Advanced SQL

8

EIGHT

In this chapter, you will learn:

- About the relational set operators UNION, UNION ALL, INTERSECT, and MINUS
- How to use the advanced SQL JOIN operator syntax
- About the different types of subqueries and correlated queries
- How to use SQL functions to manipulate dates, strings, and other data
- How to create and use updatable views
- How to create and use triggers and stored procedures
- How to create embedded SQL

Preview

In Chapter 7, Introduction to Structured Query Language (SQL), you learned the basic SQL data definition and data manipulation commands used to create and manipulate relational data. In this chapter, you build on what you learned in Chapter 7 and learn how to use more advanced SQL features.

In this chapter, you learn about the SQL relational set operators (UNION, INTERSECT, and MINUS) and how those operators are used to merge the results of multiple queries. Joins are at the heart of SQL, so you must learn how to use the SQL JOIN statement to extract information from multiple tables. In the previous chapter, you learned how cascading queries inside other queries can be useful in certain circumstances. In this chapter, you also learn about the different styles of subqueries that can be implemented in a SELECT statement. Finally, you learn more of SQL's many functions to extract information from data, including manipulation of dates and strings and computations based on stored or even derived data.

In the real world, business procedures require the execution of clearly defined actions when a specific event occurs, such as the addition of a new invoice or a student's enrollment in a class. Such procedures can be applied within the DBMS through the use of triggers and stored procedures. In addition, SQL facilitates the application of business procedures when it is embedded in a programming language such as Visual Basic .Net, C#, or COBOL.

Online Content

Although most of the examples used in this chapter are shown in Oracle, you could also use MS SQL Server. The Student Online companion provides you with the **ADVSQLDBINIT.SQL** script file (Oracle and MS SQL versions) to create the tables and load the data used in this chapter. There you will also find additional SQL script files to demonstrate each of the commands shown in this chapter.

8.1 RELATIONAL SET OPERATORS

In Chapter 3, The Relational Database Model, you learned about the eight general relational operators. In this section, you will learn how to use three SQL commands (UNION, INTERSECT, and MINUS) to implement the union, intersection, and difference relational operators.

In previous chapters, you learned that SQL data manipulation commands are set-oriented; that is, they operate over entire sets of rows and columns (tables) at once. Using sets, you can combine two or more sets to create new sets (or relations). That's precisely what the UNION, INTERSECT, and MINUS statements do. In relational database terms, you can use the words "sets," "relations," and "tables" interchangeably because they all provide a conceptual view of the data set as it is presented to the relational database user.

Note

The SQL standard defines the operations that all DBMSs must perform on data, but it leaves the implementation details to the DBMS vendors. Therefore, some advanced SQL features might not work on all DBMS implementations. Also, some DBMS vendors might implement additional features not found in the SQL standard.

UNION, INTERSECT, and MINUS are the names of the SQL statements implemented in Oracle. The SQL standard uses the keyword EXCEPT to refer to the difference (MINUS) relational operator. Other RDBMS vendors might use a different command name or might not implement a given command at all.

To learn more about the ANSI/ISO SQL standards, check the ANSI Web site (*www.ansi.org*) to find out how to obtain the latest standard documents in electronic form. As of this writing, the most recent published standard is SQL-2003. The SQL-2003 standard makes revisions and additions to the previous standard; most notable is support for XML data.

UNION, INTERSECT, and MINUS work properly only if relations are **union-compatible**, *which* means that the names of the relation attributes must be the same and their data types must be alike. In practice, some RDBMS vendors require the data types to be "compatible" but not necessarily "exactly the same." For example, compatible data types are VARCHAR (35) and CHAR (15). In that case, both attributes store character (string) values; the only difference is the string size. Another example of compatible data types is NUMBER and SMALLINT. Both data types are used to store numeric values.

Note

Some DBMS products might require union-compatible tables to have *identical* data types.

ONLINE CONTENT

The Student Online Companion provides you with SQL script files (Oracle and MS SQL Server) to demonstrate the UNION, INTERSECT, and MINUS commands. It also provides the **Ch08_SaleCo** MS Access database containing supported set operator alternative queries.

8.1.1 UNION

Suppose SaleCo has bought another company. SaleCo's management wants to make sure that the acquired company's customer list is properly merged with SaleCo's customer list. Because it is quite possible that some customers have purchased goods from both companies, the two lists might contain common customers. SaleCo's management wants to make sure that customer records are not duplicated when the two customer lists are merged. The UNION query is a perfect tool for generating a combined listing of customers—one that excludes duplicate records.

The UNION statement combines rows from two or more queries *without including duplicate rows*. The syntax of the UNION statement is:

query UNION *query*

In other words, the UNION statement combines the output of two SELECT queries. (Remember that the SELECT statements must be union-compatible. That is, they must return the same attribute names and similar data types.)

To demonstrate the use of the UNION statement in SQL, let's use the CUSTOMER and CUSTOMER_2 tables in the **Ch08_SaleCo** database. To show the combined CUSTOMER and CUSTOMER_2 records without the duplicates, the UNION query is written as follows:

```
SELECT      CUS_LNAME, CUS_FNAME, CUS_INITIAL, CUS_AREACODE, CUS_PHONE
FROM        CUSTOMER
UNION
SELECT      CUS_LNAME, CUS_FNAME, CUS_INITIAL, CUS_AREACODE, CUS_PHONE
FROM        CUSTOMER_2;
```

Figure 8.1 shows the contents of the CUSTOMER and CUSTOMER_2 tables and the result of the UNION query. Although MS Access is used to show the results here, similar results can be obtained with Oracle.

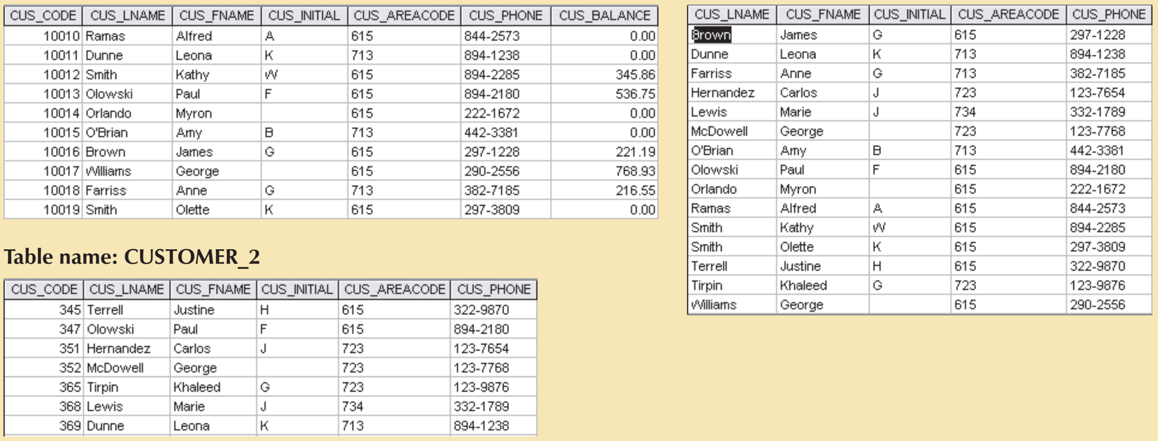
Note the following in Figure 8.1:

- The CUSTOMER table contains 10 rows, while the CUSTOMER_2 table contains 7 rows.
- Customers Dunne and Olowski are included in the CUSTOMER table as well as in the CUSTOMER_2 table.
- The UNION query yields 15 records because the duplicate records of customers Dunne and Olowski are not included. In short, the UNION query yields a unique set of records.

NOTE

The SQL standard calls for the elimination of duplicate rows when the UNION SQL statement is used. However, some DBMS vendors might not adhere to that standard. Check your DBMS manual to see if the UNION statement is supported and if so, *how* it is supported.

FIGURE 8.1 UNION query results



Database name: CH08_SaleCo

Table name: CUSTOMER

CUS_CODE	CUS_LNAME	CUS_FNAME	CUS_INITIAL	CUS_AREACODE	CUS_PHONE	CUS_BALANCE
10010	Ramas	Alfred	A	615	844-2573	0.00
10011	Dunne	Leona	K	713	894-1238	0.00
10012	Smith	Kathy	W	615	894-2285	345.86
10013	Olowski	Paul	F	615	894-2180	536.75
10014	Orlando	Myron		615	222-1672	0.00
10015	O'Brian	Amy	B	713	442-3381	0.00
10016	Brown	James	G	615	297-1228	221.19
10017	Williams	George		615	290-2556	768.93
10018	Farriss	Anne	G	713	382-7185	216.55
10019	Smith	Olette	K	615	297-3809	0.00

Table name: CUSTOMER_2

CUS_CODE	CUS_LNAME	CUS_FNAME	CUS_INITIAL	CUS_AREACODE	CUS_PHONE
345	Terrell	Justine	H	615	322-9870
347	Olowski	Paul	F	615	894-2180
351	Hernandez	Carlos	J	723	123-7654
352	McDowell	George		723	123-7768
365	Tirpin	Khaleed	G	723	123-9876
368	Lewis	Marie	J	734	332-1789
369	Dunne	Leona	K	713	894-1238

Query name: qryUNION-of-CUSTOMER-and-CUSTOMER_2

CUS_LNAME	CUS_FNAME	CUS_INITIAL	CUS_AREACODE	CUS_PHONE
Brown	James	G	615	297-1228
Dunne	Leona	K	713	894-1238
Farriss	Anne	G	713	382-7185
Hernandez	Carlos	J	723	123-7654
Lewis	Marie	J	734	332-1789
McDowell	George		723	123-7768
O'Brian	Amy	B	713	442-3381
Olowski	Paul	F	615	894-2180
Orlando	Myron		615	222-1672
Ramas	Alfred	A	615	844-2573
Smith	Kathy	W	615	894-2285
Smith	Olette	K	615	297-3809
Terrell	Justine	H	615	322-9870
Tirpin	Khaleed	G	723	123-9876
Williams	George		615	290-2556

The UNION statement can be used to unite more than just two queries. For example, assume that you have four union-compatible queries named T1, T2, T3, and T4. With the UNION statement, you can combine the output of all four queries into a single result set. The SQL statement will be similar to this:

```
SELECT column-list FROM T1
UNION
SELECT column-list FROM T2
UNION
SELECT column-list FROM T3
UNION
SELECT column-list FROM T4;
```

8.1.2 UNION ALL

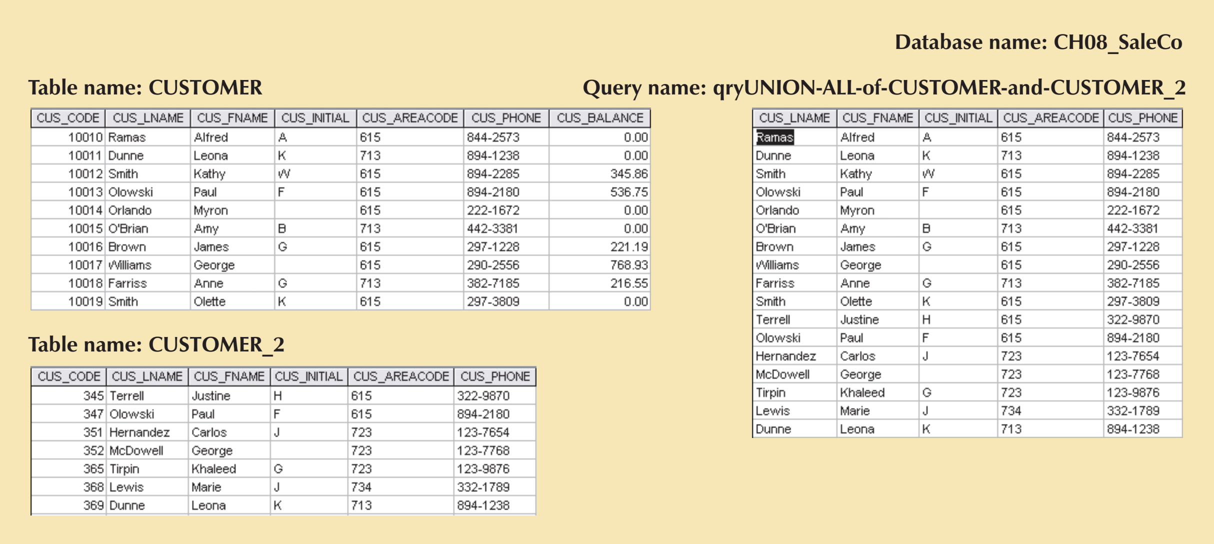
If SaleCo's management wants to know how many customers are on *both* the CUSTOMER and CUSTOMER_2 lists, a UNION ALL query can be used to produce a relation that retains the duplicate rows. Therefore, the following query will keep all rows from both queries (including the duplicate rows) and return 17 rows.

```
SELECT      CUS_LNAME, CUS_FNAME, CUS_INITIAL, CUS_AREACODE, CUS_PHONE
FROM        CUSTOMER
UNION ALL
SELECT      CUS_LNAME, CUS_FNAME, CUS_INITIAL, CUS_AREACODE, CUS_PHONE
FROM        CUSTOMER_2;
```

Running the preceding UNION ALL query produces the result shown in Figure 8.2.

Like the UNION statement, the UNION ALL statement can be used to unite more than just two queries.

FIGURE 8.2 UNION ALL query results



Database name: CH08_SaleCo

Table name: CUSTOMER

CUS_CODE	CUS_LNAME	CUS_FNAME	CUS_INITIAL	CUS_AREACODE	CUS_PHONE	CUS_BALANCE
10010	Ramas	Alfred	A	615	844-2573	0.00
10011	Dunne	Leona	K	713	894-1238	0.00
10012	Smith	Kathy	W	615	894-2285	345.86
10013	Olowski	Paul	F	615	894-2180	536.75
10014	Orlando	Myron		615	222-1672	0.00
10015	O'Brian	Amy	B	713	442-3381	0.00
10016	Brown	James	G	615	297-1228	221.19
10017	Williams	George		615	290-2556	768.93
10018	Farriss	Anne	G	713	382-7185	216.55
10019	Smith	Olette	K	615	297-3809	0.00

Table name: CUSTOMER_2

CUS_CODE	CUS_LNAME	CUS_FNAME	CUS_INITIAL	CUS_AREACODE	CUS_PHONE
345	Terrell	Justine	H	615	322-9870
347	Olowski	Paul	F	615	894-2180
351	Hernandez	Carlos	J	723	123-7654
352	McDowell	George		723	123-7768
365	Tirpin	Khaleed	G	723	123-9876
368	Lewis	Marie	J	734	332-1789
369	Dunne	Leona	K	713	894-1238

Query name: qryUNION-ALL-of-CUSTOMER-and-CUSTOMER_2

CUS_LNAME	CUS_FNAME	CUS_INITIAL	CUS_AREACODE	CUS_PHONE
Ramas	Alfred	A	615	844-2573
Dunne	Leona	K	713	894-1238
Smith	Kathy	W	615	894-2285
Olowski	Paul	F	615	894-2180
Orlando	Myron		615	222-1672
O'Brian	Amy	B	713	442-3381
Brown	James	G	615	297-1228
Williams	George		615	290-2556
Farriss	Anne	G	713	382-7185
Smith	Olette	K	615	297-3809
Terrell	Justine	H	615	322-9870
Olowski	Paul	F	615	894-2180
Hernandez	Carlos	J	723	123-7654
McDowell	George		723	123-7768
Tirpin	Khaleed	G	723	123-9876
Lewis	Marie	J	734	332-1789
Dunne	Leona	K	713	894-1238

8.1.3 INTERSECT

If SaleCo's management wants to know which customer records are duplicated in the CUSTOMER and CUSTOMER_2 tables, the INTERSECT statement can be used to combine rows from two queries, returning only the rows that appear in both sets. The syntax for the INTERSECT statement is:

query INTERSECT *query*

To generate the list of duplicate customer records, you can use:

```
SELECT      CUS_LNAME, CUS_FNAME, CUS_INITIAL, CUS_AREACODE, CUS_PHONE
FROM        CUSTOMER
INTERSECT
SELECT      CUS_LNAME, CUS_FNAME, CUS_INITIAL, CUS_AREACODE, CUS_PHONE
FROM        CUSTOMER_2;
```

The INTERSECT statement can be used to generate additional useful customer information. For example, the following query returns the customer codes for all customers who are located in area code 615 and who have made purchases. (If a customer has made a purchase, there must be an invoice record for that customer.)

```
SELECT      CUS_CODE FROM CUSTOMER WHERE CUS_AREACODE = '615'
INTERSECT
SELECT      DISTINCT CUS_CODE FROM INVOICE;
```

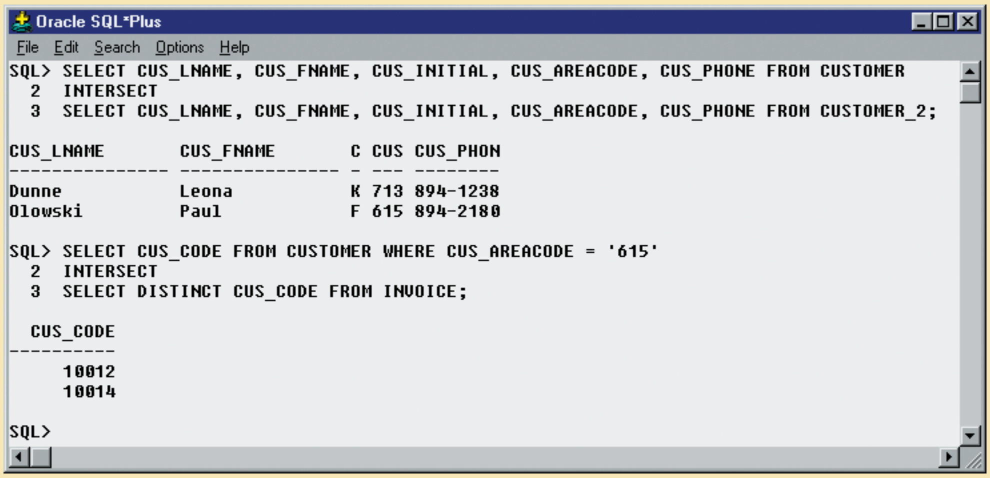
Figure 8.3 shows both sets of SQL statements and their output.

8.1.4 MINUS

The MINUS statement in SQL combines rows from two queries and returns only the rows that appear in the first set but not in the second. The syntax for the MINUS statement is:

query MINUS *query*

FIGURE 8.3 INTERSECT query results



```
SQL> SELECT CUS_LNAME, CUS_FNAME, CUS_INITIAL, CUS_AREACODE, CUS_PHONE FROM CUSTOMER
  2  INTERSECT
  3  SELECT CUS_LNAME, CUS_FNAME, CUS_INITIAL, CUS_AREACODE, CUS_PHONE FROM CUSTOMER_2;

CUS_LNAME       CUS_FNAME       C CUS CUS_PHON
--------------- --------------- - --- --------
Dunne           Leona           K 713 894-1238
Olowski         Paul            F 615 894-2180

SQL> SELECT CUS_CODE FROM CUSTOMER WHERE CUS_AREACODE = '615'
  2  INTERSECT
  3  SELECT DISTINCT CUS_CODE FROM INVOICE;

  CUS_CODE
----------
     10012
     10014

SQL>
```

NOTE

MS Access does not support the INTERSECT query, nor does it support other complex queries you will explore in this chapter. At least in some cases, Access might be able to give you the desired results if you use an alternative query format or procedure. For example, although Access does not support SQL triggers and stored procedures, you can use Visual Basic code to perform similar actions. However, the objective here is to show you how some important standard SQL features may be used.

For example, if the SaleCo managers want to know what customers in the CUSTOMER table are not found in the CUSTOMER_2 table, they can use:

```
SELECT    CUS_LNAME, CUS_FNAME, CUS_INITIAL, CUS_AREACODE, CUS_PHONE
FROM      CUSTOMER
MINUS
SELECT    CUS_LNAME, CUS_FNAME, CUS_INITIAL, CUS_AREACODE, CUS_PHONE
FROM      CUSTOMER_2;
```

If the managers want to know what customers in the CUSTOMER_2 table are not found in the CUSTOMER table, they merely switch the table designations:

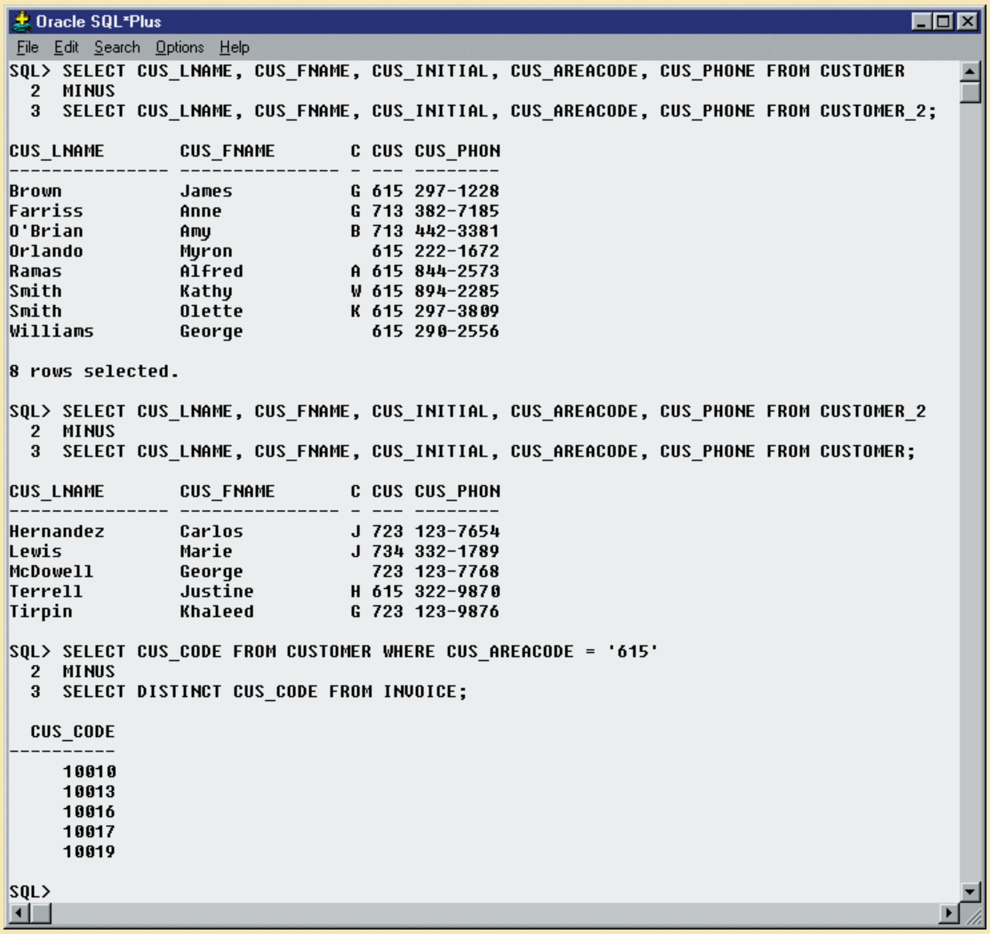
```
SELECT    CUS_LNAME, CUS_FNAME, CUS_INITIAL, CUS_AREACODE, CUS_PHONE
FROM      CUSTOMER_2
MINUS
SELECT    CUS_LNAME, CUS_FNAME, CUS_INITIAL, CUS_AREACODE, CUS_PHONE
FROM      CUSTOMER;
```

You can extract much useful information by combining MINUS with various clauses such as WHERE. For example, the following query returns the customer codes for all customers located in area code 615 minus the ones who have made purchases, leaving the customers in area code 615 who have not made purchases.

```
SELECT    CUS_CODE FROM CUSTOMER WHERE CUS_AREACODE = '615'
MINUS
SELECT    DISTINCT CUS_CODE FROM INVOICE;
```

Figure 8.4 shows the preceding three SQL statements and their output.

FIGURE 8.4 MINUS query results



```
SQL> SELECT CUS_LNAME, CUS_FNAME, CUS_INITIAL, CUS_AREACODE, CUS_PHONE FROM CUSTOMER
  2  MINUS
  3  SELECT CUS_LNAME, CUS_FNAME, CUS_INITIAL, CUS_AREACODE, CUS_PHONE FROM CUSTOMER_2;

CUS_LNAME       CUS_FNAME       C CUS CUS_PHON
--------------- --------------- - --- --------
Brown           James           G 615 297-1228
Farriss         Anne            G 713 382-7185
O'Brian         Amy             B 713 442-3381
Orlando         Myron             615 222-1672
Ramas           Alfred          A 615 844-2573
Smith           Kathy           W 615 894-2285
Smith           Olette          K 615 297-3809
Williams        George            615 290-2556

8 rows selected.

SQL> SELECT CUS_LNAME, CUS_FNAME, CUS_INITIAL, CUS_AREACODE, CUS_PHONE FROM CUSTOMER_2
  2  MINUS
  3  SELECT CUS_LNAME, CUS_FNAME, CUS_INITIAL, CUS_AREACODE, CUS_PHONE FROM CUSTOMER;

CUS_LNAME       CUS_FNAME       C CUS CUS_PHON
--------------- --------------- - --- --------
Hernandez       Carlos          J 723 123-7654
Lewis           Marie           J 734 332-1789
McDowell        George            723 123-7768
Terrell         Justine         H 615 322-9870
Tirpin          Khaleed         G 723 123-9876

SQL> SELECT CUS_CODE FROM CUSTOMER WHERE CUS_AREACODE = '615'
  2  MINUS
  3  SELECT DISTINCT CUS_CODE FROM INVOICE;

  CUS_CODE
----------
     10010
     10013
     10016
     10017
     10019

SQL>
```

NOTE

Some DBMS products do not support the INTERSECT or MINUS statements, while others might implement the difference relational operator in SQL as EXCEPT. Consult your DBMS manual to see if the statements illustrated here are supported by your DBMS.

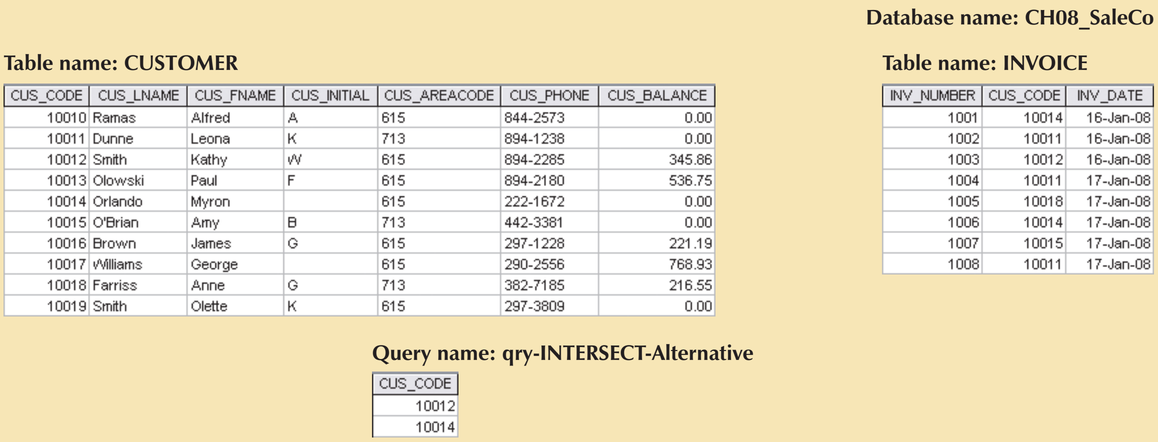
8.1.5 Syntax Alternatives

If your DBMS doesn't support the INTERSECT or MINUS statements, you can use the IN and NOT IN subqueries to obtain similar results. For example, the following query will produce the same results as the INTERSECT query shown in Section 8.1.3.

```
SELECT    CUS_CODE FROM CUSTOMER
WHERE     CUS_AREACODE = '615' AND
          CUS_CODE IN (SELECT DISTINCT CUS_CODE FROM INVOICE);
```

Figure 8.5 shows the use of the INTERSECT alternative.

FIGURE 8.5 INTERSECT alternative



Database name: CH08_SaleCo

Table name: CUSTOMER

CUS_CODE	CUS_LNAME	CUS_FNAME	CUS_INITIAL	CUS_AREACODE	CUS_PHONE	CUS_BALANCE
10010	Ramas	Alfred	A	615	844-2573	0.00
10011	Dunne	Leona	K	713	894-1238	0.00
10012	Smith	Kathy	W	615	894-2285	345.86
10013	Olowski	Paul	F	615	894-2180	536.75
10014	Orlando	Myron		615	222-1672	0.00
10015	O'Brian	Amy	B	713	442-3381	0.00
10016	Brown	James	G	615	297-1228	221.19
10017	Williams	George		615	290-2556	768.93
10018	Farriss	Anne	G	713	382-7185	216.55
10019	Smith	Olette	K	615	297-3809	0.00

Table name: INVOICE

INV_NUMBER	CUS_CODE	INV_DATE
1001	10014	16-Jan-08
1002	10011	16-Jan-08
1003	10012	16-Jan-08
1004	10011	17-Jan-08
1005	10018	17-Jan-08
1006	10014	17-Jan-08
1007	10015	17-Jan-08
1008	10011	17-Jan-08

Query name: qry-INTERSECT-Alternative

CUS_CODE
10012
10014

NOTE

MS Access will generate an input request for the CUS_AREACODE if you use apostrophes around the area code. (If you supply the 615 area code, the query will execute properly.) You can eliminate that problem by using standard double quotation marks, writing the WHERE clause in the second line of the preceding SQL statement as:

```
WHERE CUS_AREACODE = "615" AND
```

MS Access will also accept single quotation marks.

Using the same alternative to the MINUS statement, you can generate the output for the third MINUS query shown in Section 8.1.4 by using:

```
SELECT    CUS_CODE FROM CUSTOMER
WHERE     CUS_AREACODE = '615' AND
          CUS_CODE NOT IN (SELECT DISTINCT CUS_CODE FROM INVOICE);
```

The results of that query are shown in Figure 8.6. Note that the query output includes only the customers in area code 615 who have not made any purchases and, therefore, have not generated invoices.

8.2 SQL JOIN OPERATORS

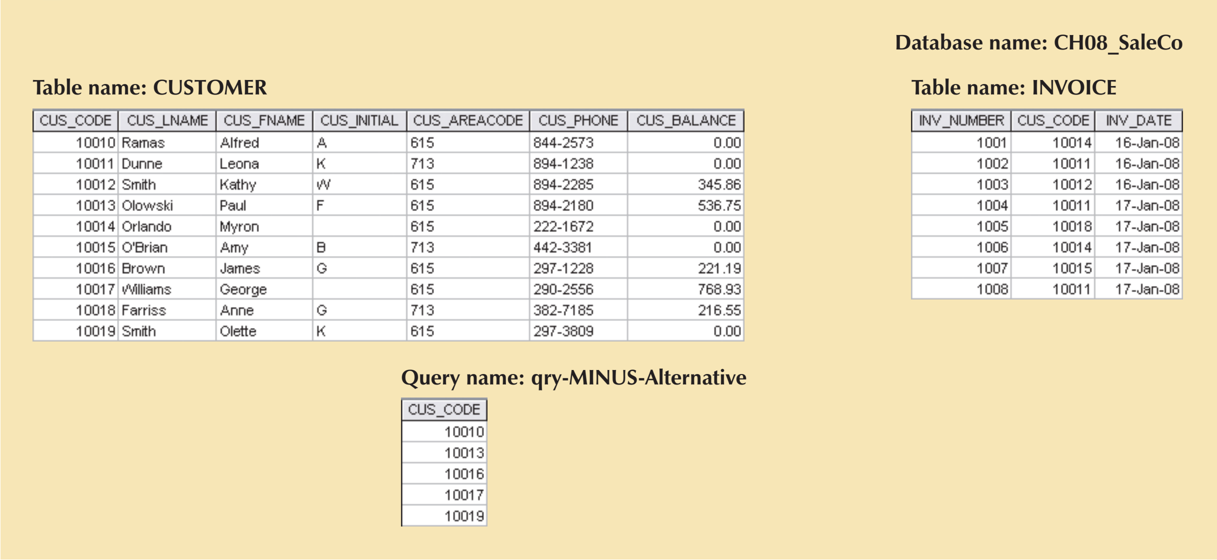
The relational join operation merges rows from two tables and returns the rows with one of the following conditions:

- Have common values in common columns (natural join).
- Meet a given join condition (equality or inequality).
- Have common values in common columns or have no matching values (outer join).

In Chapter 7, you learned how to use the SELECT statement in conjunction with the WHERE clause to join two or more tables. For example, you can join the PRODUCT and VENDOR tables through their common V_CODE by writing:

```
SELECT    P_CODE, P_DESCRIPT, P_PRICE, V_NAME
FROM      PRODUCT, VENDOR
WHERE     PRODUCT.V_CODE = VENDOR.V_CODE;
```

FIGURE 8.6 MINUS alternative



Database name: CH08_SaleCo

Table name: CUSTOMER

CUS_CODE	CUS_LNAME	CUS_FNAME	CUS_INITIAL	CUS_AREACODE	CUS_PHONE	CUS_BALANCE
10010	Ramas	Alfred	A	615	844-2573	0.00
10011	Dunne	Leona	K	713	894-1238	0.00
10012	Smith	Kathy	W	615	894-2285	345.86
10013	Olowski	Paul	F	615	894-2180	536.75
10014	Orlando	Myron		615	222-1672	0.00
10015	O'Brian	Amy	B	713	442-3381	0.00
10016	Brown	James	G	615	297-1228	221.19
10017	Williams	George		615	290-2556	768.93
10018	Farriss	Anne	G	713	382-7185	216.55
10019	Smith	Olette	K	615	297-3809	0.00

Table name: INVOICE

INV_NUMBER	CUS_CODE	INV_DATE
1001	10014	16-Jan-08
1002	10011	16-Jan-08
1003	10012	16-Jan-08
1004	10011	17-Jan-08
1005	10018	17-Jan-08
1006	10014	17-Jan-08
1007	10015	17-Jan-08
1008	10011	17-Jan-08

Query name: qry-MINUS-Alternative

CUS_CODE
10010
10013
10016
10017
10019

The preceding SQL join syntax is sometimes referred to as an "old-style" join. Note that the FROM clause contains the tables being joined and that the WHERE clause contains the condition(s) used to join the tables.

Note the following points about the preceding query:

- The FROM clause indicates which tables are to be joined. If three or more tables are included, the join operation takes place two tables at a time, starting from left to right. For example, if you are joining tables T1, T2, and T3, the first join is table T1 with T2; the results of that join are then joined to table T3.
- The join condition in the WHERE clause tells the SELECT statement which rows will be returned. In this case, the SELECT statement returns all rows for which the V_CODE values in the PRODUCT and VENDOR tables are equal.
- The number of join conditions is always equal to the number of tables being joined minus one. For example, if you join three tables (T1, T2, and T3), you will have two join conditions (j1 and j2). All join conditions are connected through an AND logical operator. The first join condition (j1) defines the join criteria for T1 and T2. The second join condition (j2) defines the join criteria for the output of the first join and T3.
- Generally, the join condition will be an equality comparison of the primary key in one table and the related foreign key in the second table.

Join operations can be classified as inner joins and outer joins. The **inner join** is the traditional join in which only rows that meet a given criteria are selected. The join criteria can be an equality condition (also called a natural join or an equijoin) or an inequality condition (also called a theta join). An **outer join** returns not only the matching rows, but also the rows with unmatched attribute values for one table or both tables to be joined. The SQL standard also introduces a special type of join that returns the same result as the Cartesian product of two sets or tables.

In this section, you will learn various ways to express join operations that meet the ANSI SQL standard. These are outlined in Table 8.1. It is useful to remember that not all DBMS vendors provide the same level of SQL support and that some do not support the join styles shown in this section. Oracle 10g is used to demonstrate the use of the following queries. Refer to your DBMS manual if you are using a different DBMS.

TABLE 8.1 SQL Join Expression Styles

JOIN CLASSIFICATION	JOIN TYPE	SQL SYNTAX EXAMPLE	DESCRIPTION

CROSS	CROSS JOIN	SELECT *
		FROM T1, T2
			Returns the Cartesian product of T1 and T2 (old style).

		SELECT *
		FROM T1 CROSS JOIN T2
			Returns the Cartesian product of T1 and T2.

INNER	Old-Style JOIN	SELECT *
		FROM T1, T2
		WHERE T1.C1=T2.C1
			Returns only the rows that meet the join condition in the WHERE clause (old style). Only rows with matching values are selected.

	NATURAL JOIN	SELECT *
		FROM T1 NATURAL JOIN T2
			Returns only the rows with matching values in the matching columns. The matching columns must have the same names and similar data types.

	JOIN USING	SELECT *
		FROM T1 JOIN T2 USING (C1)
			Returns only the rows with matching values in the columns indicated in the USING clause.

	JOIN ON	SELECT *
		FROM T1 JOIN T2
		ON T1.C1=T2.C1
			Returns only the rows that meet the join condition indicated in the ON clause.

OUTER	LEFT JOIN	SELECT *
		FROM T1 LEFT OUTER JOIN T2
		ON T1.C1=T2.C1
			Returns rows with matching values and includes all rows from the left table (T1) with unmatched values.

	RIGHT JOIN	SELECT *
		FROM T1 RIGHT OUTER JOIN T2
		ON T1.C1=T2.C1
			Returns rows with matching values and includes all rows from the right table (T2) with unmatched values.

	FULL JOIN	SELECT *
		FROM T1 FULL OUTER JOIN T2
		ON T1.C1=T2.C1
			Returns rows with matching values and includes all rows from both tables (T1 and T2) with unmatched values.

8.2.1 Cross Join

A **cross join** performs a relational product (also known as the Cartesian product) of two tables. The cross join syntax is:

SELECT *column-list* FROM *table1* CROSS JOIN *table2*

For example,

```
SELECT * FROM INVOICE CROSS JOIN LINE;
```

performs a cross join of the INVOICE and LINE tables. That CROSS JOIN query generates 144 rows. (There were 8 invoice rows and 18 line rows, thus yielding 8 × 18 = 144 rows.)

You can also perform a cross join that yields only specified attributes. For example, you can specify:

```
SELECT      INVOICE.INV_NUMBER, CUS_CODE, INV_DATE, P_CODE
FROM        INVOICE CROSS JOIN LINE;
```

The results generated through that SQL statement can also be generated by using the following syntax:

```
SELECT      INVOICE.INV_NUMBER, CUS_CODE, INV_DATE, P_CODE
FROM        INVOICE, LINE;
```

8.2.2 Natural Join

Recall from Chapter 3 that a natural join returns all rows with matching values in the matching columns and eliminates duplicate columns. That style of query is used when the tables share one or more common attributes with common names. The natural join syntax is:

SELECT *column-list* FROM *table1* NATURAL JOIN *table2*

The natural join will perform the following tasks:

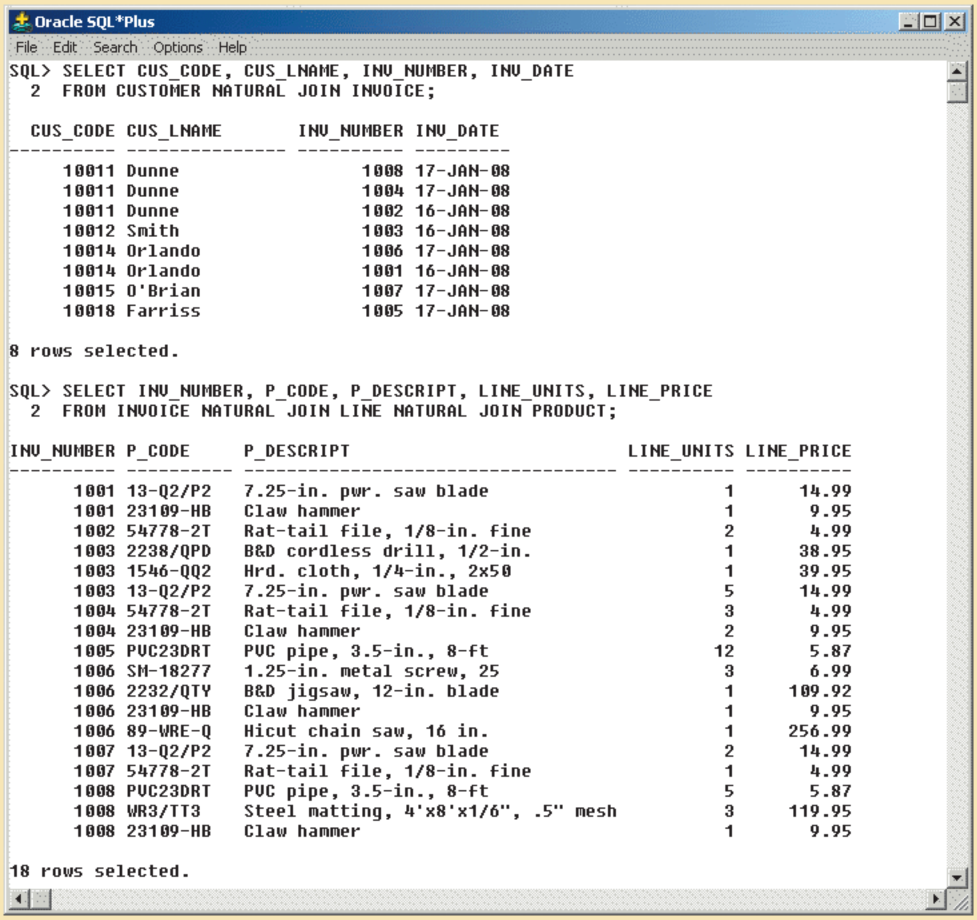
- Determine the common attribute(s) by looking for attributes with identical names and compatible data types.
- Select only the rows with common values in the common attribute(s).
- If there are no common attributes, return the relational product of the two tables.

The following example performs a natural join of the CUSTOMER and INVOICE tables and returns only selected attributes:

```
SELECT    CUS_CODE, CUS_LNAME, INV_NUMBER, INV_DATE
FROM      CUSTOMER NATURAL JOIN INVOICE;
```

The SQL code and its results are shown at the top of Figure 8.7.

FIGURE 8.7 NATURAL JOIN results



```
SQL> SELECT CUS_CODE, CUS_LNAME, INV_NUMBER, INV_DATE
  2  FROM CUSTOMER NATURAL JOIN INVOICE;

  CUS_CODE CUS_LNAME       INV_NUMBER INV_DATE
---------- --------------- ---------- ---------
     10011 Dunne                 1008 17-JAN-08
     10011 Dunne                 1004 17-JAN-08
     10011 Dunne                 1002 16-JAN-08
     10012 Smith                 1003 16-JAN-08
     10014 Orlando               1006 17-JAN-08
     10014 Orlando               1001 16-JAN-08
     10015 O'Brian               1007 17-JAN-08
     10018 Farriss               1005 17-JAN-08

8 rows selected.

SQL> SELECT INV_NUMBER, P_CODE, P_DESCRIPT, LINE_UNITS, LINE_PRICE
  2  FROM INVOICE NATURAL JOIN LINE NATURAL JOIN PRODUCT;

INV_NUMBER P_CODE     P_DESCRIPT                          LINE_UNITS LINE_PRICE
---------- ---------- ----------------------------------- ---------- ----------
      1001 13-Q2/P2   7.25-in. pwr. saw blade                      1      14.99
      1001 23109-HB   Claw hammer                                  1       9.95
      1002 54778-2T   Rat-tail file, 1/8-in. fine                  2       4.99
      1003 2238/QPD   B&D cordless drill, 1/2-in.                  1      38.95
      1003 1546-QQ2   Hrd. cloth, 1/4-in., 2x50                    1      39.95
      1003 13-Q2/P2   7.25-in. pwr. saw blade                      5      14.99
      1004 54778-2T   Rat-tail file, 1/8-in. fine                  3       4.99
      1004 23109-HB   Claw hammer                                  2       9.95
      1005 PVC23DRT   PVC pipe, 3.5-in., 8-ft                     12       5.87
      1006 SM-18277   1.25-in. metal screw, 25                     3       6.99
      1006 2232/QTY   B&D jigsaw, 12-in. blade                     1     109.92
      1006 23109-HB   Claw hammer                                  1       9.95
      1006 89-WRE-Q   Hicut chain saw, 16 in.                      1     256.99
      1007 13-Q2/P2   7.25-in. pwr. saw blade                      2      14.99
      1007 54778-2T   Rat-tail file, 1/8-in. fine                  1       4.99
      1008 PVC23DRT   PVC pipe, 3.5-in., 8-ft                      5       5.87
      1008 WR3/TT3    Steel matting, 4'x8'x1/6", .5" mesh          3     119.95
      1008 23109-HB   Claw hammer                                  1       9.95

18 rows selected.
```

You are not limited to two tables when performing a natural join. For example, you can perform a natural join of the INVOICE, LINE, and PRODUCT tables and project only selected attributes by writing:

```
SELECT      INV_NUMBER, P_CODE, P_DESCRIPT, LINE_UNITS, LINE_PRICE
FROM        INVOICE NATURAL JOIN LINE NATURAL JOIN PRODUCT;
```

The SQL code and its results are shown at the bottom of Figure 8.7.

One important difference between the natural join and the “old-style” join syntax is that the natural join does not require the use of a table qualifier for the common attributes. In the first natural join example, you projected CUS_CODE. However, the projection did not require any table qualifier, even though the CUS_CODE attribute appeared in both CUSTOMER and INVOICE tables. The same can be said of the INV_NUMBER attribute in the second natural join example.

8.2.3 Join USING Clause

A second way to express a join is through the USING keyword. That query returns only the rows with matching values in the column indicated in the USING clause—and that column must exist in both tables. The syntax is:

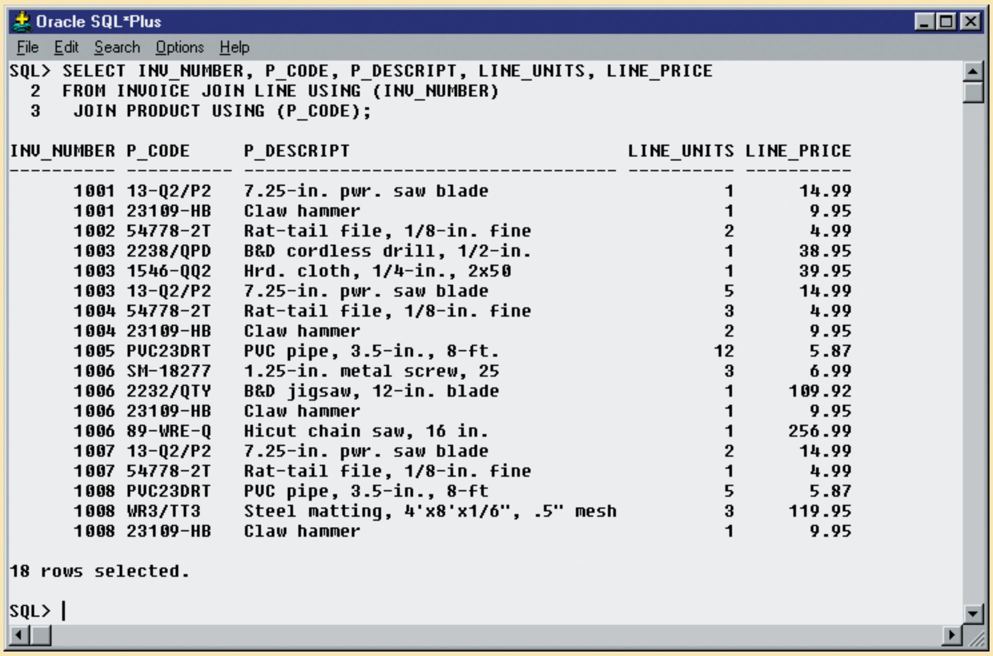
SELECT *column-list* FROM *table1* JOIN *table2* USING (*common-column*)

To see the JOIN USING query in action, let’s perform a join of the INVOICE and LINE tables by writing:

```
SELECT      INV_NUMBER, P_CODE, P_DESCRIPT, LINE_UNITS, LINE_PRICE
FROM        INVOICE JOIN LINE USING (INV_NUMBER) JOIN PRODUCT USING (P_CODE);
```

The SQL statement produces the results shown in Figure 8.8.

FIGURE 8.8 JOIN USING results



```
SQL> SELECT INU_NUMBER, P_CODE, P_DESCRIPT, LINE_UNITS, LINE_PRICE
  2  FROM INUOICE JOIN LINE USING (INU_NUMBER)
  3   JOIN PRODUCT USING (P_CODE);

INU_NUMBER P_CODE     P_DESCRIPT                          LINE_UNITS LINE_PRICE
---------- ---------- ----------------------------------- ---------- ----------
      1001 13-Q2/P2   7.25-in. pwr. saw blade                      1      14.99
      1001 23109-HB   Claw hammer                                  1       9.95
      1002 54778-2T   Rat-tail file, 1/8-in. fine                  2       4.99
      1003 2238/QPD   B&D cordless drill, 1/2-in.                  1      38.95
      1003 1546-QQ2   Hrd. cloth, 1/4-in., 2x50                    1      39.95
      1003 13-Q2/P2   7.25-in. pwr. saw blade                      5      14.99
      1004 54778-2T   Rat-tail file, 1/8-in. fine                  3       4.99
      1004 23109-HB   Claw hammer                                  2       9.95
      1005 PUC23DRT   PUC pipe, 3.5-in., 8-ft.                    12       5.87
      1006 SM-18277   1.25-in. metal screw, 25                     3       6.99
      1006 2232/QTY   B&D jigsaw, 12-in. blade                     1     109.92
      1006 23109-HB   Claw hammer                                  1       9.95
      1006 89-WRE-Q   Hicut chain saw, 16 in.                      1     256.99
      1007 13-Q2/P2   7.25-in. pwr. saw blade                      2      14.99
      1007 54778-2T   Rat-tail file, 1/8-in. fine                  1       4.99
      1008 PUC23DRT   PUC pipe, 3.5-in., 8-ft                      5       5.87
      1008 WR3/TT3    Steel matting, 4'x8'x1/6", .5" mesh          3     119.95
      1008 23109-HB   Claw hammer                                  1       9.95

18 rows selected.

SQL> |
```

As was the case with the NATURAL JOIN command, the JOIN USING operand does not require table qualifiers. As a matter of fact, Oracle will return an error if you specify the table name in the USING clause.

8.2.4 JOIN ON Clause

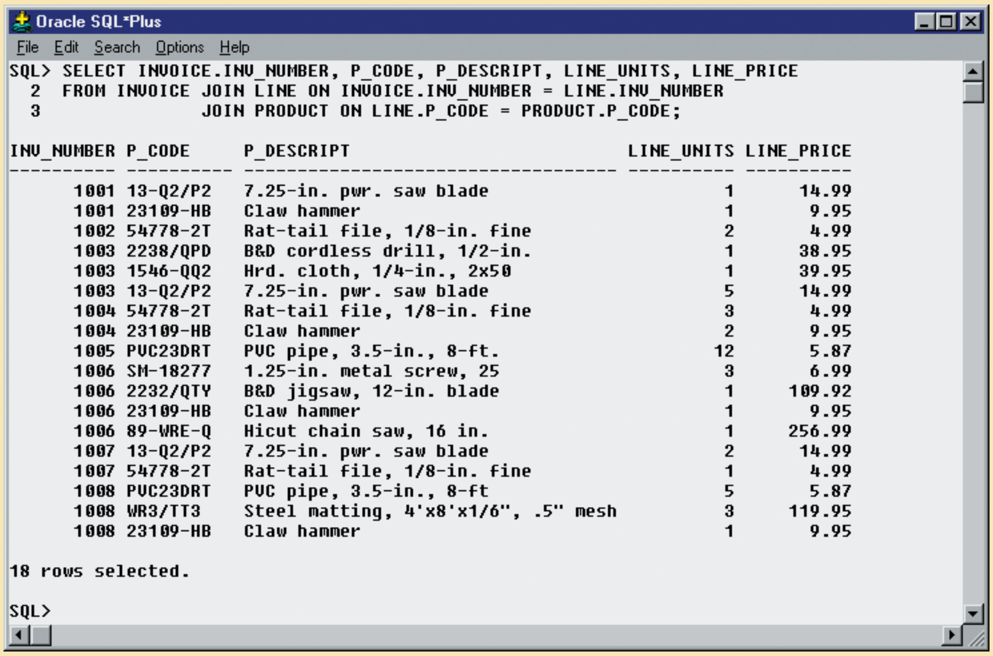
The previous two join styles used common attribute names in the joining tables. Another way to express a join when the tables have no common attribute names is to use the JOIN ON operand. That query will return only the rows that meet the indicated join condition. The join condition will typically include an equality comparison expression of two columns. (The columns may or may not share the same name but, obviously, must have comparable data types.) The syntax is:

SELECT *column-list* FROM *table1* JOIN *table2* ON *join-condition*

The following example performs a join of the INVOICE and LINE tables, using the ON clause. The result is shown in Figure 8.9.

```
SELECT      INVOICE.INV_NUMBER, P_CODE, P_DESCRIPT, LINE_UNITS, LINE_PRICE
FROM        INVOICE JOIN LINE ON INVOICE.INV_NUMBER = LINE.INV_NUMBER
            JOIN PRODUCT ON LINE.P_CODE = PRODUCT.P_CODE;
```

FIGURE 8.9 JOIN ON results



```
SQL> SELECT INVOICE.INV_NUMBER, P_CODE, P_DESCRIPT, LINE_UNITS, LINE_PRICE
  2  FROM INVOICE JOIN LINE ON INVOICE.INV_NUMBER = LINE.INV_NUMBER
  3                 JOIN PRODUCT ON LINE.P_CODE = PRODUCT.P_CODE;

INV_NUMBER P_CODE     P_DESCRIPT                          LINE_UNITS LINE_PRICE
---------- ---------- ----------------------------------- ---------- ----------
      1001 13-Q2/P2   7.25-in. pwr. saw blade                      1      14.99
      1001 23109-HB   Claw hammer                                  1       9.95
      1002 54778-2T   Rat-tail file, 1/8-in. fine                  2       4.99
      1003 2238/QPD   B&D cordless drill, 1/2-in.                  1      38.95
      1003 1546-QQ2   Hrd. cloth, 1/4-in., 2x50                    1      39.95
      1003 13-Q2/P2   7.25-in. pwr. saw blade                      5      14.99
      1004 54778-2T   Rat-tail file, 1/8-in. fine                  3       4.99
      1004 23109-HB   Claw hammer                                  2       9.95
      1005 PVC23DRT   PVC pipe, 3.5-in., 8-ft.                    12       5.87
      1006 SM-18277   1.25-in. metal screw, 25                     3       6.99
      1006 2232/QTY   B&D jigsaw, 12-in. blade                     1     109.92
      1006 23109-HB   Claw hammer                                  1       9.95
      1006 89-WRE-Q   Hicut chain saw, 16 in.                      1     256.99
      1007 13-Q2/P2   7.25-in. pwr. saw blade                      2      14.99
      1007 54778-2T   Rat-tail file, 1/8-in. fine                  1       4.99
      1008 PVC23DRT   PVC pipe, 3.5-in., 8-ft                      5       5.87
      1008 WR3/TT3    Steel matting, 4'x8'x1/6", .5" mesh          3     119.95
      1008 23109-HB   Claw hammer                                  1       9.95

18 rows selected.

SQL>
```

Note that unlike the NATURAL JOIN and the JOIN USING operands, the JOIN ON clause requires a table qualifier for the common attributes. If you do not specify the table qualifier, you will get a "column ambiguously defined" error message.

Keep in mind that the JOIN ON syntax lets you perform a join even when the tables do not share a common attribute name. For example, to generate a list of all employees with the managers' names, you can use the following (recursive) query:

```
SELECT      E.EMP_MGR, M.EMP_LNAME, E.EMP_NUM, E.EMP_LNAME
FROM        EMP E JOIN EMP M ON E.EMP_MGR = M.EMP_NUM
ORDER BY    E.EMP_MGR;
```

8.2.5 Outer Joins

An outer join returns not only the rows matching the join condition (that is, rows with matching values in the common columns), but also the rows with unmatched values. The ANSI standard defines three types of outer joins: left, right, and full. The left and right designations reflect the order in which the tables are processed by the DBMS. Remember that join operations take place two tables at a time. The first table named in the FROM clause will be the left side, and the second table named will be the right side. If three or more tables are being joined, the result of joining the first two tables becomes the left side, and the third table becomes the right side.

The left outer join returns not only the rows matching the join condition (that is, rows with matching values in the common column), but also the rows in the left side table with unmatched values in the right side table. The syntax is:

```
SELECT      column-list
FROM        table1 LEFT [OUTER] JOIN table2 ON join-condition
```

For example, the following query lists the product code, vendor code, and vendor name for all products and includes those vendors with no matching products:

```
SELECT      P_CODE, VENDOR.V_CODE, V_NAME
FROM        VENDOR LEFT JOIN PRODUCT ON VENDOR.V_CODE = PRODUCT.V_CODE;
```

The preceding SQL code and its results are shown in Figure 8.10.

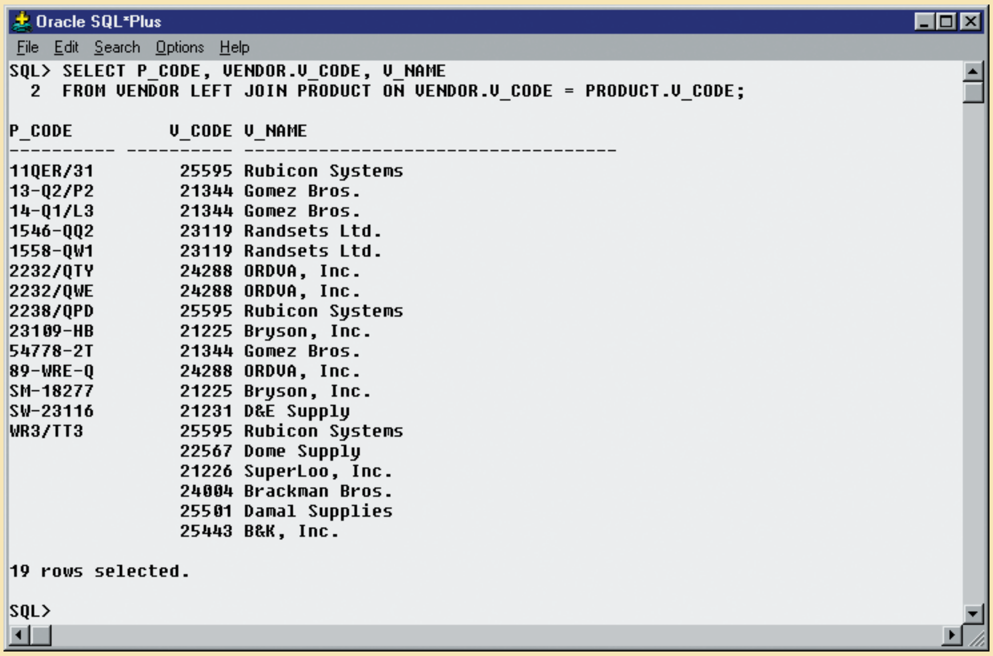
The right outer join returns not only the rows matching the join condition (that is, rows with matching values in the common column), but also the rows in the right side table with unmatched values in the left side table. The syntax is:

```
SELECT      column-list
FROM        table1 RIGHT [OUTER] JOIN table2 ON join-condition
```

For example, the following query lists the product code, vendor code, and vendor name for all products and also includes those products that do not have a matching vendor code:

```
SELECT      P_CODE, VENDOR.V_CODE, V_NAME
FROM        VENDOR RIGHT JOIN PRODUCT ON VENDOR.V_CODE = PRODUCT.V_CODE;
```

FIGURE 8.10 **LEFT JOIN results**



```
SQL> SELECT P_CODE, VENDOR.V_CODE, V_NAME
  2  FROM VENDOR LEFT JOIN PRODUCT ON VENDOR.V_CODE = PRODUCT.V_CODE;

P_CODE         V_CODE V_NAME
---------- ---------- -----------------------------------
11QER/31        25595 Rubicon Systems
13-Q2/P2        21344 Gomez Bros.
14-Q1/L3        21344 Gomez Bros.
1546-QQ2        23119 Randsets Ltd.
1558-QW1        23119 Randsets Ltd.
2232/QTY        24288 ORDVA, Inc.
2232/QWE        24288 ORDVA, Inc.
2238/QPD        25595 Rubicon Systems
23109-HB        21225 Bryson, Inc.
54778-2T        21344 Gomez Bros.
89-WRE-Q        24288 ORDVA, Inc.
SM-18277        21225 Bryson, Inc.
SW-23116        21231 D&E Supply
WR3/TT3         25595 Rubicon Systems
                22567 Dome Supply
                21226 SuperLoo, Inc.
                24004 Brackman Bros.
                25501 Damal Supplies
                25443 B&K, Inc.

19 rows selected.

SQL>
```

The SQL code and its output are shown in Figure 8.11.

The full outer join returns not only the rows matching the join condition (that is, rows with matching values in the common column), but also all of the rows with unmatched values in either side table. The syntax is:

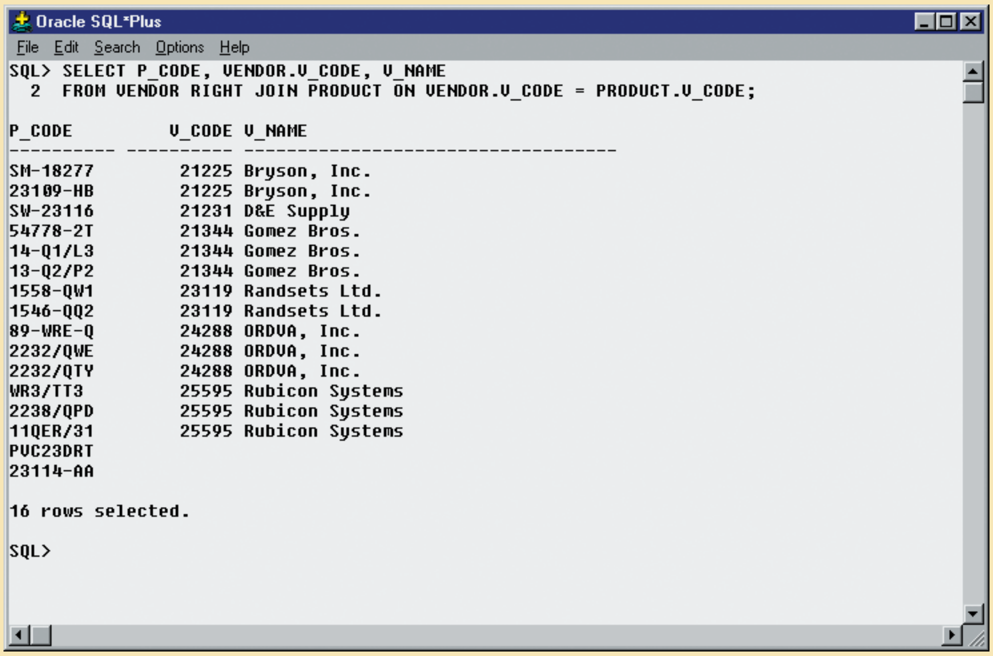
SELECT column-list
FROM *table1* FULL [OUTER] JOIN *table2* ON *join-condition*

For example, the following query lists the product code, vendor code, and vendor name for all products and includes all product rows (products without matching vendors) as well as all vendor rows (vendors without matching products).

SELECT P_CODE, VENDOR.V_CODE, V_NAME
FROM VENDOR FULL JOIN PRODUCT ON VENDOR.V_CODE = PRODUCT.V_CODE;

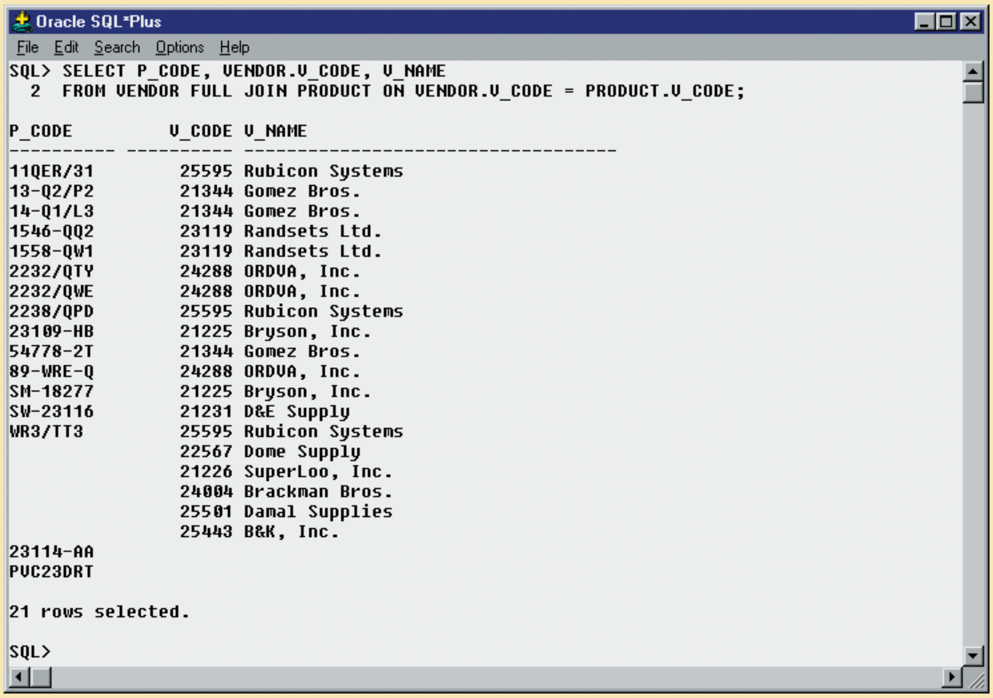
The SQL code and its results are shown in Figure 8.12.

FIGURE 8.11 RIGHT JOIN results



```
SQL> SELECT P_CODE, VENDOR.V_CODE, V_NAME
  2  FROM VENDOR RIGHT JOIN PRODUCT ON VENDOR.V_CODE = PRODUCT.V_CODE;

P_CODE         V_CODE V_NAME
---------- ---------- -----------------------------------
SM-18277        21225 Bryson, Inc.
23109-HB        21225 Bryson, Inc.
SW-23116        21231 D&E Supply
54778-2T        21344 Gomez Bros.
14-Q1/L3        21344 Gomez Bros.
13-Q2/P2        21344 Gomez Bros.
1558-QW1        23119 Randsets Ltd.
1546-QQ2        23119 Randsets Ltd.
89-WRE-Q        24288 ORDVA, Inc.
2232/QWE        24288 ORDVA, Inc.
2232/QTY        24288 ORDVA, Inc.
WR3/TT3         25595 Rubicon Systems
2238/QPD        25595 Rubicon Systems
11QER/31        25595 Rubicon Systems
PVC23DRT
23114-AA

16 rows selected.

SQL>
```

FIGURE 8.12 FULL JOIN results



```
SQL> SELECT P_CODE, VENDOR.V_CODE, V_NAME
  2  FROM VENDOR FULL JOIN PRODUCT ON VENDOR.V_CODE = PRODUCT.V_CODE;

P_CODE         V_CODE V_NAME
---------- ---------- -----------------------------------
11QER/31        25595 Rubicon Systems
13-Q2/P2        21344 Gomez Bros.
14-Q1/L3        21344 Gomez Bros.
1546-QQ2        23119 Randsets Ltd.
1558-QW1        23119 Randsets Ltd.
2232/QTY        24288 ORDVA, Inc.
2232/QWE        24288 ORDVA, Inc.
2238/QPD        25595 Rubicon Systems
23109-HB        21225 Bryson, Inc.
54778-2T        21344 Gomez Bros.
89-WRE-Q        24288 ORDVA, Inc.
SM-18277        21225 Bryson, Inc.
SW-23116        21231 D&E Supply
WR3/TT3         25595 Rubicon Systems
                22567 Dome Supply
                21226 SuperLoo, Inc.
                24004 Brackman Bros.
                25501 Damal Supplies
                25443 B&K, Inc.
23114-AA
PVC23DRT

21 rows selected.

SQL>
```

8.3 SUBQUERIES AND CORRELATED QUERIES

The use of joins in a relational database allows you to get information from two or more tables. For example, the following query allows you to get the customers' data with their respective invoices by joining the CUSTOMER and INVOICE tables.

```
SELECT     INV_NUMBER, INVOICE.CUS_CODE, CUS_LNAME, CUS_FNAME
FROM       CUSTOMER, INVOICE
WHERE      CUSTOMER.CUS_CODE = INVOICE.CUS_CODE;
```

In the previous query, the data from both tables (CUSTOMER and INVOICE) are processed at once, matching rows with shared CUS_CODE values.

However, it is often necessary to process data based on *other* processed data. Suppose, for example, you want to generate a list of vendors who provide products. (Recall that not all vendors in the VENDOR table have provided products—some of them are only *potential* vendors.) In Chapter 7, you learned that you could generate such a list by writing the following query:

```
SELECT     V_CODE, V_NAME FROM VENDOR
WHERE      V_CODE NOT IN (SELECT V_CODE FROM PRODUCT);
```

Similarly, to generate a list of all products with a price greater than or equal to the average product price, you can write the following query:

```
SELECT     P_CODE, P_PRICE FROM PRODUCT
WHERE      P_PRICE >= (SELECT AVG(P_PRICE) FROM PRODUCT);
```

In both of those cases, you needed to get information that was not previously known:

- What vendors provide products?
- What is the average price of all products?

In both cases, you used a subquery to generate the required information that could then be used as input for the originating query.

You learned how to use subqueries in Chapter 7; let's review their basic characteristics:

- A subquery is a query (SELECT statement) inside a query.
- A subquery is normally expressed inside parentheses.
- The first query in the SQL statement is known as the outer query.
- The query inside the SQL statement is known as the inner query.
- The inner query is executed first.
- The output of an inner query is used as the input for the outer query.
- The entire SQL statement is sometimes referred to as a nested query.

In this section, you learn more about the practical use of subqueries. You already know that a subquery is based on the use of the SELECT statement to return one or more values to another query. But subqueries have a wide range of uses. For example, you can use a subquery within an SQL data manipulation language (DML) statement (such as INSERT, UPDATE, or DELETE) where a value or a list of values (such as multiple vendor codes or a table) is expected. Table 8.2 uses simple examples to summarize the use of SELECT subqueries in DML statements.

TABLE 8.2 SELECT Subquery Examples

SELECT SUBQUERY EXAMPLES	EXPLANATION

INSERT INTO PRODUCT
SELECT * FROM P;
	Inserts all rows from Table P into the PRODUCT table. Both tables must have the same attributes. The subquery returns all rows from Table P.

UPDATE PRODUCT
SET P_PRICE = (SELECT AVG(P_PRICE)
FROM PRODUCT)
WHERE V_CODE IN (SELECT V_CODE
FROM VENDOR
WHERE V_AREACODE = '615')
	Updates the product price to the average product price, but only for the products that are provided by vendors who have an area code equal to 615. The first subquery returns the average price; the second subquery returns the list of vendors with an area code equal to 615.

DELETE FROM PRODUCT
WHERE V_CODE IN (SELECT V_CODE
FROM VENDOR
WHERE V_AREACODE = '615')
	Deletes the PRODUCT table rows that are provided by vendors with area code equal to 615. The subquery returns the list of vendors codes with an area code equal to 615.

Using the examples shown in Table 8.2, note that the subquery is always at the right side of a comparison or assigning expression. Also, a subquery can return one value or multiple values. To be precise, the subquery can return:

- *One single value (one column and one row).* This subquery is used anywhere a single value is expected, as in the right side of a comparison expression (such as in the preceding UPDATE example when you assign the average price to the product's price). Obviously, when you assign a value to an attribute, that value is a single value, not a list of values. Therefore, the subquery must return only one value (one column, one row). If the query returns multiple values, the DBMS will generate an error.
- *A list of values (one column and multiple rows).* This type of subquery is used anywhere a list of values is expected, such as when using the IN clause (that is, when comparing the vendor code to a list of vendors). Again, in this case, there is only one column of data with multiple value instances. This type of subquery is used frequently in combination with the IN operator in a WHERE conditional expression.
- *A virtual table (multicolumn, multirow set of values).* This type of subquery can be used anywhere a table is expected, such as when using the FROM clause. You will see this type of query later in this chapter.

It's important to note that a subquery can return no values at all; it is a NULL. In such cases, the output of the outer query might result in an error or a null empty set, depending where the subquery is used (in a comparison, an expression, or a table set).

In the following sections, you will learn how to write subqueries within the SELECT statement to retrieve data from the database.

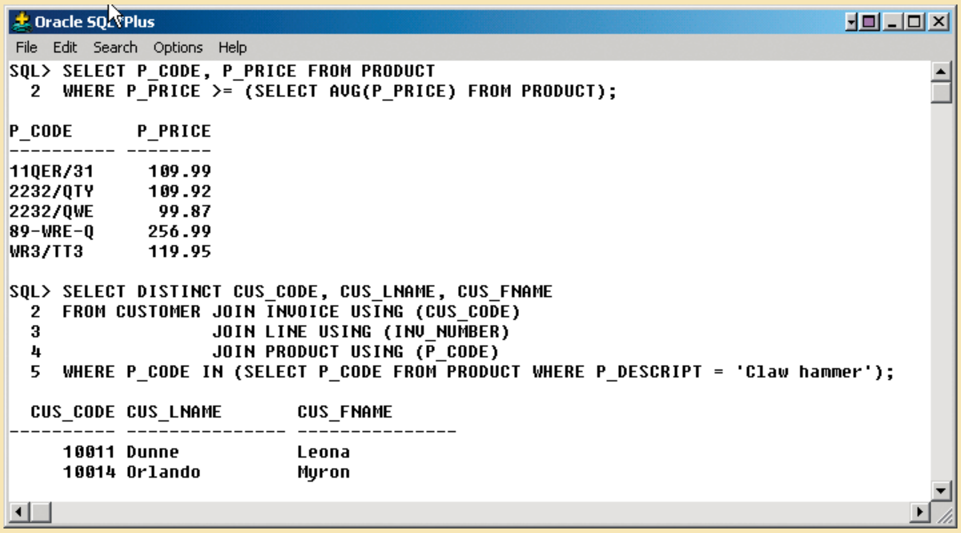
8.3.1 WHERE SUBQUERIES

The most common type of subquery uses an inner SELECT subquery on the right side of a WHERE comparison expression. For example, to find all products with a price greater than or equal to the average product price, you write the following query:

```
SELECT      P_CODE, P_PRICE FROM PRODUCT
WHERE       P_PRICE >= (SELECT AVG(P_PRICE) FROM PRODUCT);
```

The output of the preceding query is shown in Figure 8.13. Note that this type of query, when used in a >, <, =, >=, or <= conditional expression, requires a subquery that returns only one single value (one column, one row). The value generated by the subquery must be of a "comparable" data type; if the attribute to the left of the comparison symbol is a character type, the subquery must return a character string. Also, if the query returns more than a single value, the DBMS will generate an error.

FIGURE 8.13 WHERE subquery example



```
SQL> SELECT P_CODE, P_PRICE FROM PRODUCT
  2  WHERE P_PRICE >= (SELECT AVG(P_PRICE) FROM PRODUCT);

P_CODE      P_PRICE
---------- --------
11QER/31     109.99
2232/QTY     109.92
2232/QWE      99.87
89-WRE-Q     256.99
WR3/TT3      119.95

SQL> SELECT DISTINCT CUS_CODE, CUS_LNAME, CUS_FNAME
  2  FROM CUSTOMER JOIN INVOICE USING (CUS_CODE)
  3                JOIN LINE USING (INV_NUMBER)
  4                JOIN PRODUCT USING (P_CODE)
  5  WHERE P_CODE IN (SELECT P_CODE FROM PRODUCT WHERE P_DESCRIPT = 'Claw hammer');

  CUS_CODE CUS_LNAME       CUS_FNAME
---------- --------------- ---------------
     10011 Dunne           Leona
     10014 Orlando         Myron
```

Subqueries can also be used in combination with joins. For example, the following query lists all of the customers who ordered the product "Claw hammer":

```
SELECT      DISTINCT CUS_CODE, CUS_LNAME, CUS_FNAME
FROM        CUSTOMER JOIN INVOICE USING (CUS_CODE)
                     JOIN LINE USING (INV_NUMBER)
                     JOIN PRODUCT USING (P_CODE)
WHERE       P_CODE = (SELECT P_CODE FROM PRODUCT WHERE P_DESCRIPT = 'Claw hammer');
```

The result of that query is also shown in Figure 8.13.

In the preceding example, the inner query finds the P_CODE for the product "Claw hammer." The P_CODE is then used to restrict the selected rows to only those where the P_CODE in the LINE table matches the P_CODE for "Claw hammer." Note that the previous query could have been written this way:

```
SELECT      DISTINCT CUS_CODE, CUS_LNAME, CUS_FNAME
FROM        CUSTOMER JOIN INVOICE USING (CUS_CODE)
                     JOIN LINE USING (INV_NUMBER)
                     JOIN PRODUCT USING (P_CODE)
WHERE       P_DESCRIPT = 'Claw hammer';
```

But what happens if the original query encounters the "Claw hammer" string in more than one product description? You get an error message. To compare one value to a list of values, you must use an IN operand, as shown in the next section.

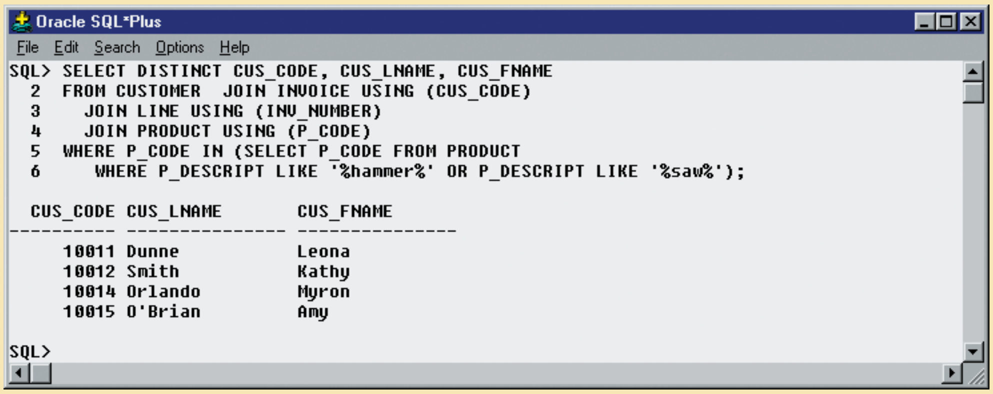
8.3.2 IN Subqueries

What would you do if you wanted to find all customers who purchased a "hammer" or any kind of saw or saw blade? Note that the product table has two different types of hammers: "Claw hammer" and "Sledge hammer." Also note that there are multiple occurrences of products that contain "saw" in their product descriptions. There are saw blades, jigsaws, and so on. In such cases, you need to compare the P_CODE not to one product code (single value), but to a list of product code values. When you want to compare a single attribute to a list of values, you use the IN operator. When the P_CODE values are not known beforehand but they can be derived using a query, you must use an IN subquery. The following example lists all customers who have purchased hammers, saws, or saw blades.

```
SELECT     DISTINCT CUS_CODE, CUS_LNAME, CUS_FNAME
FROM       CUSTOMER JOIN INVOICE USING (CUS_CODE)
                    JOIN LINE USING (INV_NUMBER)
                    JOIN PRODUCT USING (P_CODE)
WHERE      P_CODE IN (SELECT    P_CODE FROM PRODUCT
                    WHERE     P_DESCRIPT LIKE '%hammer%'
                    OR        P_DESCRIPT LIKE '%saw%');
```

The result of that query is shown in Figure 8.14.

FIGURE 8.14 IN subquery example



```
SQL> SELECT DISTINCT CUS_CODE, CUS_LNAME, CUS_FNAME
  2  FROM CUSTOMER  JOIN INVOICE USING (CUS_CODE)
  3    JOIN LINE USING (INV_NUMBER)
  4    JOIN PRODUCT USING (P_CODE)
  5  WHERE P_CODE IN (SELECT P_CODE FROM PRODUCT
  6     WHERE P_DESCRIPT LIKE '%hammer%' OR P_DESCRIPT LIKE '%saw%');

  CUS_CODE CUS_LNAME       CUS_FNAME
---------- --------------- ---------------
     10011 Dunne           Leona
     10012 Smith           Kathy
     10014 Orlando         Myron
     10015 O'Brian         Amy

SQL>
```

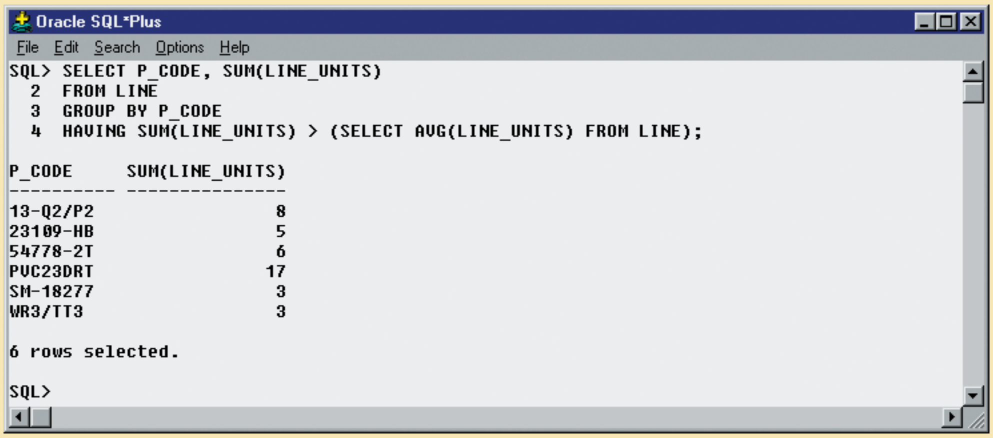
8.3.3 HAVING SUBQUERIES

Just as you can use subqueries with the WHERE clause, you can use a subquery with a HAVING clause. Remember that the HAVING clause is used to restrict the output of a GROUP BY query by applying a conditional criteria to the grouped rows. For example, to list all products with the total quantity sold greater than the average quantity sold, you would write the following query:

```
SELECT     P_CODE, SUM(LINE_UNITS)
FROM       LINE
GROUP BY   P_CODE
HAVING     SUM(LINE_UNITS) > (SELECT AVG(LINE_UNITS) FROM LINE);
```

The result of that query is shown in Figure 8.15.

FIGURE 8.15 HAVING subquery example



```
SQL> SELECT P_CODE, SUM(LINE_UNITS)
  2  FROM LINE
  3  GROUP BY P_CODE
  4  HAVING SUM(LINE_UNITS) > (SELECT AVG(LINE_UNITS) FROM LINE);

P_CODE      SUM(LINE_UNITS)
---------- ---------------
13-Q2/P2                  8
23109-HB                  5
54778-2T                  6
PVC23DRT                 17
SM-18277                  3
WR3/TT3                   3

6 rows selected.

SQL>
```

8.3.4 Multirow Subquery Operators: ANY and ALL

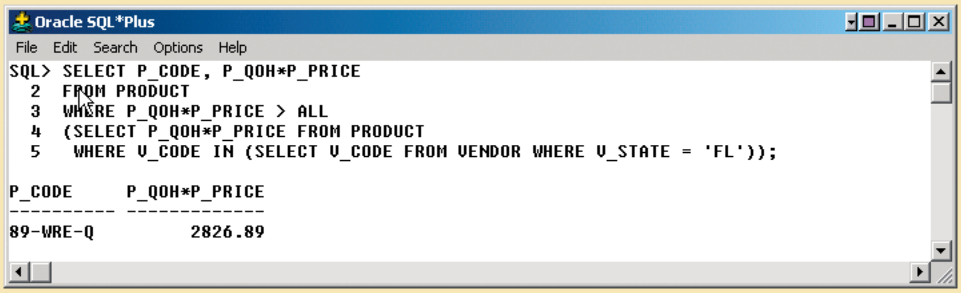
So far, you have learned that you must use an IN subquery when you need to compare a value to a list of values. But the IN subquery uses an equality operator; that is, it selects only those rows that match (are equal to) at least one of the values in the list. What happens if you need to do an inequality comparison (> or <) of one value to a list of values?

For example, suppose you want to know what products have a product cost that is greater than all individual product costs for products provided by vendors from Florida.

```
SELECT    P_CODE, P_QOH * P_PRICE
FROM      PRODUCT
WHERE     P_QOH * P_PRICE > ALL (SELECT   P_QOH * P_PRICE
                             FROM     PRODUCT
                             WHERE    V_CODE IN (SELECT V_CODE
                                                 FROM    VENDOR
                                                 WHERE  V_STATE = 'FL'));
```

The result of that query is shown in Figure 8.16.

FIGURE 8.16 Multirow subquery operator example



```
SQL> SELECT P_CODE, P_QOH*P_PRICE
  2  FROM PRODUCT
  3  WHERE P_QOH*P_PRICE > ALL
  4  (SELECT P_QOH*P_PRICE FROM PRODUCT
  5   WHERE V_CODE IN (SELECT V_CODE FROM VENDOR WHERE V_STATE = 'FL'));

P_CODE      P_QOH*P_PRICE
---------- -------------
89-WRE-Q          2826.89
```

It's important to note the following points about the query and its output in Figure 8.16:

- The query is a typical example of a nested query.
- The query has one outer SELECT statement with a SELECT subquery (call it sq^{A}) containing a second SELECT subquery (call it sq^{B}).
- The last SELECT subquery (sq^{B}) is executed first and returns a list of all vendors from Florida.
- The first SELECT subquery (sq^{A}) uses the output of the SELECT subquery (sq^{B}). The sq^{A} subquery returns the list of product costs for all products provided by vendors from Florida.
- The use of the ALL operator allows you to compare a single value (P_QOH * P_PRICE) with a list of values returned by the first subquery (sq^{A}) using a comparison operator other than equals.
- For a row to appear in the result set, it has to meet the criterion P_QOH * P_PRICE > ALL, of the individual values returned by the subquery sq^{A}. The values returned by sq^{A} are a list of product costs. In fact, "greater than ALL" is equivalent to "greater than the highest product cost of the list." In the same way, a condition of "less than ALL" is equivalent to "less than the lowest product cost of the list."

Another powerful operator is the ANY multirow operator (near cousin of the ALL multirow operator). The ANY operator allows you to compare a single value to a list of values, selecting only the rows for which the inventory cost is greater than any value of the list or less than any value of the list. You could use the equal to ANY operator, which would be the equivalent of the IN operator.

8.3.5 FROM Subqueries

So far you have seen how the SELECT statement uses subqueries within WHERE, HAVING, and IN statements and how the ANY and ALL operators are used for multirow subqueries. In all of those cases, the subquery was part of a conditional expression and it always appeared at the right side of the expression. In this section, you will learn how to use subqueries in the FROM clause.

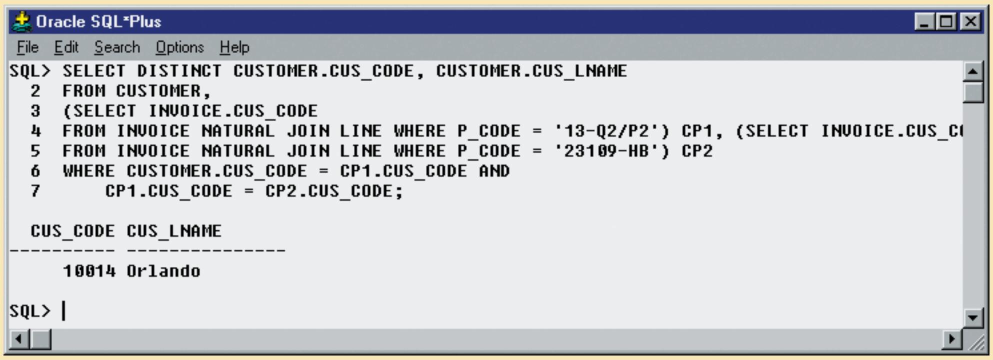
As you already know, the FROM clause specifies the table(s) from which the data will be drawn. Because the output of a SELECT statement is another table (or more precisely a "virtual" table), you could use a SELECT subquery in the FROM clause. For example, assume that you want to know all customers who have purchased products 13-Q2/P2 *and* 23109-HB. All product purchases are stored in the LINE table. It is easy to find out who purchased any given product by searching the P_CODE attribute in the LINE table. But in this case, you want to know all customers who purchased both products, not just one. You could write the following query:

```
SELECT      DISTINCT CUSTOMER.CUS_CODE, CUSTOMER.CUS_LNAME
FROM        CUSTOMER,
            (SELECT INVOICE.CUS_CODE FROM INVOICE NATURAL JOIN LINE
            WHERE P_CODE = '13-Q2/P2') CP1,
            (SELECT INVOICE.CUS_CODE FROM INVOICE NATURAL JOIN LINE
            WHERE P_CODE = '23109-HB') CP2
WHERE       CUSTOMER.CUS_CODE = CP1.CUS_CODE AND CP1.CUS_CODE = CP2.CUS_CODE;
```

The result of that query is shown in Figure 8.17.

Note in Figure 8.17 that the first subquery returns all customers who purchased product 13-Q2/P2, while the second subquery returns all customers who purchased product 23109-HB. So in this FROM subquery, you are joining the CUSTOMER table with two virtual tables. The join condition selects only the rows with matching CUS_CODE values in each table (base or virtual).

FIGURE 8.17 **FROM subquery example**



```
SQL> SELECT DISTINCT CUSTOMER.CUS_CODE, CUSTOMER.CUS_LNAME
  2  FROM CUSTOMER,
  3  (SELECT INVOICE.CUS_CODE
  4  FROM INVOICE NATURAL JOIN LINE WHERE P_CODE = '13-Q2/P2') CP1, (SELECT INVOICE.CUS_C
  5  FROM INVOICE NATURAL JOIN LINE WHERE P_CODE = '23109-HB') CP2
  6  WHERE CUSTOMER.CUS_CODE = CP1.CUS_CODE AND
  7      CP1.CUS_CODE = CP2.CUS_CODE;

  CUS_CODE CUS_LNAME
---------- ---------------
     10014 Orlando

SQL> |
```

In the previous chapter, you learned that a view is also a virtual table; therefore, you can use a view name anywhere a table is expected. So in this example, you could create two views: one listing all customers who purchased product 13-Q2/P2 and another listing all customers who purchased product 23109-HB. Doing so, you would write the query as:

```
CREATE VIEW CP1 AS
      SELECT      INVOICE.CUS_CODE FROM INVOICE NATURAL JOIN LINE
      WHERE       P_CODE = '13-Q2/P2';

CREATE VIEW CP2 AS
      SELECT      INVOICE.CUS_CODE FROM INVOICE NATURAL JOIN LINE
      WHERE       P_CODE = '23109-HB';

SELECT      DISTINCT CUS_CODE, CUS_LNAME
FROM        CUSTOMER NATURAL JOIN CP1 NATURAL JOIN CP2;
```

You might speculate that the above query could also be written using the following syntax:

```
SELECT      CUS_CODE, CUS_LNAME
FROM        CUSTOMER NATURAL JOIN INVOICE NATURAL JOIN LINE
WHERE       P_CODE = '13-Q2/P2' AND P_CODE = '23109-HB';
```

But if you examine that query carefully, you will note that a P_CODE cannot be equal to two different values at the same time. Therefore, the query will not return any rows.

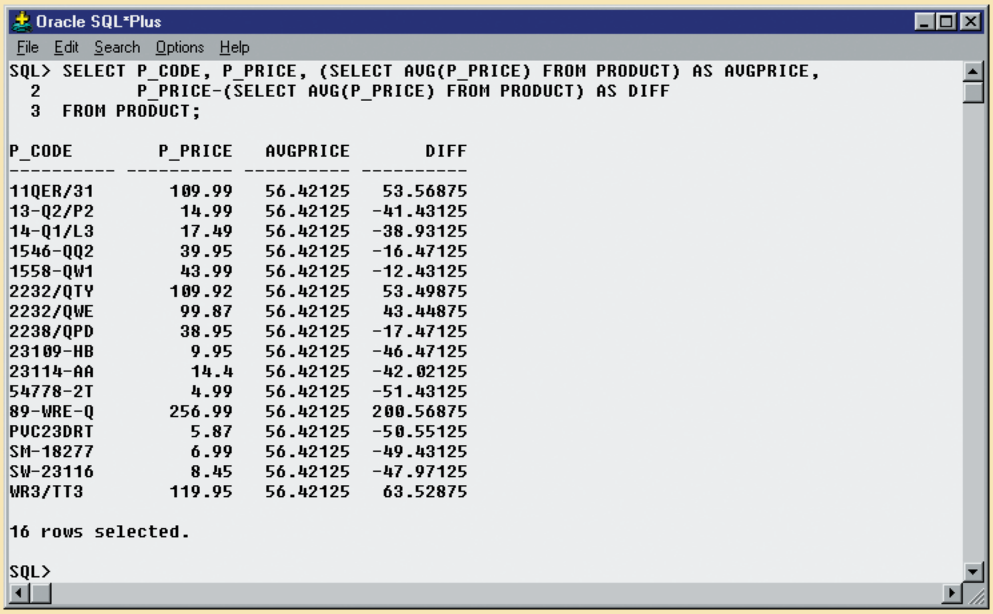
8.3.6 Attribute List Subqueries

The SELECT statement uses the attribute list to indicate what columns to project in the resulting set. Those columns can be attributes of base tables or computed attributes or the result of an aggregate function. The attribute list can also include a subquery expression, also known as an inline subquery. A subquery in the attribute list must return one single value; otherwise, an error code is raised. For example, a simple inline query can be used to list the difference between each product's price and the average product price:

```
SELECT      P_CODE, P_PRICE, (SELECT AVG(P_PRICE) FROM PRODUCT) AS AVGPRICE,
            P_PRICE – (SELECT AVG(P_PRICE) FROM PRODUCT) AS DIFF
FROM        PRODUCT;
```

Figure 8.18 shows the result of that query.

FIGURE 8.18 Inline subquery example



```
SQL> SELECT P_CODE, P_PRICE, (SELECT AVG(P_PRICE) FROM PRODUCT) AS AVGPRICE,
  2         P_PRICE-(SELECT AVG(P_PRICE) FROM PRODUCT) AS DIFF
  3  FROM PRODUCT;

P_CODE        P_PRICE   AVGPRICE       DIFF
---------- ---------- ---------- ----------
11QER/31       109.99   56.42125   53.56875
13-Q2/P2        14.99   56.42125  -41.43125
14-Q1/L3        17.49   56.42125  -38.93125
1546-QQ2        39.95   56.42125  -16.47125
1558-QW1        43.99   56.42125  -12.43125
2232/QTY       109.92   56.42125   53.49875
2232/QWE        99.87   56.42125   43.44875
2238/QPD        38.95   56.42125  -17.47125
23109-HB         9.95   56.42125  -46.47125
23114-AA         14.4   56.42125  -42.02125
54778-2T         4.99   56.42125  -51.43125
89-WRE-Q       256.99   56.42125  200.56875
PVC23DRT         5.87   56.42125  -50.55125
SM-18277         6.99   56.42125  -49.43125
SW-23116         8.45   56.42125  -47.97125
WR3/TT3        119.95   56.42125   63.52875

16 rows selected.

SQL>
```

In Figure 8.18, note that the inline query output returns one single value (the average product's price) and that the value is the same in every row. Note also that the query used the full expression instead of the column aliases when computing the difference. In fact, if you try to use the alias in the difference expression, you will get an error message. The column alias cannot be used in computations in the attribute list when the alias is defined in the same attribute list. That DBMS requirement is due to the way the DBMS parses and executes queries.

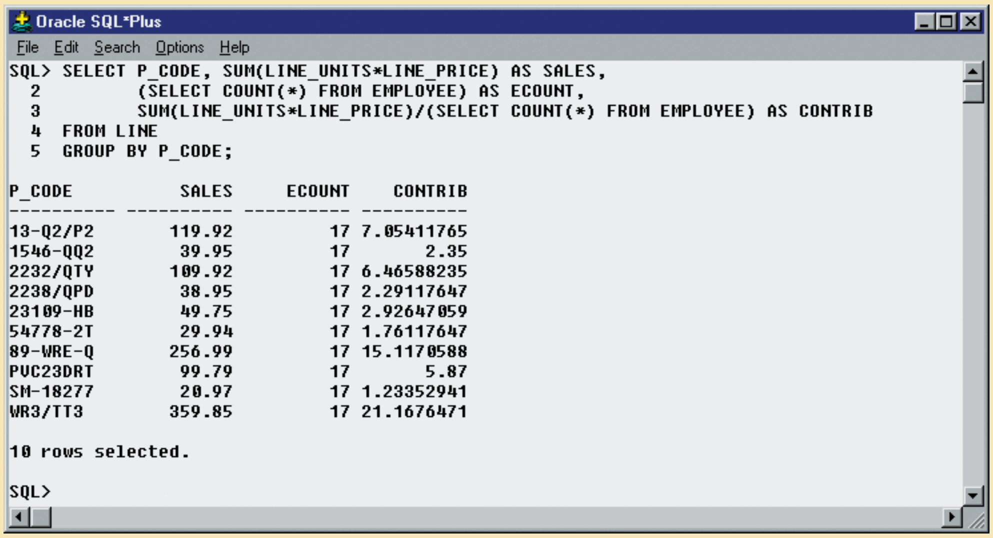
Another example will help you understand the use of attribute list subqueries and column aliases. For example, suppose you want to know the product code, the total sales by product, and the contribution by employee of each product's sales. To get the sales by product, you need to use only the LINE table. To compute the contribution by employee, you need to know the number of employees (from the EMPLOYEE table). As you study the tables' structures, you can see that the LINE and EMPLOYEE tables do not share a common attribute. In fact, you don't need a common attribute. You need to know only the total number of employees, not the total employees related to each product. So to answer the query, you would write the following code:

```
SELECT      P_CODE, SUM(LINE_UNITS * LINE_PRICE) AS SALES,
            (SELECT COUNT(*) FROM EMPLOYEE) AS ECOUNT,
            SUM(LINE_UNITS * LINE_PRICE)/(SELECT COUNT(*) FROM EMPLOYEE) AS CONTRIB
FROM        LINE
GROUP BY    P_CODE;
```

The result of that query is shown in Figure 8.19.

As you can see in Figure 8.19, the number of employees remains the same for each row in the result set. The use of that type of subquery is limited to certain instances where you need to include data from other tables that are not directly related to a main table or tables in the query. The value will remain the same for each row, like a constant in a programming language. (You will learn another use of inline subqueries in Section 8.3.7, Correlated Subqueries).

FIGURE 8.19 Another example of an inline subquery



```
SQL> SELECT P_CODE, SUM(LINE_UNITS*LINE_PRICE) AS SALES,
  2         (SELECT COUNT(*) FROM EMPLOYEE) AS ECOUNT,
  3         SUM(LINE_UNITS*LINE_PRICE)/(SELECT COUNT(*) FROM EMPLOYEE) AS CONTRIB
  4  FROM LINE
  5  GROUP BY P_CODE;

P_CODE          SALES     ECOUNT    CONTRIB
---------- ---------- ---------- ----------
13-Q2/P2       119.92         17 7.05411765
1546-QQ2        39.95         17       2.35
2232/QTY       109.92         17 6.46588235
2238/QPD        38.95         17 2.29117647
23109-HB        49.75         17 2.92647059
54778-2T        29.94         17 1.76117647
89-WRE-Q       256.99         17 15.1170588
PVC23DRT        99.79         17       5.87
SM-18277        20.97         17 1.23352941
WR3/TT3        359.85         17 21.1676471

10 rows selected.

SQL>
```

Note that you cannot use an alias in the attribute list to write the expression that computes the contribution per employee.

Another way to write the same query by using column aliases requires the use of a subquery in the FROM clause, as follows:

```
SELECT      P_CODE, SALES, ECOUNT, SALES/ECOUNT AS CONTRIB
FROM        (SELECT     P_CODE, SUM(LINE_UNITS * LINE_PRICE) AS SALES,
                        (SELECT COUNT(*) FROM EMPLOYEE) AS ECOUNT
            FROM        LINE
            GROUP BY    P_CODE);
```

In that case, you are actually using two subqueries. The subquery in the FROM clause executes first and returns a virtual table with three columns: P_CODE, SALES, and ECOUNT. The FROM subquery contains an inline subquery that returns the number of employees as ECOUNT. Because the outer query receives the output of the inner query, you can now refer to the columns in the outer subquery using the column aliases.

8.3.7 Correlated Subqueries

Until now, all subqueries you have learned execute independently. That is, each subquery in a command sequence executes in a serial fashion, one after another. The inner subquery executes first; its output is used by the outer query, which then executes until the last outer query executes (the first SQL statement in the code).

In contrast, a **correlated subquery** is a subquery that executes once for each row in the outer query. That process is similar to the typical nested loop in a programming language. For example:

```
FOR X = 1 TO 2
       FOR Y = 1 TO 3
              PRINT "X = "X, "Y = "Y
       END
END
```

will yield the output

```
X = 1        Y = 1
X = 1        Y = 2
X = 1        Y = 3
X = 2        Y = 1
X = 2        Y = 2
X = 2        Y = 3
```

Note that the outer loop X = 1 TO 2 begins the process by setting X = 1; then the inner loop Y = 1 TO 3 is completed for each X outer loop value. The relational DBMS uses the same sequence to produce correlated subquery results:

1. It initiates the outer query.
2. For each row of the outer query result set, it executes the inner query by passing the outer row to the inner query.

That process is the opposite of the subqueries you have seen so far. The query is called a *correlated* subquery because the inner query is *related* to the outer query by the fact that the inner query references a column of the outer subquery.

To see the correlated subquery in action, suppose you want to know all product sales in which the units sold value is greater than the average units sold value *for that product* (as opposed to the average for *all* products). In that case, the following procedure must be completed:

1. Compute the average-units-sold value for a product.
2. Compare the average computed in Step 1 to the units sold in each sale row; then select only the rows in which the number of units sold is greater.

The following correlated query completes the preceding two-step process:

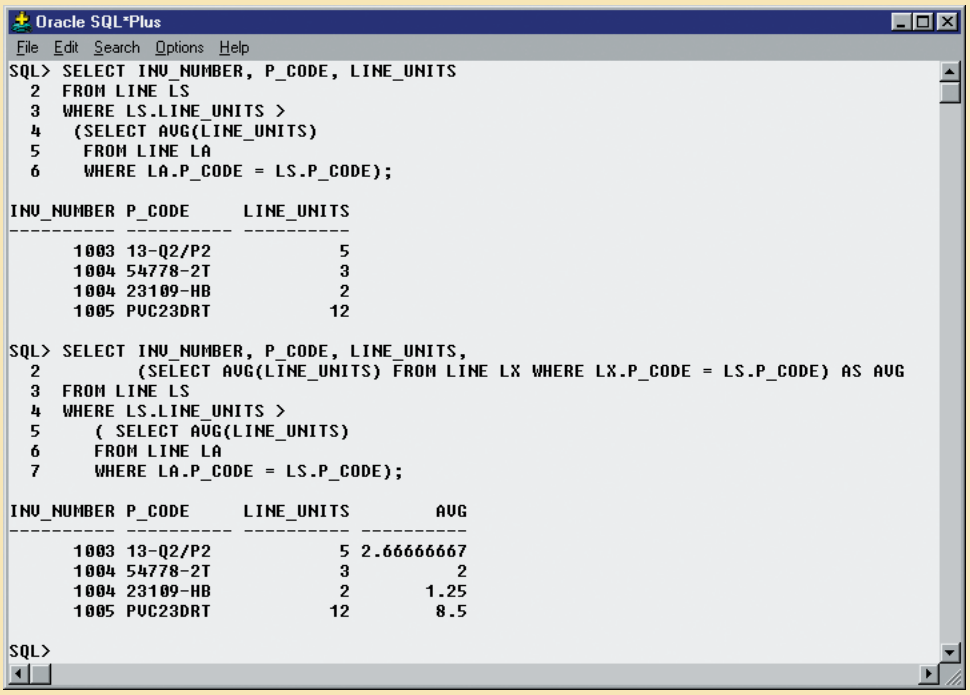
```
SELECT      INV_NUMBER, P_CODE, LINE_UNITS
FROM        LINE LS
WHERE       LS.LINE_UNITS > (SELECT     AVG(LINE_UNITS)
                             FROM       LINE LA
                             WHERE      LA.P_CODE = LS.P_CODE);
```

The first example in Figure 8.20 shows the result of that query.

In the top query and its result in Figure 8.20, note that the LINE table is used more than once; so you must use table aliases. In that case, the inner query computes the average units sold of the product that matches the P_CODE of the outer query P_CODE. That is, the inner query runs once using the first product code found in the (outer) LINE table and returns the average sale for that product. When the number of units sold in that (outer) LINE row is greater than the average computed, the row is added to the output. Then the inner query runs again, this time using the second product code found in the (outer) LINE table. The process repeats until the inner query has run for all rows in the (outer) LINE table. In that case, the inner query will be repeated as many times as there are rows in the outer query.

To verify the results and to provide an example of how you can combine subqueries, you can add a correlated inline subquery to the previous query. That correlated inline subquery will show the average units sold column for each product. (See the second query and its results in Figure 8.20.) As you can see, the new query contains a correlated inline subquery that computes the average units sold for each product. You not only get an answer, but you also can verify that the answer is correct.

FIGURE 8.20 Correlated subquery examples



```
SQL> SELECT INV_NUMBER, P_CODE, LINE_UNITS
  2  FROM LINE LS
  3  WHERE LS.LINE_UNITS >
  4   (SELECT AVG(LINE_UNITS)
  5    FROM LINE LA
  6    WHERE LA.P_CODE = LS.P_CODE);

INV_NUMBER P_CODE     LINE_UNITS
---------- ---------- ----------
      1003 13-Q2/P2            5
      1004 54778-2T            3
      1004 23109-HB            2
      1005 PVC23DRT           12

SQL> SELECT INV_NUMBER, P_CODE, LINE_UNITS,
  2         (SELECT AVG(LINE_UNITS) FROM LINE LX WHERE LX.P_CODE = LS.P_CODE) AS AVG
  3  FROM LINE LS
  4  WHERE LS.LINE_UNITS >
  5     ( SELECT AVG(LINE_UNITS)
  6     FROM LINE LA
  7     WHERE LA.P_CODE = LS.P_CODE);

INV_NUMBER P_CODE     LINE_UNITS        AVG
---------- ---------- ---------- ----------
      1003 13-Q2/P2            5 2.66666667
      1004 54778-2T            3          2
      1004 23109-HB            2       1.25
      1005 PVC23DRT           12        8.5

SQL>
```

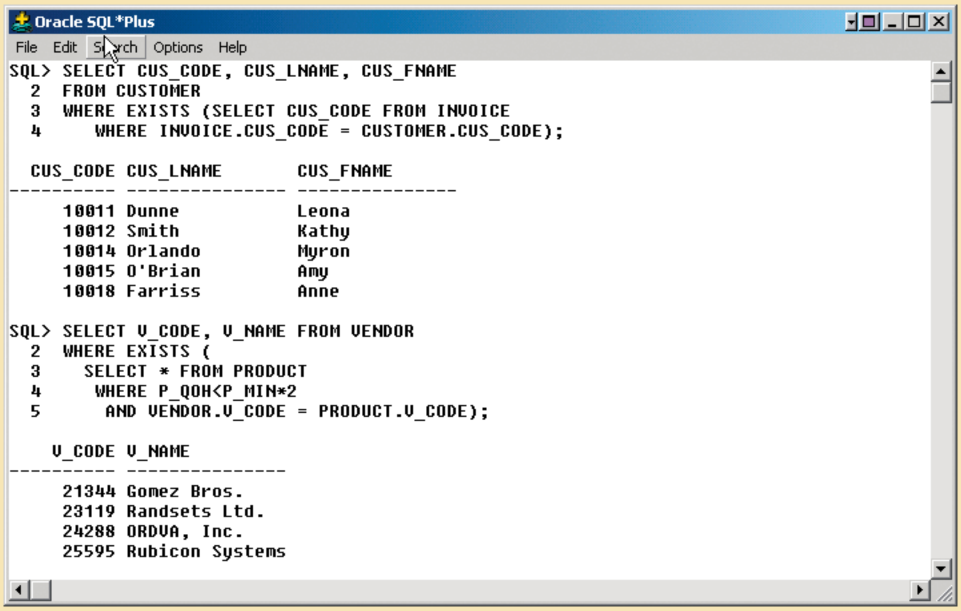
Correlated subqueries can also be used with the EXISTS special operator. For example, suppose you want to know all customers who have placed an order lately. In that case, you could use a correlated subquery like the first one shown in Figure 8.21:

```
SELECT      CUS_CODE, CUS_LNAME, CUS_FNAME
FROM        CUSTOMER
WHERE EXISTS (SELECT   CUS_CODE FROM INVOICE
              WHERE    INVOICE.CUS_CODE = CUSTOMER.CUS_CODE);
```

The second example of an EXISTS correlated subquery in Figure 8.21 will help you understand how to use correlated queries. For example, suppose you want to know what vendors you must contact to start ordering products that are approaching the minimum quantity-on-hand value. In particular, you want to know the vendor code and name of vendors for products having a quantity on hand that is less than double the minimum quantity. The query that answers that question is as follows:

```
SELECT      V_CODE, V_NAME
FROM        VENDOR
WHERE EXISTS (SELECT *
              FROM    PRODUCT
              WHERE   P_QOH < P_MIN * 2
              AND     VENDOR.V_CODE = PRODUCT.V_CODE);
```

FIGURE 8.21 **EXISTS correlated subquery examples**



```
SQL> SELECT CUS_CODE, CUS_LNAME, CUS_FNAME
     FROM CUSTOMER
     WHERE EXISTS (SELECT CUS_CODE FROM INVOICE
        WHERE INVOICE.CUS_CODE = CUSTOMER.CUS_CODE);

  CUS_CODE CUS_LNAME       CUS_FNAME
---------- --------------- ---------------
     10011 Dunne           Leona
     10012 Smith           Kathy
     10014 Orlando         Myron
     10015 O'Brian         Amy
     10018 Farriss         Anne

SQL> SELECT V_CODE, V_NAME FROM VENDOR
     WHERE EXISTS (
       SELECT * FROM PRODUCT
        WHERE P_QOH<P_MIN*2
         AND VENDOR.V_CODE = PRODUCT.V_CODE);

    V_CODE V_NAME
---------- ---------------
     21344 Gomez Bros.
     23119 Randsets Ltd.
     24288 ORDVA, Inc.
     25595 Rubicon Systems
```

In the second query in Figure 8.21, note that:

1. The inner correlated subquery runs using the first vendor.
2. If any products match the condition (quantity on hand is less than double the minimum quantity), the vendor code and name are listed in the output.
3. The correlated subquery runs using the second vendor, and the process repeats itself until all vendors are used.

8.4 SQL FUNCTIONS

The data in databases are the basis of critical business information. Generating information from data often requires many data manipulations. Sometimes such data manipulation involves the decomposition of data elements. For example, an employee's date of birth can be subdivided into a day, a month, and a year. A product manufacturing code (for example, SE-05-2-09-1234-1-3/12/04-19:26:48) can be designed to record the manufacturing region, plant, shift, production line, employee number, date, and time. For years, conventional programming languages have had special functions that enabled programmers to perform data transformations like those data decompositions. If you know a modern programming language, it's very likely that the SQL functions in this section will look familiar.

SQL functions are very useful tools. You'll need to use functions when you want to list all employees ordered by year of birth or when your marketing department wants you to generate a list of all customers ordered by zip code and the first three digits of their telephone numbers. In both of those cases, you'll need to use data elements that are not present as such in the database; instead you'll need an SQL function that can be derived from an existing attribute. Functions always use a numerical, date, or string value. The value may be part of the command itself (a constant or literal) or it may be an attribute located in a table. Therefore, a function may appear anywhere in an SQL statement where a value or an attribute can be used.

There are many types of SQL functions, such as arithmetic, trigonometric, string, date, and time functions. This section will not explain all of those types of functions in detail, but it will give you a brief overview of the most useful ones.

Note

Although the main DBMS vendors support the SQL functions covered here, the syntax or degree of support will probably differ. In fact, DBMS vendors invariably add their own functions to products to lure new customers. The functions covered in this section represent just a small portion of functions supported by your DBMS. Read your DBMS SQL reference manual for a complete list of available functions.

8.4.1 Date and Time Functions

All SQL-standard DBMSs support date and time functions. All date functions take one parameter (of a date or character data type) and return a value (character, numeric, or date type). Unfortunately, date/time data types are implemented differently by different DBMS vendors. The problem occurs because the ANSI SQL standard defines date data types, but it does not say how those data types are to be stored. Instead, it lets the vendor deal with that issue.

Because date/time functions differ from vendor to vendor, this section will cover basic date/time functions for MS Access/SQL Server and for Oracle. Table 8.3 shows a list of selected MS Access/SQL Server date/time functions.

TABLE 8.3 Selected MS Access/SQL Server Date/Time Functions

FUNCTION	EXAMPLE(S)

YEAR
Returns a four-digit year
Syntax:
YEAR(date_value)
	Lists all employees born in 1966:
	SELECT EMP_LNAME, EMP_FNAME, EMP_DOB,
	YEAR(EMP_DOB) AS YEAR
	FROM EMPLOYEE
	WHERE YEAR(EMP_DOB) = 1966;

MONTH
Returns a two-digit month code
Syntax:
MONTH(date_value)
	Lists all employees born in November:
	SELECT EMP_LNAME, EMP_FNAME, EMP_DOB,
	MONTH(EMP_DOB) AS MONTH
	FROM EMPLOYEE
	WHERE MONTH(EMP_DOB) = 11;

DAY
Returns the number of the day
Syntax:
DAY(date_value)
	Lists all employees born on the 14th day of the month:
	SELECT EMP_LNAME, EMP_FNAME, EMP_DOB,
	DAY(EMP_DOB) AS DAY
	FROM EMPLOYEE
	WHERE DAY(EMP_DOB) = 14;

DATE() – MS Access **GETDATE() – SQL Server** Returns today's date	Lists how many days are left until Christmas: SELECT #25-Dec-2008# – DATE(); Note two features: • There is no FROM clause, which is acceptable in MS Access. • The Christmas date is enclosed in # signs because you are doing date arithmetic. In MS SQL Server: Use GETDATE() to get the current system date. To compute the difference between dates, use the DATEDIFF function (see below).

TABLE 8.3 Selected MS Access/SQL Server Date/Time Functions (continued)

FUNCTION	EXAMPLE(S)

DATEADD – SQL Server
Adds a number of selected time periods to a date
Syntax:
DATEADD(datepart, number, date)
	Adds a *number* of *dateparts* to a given *date*. Dateparts can be minutes, hours, days, weeks, months, quarters, or years. For example:
	SELECT DATEADD(day,90, P_INDATE) AS DueDate
	FROM PRODUCT;
	The above example adds 90 days to P_INDATE.
	In MS Access use:
	SELECT P_INDATE+90 AS DueDate
	FROM PRODUCT;

DATEDIFF – SQL Server
Subtracts two dates
Syntax:
DATEDIFF(datepart, startdate, enddate)
	Returns the difference between two dates expressed in a selected *datepart*. For example:
	SELECT DATEDIFF(day, P_INDATE, GETDATE()) AS DaysAgo
	FROM PRODUCT;
	In MS Access use:
	SELECT DATE() - P_INDATE AS DaysAgo
	FROM PRODUCT;

Table 8.4 shows the equivalent date/time functions used in Oracle. Note that Oracle uses the same function (TO_CHAR) to extract the various parts of a date. Also, another function (TO_DATE) is used to convert character strings to a valid Oracle date format that can be used in date arithmetic.

TABLE 8.4 Selected Oracle Date/Time Functions

FUNCTION	EXAMPLE(S)

TO_CHAR
Returns a character string or a formatted string from a date value
Syntax:
TO_CHAR(date_value, fmt)
fmt = format used; can be:
MONTH: name of month
MON: three-letter month name
MM: two-digit month name
D: number for day of week
DD: number day of month
DAY: name of day of week
YYYY: four-digit year value
YY: two-digit year value
	Lists all employees born in 1982:
	SELECT EMP_LNAME, EMP_FNAME, EMP_DOB, TO_CHAR(EMP_DOB, 'YYYY') AS YEAR
	FROM EMPLOYEE
	WHERE TO_CHAR(EMP_DOB, 'YYYY') = '1982';
	Lists all employees born in November:
	SELECT EMP_LNAME, EMP_FNAME, EMP_DOB, TO_CHAR(EMP_DOB, 'MM') AS MONTH
	FROM EMPLOYEE
	WHERE TO_CHAR(EMP_DOB, 'MM') = '11';
	Lists all employees born on the 14th day of the month:
	SELECT EMP_LNAME, EMP_FNAME, EMP_DOB, TO_CHAR(EMP_DOB, 'DD') AS DAY
	FROM EMPLOYEE
	WHERE TO_CHAR(EMP_DOB, 'DD') = '14';

TABLE 8.4 Selected Oracle Date/Time Functions (continued)

FUNCTION	EXAMPLE(S)
TO_DATE Returns a date value using a character string and a date format mask; also used to translate a date between formats Syntax: TO_DATE(char_value, fmt) fmt = format used; can be: MONTH: name of month MON: three-letter month name MM: two-digit month name D: number for day of week DD: number day of month DAY: name of day of week YYYY: four-digit year value YY: two-digit year value	Lists the approximate age of the employees on the company's tenth anniversary date (11/25/2008): SELECT EMP_LNAME, EMP_FNAME, EMP_DOB, '11/25/2008' AS ANIV_DATE, (TO_DATE('11/25/1998','MM/DD/YYYY') - EMP_DOB)/365 AS YEARS FROM EMPLOYEE ORDER BY YEARS; Note the following: • '11/25/2008' is a text string, not a date. • The TO_DATE function translates the text string to a valid Oracle date used in date arithmetic. How many days between Thanksgiving and Christmas 2008? SELECT TO_DATE('2008/12/25','YYYY/MM/DD') – TO_DATE('NOVEMBER 27, 2008','MONTH DD, YYYY') FROM DUAL; Note the following: • The TO_DATE function translates the text string to a valid Oracle date used in date arithmetic. • DUAL is Oracle's pseudo table used only for cases where a table is not really needed.
SYSDATE Returns today's date	Lists how many days are left until Christmas: SELECT TO_DATE('25-Dec-2008','DD-MON-YYYY') SYSDATE FROM DUAL; Notice two things: • DUAL is Oracle's pseudo table used only for cases where a table is not really needed. • The Christmas date is enclosed in a TO_DATE function to translate the date to a valid date format.

ADD_MONTHS
Adds a number of months to a date; useful for adding months or years to a date
Syntax:
ADD_MONTHS(date_value, n)
n = number of months
	Lists all products with their expiration date (two years from the purchase date):
	SELECT P_CODE, P_INDATE, ADD_MONTHS(P_INDATE,24)
	FROM PRODUCT
	ORDER BY ADD_MONTHS(P_INDATE,24);

LAST_DAY
Returns the date of the last day of the month given in a date
Syntax:
LAST_DAY(date_value)
	Lists all employees who were hired within the last seven days of a month:
	SELECT EMP_LNAME, EMP_FNAME, EMP_HIRE_DATE
	FROM EMPLOYEE
	WHERE EMP_HIRE_DATE >=LAST_DAY(EMP_HIRE_DATE)-7;

8.4.2 Numeric Functions

Numeric functions can be grouped in many different ways, such as algebraic, trigonometric, and logarithmic. In this section, you will learn two very useful functions. Do not confuse the SQL aggregate functions you saw in the previous chapter with the numeric functions in this section. The first group operates over a set of values (multiple rows—hence, the name *aggregate functions*), while the numeric functions covered here operate over a single row. Numeric functions take one numeric parameter and return one value. Table 8.5 shows a selected group of numeric functions available.

TABLE 8.5 Selected Numeric Functions

FUNCTION	EXAMPLE(S)

ABS
Returns the absolute value of a number
Syntax:
ABS(numeric_value)
	In Oracle use:
	SELECT 1.95, -1.93, ABS(1.95), ABS(-1.93)
	FROM DUAL;
	In MS Access/SQL Server use:
	SELECT 1.95, -1.93, ABS(1.95), ABS(-1.93);

ROUND Rounds a value to a specified precision (number of digits) Syntax: ROUND(numeric_value, p) p = precision	Lists the product prices rounded to one and zero decimal places: SELECT P_CODE, P_PRICE, ROUND(P_PRICE,1) AS PRICE1, ROUND(P_PRICE,0) AS PRICE0 FROM PRODUCT;

CEIL/CEILING/FLOOR
Returns the smallest integer greater than or equal to a number or returns the largest integer equal to or less than a number, respectively
Syntax:
CEIL(numeric_value) – Oracle
CEILING(numeric_value) – SQL Server
FLOOR(numeric_value)
	Lists the product price, smallest integer greater than or equal to the product price, and the largest integer equal to or less than the product price.
	In Oracle use:
	SELECT P_PRICE, CEIL(P_PRICE), FLOOR(P_PRICE)
	FROM PRODUCT;
	In SQL Server use:
	SELECT P_PRICE, CEILING(P_PRICE), FLOOR(P_PRICE)
	FROM PRODUCT;
	MS Access does not support these functions.

8.4.3 String Functions

String manipulations are among the most-used functions in programming. If you have ever created a report using any programming language, you know the importance of properly concatenating strings of characters, printing names in uppercase, or knowing the length of a given attribute. Table 8.6 shows a subset of useful string manipulation functions.

TABLE 8.6 Selected String Functions

FUNCTION	EXAMPLE(S)

Concatenation
|| – Oracle
+ – MS Access/SQL Server
Concatenates data from two different character columns and returns a single column
Syntax:
strg_value || strg_value
strg_value + strg_value
	Lists all employee names (concatenated).
	In Oracle use:
	SELECT EMP_LNAME || ', ' || EMP_FNAME AS NAME
	FROM EMPLOYEE;
	In MS Access / SQL Server use:
	SELECT EMP_LNAME + ', ' + EMP_FNAME AS NAME
	FROM EMPLOYEE;

UPPER/LOWER
Returns a string in all capital or all lowercase letters
Syntax:
UPPER(strg_value)
LOWER(strg_value)
	Lists all employee names in all capital letters (concatenated).
	In Oracle use:
	SELECT UPPER(EMP_LNAME) || ', ' || UPPER(EMP_FNAME) AS NAME
	FROM EMPLOYEE;
	In SQL Server use:
	SELECT UPPER(EMP_LNAME) + ', ' + UPPER(EMP_FNAME) AS NAME
	FROM EMPLOYEE;
	Lists all employee names in all lowercase letters (concatenated).
	In Oracle use:
	SELECT LOWER(EMP_LNAME) || ', ' || LOWER(EMP_FNAME) AS NAME
	FROM EMPLOYEE;
	In SQL Server use:
	SELECT LOWER(EMP_LNAME) + ', ' + LOWER(EMP_FNAME) AS NAME
	FROM EMPLOYEE;
	Not supported by MS Access.

SUBSTRING
Returns a substring or part of a given string parameter
Syntax:
SUBSTR(strg_value, p, l) – Oracle
SUBSTRING(strg_value,p,l) – SQL Server
p = start position
l = length of characters
	Lists the first three characters of all employee phone numbers.
	In Oracle use:
	SELECT EMP_PHONE, SUBSTR(EMP_PHONE,1,3) AS PREFIX
	FROM EMPLOYEE;
	In SQL Server use:
	SELECT EMP_PHONE, SUBSTRING(EMP_PHONE,1,3) AS PREFIX
	FROM EMPLOYEE;
	Not supported by MS Access.

LENGTH
Returns the number of characters in a string value
Syntax:
LENGTH(strg_value) – Oracle
LEN(strg_value) – SQL Server
	Lists all employee last names and the length of their names; ordered descended by last name length.
	In Oracle use:
	SELECT EMP_LNAME, LENGTH(EMP_LNAME) AS NAMESIZE
	FROM EMPLOYEE;
	In MS Access / SQL Server use:
	SELECT EMP_LNAME, LEN(EMP_LNAME) AS NAMESIZE
	FROM EMPLOYEE;

8.4.4 Conversion Functions

Conversion functions allow you to take a value of a given data type and convert it to the equivalent value in another data type. In Section 8.4.1, you learned about two of the basic Oracle SQL conversion functions: TO_CHAR and TO_DATE. Note that the TO_CHAR function takes a date value and returns a character string representing a day, a month, or a year. In the same way, the TO_DATE function takes a character string representing a date and returns an actual date in Oracle format. SQL Server uses the CAST and CONVERT functions to convert one data type to another. A summary of the selected functions is shown in Table 8.7.

TABLE 8.7 Selected Conversion Functions

FUNCTION	EXAMPLE(S)
Numeric to Character: **TO_CHAR – Oracle** **CAST – SQL Server** **CONVERT – SQL Server** Returns a character string from a numeric value. Syntax: Oracle: TO_CHAR(numeric_value, fmt) SQL Server: CAST (numeric AS varchar(length)) CONVERT(varchar(length), numeric)	Lists all product prices, quantity on hand, percent discount, and total inventory cost using formatted values. In Oracle use: SELECT P_CODE, TO_CHAR(P_PRICE,'999.99') AS PRICE, TO_CHAR(P_QOH,'9,999.99') AS QUANTITY, TO_CHAR(P_DISCOUNT,'0.99') AS DISC, TO_CHAR(P_PRICE*P_QOH,'99,999.99') AS TOTAL_COST FROM PRODUCT; In SQL Server use: SELECT P_CODE, CAST(P_PRICE AS VARCHAR(8)) AS PRICE, CONVERT(VARCHAR(4),P_QOH) AS QUANTITY, CAST(P_DISCOUNT AS VARCHAR(4)) AS DISC, CAST(P_PRICE*P_QOH AS VARCHAR(10)) AS TOTAL_COST FROM PRODUCT; Not supported in MS Access.

Date to Character:
TO_CHAR – Oracle
CAST – SQL Server
CONVERT – SQL Server
Returns a character string or a formatted character string from a date value
Syntax:
Oracle: TO_CHAR(date_value, fmt)
SQL Server:
CAST (date AS varchar(length))
CONVERT(varchar(length), date)
	Lists all employee dates of birth, using different date formats.
	In Oracle use:
	SELECT EMP_LNAME, EMP_DOB,
	TO_CHAR(EMP_DOB, ‘DAY, MONTH DD, YYYY’)
	AS ‘DATEOFBIRTH’
	FROM EMPLOYEE;
	SELECT EMP_LNAME, EMP_DOB,
	TO_CHAR(EMP_DOB, ‘YYYY/MM/DD’)
	AS ‘DATEOFBIRTH’
	FROM EMPLOYEE;
	In SQL Server use:
	SELECT EMP_LNAME, EMP_DOB,
	CONVERT(varchar(11),EMP_DOB) AS “DATE OF BIRTH”
	FROM EMPLOYEE;
	SELECT EMP_LNAME, EMP_DOB,
	CAST(EMP_DOB as varchar(11)) AS “DATE OF BIRTH”
	FROM EMPLOYEE;
	Not supported in MS Access.

TABLE 8.7 Selected Conversion Functions (continued)

FUNCTION	EXAMPLE(S)

String to Number:
TO_NUMBER
Returns a formatted number from a character string, using a given format
Syntax:
Oracle:
TO_NUMBER(char_value, fmt)
fmt = format used; can be:
9 = displays a digit
0 = displays a leading zero
, = displays the comma
. = displays the decimal point
$ = displays the dollar sign
B = leading blank
S = leading sign
MI = trailing minus sign
	Converts text strings to numeric values when importing data to a table from another source in text format; for example, the query shown below uses the TO_NUMBER function to convert text formatted to Oracle default numeric values using the format masks given.
	In Oracle use:
	SELECT TO_NUMBER('-123.99', 'S999.99'),
	TO_NUMBER('99.78-','B999.99MI')
	FROM DUAL;
	In SQL Server use:
	SELECT CAST('-123.99' AS NUMERIC(8,2)),
	CAST('-99.78' AS NUMERIC(8,2))
	The SQL Server CAST function does not support the trailing sign on the character string.
	Not supported in MS Access.

CASE – SQL Server **DECODE – Oracle** Compares an attribute or expression with a series of values and returns an associated value or a default value if no match is found Syntax: Oracle: DECODE(e, *x, y, d*) e = attribute or expression *x* = value with which to compare e *y* = value to return in e = *x* *d* = default value to return if e is not equal to *x* SQL Server: CASE When *condition* THEN value1 ELSE value2 END	The following example returns the sales tax rate for specified states: • Compares V_STATE to 'CA'; if the values match, it returns .08. • Compares V_STATE to 'FL'; if the values match, it returns .05. • Compares V_STATE to 'TN'; if the values match, it returns .085. If there is no match, it returns 0.00 (the default value). SELECT V_CODE, V_STATE, DECODE(V_STATE,'CA',.08,'FL',.05, 'TN',.085, 0.00) AS TAX FROM VENDOR; In SQL Server use: SELECT V_CODE, V_STATE, CASE WHEN V_STATE = 'CA' THEN .08 WHEN V_STATE = 'FL' THEN .05 WHEN V_STATE = 'TN' THEN .085 ELSE 0.00 END AS TAX FROM VENDOR Not supported in MS Access.

8.5 ORACLE SEQUENCES

If you use MS Access, you might be familiar with the AutoNumber data type, which you can use to define a column in your table that will be automatically populated with unique numeric values. In fact, if you create a table in MS Access and forget to define a primary key, MS Access will offer to create a primary key column; if you accept, you will notice that MS Access creates a column named *ID* with an AutoNumber data type. After you define a column as an AutoNumber type, every time you insert a row in the table, MS Access will automatically add a value to that column, starting with 1 and increasing the value by 1 in every new row you add. Also, you cannot include that column in your INSERT statements—Access will not let you edit that value at all. MS SQL Server uses the Identity column property to serve a similar purpose. In MS SQL Server a table can have at most one column defined as an Identity column. This column behaves similarly to an MS Access column with the AutoNumber data type.

Oracle does not support the AutoNumber data type or the Identity column property. Instead, you can use a “sequence” to assign values to a column on a table. But an Oracle sequence is very different from the Access AutoNumber data type and deserves close scrutiny:

- Oracle sequences are an independent object in the database. (Sequences are not a data type.)
- Oracle sequences have a name and can be used anywhere a value is expected.

- Oracle sequences are not tied to a table or a column.
- Oracle sequences generate a numeric value that can be assigned to any column in any table.
- The table attribute to which you assigned a value based on a sequence can be edited and modified.
- An Oracle sequence can be created and deleted anytime.

The basic syntax to create a sequence in Oracle is:

CREATE SEQUENCE *name* [START WITH *n*] [INCREMENT BY *n*] [CACHE | NOCACHE]

where:

- *name* is the name of the sequence.
- *n* is an integer value that can be positive or negative.
- *START WITH* specifies the initial sequence value. (The default value is 1.)
- *INCREMENT BY* determines the value by which the sequence is incremented. (The default increment value is 1. The sequence increment can be positive or negative to enable you to create ascending or descending sequences.)
- The *CACHE* or *NOCACHE* clause indicates whether Oracle will preallocate sequence numbers in memory. (Oracle preallocates 20 values by default.)

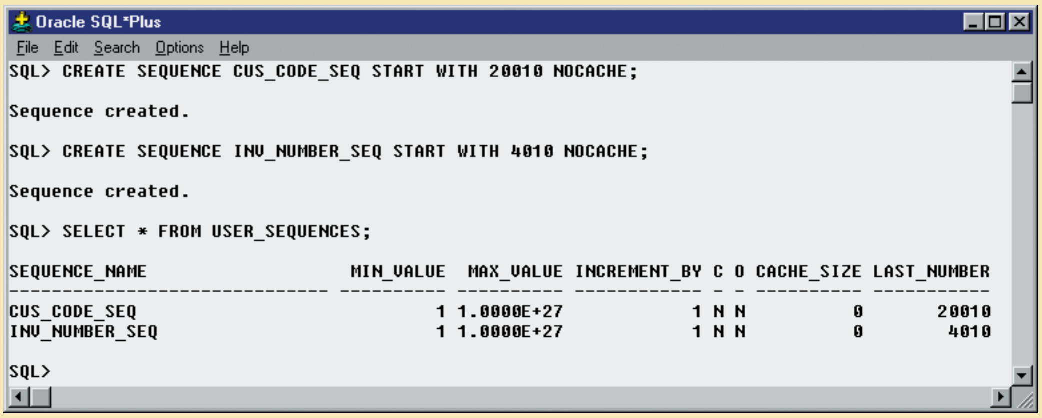
For example, you could create a sequence to automatically assign values to the customer code each time a new customer is added and create another sequence to automatically assign values to the invoice number each time a new invoice is added. The SQL code to accomplish those tasks is:

CREATE SEQUENCE CUS_CODE_SEQ START WITH 20010 NOCACHE;
CREATE SEQUENCE INV_NUMBER_SEQ START WITH 4010 NOCACHE;

You can check all of the sequences you have created by using the following SQL command, illustrated in Figure 8.22:

SELECT * FROM USER_SEQUENCES;

FIGURE 8.22 Oracle sequence



```
SQL> CREATE SEQUENCE CUS_CODE_SEQ START WITH 20010 NOCACHE;

Sequence created.

SQL> CREATE SEQUENCE INV_NUMBER_SEQ START WITH 4010 NOCACHE;

Sequence created.

SQL> SELECT * FROM USER_SEQUENCES;

SEQUENCE_NAME                   MIN_VALUE  MAX_VALUE INCREMENT_BY C O CACHE_SIZE LAST_NUMBER
------------------------------ ---------- ---------- ------------ - - ---------- -----------
CUS_CODE_SEQ                            1 1.0000E+27            1 N N          0       20010
INV_NUMBER_SEQ                          1 1.0000E+27            1 N N          0        4010

SQL>
```

To use sequences during data entry, you must use two special pseudo columns: NEXTVAL and CURRVAL. NEXTVAL retrieves the next available value from a sequence, and CURRVAL retrieves the current value of a sequence. For example, you can use the following code to enter a new customer:

```
INSERT INTO CUSTOMER
VALUES (CUS_CODE_SEQ.NEXTVAL, ‘Connery’, ‘Sean’, NULL, ‘615’, ‘898-2008’, 0.00);
```

The preceding SQL statement adds a new customer to the CUSTOMER table and assigns the value 20010 to the CUS_CODE attribute. Let’s examine some important sequence characteristics:

- CUS_CODE_SEQ.NEXTVAL retrieves the next available value from the sequence.
- Each time you use NEXTVAL, the sequence is incremented.
- Once a sequence value is used (through NEXTVAL), it cannot be used again. If, for some reason, your SQL statement rolls back, *the sequence value does not roll back*. If you issue another SQL statement (with another NEXTVAL), the next available sequence value will be returned to the user—it will look as though the sequence skips a number.
- You can issue an INSERT statement without using the sequence.

CURRVAL retrieves the current value of a sequence—that is, the last sequence number used, which was generated with a NEXTVAL. You cannot use CURRVAL unless a NEXTVAL was issued previously in the same session. The main use for CURRVAL is to enter rows in dependent tables. For example, the INVOICE and LINE tables are related in a one-to-many relationship through the INV_NUMBER attribute. You can use the INV_NUMBER_SEQ sequence to automatically generate invoice numbers. Then, using CURRVAL, you can get the latest INV_NUMBER used and assign it to the related INV_NUMBER foreign key attribute in the LINE table. For example:

```
INSERT INTO INVOICE    VALUES (INV_NUMBER_SEQ.NEXTVAL, 20010, SYSDATE);
INSERT INTO LINE       VALUES (INV_NUMBER_SEQ.CURRVAL, 1,’13-Q2/P2’, 1, 14.99);
INSERT INTO LINE       VALUES (INV_NUMBER_SEQ.CURRVAL, 2,’23109-HB’, 1, 9.95);
COMMIT;
```

The results are shown in Figure 8.23.

In the example shown in Figure 8.23, INV_NUMBER_SEQ.NEXTVAL retrieves the next available sequence number (4010) and assigns it to the INV_NUMBER column in the INVOICE table. Also note the use of the SYSDATE attribute to automatically insert the current date in the INV_DATE attribute. Next, the following two INSERT statements add the products being sold to the LINE table. In this case, INV_NUMBER_SEQ.CURRVAL refers to the last-used INV_NUMBER_SEQ sequence number (4010). In this way, the relationship between INVOICE and LINE is established automatically. The COMMIT statement at the end of the command sequence makes the changes permanent. Of course, you can also issue a ROLLBACK statement, in which case the rows you inserted in INVOICE and LINE tables would be rolled back (but remember that the sequence number would not). Once you use a sequence number (with NEXTVAL), there is no way to reuse it! This “no-reuse” characteristic is designed to guarantee that the sequence will always generate unique values.

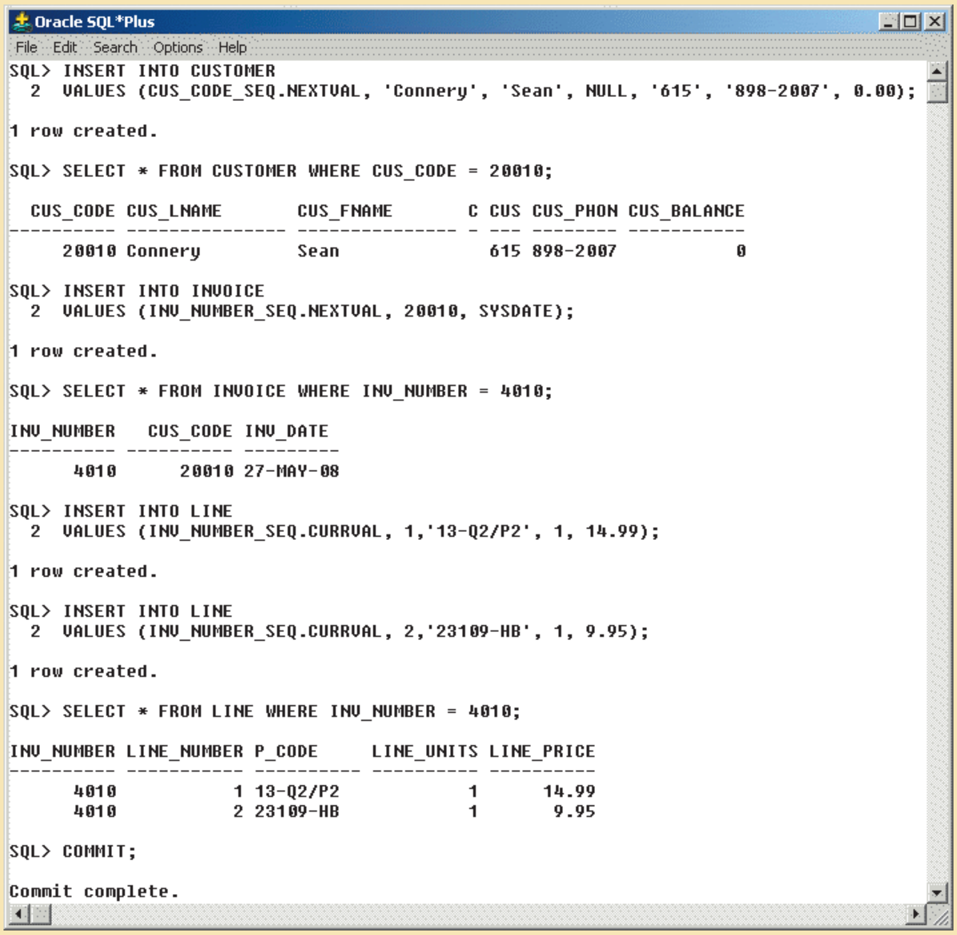
Remember these points when you think about sequences:

- The use of sequences is optional. You can enter the values manually.
- A sequence is not associated with a table. As in the examples in Figure 8.23, two distinct sequences were created (one for customer code values and one for invoice number values), but you could have created just one sequence and used it to generate unique values for both tables.

Finally, you can drop a sequence from a database with a DROP SEQUENCE command. For example, to drop the sequences created earlier, you would type:

```
DROP SEQUENCE CUS_CODE_SEQ;
DROP SEQUENCE INV_NUMBER_SEQ;
```

FIGURE 8.23 Oracle sequence examples



```
SQL> INSERT INTO CUSTOMER
  2  VALUES (CUS_CODE_SEQ.NEXTVAL, 'Connery', 'Sean', NULL, '615', '898-2007', 0.00);

1 row created.

SQL> SELECT * FROM CUSTOMER WHERE CUS_CODE = 20010;

  CUS_CODE CUS_LNAME       CUS_FNAME       C CUS CUS_PHON CUS_BALANCE
---------- --------------- --------------- - --- -------- -----------
     20010 Connery         Sean              615 898-2007           0

SQL> INSERT INTO INVOICE
  2  VALUES (INV_NUMBER_SEQ.NEXTVAL, 20010, SYSDATE);

1 row created.

SQL> SELECT * FROM INVOICE WHERE INV_NUMBER = 4010;

INV_NUMBER   CUS_CODE INV_DATE
---------- ---------- ---------
      4010      20010 27-MAY-08

SQL> INSERT INTO LINE
  2  VALUES (INV_NUMBER_SEQ.CURRVAL, 1,'13-Q2/P2', 1, 14.99);

1 row created.

SQL> INSERT INTO LINE
  2  VALUES (INV_NUMBER_SEQ.CURRVAL, 2,'23109-HB', 1, 9.95);

1 row created.

SQL> SELECT * FROM LINE WHERE INV_NUMBER = 4010;

INV_NUMBER LINE_NUMBER P_CODE     LINE_UNITS LINE_PRICE
---------- ----------- ---------- ---------- ----------
      4010           1 13-Q2/P2            1      14.99
      4010           2 23109-HB            1       9.95

SQL> COMMIT;

Commit complete.
```

NOTE

The latest SQL standard (SQL-2003) defines the use of Identity columns and sequence objects. However, some DBMS vendors might not adhere to the standard. Check your DBMS documentation.

Dropping a sequence does not delete the values you assigned to table attributes (CUS_CODE and INV_NUMBER); it deletes only the sequence object from the database. The *values* you assigned to the table columns (CUS_CODE and INV_NUMBER) remain in the database.

Because the CUSTOMER and INVOICE tables are used in the following examples, you'll want to keep the original data set. Therefore, you can delete the customer, invoice, and line rows you just added by using the following commands:

```
DELETE FROM INVOICE WHERE INV_NUMBER = 4010;
DELETE FROM CUSTOMER WHERE CUS_CODE = 20010;
COMMIT;
```

Those commands delete the recently added invoice and all of the invoice line rows associated with the invoice (the LINE table's INV_NUMBER foreign key was defined with the ON DELETE CASCADE option) and the recently added customer. The COMMIT statement saves all changes to permanent storage.

NOTE

At this point, you'll need to re-create the CUS_CODE_SEQ and INV_NUMBER_SEQ sequences, as they will be used again later in the chapter. Enter:

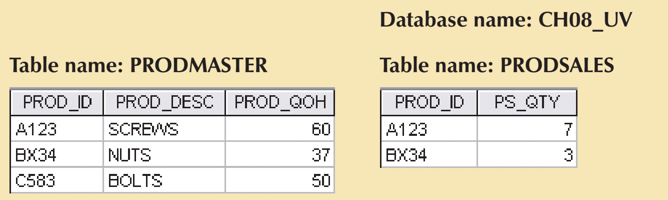
```
CREATE SEQUENCE CUS_CODE_SEQ START WITH 20010 NOCACHE;
CREATE SEQUENCE INV_NUMBER_SEQ START WITH 4010 NOCACHE;
```

8.6 UPDATABLE VIEWS

In Chapter 7, you learned how to create a view and why and how views are used. You will now take a look at how views can be made to serve common data management tasks executed by database administrators.

One of the most common operations in production database environments is using batch update routines to update a master table attribute (field) with transaction data. As the name implies, a **batch update routine** pools multiple transactions into a single batch to update a master table field *in a single operation*. For example, a batch update routine is commonly used to update a product's quantity on hand based on summary sales transactions. Such routines are typically run as overnight batch jobs to update the quantity on hand of products in inventory. The sales transactions performed, for example, by traveling salespeople were entered during periods when the system was offline.

FIGURE 8.24 The PRODMASTER and PRODSALES tables



Database name: CH08_UV

Table name: PRODMASTER

PROD_ID	PROD_DESC	PROD_QOH
A123	SCREWS	60
BX34	NUTS	37
C583	BOLTS	50

Table name: PRODSALES

PROD_ID	PS_QTY
A123	7
BX34	3

To demonstrate a batch update routine, let's begin by defining the master product table (PRODMASTER) and the product monthly sales totals table (PRODSALES) shown in Figure 8.24. Note the 1:1 relationship between the two tables.

ONLINE CONTENT

For MS Access users, the PRODMASTER and PRODSALES tables are located in the **Ch08_UV** database, which is located in the Student Online Companion.

ONLINE CONTENT

For Oracle users, all SQL commands you see in this section are located in the Student Online Companion. After you locate the script files (**uv-01.sql** through **uv-04.sql**), you can copy and paste the command sequences into your SQL*Plus program.

Using the tables in Figure 8.24, let's update the PRODMASTER table by subtracting the PRODSALES table's product monthly sales quantity (PS_QTY) from the PRODMASTER table's PROD_QOH. To produce the required update, the update query would be written like this:

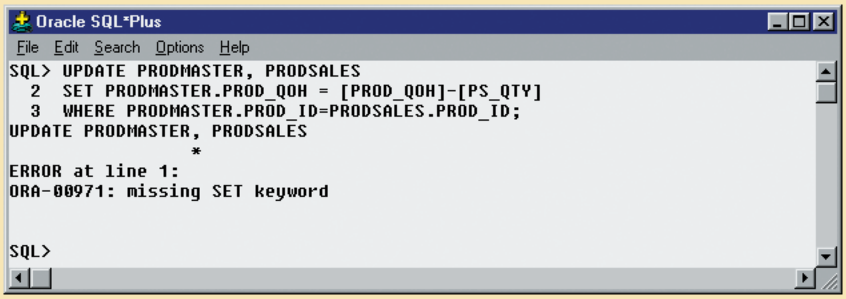
```
UPDATE    PRODMASTER, PRODSALES
SET       PRODMASTER.PROD_QOH = PROD_QOH – PS_QTY
WHERE     PRODMASTER.PROD_ID = PRODSALES.PROD_ID;
```

Note that the update statement reflects the following sequence of events:

- Join the PRODMASTER and PRODSALES tables.
- Update the PROD_QOH attribute (using the PS_QTY value in the PRODSALES table) for each row of the PRODMASTER table with matching PROD_ID values in the PRODSALES table.

To be used in a batch update, the PRODSALES data must be stored in a base table rather than in a view. That query will work fine in Access, but Oracle will return the error message shown in Figure 8.25.

FIGURE 8.25 **The Oracle UPDATE error message**



```
SQL> UPDATE PRODMASTER, PRODSALES
  2  SET PRODMASTER.PROD_QOH = [PROD_QOH]-[PS_QTY]
  3  WHERE PRODMASTER.PROD_ID=PRODSALES.PROD_ID;
UPDATE PRODMASTER, PRODSALES
                 *
ERROR at line 1:
ORA-00971: missing SET keyword


SQL>
```

Oracle produced the error message because Oracle expects to find a single table name in the UPDATE statement. In fact, you cannot join tables in the UPDATE statement in Oracle. To solve that problem, you have to create an *updatable* view. As its name suggests, an **updatable view** is a view that can be used to update attributes in the base table(s) that is (are) used in the view. You must realize that *not all views are updatable*. Actually, several restrictions govern updatable views, and some of them are vendor-specific.

NOTE

Keep in mind that the examples in this section are generated in Oracle. To see what restrictions are placed on updatable views by the DBMS you are using, check the appropriate DBMS documentation.

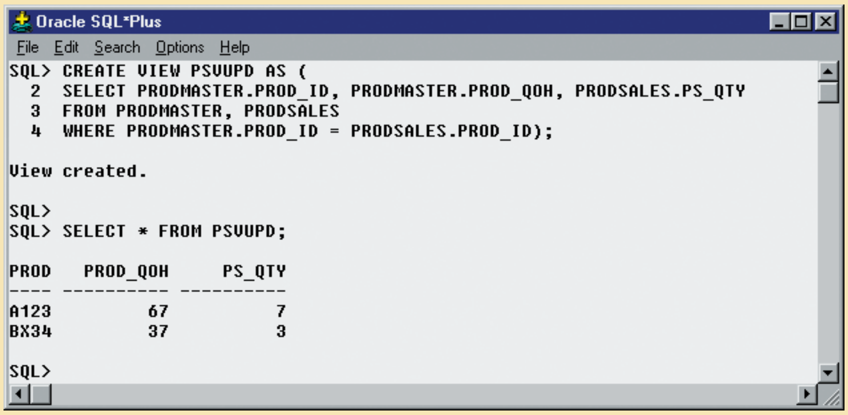
The most common updatable view restrictions are as follows:

- GROUP BY expressions or aggregate functions cannot be used.
- You cannot use set operators such as UNION, INTERSECT, and MINUS.
- Most restrictions are based on the use of JOINs or group operators in views.

To meet the Oracle limitations, an updatable view named PSVUPD has been created, as shown in Figure 8.26.

One easy way to determine whether a view can be used to update a base table is to examine the view's output. If the primary key columns of the base table you want to update still have unique values in the view, the base table is updatable. For example, if the PROD_ID column of the view returns the A123 or BX34 values more than once, the PRODMASTER table cannot be updated through the view.

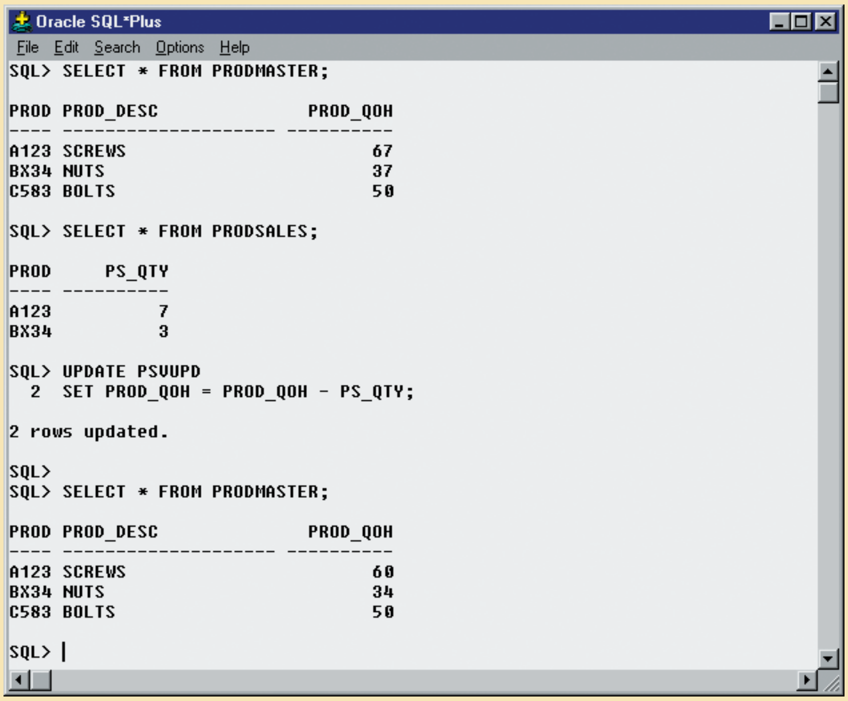
FIGURE 8.26 Creating an updatable view in Oracle



```
SQL> CREATE VIEW PSVUPD AS (
  2  SELECT PRODMASTER.PROD_ID, PRODMASTER.PROD_QOH, PRODSALES.PS_QTY
  3  FROM PRODMASTER, PRODSALES
  4  WHERE PRODMASTER.PROD_ID = PRODSALES.PROD_ID);

View created.

SQL>
SQL> SELECT * FROM PSVUPD;

PROD   PROD_QOH     PS_QTY
---- ---------- ----------
A123         67          7
BX34         37          3

SQL>
```

After creating the updatable view shown in Figure 8.26, you can use the UPDATE command to update the view, thereby updating the PRODMASTER table. Figure 8.27 shows how the UPDATE command is used and what the final contents of the PRODMASTER table are after the UPDATE has been executed.

FIGURE 8.27 PRODMASTER table update, using an updatable view



```
SQL> SELECT * FROM PRODMASTER;

PROD PROD_DESC              PROD_QOH
---- -------------------- ----------
A123 SCREWS                       67
BX34 NUTS                         37
C583 BOLTS                        50

SQL> SELECT * FROM PRODSALES;

PROD     PS_QTY
---- ----------
A123          7
BX34          3

SQL> UPDATE PSVUPD
  2  SET PROD_QOH = PROD_QOH - PS_QTY;

2 rows updated.

SQL>
SQL> SELECT * FROM PRODMASTER;

PROD PROD_DESC              PROD_QOH
---- -------------------- ----------
A123 SCREWS                       60
BX34 NUTS                         34
C583 BOLTS                        50

SQL> |
```

Although the batch update procedure just illustrated meets the goal of updating a master table with data from a transaction table, the preferred real-world solution to the update problem is to use procedural SQL, which you'll learn about next.

8.7 PROCEDURAL SQL

Thus far, you have learned to use SQL to read, write, and delete data in the database. For example, you learned to update values in a record, to add records, and to delete records. Unfortunately, SQL does not support the *conditional* execution of procedures that are typically supported by a programming language using the general format:

```
IF <condition>
    THEN <perform procedure>
        ELSE <perform alternate procedure>
END IF
```

SQL also fails to support the looping operations in programming languages that permit the execution of repetitive actions typically encountered in a programming environment. The typical format is:

```
DO WHILE
    <perform procedure>
END DO
```

Traditionally, if you wanted to perform a conditional (IF-THEN-ELSE) or looping (DO-WHILE) type of operation (that is, a procedural type of programming), you would use a programming language such as Visual Basic.Net, C#, or COBOL. That's why many older (so-called "legacy") business applications are based on enormous numbers of COBOL program lines. Although that approach is still common, it usually involves the duplication of application code in many programs. Therefore, when procedural changes are required, program modifications must be made in many different programs. An environment characterized by such redundancies often creates data management problems.

A better approach is to isolate critical code and then have all application programs call the shared code. The advantage of that modular approach is that the application code is isolated in a single program, thus yielding better maintenance and logic control. In any case, the rise of distributed databases (see Chapter 12, Distributed Database Management Systems) and object-oriented databases (see Appendix G in the Student Online Companion) required that more application code be stored and executed within the database. To meet that requirement, most RDBMS vendors created numerous programming language extensions. Those extensions include:

- Flow-control procedural programming structures (IF-THEN-ELSE, DO-WHILE) for logic representation.
- Variable declaration and designation within the procedures.
- Error management.

To remedy the lack of procedural functionality in SQL and to provide some standardization within the many vendor offerings, the SQL-99 standard defined the use of persistent stored modules. A **persistent stored module** (**PSM**) is a block of code containing standard SQL statements and procedural extensions that is stored and executed at the DBMS server. The PSM represents business logic that can be encapsulated, stored, and shared among multiple database users. A PSM lets an administrator assign specific access rights to a stored module to ensure that only authorized users can use it. Support for persistent stored modules is left to each vendor to implement. In fact, for many years, some RDBMSs (such as Oracle, SQL Server, and DB2) supported stored procedure modules within the database before the official standard was promulgated.

MS SQL Server implements persistent stored modules via Transact-SQL and other language extensions, the most notable of which are the .NET family of programming languages. Oracle implements PSMs through its procedural SQL language. **Procedural SQL** (**PL/SQL**) is a language that makes it possible to use and store procedural code and SQL statements within the database and to merge SQL and traditional programming constructs, such as variables, conditional processing (IF-THEN-ELSE), basic loops (FOR and WHILE loops,) and error trapping. The procedural code is executed as a unit by the DBMS when it is invoked (directly or indirectly) by the end user. End users can use PL/SQL to create:

- Anonymous PL/SQL blocks.
- Triggers (covered in Section 8.7.1).
- Stored procedures (covered in Section 8.7.2 and Section 8.7.3).
- PL/SQL functions (covered in Section 8.7.4).

Do not confuse PL/SQL functions with SQL's built-in aggregate functions such as MIN and MAX. SQL built-in functions can be used only within SQL statements, while PL/SQL functions are mainly invoked within PL/SQL programs such as triggers and stored procedures. Functions can also be called within SQL statements, provided they conform to very specific rules that are dependent on your DBMS environment.

NOTE

PL/SQL, triggers, and stored procedures are illustrated within the context of an Oracle DBMS. All examples in the following sections assume the use of Oracle RDBMS.

Using Oracle SQL*Plus, you can write a PL/SQL code block by enclosing the commands inside BEGIN and END clauses. For example, the following PL/SQL block inserts a new row in the VENDOR table, as shown in Figure 8.28.

```
BEGIN
        INSERT INTO VENDOR
        VALUES (25678,'Microsoft Corp. ', 'Bill Gates','765','546-8484','WA','N');
END;
/
```

The PL/SQL block shown in Figure 8.28 is known as an **anonymous PL/SQL block** because it has not been given a specific name. (Incidentally, note that the block's last line uses a forward slash ("/") to indicate the end of the command-line entry.) That type of PL/SQL block executes as soon as you press the Enter key after typing the forward slash. Following the PL/SQL block's execution, you will see the message "PL/SQL procedure successfully completed."

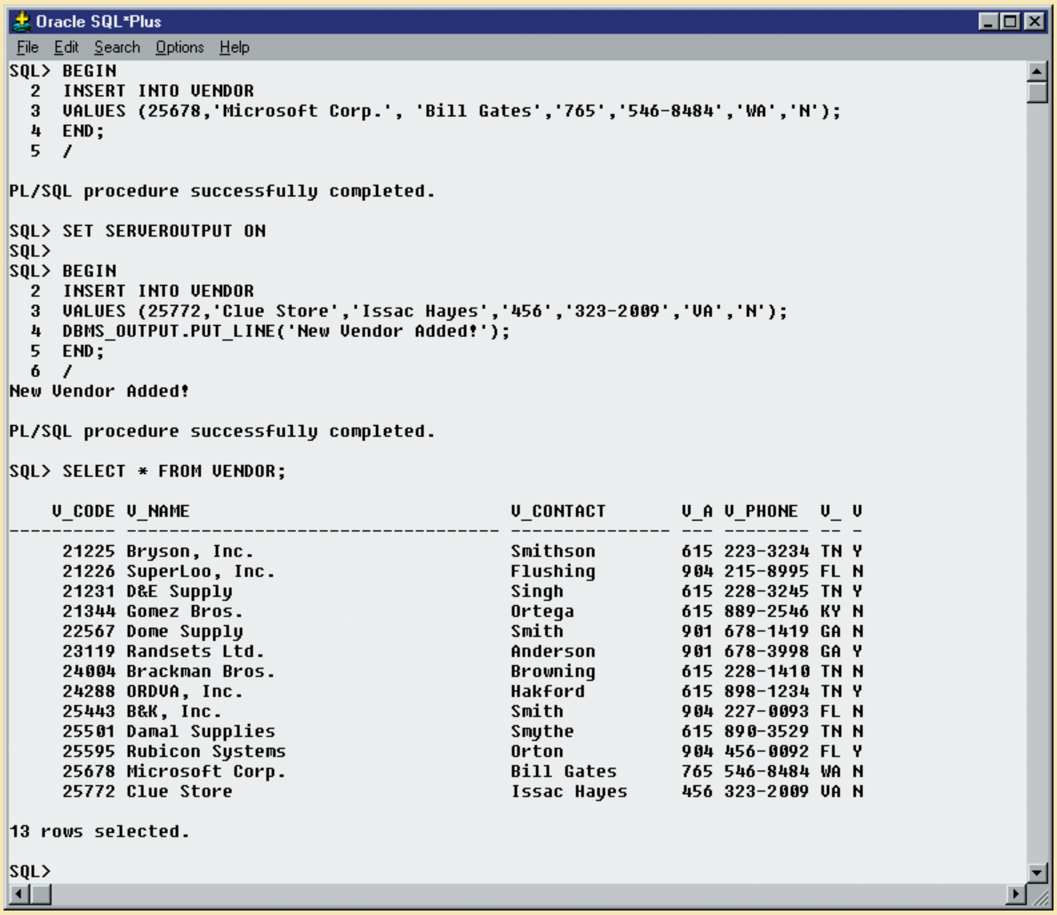
But suppose you want a more specific message displayed on the SQL*Plus screen after a procedure is completed, such as "New Vendor Added." To produce a more specific message, you must do two things:

1. At the SQL > prompt, type SET SERVEROUTPUT ON. This SQL*Plus command enables the client console (SQL*Plus) to receive messages from the server side (Oracle DBMS). Remember, just like standard SQL, the PL/SQL code (anonymous blocks, triggers, and procedures) are executed at the server side, not at the client side. (To stop receiving messages from the server, you would enter SET SERVEROUT OFF.)
2. To send messages from the PL/SQL block to the SQL*Plus console, use the DBMS_OUTPUT.PUT_LINE function.

The following anonymous PL/SQL block inserts a row in the VENDOR table and displays the message "New Vendor Added!" (See Figure 8.28).

```
BEGIN
        INSERT INTO VENDOR
        VALUES (25772,'Clue Store', 'Issac Hayes', '456','323-2009', 'VA', 'N');
        DBMS_OUTPUT.PUT_LINE('New Vendor Added!');
END;
/
```

FIGURE 8.28 Anonymous PL/SQL block examples



```
SQL> BEGIN
  2  INSERT INTO VENDOR
  3  VALUES (25678,'Microsoft Corp.', 'Bill Gates','765','546-8484','WA','N');
  4  END;
  5  /

PL/SQL procedure successfully completed.

SQL> SET SERVEROUTPUT ON
SQL>
SQL> BEGIN
  2  INSERT INTO VENDOR
  3  VALUES (25772,'Clue Store','Issac Hayes','456','323-2009','VA','N');
  4  DBMS_OUTPUT.PUT_LINE('New Vendor Added!');
  5  END;
  6  /
New Vendor Added!

PL/SQL procedure successfully completed.

SQL> SELECT * FROM VENDOR;

    V_CODE V_NAME                              V_CONTACT       V_A V_PHONE  V_ V
---------- ----------------------------------- --------------- --- -------- -- -
     21225 Bryson, Inc.                        Smithson        615 223-3234 TN Y
     21226 SuperLoo, Inc.                      Flushing        904 215-8995 FL N
     21231 D&E Supply                          Singh           615 228-3245 TN Y
     21344 Gomez Bros.                         Ortega          615 889-2546 KY N
     22567 Dome Supply                         Smith           901 678-1419 GA N
     23119 Randsets Ltd.                       Anderson        901 678-3998 GA Y
     24004 Brackman Bros.                      Browning        615 228-1410 TN N
     24288 ORDVA, Inc.                         Hakford         615 898-1234 TN Y
     25443 B&K, Inc.                           Smith           904 227-0093 FL N
     25501 Damal Supplies                      Smythe          615 890-3529 TN N
     25595 Rubicon Systems                     Orton           904 456-0092 FL Y
     25678 Microsoft Corp.                     Bill Gates      765 546-8484 WA N
     25772 Clue Store                          Issac Hayes     456 323-2009 VA N

13 rows selected.

SQL>
```

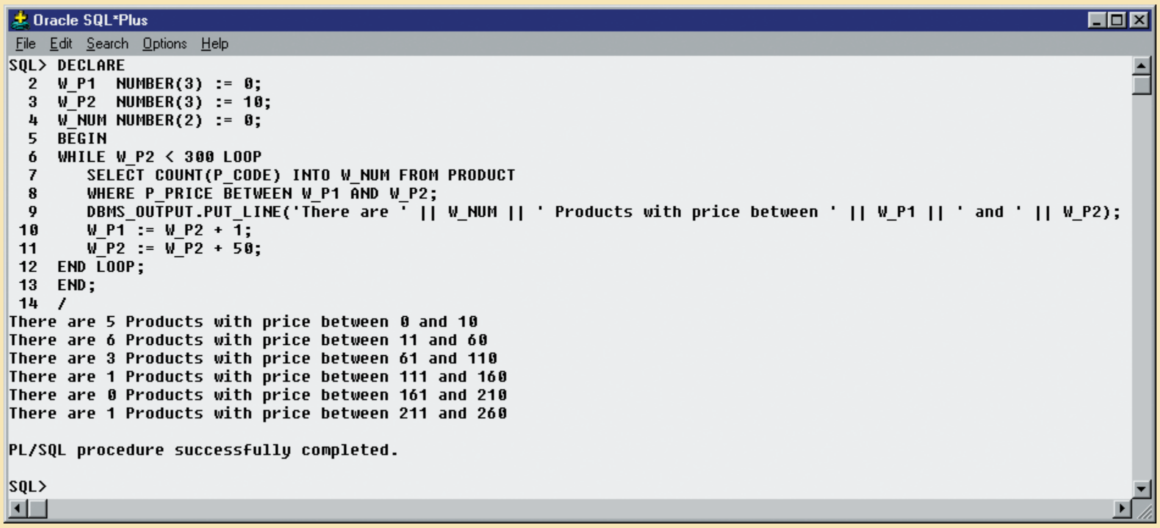
In Oracle, you can use the SQL*Plus command SHOW ERRORS to help you diagnose errors found in PL/SQL blocks. The SHOW ERRORS command yields additional debugging information whenever you generate an error after creating or executing a PL/SQL block.

The following example of an anonymous PL/SQL block demonstrates several of the constructs supported by the procedural language. Remember that the exact syntax of the language is vendor-dependent; in fact, many vendors enhance their products with proprietary features.

```
DECLARE
W_P1 NUMBER(3) := 0;
W_P2 NUMBER(3) := 10;
W_NUM NUMBER(2) := 0;
BEGIN
WHILE W_P2 < 300 LOOP
        SELECT COUNT(P_CODE) INTO W_NUM FROM PRODUCT
        WHERE P_PRICE BETWEEN W_P1 AND W_P2;
        DBMS_OUTPUT.PUT_LINE('There are ' || W_NUM || ' Products with price between ' || W_P1 ||
                                        ' and ' || W_P2);
        W_P1 := W_P2 + 1;
        W_P2 := W_P2 + 50;
END LOOP;
END;
/
```

The block's code and execution are shown in Figure 8.29.

FIGURE 8.29 **Anonymous PL/SQL block with variables and loops**



```
SQL> DECLARE
     W_P1  NUMBER(3) := 0;
     W_P2  NUMBER(3) := 10;
     W_NUM NUMBER(2) := 0;
     BEGIN
     WHILE W_P2 < 300 LOOP
        SELECT COUNT(P_CODE) INTO W_NUM FROM PRODUCT
        WHERE P_PRICE BETWEEN W_P1 AND W_P2;
        DBMS_OUTPUT.PUT_LINE('There are ' || W_NUM || ' Products with price between ' || W_P1 || ' and ' || W_P2);
        W_P1 := W_P2 + 1;
        W_P2 := W_P2 + 50;
     END LOOP;
     END;
     /
There are 5 Products with price between 0 and 10
There are 6 Products with price between 11 and 60
There are 3 Products with price between 61 and 110
There are 1 Products with price between 111 and 160
There are 0 Products with price between 161 and 210
There are 1 Products with price between 211 and 260

PL/SQL procedure successfully completed.

SQL>
```

The PL/SQL block shown in Figure 8.29 has the following characteristics:

- The PL/SQL block starts with the DECLARE section in which you declare the variable names, the data types, and, if desired, an initial value. Supported data types are shown in Table 8.8.

TABLE 8.8 **PL/SQL Basic Data Types**

DATA TYPE	DESCRIPTION
CHAR	Character values of a fixed length; for example: W_ZIPCHAR(5)
VARCHAR2	Variable length character values; for example: W_FNAMEVARCHAR2(15)
NUMBER	Numeric values; for example: W_PRICENUMBER(6,2)
DATE	Date values; for example: W_EMP_DOBDATE
%TYPE	Inherits the data type from a variable that you declared previously or from an attribute of a database table; for example: W_PRICEPRODUCT.P_PRICE%TYPE Assigns W_PRICE the same data type as the P_PRICE column in the PRODUCT table

- A WHILE loop is used. Note the syntax:

```
WHILE condition LOOP
        PL/SQL statements;
END LOOP
```

- The SELECT statement uses the INTO keyword to assign the output of the query to a PL/SQL variable. You can use the INTO keyword only inside a PL/SQL block of code. If the SELECT statement returns more than one value, you will get an error.
- Note the use of the string concatenation symbol "| |" to display the output.
- Each statement inside the PL/SQL code must end with a semicolon ";".

NOTE

PL/SQL blocks can contain only standard SQL data manipulation language (DML) commands such as SELECT, INSERT, UPDATE, and DELETE. The use of data definition language (DDL) commands is not directly supported in a PL/SQL block.

The most useful feature of PL/SQL blocks is that they let you create code that can be named, stored, and executed—either implicitly or explicitly—by the DBMS. That capability is especially desirable when you need to use triggers and stored procedures, which you will explore next.

8.7.1 TRIGGERS

Automating business procedures and automatically maintaining data integrity and consistency are critical in a modern business environment. One of the most critical business procedures is proper inventory management. For example, you want to make sure that current product sales can be supported with sufficient product availability. Therefore, it is necessary to ensure that a product order be written to a vendor when that product's inventory drops below its minimum allowable quantity on hand. Better yet, how about ensuring that the task is completed automatically?

To accomplish automatic product ordering, you first must make sure the product's quantity on hand reflects an up-to-date and consistent value. After the appropriate product availability requirements have been set, two key issues must be addressed:

1. Business logic requires an update of the product quantity on hand each time there is a sale of that product.
2. If the product's quantity on hand falls below its minimum allowable inventory (quantity-on-hand) level, the product must be reordered.

To accomplish those two tasks, you could write multiple SQL statements: one to update the product quantity on hand and another to update the product reorder flag. Next, you would have to run each statement in the correct order each time there was a new sale. Such a multistage process would be inefficient because a series of SQL statements must be written and executed each time a product is sold. Even worse, that SQL environment requires that somebody must remember to perform the SQL tasks.

A **trigger** is procedural SQL code that is *automatically* invoked by the RDBMS upon the occurrence of a given data manipulation event. It is useful to remember that:

- A trigger is invoked before or after a data row is inserted, updated, or deleted.
- A trigger is associated with a database table.
- Each database table may have one or more triggers.
- A trigger is executed as part of the transaction that triggered it.

Triggers are critical to proper database operation and management. For example:

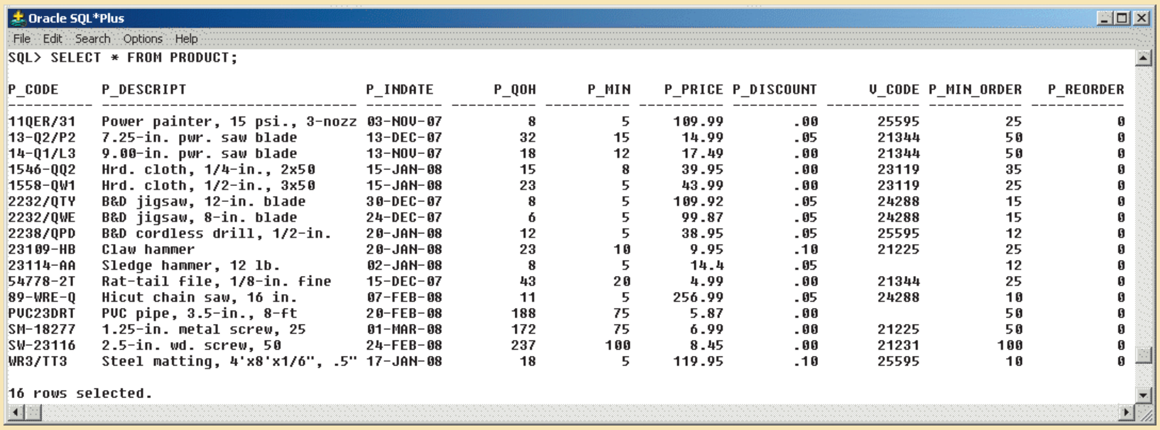
- Triggers can be used to enforce constraints that cannot be enforced at the DBMS design and implementation levels.
- Triggers add functionality by automating critical actions and providing appropriate warnings and suggestions for remedial action. In fact, one of the most common uses for triggers is to facilitate the enforcement of referential integrity.
- Triggers can be used to update table values, insert records in tables, and call other stored procedures.

Triggers play a critical role in making the database truly useful; they also add processing power to the RDBMS and to the database system as a whole. Oracle recommends triggers for:

- Auditing purposes (creating audit logs).
- Automatic generation of derived column values.
- Enforcement of business or security constraints.
- Creation of replica tables for backup purposes.

To see how a trigger is created and used, let's examine a simple inventory management problem. For example, if a product's quantity on hand is updated when the product is sold, the system should automatically check whether the quantity on hand falls below its minimum allowable quantity. To demonstrate that process, let's use the PRODUCT table in Figure 8.30. Note the use of the minimum order quantity (P_MIN_ORDER) and the product reorder flag (P_REORDER) columns. The P_MIN_ORDER indicates the minimum quantity for restocking an order. The P_REORDER column is a numeric field that indicates whether the product needs to be reordered (1 = Yes, 0 = No). The initial P_REORDER values will be set to 0 (No) to serve as the basis for the initial trigger development.

FIGURE 8.30 The PRODUCT table



```
SQL> SELECT * FROM PRODUCT;

P_CODE     P_DESCRIPT                     P_INDATE       P_QOH      P_MIN    P_PRICE P_DISCOUNT      V_CODE P_MIN_ORDER   P_REORDER
---------- ------------------------------ --------- ---------- ---------- ---------- ---------- ----------- ----------- -----------
11QER/31   Power painter, 15 psi., 3-nozz 03-NOV-07          8          5     109.99        .00       25595          25           0
13-Q2/P2   7.25-in. pwr. saw blade        13-DEC-07         32         15      14.99        .05       21344          50           0
14-Q1/L3   9.00-in. pwr. saw blade        13-NOV-07         18         12      17.49        .00       21344          50           0
1546-QQ2   Hrd. cloth, 1/4-in., 2x50      15-JAN-08         15          8      39.95        .00       23119          35           0
1558-QW1   Hrd. cloth, 1/2-in., 3x50      15-JAN-08         23          5      43.99        .00       23119          25           0
2232/QTY   B&D jigsaw, 12-in. blade       30-DEC-07          8          5     109.92        .05       24288          15           0
2232/QWE   B&D jigsaw, 8-in. blade        24-DEC-07          6          5      99.87        .05       24288          15           0
2238/QPD   B&D cordless drill, 1/2-in.    20-JAN-08         12          5      38.95        .05       25595          12           0
23109-HB   Claw hammer                    20-JAN-08         23         10       9.95        .10       21225          25           0
23114-AA   Sledge hammer, 12 lb.          02-JAN-08          8          5       14.4        .05                      12           0
54778-2T   Rat-tail file, 1/8-in. fine    15-DEC-07         43         20       4.99        .00       21344          25           0
89-WRE-Q   Hicut chain saw, 16 in.        07-FEB-08         11          5     256.99        .05       24288          10           0
PVC23DRT   PVC pipe, 3.5-in., 8-ft        20-FEB-08        188         75       5.87        .00                      50           0
SM-18277   1.25-in. metal screw, 25       01-MAR-08        172         75       6.99        .00       21225          50           0
SW-23116   2.5-in. wd. screw, 50          24-FEB-08        237        100       8.45        .00       21231         100           0
WR3/TT3    Steel matting, 4'x8'x1/6", .5" 17-JAN-08         18          5     119.95        .10       25595          10           0

16 rows selected.
```

ONLINE CONTENT

Oracle users can run the **PRODLIST.SQL** script file to format the output of the PRODUCT table shown in Figure 8.30. The script file is located in the Student Online Companion.

Given the PRODUCT table listing shown in Figure 8.30, let's create a trigger to evaluate the product's quantity on hand, P_QOH. If the quantity on hand is below the minimum quantity shown in P_MIN, the trigger will set the P_REORDER column to 1. (Remember that the number 1 in the P_REORDER column represents "Yes.") The syntax to create a trigger in Oracle is:

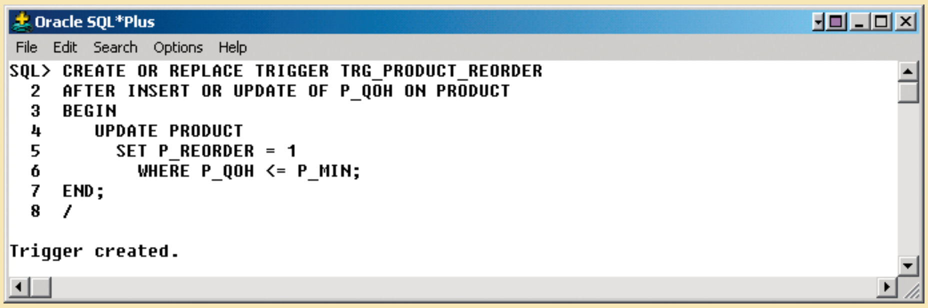
CREATE OR REPLACE TRIGGER *trigger_name*
[BEFORE / AFTER] [DELETE / INSERT / UPDATE OF *column_name*] ON *table_name*
[FOR EACH ROW]
[DECLARE]
        [*variable_namedata type*[:=*initial_value*]]
BEGIN
        PL/SQL instructions;
        ....
END;

As you can see, a trigger definition contains the following parts:

- The triggering timing: BEFORE or AFTER. This timing indicates when the trigger's PL/SQL code executes; in this case, before or after the triggering statement is completed.
- The triggering event: the statement that causes the trigger to execute (INSERT, UPDATE, or DELETE).
- The triggering level: There are two types of triggers: statement-level triggers and row-level triggers.
  - A **statement-level trigger** is assumed if you omit the FOR EACH ROW keywords. This type of trigger is executed once, before or after the triggering statement is completed. This is the default case.
  - A **row-level trigger** requires use of the FOR EACH ROW keywords. This type of trigger is executed once for each row affected by the triggering statement. (In other words, if you update 10 rows, the trigger executes 10 times.)
- The triggering action: The PL/SQL code enclosed between the BEGIN and END keywords. Each statement inside the PL/SQL code must end with a semicolon ";".

In the PRODUCT table's case, you will create a statement-level trigger that is implicitly executed AFTER an UPDATE of the P_QOH attribute for an existing row or AFTER an INSERT of a new row in the PRODUCT table. The trigger action executes an UPDATE statement that compares the P_QOH with the P_MIN column. If the value of P_QOH is equal to or less than P_MIN, the trigger updates the P_REORDER to 1. To create the trigger, Oracle's SQL*Plus will be used. The trigger code is shown in Figure 8.31.

FIGURE 8.31 Creating the TRG_PRODUCT_REORDER trigger



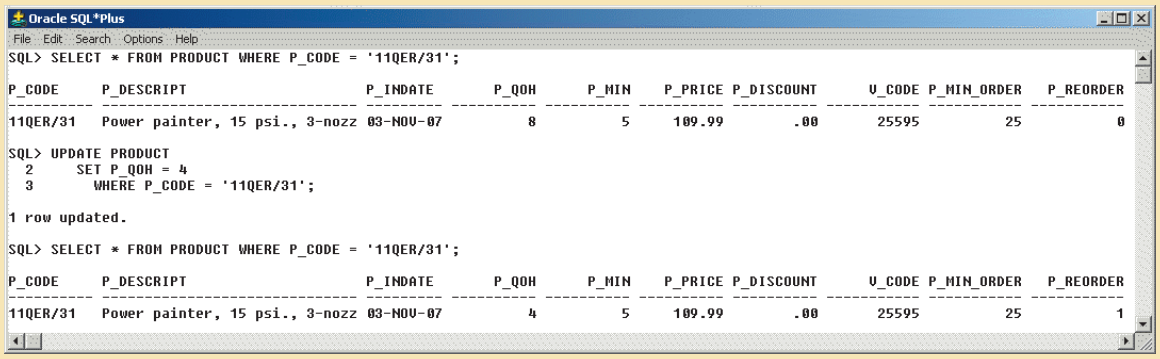
```
SQL> CREATE OR REPLACE TRIGGER TRG_PRODUCT_REORDER
     AFTER INSERT OR UPDATE OF P_QOH ON PRODUCT
     BEGIN
        UPDATE PRODUCT
           SET P_REORDER = 1
              WHERE P_QOH <= P_MIN;
     END;
     /

Trigger created.
```

Online Content

The source code for all of the triggers shown in this section can be found in the Student Online Companion.

To test the TRG_PRODUCT_REORDER trigger, let's update the quantity on hand of product '11QER/31' to 4. After the UPDATE completes, the trigger is automatically fired and the UPDATE statement (inside the trigger code) sets the P_REORDER to 1 for all products that are below the minimum. See Figure 8.32.

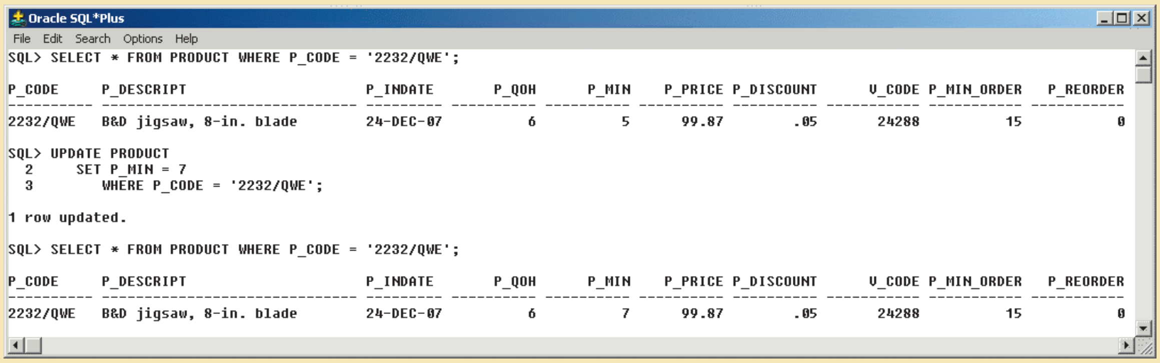
FIGURE 8.32 Verifying the TRG_PRODUCT_REORDER trigger execution



```
SQL> SELECT * FROM PRODUCT WHERE P_CODE = '11QER/31';

P_CODE     P_DESCRIPT                     P_INDATE       P_QOH      P_MIN    P_PRICE P_DISCOUNT      V_CODE P_MIN_ORDER   P_REORDER
---------- ------------------------------ --------- ---------- ---------- ---------- ---------- ----------- ----------- -----------
11QER/31   Power painter, 15 psi., 3-nozz 03-NOV-07          8          5     109.99        .00       25595          25           0

SQL> UPDATE PRODUCT
  2     SET P_QOH = 4
  3       WHERE P_CODE = '11QER/31';

1 row updated.

SQL> SELECT * FROM PRODUCT WHERE P_CODE = '11QER/31';

P_CODE     P_DESCRIPT                     P_INDATE       P_QOH      P_MIN    P_PRICE P_DISCOUNT      V_CODE P_MIN_ORDER   P_REORDER
---------- ------------------------------ --------- ---------- ---------- ---------- ---------- ----------- ----------- -----------
11QER/31   Power painter, 15 psi., 3-nozz 03-NOV-07          4          5     109.99        .00       25595          25           1
```

The trigger shown in Figure 8.32 seems to work fine, but what happens if you reduce the minimum quantity of product '2232/QWE'? Figure 8.33 shows that when you update the minimum quantity, the quantity on hand of the product '2232/QWE' falls below the new minimum, but the reorder flag is still 0. Why?

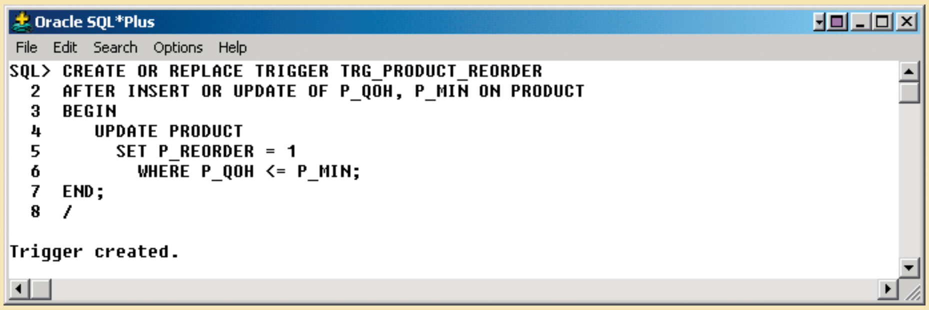
FIGURE 8.33 The P_REORDER value mismatch after update of the P_MIN attribute



```
SQL> SELECT * FROM PRODUCT WHERE P_CODE = '2232/QWE';

P_CODE     P_DESCRIPT                     P_INDATE       P_QOH      P_MIN    P_PRICE P_DISCOUNT      V_CODE P_MIN_ORDER   P_REORDER
---------- ------------------------------ --------- ---------- ---------- ---------- ---------- ----------- ----------- -----------
2232/QWE   B&D jigsaw, 8-in. blade        24-DEC-07          6          5      99.87        .05       24288          15           0

SQL> UPDATE PRODUCT
  2     SET P_MIN = 7
  3        WHERE P_CODE = '2232/QWE';

1 row updated.

SQL> SELECT * FROM PRODUCT WHERE P_CODE = '2232/QWE';

P_CODE     P_DESCRIPT                     P_INDATE       P_QOH      P_MIN    P_PRICE P_DISCOUNT      V_CODE P_MIN_ORDER   P_REORDER
---------- ------------------------------ --------- ---------- ---------- ---------- ---------- ----------- ----------- -----------
2232/QWE   B&D jigsaw, 8-in. blade        24-DEC-07          6          7      99.87        .05       24288          15           0
```

The answer is simple: you updated the P_MIN column, but the trigger is never executed. TRG_PRODUCT_REORDER executes only *after* an update of the P_QOH column! To avoid that inconsistency, you must modify the trigger event to execute after an update of the P_MIN field, too. The updated trigger code is shown in Figure 8.34.

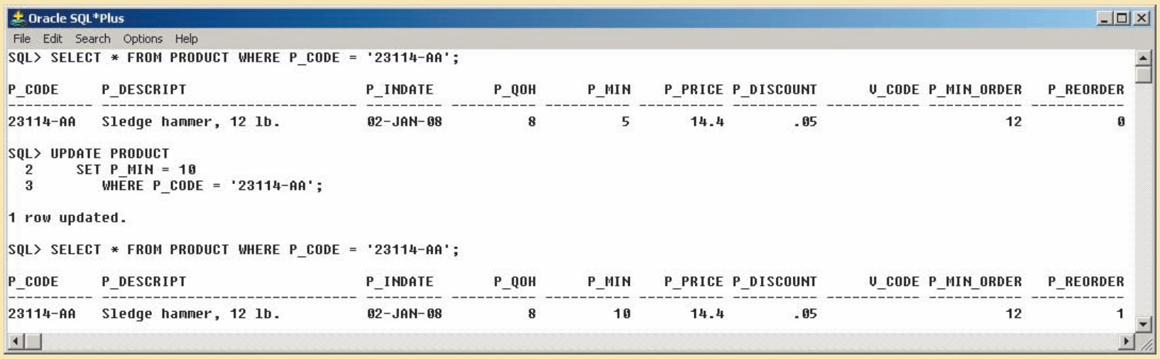
FIGURE 8.34 Second version of the TRG_PRODUCT_REORDER trigger



```
SQL> CREATE OR REPLACE TRIGGER TRG_PRODUCT_REORDER
  2  AFTER INSERT OR UPDATE OF P_QOH, P_MIN ON PRODUCT
  3  BEGIN
  4     UPDATE PRODUCT
  5       SET P_REORDER = 1
  6         WHERE P_QOH <= P_MIN;
  7  END;
  8  /

Trigger created.
```

To test this new trigger version, let's change the minimum quantity for product '23114-AA' to 8. After that update, the trigger makes sure that the reorder flag is properly set for all of the products in the PRODUCT table. See Figure 8.35.

FIGURE 8.35 Successful trigger execution after the P_MIN value is updated



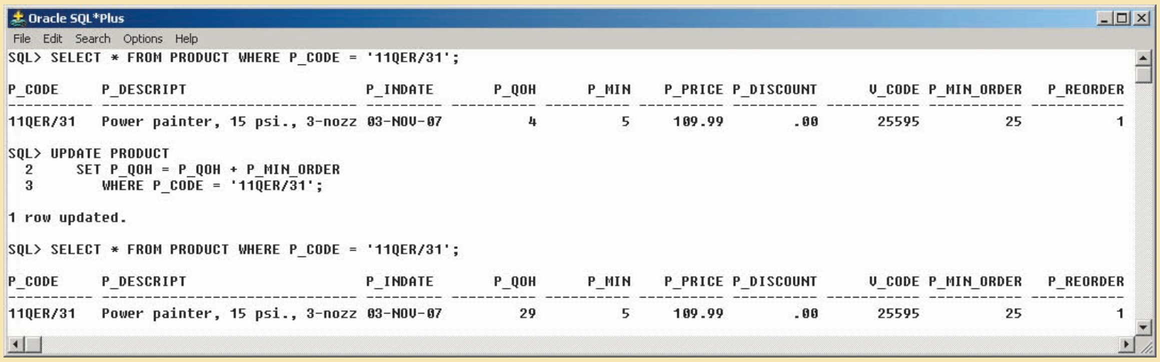
```
SQL> SELECT * FROM PRODUCT WHERE P_CODE = '23114-AA';

P_CODE     P_DESCRIPT                          P_INDATE       P_QOH      P_MIN    P_PRICE P_DISCOUNT     V_CODE P_MIN_ORDER  P_REORDER
---------- ----------------------------------- --------- ---------- ---------- ---------- ---------- ---------- ----------- ----------
23114-AA   Sledge hammer, 12 lb.               02-JAN-08          8          5       14.4        .05                     12          0

SQL> UPDATE PRODUCT
  2     SET P_MIN = 10
  3        WHERE P_CODE = '23114-AA';

1 row updated.

SQL> SELECT * FROM PRODUCT WHERE P_CODE = '23114-AA';

P_CODE     P_DESCRIPT                          P_INDATE       P_QOH      P_MIN    P_PRICE P_DISCOUNT     V_CODE P_MIN_ORDER  P_REORDER
---------- ----------------------------------- --------- ---------- ---------- ---------- ---------- ---------- ----------- ----------
23114-AA   Sledge hammer, 12 lb.               02-JAN-08          8         10       14.4        .05                     12          1
```

This second version of the trigger seems to work well, but what happens if you change the P_QOH value for product '11QER/31', as shown in Figure 8.36? Nothing! (Note that the reorder flag is *still* set to 1.) Why didn't the trigger change the reorder flag to 0?

The answer is that the trigger does not consider all possible cases. Let's examine the second version of the TRG_PRODUCT_REORDER trigger code (Figure 8.34) in more detail:

- The trigger fires after the triggering statement is completed. Therefore, the DBMS always executes two statements (INSERT plus UPDATE or UPDATE plus UPDATE). That is, after you do an update of P_MIN or P_QOH or you insert a new row in the PRODUCT table, the trigger executes another UPDATE statement automatically.
- The triggering action performs an UPDATE that updates *all* of the rows in the PRODUCT table, *even if the triggering statement updates just one row!* This can affect the performance of the database. Imagine what will happen if you have a PRODUCT table with 519,128 rows and you insert just one product. The trigger will update all 519,129 rows (519,128 original rows plus the one you inserted), including the rows that do not need an update!
- The trigger sets the P_REORDER value only to 1; it does not reset the value to 0, even if such an action is clearly required when the inventory level is back to a value greater than the minimum value.

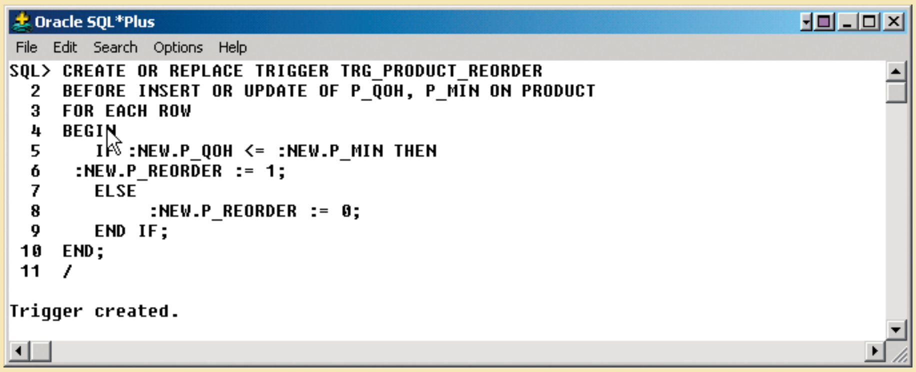
FIGURE 8.36 The P_REORDER value mismatch after increasing the P_QOH value



```
SQL> SELECT * FROM PRODUCT WHERE P_CODE = '11QER/31';

P_CODE     P_DESCRIPT                     P_INDATE       P_QOH      P_MIN    P_PRICE P_DISCOUNT      V_CODE P_MIN_ORDER   P_REORDER
---------- ------------------------------ --------- ---------- ---------- ---------- ---------- ----------- ----------- -----------
11QER/31   Power painter, 15 psi., 3-nozz 03-NOV-07          4          5     109.99        .00       25595          25           1

SQL> UPDATE PRODUCT
  2     SET P_QOH = P_QOH + P_MIN_ORDER
  3        WHERE P_CODE = '11QER/31';

1 row updated.

SQL> SELECT * FROM PRODUCT WHERE P_CODE = '11QER/31';

P_CODE     P_DESCRIPT                     P_INDATE       P_QOH      P_MIN    P_PRICE P_DISCOUNT      V_CODE P_MIN_ORDER   P_REORDER
---------- ------------------------------ --------- ---------- ---------- ---------- ---------- ----------- ----------- -----------
11QER/31   Power painter, 15 psi., 3-nozz 03-NOV-07         29          5     109.99        .00       25595          25           1
```

In short, the second version of the TRG_PRODUCT_REORDER trigger still does not complete all of the necessary steps. Now let's modify the trigger to handle all update scenarios, as shown in Figure 8.37.

FIGURE 8.37 The third version of the TRG_PRODUCT_REORDER trigger



```
SQL> CREATE OR REPLACE TRIGGER TRG_PRODUCT_REORDER
  2  BEFORE INSERT OR UPDATE OF P_QOH, P_MIN ON PRODUCT
  3  FOR EACH ROW
  4  BEGIN
  5     IF :NEW.P_QOH <= :NEW.P_MIN THEN
  6   :NEW.P_REORDER := 1;
  7     ELSE
  8          :NEW.P_REORDER := 0;
  9     END IF;
 10  END;
 11  /

Trigger created.
```

The trigger in Figure 8.37 sports several new features:

- The trigger is executed *before* the actual triggering statement is completed. In Figure 8.37, the triggering timing is defined in line 2, BEFORE INSERT OR UPDATE. This clearly indicates that the triggering statement is executed before the INSERT or UPDATE completes, unlike the previous trigger examples.
- The trigger is a row-level trigger instead of a statement-level trigger. The FOR EACH ROW keywords make the trigger a row-level trigger. Therefore, this trigger executes once for each row affected by the triggering statement.
- The trigger action uses the :NEW attribute reference to change the value of the P_REORDER attribute.

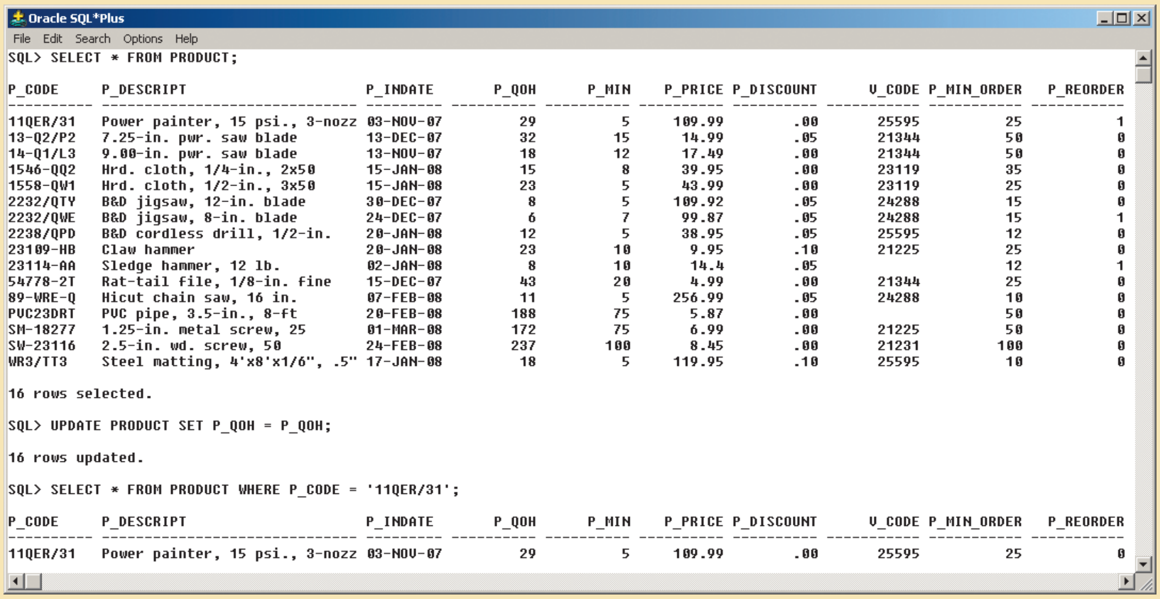
The use of the :NEW attribute references deserves a more detailed explanation. To understand its use, you must first consider a basic computing tenet: *all changes are done first in primary memory, then transferred to permanent memory*. In other words, the computer cannot change anything directly in permanent storage (disk). It must first read the data from permanent storage to primary memory; then it makes the change in primary memory; and finally, it writes the changed data back to permanent memory (disk).

The DBMS does the same thing, and one thing more. Because ensuring data integrity is critical, the DBMS makes two copies of every row being changed by a DML (INSERT, UPDATE, or DELETE) statement. (You will learn more about this in Chapter 10, Transaction Management and Concurrency Control.) The first copy contains the original ("old") values of the attributes before the changes. The second copy contains the changed ("new") values of the attributes that will be permanently saved to the database (after any changes made by an INSERT, UPDATE, or DELETE). You can use :OLD to refer to the original values; you can use :NEW to refer to the changed values (the values that will be stored in the table). You can use :NEW and :OLD attribute references only within the PL/SQL code of a database trigger action. For example:

- IF :NEW.P_QOH < = :NEW.P_MIN compares the quantity on hand with the minimum quantity of a product. Remember that this is a row-level trigger. Therefore, this comparison is done for each row that is updated by the triggering statement.
- Although the trigger is a BEFORE trigger, this does not mean that the triggering statement hasn't executed yet. To the contrary, the triggering statement has already taken place; otherwise, the trigger would not have fired and the :NEW values would not exist. Remember, BEFORE means *before* the changes are permanently saved to disk, but *after* the changes are made in memory.
- The trigger uses the :NEW reference to assign a value to the P_REORDER column before the UPDATE or INSERT results are permanently stored in the table. The assignment is always done to the :NEW value (never to the :OLD value), and the assignment always uses the " := " assignment operator. The :OLD values are *read-only* values; you cannot change them. Note that :NEW.P_REORDER := 1; assigns the value 1 to the P_REORDER column and :NEW.P_REORDER := 0; assigns the value 0 to the P_REORDER column.
- This new trigger version does not use any DML statement!

Before testing the new trigger, note that product '11QER/31' currently has a quantity on hand that is above the minimum quantity, yet the reorder flag is set to 1. Given that condition, the reorder flag must be 0. After creating the new trigger, you can execute an UPDATE statement to fire it, as shown in Figure 8.38.

FIGURE 8.38 Execution of the third trigger version



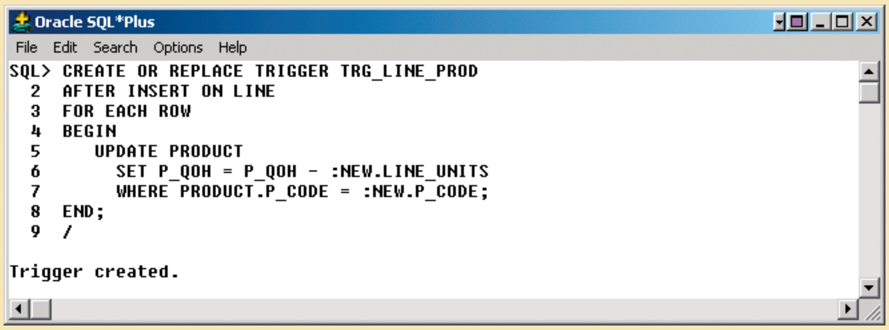
```
SQL> SELECT * FROM PRODUCT;

P_CODE     P_DESCRIPT                     P_INDATE       P_QOH      P_MIN    P_PRICE P_DISCOUNT      V_CODE P_MIN_ORDER   P_REORDER
---------- ------------------------------ --------- ---------- ---------- ---------- ---------- ----------- ----------- -----------
11QER/31   Power painter, 15 psi., 3-nozz 03-NOV-07         29          5     109.99        .00       25595          25           1
13-Q2/P2   7.25-in. pwr. saw blade        13-DEC-07         32         15      14.99        .05       21344          50           0
14-Q1/L3   9.00-in. pwr. saw blade        13-NOV-07         18         12      17.49        .00       21344          50           0
1546-QQ2   Hrd. cloth, 1/4-in., 2x50      15-JAN-08         15          8      39.95        .00       23119          35           0
1558-QW1   Hrd. cloth, 1/2-in., 3x50      15-JAN-08         23          5      43.99        .00       23119          25           0
2232/QTY   B&D jigsaw, 12-in. blade       30-DEC-07          8          5     109.92        .05       24288          15           0
2232/QWE   B&D jigsaw, 8-in. blade        24-DEC-07          6          7      99.87        .05       24288          15           1
2238/QPD   B&D cordless drill, 1/2-in.    20-JAN-08         12          5      38.95        .05       25595          12           0
23109-HB   Claw hammer                    20-JAN-08         23         10       9.95        .10       21225          25           0
23114-AA   Sledge hammer, 12 lb.          02-JAN-08          8         10       14.4        .05                      12           1
54778-2T   Rat-tail file, 1/8-in. fine    15-DEC-07         43         20       4.99        .00       21344          25           0
89-WRE-Q   Hicut chain saw, 16 in.        07-FEB-08         11          5     256.99        .05       24288          10           0
PVC23DRT   PVC pipe, 3.5-in., 8-ft        20-FEB-08        188         75       5.87        .00                      50           0
SM-18277   1.25-in. metal screw, 25       01-MAR-08        172         75       6.99        .00       21225          50           0
SW-23116   2.5-in. wd. screw, 50          24-FEB-08        237        100       8.45        .00       21231         100           0
WR3/TT3    Steel matting, 4'x8'x1/6", .5" 17-JAN-08         18          5     119.95        .10       25595          10           0

16 rows selected.

SQL> UPDATE PRODUCT SET P_QOH = P_QOH;

16 rows updated.

SQL> SELECT * FROM PRODUCT WHERE P_CODE = '11QER/31';

P_CODE     P_DESCRIPT                     P_INDATE       P_QOH      P_MIN    P_PRICE P_DISCOUNT      V_CODE P_MIN_ORDER   P_REORDER
---------- ------------------------------ --------- ---------- ---------- ---------- ---------- ----------- ----------- -----------
11QER/31   Power painter, 15 psi., 3-nozz 03-NOV-07         29          5     109.99        .00       25595          25           0
```

Note the following important features of the code in Figure 8.38:

- The trigger is automatically invoked for each affected row—in this case, all rows of the PRODUCT table. If your triggering statement would have affected only three rows, not all PRODUCT rows would have the correct P_REORDER value set. That's the reason the triggering statement was set up as shown in Figure 8.38.
- The trigger will run only if you insert a new product row or update P_QOH or P_MIN. If you update any other attribute, the trigger won't run.

You can also use a trigger to update an attribute in a table other than the one being modified. For example, suppose you would like to create a trigger that automatically reduces the quantity on hand of a product with every sale. To accomplish that task, you must create a trigger for the LINE table that updates a row in the PRODUCT table. The sample code for that trigger is shown in Figure 8.39.

FIGURE 8.39 **TRG_LINE_PROD trigger to update the PRODUCT quantity on hand**



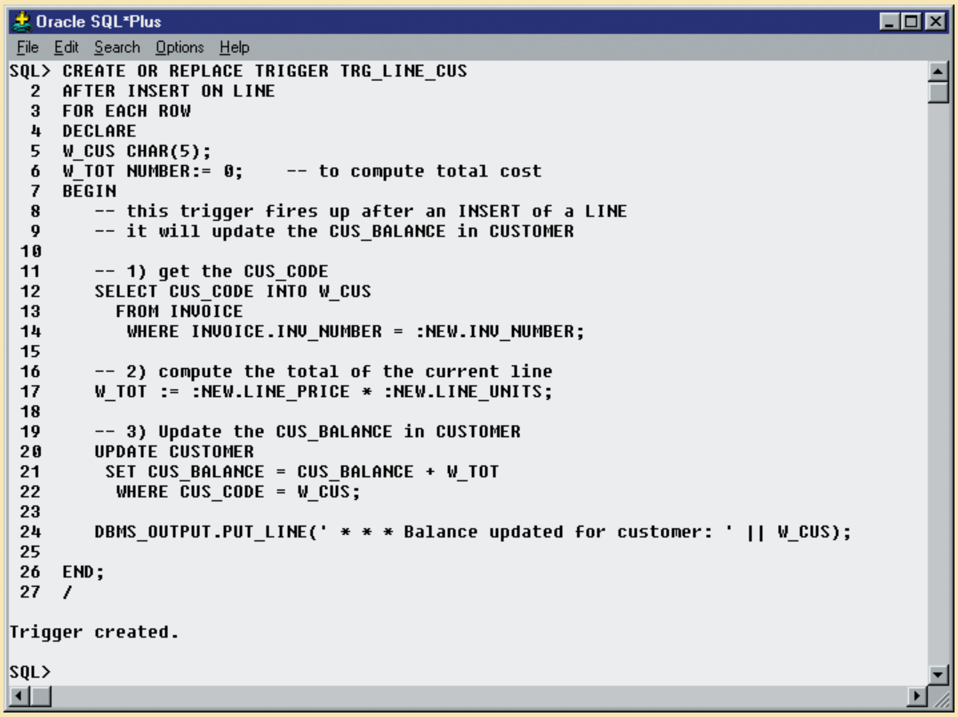
```
SQL> CREATE OR REPLACE TRIGGER TRG_LINE_PROD
     AFTER INSERT ON LINE
     FOR EACH ROW
     BEGIN
        UPDATE PRODUCT
          SET P_QOH = P_QOH - :NEW.LINE_UNITS
          WHERE PRODUCT.P_CODE = :NEW.P_CODE;
     END;
     /

Trigger created.
```

Note that the TRG_LINE_PROD row-level trigger executes after inserting a new invoice's LINE and reduces the quantity on hand of the recently sold product by the number of units sold. This row-level trigger updates a row in a different table (PRODUCT), using the :NEW values of the recently added LINE row.

A third trigger example shows the use of variables within a trigger. In this case, you want to update the customer balance (CUS_BALANCE) in the CUSTOMER table after inserting every new LINE row. This trigger code is shown in Figure 8.40.

Let's carefully examine the trigger in Figure 8.40.

- The trigger is a row-level trigger that executes after each new LINE row is inserted.
- The DECLARE section in the trigger is used to declare any variables used inside the trigger code.
- You can declare a variable by assigning a name, a data type, and (optionally) an initial value, as in the case of the W_TOT variable.
- The first step in the trigger code is to get the customer code (CUS_CODE) from the related INVOICE table. Note that the SELECT statement returns only one attribute (CUS_CODE) from the INVOICE table. Also note that that attribute returns only one value as specified by the use of the WHERE clause *to restrict the query output to a single value*.
- Note the use of the INTO clause within the SELECT statement. You use the INTO clause to assign a value from a SELECT statement to a variable (W_CUS) used within a trigger.
- The second step in the trigger code computes the total of the line by multiplying the :NEW.LINE_UNITS times :NEW.LINE_PRICE and assigning the result to the W_TOT variable.



```
SQL> CREATE OR REPLACE TRIGGER TRG_LINE_CUS
     AFTER INSERT ON LINE
     FOR EACH ROW
     DECLARE
     W_CUS CHAR(5);
     W_TOT NUMBER:= 0;    -- to compute total cost
     BEGIN
        -- this trigger fires up after an INSERT of a LINE
        -- it will update the CUS_BALANCE in CUSTOMER

        -- 1) get the CUS_CODE
        SELECT CUS_CODE INTO W_CUS
          FROM INVOICE
           WHERE INVOICE.INV_NUMBER = :NEW.INV_NUMBER;

        -- 2) compute the total of the current line
        W_TOT := :NEW.LINE_PRICE * :NEW.LINE_UNITS;

        -- 3) Update the CUS_BALANCE in CUSTOMER
        UPDATE CUSTOMER
         SET CUS_BALANCE = CUS_BALANCE + W_TOT
          WHERE CUS_CODE = W_CUS;

        DBMS_OUTPUT.PUT_LINE(' * * * Balance updated for customer: ' || W_CUS);

     END;
     /

Trigger created.

SQL>
```

FIGURE 8.40 TRG_LINE_CUS trigger to update the customer balance

- The final step updates the customer balance by using an UPDATE statement and the W_TOT and W_CUS trigger variables.
- Double dashes "--" are used to indicate comments within the PL/SQL block.

Let's summarize the triggers created in this section.

- The TRG_PROD_REORDER is a row-level trigger that updates P_REORDER in PRODUCT when a new product is added or when the P_QOH or P_MIN columns are updated.
- The TRG_LINE_PROD is a row-level trigger that automatically reduces the P_QOH in PRODUCT when a new row is added to the LINE table.
- TRG_LINE_CUS is a row-level trigger that automatically increases the CUS_BALANCE in CUSTOMER when a new row is added in the LINE table.

The use of triggers facilitates the automation of multiple data management tasks. Although triggers are independent objects, they are associated with database tables. When you delete a table, all its trigger objects are deleted with it. However, if you needed to delete a trigger without deleting the table, you could use the following command:

DROP TRIGGER *trigger_name*

Trigger Action Based on Conditional DML Predicates

You could also create triggers whose actions depend on the type of DML statement (INSERT, UPDATE, or DELETE) that fires the trigger. For example, you could create a trigger that executes after an insert, an update, or a delete on the PRODUCT table.But how do you know which one of the three statements caused the trigger to execute? In those cases, you could use the following syntax:

```
IF INSERTING THEN … END IF;
IF UPDATING THEN … END IF;
IF DELETING THEN … END IF;
```

8.7.2 Stored Procedures

A **stored procedure** is a named collection of procedural and SQL statements. Just like database triggers, stored procedures are stored in the database. One of the major advantages of stored procedures is that they can be used to encapsulate and represent business transactions. For example, you can create a stored procedure to represent a product sale, a credit update, or the addition of a new customer. By doing that, you can encapsulate SQL statements within a single stored procedure and execute them as a single transaction. There are two clear advantages to the use of stored procedures:

- Stored procedures substantially reduce network traffic and increase performance. Because the procedure is stored at the server, there is no transmission of individual SQL statements over the network. The use of stored procedures improves system performance because all transactions are executed locally on the RDBMS, so each SQL statement does not have to travel over the network.
- Stored procedures help reduce code duplication by means of code isolation and code sharing (creating unique PL/SQL modules that are called by application programs), thereby minimizing the chance of errors and the cost of application development and maintenance.

To create a stored procedure, you use the following syntax:

```
CREATE OR REPLACE PROCEDURE procedure_name [(argument [IN/OUT] data-type, … )] [IS/AS]
                                        [variable_name data type[:=initial_value] ]
BEGIN
      PL/SQL or SQL statements;
      …
END;
```

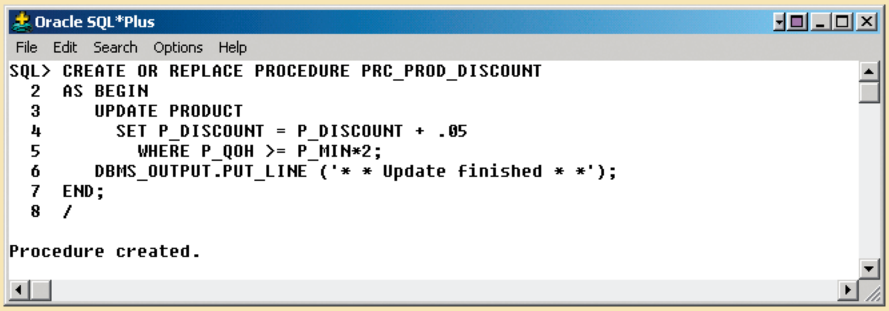
Note the following important points about stored procedures and their syntax:

- *argument* specifies the parameters that are passed to the stored procedure. A stored procedure could have zero or more arguments or parameters.
- *IN/OUT* indicates whether the parameter is for input, output, or both.
- *data-type* is one of the procedural SQL data types used in the RDBMS. The data types normally match those used in the RDBMS table-creation statement.
- Variables can be declared between the keywords IS and BEGIN. You must specify the variable name, its data type, and (optionally) an initial value.

To illustrate stored procedures, assume that you want to create a procedure (PRC_PROD_DISCOUNT) to assign an additional 5 percent discount for all products when the quantity on hand is more than or equal to twice the minimum quantity. Figure 8.41 shows how the stored procedure is created.

Note in Figure 8.41 that the PRC_PROD_DISCOUNT stored procedure uses the DBMS_OUTPUT.PUT_LINE function to display a message when the procedure executes. (This action assumes you previously ran SET SERVEROUTPUT ON.)

FIGURE 8.41 Creating the PRC_PROD_DISCOUNT stored procedure



```
SQL> CREATE OR REPLACE PROCEDURE PRC_PROD_DISCOUNT
  2  AS BEGIN
  3     UPDATE PRODUCT
  4       SET P_DISCOUNT = P_DISCOUNT + .05
  5         WHERE P_QOH >= P_MIN*2;
  6     DBMS_OUTPUT.PUT_LINE ('* * Update finished * *');
  7  END;
  8  /

Procedure created.
```

ONLINE CONTENT

The source code for all of the stored procedures shown in this section can be found in the Student Online Companion.

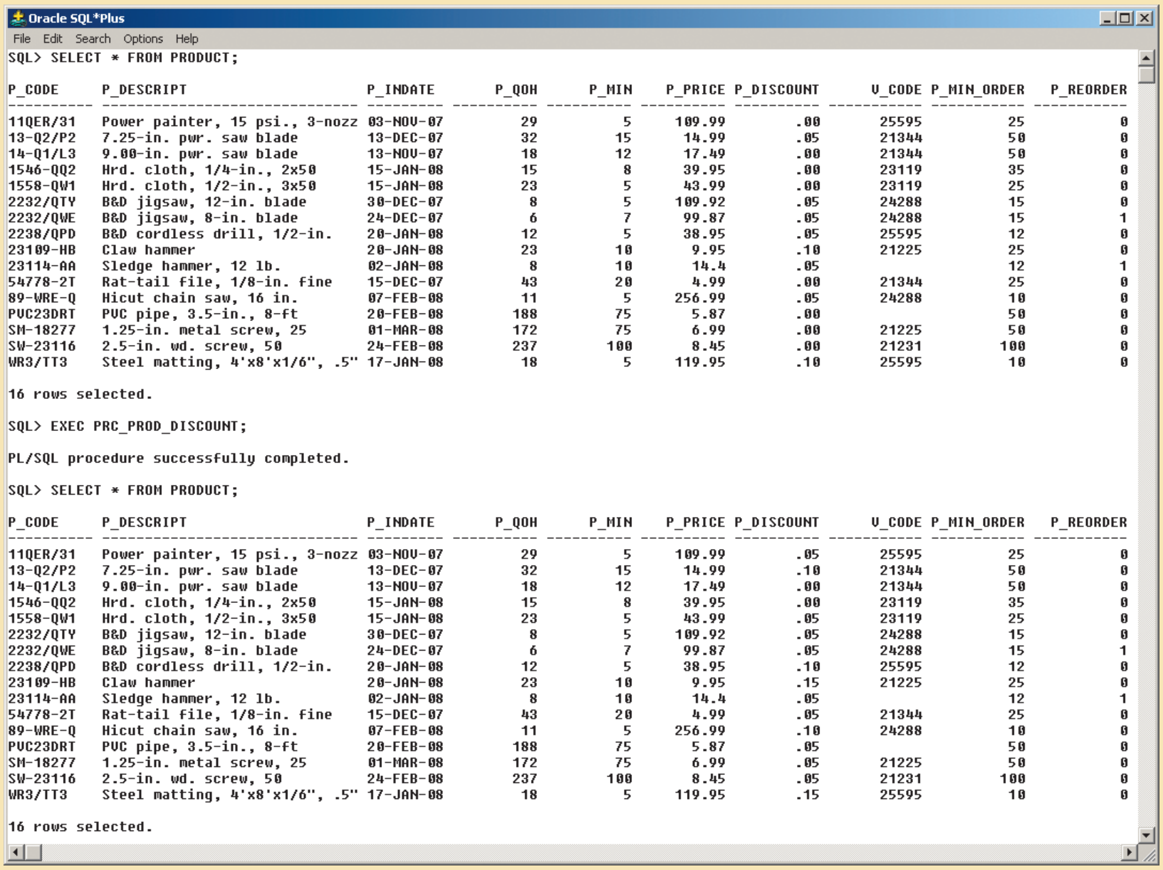
To execute the stored procedure, you must use the following syntax:

EXEC *procedure_name*[(*parameter_list*)];

For example, to see the results of running the PRC_PROD_DISCOUNT stored procedure, you can use the EXEC PRC_PROD_DISCOUNT command shown in Figure 8.42.

Using Figure 8.42 as your guide, you can see how the product discount attribute for all products with a quantity on hand more than or equal to twice the minimum quantity was increased by 5 percent. (Compare the first PRODUCT table listing to the second PRODUCT table listing.)

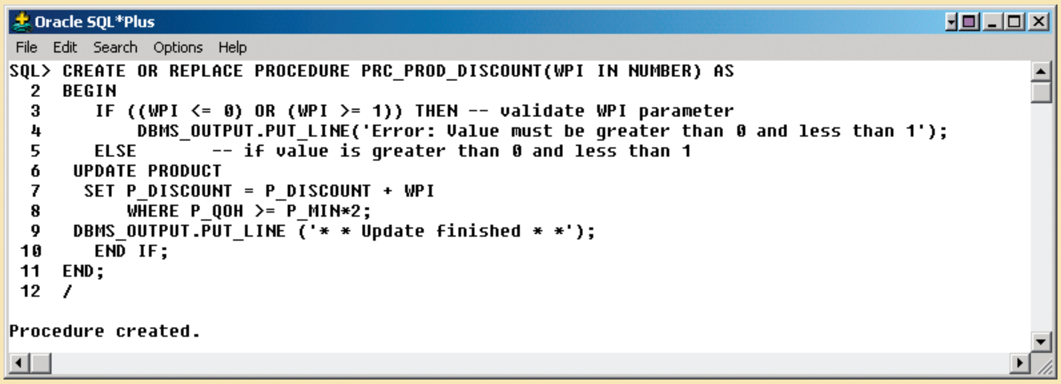
FIGURE 8.42 Results of the PRC_PROD_DISCOUNT stored procedure



```
SQL> SELECT * FROM PRODUCT;

P_CODE     P_DESCRIPT                     P_INDATE       P_QOH      P_MIN    P_PRICE P_DISCOUNT     V_CODE P_MIN_ORDER  P_REORDER
---------- ------------------------------ --------- ---------- ---------- ---------- ---------- ---------- ----------- ----------
11QER/31   Power painter, 15 psi., 3-nozz 03-NOV-07         29          5     109.99        .00      25595          25          0
13-Q2/P2   7.25-in. pwr. saw blade        13-DEC-07         32         15      14.99        .05      21344          50          0
14-Q1/L3   9.00-in. pwr. saw blade        13-NOV-07         18         12      17.49        .00      21344          50          0
1546-QQ2   Hrd. cloth, 1/4-in., 2x50      15-JAN-08         15          8      39.95        .00      23119          35          0
1558-QW1   Hrd. cloth, 1/2-in., 3x50      15-JAN-08         23          5      43.99        .00      23119          25          0
2232/QTY   B&D jigsaw, 12-in. blade       30-DEC-07          8          5     109.92        .05      24288          15          0
2232/QWE   B&D jigsaw, 8-in. blade        24-DEC-07          6          7      99.87        .05      24288          15          1
2238/QPD   B&D cordless drill, 1/2-in.    20-JAN-08         12          5      38.95        .05      25595          12          0
23109-HB   Claw hammer                    20-JAN-08         23         10       9.95        .10      21225          25          0
23114-AA   Sledge hammer, 12 lb.          02-JAN-08          8         10       14.4        .05                     12          1
54778-2T   Rat-tail file, 1/8-in. fine    15-DEC-07         43         20       4.99        .00      21344          25          0
89-WRE-Q   Hicut chain saw, 16 in.        07-FEB-08         11          5     256.99        .05      24288          10          0
PVC23DRT   PVC pipe, 3.5-in., 8-ft        20-FEB-08        188         75       5.87        .00                     50          0
SM-18277   1.25-in. metal screw, 25       01-MAR-08        172         75       6.99        .00      21225          50          0
SW-23116   2.5-in. wd. screw, 50          24-FEB-08        237        100       8.45        .00      21231         100          0
WR3/TT3    Steel matting, 4'x8'x1/6", .5" 17-JAN-08         18          5     119.95        .10      25595          10          0

16 rows selected.

SQL> EXEC PRC_PROD_DISCOUNT;

PL/SQL procedure successfully completed.

SQL> SELECT * FROM PRODUCT;

P_CODE     P_DESCRIPT                     P_INDATE       P_QOH      P_MIN    P_PRICE P_DISCOUNT     V_CODE P_MIN_ORDER  P_REORDER
---------- ------------------------------ --------- ---------- ---------- ---------- ---------- ---------- ----------- ----------
11QER/31   Power painter, 15 psi., 3-nozz 03-NOV-07         29          5     109.99        .05      25595          25          0
13-Q2/P2   7.25-in. pwr. saw blade        13-DEC-07         32         15      14.99        .10      21344          50          0
14-Q1/L3   9.00-in. pwr. saw blade        13-NOV-07         18         12      17.49        .00      21344          50          0
1546-QQ2   Hrd. cloth, 1/4-in., 2x50      15-JAN-08         15          8      39.95        .00      23119          35          0
1558-QW1   Hrd. cloth, 1/2-in., 3x50      15-JAN-08         23          5      43.99        .05      23119          25          0
2232/QTY   B&D jigsaw, 12-in. blade       30-DEC-07          8          5     109.92        .05      24288          15          0
2232/QWE   B&D jigsaw, 8-in. blade        24-DEC-07          6          7      99.87        .05      24288          15          1
2238/QPD   B&D cordless drill, 1/2-in.    20-JAN-08         12          5      38.95        .10      25595          12          0
23109-HB   Claw hammer                    20-JAN-08         23         10       9.95        .15      21225          25          0
23114-AA   Sledge hammer, 12 lb.          02-JAN-08          8         10       14.4        .05                     12          1
54778-2T   Rat-tail file, 1/8-in. fine    15-DEC-07         43         20       4.99        .05      21344          25          0
89-WRE-Q   Hicut chain saw, 16 in.        07-FEB-08         11          5     256.99        .10      24288          10          0
PVC23DRT   PVC pipe, 3.5-in., 8-ft        20-FEB-08        188         75       5.87        .05                     50          0
SM-18277   1.25-in. metal screw, 25       01-MAR-08        172         75       6.99        .05      21225          50          0
SW-23116   2.5-in. wd. screw, 50          24-FEB-08        237        100       8.45        .05      21231         100          0
WR3/TT3    Steel matting, 4'x8'x1/6", .5" 17-JAN-08         18          5     119.95        .15      25595          10          0

16 rows selected.
```

One of the main advantages of procedures is that you can pass values to them. For example, the previous PRC_PRODUCT_DISCOUNT procedure worked fine, but what if you wanted to make the percentage increase an input variable? In that case, you can pass an argument to represent the rate of increase to the procedure. Figure 8.43 shows the code for that procedure.

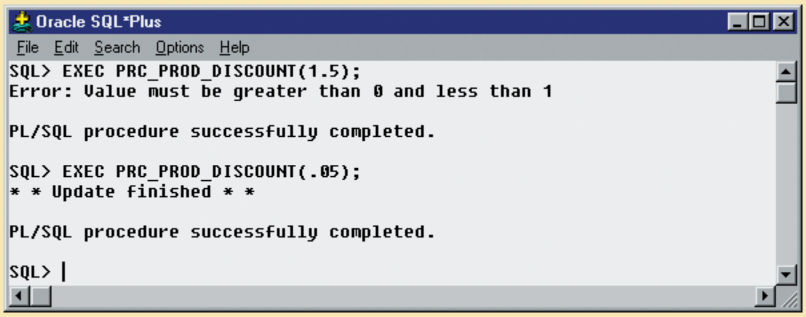
FIGURE 8.43 Second version of the PRC_PROD_DISCOUNT stored procedure



```
SQL> CREATE OR REPLACE PROCEDURE PRC_PROD_DISCOUNT(WPI IN NUMBER) AS
     BEGIN
        IF ((WPI <= 0) OR (WPI >= 1)) THEN -- validate WPI parameter
            DBMS_OUTPUT.PUT_LINE('Error: Value must be greater than 0 and less than 1');
        ELSE       -- if value is greater than 0 and less than 1
      UPDATE PRODUCT
       SET P_DISCOUNT = P_DISCOUNT + WPI
           WHERE P_QOH >= P_MIN*2;
      DBMS_OUTPUT.PUT_LINE ('* * Update finished * *');
        END IF;
     END;
     /

Procedure created.
```

Figure 8.44 shows the execution of the second version of the PRC_PROD_DISCOUNT stored procedure. Note that if the procedure requires arguments, those arguments must be enclosed in parentheses and they must be separated by commas.

FIGURE 8.44 Results of the second version of the PRC_PROD_DISCOUNT stored procedure



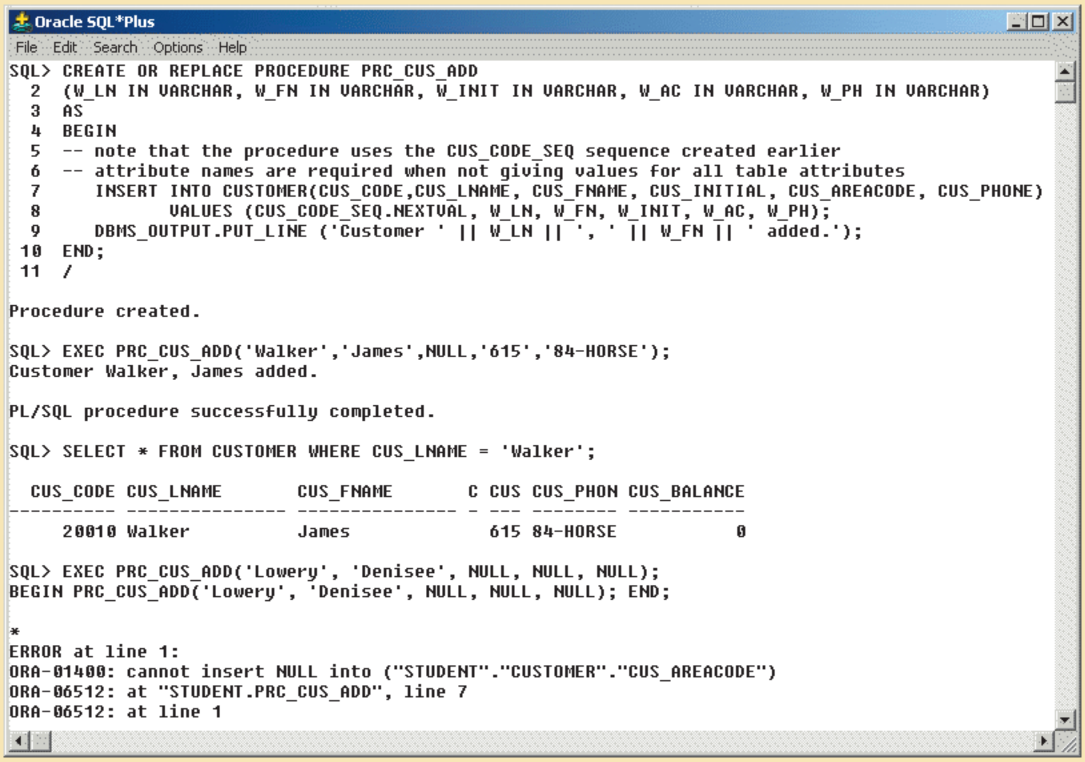
```
SQL> EXEC PRC_PROD_DISCOUNT(1.5);
Error: Value must be greater than 0 and less than 1

PL/SQL procedure successfully completed.

SQL> EXEC PRC_PROD_DISCOUNT(.05);
* * Update finished * *

PL/SQL procedure successfully completed.

SQL>
```

Stored procedures are also useful to encapsulate shared code to represent business transactions. For example, you can create a simple stored procedure to add a new customer. By using a stored procedure, all programs can call the stored procedure by name each time a new customer is added. Naturally, if new customer attributes are added later, you would need to modify the stored procedure. However, the programs that use the stored procedure would not need to know the name of the newly added attribute and would need to add only a new parameter to the procedure call. (Notice the PRC_CUS_ADD stored procedure shown in Figure 8.45.)

As you examine Figure 8.45, note these features:

- The PRC_CUS_ADD procedure uses several parameters, one for each required attribute in the CUSTOMER table.
- The stored procedure uses the CUS_CODE_SEQ sequence to generate a new customer code.

FIGURE 8.45 The PRC_CUS_ADD stored procedure



```
SQL> CREATE OR REPLACE PROCEDURE PRC_CUS_ADD
  2  (W_LN IN VARCHAR, W_FN IN VARCHAR, W_INIT IN VARCHAR, W_AC IN VARCHAR, W_PH IN VARCHAR)
  3  AS
  4  BEGIN
  5  -- note that the procedure uses the CUS_CODE_SEQ sequence created earlier
  6  -- attribute names are required when not giving values for all table attributes
  7     INSERT INTO CUSTOMER(CUS_CODE,CUS_LNAME, CUS_FNAME, CUS_INITIAL, CUS_AREACODE, CUS_PHONE)
  8            VALUES (CUS_CODE_SEQ.NEXTVAL, W_LN, W_FN, W_INIT, W_AC, W_PH);
  9     DBMS_OUTPUT.PUT_LINE ('Customer ' || W_LN || ', ' || W_FN || ' added.');
 10  END;
 11  /

Procedure created.

SQL> EXEC PRC_CUS_ADD('Walker','James',NULL,'615','84-HORSE');
Customer Walker, James added.

PL/SQL procedure successfully completed.

SQL> SELECT * FROM CUSTOMER WHERE CUS_LNAME = 'Walker';

  CUS_CODE CUS_LNAME       CUS_FNAME       C CUS CUS_PHON CUS_BALANCE
---------- --------------- --------------- - --- -------- -----------
     20010 Walker          James             615 84-HORSE           0

SQL> EXEC PRC_CUS_ADD('Lowery', 'Denisee', NULL, NULL, NULL);
BEGIN PRC_CUS_ADD('Lowery', 'Denisee', NULL, NULL, NULL); END;

*
ERROR at line 1:
ORA-01400: cannot insert NULL into ("STUDENT"."CUSTOMER"."CUS_AREACODE")
ORA-06512: at "STUDENT.PRC_CUS_ADD", line 7
ORA-06512: at line 1
```

- The required parameters—those specified in the table definition—must be included and can be null *only* when the table specifications permit nulls for that parameter. For example, note that the second customer addition was unsuccessful because the CUS_AREACODE is a required attribute and cannot be null.
- The procedure displays a message in the SQL*Plus console to let the user know that the customer was added.

The next two examples further illustrate the use of sequences within stored procedures. In this case, let's create two stored procedures:

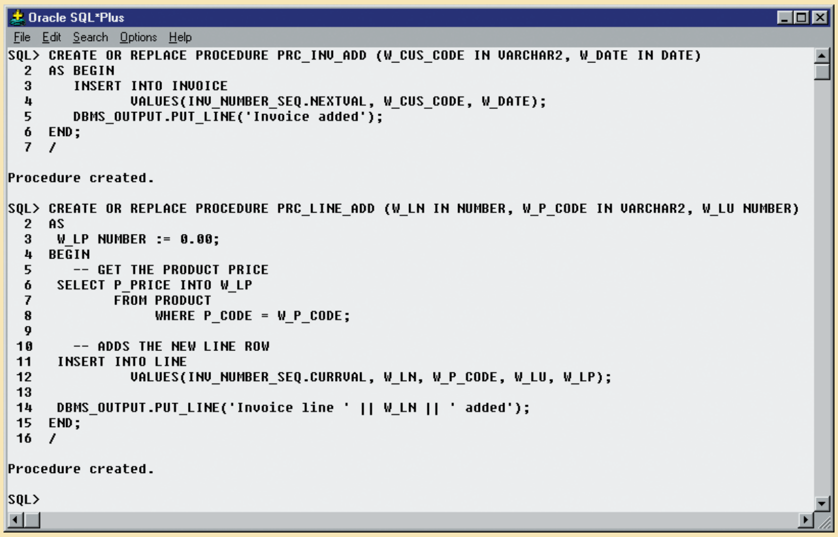
1. The PRC_INV_ADD procedure adds a new invoice.
2. The PRC_LINE_ADD procedure adds a new product line row for a given invoice.

Both procedures are shown in Figure 8.46. Note the use of a variable in the PRC_LINE_ADD procedure to get the product price from the PRODUCT table.

To test the procedures shown in Figure 8.46:

1. Call the PRC_INV_ADD procedure with the new invoice data as arguments.
2. Call the PRC_LINE_ADD procedure and pass the product line arguments.

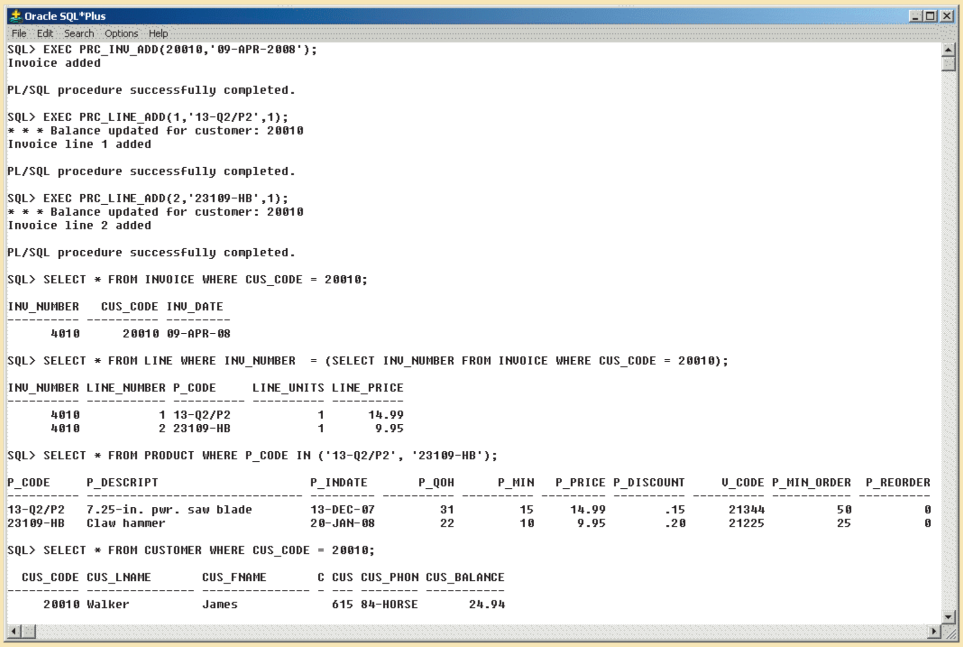
FIGURE 8.46 The PRC_INV_ADD and PRC_LINE_ADD stored procedures



```
SQL> CREATE OR REPLACE PROCEDURE PRC_INV_ADD (W_CUS_CODE IN VARCHAR2, W_DATE IN DATE)
     AS BEGIN
        INSERT INTO INVOICE
               VALUES(INV_NUMBER_SEQ.NEXTVAL, W_CUS_CODE, W_DATE);
        DBMS_OUTPUT.PUT_LINE('Invoice added');
     END;
     /

Procedure created.

SQL> CREATE OR REPLACE PROCEDURE PRC_LINE_ADD (W_LN IN NUMBER, W_P_CODE IN VARCHAR2, W_LU NUMBER)
     AS
      W_LP NUMBER := 0.00;
     BEGIN
        -- GET THE PRODUCT PRICE
      SELECT P_PRICE INTO W_LP
             FROM PRODUCT
                  WHERE P_CODE = W_P_CODE;

        -- ADDS THE NEW LINE ROW
      INSERT INTO LINE
               VALUES(INV_NUMBER_SEQ.CURRVAL, W_LN, W_P_CODE, W_LU, W_LP);

      DBMS_OUTPUT.PUT_LINE('Invoice line ' || W_LN || ' added');
     END;
     /

Procedure created.

SQL>
```

That process is illustrated in Figure 8.47.

FIGURE 8.47 Testing the PRC_INV_ADD and PRC_LINE_ADD procedures



```
SQL> EXEC PRC_INV_ADD(20010,'09-APR-2008');
Invoice added

PL/SQL procedure successfully completed.

SQL> EXEC PRC_LINE_ADD(1,'13-Q2/P2',1);
* * * Balance updated for customer: 20010
Invoice line 1 added

PL/SQL procedure successfully completed.

SQL> EXEC PRC_LINE_ADD(2,'23109-HB',1);
* * * Balance updated for customer: 20010
Invoice line 2 added

PL/SQL procedure successfully completed.

SQL> SELECT * FROM INVOICE WHERE CUS_CODE = 20010;

INV_NUMBER   CUS_CODE INV_DATE
---------- ---------- ---------
      4010      20010 09-APR-08

SQL> SELECT * FROM LINE WHERE INV_NUMBER  = (SELECT INV_NUMBER FROM INVOICE WHERE CUS_CODE = 20010);

INV_NUMBER LINE_NUMBER P_CODE     LINE_UNITS LINE_PRICE
---------- ----------- ---------- ---------- ----------
      4010           1 13-Q2/P2            1      14.99
      4010           2 23109-HB            1       9.95

SQL> SELECT * FROM PRODUCT WHERE P_CODE IN ('13-Q2/P2', '23109-HB');

P_CODE     P_DESCRIPT                     P_INDATE       P_QOH      P_MIN   P_PRICE P_DISCOUNT     V_CODE P_MIN_ORDER  P_REORDER
---------- ------------------------------ --------- ---------- ---------- --------- ---------- ---------- ----------- ----------
13-Q2/P2   7.25-in. pwr. saw blade        13-DEC-07         31         15     14.99        .15      21344          50          0
23109-HB   Claw hammer                    20-JAN-08         22         10      9.95        .20      21225          25          0

SQL> SELECT * FROM CUSTOMER WHERE CUS_CODE = 20010;

  CUS_CODE CUS_LNAME       CUS_FNAME       C CUS CUS_PHON CUS_BALANCE
---------- --------------- --------------- - --- -------- -----------
     20010 Walker          James             615 84-HORSE       24.94
```

8.7.3 PL/SQL Processing with Cursors

Until now, all of the SQL statements you have used inside a PL/SQL block (trigger or stored procedure) have returned a single value. If the SQL statement returns more than one value, you will generate an error. If you want to use an SQL statement that returns more than one value inside your PL/SQL code, you need to use a cursor. A **cursor** is a special construct used in procedural SQL to hold the data rows returned by an SQL query. You can think of a cursor as a reserved area of memory in which the output of the query is stored, like an array holding columns and rows. Cursors are held in a reserved memory area in the DBMS server, not in the client computer.

There are two types of cursors: implicit and explicit. An **implicit cursor** is automatically created in procedural SQL when the SQL statement returns only one value. Up to this point, all of the examples created an implicit cursor. An **explicit cursor** is created to hold the output of an SQL statement that may return two or more rows (but could return 0 or only one row). To create an explicit cursor, you use the following syntax inside a PL/SQL DECLARE section:

CURSOR *cursor_name* IS *select-query*;

Once you have declared a cursor, you can use specific PL/SQL cursor processing commands (OPEN, FETCH, and CLOSE) anywhere between the BEGIN and END keywords of the PL/SQL block. Table 8.9 summarizes the main use of each of those commands.

TABLE 8.9 Cursor Processing Commands

CURSOR COMMAND	EXPLANATION
OPEN	Opening the cursor executes the SQL command and populates the cursor with data, opening the cursor for processing. The cursor declaration command only reserves a named memory area for the cursor; it doesn´t populate the cursor with the data. Before you can use a cursor, you need to open it. For example: OPEN *cursor_name*
FETCH	Once the cursor is opened, you can use the FETCH command to retrieve data from the cursor and copy it to the PL/SQL variables for processing. The syntax is: FETCH cursor_name INTO variable1 [, variable2, ...] The PL/SQL variables used to hold the data must be declared in the DECLARE section and must have data types compatible with the columns retrieved by the SQL command. If the cursors SQL statement returns five columns, there must be five PL/SQL variables to receive the data from the cursor. This type of processing resembles the one-record-at-a-time processing used in previous database models. The first time you fetch a row from the cursor, the first row of data from the cursor is copied to the PL/SQL variables; the second time you fetch a row from the cursor, the second row of data is placed in the PL/SQL variables; and so on.
CLOSE	The CLOSE command closes the cursor for processing.

Cursor-style processing involves retrieving data from the cursor one row at a time. Once you open a cursor, it becomes an active data set. That data set contains a “current” row pointer. Therefore, after opening a cursor, the current row is the first row of the cursor.

When you fetch a row from the cursor, the data from the “current” row in the cursor is copied to the PL/SQL variables. After the fetch, the “current” row pointer moves to the next row in the set and continues until it reaches the end of the cursor.

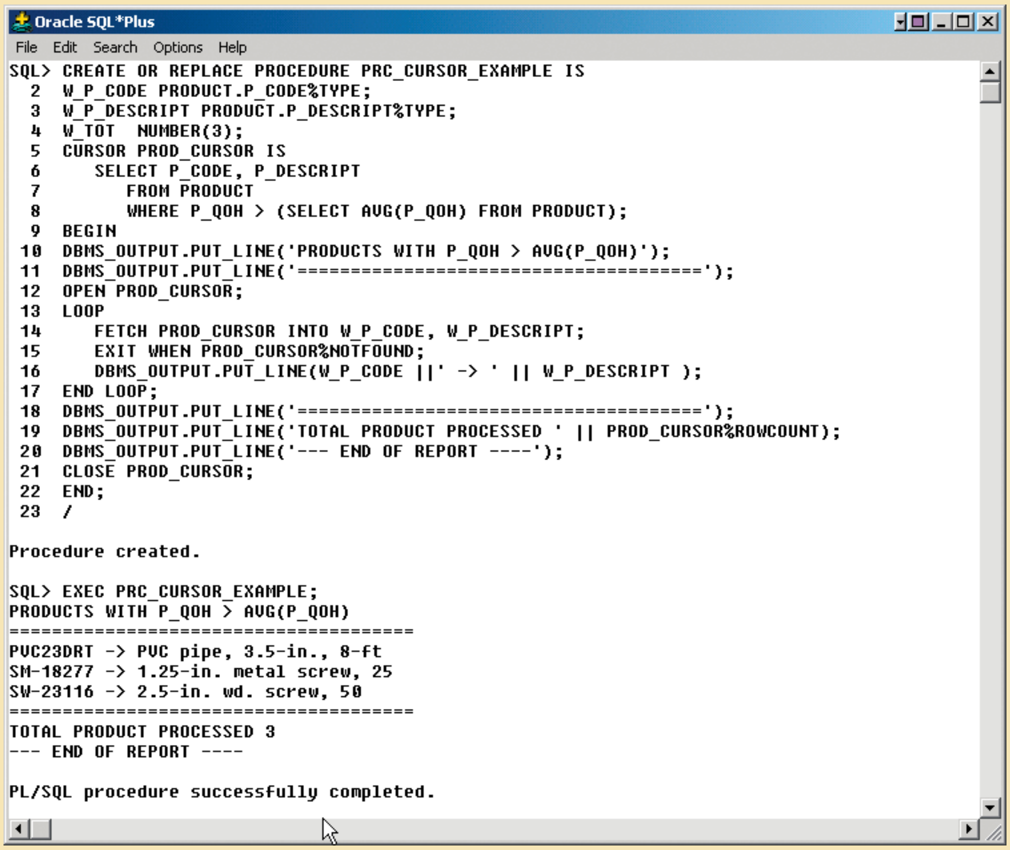
How do you know what number of rows are in the cursor? Or how do you know when you have reached the end of the cursor data set? You know because cursors have special attributes that convey important information. Table 8.10 summarizes the cursor attributes.

TABLE 8.10 Cursor Attributes

ATTRIBUTE	DESCRIPTION
%ROWCOUNT	Returns the number of rows fetched so far. If the cursor is not OPEN, it returns an error. If no FETCH has been done but the cursor is OPEN, it returns 0.
%FOUND	Returns TRUE if the last FETCH returned a row and FALSE if not. If the cursor is not OPEN, it returns an error. If no FETCH has been done, it contains NULL.
%NOTFOUND	Returns TRUE if the last FETCH did not return any row and FALSE if it did. If the cursor is not OPEN, it returns an error. If no FETCH has been done, it contains NULL.
%ISOPEN	Returns TRUE if the cursor is open (ready for processing) or FALSE if the cursor is closed. Remember, before you can use a cursor, you must open it.

To illustrate the use of cursors, let's use a simple stored procedure example that lists all products that have a quantity on hand greater than the average quantity on hand for all products. The code is shown in Figure 8.48.

FIGURE 8.48 A simple PRC_CURSOR_EXAMPLE



```
SQL> CREATE OR REPLACE PROCEDURE PRC_CURSOR_EXAMPLE IS
     W_P_CODE PRODUCT.P_CODE%TYPE;
     W_P_DESCRIPT PRODUCT.P_DESCRIPT%TYPE;
     W_TOT   NUMBER(3);
     CURSOR PROD_CURSOR IS
        SELECT P_CODE, P_DESCRIPT
           FROM PRODUCT
           WHERE P_QOH > (SELECT AVG(P_QOH) FROM PRODUCT);
     BEGIN
     DBMS_OUTPUT.PUT_LINE('PRODUCTS WITH P_QOH > AVG(P_QOH)');
     DBMS_OUTPUT.PUT_LINE('======================================');
     OPEN PROD_CURSOR;
     LOOP
        FETCH PROD_CURSOR INTO W_P_CODE, W_P_DESCRIPT;
        EXIT WHEN PROD_CURSOR%NOTFOUND;
        DBMS_OUTPUT.PUT_LINE(W_P_CODE ||' -> ' || W_P_DESCRIPT );
     END LOOP;
     DBMS_OUTPUT.PUT_LINE('======================================');
     DBMS_OUTPUT.PUT_LINE('TOTAL PRODUCT PROCESSED ' || PROD_CURSOR%ROWCOUNT);
     DBMS_OUTPUT.PUT_LINE('--- END OF REPORT ----');
     CLOSE PROD_CURSOR;
     END;
     /

Procedure created.

SQL> EXEC PRC_CURSOR_EXAMPLE;
PRODUCTS WITH P_QOH > AVG(P_QOH)
======================================
PVC23DRT -> PVC pipe, 3.5-in., 8-ft
SM-18277 -> 1.25-in. metal screw, 25
SW-23116 -> 2.5-in. wd. screw, 50
======================================
TOTAL PRODUCT PROCESSED 3
--- END OF REPORT ----

PL/SQL procedure successfully completed.
```

As you examine the stored procedure code shown in Figure 8.48, note the following important characteristics:

- Lines 2 and 3 use the %TYPE data type in the variable definition section. As indicated in Table 8.8, the %TYPE data type is used to indicate that the given variable inherits the data type from a variable previously declared or from an attribute of a database table. In this case, you are using the %TYPE to indicate that the W_P_CODE and W_P_DESCRIPT will have the same data type as the respective columns in the PRODUCT table. This way, you ensure that the PL/SQL variable will have a compatible data type.
- Line 5 declares the PROD_CURSOR cursor.
- Line 12 opens the PROD_CURSOR cursor and populates it.
- Line 13 uses the LOOP statement to loop through the data in the cursor, fetching one row at a time.
- Line 14 uses the FETCH command to retrieve a row from the cursor and place it in the respective PL/SQL variables.
- Line 15 uses the EXIT command to evaluate when there are no more rows in the cursor (using the %NOTFOUND cursor attribute) and to exit the loop.
- Line 19 uses the %ROWCOUNT cursor attribute to obtain the total number of rows processed.
- Line 21 issues the CLOSE PROD_CURSOR command to close the cursor.

The use of cursors, combined with standard SQL, makes relational databases very desirable because programmers can work in the best of both worlds: set-oriented processing and record-oriented processing. Any experienced programmer knows to use the tool that best fits the job. Sometimes you will be better off manipulating data in a set-oriented environment; at other times, it might be better to use a record-oriented environment. Procedural SQL lets you have your proverbial cake and eat it, too. Procedural SQL provides functionality that enhances the capabilities of the DBMS while maintaining a high degree of manageability.

8.7.4 PL/SQL Stored Functions

Using programmable or procedural SQL, you can also create your own stored functions. Stored procedures and functions are very similar. A **stored function** is basically a named group of procedural and SQL statements that returns a value (indicated by a RETURN statement in its program code). To create a function, you use the following syntax:

```
CREATE FUNCTION function_name (argument IN data-type, … ) RETURN data-type [IS]
BEGIN
        PL/SQL statements;
        …
        RETURN (value or expression);
END;
```

Stored functions can be invoked only from within stored procedures or triggers and cannot be invoked from SQL statements (unless the function follows some very specific compliance rules). Remember not to confuse built-in SQL functions (such as MIN, MAX, and AVG) with stored functions.

8.8 EMBEDDED SQL

There is little doubt that SQL's popularity as a data manipulation language is in part due to its ease of use and its powerful data-retrieval capabilities. But in the real world, database systems are related to other systems and programs, and you still need a conventional programming language such as Visual Basic.Net, C#, or COBOL to integrate database systems with other programs and systems. If you are developing Web applications, you are most likely familiar with Visual Studio.Net, Java, ASP, or ColdFusion. Yet, almost regardless of the programming tools you use, if your

Web application or Windows-based GUI system requires access to a database such as MS Access, SQL Server, Oracle, or DB2, you will likely need to use SQL to manipulate the data in the database.

Embedded SQL is a term used to refer to SQL statements that are contained within an application programming language such as Visual Basic.Net, C#, COBOL, or Java. The program being developed might be a standard binary executable in Windows or Linux, or it might be a Web application designed to run over the Internet. No matter what language you use, if it contains embedded SQL statements, it is called the **host language**. Embedded SQL is still the most common approach to maintaining procedural capabilities in DBMS-based applications. However, mixing SQL with procedural languages requires that you understand some key differences between SQL and procedural languages.

- *Run-time mismatch*: Remember that SQL is a nonprocedural, interpreted language; that is, each instruction is parsed, its syntax is checked, and it is executed one instruction at a time.[1] All of the processing takes place at the server side. Meanwhile, the host language is generally a binary-executable program (also known as a compiled program). The host program typically runs at the client side in its own memory space (which is different from the DBMS environment).
- *Processing mismatch*: Conventional programming languages (COBOL, ADA, FORTRAN, PASCAL, C++, and PL/I) process one data element at a time. Although you can use arrays to hold data, you still process the array elements one row at a time. This is especially true for file manipulation, where the host language typically manipulates data one record at a time. However, newer programming environments (such as Visual Studio.Net) have adopted several object-oriented extensions that help the programmer manipulate data sets in a cohesive manner.
- *Data type mismatch*: SQL provides several data types, but some of those data types might not match data types used in different host languages (for example, the date and varchar2 data types).

To bridge the differences, the Embedded SQL standard[2] defines a framework to integrate SQL within several programming languages. The Embedded SQL framework defines the following:

- A standard syntax to identify embedded SQL code within the host language (EXEC SQL/END-EXEC).
- A standard syntax to identify host variables. Host variables are variables in the host language that receive data from the database (through the embedded SQL code) and process the data in the host language. All host variables are preceded by a colon (":").
- A communication area used to exchange status and error information between SQL and the host language. This communications area contains two variables—SQLCODE and SQLSTATE.

Another way to interface host languages and SQL is through the use of a call level interface (CLI)[3] , in which the programmer writes to an application programming interface (API). A common CLI in Windows is provided by the Open Database Connectivity (ODBC) interface.

ONLINE CONTENT

Additional coverage of CLIs and ODBC is found in **Appendix F, Client/Server Systems**, and **Appendix J, Web Database Development with ColdFusion** in the Student Online Companion.

[1] The authors are particularly grateful for the thoughtful comments provided by Emil T. Cipolla, who teaches at Mount Saint Mary College and whose IBM experience is the basis for his considerable and practical expertise.

[2] You can obtain more details about the Embedded SQL standard at *www.ansi.org*, SQL/Bindings is in the SQL Part II – SQL/Foundation section of the SQL 2003 standard.

[3] You can find additional information about the SQL Call Level Interface standard at *www.ansi.org*, in the SQL Part 3: Call Level Interface (SQL/CLI) section of the SQL 2003 standard.

Before continuing, let's explore the process required to create and run an executable program with embedded SQL statements. If you have ever programmed in COBOL or C++, you are familiar with the multiple steps required to generate the final executable program. Although the specific details vary among language and DBMS vendors, the following general steps are standard:

1. The programmer writes embedded SQL code within the host language instructions. The code follows the standard syntax required for the host language and embedded SQL.
2. A preprocessor is used to transform the embedded SQL into specialized procedure calls that are DBMS- and language-specific. The preprocessor is provided by the DBMS vendor and is specific to the host language.
3. The program is compiled using the host language compiler. The compiler creates an object code module for the program containing the DBMS procedure calls.
4. The object code is linked to the respective library modules and generates the executable program. This process binds the DBMS procedure calls to the DBMS run-time libraries. Additionally, the binding process typically creates an "access plan" module that contains instructions to run the embedded code at run time.
5. The executable is run, and the embedded SQL statement retrieves data from the database.

Note that you can embed individual SQL statements or even an entire PL/SQL block. Up to this point in the book, you have used a DBMS-provided application (SQL*Plus) to write SQL statements and PL/SQL blocks in an interpretive mode to address one-time or ad hoc data requests. However, it is extremely difficult and awkward to use ad hoc queries to process transactions inside a host language. Programmers typically embed SQL statements within a host language that it is compiled once and executed as often as needed. To embed SQL into a host language, follow this syntax:

```
EXEC SQL
      SQL statement;
END-EXEC.
```

The preceding syntax will work for SELECT, INSERT, UPDATE, and DELETE statements. For example, the following embedded SQL code will delete employee 109, George Smith, from the EMPLOYEE table:

```
EXEC SQL
      DELETE FROM EMPLOYEE WHERE EMP_NUM = 109;
END-EXEC.
```

Remember, the preceding embedded SQL statement is compiled to generate an executable statement. Therefore, the statement is fixed permanently and cannot change (unless, of course, the programmer changes it). Each time the program runs, it deletes the same row. In short, the preceding code is good only for the first run; all subsequent runs will likely generate an error. Clearly, this code would be more useful if you could specify a variable to indicate the employee number to be deleted.

In embedded SQL, all host variables are preceded by a colon (":"). The host variables may be used to send data from the host language to the embedded SQL, or they may be used to receive the data from the embedded SQL. To use a host variable, you must first declare it in the host language. Common practice is to use similar host variable names as the SQL source attributes. For example, if you are using COBOL, you would define the host variables in the Working Storage section. Then you would refer to them in the embedded SQL section by preceding them with a colon (":"). For example, to delete an employee whose employee number is represented by the host variable W_EMP_NUM, you would write the following code:

```
EXEC SQL
      DELETE FROM EMPLOYEE WHERE EMP_NUM = :W_EMP_NUM;
END-EXEC.
```

At run time, the host variable value will be used to execute the embedded SQL statement. What happens if the employee you are trying to delete doesn't exist in the database? How do you know that the statement has been completed without errors? As mentioned previously, the embedded SQL standard defines a SQL communication area to hold status and error information. In COBOL, such an area is known as the SQLCA area and is defined in the Data Division as follows:

```
EXEC SQL
        INCLUDE SQLCA
END-EXEC.
```

The SQLCA area contains two variables for status and error reporting. Table 8.11 shows some of the main values returned by the variables and their meaning.

TABLE 8.11 SQL Status and Error Reporting Variables

VARIABLE NAME	VALUE	EXPLANATION
SQLCODE		Old-style error reporting supported for backward compatibility only; returns an integer value (positive or negative).
	0	Successful completion of command.
	100	No data; the SQL statement did not return any rows or did not select, update, or delete any rows.
	-999	Any negative value indicates that an error occurred.
SQLSTATE		Added by SQL-92 standard to provide predefined error codes; defined as a character string (5 characters long).
	00000	Successful completion of command.
		Multiple values in the format XXYYY where: XX-> represents the class code. YYY-> represents the subclass code.

The following embedded SQL code illustrates the use of the SQLCODE within a COBOL program.

```
EXEC SQL
EXEC SQL
        SELECT       EMP_LNAME, EMP_LNAME INTO :W_EMP_FNAME, :W_EMP_LNAME
                     WHERE EMP_NUM = :W_EMP_NUM;
END-EXEC.
IF SQLCODE = 0 THEN
        PERFORM DATA_ROUTINE
ELSE
        PERFORM ERROR_ROUTINE
END-IF.
```

In this example, the SQLCODE host variable is checked to determine whether the query completed successfully. If that is the case, the DATA_ROUTINE is performed; otherwise, the ERROR_ROUTINE is performed.

Just as with PL/SQL, embedded SQL requires the use of cursors to hold data from a query that returns more than one value. If COBOL is used, the cursor can be declared either in the Working Storage Section or in the Procedure Division. The cursor must be declared and processed as you learned earlier in Section 8.7.3. To declare a cursor, you use the syntax shown in the following example:

```
EXEC SQL
      DECLARE PROD_CURSOR FOR
            SELECT      P_CODE, P_DESCRIPT FROM PRODUCT
            WHERE       P_QOH > (SELECT AVG(P_QOH) FROM PRODUCT);
END-EXEC.
```

Next, you must open the cursor to make it ready for processing:

```
EXEC SQL
      OPEN PROD_CURSOR;
END-EXEC.
```

To process the data rows in the cursor, you use the FETCH command to retrieve one row of data at a time and place the values in the host variables. The SQLCODE must be checked to ensure that the FETCH command completed successfully. This section of code typically constitutes part of a routine in the COBOL program. Such a routine is executed with the PERFORM command. For example:

```
EXEC SQL
      FETCH PROD_CURSOR INTO :W_P_CODE, :W_P_DESCRIPT;
END-EXEC.
IF SQLCODE = 0 THEN
      PERFORM DATA_ROUTINE
ELSE
      PERFORM ERROR_ROUTINE
END-IF.
```

When all rows have been processed, you close the cursor as follows:

```
EXEC SQL
      CLOSE PROD_CURSOR;
END-EXEC.
```

Thus far, you have seen examples of embedded SQL in which the programmer used predefined SQL statements and parameters. Therefore, the end users of the programs are limited to the actions that were specified in the application programs. That style of embedded SQL is known as **static SQL**, meaning that the SQL statements will not change while the application is running. For example, the SQL statement might read like this:

```
SELECT      P_CODE, P_DESCRIPT, P_QOH, P_PRICE
FROM        PRODUCT
WHERE       P_PRICE > 100;
```

Note that the attributes, tables, and conditions are known in the preceding SQL statement. Unfortunately, end users seldom work in a static environment. They are more likely to require the flexibility of defining their data access requirements on the fly. Therefore, the end user requires that SQL be as dynamic as the data access requirements.

Dynamic SQL is a term used to describe an environment in which the SQL statement is not known in advance; instead, the SQL statement is generated at run time. At run time in a dynamic SQL environment, a program can generate the SQL statements that are required to respond to ad hoc queries. In such an environment, neither the programmer nor the end user is likely to know precisely what kind of queries are to be generated or how those queries are to be structured. For example, a dynamic SQL equivalent of the preceding example could be:

```
SELECT      :W_ATTRIBUTE_LIST
FROM        :W_TABLE
WHERE       :W_CONDITION;
```

Note that the attribute list and the condition are not known until the end user specifies them. W_TABLE, W_ATRIBUTE_LIST, and W_CONDITION are text variables that contain the end-user input values used in the query generation. Because the program uses the end-user input to build the text variables, the end user can run the same program multiple times to generate varying outputs. For example, in one instance, the end user might want to know what products have a price less than $100; in another case, the end user might want to know how many units of a given product are available for sale at any given moment.

Although dynamic SQL is clearly flexible, such flexibility carries a price. Dynamic SQL tends to be much slower than static SQL. Dynamic SQL also requires more computer resources (overhead). Finally, you are more likely to find inconsistent levels of support and incompatibilities among DBMS vendors.

SUMMARY

- SQL provides relational set operators to combine the output of two queries to generate a new relation. The UNION and UNION ALL set operators combine the output of two (or more) queries and produce a new relation with all unique (UNION) or duplicate (UNION ALL) rows from both queries. The INTERSECT relational set operator selects only the common rows. The MINUS set operator selects only the rows that are different. UNION, INTERSECT, and MINUS require union-compatible relations.
- Operations that join tables can be classified as inner joins and outer joins. An inner join is the traditional join in which only rows that meet a given criteria are selected. An outer join returns the matching rows as well as the rows with unmatched attribute values for one table or both tables to be joined.
- A natural join returns all rows with matching values in the matching columns and eliminates duplicate columns. This style of query is used when the tables share a common attribute with a common name. One important difference between the syntax for a natural join and for the "old-style" join is that the natural join does not require the use of a table qualifier for the common attributes.
- Joins may use keywords such as USING and ON. If the USING clause is used, the query will return only the rows with matching values in the column indicated in the USING clause; that column must exist in both tables. If the ON clause is used, the query will return only the rows that meet the specified join condition.
- Subqueries and correlated queries are used when it is necessary to process data based on *other* processed data. That is, the query uses results that were previously unknown and that are generated by another query. Subqueries may be used with the FROM, WHERE, IN, and HAVING clauses in a SELECT statement. A subquery may return a single row or multiple rows.
- Most subqueries are executed in a serial fashion. That is, the outer query initiates the data request, and then the inner subquery is executed. In contrast, a correlated subquery is a subquery that is executed once for each row in the outer query. That process is similar to the typical nested loop in a programming language. A correlated subquery is so named because the inner query is related to the outer query—the inner query references a column of the outer subquery.
- SQL functions are used to extract or transform data. The most frequently used functions are date and time functions. The results of the function output can be used to store values in a database table, to serve as the basis for the computation of derived variables, or to serve as a basis for data comparisons. Function formats can be vendor-specific. Aside from time and date functions, there are numeric and string functions as well as conversion functions that convert one data format to another.
- Oracle sequences may be used to generate values to be assigned to a record. For example, a sequence may be used to number invoices automatically. MS Access uses an AutoNumber data type to generate numeric sequences. MS SQL Server uses the Identity column property to designate the column that will have sequential numeric values automatically assigned to it. There can only be one Identity column per SQL Server table.
- Procedural SQL (PL/SQL) can be used to create triggers, stored procedures, and PL/SQL functions. A trigger is procedural SQL code that is automatically invoked by the DBMS upon the occurrence of a specified data manipulation event (UPDATE, INSERT, or DELETE). Triggers are critical to proper database operation and management. They help automate various transaction and data management processes, and they can be used to enforce constraints that are not enforced at the DBMS design and implementation levels.
- A stored procedure is a named collection of SQL statements. Just like database triggers, stored procedures are stored in the database. One of the major advantages of stored procedures is that they can be used to encapsulate and represent complete business transactions. Use of stored procedures substantially reduces network traffic and increases system performance. Stored procedures help reduce code duplication by creating unique PL/SQL

modules that are called by the application programs, thereby minimizing the chance of errors and the cost of application development and maintenance.

- When SQL statements are designed to return more than one value inside the PL/SQL code, a cursor is needed. You can think of a cursor as a reserved area of memory in which the output of the query is stored, like an array holding columns and rows. Cursors are held in a reserved memory area in the DBMS server, rather than in the client computer. There are two types of cursors: implicit and explicit.
- Embedded SQL refers to the use of SQL statements within an application programming language such as Visual Basic.Net, C#, COBOL, or Java. The language in which the SQL statements are embedded is called the host language. Embedded SQL is still the most common approach to maintaining procedural capabilities in DBMS-based applications.

Key Terms

anonymous PL/SQL block, 339
batch update routine, 335
correlated subquery, 321
cross join, 306
cursor, 357
dynamic SQL, 364
embedded SQL, 360
explicit cursor, 357
host language, 360
implicit cursor, 357
inner join, 305
outer join, 305
persistent stored module (PSM), 338
procedural SQL (PL/SQL), 338
row-level trigger, 344
statement-level trigger, 344
static SQL, 363
stored function, 359
stored procedure, 359
trigger, 342
union-compatible, 298
updatable view, 336

Online Content

Answers to selected Review Questions and Problems for this chapter are contained in the Student Online Companion for this book.

Review Questions

1. The relational set operators UNION, INTERSECT, and MINUS work properly only when the relations are union-compatible. What does *union-compatible* mean, and how would you check for this condition?
2. What is the difference between UNION and UNION ALL? Write the syntax for each.
3. Suppose you have two tables: EMPLOYEE and EMPLOYEE_1. The EMPLOYEE table contains the records for three employees: Alice Cordoza, John Cretchakov, and Anne McDonald. The EMPLOYEE_1 table contains the records for employees John Cretchakov and Mary Chen. Given that information, list the query output for the UNION query.
4. Given the employee information in Question 3, list the query output for the UNION ALL query.
5. Given the employee information in Question 3, list the query output for the INTERSECT query.
6. Given the employee information in Question 3, list the query output for the MINUS query.
7. What is a CROSS JOIN? Give an example of its syntax.
8. What three join types are included in the OUTER JOIN classification?
9. Using tables named T1 and T2, write a query example for each of the three join types you described in Question 8. Assume that T1 and T2 share a common column named C1.

10. What is a subquery, and what are its basic characteristics?
11. What is a correlated subquery? Give an example.
12. What MS Access/SQL Server function should you use to calculate the number of days between the current date and January 25, 1999?
13. What Oracle function should you use to calculate the number of days between the current date and January 25, 1999?
14. Suppose a PRODUCT table contains two attributes, PROD_CODE and VEND_CODE. Those two attributes have values of ABC, 125, DEF, 124, GHI, 124, and JKL, 123, respectively. The VENDOR table contains a single attribute, VEND_CODE, with values 123, 124, 125, and 126, respectively. (The VEND_CODE attribute in the PRODUCT table is a foreign key to the VEND_CODE in the VENDOR table.) Given that information, what would be the query output for:
    a. A UNION query based on the two tables?
    b. A UNION ALL query based on the two tables?
    c. An INTERSECT query based on the two tables?
    d. A MINUS query based on the two tables?
15. What string function should you use to list the first three characters of a company's EMP_LNAME values? Give an example using a table named EMPLOYEE. Provide examples for Oracle and SQL Server.
16. What is an Oracle sequence? Write its syntax.
17. What is a trigger, and what is its purpose? Give an example.
18. What is a stored procedure, and why is it particularly useful? Give an example.
19. What is embedded SQL, and how is it used?
20. What is dynamic SQL, and how does it differ from static SQL?

Problems

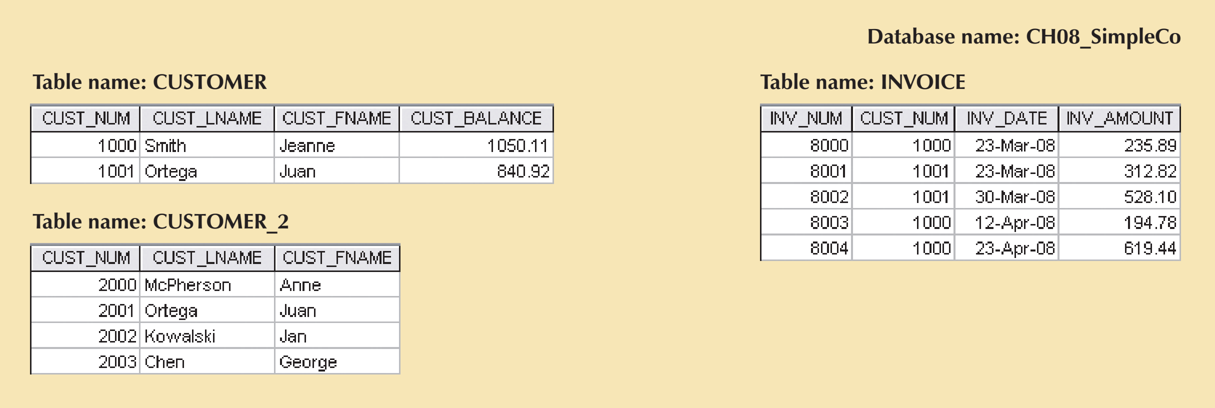
Use the database tables in Figure P8.1 as the basis for Problems 1–18.

Online Content

The **Ch08_SimpleCo** database is located in the Student Online Companion, as are the script files to duplicate this data set in Oracle.

1. Create the tables. (Use the MS Access example shown in Figure P8.1 to see what table names and attributes to use.)
2. Insert the data into the tables you created in Problem 1.
3. Write the query that will generate a combined list of customers (from the tables CUSTOMER and CUSTOMER_2) that do not include the duplicate customer records. (Note that only the customer named Juan Ortega shows up in both customer tables.)
4. Write the query that will generate a combined list of customers to include the duplicate customer records.
5. Write the query that will show only the duplicate customer records.
6. Write the query that will generate only the records that are unique to the CUSTOMER_2 table.
7. Write the query to show the invoice number, the customer number, the customer name, the invoice date, and the invoice amount for all customers with a customer balance of $1,000 or more.

FIGURE P8.1 Ch08_SimpleCo database tables



Database name: CH08_SimpleCo

Table name: CUSTOMER

CUST_NUM	CUST_LNAME	CUST_FNAME	CUST_BALANCE
1000	Smith	Jeanne	1050.11
1001	Ortega	Juan	840.92

Table name: CUSTOMER_2

CUST_NUM	CUST_LNAME	CUST_FNAME
2000	McPherson	Anne
2001	Ortega	Juan
2002	Kowalski	Jan
2003	Chen	George

Table name: INVOICE

INV_NUM	CUST_NUM	INV_DATE	INV_AMOUNT
8000	1000	23-Mar-08	235.89
8001	1001	23-Mar-08	312.82
8002	1001	30-Mar-08	528.10
8003	1000	12-Apr-08	194.78
8004	1000	23-Apr-08	619.44

8. Write the query that will show (for all the invoices) the invoice number, the invoice amount, the average invoice amount, and the difference between the average invoice amount and the actual invoice amount.
9. Write the query that will write Oracle sequences to produce automatic customer number and invoice number values. Start the customer numbers at 1000 and the invoice numbers at 5000.
10. Modify the CUSTOMER table to included two new attributes: CUST_DOB and CUST_AGE. Customer 1000 was born on March 15, 1979, and customer 1001 was born on December 22, 1988.
11. Assuming you completed Problem 10, write the query that will list the names and ages of your customers.
12. Assuming the CUSTOMER table contains a CUST_AGE attribute, write the query to update the values in that attribute. (*Hint*: Use the results of the previous query.)
13. Write the query that lists the average age of your customers. (Assume that the CUSTOMER table has been modified to include the CUST_DOB and the derived CUST_AGE attribute.)
14. Write the trigger to update the CUST_BALANCE in the CUSTOMER table when a new invoice record is entered. (Assume that the sale is a credit sale.) Test the trigger, using the following new INVOICE record:

    8005, 1001, '27-APR-08', 225.40

    Name the trigger **trg_updatecustbalance**.
15. Write a procedure to add a new customer to the CUSTOMER table. Use the following values in the new record:

    1002, 'Rauthor', 'Peter', 0.00

    Name the procedure **prc_cust_add**. Run a query to see if the record has been added.
16. Write a procedure to add a new invoice record to the INVOICE table. Use the following values in the new record:

    8006, 1000, '30-APR-08', 301.72

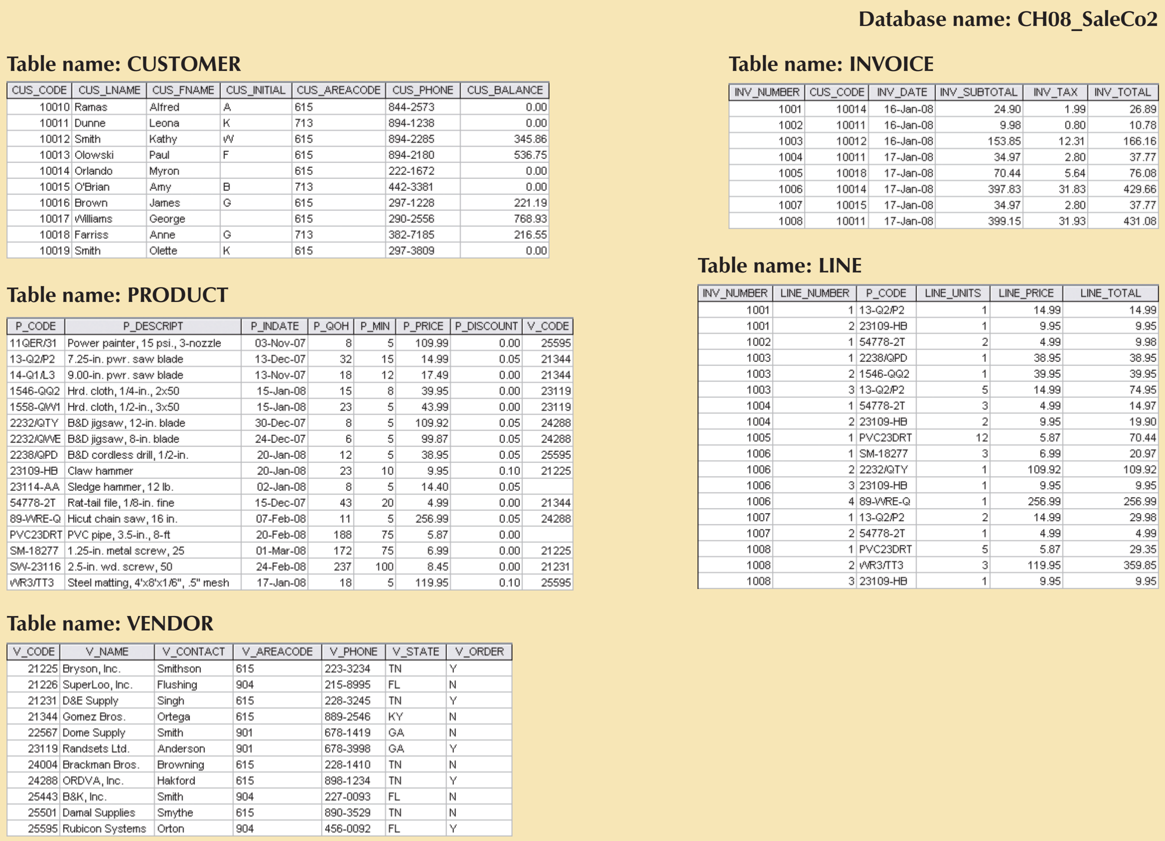
    Name the procedure **prc_invoice_add**. Run a query to see if the record has been added.
17. Write a trigger to update the customer balance when an invoice is deleted. Name the trigger **trg_updatecustbalance2**.
18. Write a procedure to delete an invoice, giving the invoice number as a parameter. Name the procedure **prc_inv_delete**. Test the procedure by deleting invoices 8005 and 8006.

NOTE

The following problem sets can serve as the basis for a class project or case.

Use the **Ch08_SaleCo2** database to work Problems 19–22, shown in Figure P8.19.

FIGURE P8.19 Ch08_SaleCo2 database tables



Database name: CH08_SaleCo2

Table name: CUSTOMER

CUS_CODE	CUS_LNAME	CUS_FNAME	CUS_INITIAL	CUS_AREACODE	CUS_PHONE	CUS_BALANCE
10010	Ramas	Alfred	A	615	844-2573	0.00
10011	Dunne	Leona	K	713	894-1238	0.00
10012	Smith	Kathy	W	615	894-2285	345.86
10013	Olowski	Paul	F	615	894-2180	536.75
10014	Orlando	Myron		615	222-1672	0.00
10015	O'Brian	Amy	B	713	442-3381	0.00
10016	Brown	James	G	615	297-1228	221.19
10017	Williams	George		615	290-2556	768.93
10018	Farriss	Anne	G	713	382-7185	216.55
10019	Smith	Olette	K	615	297-3809	0.00

Table name: INVOICE

INV_NUMBER	CUS_CODE	INV_DATE	INV_SUBTOTAL	INV_TAX	INV_TOTAL
1001	10014	16-Jan-08	24.90	1.99	26.89
1002	10011	16-Jan-08	9.98	0.80	10.78
1003	10012	16-Jan-08	153.85	12.31	166.16
1004	10011	17-Jan-08	34.97	2.80	37.77
1005	10018	17-Jan-08	70.44	5.64	76.08
1006	10014	17-Jan-08	397.83	31.83	429.66
1007	10015	17-Jan-08	34.97	2.80	37.77
1008	10011	17-Jan-08	399.15	31.93	431.08

Table name: LINE

INV_NUMBER	LINE_NUMBER	P_CODE	LINE_UNITS	LINE_PRICE	LINE_TOTAL
1001	1	13-Q2/P2	1	14.99	14.99
1001	2	23109-HB	1	9.95	9.95
1002	1	54778-2T	2	4.99	9.98
1003	1	2238/QPD	1	38.95	38.95
1003	2	1546-QQ2	1	39.95	39.95
1003	3	13-Q2/P2	5	14.99	74.95
1004	1	54778-2T	3	4.99	14.97
1004	2	23109-HB	2	9.95	19.90
1005	1	PVC23DRT	12	5.87	70.44
1006	1	SM-18277	3	6.99	20.97
1006	2	2232/QTY	1	109.92	109.92
1006	3	23109-HB	1	9.95	9.95
1006	4	89-WRE-Q	1	256.99	256.99
1007	1	13-Q2/P2	2	14.99	29.98
1007	2	54778-2T	1	4.99	4.99
1008	1	PVC23DRT	5	5.87	29.35
1008	2	WR3/TT3	3	119.95	359.85
1008	3	23109-HB	1	9.95	9.95

Table name: PRODUCT

P_CODE	P_DESCRIPT	P_INDATE	P_QOH	P_MIN	P_PRICE	P_DISCOUNT	V_CODE
11QER/31	Power painter, 15 psi., 3-nozzle	03-Nov-07	8	5	109.99	0.00	25595
13-Q2/P2	7.25-in. pwr. saw blade	13-Dec-07	32	15	14.99	0.05	21344
14-Q1/L3	9.00-in. pwr. saw blade	13-Nov-07	18	12	17.49	0.00	21344
1546-QQ2	Hrd. cloth, 1/4-in., 2x50	15-Jan-08	15	8	39.95	0.00	23119
1558-QW1	Hrd. cloth, 1/2-in., 3x50	15-Jan-08	23	5	43.99	0.00	23119
2232/QTY	B&D jigsaw, 12-in. blade	30-Dec-07	8	5	109.92	0.05	24288
2232/QWE	B&D jigsaw, 8-in. blade	24-Dec-07	6	5	99.87	0.05	24288
2238/QPD	B&D cordless drill, 1/2-in.	20-Jan-08	12	5	38.95	0.05	25595
23109-HB	Claw hammer	20-Jan-08	23	10	9.95	0.10	21225
23114-AA	Sledge hammer, 12 lb.	02-Jan-08	8	5	14.40	0.05	
54778-2T	Rat-tail file, 1/8-in. fine	15-Dec-07	43	20	4.99	0.00	21344
89-WRE-Q	Hicut chain saw, 16 in.	07-Feb-08	11	5	256.99	0.05	24288
PVC23DRT	PVC pipe, 3.5-in., 8-ft	20-Feb-08	188	75	5.87	0.00	
SM-18277	1.25-in. metal screw, 25	01-Mar-08	172	75	6.99	0.00	21225
SW-23116	2.5-in. wd. screw, 50	24-Feb-08	237	100	8.45	0.00	21231
WR3/TT3	Steel matting, 4'x8'x1/6", .5" mesh	17-Jan-08	18	5	119.95	0.10	25595

Table name: VENDOR

V_CODE	V_NAME	V_CONTACT	V_AREACODE	V_PHONE	V_STATE	V_ORDER
21225	Bryson, Inc.	Smithson	615	223-3234	TN	Y
21226	SuperLoo, Inc.	Flushing	904	215-8995	FL	N
21231	D&E Supply	Singh	615	228-3245	TN	Y
21344	Gomez Bros.	Ortega	615	889-2546	KY	N
22567	Dome Supply	Smith	901	678-1419	GA	N
23119	Randsets Ltd.	Anderson	901	678-3998	GA	Y
24004	Brackman Bros.	Browning	615	228-1410	TN	N
24288	ORDVA, Inc.	Hakford	615	898-1234	TN	Y
25443	B&K, Inc.	Smith	904	227-0093	FL	N
25501	Damal Supplies	Smythe	615	890-3529	TN	N
25595	Rubicon Systems	Orton	904	456-0092	FL	Y

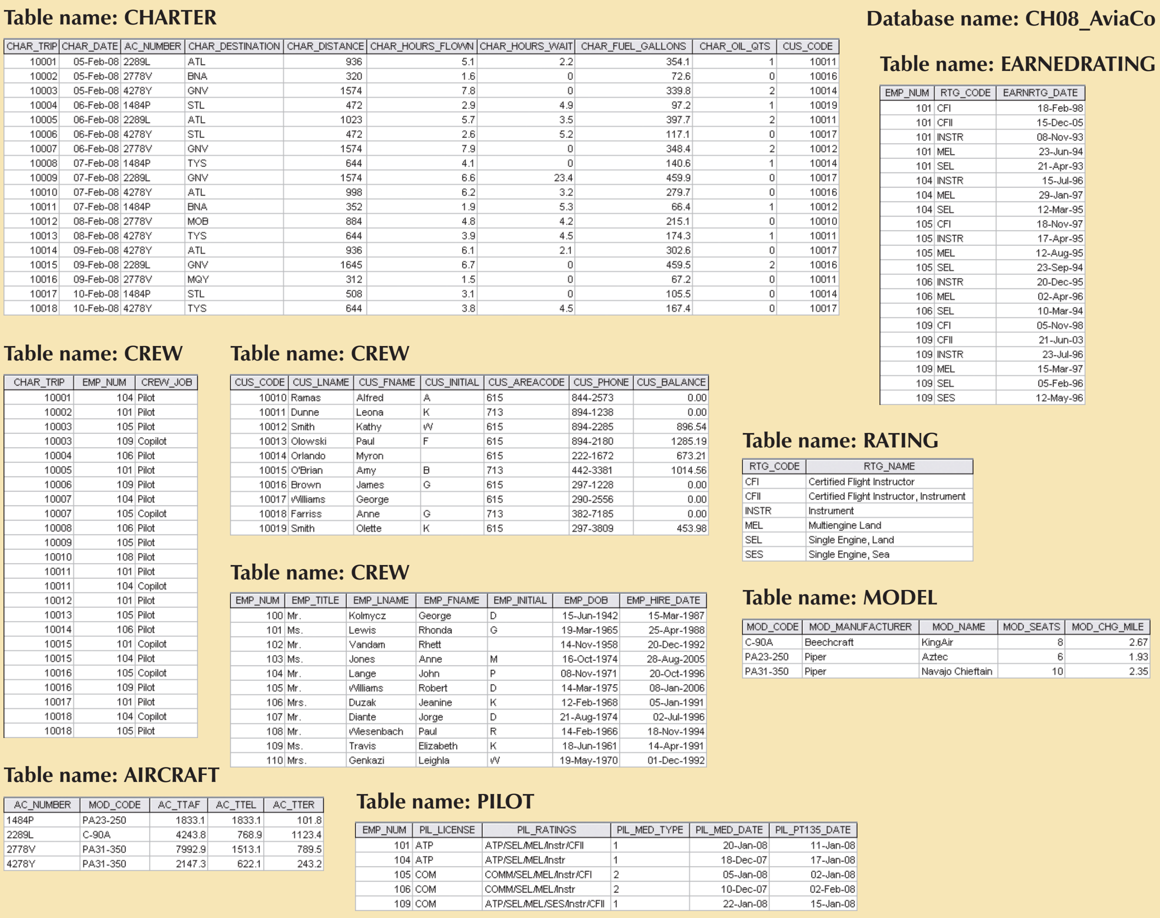
ONLINE CONTENT

The **Ch08_SaleCo2** database used in Problems 19–22 is located in the Student Online Companion for this book, as are the script files to duplicate this data set in Oracle.

19. Create a trigger named **trg_line_total** to write the LINE_TOTAL value in the LINE table every time you add a new LINE row. (The LINE_TOTAL value is the product of the LINE_UNITS and the LINE_PRICE values.)
20. Create a trigger named **trg_line_prod** that will automatically update the quantity on hand for each product sold after a new LINE row is added.
21. Create a stored procedure named **prc_inv_amounts** to update the INV_SUBTOTAL, INV_TAX, and INV_TOTAL. The procedure takes the invoice number as a parameter. The INV_SUBTOTAL is the sum of the LINE_TOTAL amounts for the invoice, the INV_TAX is the product of the INV_SUBTOTAL and the tax rate (8%), and the INV_TOTAL is the sum of the INV_SUBTOTAL and the INV_TAX.
22. Create a procedure named **prc_cus_balance_update** that will take the invoice number as a parameter and update the customer balance. (*Hint*: You can use the DECLARE section to define a TOTINV numeric variable that holds the computed invoice total.)

Use the **Ch08_AviaCo** database to work Problems 23–34, shown in Figure P8.23.

FIGURE P8.23 Ch08_AviaCo database tables



Database name: CH08_AviaCo

Table name: CHARTER

CHAR_TRIP	CHAR_DATE	AC_NUMBER	CHAR_DESTINATION	CHAR_DISTANCE	CHAR_HOURS_FLOWN	CHAR_HOURS_WAIT	CHAR_FUEL_GALLONS	CHAR_OIL_QTS	CUS_CODE
10001	05-Feb-08	2289L	ATL	936	5.1	2.2	354.1	1	10011
10002	05-Feb-08	2778V	BNA	320	1.6	0	72.6	0	10016
10003	05-Feb-08	4278Y	GNV	1574	7.8	0	339.8	2	10014
10004	06-Feb-08	1484P	STL	472	2.9	4.9	97.2	1	10019
10005	06-Feb-08	2289L	ATL	1023	5.7	3.5	397.7	2	10011
10006	06-Feb-08	4278Y	STL	472	2.6	5.2	117.1	0	10017
10007	06-Feb-08	2778V	GNV	1574	7.9	0	348.4	2	10012
10008	07-Feb-08	1484P	TYS	644	4.1	0	140.6	1	10014
10009	07-Feb-08	2289L	GNV	1574	6.6	23.4	459.9	0	10017
10010	07-Feb-08	4278Y	ATL	998	6.2	3.2	279.7	0	10016
10011	07-Feb-08	1484P	BNA	352	1.9	5.3	66.4	1	10012
10012	08-Feb-08	2778V	MOB	884	4.8	4.2	215.1	0	10010
10013	08-Feb-08	4278Y	TYS	644	3.9	4.5	174.3	1	10011
10014	09-Feb-08	4278Y	ATL	936	6.1	2.1	302.6	0	10017
10015	09-Feb-08	2289L	GNV	1645	6.7	0	459.5	2	10016
10016	09-Feb-08	2778V	MQY	312	1.5	0	67.2	0	10011
10017	10-Feb-08	1484P	STL	508	3.1	0	105.5	0	10014
10018	10-Feb-08	4278Y	TYS	644	3.8	4.5	167.4	0	10017

Table name: EARNEDRATING

EMP_NUM	RTG_CODE	EARNRTG_DATE
101	CFI	18-Feb-98
101	CFII	15-Dec-05
101	INSTR	08-Nov-93
101	MEL	23-Jun-94
101	SEL	21-Apr-93
104	INSTR	15-Jul-96
104	MEL	29-Jan-97
104	SEL	12-Mar-95
105	CFI	18-Nov-97
105	INSTR	17-Apr-95
105	MEL	12-Aug-95
105	SEL	23-Sep-94
106	INSTR	20-Dec-95
106	MEL	02-Apr-96
106	SEL	10-Mar-94
109	CFI	05-Nov-98
109	CFII	21-Jun-03
109	INSTR	23-Jul-96
109	MEL	15-Mar-97
109	SEL	05-Feb-96
109	SES	12-May-96

Table name: CREW

CHAR_TRIP	EMP_NUM	CREW_JOB
10001	104	Pilot
10002	101	Pilot
10003	105	Pilot
10003	109	Copilot
10004	106	Pilot
10005	101	Pilot
10006	109	Pilot
10007	104	Pilot
10007	105	Copilot
10008	106	Pilot
10009	105	Pilot
10010	108	Pilot
10011	101	Pilot
10011	104	Copilot
10012	101	Pilot
10013	105	Pilot
10014	106	Pilot
10015	101	Copilot
10015	104	Pilot
10016	105	Copilot
10016	109	Pilot
10017	101	Pilot
10018	104	Copilot
10018	105	Pilot

Table name: CREW

CUS_CODE	CUS_LNAME	CUS_FNAME	CUS_INITIAL	CUS_AREACODE	CUS_PHONE	CUS_BALANCE
10010	Ramas	Alfred	A	615	844-2573	0.00
10011	Dunne	Leona	K	713	894-1238	0.00
10012	Smith	Kathy	W	615	894-2285	896.54
10013	Olowski	Paul	F	615	894-2180	1285.19
10014	Orlando	Myron		615	222-1672	673.21
10015	O'Brian	Amy	B	713	442-3381	1014.56
10016	Brown	James	G	615	297-1228	0.00
10017	Williams	George		615	290-2556	0.00
10018	Farriss	Anne	G	713	382-7185	0.00
10019	Smith	Olette	K	615	297-3809	453.98

Table name: RATING

RTG_CODE	RTG_NAME
CFI	Certified Flight Instructor
CFII	Certified Flight Instructor, Instrument
INSTR	Instrument
MEL	Multiengine Land
SEL	Single Engine, Land
SES	Single Engine, Sea

Table name: CREW

EMP_NUM	EMP_TITLE	EMP_LNAME	EMP_FNAME	EMP_INITIAL	EMP_DOB	EMP_HIRE_DATE
100	Mr.	Kolmycz	George	D	15-Jun-1942	15-Mar-1987
101	Ms.	Lewis	Rhonda	G	19-Mar-1965	25-Apr-1988
102	Mr.	Vandam	Rhett		14-Nov-1958	20-Dec-1992
103	Ms.	Jones	Anne	M	16-Oct-1974	28-Aug-2005
104	Mr.	Lange	John	P	08-Nov-1971	20-Oct-1996
105	Mr.	Williams	Robert	D	14-Mar-1975	08-Jan-2006
106	Mrs.	Duzak	Jeanine	K	12-Feb-1968	05-Jan-1991
107	Mr.	Diante	Jorge	D	21-Aug-1974	02-Jul-1996
108	Mr.	Wiesenbach	Paul	R	14-Feb-1966	18-Nov-1994
109	Ms.	Travis	Elizabeth	K	18-Jun-1961	14-Apr-1991
110	Mrs.	Genkazi	Leighla	W	19-May-1970	01-Dec-1992

Table name: MODEL

MOD_CODE	MOD_MANUFACTURER	MOD_NAME	MOD_SEATS	MOD_CHG_MILE
C-90A	Beechcraft	KingAir	8	2.67
PA23-250	Piper	Aztec	6	1.93
PA31-350	Piper	Navajo Chieftain	10	2.35

Table name: AIRCRAFT

AC_NUMBER	MOD_CODE	AC_TTAF	AC_TTEL	AC_TTER
1484P	PA23-250	1833.1	1833.1	101.8
2289L	C-90A	4243.8	768.9	1123.4
2778V	PA31-350	7992.9	1513.1	789.5
4278Y	PA31-350	2147.3	622.1	243.2

Table name: PILOT

EMP_NUM	PIL_LICENSE	PIL_RATINGS	PIL_MED_TYPE	PIL_MED_DATE	PIL_PT135_DATE
101	ATP	ATP/SEL/MEL/Instr/CFII	1	20-Jan-08	11-Jan-08
104	ATP	ATP/SEL/MEL/Instr	1	18-Dec-07	17-Jan-08
105	COM	COMM/SEL/MEL/Instr/CFI	2	05-Jan-08	02-Jan-08
106	COM	COMM/SEL/MEL/Instr	2	10-Dec-07	02-Feb-08
109	COM	ATP/SEL/MEL/SES/Instr/CFII	1	22-Jan-08	15-Jan-08

ONLINE CONTENT

The **Ch08_AviaCo** database used for Problems 23–34 is located in the Student Online Companion for this book, as are the script files to duplicate this data set in Oracle.

23. Modify the MODEL table to add the attribute and insert the values shown in the following table.

ATTRIBUTE NAME	ATTRIBUTE DESCRIPTION	ATTRIBUTE TYPE	ATTRIBUTE VALUES
MOD_WAIT_CHG	Waiting charge per hour for each model	Numeric	$100 for C-90A $50 for PA23-250 $75 for PA31-350

24. Write the queries to update the MOD_WAIT_CHG attribute values based on Problem 23.
25. Modify the CHARTER table to add the attributes shown in the following table.

ATTRIBUTE NAME	ATTRIBUTE DESCRIPTION	ATTRIBUTE TYPE
CHAR_WAIT_CHG	Waiting charge for each model (copied from the MODEL table)	Numeric
CHAR_FLT_CHG_HR	Flight charge per mile for each model (copied from the MODEL table using the MOD_CHG_MILE attribute)	Numeric
CHAR_FLT_CHG	Flight charge (calculated by CHAR_HOURS_FLOWN x CHAR_FLT_CHG_HR)	Numeric
CHAR_TAX_CHG	CHAR_FLT_CHG x tax rate (8%)	Numeric
CHAR_TOT_CHG	CHAR_FLT_CHG + CHAR_TAX_CHG	Numeric
CHAR_PYMT	Amount paid by customer	Numeric
CHAR_BALANCE	Balance remaining after payment	Numeric

26. Write the sequence of commands required to update the CHAR_WAIT_CHG attribute values in the CHARTER table. (*Hint*: Use either an updatable view or a stored procedure.)
27. Write the sequence of commands required to update the CHAR_FLT_CHG_HR attribute values in the CHARTER table. (*Hint*: Use either an updatable view or a stored procedure.)
28. Write the command required to update the CHAR_FLT_CHG attribute values in the CHARTER table.
29. Write the command required to update the CHAR_TAX_CHG attribute values in the CHARTER table.
30. Write the command required to update the CHAR_TOT_CHG attribute values in the CHARTER table.
31. Modify the PILOT table to add the attribute shown in the following table.

ATTRIBUTE NAME	ATTRIBUTE DESCRIPTION	ATTRIBUTE TYPE
PIL_PIC_HRS	Pilot in command (PIC) hours; updated by adding the CHARTER table's CHAR_HOURS_FLOWN to the PIL_PIC_HRS when the CREW table shows the CREW_JOB to be pilot	Numeric

32. Create a trigger named **trg_char_hours** that will automatically update the AIRCRAFT table when a new CHARTER row is added. Use the CHARTER table's CHAR_HOURS_FLOWN to update the AIRCRAFT table's AC_TTAF, AC_TTEL, and AC_TTER values.
33. Create a trigger named **trg_pic_hours** that will automatically update the PILOT table when a new CREW row is added and the CREW table uses a 'pilot' CREW_JOB entry. Use the CHARTER table's CHAR_HOURS_FLOWN to update the PILOT table's PIL_PIC_HRS only when the CREW table uses a 'pilot' CREW_JOB entry.
34. Create a trigger named **trg_cust_balance** that will automatically update the CUSTOMER table's CUST_BALANCE when a new CHARTER row is added. Use the CHARTER table's CHAR_TOT_CHG as the update source. (Assume that all charter charges are charged to the customer balance.)