

CHAPTER 8

SQL FUNCTIONS AND PROCEDURES

LEARNING OBJECTIVES

Objectives

- Understand how to use functions in queries
- Use the UPPER and LOWER functions with character data
- Use the ROUND and FLOOR functions with numeric data
- Add a specific number of months or days to a date
- Calculate the number of days between two dates
- Use concatenation in a query
- Embed SQL commands in PL/SQL and T-SQL procedures
- Retrieve single rows using embedded SQL
- Update a table using embedded INSERT, UPDATE, and DELETE commands
- Use cursors to retrieve multiple rows in embedded SQL
- Manage errors in procedures containing embedded SQL commands
- Use SQL in a language that does not support embedded SQL commands
- Use triggers

INTRODUCTION

You already have used functions that apply to groups (such as SUM and AVG). In this chapter, you will learn to use functions that apply to values in individual rows. Specifically, you will see how to use functions with characters or text, numbers, and dates. You will learn how to concatenate values in a query. You will embed SQL commands in PL/SQL and T-SQL procedures to retrieve rows and update data. You will

examine the different ways to manage errors in procedures. Finally, you will learn how to create and use cursors and triggers.

USING SQL IN A PROGRAMMING ENVIRONMENT

234

SQL is a powerful **nonprocedural language** in which you communicate tasks to the computer using simple commands. As in other nonprocedural languages, you can accomplish many tasks using a single command. Although SQL and other nonprocedural languages are well-equipped to store and query data, sometimes you might need to complete tasks that are beyond the capabilities of SQL. In such cases, you need to use a procedural language.

A **procedural language** is one in which you must give the computer the step-by-step process for accomplishing a task. **PL/SQL**, which was developed by Oracle as an extension of SQL, is an example of a procedural language. This chapter uses PL/SQL to illustrate how to use SQL in a programming environment by **embedding** SQL commands in another language. The examples in this chapter illustrate how to use embedded SQL commands to retrieve a single row, insert new rows, update and delete existing rows, and retrieve multiple rows. In the process, you will create stored procedures that are saved and are available for use at any time.

T-SQL, which stands for **Transact-SQL**, is another extension of SQL. T-SQL is the procedural language that SQL Server uses. You can perform tasks, such as retrieving a single row, inserting new rows, and retrieving multiple rows, using T-SQL in SQL Server. Although the language syntax is slightly different in T-SQL when compared to PL/SQL, the functionality and the results are the same.

You cannot embed SQL commands in Access programs the way you can in PL/SQL and T-SQL. There are ways to use the commands, however, as you'll learn later in this chapter.

NOTE

This chapter assumes that you have some programming background and does not cover programming basics. To understand the first part of this chapter, you should be familiar with variables, declaring variables, and creating procedural code, including IF statements and loops. To understand the Access section at the end of the chapter, you should be familiar with Function and Sub procedures, and the process for sequentially accessing all records in a recordset, such as using a loop to process all the records in a table.

ACCESS USER NOTE

If you are using Access, you will not be able to complete the material in this chapter that deals with PL/SQL and T-SQL procedures. Be sure to read this information so you will understand these important concepts. You will, however, be able to complete the steps in the "Using SQL in Microsoft Access" section.

USING FUNCTIONS

You already have used aggregate functions to perform calculations based on groups of records. For example, SUM(BALANCE) calculates the sum of the balances on all records that satisfy the condition in the WHERE clause. When you use a GROUP BY clause, the DBMS will calculate the sum for each record in a group.

SQL also includes functions that affect single records. Some functions affect character data and others let you manipulate numeric data. The supported SQL functions vary between SQL implementations. This section will illustrate some common functions. For additional information about the functions your SQL implementation supports, consult the program's documentation.

Character Functions

SQL includes several functions that affect character data. Example 1 illustrates the use of the UPPER function.

EXAMPLE 1

List the rep number and last name for each sales rep. Display the last name in uppercase letters.

The **UPPER** function displays a value in uppercase letters; for example, the function UPPER(LAST_NAME) displays the last name Kaiser as KAISER. (Note that the UPPER function simply displays the last name in uppercase letters; it does not change the last name stored in the table to uppercase letters.) The item in parentheses (LAST_NAME) is called the **argument** for the function. The value produced by the function is the result of displaying all lowercase letters in the value stored in the LAST_NAME column in uppercase letters. The query and its results are shown in Figure 8-1.

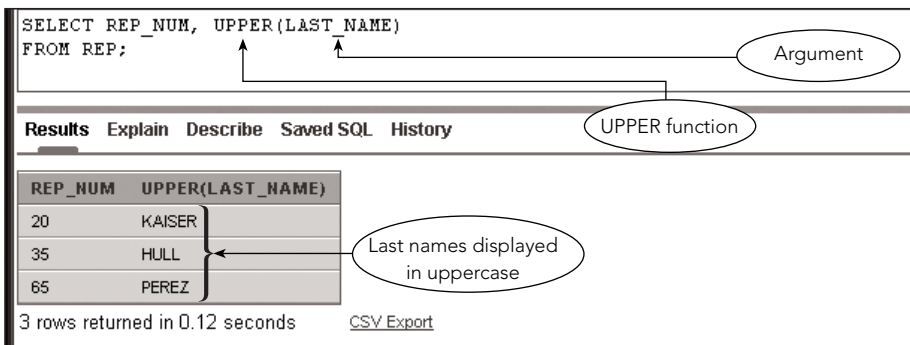


FIGURE 8-1 Using the UPPER function to display character data in uppercase letters

You can use functions in WHERE clauses as well. For example, the condition UPPER(LAST_NAME) = 'KAISER' would be true for names like Kaiser, KAISER, and KaIsER,

because the result of applying the UPPER function to any of these values would result in the value KAISER.

To display a value in lowercase letters, you can use the **LOWER** function. SQL Server supports both the UPPER and LOWER function.

ACCESS USER NOTE

In Access, the UCASE() function displays a value in uppercase letters and the LCASE() function displays a value in lowercase letters. For example, if the value stored in the LAST_NAME column is Kaiser, UCASE(LAST_NAME) would result in the value KAISER and LCASE(LAST_NAME) would result in the value kaiser.

236

Number Functions

SQL also includes functions that affect numeric data. The **ROUND** function, which rounds values to a specified number of decimal places, is illustrated in Example 2.

EXAMPLE 2

List the part number and price for all parts. Round the price to the nearest whole dollar amount.

A function can have more than one argument. The ROUND function, which rounds a numeric value to a desired number of decimal places, has two arguments. The first argument is the value to be rounded; the second argument indicates the number of decimal places to which to round the result. For example, ROUND(PRICE,0) will round the values in the PRICE column to zero decimal places (a whole number). If a price is 24.95, the result will be 25. If the price is 24.25, on the other hand, the result will be 24. Figure 8-2 shows the query and results to round values in the PRICE column to zero decimal places. The computed column ROUND(PRICE,0) is named **ROUNDED_PRICE**.

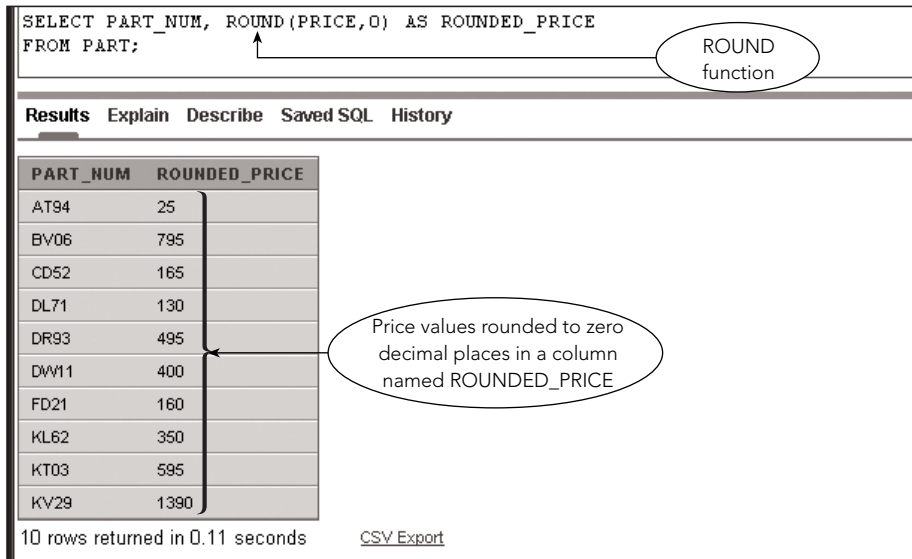


FIGURE 8-2 Using the ROUND function to round numeric values

Rather than rounding (using the ROUND function), you might need to truncate (remove) everything to the right of the decimal point. To do so, use the **FLOOR** function, which has only one argument. If a price is 24.95, for example, ROUND(PRICE,0) would result in 25, whereas FLOOR(PRICE) would result in 24. SQL Server supports both the ROUND and the FLOOR functions. Microsoft Access supports only the ROUND function.

Working with Dates

SQL uses functions and calculations for manipulating dates. To add a specific number of months to a date, you can use the **ADD_MONTHS** function as illustrated in Example 3.

EXAMPLE 3

For each order, list the order number and the date that is two months after the order date. Name this date FUTURE_DATE.

The ADD_MONTHS function has two arguments. The first argument is the date to which you want to add a specific number of months, and the second argument is the number of months. To add two months to the order date, for example, the expression is ADD_MONTHS(ORDER_DATE,2) as illustrated in Figure 8-3.

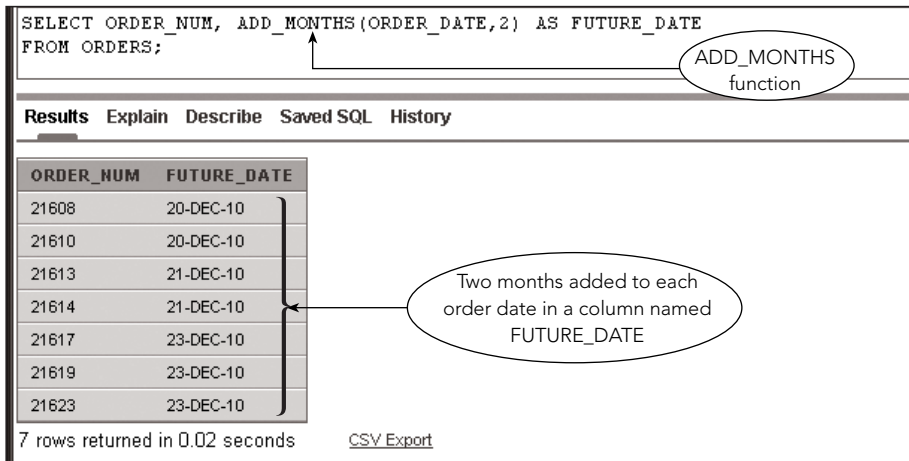


FIGURE 8-3 Using the ADD_MONTHS function to add months to a date

ACCESS USER NOTE

To add a number of months to a date in Access, use the DATEADD() function, which has three arguments. The first argument includes the interval of time to be added; the letter “m” indicates that months will be added. The second argument includes the number of intervals to be added. The third argument includes the date to be manipulated. For example, to add two months to the dates stored in the ORDER_DATE column, the appropriate function would be DATEADD(“m”, 2, ORDER_DATE).

SQL SERVER USER NOTE

To add a number of months to a date in SQL Server, use the DATEADD() function, which has three arguments. The first argument includes the interval of time to be added; the letter “m” indicates that months will be added. The second argument includes the number of intervals to be added. The third argument includes the date to be manipulated. For example, to add two months to the dates stored in the ORDER_DATE column, the appropriate function would be DATEADD(“m”, 2, ORDER_DATE).

EXAMPLE 4

For each order, list the order number and the date that is seven days after the order date. Name this date FUTURE_DATE.

To add a specific number of days to a date, you do not need a function. You can add the number of days to the order date as illustrated in Figure 8-4. (You can also subtract dates in the same way.) This method works in Oracle, Access, and SQL Server.

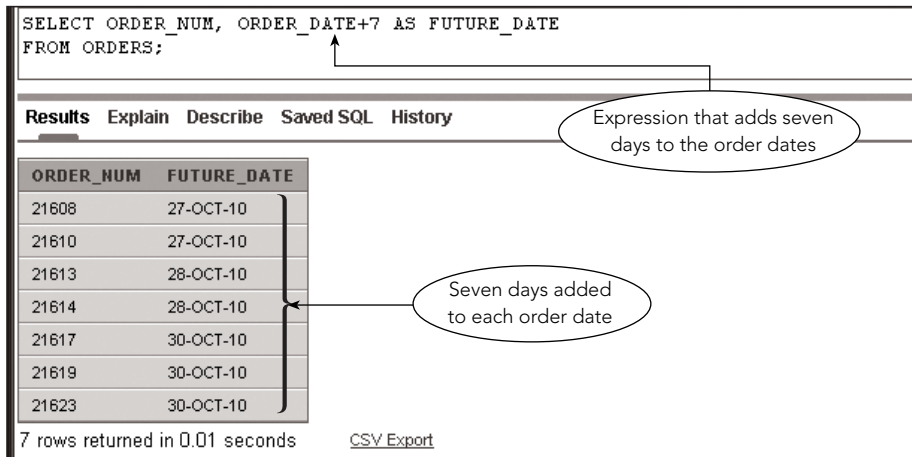


FIGURE 8-4 Adding days to dates

EXAMPLE 5

For each order, list the order number, today's date, the order date, and the number of days between the order date and today's date. Name today's date `TODAYS_DATE` and name the number of days between the order date and today's date `DAYS_PAST`.

You can use the **SYSDATE** function to obtain today's date, as shown in Figure 8-5. The command in the figure uses `SYSDATE` to display today's date and also uses `SYSDATE` in a computation to determine the number of days between the order date and today's date. The values for `DAYS_PAST` include decimal places. You could remove these decimal places by using the `ROUND` or `FLOOR` functions, if desired.

```

SELECT ORDER_NUM, SYSDATE AS TODAYS_DATE, ORDER_DATE,
SYSDATE - ORDER_DATE AS DAYS_PAST
FROM ORDERS;

```

Number of days between today's date and the order date

Expression to calculate the number of days between today's date and the order date

ORDER_NUM	TODAYS_DATE	ORDER_DATE	DAYS_PAST
21608	02-NOV-10	20-OCT-10	13.4922916666666666666666666666667
21610	02-NOV-10	20-OCT-10	13.4922916666666666666666666666667
21613	02-NOV-10	21-OCT-10	12.4922916666666666666666666666667
21614	02-NOV-10	21-OCT-10	12.4922916666666666666666666666667
21617	02-NOV-10	23-OCT-10	10.4922916666666666666666666666667
21619	02-NOV-10	23-OCT-10	10.4922916666666666666666666666667
21623	02-NOV-10	23-OCT-10	10.4922916666666666666666666666667

7 rows returned in 0.00 seconds [CSV Export](#)

FIGURE 8-5 Calculating the number of days between two dates

ACCESS USER NOTE

In Access, use the DATE() function to obtain today's date, rather than SYSDATE. The DATE() function has no arguments, so you would write DATE() in place of SYSDATE.

SQL SERVER USER NOTE

In SQL Server, use the GETDATE() function to obtain today's date, rather than SYSDATE. The GETDATE() function has no arguments, so you would write GETDATE() in place of SYSDATE.

CONCATENATING COLUMNS

Sometimes you need to **concatenate**, or combine, two or more character columns into a single expression when displaying them in a query; the process is called **concatenation**. To concatenate columns, you type two vertical lines (||) between the column names, as illustrated in Example 6.

EXAMPLE 6

List the number and name of each sales rep. Concatenate the FIRST_NAME and LAST_NAME columns into a single value, with a space separating the first and last names.

To concatenate the FIRST_NAME and LAST_NAME columns, the expression is FIRST_NAME||LAST_NAME. When the first name doesn't include sufficient characters to fill the width of the column (as determined by the number of characters specified in the CREATE TABLE command), SQL inserts extra spaces. For example, when the

FIRST_NAME column is 12 characters wide, the first name is Mary, and the last name is Johnson, the expression FIRST_NAME||LAST_NAME appears as Mary, followed by eight spaces, and then Johnson. To remove the extra spaces following the first name value, you use the **RTRIM** (right trim) function. When you apply this function to the value in a column, SQL displays the original value and removes any spaces inserted at the end of the value. Figure 8-6 shows the query and output with the extra spaces removed. For sales rep 20, for example, this command trims the first name to “Valerie,” concatenates it with a single space, and then concatenates the last name “Kaiser.”

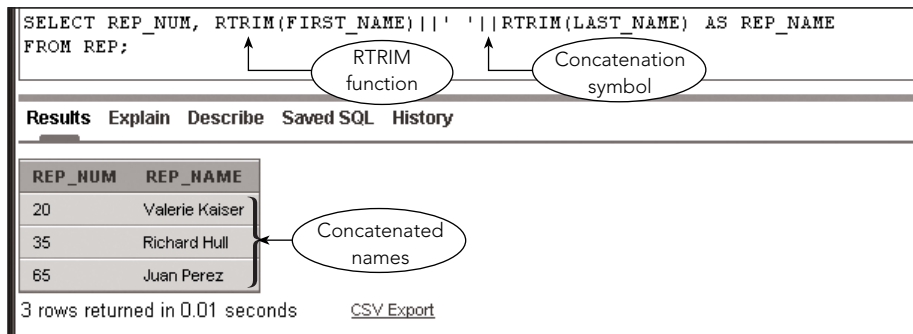


FIGURE 8-6 Concatenating two columns and using the RTRIM function

Q & A

Question: Why is it necessary to insert a single space character in single quotation marks in the query?

Answer: Without the space character, there would be no space between the first and last names. The name of sales rep 20, for example, would be displayed as “ValerieKaiser.”

ACCESS USER NOTE

In Access, use the & symbol to concatenate columns. It is not necessary to trim the columns because Access will trim them automatically. The corresponding query in Access is:

```
SELECT REP_NUM, FIRST_NAME&' '&LAST_NAME
FROM REP;
```

SQL SERVER USER NOTE

In SQL Server, use the + symbol to concatenate columns. The corresponding query in SQL Server is:

```
SELECT REP_NUM, RTRIM(FIRST_NAME)+' '+RTRIM(LAST_NAME)
FROM REP;
```

STORED PROCEDURES

In a **client/server system**, the database is stored on a computer called the **server** and users access the database through clients. A **client** is a computer that is connected to a network and has access through the server to the database. Every time a user executes a query, the DBMS must determine the best way to process the query and provide the results. For example, the DBMS must determine which indexes are available and whether it can use those indexes to make the processing of the query more efficient.

When you anticipate running a particular query often, you can improve overall performance by saving the query in a file called a **stored procedure**. The stored procedure is placed on the server. The DBMS compiles the stored procedure (translating it into machine code) and creates an execution plan, which is the most efficient way of obtaining the results. From that point on, users execute the compiled, optimized code in the stored procedure.

Another reason for saving a query as a stored procedure, even when you are not working in a client/server system, is convenience. Rather than retyping the entire query each time you need it, you can use the stored procedure. For example, suppose you frequently execute a query that selects a sales rep with a given number and then displays the concatenation of the first name and last name of the sales rep. Instead of typing the query each time you want to display a sales rep's name, you can store the query in a stored procedure. You would then only need to run the stored procedure when you want to display a sales rep's name.

ACCESS USER NOTE

Although Access does not support stored procedures, you can achieve some of the same convenience by creating a parameter query that prompts the user for the arguments you would otherwise use in a stored procedure.

In Oracle, you create stored procedures using a language called PL/SQL. You create and save the procedures as script files.

Retrieving a Single Row and Column

Example 7 illustrates using a stored procedure to retrieve a single row and column from a table.

EXAMPLE 7

Write a PL/SQL procedure that takes a rep number as input and displays the corresponding rep name.

Figure 8-7 shows a procedure to find the name of the representative whose number is stored in the `I_REP_NUM` argument. Because the restriction involves the primary key, the query will produce only one row of output. (You will see how to handle queries whose results can contain multiple rows later in this chapter.) The command shown in Figure 8-7 is stored in a script file and is displayed in the Script Editor. To create the procedure, you would run the script file. Assuming that the script file does not contain any errors, Oracle would then create the procedure and it would be available for use.

243

```
Home > SQL > SQL Scripts > Script Editor
Script Name: DISPLAY_REP_NAME.sql
[Cancel] [Download] [Delete]
[Undo] [Redo] [Find]
1 CREATE OR REPLACE PROCEDURE DISP_REP_NAME (I_REP_NUM IN REP.REP_NUM%TYPE) AS
2 I_LAST_NAME REP.LAST_NAME%TYPE;
3 I_FIRST_NAME REP.FIRST_NAME%TYPE;
4
5 BEGIN
6 SELECT LAST_NAME, FIRST_NAME
7 INTO I_LAST_NAME, I_FIRST_NAME
8 FROM REP
9 WHERE REP_NUM = I_REP_NUM;
10
11 DBMS_OUTPUT.PUT_LINE(RTRIM(I_FIRST_NAME) || ' ' || RTRIM(I_LAST_NAME));
12
13 END;
14 /
```

FIGURE 8-7 Procedure to find a rep's name given the rep's number

NOTE

PL/SQL commands, like SQL commands, are free-format and can include blank lines to separate important sections of the procedure and spaces on the lines to make the commands more readable.

The `CREATE PROCEDURE` command in the stored procedure causes Oracle to create a procedure named `DISP_REP_NAME`. By including the optional `OR REPLACE` clause in the `CREATE PROCEDURE` command, you can use the command to modify an existing procedure. If you omit the `OR REPLACE` clause, you would need to drop the procedure and then re-create it in order to change the procedure later.

The first line of the command contains a single argument, `I_REP_NUM`. The word `IN` following the single argument name indicates that `I_REP_NUM` will be used for input. That

is, the user must enter a value for I_REP_NUM to use the procedure. Other possibilities are OUT, which indicates that the procedure will set a value for the argument, and INOUT, which indicates that the user will enter a value that the procedure can later change.

Variable names in PL/SQL must start with a letter and can contain letters, dollar signs, underscores, and number signs, but cannot exceed 30 characters. When declaring variables, you must assign the variable a data type, just as you do in the SQL CREATE TABLE command. You can ensure that a variable has the same data type as a particular column in a table by using the %TYPE attribute. To do so, you include the name of the table, followed by a period and the name of the column, and then %TYPE. When you use %TYPE, you do not enter a data type because the variable is automatically assigned the same type as the corresponding column. In the first line of the script file shown in Figure 8-7, assigning the variable I_REP_NUM the same type as the REP_NUM column in the REP table is written as REP.REP_NUM%TYPE.

The first line of the CREATE PROCEDURE command ends with the word AS and is followed by the commands in the procedure. The commands on lines 2 and 3 declare the local variables the procedure requires. In Figure 8-7, lines 2 and 3 create two variables named I_LAST_NAME and I_FIRST_NAME. Both variables are assigned data types using %TYPE.

The **procedural code**, which contains the commands that specify the procedure's function, appears between the BEGIN and END commands. In Figure 8-7, the procedural code begins with the SQL command to select the last name and first name of the sales rep whose number is stored in I_REP_NUM. The SQL command uses the INTO clause to place the results in the I_LAST_NAME and I_FIRST_NAME variables. The next command uses the DBMS_OUTPUT.PUT_LINE procedure to display the concatenation of the trimmed I_FIRST_NAME and I_LAST_NAME variables. Notice that a semicolon ends each variable declaration, command, and the word END. The slash (/) at the end of the procedure appears on its own line. In some Oracle environments, the slash is optional. A good practice is to include the slash, even when it's not necessary, so your procedure will always work correctly.

NOTE

DBMS_OUTPUT is a package that contains multiple procedures, including PUT_LINE. The SQL Commands page automatically displays the output produced by DBMS_OUTPUT.

To **call** (or use) the procedure from the SQL Commands page, type the word BEGIN, followed by the name of the procedure including the desired value for the argument in parentheses, followed by the word END, a semicolon, and a slash on a separate line. To use the DISP_REP_NAME procedure to find the name of sales rep 20, for example, type the command shown in Figure 8-8.

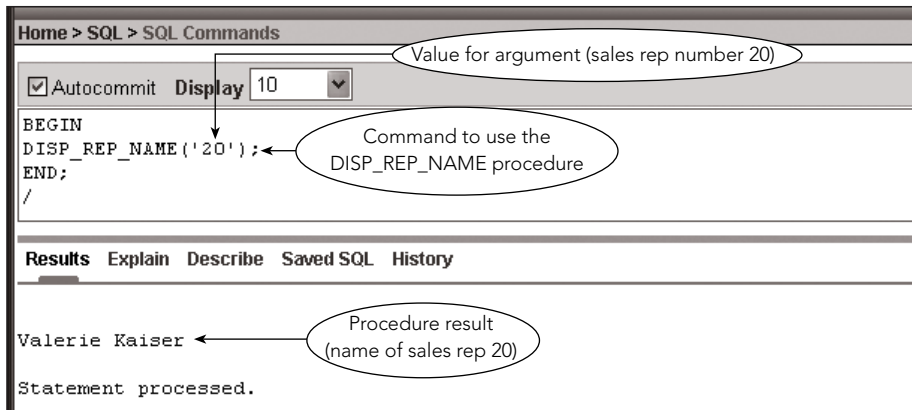


FIGURE 8-8 Using the DISP_REP_NAME procedure within an SQL command

ERROR HANDLING

Procedures must be able to handle conditions that can arise when accessing the database. For example, the user enters a rep number and the DISP_REP_NAME procedure displays the corresponding rep’s name. What happens when the user enters an invalid rep number? This situation results in the error message shown in Figure 8-9 because Oracle will not find any last name to display.

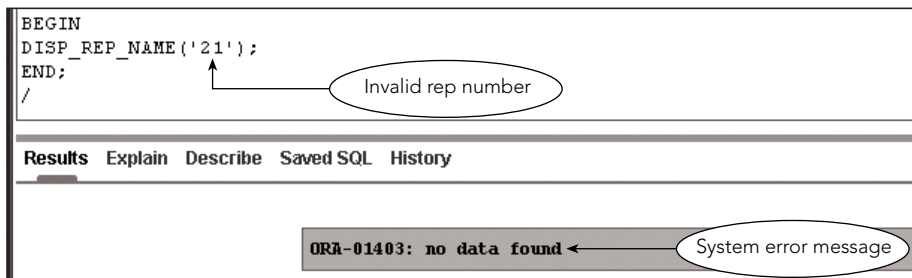


FIGURE 8-9 System error that occurs when a user enters an invalid rep number

You can include the EXCEPTION clause shown in Figure 8-10 to handle processing an invalid rep number. When a user enters a rep number that does not match any rep number in the REP table, the NO_DATA_FOUND condition on line 13 will be true. When the NO_DATA_FOUND condition is true, the procedure displays the “No rep with this number:” message followed by the invalid rep number.

```

1 CREATE OR REPLACE PROCEDURE DISP_REP_NAME (I_REP_NUM IN REP.REP_NUM%TYPE) AS
2 I_LAST_NAME  REP.LAST_NAME%TYPE;
3 I_FIRST_NAME  REP.FIRST_NAME%TYPE;
4
5 BEGIN
6 SELECT LAST_NAME, FIRST_NAME
7 INTO I_LAST_NAME, I_FIRST_NAME
8 FROM REP
9 WHERE REP_NUM = I_REP_NUM;
10
11 DBMS_OUTPUT.PUT_LINE (RTRIM(I_FIRST_NAME) || ' ' || RTRIM(I_LAST_NAME));
12 EXCEPTION
13 WHEN NO_DATA_FOUND THEN
14 DBMS_OUTPUT.PUT_LINE ('No rep with this number: ' || I_REP_NUM);
15
16 END;
17 /

```

EXCEPTION clause

Tests for NO_DATA_FOUND condition

Action to take when no data is found

FIGURE 8-10 PL/SQL procedure with error handling

When you use this version of the procedure and enter an invalid rep number, you will see the error message from the procedure (Figure 8-11) instead of the system error message (Figure 8-9).

```

BEGIN
DISP_REP_NAME('21');
END;
/

```

Invalid rep number

Results Explain Describe Saved SQL History

Error message from the procedure

No rep with this number: 21

Statement processed.

FIGURE 8-11 Error message that occurs when a user enters an invalid rep number

The DISP_REP_NAME procedure must handle an error that results when a user enters an invalid rep number. There are other types of errors that procedures must handle, depending on the processing required. For example, a user might enter a commission rate in a procedure to find the name of the sales rep who has that commission rate. When the user enters the rate 0.05, the procedure will display the TOO_MANY_ROWS error because Valerie Kaiser and Juan Perez both have this same commission rate—the procedure finds two rows instead of one. You can manage this error by writing a WHEN clause that contains a TOO_MANY_ROWS condition, following the EXCEPTION clause in the procedure. You can write both WHEN clauses in the same procedure or in separate procedures. When adding both WHEN clauses to the same procedure, however, the EXCEPTION clause appears only once.

USING UPDATE PROCEDURES

In Chapter 6, you learned how to use SQL commands to update data. You can use the same commands within procedures. A procedure that updates data is called an **update procedure**.

Changing Data with a Procedure

You can use an update procedure to change a row in a table, as illustrated in Example 8.

EXAMPLE 8

Change the name of the customer whose number is stored in I_CUSTOMER_NUM to the value currently stored in I_CUSTOMER_NAME.

247

This procedure is similar to the procedures used in previous examples with two main differences: it uses an UPDATE command instead of a SELECT command, and there are two arguments, I_CUSTOMER_NUM and I_CUSTOMER_NAME. The I_CUSTOMER_NUM argument stores the customer number to be updated and the I_CUSTOMER_NAME argument stores the new value for the customer name. The procedure appears in Figure 8-12.

```
1 CREATE OR REPLACE PROCEDURE CHG_CUST_NAME (I_CUSTOMER_NUM IN CUSTOMER.CUSTOMER_NUM%TYPE,  
2 I_CUSTOMER_NAME IN CUSTOMER.CUSTOMER_NAME%TYPE) AS  
3  
4 BEGIN  
5 UPDATE CUSTOMER  
6 SET CUSTOMER_NAME = I_CUSTOMER_NAME  
7 WHERE CUSTOMER_NUM = I_CUSTOMER_NUM;  
8  
9 END;  
10 /
```

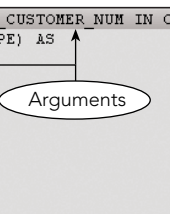



FIGURE 8-12 Using a procedure to update a row

When you run this procedure, you will need to furnish values for two arguments. Figure 8-13 uses this procedure to change the name of customer 725 to Deerfield's.

```
BEGIN  
CHG_CUST_NAME('725','Deerfield's');  
END;  
/
```



Results	Explain	Describe	Saved SQL	History
Statement processed.				

FIGURE 8-13 Using a procedure to update the name of customer 725

Deleting Data with a Procedure

Just as you would expect, if you can use an update procedure to change a row in a table, you can also use one to delete a row from a table, as illustrated in Example 9.

EXAMPLE 9

Delete the order whose number is stored in `I_ORDER_NUM` from the `ORDERS` table, and also delete each order line for the order whose order number is currently stored in the variable from the `ORDER_LINE` table.

248

If you attempt to delete the order in the `ORDERS` table first, referential integrity will prevent the deletion because matching rows would still exist in the `ORDER_LINE` table, so it is a good idea to delete the orders from the `ORDER_LINE` table first. The procedure to delete an order and its related order lines appears in Figure 8-14. This procedure contains two `DELETE` commands. The first command deletes all order lines in the `ORDER_LINE` table on which the order number matches the value stored in the `I_ORDER_NUM` argument. The second command deletes the order in the `ORDERS` table whose order number matches the value stored in the `I_ORDER_NUM` argument.

```
1 CREATE OR REPLACE PROCEDURE DEL_ORDER (I_ORDER_NUM IN ORDERS.ORDER_NUM%TYPE) AS
2
3 BEGIN
4 DELETE
5 FROM ORDER_LINE
6 WHERE ORDER_NUM = I_ORDER_NUM;
7
8 DELETE
9 FROM ORDERS
10 WHERE ORDER_NUM = I_ORDER_NUM;
11
12 END;
13 /
```

Argument that stores the order number to delete

Command to delete all rows in the `ORDER_LINE` table that matches the entered order number

Command to delete the row in the `ORDERS` table that matches the entered order number

FIGURE 8-14 Procedure to delete a row and related rows from multiple tables

Figure 8-15 shows the use of this procedure to delete order number 21610. Even though there are two `DELETE` commands in the procedure, the user enters the order number only once.


```
BEGIN
DEL_ORDER('21610');
END;
/
```

Results Explain Describe Saved SQL History

Statement processed.

FIGURE 8-15 Using the procedure to delete an order

SELECTING MULTIPLE ROWS WITH A PROCEDURE

The procedures you have seen so far include commands that retrieve individual rows. You can use an UPDATE or a DELETE command in PL/SQL to update or delete multiple rows. The commands are executed and the updates or deletions occur. Then the procedure can move on to the next task.

What happens when a SELECT command in a procedure retrieves multiple rows? For example, suppose the SELECT command retrieves the number and name of each customer represented by the sales rep whose number is stored in I_REP_NUM. There is a problem—PL/SQL can process only one record at a time, but this SQL command retrieves more than one row. Whose number and name is placed in I_CUSTOMER_NUM and I_CUSTOMER_NAME when the command retrieves more than one customer row? Should you make I_CUSTOMER_NUM and I_CUSTOMER_NAME arrays capable of holding multiple rows and, if so, what should be the size of these arrays? Fortunately, you can solve this problem by using a cursor.

Using a Cursor

A **cursor** is a pointer to a row in the collection of rows retrieved by an SQL command. (This is *not* the same cursor that you see on your computer screen.) The cursor advances one row at a time to provide sequential, one-record-at-a-time access to the retrieved rows so PL/SQL can process the rows. By using a cursor, PL/SQL can process the set of retrieved rows as though they were records in a sequential file.

To use a cursor, you must first declare it, as illustrated in Example 10.

EXAMPLE 10

Retrieve and list the number and name of each customer represented by the sales rep whose number is stored in the variable I_REP_NUM.

The first step in using a cursor is to declare the cursor and describe the associated query in the declaration section of the procedure. In this example, assuming the cursor is named CUSTGROUP, the command to declare the cursor is:

```
CURSOR CUSTGROUP IS
SELECT CUSTOMER_NUM, CUSTOMER_NAME
FROM CUSTOMER
WHERE REP_NUM = I_REP_NUM;
```

This command does *not* cause the query to be executed at this time; it only declares a cursor named CUSTGROUP and associates the cursor with the indicated query. Using a cursor in a procedure involves three commands: OPEN, FETCH, and CLOSE. The **OPEN** command opens the cursor and causes the query to be executed, making the results available to the procedure. Executing a **FETCH** command advances the cursor to the next row in the set of rows retrieved by the query and places the contents of the row in the indicated variables. Finally, the **CLOSE** command closes a cursor and deactivates it. Data retrieved by the execution of the query is no longer available. The cursor could be opened again later and processing could begin again.

The OPEN, FETCH, and CLOSE commands used in processing a cursor are analogous to the OPEN, READ, and CLOSE commands used in processing a sequential file.

Opening a Cursor

Prior to opening the cursor, there are no rows available to be fetched. In Figure 8-16, this is indicated by the absence of data in the CUSTGROUP portion of the figure. The right side of the figure illustrates the variables into which the data will be placed (I_CUSTOMER_NUM and I_CUSTOMER_NAME) and the value CUSTGROUP%NOTFOUND. Once the cursor has been opened and all the records have been fetched, the CUSTGROUP%NOTFOUND value is set to TRUE. Procedures using the cursor can use this value to indicate when the fetching of rows is complete.

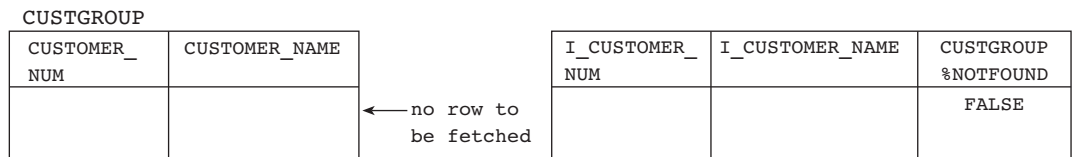


FIGURE 8-16 Before OPEN

The OPEN command is written as follows:

```
OPEN CUSTGROUP;
```

Figure 8-17 shows the result of opening the CUSTGROUP cursor. In the figure, assume that I_REP_NUM is set to 20 before the OPEN command is executed; there are now three rows available to be fetched. No rows have yet been fetched, as indicated by the absence of values in I_CUSTOMER_NUM and I_CUSTOMER_NAME. CUSTGROUP%NOTFOUND is still FALSE. The cursor is positioned at the first row; that is, the next FETCH command causes the contents of the first row to be placed in the indicated variables.

CUSTGROUP			I_CUSTOMER_	I_CUSTOMER_	CUSTGROUP
CUSTOMER_	CUSTOMER_		NUM	NAME	%NOTFOUND
148	Al's Appliance and Sport	← next row to be fetched			FALSE
524	Kline's				
842	All Season				

FIGURE 8-17 After OPEN, but before first FETCH

Fetching Rows from a Cursor

To fetch (get) the next row from a cursor, use the FETCH command. The FETCH command is written as follows:

```
FETCH CUSTGROUP INTO I_CUSTOMER_NUM, I_CUSTOMER_NAME;
```

Note that the INTO clause is associated with the FETCH command itself and not with the query used in the cursor definition. The execution of this query could produce multiple rows. The execution of the FETCH command produces only a single row, so it is appropriate that the FETCH command causes data to be placed in the indicated variables.

Figure 8-18 through Figure 8-21 show the result of four FETCH commands. The first three fetches are successful. In each case, the data from the appropriate row in the cursor is placed in the indicated variables and CUSTGROUP%NOTFOUND is still FALSE. The fourth FETCH command is different, however, because there is no more data to fetch. In this case, the contents of the variables are left untouched and CUSTGROUP%NOTFOUND is set to TRUE.

CUSTGROUP			I_CUSTOMER_	I_CUSTOMER_	CUSTGROUP
CUSTOMER_	CUSTOMER_		NUM	NAME	%NOTFOUND
148	Al's Appliance and Sport		148	Al's Appliance and Sport	FALSE
524	Kline's	← next row to be fetched			
842	All Season				

FIGURE 8-18 After first FETCH

CUSTGROUP			I_CUSTOMER_	I_CUSTOMER_	CUSTGROUP
CUSTOMER_	CUSTOMER_		NUM	NAME	%NOTFOUND
148	Al's Appliance and Sport		524	Kline's	FALSE
524	Kline's				
842	All Season	← next row to be fetched			

FIGURE 8-19 After second FETCH

CUSTGROUP	
CUSTOMER_ NUM	CUSTOMER_ NAME
148	Al's Appliance and Sport
524	Kline's
842	All Season

← next row to be fetched

I_CUSTOMER_ NUM	I_CUSTOMER_ NAME	CUSTGROUP %NOTFOUND
842	All Season	FALSE

FIGURE 8-20 After third FETCH

CUSTGROUP	
CUSTOMER_ NUM	CUSTOMER_ NAME
148	Al's Appliance and Sport
524	Kline's
842	All Season

← no more rows to be fetched

I_CUSTOMER_ NUM	I_CUSTOMER_ NAME	CUSTGROUP %NOTFOUND
842	All Season	TRUE

FIGURE 8-21 After attempting a fourth FETCH (CUSTGROUP%NOTFOUND is TRUE)

Closing a Cursor

The CLOSE command is written as follows:

```
CLOSE CUSTGROUP;
```

Figure 8-22 shows the result of closing the CUSTGROUP cursor. The data is no longer available.

CUSTGROUP	
CUSTOMER_ NUM	CUSTOMER_ NAME

← no rows to be fetched

FIGURE 8-22 After CLOSE

Writing a Complete Procedure Using a Cursor

Figure 8-23 shows a complete procedure using a cursor. The declaration portion contains the CUSTGROUP cursor definition. The procedural portion begins with the command to open the CUSTGROUP cursor. The statements between the LOOP and END LOOP commands create a loop that begins by fetching the next row from the cursor and placing the

results in I_CUSTOMER_NUM and I_CUSTOMER_NAME. The EXIT command tests the condition CUSTGROUP%NOTFOUND. If the condition is true, the loop is terminated. If the condition is not true, the DBMS_OUTPUT.PUT_LINE commands display the contents of I_CUSTOMER_NUM and I_CUSTOMER_NAME.

```

1 CREATE OR REPLACE PROCEDURE DISP_REP_CUST(I_REP_NUM IN REP.REP_NUM%TYPE) AS
2 I_CUSTOMER_NUM CUSTOMER.CUSTOMER_NUM%TYPE;
3 I_CUSTOMER_NAME CUSTOMER.CUSTOMER_NAME%TYPE;
4
5 CURSOR CUSTGROUP IS
6 SELECT CUSTOMER_NUM, CUSTOMER_NAME
7 FROM CUSTOMER
8 WHERE REP_NUM = I_REP_NUM;
9
10 BEGIN
11 OPEN CUSTGROUP;
12 LOOP
13   FETCH CUSTGROUP INTO I_CUSTOMER_NUM, I_CUSTOMER_NAME;
14   EXIT WHEN CUSTGROUP%NOTFOUND;
15   DBMS_OUTPUT.PUT_LINE(I_CUSTOMER_NUM||' '||I_CUSTOMER_NAME);
16 END LOOP;
17 CLOSE CUSTGROUP;
18 END;
19 /

```

FIGURE 8-23 Procedure with a cursor

Figure 8-24 shows the results of using the procedure. After the user enters 20 as the value for the rep number, the procedure displays the number and name of each customer of sales rep 20.

```

BEGIN
DISP_REP_CUST('20');
END;
/

```

Results Explain Describe Saved SQL History

```

148 Al's Appliance and Sport
524 Kline's
842 All Season
Statement processed.

```

FIGURE 8-24 Results of using the procedure

Using More Complex Cursors

The query formulation that defined the cursor in Example 10 was straightforward. Any SQL query is legitimate in a cursor definition. In fact, the more complicated the requirements for retrieval, the more numerous the benefits derived by the programmer who uses embedded SQL. Consider the query in Example 11.

EXAMPLE 11

For each order that contains an order line for the part whose part number is stored in `I_PART_NUM`, retrieve the order number, order date, customer number, name of the customer that placed the order, and last and first names of the sales rep who represents the customer.

254

Opening and closing the cursor is done exactly as shown in Example 10. The only difference in the `FETCH` command is that a different set of variables is used in the `INTO` clause. Thus, the only real difference is the cursor definition. The procedure shown in Figure 8-25 contains the appropriate cursor definition.

```
1 CREATE OR REPLACE PROCEDURE DISP_PART_ORDERS (I_PART_NUM IN PART.PART_NUM%TYPE) AS
2
3 I_ORDER_NUM ORDERS.ORDER_NUM%TYPE;
4 I_ORDER_DATE ORDERS.ORDER_DATE%TYPE;
5 I_CUSTOMER_NUM ORDERS.CUSTOMER_NUM%TYPE;
6 I_REP_NUM REP.REP_NUM%TYPE;
7 I_LAST_NAME REP.LAST_NAME%TYPE;
8 I_FIRST_NAME REP.FIRST_NAME%TYPE;
9
10 CURSOR ORDGROUP IS
11 SELECT ORDERS.ORDER_NUM, ORDER_DATE, ORDERS.CUSTOMER_NUM, CUSTOMER.REP_NUM,
12 LAST_NAME, FIRST_NAME
13 FROM ORDER_LINE, ORDERS, CUSTOMER, REP
14 WHERE ORDER_LINE.ORDER_NUM = ORDERS.ORDER_NUM
15 AND ORDERS.CUSTOMER_NUM = CUSTOMER.CUSTOMER_NUM
16 AND CUSTOMER.REP_NUM = REP.REP_NUM
17 AND PART_NUM = I_PART_NUM;
18
19 BEGIN
20
21 OPEN ORDGROUP;
22 LOOP
23     FETCH ORDGROUP INTO I_ORDER_NUM, I_ORDER_DATE, I_CUSTOMER_NUM, I_REP_NUM,
24     I_LAST_NAME, I_FIRST_NAME;
25     EXIT WHEN ORDGROUP%NOTFOUND;
26
27     DBMS_OUTPUT.PUT_LINE(I_ORDER_NUM);
28     DBMS_OUTPUT.PUT_LINE(I_ORDER_DATE);
29     DBMS_OUTPUT.PUT_LINE(I_CUSTOMER_NUM);
30     DBMS_OUTPUT.PUT_LINE(I_LAST_NAME);
31     DBMS_OUTPUT.PUT_LINE(I_FIRST_NAME);
32
33 END LOOP;
34
35 CLOSE ORDGROUP;
36 END;
37 /
```

FIGURE 8-25 Procedure with a cursor that involves joining multiple tables

The results of using this procedure to display the results for part DR93 are shown in Figure 8-26.

```
BEGIN
DISP_PART_ORDERS('DR93');
END;
/
```

Results	Explain	Describe	Saved SQL	History
21610				
20-OCT-10				
356				
Perez				
Juan				
21619				
23-OCT-10				
148				
Kaiser				
Valerie				
Statement processed.				

FIGURE 8-26 Results of using the procedure to display orders containing part DR93

Advantages of Cursors

The retrieval requirements in Example 11 are substantial. Beyond coding the preceding cursor definition, the programmer doesn't need to worry about the mechanics of obtaining the necessary data or placing it in the right order, because this happens automatically when the cursor is opened. To the programmer, it seems as if a sequential file already exists that contains the correct data, sorted in the right order. This assumption leads to three main advantages:

1. The coding in the procedure is greatly simplified.
2. In a normal program, the programmer must determine the most efficient way to access the data. In a program or procedure using embedded SQL, the optimizer determines the best way to access the data. The programmer isn't concerned with the best way to retrieve the data. In addition, when an underlying structure changes (for example, an additional index is created), the optimizer determines the best way to execute the query with the new structure. The program or procedure does not have to change at all.
3. When the database structure changes in such a way that the necessary information is still obtainable using a different query, the only change required in the program or procedure is the cursor definition. The procedural code is not affected.

USING T-SQL IN SQL SERVER

SQL Server uses an extended version of SQL called T-SQL (Transact-SQL). You can use T-SQL to create stored procedures and use cursors. The reasons for creating and using stored procedures and cursors are identical to those discussed in the PL/SQL section. Only the command syntax is different.

Retrieving a Single Row and Column

In Example 7, you learned how to write a procedure in PL/SQL that takes a rep number as input and displays the corresponding rep name. The following code shows how you would create the stored procedure in T-SQL:

256

```
CREATE PROCEDURE usp_DISP_REP_NAME
@repnum char(2)
AS
SELECT RTRIM(FIRST_NAME) + ' ' + RTRIM(LAST_NAME)
FROM REP
WHERE REP_NUM = @repnum
```

The CREATE PROCEDURE command in the stored procedure causes SQL Server to create a procedure named `usp_DISP_REP_NAME`. The `usp_` prefix identifies the procedure as a user-stored procedure. Although using the prefix is optional, it is an easy way to differentiate user-stored procedures from SQL Server system-stored procedures. The argument for this procedure is `@repnum`. In T-SQL, you must assign a data type to parameters. All arguments start with the at (`@`) sign. Arguments should have the same data type and length as the particular column in a table that they represent. In the REP table, REP_NUM was defined with a CHAR data type and a length of 2. The CREATE PROCEDURE ends with the word AS followed by the SELECT command that comprises the procedure.

To call the procedure, use the EXEC command and include any arguments in single quotes. The procedure to find the name of sales rep 20 is:

```
EXEC usp_DISP_REP_NAME '20'
```

The result of executing this procedure is the same as that shown in Figure 8-8.

Changing Data with a Stored Procedure

In Example 8, you learned how to write a procedure in PL/SQL that changes the name of a customer. The following commands show how to create the stored procedure in T-SQL:

```
CREATE PROCEDURE usp_CHG_CUST_NAME
@custnum char(3),
@custname char(35)
AS
UPDATE CUSTOMER
SET CUSTOMER_NAME = @custname
WHERE CUSTOMER_NUM = @custnum
```

The procedure has two arguments, `@custnum` and `@custname`, and uses an UPDATE command instead of a SELECT command. To execute a stored procedure with two arguments, separate the arguments with a comma as shown in the following command:

```
EXEC usp_CHG_CUST_NAME '725', 'Deerfield' 's'
```


Deleting Data with a Stored Procedure

In Example 9, you learned how to write a procedure in PL/SQL that deletes an order number from both the ORDER_LINE table and the ORDERS table. The following commands show how to create the stored procedure in T-SQL:

```
CREATE PROCEDURE usp_DEL_ORDER
@ordernum char(5)
AS
DELETE
FROM
ORDER_LINE
WHERE ORDER_NUM = @ordernum

DELETE
FROM ORDERS
WHERE ORDER_NUM = @ordernum
```

257

Using a Cursor

Cursors serve the same purpose in T-SQL as they do in PL/SQL and work exactly the same way. You need to declare a cursor, open a cursor, fetch rows from a cursor, and close a cursor. The only difference is in the command syntax. The following T-SQL code performs exactly the same task as that shown in Example 10:

```
CREATE PROCEDURE usp_DISP_REP_CUST
@repnum char(2)
AS
DECLARE @custnum char(3)
DECLARE @custname char(35)
DECLARE mycursor CURSOR READ_ONLY

FOR
SELECT CUSTOMER_NUM, CUSTOMER_NAME
FROM CUSTOMER
WHERE REP_NUM = @repnum

OPEN mycursor

FETCH NEXT FROM mycursor
INTO @custnum, @custname

WHILE @@FETCH_STATUS = 0
BEGIN

    PRINT @custnum+' '+@custname

    FETCH NEXT FROM mycursor
    INTO @custnum, @custname

END

CLOSE mycursor
DEALLOCATE mycursor
```

The procedure uses one argument, @repnum. It also uses two variables, and each variable must be declared using a DECLARE statement. You also declare the cursor by giving it a

name, describing its properties, and associating it with a SELECT statement. The cursor property, READ_ONLY, means that the cursor is used for retrieval purposes only. The OPEN, FETCH, and CLOSE commands perform exactly the same tasks in T-SQL as they do in PL/SQL. The OPEN command opens the cursor and causes the query to be executed. The FETCH command advances the cursor to the next row and places the contents of the row in the indicated variables. The CLOSE command closes a cursor and the DEALLOCATE command deletes the cursor. The DEALLOCATE command is not necessary but it does enable the user to use the same cursor name with another procedure.

The WHILE loop will repeat until the value of the system variable @@FETCH_STATUS is not zero. The PRINT command will output the values stored in the @custnum and @custname variables.

Using More Complex Cursors

T-SQL also can handle more complex queries. The T-SQL code for Example 11 is shown below:

```
CREATE PROCEDURE usp_DISP_PART_ORDERS
@partnum char(4)
AS
DECLARE @ordernum char(5)
DECLARE @orderdate datetime
DECLARE @custnum char(3)
DECLARE @repnum char(2)
DECLARE @lastname char(15)
DECLARE @firstname char(15)

DECLARE mycursor CURSOR READ_ONLY

FOR
SELECT ORDERS.ORDER_NUM, ORDER_DATE, ORDERS.CUSTOMER_NUM,
      CUSTOMER.REP_NUM, LAST_NAME, FIRST_NAME
FROM ORDER_LINE, ORDERS, CUSTOMER, REP
WHERE ORDER_LINE.ORDER_NUM = ORDERS.ORDER_NUM
AND ORDERS.CUSTOMER_NUM = CUSTOMER.CUSTOMER_NUM
AND CUSTOMER.REP_NUM = REP.REP_NUM
AND PART_NUM = @partnum

OPEN mycursor

FETCH NEXT FROM mycursor
INTO @ordernum, @orderdate, @custnum, @repnum, @lastname, @firstname

WHILE @@FETCH_STATUS = 0
BEGIN

    PRINT @ordernum
    PRINT @orderdate
    PRINT @custnum
    PRINT @lastname
    PRINT @firstname

    FETCH NEXT FROM mycursor
    INTO @ordernum, @orderdate, @custnum, @repnum, @lastname, @firstname

END

CLOSE mycursor
DEALLOCATE mycursor
```

USING SQL IN MICROSOFT ACCESS

Not every programming language accepts SQL commands as readily as PL/SQL and T-SQL. In Microsoft Access, programs are written in Visual Basic, which does not support embedded SQL commands directly in the code. When the SQL command is stored in a string variable, however, you can use the DoCmd.RunSQL command to run the command. The procedure in which you place the SQL command can include arguments.

Deleting Data with Visual Basic

To delete the sales rep whose number is 20, the command is:

```
DELETE FROM REP WHERE REP_NUM = '20';
```

When you write this type of command, you usually don't know in advance the specific sales rep number that you want to delete; it would be passed as an argument to the procedure containing this DELETE command. In the following example, the sales rep number is stored in an argument named I_REP_NUM.

EXAMPLE 12

Delete from the REP table the sales rep whose number currently is stored in I_REP_NUM.

Statements in the procedure usually create the appropriate DELETE command, using the value in any necessary arguments. For example, when the command is stored in the variable named strSQL (which must be a string variable) and the rep number is stored in the argument I_REP_NUM, the following command is appropriate:

```
strSQL = "DELETE FROM REP WHERE REP_NUM = '"  
strSQL = strSQL & I_REP_NUM  
strSQL = strSQL & "'";
```

The first command sets the strSQL string variable to DELETE FROM REP WHERE REP_NUM = '; that is, it creates everything necessary in the command up to and including the single quotation mark preceding the rep number. The second command uses concatenation (&). It changes strSQL to the result of the previous value concatenated with the value in I_REP_NUM. When I_REP_NUM contains the value 20, for example, the command would be DELETE FROM REP WHERE REP_NUM = '20'. The final command sets strSQL to the result of the value already created, concatenated with a single quotation mark and a semicolon. The command is now complete.

Figure 8-27 shows a completed procedure to accomplish the necessary deletion in Access. You enter this procedure in the Microsoft Visual Basic window. In the program, the Dim statement creates a string variable named strSQL. The next three commands set strSQL to the appropriate SQL command. Finally, the DoCmd.RunSQL command runs the SQL command stored in strSQL.

```
Microsoft Visual Basic - Premiere Products - [Module1 (Code)]
File Edit View Insert Debug Run Tools Add-Ins Window Help
Ln 13, Col 1
(General) RepDelete
Option Compare Database

Public Function RepDelete(I_REP_NUM) ← Argument (I_REP_NUM)
    Dim strSQL As String

    strSQL = "DELETE FROM REP WHERE REP_NUM = '"
    strSQL = strSQL & I_REP_NUM
    strSQL = strSQL & "';" } Commands to set strSQL

    DoCmd.RunSQL strSQL ← Runs command stored in strSQL

End Function
```

FIGURE 8-27 Visual Basic code to delete a sales rep

NOTE

If you have concerns about how you constructed the SQL command in `strSQL`, you can include the `Debug.Print (strSQL)` command after the set of commands that construct `strSQL`. The `Debug.Print` command displays the entire command before it is executed so you can review it for accuracy. If you need to correct an error, rerun the program after making the necessary changes. If you get an error in your program, check your SQL command carefully to make sure that you concatenated it correctly.

Running the Code

Normally, you run code like the function shown in Figure 8-27 by calling it from another procedure or associating it with some event, such as clicking a button on a form. However, you can run it directly by using the Immediate window (click View on the menu bar, and then click Immediate Window to open it). Normally, you would use this window only for testing purposes, but you can use it to see the result of running the code. To run a Function procedure, such as the one shown in Figure 8-27, in the Immediate window, type a question mark followed by the name of the procedure and a set of parentheses, as shown in Figure 8-28. Place the values for any arguments in the parentheses. Assuming that you wanted to delete a sales rep whose number is 50, you would include "50" inside the parentheses as shown in the figure.

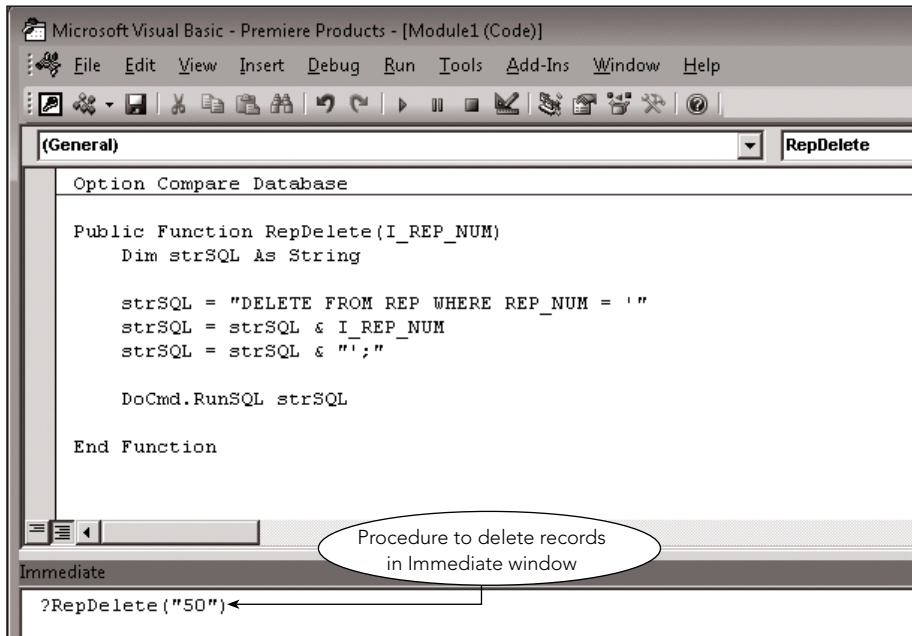


FIGURE 8-28 Running the code in the Immediate window

After you type the command and press the Enter key, the code will run and the appropriate action will occur. In this case, the command deletes the sales rep with the number 50 (assuming there is a sales rep 50).

Updating Data with Visual Basic

A procedure that updates a table using an UPDATE command is similar to the one used to delete a sales rep. In Example 13, two arguments are required. One of them, `I_LAST_NAME`, contains the new name for the sales rep. The other, `I_REP_NUM`, contains the number of the rep whose name is to be changed.

EXAMPLE 13

Change the last name of the sales rep whose number is stored in `I_REP_NUM` to the value currently stored in `I_LAST_NAME`.

This example is similar to the previous one with two important differences. First, you need to use the UPDATE command instead of the DELETE command. Second, there are two arguments, so there are two portions of the construction of the SQL command that involve variables. The complete procedure is shown in Figure 8-29.

```

Public Function RepUpdate(I_LAST_NAME, I_REP_NUM)
    Dim strSQL As String

    strSQL = "UPDATE REP SET LAST_NAME = '"
    strSQL = strSQL + I_LAST_NAME
    strSQL = strSQL + "' WHERE REP_NUM = '"
    strSQL = strSQL & I_REP_NUM
    strSQL = strSQL & "';"

    DoCmd.RunSQL strSQL

End Function

```

Argument giving number of rep whose name is to be changed

Argument giving new name

FIGURE 8-29 Code to change a rep's last name

To run this procedure, you would enter values for both arguments as shown in Figure 8-30.

New last name

?RepUpdate("Webb", "20")

Rep number

FIGURE 8-30 Running the code to change a rep's last name

Inserting Data with Visual Basic

The process for inserting rows is similar in Access when compared to PL/SQL or T-SQL. You create the appropriate INSERT command in the strSQL variable. There will be multiple arguments in the procedure—one for each value to be inserted.

Finding Multiple Rows with Visual Basic

Just as when embedding SQL in PL/SQL, deleting or updating multiple rows causes no problems, because these procedures still represent a single operation, with all the work happening behind the scenes. A SELECT command that returns several rows, however, poses serious problems for record-at-a-time languages like PL/SQL and Visual Basic. You handle SELECT commands differently in Access than you do in PL/SQL or T-SQL. In particular, there are no cursors in Access. Instead, you handle the results of a query just as you might use a loop to process through the records in a table.

EXAMPLE 14

Retrieve and list the number and name of each customer represented by the sales rep whose number is stored in the variable I_REP_NUM.

Figure 8-31 shows a procedure to accomplish the indicated task. The statements involving `rs` and `cnn` are a typical way of processing through a recordset, that is, through all the records contained in a table or in the results of a query. The only difference between this program and one to process all the records in a table is that the `Open` command refers to an SQL command and not a table. (The SQL command is stored in the variable named `strSQL` and is created in the same manner as shown in the previous examples.)

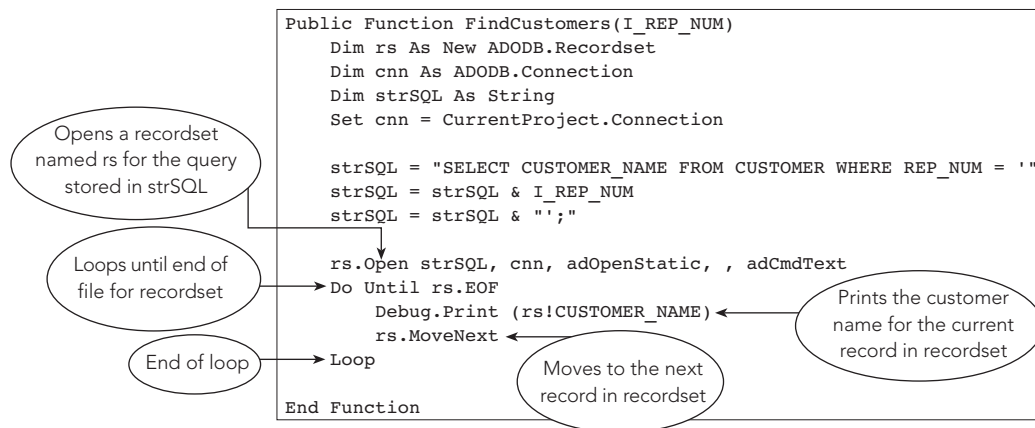


FIGURE 8-31 Code to find customers of a specific rep

The loop continues until reaching the end of file for the recordset, that is, until all records have been processed. Within the loop, you can use the `Debug.Print` command to print a value. In this case, the value to be printed is `rs!CUSTOMER_NAME`. This indicates the contents of the `CUSTOMER_NAME` column for the record in the recordset (`rs`) on which Access is currently positioned. The next command, `rs.MoveNext`, moves to the next record in the recordset. The loop continues until all records in the recordset have been processed.

Figure 8-32 shows the results of running this procedure and entering a value of "35" as an argument. Access displays the four customers of sales rep 35.



FIGURE 8-32 Running the code to find customers of a sales rep

NOTE When you expect an SQL query to return only one record, you use the same process but would not need a loop.

USING A TRIGGER

A **trigger** is a procedure that is executed automatically in response to an associated database operation, such as an INSERT, UPDATE, or DELETE command. Unlike a stored procedure, which is executed in response to a user request, a trigger is executed in response to a command that causes the associated database operation to occur.

The examples in this section assume there is a new column named ON_ORDER in the PART table. This column represents the number of units of a part currently on order. For example, if there are two separate order lines for a part and the number ordered on one order line is 3 and the number ordered on the other order line is 2, the ON_ORDER value for that part will be 5. Adding, changing, or deleting order lines affects the value in the ON_ORDER column for the part. To ensure that the value is updated appropriately, you can use a trigger.

If you created the ADD_ORDER_LINE trigger shown in Figure 8-33, the SQL command in the trigger would be executed when a user adds an order line. The trigger must update the ON_ORDER value for the corresponding part to reflect the order line. For example, if the value in the ON_ORDER column for part CD52 is 4 and the user adds an order line on which the part number is CD52 and the number of units ordered is 2, six units of part CD52 will be on order. When a record is added to the ORDER_LINE table, the ADD_ORDER_LINE trigger updates the PART table by adding the number of units ordered on the order line to the previous value in the ON_ORDER column.

```
1 CREATE OR REPLACE TRIGGER ADD_ORDER_LINE
2 AFTER INSERT ON ORDER_LINE FOR EACH ROW
3 BEGIN
4 UPDATE PART
5 SET ON_ORDER = ON_ORDER + :NEW.NUM_ORDERED ;
6 END;
```

FIGURE 8-33 ADD_ORDER_LINE trigger

The first line indicates that the command is creating a trigger named ADD_ORDER_LINE. The second line indicates that this trigger will be executed after an order line is inserted and that the SQL command is to occur for each row that is added. Like stored procedures, the SQL command is enclosed between the words BEGIN and END. In this case, the SQL command is an UPDATE command. The command uses the NEW qualifier, which refers to the row that is added to the ORDER_LINE table. If an order line is added on which the part number is CD52 and the number ordered is 2, for example, NEW.PART_NUM will be CD52 and NEW.NUM_ORDERED will be 2.

The following UPDATE_ORDER_LINE trigger shown in Figure 8-34 is executed when a user attempts to update an order line. There are two differences between the UPDATE_ORDER_LINE trigger and the ADD_ORDER_LINE trigger. First, the second line of the UPDATE_ORDER_LINE trigger indicates that this trigger is executed after an UPDATE of an order line rather than an INSERT. Second, the computation to update the ON_ORDER column includes both NEW.NUM_ORDERED and OLD.NUM_ORDERED. As with the ADD_ORDER_LINE trigger, NEW.NUM_ORDERED refers to the new value. In an

UPDATE command, however, there is also an old value, which is the value before the update takes place. If an update changes the value for ON_ORDER from 1 to 3, OLD.NUM_ORDERED is 1 and NEW.NUM_ORDERED is 3. Adding NEW.NUM_ORDERED and subtracting OLD.NUM_ORDERED results in a net change of an increase of 2. (The net change could also be negative, in which case the ON_ORDER value decreases.)

```
1 CREATE OR REPLACE TRIGGER UPDATE_ORDER_LINE
2 AFTER UPDATE ON ORDER_LINE FOR EACH ROW
3 BEGIN
4 UPDATE PART
5 SET ON_ORDER = ON_ORDER + :NEW.NUM_ORDERED - :OLD.NUM_ORDERED;
6 END;
```

265

FIGURE 8-34 UPDATE_ORDER_LINE trigger

The DELETE_ORDER_LINE trigger shown in Figure 8-35 performs a function similar to the other two. When an order line is deleted, the ON_ORDER value for the corresponding part is updated by subtracting OLD.NUM_ORDERED from the current ON_ORDER value. (In a delete operation, there is no NEW.NUM_ORDERED.)

```
1 CREATE OR REPLACE TRIGGER DELETE_ORDER_LINE
2 AFTER DELETE ON ORDER_LINE FOR EACH ROW
3 BEGIN
4 UPDATE PART
5 SET ON_ORDER = ON_ORDER - :OLD.NUM_ORDERED ;
6 END;
```

FIGURE 8-35 DELETE_ORDER_LINE trigger

ACCESS USER NOTE

Access does not support triggers. When using a form to update table data, you can achieve some of the same functionality by creating VBA code to be executed after the insertion, update, or deletion of records.

In SQL Server, you create triggers using T-SQL. The code to create the ADD_ORDER_LINE trigger is:

```
CREATE TRIGGER ADD_ORDER_LINE
ON ORDER_LINE
AFTER INSERT
AS

DECLARE @numbord decimal(3,0)
SELECT @numbord = (SELECT NUM_ORDERED FROM INSERTED)
UPDATE PART
SET ON_ORDER = ON_ORDER + @numbord
```

266

This trigger uses one variable, @numbord, and the value placed in that variable is obtained from the SELECT statement. The INSERTED table is a temporary system table that contains a copy of the values that the last SQL command inserted. The column names are the same column names as in the ORDER_LINE table. The INSERTED table holds the most recent value of the NUM_ORDERED column which is what you need to update the PART table.

The T-SQL trigger that executes after an UPDATE of an order line is:

```
CREATE TRIGGER UPDATE_ORDER_LINE
ON ORDER_LINE
AFTER UPDATE
AS

DECLARE @newnumbord decimal(3,0)
DECLARE @oldnumbord decimal(3,0)
SELECT @newnumbord = (SELECT NUM_ORDERED FROM INSERTED)
SELECT @oldnumbord = (SELECT NUM_ORDERED FROM DELETED)
UPDATE PART
SET ON_ORDER = ON_ORDER + @newnumbord - @oldnumbord
```

This trigger uses the INSERTED table and the DELETED table. The DELETED table contains the previous value of the NUM_ORDERED column while the INSERTED column contains the updated value. The DELETE_ORDER_LINE trigger uses only the DELETED system table:

```
CREATE TRIGGER DELETE_ORDER_LINE
ON ORDER_LINE
AFTER DELETE
AS

DECLARE @numbord decimal(3,0)
SELECT @numbord = (SELECT NUM_ORDERED FROM DELETED)
UPDATE PART
SET ON_ORDER = ON_ORDER - @numbord
```

Chapter Summary

- There are functions whose results are based on the values in single records. UPPER and LOWER are two examples of functions that act on character data. UPPER displays each lowercase letter in the argument in uppercase. LOWER displays each uppercase letter in the argument in lowercase.
- ROUND and FLOOR are two examples of functions that act on numeric data. ROUND produces its result by rounding the value to the specified number of decimal places. FLOOR produces its result by truncating (removing) everything to the right of the decimal point.
- Use the ADD_MONTHS function in Oracle to add a specific number of months to a date. In Access and in SQL Server, use the DATEADD() function.
- To add a specific number of days to a date, use normal addition. You can subtract one date from another to produce the number of days between two dates.
- To obtain today's date, use the SYSDATE function in Oracle, the GETDATE() function in SQL Server, and the DATE() function in Access.
- To concatenate values in character columns in Oracle, separate the column names with two vertical lines (||). Use the RTRIM function to delete any extra spaces that follow the values. In SQL Server, use the + symbol to concatenate values. In Access, use the ampersand (&) symbol to concatenate values.
- A stored procedure is a query saved in a file that users can execute later.
- To create a stored procedure in PL/SQL or T-SQL, use the CREATE PROCEDURE command.
- Variables in PL/SQL procedures are declared after the word DECLARE. To assign variables the same type as a column in the database, use the %TYPE attribute.
- Use the INTO clause in the SELECT command to place the results of a SELECT command in variables in Oracle.
- You can use INSERT, UPDATE, and DELETE commands in PL/SQL and T-SQL procedures, even when they affect more than one row.
- When a SELECT command is used to retrieve more than one row in PL/SQL or T-SQL, it must define a cursor that will select one row at a time.
- Use the OPEN command to activate a cursor and execute the query in the cursor definition.
- Use the FETCH command to select the next row in PL/SQL and T-SQL.
- Use the CLOSE command to deactivate a cursor. The rows initially retrieved will no longer be available to PL/SQL or T-SQL.
- To use SQL commands in Access, create the command in a string variable. To run the command stored in the string variable, use the DoCmd.RunSQL command.
- To process a collection of rows retrieved by a SELECT command in Access, use a recordset. Create the SQL command in a string variable and use the string variable in the command to open the recordset.
- To move to the next record in a recordset in Access, use the MoveNext command.

- A trigger is an action that occurs automatically in response to an associated database operation, such as an INSERT, UPDATE, or DELETE command. Like a stored procedure, a trigger is stored and compiled on the server. Unlike a stored procedure, which is executed in response to a user request, a trigger is executed in response to a command that causes the associated database operation to occur.

Key Terms

ADD_MONTHS	OPEN
argument	PL/SQL
call	procedural code
client	procedural language
client/server system	ROUND
CLOSE	RTRIM
concatenate	server
concatenation	stored procedure
cursor	SYSDATE
embed	Transact-SQL
FETCH	trigger
FLOOR	T-SQL
LOWER	update procedure
nonprocedural language	UPPER

Review Questions

1. How do you display letters in uppercase in Oracle, Access, and SQL Server? How do you display letters in lowercase in Oracle, Access, and SQL Server?
2. How do you round a number to a specific number of decimal places in Oracle, Access, and SQL Server? How do you remove everything to the right of the decimal place in Oracle and SQL Server?
3. How do you add months to a date in Oracle, Access, and SQL Server? How do you add days to a date? How would you find the number of days between two dates?
4. How do you obtain today's date in Oracle, Access, and SQL Server?
5. How do you concatenate values in character columns in Oracle, Access, and SQL Server?
6. Which function deletes extra spaces at the end of a value?
7. What are stored procedures? What purpose do they serve?
8. In which portion of a PL/SQL procedure do you embed SQL commands?
9. Where do you declare variables in PL/SQL procedures?
10. In PL/SQL, how do you assign variables the same type as a column in the database?
11. How do you place the results of a SELECT command into variables in PL/SQL?

12. Can you use INSERT, UPDATE, or DELETE commands that affect more than one row in PL/SQL procedures?
13. How do you use a SELECT command that retrieves more than one row in a PL/SQL procedure?
14. Which PL/SQL command activates a cursor?
15. Which PL/SQL command selects the next row in a cursor?
16. Which PL/SQL command deactivates a cursor?
17. How do you use SQL commands in Access?
18. How do you process a collection of rows retrieved by a SELECT command in Access?
19. How do you move to the next record in a recordset in Access?
20. What are triggers? What purpose do they serve?
21. What is the purpose of the INSERTED and DELETED tables in SQL Server?

Exercises

Premiere Products

Use the Premiere Products database (see Figure 1-2 in Chapter 1) to complete the following exercises. If directed to do so by your instructor, use the information provided with the Chapter 3 Exercises to print your output.

1. List the part number and description for all parts. The part descriptions should appear in uppercase letters.
2. List the customer number and name for all customers located in the city of Grove. Your query should ignore case. For example, a customer with the city Grove should be included as should customers whose city is GROVE, grove, GrOvE, and so on.
3. List the customer number, name, and balance for all customers. The balance should be rounded to the nearest dollar.
4. Premiere Products is running a promotion that is valid for up to 20 days after an order is placed. List the order number, customer number, customer name, and the promotion date for each order. The promotion date is 20 days after the order was placed.
5. Write PL/SQL or T-SQL procedures to accomplish the following tasks:
 - a. Obtain the name and credit limit of the customer whose number currently is stored in I_CUSTOMER_NUM. Place these values in the variables I_CUSTOMER_NAME and I_CREDIT_LIMIT, respectively. Output the contents of I_CUSTOMER_NAME and I_CREDIT_LIMIT.
 - b. Obtain the order date, customer number, and name for the order whose number currently is stored in I_ORDER_NUM. Place these values in the variables I_ORDER_DATE, I_CUSTOMER_NUM and I_CUSTOMER_NAME, respectively. Output the contents of I_ORDER_DATE, I_CUSTOMER_NUM, and I_CUSTOMER_NAME.
 - c. Add a row to the ORDERS table.

- d. Change the date of the order whose number is stored in I_ORDER_NUM to the date currently found in I_ORDER_DATE.
 - e. Delete the order whose number is stored in I_ORDER_NUM.
6. Write a PL/SQL or T-SQL procedure to retrieve and output the part number, part description, warehouse number, and unit price of every part in the item class stored in I_CLASS.
 7. Write Access functions to accomplish the following tasks:
 - a. Delete the order whose number is stored in I_ORDER_NUM.
 - b. Change the date of the order whose number is stored in I_ORDER_NUM to the date currently found in I_ORDER_DATE.
 - c. Retrieve and output the part number, part description, warehouse number, and unit price of every part in the item class stored in I_CLASS.
 8. Write a stored procedure in PL/SQL or T-SQL that will change the price of a part with a given part number. How would you use this stored procedure to change the price of part AT94 to \$26.95?
 9. Write the code for the following triggers in PL/SQL or T-SQL following the style shown in the text:
 - a. When adding a customer, add the customer's balance times the sales rep's commission rate to the commission for the corresponding sales rep.
 - b. When updating a customer, add the difference between the new balance and the old balance multiplied by the sales rep's commission rate to the commission for the corresponding sales rep.
 - c. When deleting a customer, subtract the balance multiplied by the sales rep's commission rate from the commission for the corresponding sales rep.

Henry Books

Use the Henry Books database (see Figures 1-4 through 1-7 in Chapter 1) to complete the following exercises. If directed to do so by your instructor, use the information provided with the Chapter 3 Exercises to print your output.

1. List the author number, first name, and last name for all authors. The first name should appear in lowercase letters and the last name should appear in uppercase letters.
2. List the publisher code and name for all publishers located in the city of New York. Your query should ignore case. For example, a customer with the city New York should be included as should customers whose city is NEW YORK, New york, NeW yOrK, and so on.
3. List the book code, title, and price for all books. The price should be rounded to the nearest dollar.
4. Write PL/SQL or T-SQL procedures to accomplish the following tasks:
 - a. Obtain the first name and last name of the author whose number currently is stored in I_AUTHOR_NUM. Place these values in the variables I_AUTHOR_FIRST and I_AUTHOR_LAST. Output the contents of I_AUTHOR_NUM, I_AUTHOR_FIRST, and I_AUTHOR_LAST.

- b. Obtain the book title, publisher code, and publisher name for every book whose code currently is stored in I_BOOK_CODE. Place these values in the variables I_TITLE, I_PUBLISHER_CODE, and I_PUBLISHER_NAME, respectively. Output the contents of I_TITLE, I_PUBLISHER_CODE, and I_PUBLISHER_NAME.
 - c. Add a row to the AUTHOR table.
 - d. Change the last name of the author whose number is stored in I_AUTHOR_NUM to the value currently found in I_AUTHOR_LAST.
 - e. Delete the author whose number is stored in I_AUTHOR_NUM.
5. Write a PL/SQL or T-SQL procedure to retrieve and output the book code, title, book type, and price for every book whose publisher code is stored in I_PUBLISHER_CODE.
 6. Write Access functions to accomplish the following tasks:
 - a. Delete the author whose number is stored in I_AUTHOR_NUM.
 - b. Change the last name of the author whose number is stored in I_AUTHOR_NUM to the value currently found in I_AUTHOR_LAST.
 - c. Retrieve and output the book code, title, book type, and price for every book whose publisher code is stored in I_PUBLISHER_CODE.
 7. Write a stored procedure in PL/SQL or T-SQL that will change the price of a book with a given book code. How would you use this stored procedure to change the price of book 0189 to \$8.49?
 8. Assume the BOOK table contains a column called TOTAL_ON_HAND that represents the total units on hand in all branches for that book. Following the style shown in the text, write the code in PL/SQL or T-SQL for the following triggers:
 - a. When inserting a row in the INVENTORY table, add the ON_HAND value to the TOTAL_ON_HAND value for the appropriate book.
 - b. When updating a row in the INVENTORY table, add the difference between the new ON_HAND value and the old ON_HAND value to the TOTAL_ON_HAND value for the appropriate book.
 - c. When deleting a row in the INVENTORY table, subtract the ON_HAND value from the TOTAL_ON_HAND value for the appropriate book.

Alexamara Marina Group

Use the Alexamara Marina Group database (see Figures 1-8 through 1-12 in Chapter 1) to complete the following exercises. If directed to do so by your instructor, use the information provided with the Chapter 3 Exercises to print your output.

1. List the owner number, first name, and last name for all owners. The first name should appear in uppercase letters and the last name should appear in lowercase letters.
2. List the owner number and last name for all owners located in the city of Bowton. Your query should ignore case. For example, a customer with the city Bowton should be included as should customers whose city is BOWTON, BowTon, BoWtOn, and so on.

3. Alexamara is offering a discount for owners who sign up early for slips for next year. The discount is 1.75 percent of the rental fee. For each slip, list the marina number, slip number, owner number, owner's last name, rental fee, and discount. The discount should be rounded to the nearest dollar.
4. Write PL/SQL or T-SQL procedures to accomplish the following tasks:
 - a. Obtain the first name and last name of the owner whose number currently is stored in `I_OWNER_NUM`. Place these values in the variables `I_FIRST_NAME` and `I_LAST_NAME`. Output the contents of `I_OWNER_NUM`, `I_FIRST_NAME`, and `I_LAST_NAME`.
 - b. Obtain the marina number, slip number, boat name, owner number, owner first name, and owner last name for the slip whose slip ID is currently stored in `I_SLIP_ID`. Place these values in the variables `I_MARINA_NUM`, `I_SLIP_NUM`, `I_BOAT_NAME`, `I_OWNER_NUM`, `I_FIRST_NAME`, and `I_LAST_NAME`, respectively. Output the contents of `I_SLIP_ID`, `I_MARINA_NUM`, `I_SLIP_NUM`, `I_BOAT_NAME`, `I_OWNER_NUM`, `I_FIRST_NAME`, and `I_LAST_NAME`.
 - c. Add a row to the `OWNER` table.
 - d. Change the last name of the owner whose number is stored in `I_OWNER_NUM` to the value currently found in `I_LAST_NAME`.
 - e. Delete the owner whose number is stored in `I_OWNER_NUM`.
5. Write a PL/SQL or T-SQL procedure to retrieve and output the marina number, slip number, rental fee, boat name, and owner number for every slip whose length is equal to the length stored in `I_LENGTH`.
6. Write Access functions to accomplish the following tasks:
 - a. Delete the owner whose number is stored in `I_OWNER_NUM`.
 - b. Change the last name of the owner whose number is stored in `I_OWNER_NUM` to the value currently found in `I_LAST_NAME`.
 - c. Retrieve and output the marina number, slip number, rental fee, boat name, and owner number for every slip whose length is equal to the length stored in `I_LENGTH`.
7. Write a stored procedure in PL/SQL or T-SQL that will change the rental fee of a slip with a given slip ID and marina number. How would you use this stored procedure to change the rental fee for the boat with the slip ID 3 in marina 1 to \$3,700?
8. Assume the `OWNER` table contains a column called `TOTAL_RENTAL` that represents the total rental fee for all slips rented by that owner. Write the code in PL/SQL or T-SQL for the following triggers following the style shown in the text:
 - a. When inserting a row in the `MARINA_SLIP` table, add the rental fee to the total rental for the appropriate owner.
 - b. When updating a row in the `MARINA_SLIP` table, add the difference between the new rental fee and the old rental fee to the total rental for the appropriate owner.
 - c. When deleting a row in the `MARINA_SLIP` table, subtract the rental fee from the total rental for the appropriate owner.

CHAPTER 8—SQL FUNCTIONS AND PROCEDURES

1. Use the UPPER function to display letters in uppercase in Oracle and SQL Server. In Access, use the UCASE() function. Use the LOWER function to display letters in lowercase in Oracle and SQL Server. In Access, use the LCASE() function.
3. To add months to a date, use the ADD_MONTHS function (Oracle), or the DATEADD() function (Access and SQL Server). To add days to a date, add the desired number of days to a date. To find the number of days between two dates, subtract the earlier date from the later date.
5. In Oracle, separate the column names with two vertical lines (||) in the SELECT clause. In SQL Server, separate the column names with the + symbol. In Access, separate the column names with the & symbol.
7. A stored procedure is a file that is stored on a server and contains commands that can be used repeatedly. Stored procedures eliminate the need for users to retype a query each time it is needed.
9. In PL/SQL procedures, you declare variables first before any procedural code.
11. Use the INTO clause to place the results of a SELECT statement in variables.
13. When retrieving multiple rows with a SELECT statement, use a cursor.
15. FETCH
17. To use SQL commands in Access, create the command in a string variable. To run the command stored in the string variable, use the DoCmd.RunSQL command.
19. To move to the next record in an Access recordset, use the MoveNext command.
21. The INSERTED and DELETED tables are temporary system tables created by SQL Server. The INSERTED table contains the most recent (updated) values in a record and the DELETED table contains the previous (before update) value.