to delete only rows that have already been copied to the backup table. Be sure you recognize the difference between the DROP and DELETE commands. The DROP command removes an entire table. The DELETE command deletes rows within a table.

UPDATE

The syntax of the **UPDATE** command is similar to the INSERT and DELETE commands. It, too, makes full use of the WHERE clause, including subqueries. The key to the UPDATE command is to remember that it acts on an entire collec-

Figure 5.42

DELETE command to remove the older data. Use cut and paste to make sure the subquery is exactly the same as the previous query.

```
DELETE
FROM Customer
WHERE CustomerID IN
(SELECT FROM Customer INNER JOIN Sale
ON Customer.CustomerID=Sale.CustomerID
GROUP BY Sale.CustomerID
HAVING (Max(Sale.SaleDate) < '01-Jul-2013' );
```

```
UPDATE Merchandise
SET ListPrice = ListPrice * 1.10
WHERE Category = 'Cat';

UPDATE Merchandise
SET ListPrice = ListPrice * 1.20
WHERE Category = 'Dog';
```

Figure 5.43

Sample UPDATE command. If the CASE function is not available, use two separate statements to increase the list price by 10 percent for cats and 20 percent for dogs.

tion of rows at one time. You use the WHERE clause to specify which set of rows need to be changed.

In the example in Figure 5.43, managers wish to increase the ListPrice of the merchandise for cats and dogs. The price for cat merchandise should increase by 10 percent and the price for dog merchandise by 20 percent. Because these are two different categories, you will often use two separate UPDATE statements. However, this operation provides a good use for the CASE function. You can reduce the operation to one UPDATE statement by replacing the 1.10 and 1.20 values with a CASE statement that selects 1.10 for Cats and 1.20 for Dogs.

The UPDATE statement has some additional features. For example, you can change several columns at the same time. Just separate the calculations with a comma. You can also build calculations from any row within the table or query. For example, merchandise list price could take into consideration the quantity on hand with the command `SET ListPrice = ListPrice*(1 - 0.001*QuantityOnHand)`. This command takes 1/10 of 1 percent off the price for extra items in inventory.

Notice the use of the internal `Date()` function to provide today's date in the last example. Most database systems provide several internal functions that can be used within any calculation. These functions are not standardized, but you can generally get a list (and the syntax chart) from the system's Help commands. The Date, String, and Format functions are particularly useful.

When using the UPDATE command, remember that all the data in the calculation must exist on one row within the query. There is no way to refer to a previous or next row within the table. If you need data from other rows or tables, you can build a query to join tables. However, you can update data in only a single table at a time.

Quality: Testing Queries

How do you know if your query is correct? The greatest challenge with complex queries is that even if you make a mistake, you usually get results. The problem is that the results are not the answer to the question you wanted to ask. The only way to ensure the results are correct is to thoroughly understand SQL, to build your queries carefully, and to test your queries.

Figure 5.44 outlines the basic steps for dealing with complex queries. The first step is to break complex queries into smaller pieces, particularly when the query involves subqueries. You need to examine and test each subquery separately. You can do the same thing with complex Boolean conditions. Start with a simple condition, check the results, and then add new conditions. When the subqueries are

```

Break questions into smaller pieces.
Test each query.
    Check the SQL.
    Look at the data.
    Check computations.
Combine into subqueries.
    Use the cut-and-paste features to reduce errors.
    Check for correlated subqueries.
Test sample data.
    Identify different cases.
    Check final query and subqueries.
    Verify calculations.
Test SELECT queries before executing UPDATE queries.
Optimize queries that run multiple times.
    Run a query optimizer.
    Think about new ways to structure the query.

```

Figure 5.44

Steps to building quality queries. Be sure there are recent backups of the database before you execute UPDATE or DELETE queries.

correct, use cut-and-paste techniques to combine them into one main query. If necessary, save the initial queries as views, and use a completely new query to combine the results from the views. The third step is to create sample data to test the queries. Find or create data that represents the different possible cases. Optimize queries that will become part of an application and run multiple times. Most DBMSs have an optimizer that will suggest performance improvements. You should also look for alternate ways to write the query to find a faster approach.

Figure 5.45

Sample query: Which customers who adopted dogs also bought cat products (at any time)? Build each query separately. Then paste them together in SQL and add the connecting link. Use sample data to test the results.

```

SELECT DISTINCT Animal.Category, Sale.CustomerID
FROM Sale INNER JOIN Animal
  ON Animal.SaleID = Sale.SaleID
WHERE (Animal.Category=N'Dog')

      AND Sale.CustomerID IN (

      SELECT DISTINCT Sale.CustomerID
      FROM Sale INNER JOIN (Merchandise INNER JOIN
SaleItem
  ON Merchandise.ItemID = SaleItem.ItemID)
  ON Sale.SaleID = SaleItem.SaleID
      WHERE (Merchandise.Category=N'Cat')
      );

```

In terms of quality issues, consider the example in Figure 5.45: List customers who adopted dogs and also bought cat products. The query consists of four situations:

1. Customers adopted dogs and cat products on the same sale.
2. Customers adopted dogs and then cat products at a different time.
3. Customers adopted dogs and never bought cat products.
4. Customers never adopted dogs but did buy cat products.

Because there are only four cases, you should create data and test each one. If there were thousands of possible cases, you might have to limit your testing to the major possibilities.

The final step in building queries involves data manipulation queries (such as UPDATE). You should first create a SELECT query that retrieves the rows you plan to change. Examine and test the rows to make sure they are the ones you want to alter. When you are satisfied that the query is correct, make sure you have a recent backup of the database—or at least a recent copy of the tables you want to change. Now you can convert the SELECT query to an UPDATE or DELETE statement and execute it.

Summary

Always remember that SQL operates on sets of data. The SELECT command returns a set of data that matches some criteria. The UPDATE command changes values of data, and the DELETE command deletes rows of data that are in a specified set. Sets can be defined in terms of a simple WHERE clause. The key to understanding SQL is to think of the WHERE clause as defining a set of data.

To create queries to answer complex business questions, break the question into pieces and build simple queries to retrieve data for each piece. Then combine the sets of data using inner joins, outer joins, subqueries, or set operators. Subqueries are powerful, but be careful to ensure that the query accurately represents the business questions. You must test subqueries in pieces and make sure you understand exactly what each piece is returning.

In everyday situations, data can exist in one table but not another. For example, you might need a list of customers who have not placed orders recently. The problem can also arise if the DBMS does not maintain referential integrity—and you need to find which orders have customers with no matching data in the customer table. Outer joins (or the NOT IN subquery) are useful in these situations.

The most important thing to remember when building queries is that if you make a mistake, most likely the query will still execute. Unfortunately, it will not give you the results you wanted. That means you have to build your queries carefully and always check your work. Begin with a smaller query and then add elements until you get the query you want. To build an UPDATE or DELETE query, always start with a SELECT statement and check the results. Then change it to UPDATE or DELETE.




A Developer's View

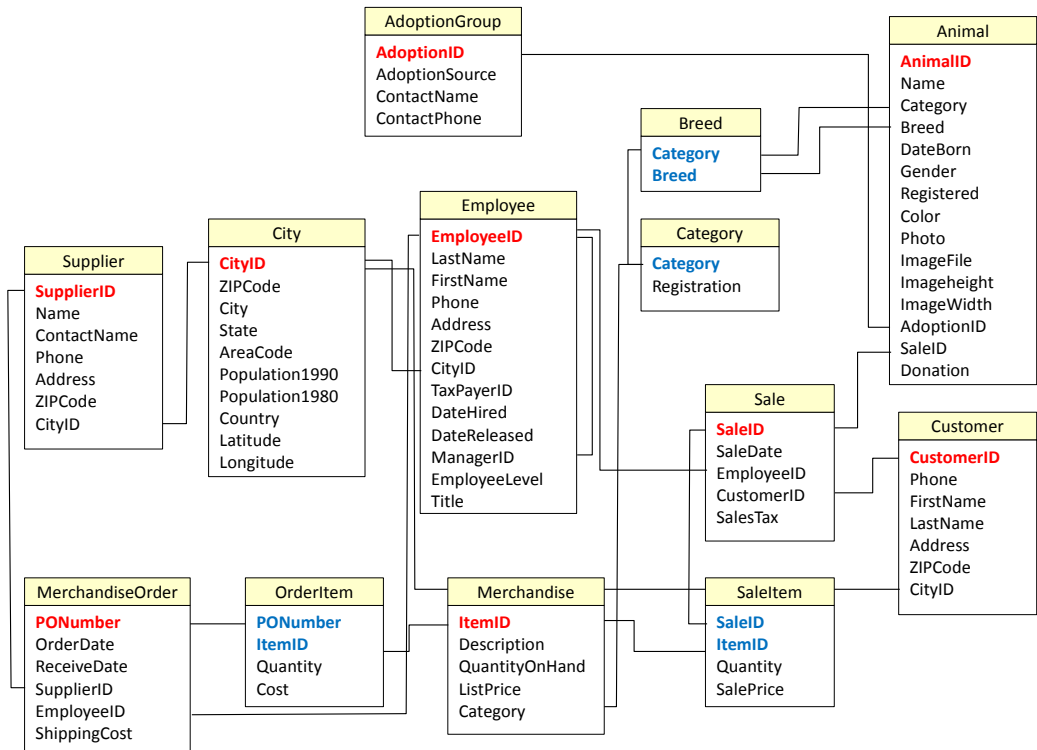
Miranda saw that some business questions are more complex than others. SQL subqueries and outer joins are often used to answer these questions. Practice the SQL subqueries until you thoroughly understand them. They will save you hundreds of hours of work. Think about how long it would take to write code to answer some of the questions in this chapter! For your class project, you should create several queries to test your skills, including subqueries and outer joins. You should build and test some SQL UPDATE queries to change sets of data. You should be able to use SQL to create and modify tables.

Key Terms

ALL	FULL JOIN
ALTER TABLE	IN
ANY	inequality join
CASE	INSERT
constraint	INTERSECT
correlated subquery	LEFT JOIN
CREATE DOMAIN	nested query
CREATE SCHEMA	outer join
CREATE TABLE	reflexive join
CREATE VIEW	RIGHT JOIN
DELETE	schema
DROP TABLE	self join
equi-join	subquery
EXCEPT	UNION
EXISTS	UPDATE

Review Questions

1. What is a subquery and in what situations is it useful?
2. What is a correlated subquery and why does it present problems?
-  3. How do you find items that are not in a list, such as customers who have not placed orders recently?
4. How do you join tables when the JOIN column for one table contains data that is not in the related column of the second table?
5. How do you join a column in one table to a related column in the same table?
6. What are inequality joins and when are they useful?
-  7. What is the SQL UNION command and when is it useful?
8. What is the purpose of the SQL CASE function?
9. What are the basic SQL data definition commands?
10. What are the basic SQL data manipulation commands?
-  11. How are UPDATE and DELETE commands similar to the SELECT statement?



Exercises



Sally's Pet Store

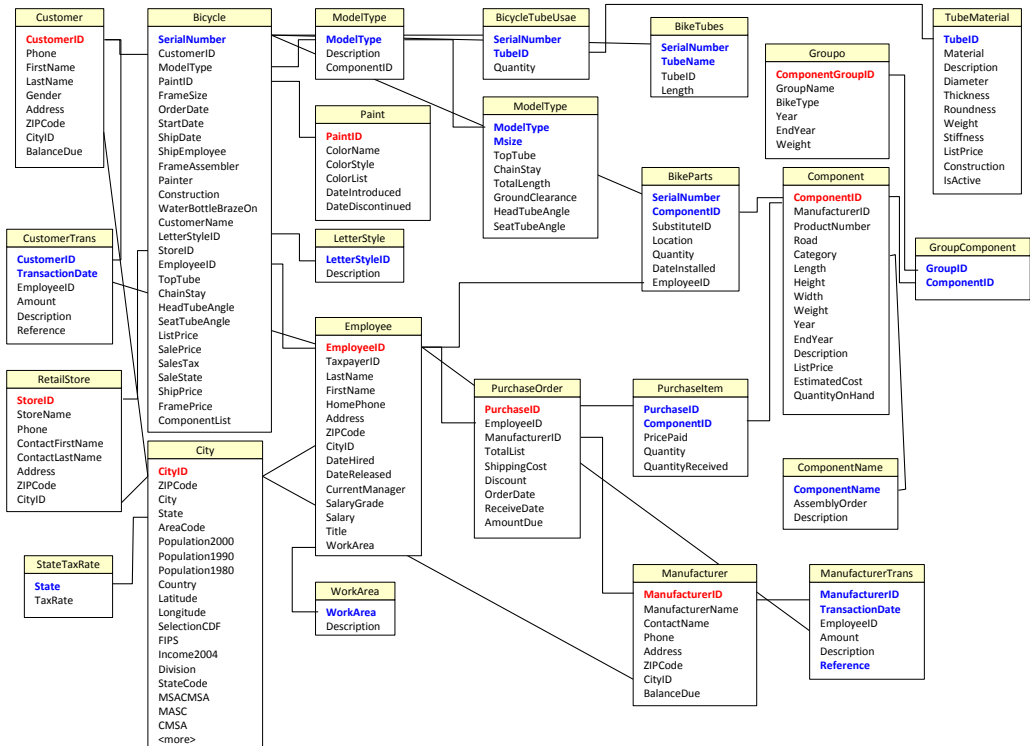
Write the SQL statements that will answer questions 1 through 16 based on the tables in the Pet Store database. Test your queries in the database. Hint: Many are easier if you split the question into multiple queries.

1. Which suppliers did not deliver any items in September?
- ✓ 2. Which employees did not sell any items in June?
3. Which categories of merchandise were not sold during May?
- ✓ 4. Which breed of Cat has never been adopted through the store?
5. What was the percentage of sales value by merchandise category in March?
6. Which category of animal was most likely (percent) to be adopted in the first three months?
- ✓ 7. Which employee had the highest percent of the number of sales (not value) in January?
8. Which supplier has the highest average percentage of shipping cost to total order value?
9. List the total adoptions and percentage by adoption group in April.

10. Which employee has been the top monthly seller the most number of times?
11. What is the amount of money customers spent on cat products after they adopted a cat?
12. List customers who purchased Cat merchandise in January and March?
13. List employees who ordered items from the same supplier in March and April (could be different products).

Make a backup copy before attempting the remaining Pet Store queries.

14. Write the SQL CREATE TABLE command to create a new Employee table with no data.
15. Write the SQL command to copy the data to the new Employee table for employees who did not sell anything in December.
16. Write the SQL command to delete the employees from the original Employee table who did not sell anything in December—except for Ms. Smith, the owner.
17. Write a query to increase the list price of Dog merchandise by 5 percent.




Rolling Thunder Bicycles


Write the SQL statements that will answer questions 17 through 32 based on the tables in the Rolling Thunder database. Build your queries in Access.

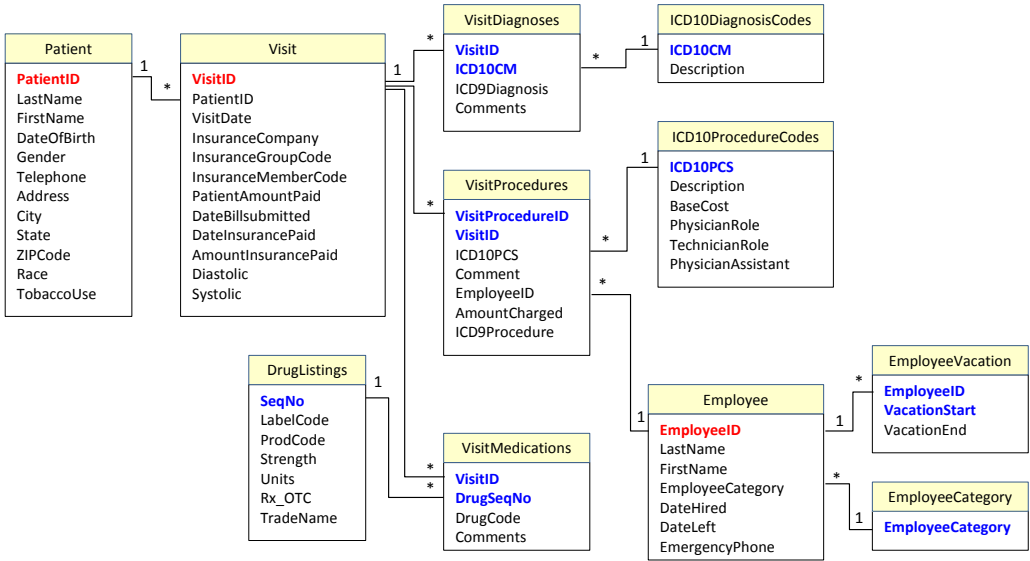
18. Which employee has been #1 in monthly sales value for the most number of months in 2010-2011?
- ✓ 19. Which paint colors were not used in 2012?
20. What percent of race bikes sold in 2012 used Shimano, Campy, and SRAM cranks? (Give the percent of the total for each manufacturer.)
21. List customers who bought a full suspension mountain bike after they had purchased a regular mountain bike.
22. List all of the people who are managed by Roland Venetiaan.
23. In 2012, which employees who took an order for a bicycle also shipped that same bicycle? Hint: Connect the Employee table to ShipEmployee.
24. Compute the percentage of value of sales by model type for each year 2010 – 2013.
- ✓ 25. Using a UNION query, list the employees who painted bicycles on March 15, 2012 or framed them on that date (StartDate) (or both). Hint: Join the Employee table to Painter and then to FrameAssembler.

26. In 2012, what percent of bicycle sales (by count) were made without the help of a retail store (StoreID=1 or 2).
27. Which manufacturers did not sell any items to Rolling Thunder Bicycles in 2012?
28. For Road component groups in 2012 (Component.Year), what is the average percent of the total group weight contributed by the crank?
29. How has the percent share of sales value for Race bikes (to total) changed over time (by year)?
30. Use SQL to create a new SalesRanking table as shown.



Category	SalesLow	SalesHigh
Top	0.10	1.0
Acceptable	0.05	0.10
Weak	0.0	0.05

31. Write the query to insert the rows of data row into the table shown in the previous exercise.
 32. Create a query to compute sales by employee by month and the employee's percent of total monthly sales. Combine the table data from the table in the previous exercise to assign the appropriate category to each employee for each month.
 33. Write a query to delete the last row (Weak) in the new SalesRanking table.
 34. Write a query to delete the entire SalesRanking table.
- 



Corner Med

35. List the physicians and the percentage of patients/visits seen by each one for the month of May. Do not include non-physicians in the computations.
36. For the year, list the top 10 diagnoses and the percentage of times each was applied.
- ✓ 37. For the month of March, list the percentage of visits covered by each type of insurance company.
38. For each month, compute the percentage of the number of visits by patient gender.
39. List the patients who returned for at least one visit after being diagnosed with J069 (respiratory infection).
40. Which two-letter procedures have not been performed?
- ✓ 41. What is the average number of medications prescribed per visit for each physician?
42. Which patients who have been diagnosed with ICD10 code E784 have also been diagnosed (at any time) with code E039?
43. Which patients have been seen by all three physicians (at any time)?
44. Create a summarization of patients that show the percentage by gender and tobacco use.
45. Use SQL to create a table (VisitCategory) that can be used to categorize patients by the number of visits in a year:

Category	MinVisits	MaxVisits
Many	2	20
Seldom	1	2
Rare	0	1

46. Write the INSERT commands to add the rows in the table for the previous exercise.
47. Write a query using the table in the previous query to categorize the patients by number of visits for one year.
48. Write the SQL command to change the MaxVisits value in the “Many” row to 30.
49. Write the SQL command to remove the table.
- ✓ 50. The GEMICD9xICD10_CM crosswalk table matches the older ICD9 diagnostic codes to the newer ICD10 codes. Create a query that ignores the NoDx entries. Create a second query to find the older ICD9 codes in the VisitDiagnoses table that do not have an official match in the new ICD10 code. (Ignore the ICD10 values in the VisitDiagnoses table—which were created using this process.) Bonus: How would you find codes for the ICD9 entries that are missing cross matches?
51. The GEMICD9xICD10_PCS crosswalk table matches procedure codes between the older ICD9 and newer ICD10 classifications. Assume that the VisitProcedures table has only the older ICD9 procedure code and blank values for the ICD10 codes. Write the query to use the crosswalk table to match the values and transfer the correct ICD10 entry into the VisitProcedures table. Note: If you run the query, make a backup copy of the table and database.
52. The GEMICD9xICD10_PCS crosswalk table maps older ICD9 procedure codes to the newer ICD10 codes. Are any of the ICD9 codes mapped to more than one ICD10 code? If so, in the process used in the previous exercise, what will happen? Which codes will be transferred?

Web Site References

http://www.sigmod.org/	Association for Computing Machinery— Special Interest Group: Management of Data.
http://www.acm.org/dl	ACM digital library containing thousands of searchable full-text articles. Check library.
http://www.oracle.com/technetwork/indexes/ documentation/index.html	Oracle online documentation library, including SQL Reference. (Version db102 will change.)
http://msdn.microsoft.com/en-us/library/ ms130214.aspx	Microsoft SQL Server Books Online reference.
http://pic.dhe.ibm.com/infocenter/db2luw/v9r7/ index.jsp	IBM DB2 reference.
http://dev.mysql.com/doc/	MySQL Reference.

Additional Reading

- Celko, J., Joe Celko’s SQL Puzzles and Answers, 2e, San Mateo: Morgan Kaufmann, 2006. [Challenging SQL problems with solutions.]
- Faroult, Stephane, The Art of SQL, O’Reilly, 2006. [Strategy and performance in building queries.]

Appendix: Introduction to Programming

Many books will help you learn to write computer programs. The purpose of this appendix is to review the highlights of programming and to point out some of the features that are important to programming within a DBMS. If you are new to programming, you should consider reading several other books to explain the details and logic behind programming.

Variables and Data

One of the most important consequences of programming in a database environment is that there can be three categories of data: (1) data stored in a table, (2) data held in a control on a form or report, and (3) traditional data variables that are used to hold temporary results. Chapter 3 focuses on storing data within tables. Chapter 6 describes how to create forms and the role of data controls. Chapter 8 provides more details of how the three types of variables interact when building applications. For now, you must learn details about basic programming variables.

Any procedure can create variables to hold data. A program **variable** is like a small box: it can hold values that will be used or transferred later. Variables have unique names. More importantly, variables can hold a certain **data type**. Common types of variables are displayed in Figure 5.1A. They can generally be classified into three categories: integers (1, 2, -10, ...); reals (1.55, 3.14, ...); and strings ("123 Main Street", "Jose Rojas", ...).

Each type of variable takes up a defined amount of storage space. This space affects the size of the data that the variable can hold. The exact size depends on the particular DBMS and the operating system. For example, a short integer typically takes 2 bytes of storage, which is 16 bits. Hence it can hold 216 values or numbers between -32,768 and 32,767. Real numbers can have fractional values. There are usually two sizes: single and double precision. If you do not need many variables, it is often wise to choose the larger variables (integers and double-precision reals). Although double-precision variables require more space and take longer to process, they provide room for expansion. If you choose too small of a variable, a user might crash your application or get invalid results. For example, it would be a mistake to use a 2-byte integer to count the number of customers—since a firm could generally anticipate having more than 65,000 customers. Along the same

Figure 5.1A

Program variable types. Ranges are approximate but supported by most vendors. Note that decimal variables help prevent round-off errors

Type	Bytes	Range
Short Integer	2	-32,768 to 32,767
Long Integer	4	+/- 2,147,483,647
	8	+/- 9,223,372,036,854,775,807
Float	4	+/- 1.5 10e45 (7 digits)
Double	8	+/- 5.0 10e324 (15 digits)
Decimal	16	+/- 1.0 10e28 (28 digits)
String	any	any

lines, you should use the Currency data type for monetary values. In addition to handling large numbers, it avoids round-off errors that are common to floating-point numbers.

Variable Scope

The scope and lifetime of a variable are crucial elements of programming, particularly in an event-driven environment. Variable **scope** refers to where the variable is accessible, that is, which procedures or code can access the data in that variable. The **lifetime** identifies when the variable is created and when it is destroyed. The two properties are related and are generally automatic. However, you can override the standard procedures by changing the way you declare the variable. In most systems, the scope and lifetime are based on where the variable is declared.

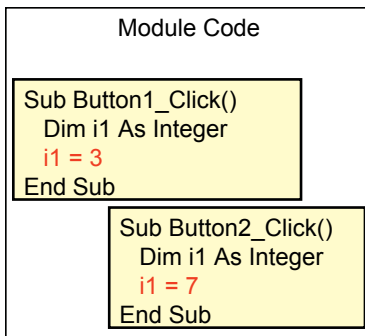
All data variables should be explicitly declared: they should be identified before they are used. The Basic language uses a Dim statement to declare variables. Many other languages declare variables by specifying the data type first. Most commonly, the variable is created within the event procedure and is a **local variable**. When the procedure starts, the local variable is created. Any code within that procedure can use the variable. Code in other procedures cannot see the variable. When the procedure ends, the local variable and its data are destroyed.

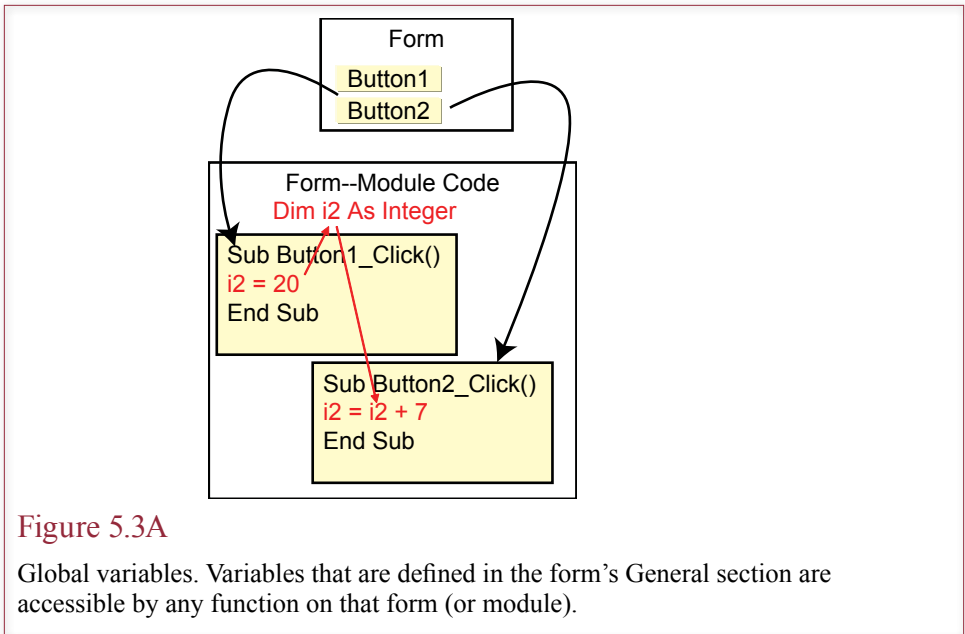
Figure 5.2A shows two buttons on a form. Each button responds to a Click event, so two procedures are defined. Each procedure can have a variable called *i1*, but these two variables are completely separate. In fact, the variables are not created until the button is clicked. Think of the procedures as two different rooms. When you are in one room, you can see the data for that room only. When you leave the room, the data is destroyed.

However, what if you do not want the data to be destroyed when the code ends, or you want to access the variable from other procedures? You have two choices: (1) Change the lifetime of the variable by declaring it static, or (2) Change the scope of the variable by declaring it in a different location. You should avoid declaring a static variable unless it is absolutely necessary (which is rare). If the variable is static, it keeps its value from the previous time the procedure was called. In the example, each time the button is clicked, the value for *i3* will remain from the prior click. You might use this trick if you need to count the number of times the button is clicked.

Figure 5.2A

Variable scope and lifetime. Each event has its own procedure with independent variables that are created and destroyed each time a routine is executed.





A more useful technique is to change where the variable is defined. Figure 5.3A shows that event procedures are defined within a form or a **module**, which is a collection of related procedures. The variable `i2` is defined for the entire form or module. The lifetime of the variable is established by the form, that is, the variable is created and destroyed as the form is opened and closed. The scope of the variable is that all procedures in the form can see and change the value. On the other hand, procedures in other forms or modules do not know that this variable exists.

Procedures or functions also have a scope. Any procedure that you define on a form can be used by other procedures on that form. If you need to access a variable or a procedure from many different forms or reports, you should define it on a separate module and then declare it as global (or public).

Be careful with global or public variables. A programmer who tries to revise your code might not know that the variable is used in other procedures and might accidentally destroy an important value. On forms the main purpose of a global variable is to transfer a value from one event to another one. For example, you might need to keep the original value of a text control—before it is changed by a user—and compare it to the new value. You need a global variable because two separate events examine the text control: (1) The user first enters the control, and (2) The user changes the data. It is sometimes difficult to create global or shared variables in certain systems. In these cases, you might need to store the global variables within a special database table.

Computations

One of the main purposes of variables is to perform calculations. Keep in mind that these computations apply to individual variables—one piece of data at a time. If you need to manipulate data in an entire table, it is usually best to use the SQL commands described in Chapter 5. Nonetheless, there are times when you need more complex calculations.

Operation	Common Syntax
Arithmetic	+ - * /
Exponentiation	^ or Power
Integer Divide	\
Modulus	mod

Figure 5.4A

Common arithmetic operators. Add (+), subtract (-), multiply (*), and divide (/). Exponentiation and integer arithmetic are often used for special tasks. For example, integer arithmetic is useful for dividing objects into groups.

Standard arithmetic operations (add, subtract, multiply, and divide) are shown in Figure 5.4A. These operators are common to most programming languages. Some nonstandard, but useful operators include exponentiation (raise to a power, e.g., $2^3 = 2 * 2 * 2 = 8$), and integer divide (e.g., $9 \setminus 2 = 4$), which always returns an integer value. The mod function returns the modulus or remainder of an integer division (e.g., $15 \text{ mod } 4 = 3$, since $15 - 12 = 3$). These last two functions are useful when you need to know how many of some objects will fit into a fixed space. For example, if there are 50 possible lines on a page and you need to print a report with 185 lines, then $185 \setminus 50 = 3$ pages, and $185 \text{ Mod } 50$ leaves 35 lines on a fourth page.

Most languages support string variables, which are used to hold basic text data, such as names, addresses, or short messages. A string is a collection (or array) of characters. Sometimes you will need to perform computations on string variables. How can you perform computations on text data? The most common technique is to **concatenate** (or add) two strings together. For example, if `FirstName` is "George" and `LastName` is "Jones", then `FirstName & LastName` is "George-Jones". Notice that if you want a space to appear between the names, you have to add one: `FirstName & " " & LastName`.

Figure 5.5A lists some of the common string functions. You can learn more about the functions and their syntax from the Help system. Commonly used functions include the `Left`, `Right`, and `Mid`, which examine portions of the string. For

Figure 5.5A

Common string functions to add strings, extract portions, examine characters, convert case, compare two strings, and format numerical data into a string variable.

Concatenation (& or +) Left, Right, Mid, or SubStr Trim, LTrim, RTrim LCase, UCase InStr or IndexOf
"Frank" + "Rose" → "Frank Rose" Left("Jackson", 5) → "Jacks" Trim(" Maria ") → "Maria" Len("Ramanujan") → 9 "8764 Main".IndexOf(" ") → 5

Exp, Log	$x = \log_e(e^x)$
Atn, Cos, Sin, Tan	Trigonometric functions
Sqr or Sqrt	Square root
Abs	Absolute value: Abs(-35) → 35
Sgn	Signum: Sgn(-35) → -1
Int	Integer: Int(2.718) → 2
Rnd	Random number

Figure 5.6A

Standard mathematical functions. Even in business applications, you often need basic mathematical functions.

example, you might want to see only the first five characters on the left side of a string.

Standard Internal Functions

As you may recall from courses in mathematics, several common functions are used in a variety of situations. As shown in Figure 5.6A, these functions include the standard trigonometric and logarithmic functions, which can be useful in mapping and procedures involving measurements. You also will need a function to compute the square root and absolute value of numbers. The Int (integer) function is useful for dropping the fractional portion of a number. Most languages also provide a random number generator, which will randomly create numbers between 0 and 1. If you need another range of numbers, you can get them with a simple conversion. For example, to generate numbers between 40 and 90, use the following function: $y = 40 + (90 - 40) * \text{Rnd}$.

In a database environment, you will often need to evaluate and modify dates. It is also useful to have functions that provide the current date (Date) and time (Now). Two functions that are useful in business are the DateAdd and DateDiff functions. As illustrated in Figure 5.7A, the DateAdd function adds days to a given date to find some date in the future. The DateDiff function computes the difference between two dates. Usually, you will want to compute the number of days between various dates. However, the functions can often compute the number of months, weeks, years and so on.

Input and Output

Handling input and output were crucial topics in traditional programming. These topics are still important, but the DBMS now performs most data-handling rou-

Figure 5.7A

Date and time functions. Business problems often require computing the number of days between two dates or adding days to a date to determine when payments are due.

Date, Now, Time	Current date and time
DateAdd, DateDiff	Date arithmetic: DateDue = DateAdd("d", 30, Date())

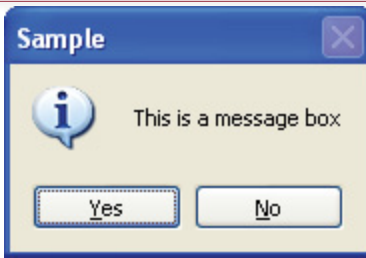


Figure 5.8A

Sample message box. The message box interrupts the user and displays a few limited choices. It often handles errors or problems.

times and the operating system or Web browser handles most of the user interface. Common forms and reports (Chapters 6 and 7) are used for most input and output tasks.

Remember that an important feature of a Windows interface is that users control the flow of data entry; that is, the designer provides a form, and users work at their own pace without interruption. Occasionally, you might choose to interrupt the user—either to provide information or to get a specific piece of data. One common reason is to display error messages. Two basic functions serve this purpose: `MsgBox` and `InputBox`. As shown in Figure 5.8A, a message box can contain buttons. The buttons are often used to indicate how the user wants to respond to some problem or error.

An `InputBox` is a special form that can be used to enter very small amounts of text or a single number. Neither the user nor the developer has much control over the form. In most cases you would be better off creating your own blank form. Then you can have more than one text box, and you can specify and control the buttons. The `InputBox` is usually for temporary use when development time is extremely limited.

Conditions

The ability to test and respond to conditions is one of the most common reasons for writing your own procedures. The basic conditional statement (if ...then ... else) is relatively easy to understand. The structure is shown in Figure 5.9A. A

Figure 5.9A

Conditions. Basic conditions are straightforward. Indenting conditions highlights the relationships.

```
If (Condition1) Then
    Statements for true
Else
    Statements for false
    If (Condition2) Then
        Statements for true
    End If
End If
```

```
response = 1, 2, 3, 4, 5
If (response = 1) Then
  ' Statements for 1
Else
  If (response = 2) Then
    ' Statements for 2
  Else
    If (response = 3) Then
      ' More If statements
    End If
  End If
End If
```

Figure 5.10A

Nested conditions to test for a user response. The code becomes harder to read as more conditions are added.

condition is evaluated to be true or false. If it is true, then one set of statements is executed; otherwise, the second set is performed.

Conditions can be complex, particularly when the condition contains several AND, and OR connectors. Some developers use a NOT statement to reverse the value of a condition. Be careful when writing conditions. Your goals are to make sure that the condition evaluates to the correct value and to make sure that other developers can understand the code.

You should always include parentheses to specify the order of evaluation and, for complex conditions, create sample data and test the conditions. Also, indent your code. Indenting is particularly important for **nested conditions**, in which the statements for one condition contain another conditional statement.

The Select Case statement is a special type of conditional statement. Many procedures will need to evaluate a set of related conditions. As a simple example, consider what happens if you use a message box with three buttons (Yes, No, and Cancel). You will have to test the user's choice for each option. Figure 5.10A shows how the code might look when you use nested conditions.

Figure 5.11A

The Select statement. The select statement tests the response variable against several conditions. If the response matches a case in the list, the corresponding code is executed.

```
Response = 1, 2, 3, 4, 5
Select Case response
  Case 1
    ' Statements for 1
  Case 2
    ' Statements for 2
  Case 3
    ' More Case statements
  Default
End Case
```

Figure 5.11A shows the same problem written with the Select Case statement. Note that this code is much easier to read. Now think about what will happen if you have 10 choices. The If-Then code gets much worse, but the Select Case code just adds new lines to the bottom of the list.

Loops

Iteration or **loops** are another common feature in procedures. Although you should use SQL statements (UPDATE, INSERT, etc.) as much as possible, sometimes you will need to loop through a table or query to examine each row individually.

Some of the basic loop formats are illustrated in Figure 5.12A. The For/Next loop is generally used only if you need a fixed number of iterations. The Do loop is more common. An important feature of loops is the ability to test the condition at the top or the bottom of the loop. Consider the example in which the condition says to execute the statements if ($x \leq 10$). What happens when the starting value of x is 15? If you test the condition at the top of the loop, then the statements in the loop will never be executed. On the other hand, if you test the condition at the bottom, then the statements in the loop will be executed exactly one time—before the condition is tested.

Just as with conditions, it is good programming practice to indent the statements of the loop. Indents help others to read your code and to understand the logic. If there are no problems within a loop, your eye can easily find the end of the loop.

Be careful with loops: if you make a mistake, the computer may execute the statements of your loop forever. (On most personal computers, Ctrl+Break will usually stop a runaway loop.) A common mistake occurs when you forget to change the conditional variable (x in the examples). In tracking through a data query, you might forget to get the next row of data, in which case your code will perform the same operations forever on one row of data. A good programming practice is to always write loops in four steps: (1) Write the initial condition, (2) Write the ending statement, (3) Write a statement to update the conditional variable, and (4) Write the interior code. The first three statements give you the structure. By writing and testing them first, you know that you will be using the correct data.

Figure 5.12A

Iteration. All versions of loops follow a common format: initialize a counter value, perform statements, increment the counter, and test the exit condition. You can test the condition at the start or end of the loop.

Do Until ($x > 10$) ' Statements $x = x + 1$ Loop	Do While ($x \leq 10$) ' Statements $x = x + 1$ Loop
Do ' Statements $x = x + 1$ Loop Until ($x > 10$)	For $x = 1$ to 10 ' Statements Next x

```

Main program
...
StatusMessage "Trying to connect."
...
StatusMessage "Verifying access."
...
End main program

Sub StatusMessage (Msg As String)
    ' Display Msg, location, color
End Sub

```

Figure 5.13A

Subroutine. The StatusMessage subroutine can be called from any location. When the subroutine is finished, it returns to the calling program.

Subroutines

An important concept in programming is the ability to break the program into smaller pieces as subroutines or functions. A **subroutine** is a portion of code that can be called from other routines. When the subroutine is finished, control returns to the program code that called it. The goal of using subroutines is to break the program into smaller pieces that are relatively easy to understand, test, and modify.

A subroutine is essentially a self-contained program that can be used by many other parts of the program. For example, you might create a subroutine that displays a status message on the screen. As illustrated in Figure 5.13A, you would write the basic routine once. Then anytime you need to display a status message, your program calls this routine. By passing the message to the subroutine, the actual message can change each time. The advantage of using the subroutine is that you have to write it only once. In addition, your status messages can be standardized because the subroutine specifies the location, style, and color. To change the format, you simply modify the few lines of code in the one subroutine. Without the subroutine, you would have to find and modify code in every location that displayed a status message.

A data variable that is passed to a function or a subroutine is known as a **parameter**. There are two basic ways to pass a parameter: by reference and by value. The default method used by Microsoft Access is pass-by-reference. In this case the variable in the subroutine is essentially the same variable as in the original program. Any changes made to the data in the subroutine will automatically be returned to the calling program. For example, consider the two examples in Figure 5.14A. Changes to the variable *j2* in the subroutine will automatically be passed back to the calling program. However, when only the value is passed, a copy is made in the subroutine. Changes made to the data in the subroutine will not be returned to the calling program. Unless you are absolutely certain that you want to alter the original value in the calling program, you should always pass variables by value. Subroutines that use pass-by-reference can cause errors that are difficult to find in programs. Some other programmer might not realize that your subroutine changed the value of a parameter.

<pre> Main: j = 3 DoSum(j) ' j is now equal to 8 ... Sub DoSum(By Ref j2 As Integer) j2 = 8 End Sub </pre>	<p><u>By Reference</u> Changes to data in the subroutine are passed back to the calling program.</p>
<pre> Main: j = 3 DoSum(j) ' j is still equal to 3 ... Sub DoSum(By Val j2 As Integer) J2 = 8 End Sub </pre>	<p><u>By Value</u> Creates a copy of the variable, so changes are not returned.</p>

Figure 5.14A

Two methods to pass data to a subroutine. Pass parameters by value as much as possible to avoid unwanted changes to data.

Most languages also enable you to create new functions. There is a slight technical difference between functions and subroutines. Although subroutines and functions can receive or return data through pass-by-reference parameters, a function can return a result or a single value directly to the calling program. For instance, your main program might have a statement such as $v1 = \text{Min}(x, y)$. The function would choose the smaller of the two values and return it to the main program, where it is assigned to the variable $v1$.

Summary

The only way to learn how to program is to write your own programs. Reading books, syntax documentation, and studying code written by others will help, but the only way to become a programmer is through experience.

As you write programs, remember that you (or someone else) might have to modify your code later. Choose descriptive variable names. Document your statements with comments that explain tricky sections and outline the purpose of each section of code. Write in small sections and subroutines. Test each section, and keep the test data and results in the documentation. Keep revision notes so that you know when each section was changed and why you changed it.