CHAPTER

3

USING SQL QUERIES TO INSERT, UPDATE, DELETE, AND VIEW DATA

◀ LESSON A ▶

After completing this chapter, you should be able to:

- Run a script to create database tables automatically
- Insert data into database tables
- Create database transactions and commit data to the database
- Create search conditions in SQL queries
- Update and delete database records and truncate tables
- Create and use sequences to generate surrogate key values automatically
- Grant and revoke database object privileges

Recall from Chapter 2 that when database developers create a database application, they must specify the underlying SQL commands that the application forwards to the DBMS for processing. Also recall that SQL commands can be data definition language (DDL) commands, which create new database objects, and data manipulation language (DML) commands, which allow users to insert, update, delete, and view database data. In Chapter 2, you became familiar with the Oracle10*g* SQL DDL commands to create new database objects. The next step in developing a database is to use DML commands to insert, modify, and delete data records.

Ultimately, the people who use databases for daily tasks such as creating new customer orders or enrolling students in courses perform DML operations using forms and reports that automate data entry, modification, and summarization. To create these forms and reports, database developers must translate user inputs into SQL commands that the forms or reports submit to the database. Therefore, database developers must be very proficient with SQL DML commands. DML commands that allow users to retrieve database data are called **queries**, because the data that these commands retrieve often answers a question. DML commands that insert, update, or delete database data are called **action queries**, because the commands perform an action that changes the data values in the database.

Because of the complexity of the chapter material, this chapter is divided into three lessons. In this lesson, you learn how to create action queries to insert, update, and delete database records. In Lesson B, you learn various methods of retrieving rows from database tables. In Lesson C, methods for linking data from different tables are presented. You also learn how to issue commands to commit or discard the changes. Before you begin, you learn how to run a script to create the tables in the case study databases automatically.

USING SCRIPTS TO CREATE DATABASE TABLES

Recall from Chapter 2 that a script is a text file that contains one or more SQL commands. You can run a script in SQL*Plus to execute the SQL commands in the script automatically. Usually, database developers use script commands to create, modify, or delete database tables, or to insert or update data records, because scripts provide a way to execute a series of SQL commands quickly and easily. You created scripts in Chapter 2 when you typed a series of SQL commands, one after another, in your Notepad text file, and then saved the file using the .sql file extension.

To run a script, you type `start` at the SQL prompt, a blank space, and then the full path and filename of the script file. For example, to run the script in the c:\OraData\Chapter03\MyScript.sql file, you type the command: **`START c:\OraData\Chapter3\MyScript.sql`**. The folder path and script filename and extension can be any legal Windows filename, but they *cannot* contain any blank spaces.

NOTE

When you run a script, you can omit the .sql file extension because this is the default extension for script files. Also, you can type the at symbol (@) instead of START to run a script. For example, an alternate way to run the MyScript.sql file is to type @c:\OraData\Chapter3\MyScript.

Before you can complete the chapter tutorials, you need to run a script named Ch3EmptyNorthwoods.sql. This script first deletes all the Northwoods University database tables that you may have created in Chapter 2, and then it executes the commands to re-create the tables. In this next set of steps, you examine the script file in Notepad, and then run it through Sql*Plus.

To examine and run the Ch3EmptyNorthwoods.sql script file:

1. Start Notepad, and open Ch3EmptyNorthwoods.sql from the Chapter03 folder on your Data Disk. The file contains a DROP TABLE command to drop each of the tables in the Northwoods University database. It also contains a CREATE TABLE command to re-create each table. These commands are similar to the ones you learned in Chapter 2.
2. Start SQL*Plus and log onto the database.
3. To run the Ch3EmptyNorthwoods.sql script, type **`START c:\OraData\Chapter03\Ch3EmptyNorthwoods.sql`** at the SQL prompt, and then press **Enter**. (If your Data Disk is on a different drive letter, or has a different folder path, type that drive letter or folder path instead.)

TIP

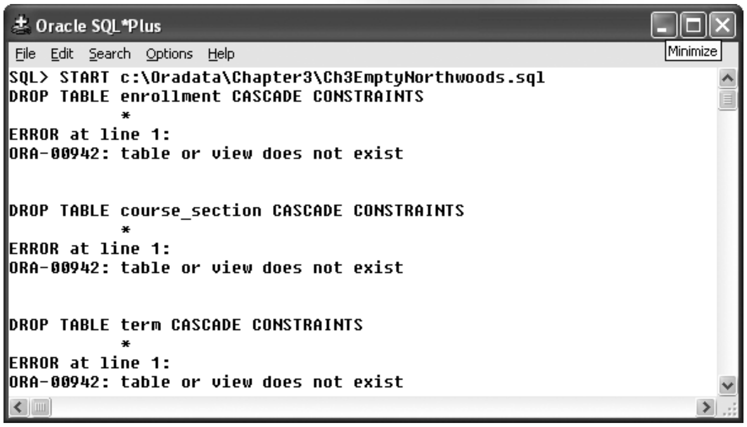
You could also type START c:\OraData\Chapter03\Ch3EmptyNorthwoods (omitting the .sql file extension), or type @c:\OraData\Chapter03\Ch3EmptyNorthwoods (using @ instead of START).

NOTE

This book assumes that your Data Disk files are stored on your workstation's C: drive in a folder named OraData. If you store your Data Disk files on a different drive or in a different folder, type the appropriate drive letter and path specification when you are instructed to open a file on your Data Disk.

HELP

Don't worry if you receive an error message in the DROP TABLE command as shown in Figure 3-1. This error message indicates that the script is trying to drop a table that does not exist. The script commands drop all existing Northwoods University database table definitions before creating the new tables; otherwise, the script would try to create a table that already exists, and SQL*Plus would generate an error and would not re-create the table.



```
SQL> START c:\Oradata\Chapter3\Ch3EmptyNorthwoods.sql
DROP TABLE enrollment CASCADE CONSTRAINTS
           *
ERROR at line 1:
ORA-00942: table or view does not exist


DROP TABLE course_section CASCADE CONSTRAINTS
           *
ERROR at line 1:
ORA-00942: table or view does not exist


DROP TABLE term CASCADE CONSTRAINTS
           *
ERROR at line 1:
ORA-00942: table or view does not exist
```

Figure 3-1 Error message that occurs while typing — script attempts to drop a nonexistent table

INSERTING DATA INTO TABLES

You use the SQL INSERT command to add new records to database tables. The following sections describe how to use the INSERT command, how to use format models to format input data, how to insert date and interval data values, and how to create transactions and commit data.

Using the INSERT Command

You can use the INSERT command either to insert a value for each column in the table or to insert values only into selected columns. The basic syntax of the INSERT statement for inserting a value for each table column is:

```
INSERT INTO tablename
VALUES (column1_value, column2_value, ...);
```

When you insert a value for every table column, the INSERT command's VALUES clause must contain a value for each column in the table. If a data value is unknown or undetermined, you insert the word NULL in place of the data value, and that column value remains undefined in that record. You must list column values in the same order that you defined the columns in the CREATE TABLE command. You can use the DESCRIBE command to determine the order of the columns in a table.

You use the following command to insert the first record in the Northwoods University LOCATION table in Figure 1-26 and specify that the CAPACITY column value is NULL:

```
INSERT INTO location
VALUES (1, 'CR', '101', NULL);
```

Recall that in the LOCATION table, the LOC_ID column has a NUMBER data type, and the BLDG_CODE and ROOM columns have the VARCHAR2 data type. When you insert data values into columns that have a character data type, you must enclose the values in single quotation marks, and the text within the single quotation marks is case sensitive. If you want to insert a character string that contains a single quotation mark, you type the single quotation mark two times. For example, you specify the address 454 St. John's Place as `'454 St. John''s Place'`. Note that the characters after the *n* in *John* are two single quotation marks (`''`), not a double quotation mark (").

You can also use the INSERT command to insert values only in specific table columns. The basic syntax of the INSERT statement for inserting values into selected table columns is:

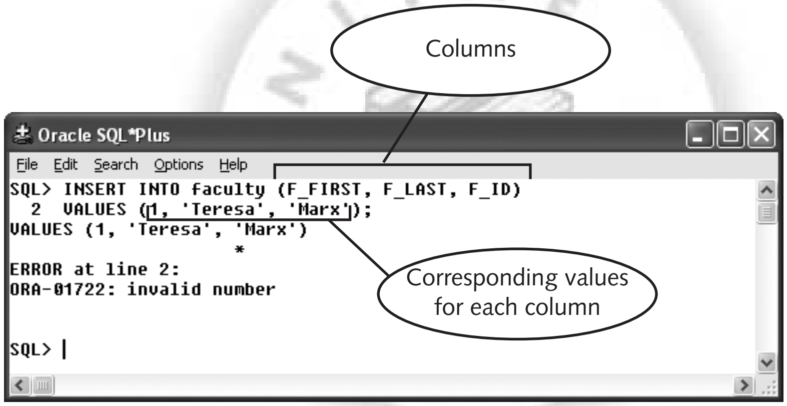
```
INSERT INTO tablename (columnname1, columnname2, ...)
VALUES (column1_value, column2_value, ...);
```

When using this syntax, you specify the names of the columns in which you want to insert data values in the INSERT INTO clause, then list the associated values in the VALUES clause. You can list the column names in any order in the INSERT INTO clause. However, you must list the data values in the VALUES clause in the same order as their associated columns appear in the INSERT INTO clause. If you omit one of the table's columns in the INSERT INTO clause (F_PHONE, F_RANK, etc.), the Oracle10*g* database automatically inserts NULL as the omitted value. You use the following command to insert only the F_ID, F_LAST, and F_FIRST values for the first row in the FACULTY table:

```
INSERT INTO faculty (F_FIRST, F_LAST, F_ID)
VALUES ('Teresa', 'Marx', 1);
```

Note that in this command, the column names appear in a different order than the order they appear in the FACULTY table. However, the command succeeds, because each column's corresponding data value is listed in the VALUES clause in the same order as the column names in the INSERT INTO clause.

You must be very careful to place the data values in the correct order in the VALUES clause. Consider the INSERT command in Figure 3-2.


Columns
Corresponding values for each column
Oracle SQL*Plus
File Edit Search Options Help

```
SQL> INSERT INTO faculty (F_FIRST, F_LAST, F_ID)
  2  VALUES (1, 'Teresa', 'Marx');
VALUES (1, 'Teresa', 'Marx')
                     *
ERROR at line 2:
ORA-01722: invalid number


SQL> |
```

Figure 3-2 INSERT command with data values in the wrong order

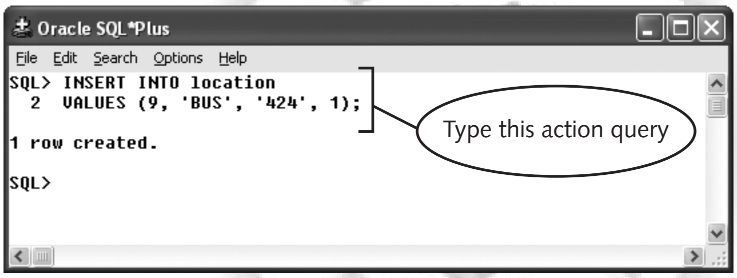
Note that the column names in the INSERT INTO clause are F_FIRST, F_LAST, and F_ID, which have the VARCHAR2, VARCHAR2, and NUMBER data types, respectively. The DBMS expects the data values in the VALUES clause to have these data types. Note that the values in the VALUES clause (1, 'Teresa', 'Marx') are not in the same order as the column names in the INSERT INTO clause: The values are listed as F_ID, F_FIRST, and then F_LAST. The error occurs because in the VALUES clause, the values have the NUMBER, VARCHAR2, and VARCHAR2 data types. The DBMS automatically converts the number 1 to a character, but the DBMS cannot convert the character string 'Marx' to a number.

Before you can insert a new data row, you must also ensure that all the foreign keys that the new row references have already been added to the database. For example, suppose you want to insert the first row in the Northwoods University STUDENT table. In the first STUDENT row, Tammy Jones' F_ID value is 1. This value refers to F_ID 1 (Teresa

Marx) in the FACULTY table. Therefore, the FACULTY row for F_ID 1 must already be in the database before you can add the first STUDENT row, or a foreign key reference error occurs. Now look at Teresa Marx's FACULTY record, and note that it has a foreign key value of LOC_ID 9. Similarly, this LOCATION row must already be in the database before you can add the row to the FACULTY table. The LOC_ID 9 record in the LOCATION table has no foreign key values to reference. Therefore, you can insert LOC_ID 9 in the LOCATION table, insert the associated FACULTY record, and then add the STUDENT record. In the following steps, you insert a row into the LOCATION table.

To insert a row into the LOCATION table:

1. Switch to Notepad, create a new file, and then type the SQL action query shown in Figure 3-3. Save the Notepad file as **Ch3AQueries.sql** in the Chapter03 folder on your Data Disk. Do not close the file.
2. Copy the action query text, paste it into SQL*Plus, and then press **Enter**. The message "1 row created." confirms that the row has been added to the LOCATION table.



```
SQL> INSERT INTO location
  2  VALUES (9, 'BUS', '424', 1);

1 row created.

SQL>
```

Figure 3-3 Action query to insert the first row into the LOCATION table

NOTE For the rest of the lesson, assume that you type all commands into your Ch3AQueries.sql Notepad text file, and then copy and paste them into SQL*Plus.

Note that the LOCATION table has four columns—LOC_ID, BLDG_CODE, ROOM, and CAPACITY—and that their associated data types are NUMBER, VARCHAR2, VARCHAR2, and NUMBER. You enter the NUMBER column values as digits, and the VARCHAR2 columns as text strings within single quotation marks ('). Remember that you must enter the column values in the INSERT clause in the same order as the columns in the table, and you must include a value for every column in the table. When you insert a row into the LOCATION table, the DBMS expects a NUMBER data type, then a VARCHAR2, another VARCHAR2, and then another NUMBER. An error occurs if you try to insert the values in the wrong order or if you omit a column value.

NOTE

If you cannot remember the order of the columns or their data types, remember that you can use the DESCRIBE command to verify the table's structure.

Now you insert the row for F_ID 1 (Teresa Marx) in the FACULTY table. To practice using the INSERT action query when you insert values for specific columns, you enter values only for the F_ID, F_LAST, F_FIRST, F_MI, and LOC_ID columns, and omit the values for F_PHONE, F_RANK, F_SUPER, F_PIN, and F_IMAGE. Recall that when you omit values in an INSERT action query, the DBMS automatically specifies that the omitted values are NULL.

To insert specific columns in a row:

1. Type the following action query to insert the first row into the FACULTY table:

```
INSERT INTO faculty (F_ID, F_LAST, F_FIRST, F_MI,↵
LOC_ID)
VALUES (1, 'Marx', 'Teresa', 'J', 9);
```

2. Execute the query. The message "1 row created." confirms that the DBMS has added the row to the table.

Format Models

Oracle stores data values in an internal binary format, and you can display data values using a variety of output formats. Suppose that a column with the DATE data type contains a value that represents 07/22/2006 9:29:00 PM. The default Oracle10*g* output format for DATE data columns is DD-MON-YY, so by default, this value appears as 22-JUL-06 when you retrieve the value from the database. (The database stores the time portion of the date, but does not display the time in the default output format.) Instead of using the default output format, you can use an alphanumeric character string called a **format model**, also called a **format mask**, to specify a different output format, such as July 22, 2004 9:29 PM, or 21:29:00 PM (which uses 24-hour clock notation). Similarly, you could use a format model to format a NUMBER value of 1257.33 to appear as $1,257.33 or as 1257. Format models only change the way the data appears, and do not affect how Oracle10*g* stores the data value in the database. Table 3-1 lists how you can use some common numerical data format models to display the numeric data value stored as 012345.67. Within a format model, the digit *9* is a placeholder that represents how number data values appear within the formatting characters.

Format Model	Description	Displayed Value
999999	Returns the value rounded to the number of placeholders, and suppresses leading zeroes	12346
099999	Returns the value rounded to the number of placeholders, and displays leading zeroes	012346
$99999	Returns the value rounded to the number of placeholders, and prefaces the value with a dollar sign; suppresses leading zeroes	$12346
99999MI	Prefaces negative values with – (minus sign)	–12346
99999PR	Displays negative values in angle brackets	<12346>
99,999	Displays a comma in the indicated position	12,346
99999.99	Displays the specified number of placeholders, with a decimal point in the indicated position	12345.67

Table 3-1 Common numerical format models

Table 3-2 shows common date format models using the example date of 5:45:35 PM, Sunday, February 15, 2006.

Format Model	Description	Displayed Value
YYYY	Displays all 4 digits of the year	2006
YYY or YY or Y	Displays last 3, 2, or 1 digit(s) of the year	006, 06, 6
RR	Displays dates from different centuries using two digits; year values from 0 to 49 are assumed to belong to the current century, and year values numbered from 50 to 99 are assumed to belong to the previous century	06
MM	Displays the month as digits (01-12)	02
MONTH	Displays the name of the month, spelled out, uppercase; for months with fewer than 9 characters in their names, the DBMS adds trailing blank spaces to pad the name to 9 characters	FEBRUARY
Month	Displays the name of the month, spelled out in mixed case; the DBMS adds trailing blank spaces to pad the name to 9 characters if necessary	February
DD	Displays the day of the month (01-31)	15
DDTH	Displays the day of the month as an ordinal number	15TH

Table 3-2 Common date format models

Format Model	Description	Displayed Value
DDD	Displays the day of the year (01-366)	46
DAY	Displays the day of the week, spelled out, uppercase	SUNDAY
Day	Displays the day of the week, spelled out, mixed case	Sunday
DY	Displays the name of the day as a 3-letter abbreviation	SUN
AM, PM, A.M., P.M.	Meridian indicator (without or with periods)	PM
HH	Displays the hour of the day using a 12-hour clock	05
HH24	Displays the hour of the day using a 24-hour clock	17
MI	Displays minutes (0-59)	45
SS	Displays seconds (0-59)	35

Table 3-2 Common date format models (continued)

You can specify to include front slashes (/), hyphens (-), and colons (:) as formatting characters between different date elements. For example, the format model MM/DD/YYYY appears as 02/15/2006, and the model HH:MI:SS appears as 05:45:35. You can also include additional formatting characters such as commas, periods, and blank spaces. For example, the format model DAY, MONTH DDTH, YYYY appears as SUNDAY, FEBRUARY 15TH, 2006.

Inserting Date and Interval Values

Recall that Oracle10*g* stores date values in columns that have the DATE data type, and time interval values in columns that have the INTERVAL data type. The following sections describe how to insert data values into columns that have the DATE and INTERVAL data types.

Inserting Values into DATE Columns

To insert a value into a DATE column, you specify the date value as a character string, then convert the date character string to an internal DATE format using the TO_DATE function. The general syntax of the TO_DATE function is:

```
TO_DATE('date_string', 'date_format_model')
```

In this syntax, *date_string* represents the date value as a text string, such as '08/24/2006', and *date_format_model* is the format model that represents the *date_string* value's format, such as `MM/DD/YYYY`. You convert the text string '08/24/2006' to a DATE data type using the following command:

```
TO_DATE('08/24/2006', 'MM/DD/YYYY')
```

Similarly, the following command converts the character string '24-AUG-2005' to a DATE data type:

```
TO_DATE('24-AUG-2005', 'DD-MON-YYYY')
```

Recall from Chapter 2 that the DATE data type also stores time values. Time is stored in the default format HH:MI:SS, where HH represents hours, MI represents minutes, and SS represents seconds. Note that in the COURSE_SECTION table, the C_SEC_TIME column stores values for the times when course section classes begin. To convert a 10:00 AM value to a DATE format for C_SEC_ID 1, you use the following command:

```
TO_DATE('10:00 AM', 'HH:MI AM')
```

Inserting Values into INTERVAL Columns

Recall from Chapter 2 that Oracle10*g* has two data types that store time intervals. The INTERVAL YEAR TO MONTH data type stores time intervals that consist of years and months, and the INTERVAL DAY TO SECOND data type stores time intervals that consist of days, hours, minutes, seconds, and fractional seconds. As with the DATE data type, you insert INTERVAL data values as character strings, then use a function to convert the character string to an internal INTERVAL data format.

To convert a character string that represents elapsed years and months to an INTERVAL YEAR TO MONTH data value, you use the TO_YMINTERVAL function. This function has the following general syntax:

```
TO_YMINTERVAL('years-months')
```

In this syntax, *years* is an integer that represents the interval years, and *months* is an integer that represents the interval months. Note that the command encloses the values in single quotation marks, and separates the *years* and *months* values with a hyphen (-). For example, you use the following command to convert an interval of four years and nine months to the INTERVAL YEAR TO MONTH data format:

```
TO_YMINTERVAL('4-9')
```

Similarly, you use the TO_DSINTERVAL function to convert a character string that represents elapsed days, minutes, hours, and seconds to an INTERVAL DAY TO SECOND data format. This function has the following general syntax:

```
TO_DSINTERVAL('days HH:MI:SS.99')
```

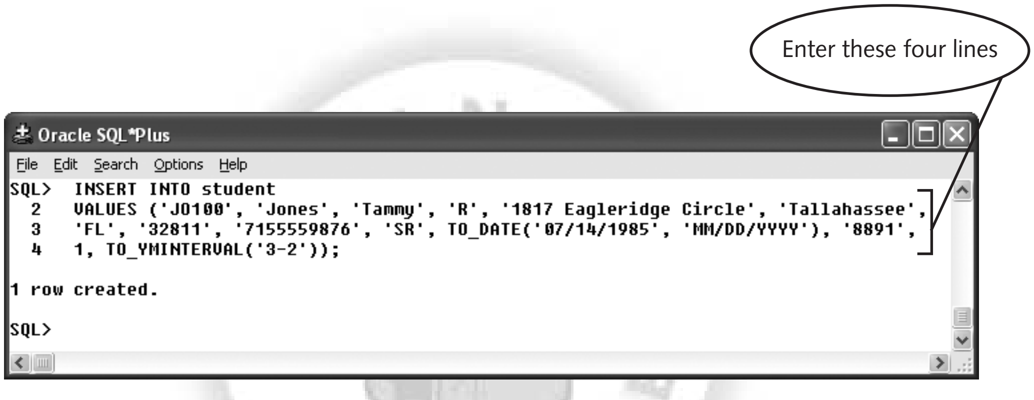
In this syntax, *days* is an integer that represents the interval days. *HH:MI:SS.99* represents the interval hours, minutes, seconds, and fractional seconds. As before, the command encloses the values in single quotation marks. The command separates the *days* and *HH:MI:SS.99* values with a blank space. The fractional seconds value is optional. You use the following command to convert an interval of 1 hour and 15 minutes to the INTERVAL DAYS TO SECONDS data format:

```
TO_DSINTERVAL('0 01:15:00')
```

Now you practice inserting DATE and INTERVAL data values into the database by adding the row for S_ID 1 (Tammy Jones) to the STUDENT table. The STUDENT table contains the S_DOB (student date of birth) column, which has the DATE data type. The STUDENT table also contains the TIME_ENROLLED column, which has the INTERVAL YEAR TO MONTH data type, and represents how long the student has been enrolled at Northwoods University.

To add a row to the STUDENT table

1. Type the action query shown in Figure 3-4 to insert the first row into the STUDENT table, using the functions to convert character strings to DATE and INTERVAL formats.



```
SQL> INSERT INTO student
  2  VALUES ('JO100', 'Jones', 'Tammy', 'R', '1817 Eagleridge Circle', 'Tallahassee',
  3  'FL', '32811', '7155559876', 'SR', TO_DATE('07/14/1985', 'MM/DD/YYYY'), '8891',
  4  1, TO_YMINTERVAL('3-2'));

1 row created.

SQL>
```

Figure 3-4 Action query to insert data values with DATE and INTERVAL data types

2. Execute the query. The message "1 row created." confirms that the DBMS has added the row to the table.

Inserting LOB Column Locators

Recall that the F_IMAGE column in the FACULTY table is a large object (LOB) data column of type BLOB, which stores the binary data for the image in the database. To make data retrievals for both LOB and non-LOB data more efficient, Oracle stores LOB data in a separate physical location from other types of data in a row. Whenever you insert data in an LOB column, you must initially insert a locator for the data. An **LOB locator** is a structure that contains information that identifies the LOB data type and points to the alternate memory location. After you insert the LOB locator, you then write a program or use a utility to insert the binary data into the database. (You will learn how to load BLOB data into the database in Chapter 10.)

To create a locator for a BLOB data column, you use the following syntax:

```
EMPTY_BLOB()
```

In the following steps you insert the row for faculty member Mark Zhulin that includes the BLOB locator. You insert data values for the F_ID, F_LAST, F_FIRST, and F_IMAGE columns only.

To insert the row for faculty member Mark Zhulin:

1. Type the following action query to insert the second row into the FACULTY table:

   ```
   INSERT INTO faculty (F_ID, F_LAST, F_FIRST, F_IMAGE)
   VALUES (2, 'Zhulin', 'Mark', EMPTY_BLOB());
   ```

2. Execute the query. The message "1 row created." confirms that the DBMS has added the row to the table.

CREATING TRANSACTIONS AND COMMITTING NEW DATA

When you create a new table or update the structure of an existing table, the DBMS changes the rows immediately and makes the result of the change visible to other users. This is not the case when you insert, update, or delete data rows. Recall that commands for operations that add, update, or delete data are called action queries. An Oracle10*g* database allows users to execute a series of action queries as a **transaction**, which represents a logical unit of work. In a transaction, all of the action queries must succeed, or none of the transactions can succeed.

An example of a transaction is when a Clearwater Traders customer purchases an item. The DBMS must insert the purchase information in the ORDERS and ORDER_LINE tables, and reduce the item's quantity on hand value in the INVENTORY table. Suppose that the DBMS successfully inserts the purchase information in the ORDERS and ORDER_LINE tables, but then experiences a power failure and does not successfully reduce the quantity on hand in the INVENTORY table. The customer purchases the item, but the INVENTORY table does not contain the updated value for the quantity on hand. All three action queries need to succeed, or none of them should succeed.

After the user enters all of the action queries in a transaction, he or she can either **commit** (save) all of the changes or **roll back** (discard) all of the changes. When the Oracle10*g* DBMS executes an action query, it updates the data in the database and also records information that enables the DBMS to undo the action query's changes. The DBMS saves this information to undo the changes until the user commits or rolls back the transaction that includes the action query.

The purpose of transaction processing is to enable every user to see a consistent view of the database. To achieve this consistency, a user cannot view or update data values that are part of another user's uncommitted transactions because these uncommitted transactions, which are called **pending transactions**, might be rolled back. The Oracle DBMS implements transaction processing by locking data rows associated with pending transactions. When the DBMS locks a row, other users cannot view or modify the row.

When the user commits the transaction, the DBMS releases the locks on the rows, and other users can view and update the rows again.

You do not explicitly create new transactions in Oracle10*g*. Rather, a new transaction begins when you start SQL*Plus and execute a command, and the transaction ends when you commit the current transaction. After you commit the current transaction, another transaction begins when you type a new query. You commit a transaction by executing the COMMIT command. After you execute the COMMIT command, a new transaction begins.

TIP

You can configure SQL*Plus to commit every query automatically after you execute the query. However, when you configure SQL*Plus this way, you cannot create transactions, so this is not recommended.

Now you commit your current transaction, which contains the two INSERT action queries that you have executed so far.

To commit your changes:

1. Type **`COMMIT;`** at the SQL prompt.
2. Press **Enter**. The message "Commit complete." indicates that your changes are permanent and visible to other users.

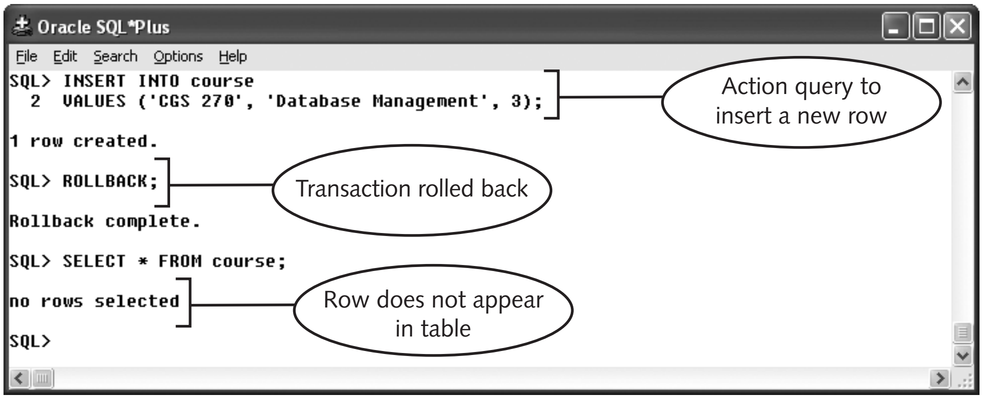
TIP

The Oracle10*g* DBMS automatically commits the current transaction when you exit SQL*Plus. However, it is a good idea to explicitly commit your changes often so that the rows are available to other users. This ensures your changes are saved if you do not exit normally because of a power failure or workstation malfunction.

The rollback process enables you to restore the database to its state the last time you issued the COMMIT command. To do this, the DBMS rolls back all of the changes made in subsequent action queries. Figure 3-5 shows an example of a rolled back transaction. The first INSERT action query inserts a course into the Northwoods University COURSE table. Then, the user issues the ROLLBACK command. The final SELECT query shows that the row was not inserted into the COURSE table, and that the DBMS rolled back the INSERT command.

NOTE

You will learn how to use the SELECT command to view data later in this chapter. For now, to view the current data in all of the columns in a table, you type SELECT * FROM *tablename*. The asterisk (*) specifies that the values of all table columns are displayed. However, the asterisk cannot be used with tables, such as the FACULTY table that contain the BLOB datatypes.

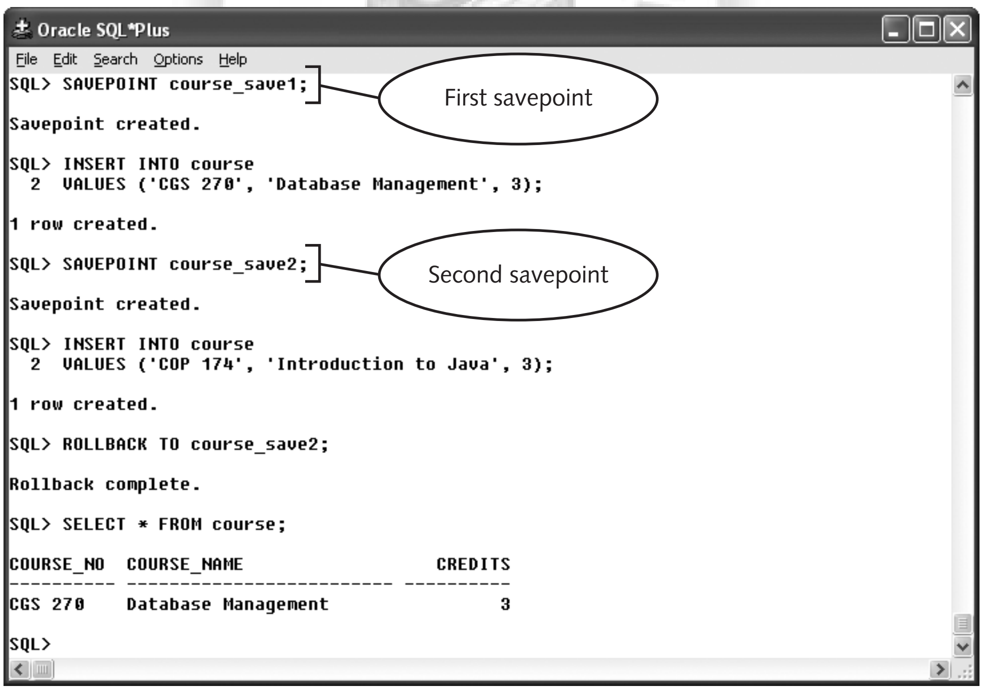


```
SQL> INSERT INTO course
  2  VALUES ('CGS 270', 'Database Management', 3);

1 row created.

SQL> ROLLBACK;

Rollback complete.

SQL> SELECT * FROM course;

no rows selected

SQL>
```

Oracle SQL*Plus
File Edit Search Options Help
Action query to insert a new row
Transaction rolled back
Row does not appear in table

Figure 3-5 Rolling back a transaction

You can use rollbacks with savepoints. A **savepoint** is a bookmark that designates the beginning of an individual section of a transaction. You can roll back part of a transaction by rolling back the transaction only to the savepoint. Figure 3-6 shows how to create savepoints and then roll back a SQL*Plus session to an intermediate savepoint.



```
SQL> SAVEPOINT course_save1;

Savepoint created.

SQL> INSERT INTO course
  2  VALUES ('CGS 270', 'Database Management', 3);

1 row created.

SQL> SAVEPOINT course_save2;

Savepoint created.

SQL> INSERT INTO course
  2  VALUES ('COP 174', 'Introduction to Java', 3);

1 row created.

SQL> ROLLBACK TO course_save2;

Rollback complete.

SQL> SELECT * FROM course;

COURSE_NO  COURSE_NAME                  CREDITS
---------- ------------------------- ----------
CGS 270    Database Management                3

SQL>
```

Figure 3-6 Using the ROLLBACK command with savepoints

In the first command in Figure 3-6, the user creates a savepoint named COURSE_SAVE1. Next, the user inserts course information for course CGS 270 into the COURSE table. Next, the user creates a second savepoint named COURSE_SAVE2, and then inserts a second course, COP 174, into the COURSE table. Finally, the user issues the command `ROLLBACK TO course_save2;`, which rolls back the transaction to the COURSE_SAVE2 savepoint. This causes the DBMS to restore the database to it's state when the user created the savepoint: the row in the COURSE table for CGS 270 is still inserted, but the row for the second course is discarded. If the user issues the command `ROLLBACK TO course_save1;`, the DBMS rolls back the command for inserting course COP 174.

CREATING SEARCH CONDITIONS IN SQL QUERIES

To write queries to update, delete, or retrieve database rows, you often need to specify which rows the query updates, deletes, or retrieves. For example, to delete the first row in the STUDENT table, you must specify to delete the row in which the S_ID value is JO100. To identify specific rows in a query, you use a **search condition**, which is an expression that seeks to match specific table rows. You use search conditions to compare column values to numerical, date, and character values.

The general syntax of a SQL search condition is:

```
WHERE columnname comparison_operator search_expression
```

In this syntax, *columnname* specifies the database column whose value you seek to match. The *comparison_operator* compares the *columnname* value with *search_expression*. *Search_expression* is usually a constant, such as the number 1 or the character string 'SR'. Table 3-3 lists common SQL comparison operators.

Operator	Description	Example
=	Equal to	`S_CLASS = 'SR'`
>	Greater than	`CAPACITY > 50`
<	Less than	`CAPACITY < 100`
>=	Greater than or equal to	`S_DOB >= TO_DATE ('01-JAN-1980', 'DD-MON-YYYY')`
<=	Less than or equal to	`MAX_ENRL <= 30`

<>
 !=
 ^=
	Not equal to	`STATUS <> 'CLOSED'`
		 `STATUS != 'CLOSED'`
		 `STATUS ^= 'CLOSED'`

Table 3-3 Comparison operators commonly used with search conditions

Operator	Description	Example
`LIKE`	Uses pattern matching in text strings; is usually used with the wildcard character (%), which indicates that part of the string can contain any characters; search string within single quotation marks is case sensitive	`term_desc LIKE 'Summer%'`
`IN`	Determines if a value is a member of a specific search set	`s_class IN ('FR','SO')`
`NOT IN`	Determines if a value is not a member of a specific search set	`s_class NOT IN ('FR','SO')`
`IS NULL`	Determines if a value is NULL	`s_mi IS NULL`
`IS NOT NULL`	Determines if a value is not NULL	`s_mi IS NOT NULL`

Table 3-3 Comparison operators commonly used with search conditions (continued)

The equal to (=), greater than (>), less than (<), and not equal to (<>, !=, or ^=) comparison operators are similar to the comparison operators you use in other programming languages. (You will learn how to use the `LIKE`, `IN`, `NOT IN`, `IS NULL`, and `IS NOT NULL` comparison operators in Lesson B in this chapter.) Now you learn how to create search expressions, and how to use the AND, OR, and NOT logical operators to create complex search conditions.

Defining Search Expressions

Recall that in a search condition, *search_expression* is the value that the search column matches. When the search expression is a number, you simply insert the number value. An example of a search condition in which *search_expression* is a number value is `WHERE f_id = 1`. This search condition retrieves the row for F_ID 1.

You must enclose text strings in single quotation marks. For example, you use the following search condition to match the S_CLASS value in the STUDENT table:

```
WHERE s_class = 'SR'
```

Search_expression values within single quotation marks are case sensitive. The search condition `S_CLASS = 'sr'` does not retrieve rows in which the S_CLASS value is 'SR'.

When *search_expression* involves a DATE data value, you must use the TO_DATE function to convert the DATE character string representation to an internal DATE data format. The following search condition matches dates of January 1, 1980:

```
    WHERE s_dob = TO_DATE('01/01/1980', 'MM/DD/YYYY')
```

NOTE If you choose to search for a date value without using the TO_DATE function, the search condition must be in the same format as Oracle10*g*'s internal storage format (that is, DD-MON-YY).

To search for dates before a given date, you use the less than (<) comparison operator. To search for dates after a given date, you use the greater than (>) operator. To search for dates that match a given date, you use the equal to (=) operator.

To search for data values that match an INTERVAL column, you must use the TO_YMINTERVAL and TO_DSINTERVAL functions to convert the character string representation of the interval to the INTERVAL data type. For example, you use the following search condition to match all rows in the COURSE_SECTION table for which the C_SEC_DURATION value is 1 hour and 15 minutes:

```
WHERE c_sec_duration = TO_DSINTERVAL('0 1:15:00')
```

You can also create search conditions that specify time intervals that are greater than or less than a given interval. For example, the following search condition specifies all Northwoods University students who have been enrolled for less than one year:

```
WHERE time_enrolled < TO_YMINTERVAL('1-0')
```

Creating Complex Search Conditions

A **complex search condition** combines multiple search conditions using the AND, OR, and NOT logical operators. When you use the **AND logical operator** to combine two search conditions, both conditions must be true for the complex search condition to be true. If no rows exist that match *both* conditions, then the complex search condition is false, and no matching rows are found. For example, the following complex search condition matches all rows in which BLDG_CODE is 'CR' and the capacity is greater than 50:

```
WHERE bldg_code = 'CR' AND capacity > 50
```

When you use the **OR logical operator** to create a complex search condition, only one of the conditions must be true for the complex search condition to be true. For example, the following search condition matches all course section rows that meet either on Tuesday and Thursday or on Monday, Wednesday, and Friday (at Northwoods University, R denotes courses that meet on Thursday):

```
WHERE day = 'TR' OR day = 'MWF'
```

You can use the **NOT logical operator** to match the logical opposite of a search expression, using this syntax:

```
WHERE NOT(search_expression)
```

For example, the following search condition finds all rows in the STUDENT table in which the S_CLASS column has any value other than 'FR':

```
WHERE NOT (s_class = 'FR')
```

UPDATING AND DELETING EXISTING TABLE ROWS

At Northwoods University, student addresses and telephone numbers often change, and every year students (it is hoped) move up to the next class. Existing data rows that are no longer relevant must be removed or saved in different locations. The following sections describe how to update and delete table rows.

Updating Table Rows

To update column values in one or more rows in a table, you use an **UPDATE action query**. An UPDATE action query specifies the name of the table to update, and lists the name of the column (or columns) to update, along with the new data value (or values). An UPDATE action query also usually contains a search condition to identify the row or rows to update. The general syntax of an UPDATE action query is:

```
UPDATE tablename
SET column1 = new_value1, column2 = new_value2, ...
WHERE search condition;
```

In this syntax, *tablename* specifies the table whose rows you wish to update. The SET clause lists the columns to update and their associated new values.

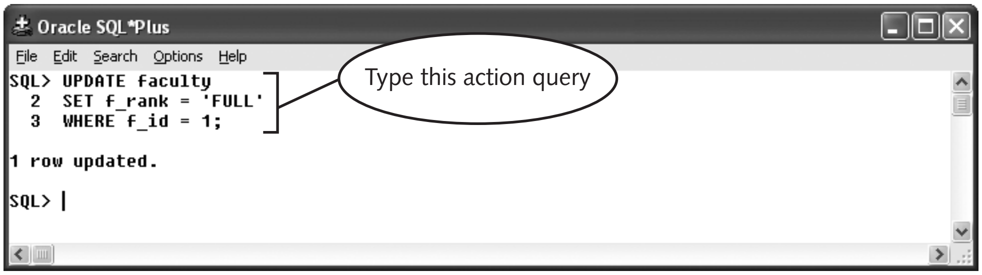
NOTE

You can update multiple columns in the same table using a single UPDATE action query, but you can update rows in only one table at a time in a single UPDATE action query.

The WHERE clause specifies the search condition. If you omit the search condition, the query updates every table row. In this next set of steps, you create an UPDATE action query to update faculty member Teresa Marx's F_RANK value to 'FULL'. (When you originally inserted the row, you omitted her F_RANK column value, so the DBMS inserted the value as NULL.) You will use her F_ID value (1) as the *search_expression* in the search condition.

To create an UPDATE action query:

1. Type the action query shown in Figure 3-7 to update faculty member Teresa Marx's F_RANK value.
2. Execute the query. The "1 row updated." message confirms that the row was updated.



```
SQL> UPDATE faculty
  2  SET f_rank = 'FULL'
  3  WHERE f_id = 1;

1 row updated.

SQL>
```

Figure 3-7 Creating an UPDATE action query

Recall that you can use a single UPDATE action query to update multiple column values in the same table. For example, the following action query changes Teresa Marx's F_RANK value to 'ASSOCIATE' and her F_PIN value to 1181:

```
UPDATE faculty
SET f_rank = 'ASSOCIATE', f_pin = 1181
WHERE f_id = 1;
```

TIP

Because the F_RANK column has the VARCHAR2 data type, the action query encloses the value in single quotation marks.

You can update multiple rows in a table with a single UPDATE command by specifying a search condition that matches multiple rows using the greater than (>) or less than (<) operators. Be careful to always include a search condition in an UPDATE action query: If you omit the search condition in the UPDATE command, then the command updates all table rows. For example, the following command updates the value of S_CLASS to 'SR' for all rows in the STUDENT table:

```
UPDATE student
SET s_class = 'SR';
```

Deleting Table Rows

You use the SQL DELETE action query to remove specific rows from a database table, and you truncate the table to remove all of the table rows. The following sections describe these two approaches for removing table rows.

The SQL DELETE Action Query

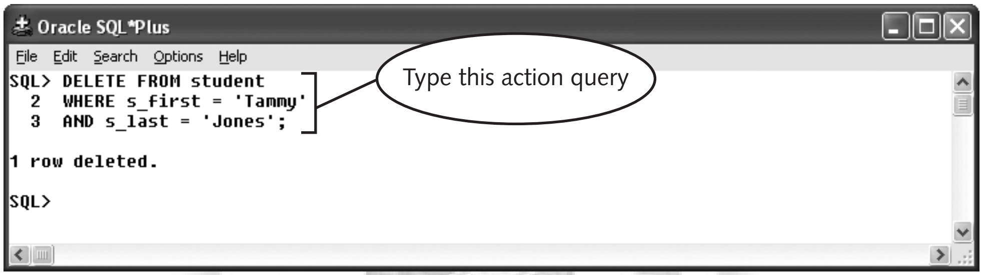
The general syntax for a DELETE action query is:

```
DELETE FROM tablename
WHERE search condition;
```

In this syntax, *tablename* specifies the table from which to delete the row(s). A single DELETE action query can delete rows from only one table. The search condition specifies the row or rows to delete. Be careful to always include a search condition in a DELETE action query—if you accidentally omit the search condition, the DELETE action query deletes all of the table rows! In the following set of steps, you delete Tammy Jones from the STUDENT table. You specify 'Tammy' and 'Jones' in the action query's search condition.

To delete a row:

1. Type the DELETE action query shown in Figure 3-8 to delete the Tammy Jones row from the STUDENT table.



```
SQL> DELETE FROM student
  2  WHERE s_first = 'Tammy'
  3  AND s_last = 'Jones';

1 row deleted.

SQL>
```

Figure 3-8 Creating a DELETE action query

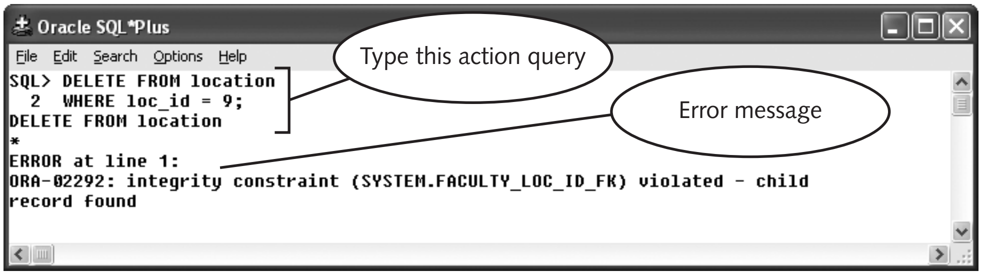
2. Execute the query. The message "1 row deleted." confirms that the DBMS has deleted the row.

You can use a single DELETE action query to delete multiple rows from a table if the search condition matches several rows. If you omit the search condition, a DELETE action query deletes every table row.

When a row's value is a foreign key that references another row, it is called a **child row**. For example, recall that the row for F_ID 1 (Teresa Marx) references LOC_ID 9, which is Teresa's office location ID. The row for Teresa Marx is a child row of the row for LOC_ID 9. You cannot delete a row if it has a child row; therefore, you cannot delete the row for LOC_ID 9 (Teresa Marx's office), unless you first delete the row in which the foreign key value exists (F_ID 1, faculty member Teresa Marx). In the next set of steps, you attempt to delete a row that has a child row, and view the error message.

To try to delete a row that has a child row:

1. Type the action query shown in Figure 3-9 to attempt to delete LOC_ID 9 (Teresa Marx's office) from the LOCATION table.
2. Execute the query. The error message indicates that the row contains a value that is referenced as a foreign key. To delete the row for LOC_ID 9 successfully, you must first delete the row for F_ID 1, which is the row that references it as a foreign key.



```
SQL> DELETE FROM location
  2  WHERE loc_id = 9;
DELETE FROM location
*
ERROR at line 1:
ORA-02292: integrity constraint (SYSTEM.FACULTY_LOC_ID_FK) violated - child
record found
```

Figure 3-9 Error message that appears when you attempt to delete a row that has a child row

Truncating Tables

Recall that after a DML statement executes, you must commit the transaction before the DBMS makes the changes visible to other users. The DELETE action query is a DML statement. Whenever you delete a row using a DELETE action query, an Oracle10*g* DBMS process records rollback information that the DBMS uses to restore the row if you roll back the transaction. For a DELETE action query, rollback information includes all of the column values for the deleted data row. If you use a DELETE action query to delete many rows, such as all of the rows in a table, the DBMS must make a copy of each deleted value. For a table with many columns and rows, this process can take a long time.

Sometimes you need to delete all of the rows in a table, and you know that you will not roll back the transaction. This may occur because you may have moved the data to a different location, or the data is outdated or incorrect. When you need to delete all of the rows in a table quickly, you can **truncate** the table, which means you remove all of the table data without saving any rollback information. When you truncate a table, the table structure and constraints remain intact. To truncate a table, you use the TRUNCATE TABLE command, which has the following general syntax:

```
TRUNCATE TABLE tablename;
```

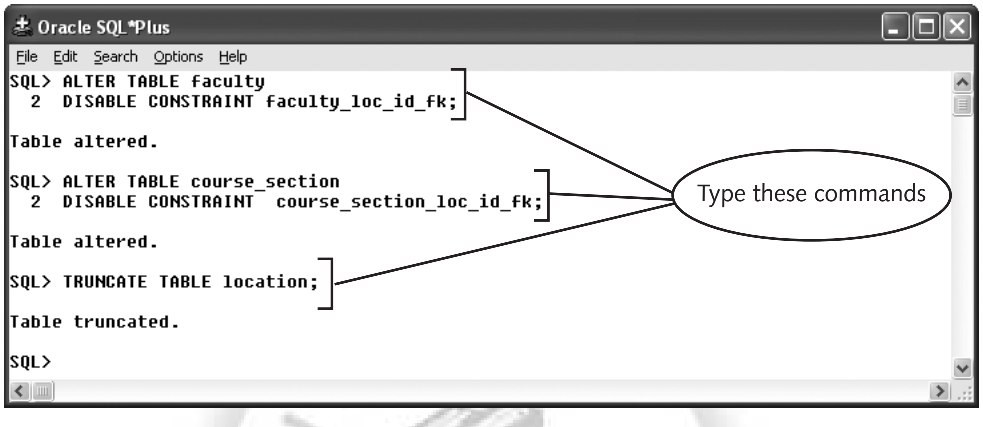
You cannot truncate a table that has foreign key constraints as long as the foreign key constraints are enabled. (Recall from Chapter 2 that when a constraint is enabled, the DBMS enforces the constraint.) Therefore, you must disable a table's foreign key constraints before you can truncate the table. Recall that you use the following command to disable an existing constraint:

```
ALTER TABLE tablename
DISABLE CONSTRAINT constraint_name;
```

In the next set of steps, you truncate the LOCATION table to delete all of its rows. Recall that the LOC_ID column in the LOCATION table is a foreign key in both the FACULTY table and the COURSE_SECTION table, so you must first disable the LOC_ID foreign key constraints in the FACULTY and COURSE_SECTION tables. Then you truncate the LOCATION table.

To truncate the LOCATION table:

1. Type **`ALTER TABLE faculty DISABLE CONSTRAINT faculty_loc_id_fk;`** at the SQL> prompt, as shown in Figure 3-10, to disable the foreign key constraint for the FACULTY table.



```
SQL> ALTER TABLE faculty
  2  DISABLE CONSTRAINT faculty_loc_id_fk;

Table altered.

SQL> ALTER TABLE course_section
  2  DISABLE CONSTRAINT  course_section_loc_id_fk;

Table altered.

SQL> TRUNCATE TABLE location;

Table truncated.

SQL>
```

Figure 3-10 Commands that disable constraints and then truncate a table

2. Type **`ALTER TABLE course_section DISABLE CONSTRAINT course_section_loc_id_fk;`** to display the foreign key constraint for the COURSE_SECTION table.
3. Type **`TRUNCATE TABLE location;`** to remove the rows contained in the LOCATION table.

The "Table truncated." message confirms that the DBMS successfully truncated the table.

SEQUENCES

Recall from Chapter 1 that sequences are sequential lists of numbers that the Oracle10*g* DBMS generates to create unique surrogate key values. For example, you might create a sequence to assign values for the faculty ID (F_ID) column in the Northwoods University FACULTY table. Columns that use sequence values must have the NUMBER data type. The following sections describe how to use SQL commands to create and manipulate sequences.

Creating New Sequences

A sequence is a database object, so you use the DDL (data description language) CREATE command to create a new sequence. (Recall that you used the DDL CREATE command in Chapter 2 to create new database tables.) Because the CREATE SEQUENCE command is a DDL command, the DBMS immediately creates the sequence when you execute the command, and you do not need to type `COMMIT` to save the sequence.

Figure 3-11 shows the general syntax for the command to create a new sequence. In Figure 3-11, square brackets enclose optional parameters, and a bar (|) separates parameters that include one of two values.

```
CREATE SEQUENCE sequence_name
   [INCREMENT BY number]
   [START WITH start_value]
   [MAXVALUE maximum_value] | [NOMAXVALUE]
   [MINVALUE minimum_value] | [NOMINVALUE]
   [CYCLE] | [NOCYCLE]
   [CACHE number_of_values] | [NOCACHE]
   [ORDER] | [NOORDER];
```

Figure 3-11 General syntax used to create a new sequence

In this syntax, *sequence_name* is the name of the sequence. Every sequence in your user schema must have a unique name, and the name must follow the Oracle naming standard rules. The CREATE SEQUENCE command can include the following optional parameters:

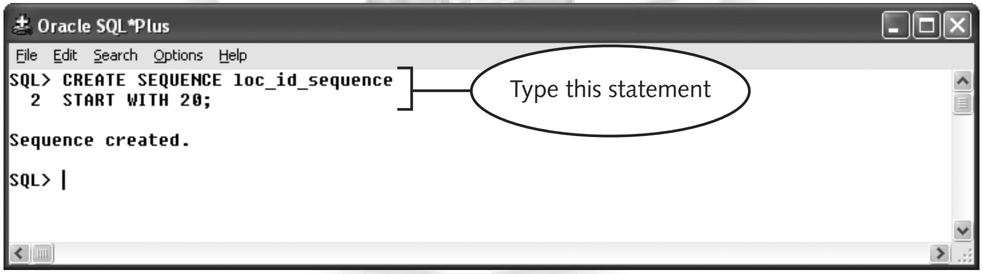
- INCREMENT BY specifies the value by which the sequence is incremented, and *number* can be any integer. By default, the DBMS increments a sequence by one.
- START WITH specifies the first sequence value. The *start_value* parameter can be any positive or negative integer value. When you omit the START WITH clause, the sequence starts with the value 1.
- MAXVALUE specifies the maximum value to which you can increment the sequence. The default maximum value is NOMAXVALUE, which allows the DBMS to increment the sequence to a maximum value of 10^{27}.
- MINVALUE specifies the minimum value for a sequence that has a negative increment value. The default minimum value is NOMINVALUE, which has a minimum value of 10^{-26}.
- CYCLE specifies that when the sequence reaches its MAXVALUE, it cycles back and starts again at its MINVALUE. For example, if you create a sequence with a maximum value of 10 and a minimum value of 5, the sequence increments up to 10 and then starts again at 5. If you omit the CYCLE parameter, or replace it with NOCYCLE, the sequence continues to generate values until it reaches its MAXVALUE, then it quits generating values.

- CACHE specifies that whenever you request a sequence value the DBMS automatically generates the specified *number_of_values* and stores them in a server memory area called a cache. This improves database performance when many users are simultaneously requesting sequence values. By default, the DBMS stores 20 sequence numbers in the cache. The NOCACHE parameter directs the DBMS not to cache any sequence values.
- ORDER ensures that the DBMS grants sequence numbers to users in the exact chronological order in which the users request the values. For example, the first user who requests a number from the sequence is granted sequence number 1, the second user who requests a value is granted sequence number 2, and so forth. This is useful for tracking the order in which users request sequence values. By default, this value is NOORDER, which specifies that the DBMS does not necessarily grant the sequence values in the same order that users request the values.

Next, you create a sequence named LOC_ID_SEQUENCE to generate values for the LOC_ID column in the LOCATION table. You specify that the sequence start with 20.

To create LOC_ID_SEQUENCE:

1. To create the sequence, type the command shown in Figure 3-12.



```
SQL> CREATE SEQUENCE loc_id_sequence
  2  START WITH 20;

Sequence created.

SQL> |
```

Figure 3-12 Creating a sequence

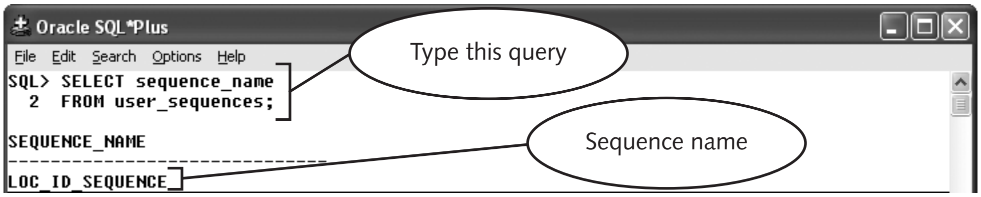
2. Execute the command. The "Sequence created." message confirms that the DBMS has created LOC_ID_SEQUENCE.

Viewing Sequence Information

A large database application may use many sequences, so sometimes you might forget the names of the sequences that you create. To review the names of your sequences, you can query the USER_SEQUENCES data dictionary view. (Recall that the data dictionary views contain information about the database structure.) The USER_SEQUENCES view has a column named SEQUENCE_NAME, which displays the names of all of the sequences in your user schema. In the next set of steps, you retrieve the names of your sequences.

To retrieve the names of your sequences:

1. Type the query shown in Figure 3-13.



```
SQL> SELECT sequence_name
  2  FROM user_sequences;

SEQUENCE_NAME
------------------------------
LOC_ID_SEQUENCE
```

Figure 3-13 Viewing sequence names

2. Press **Enter**. The output shows the names of your sequences. (If you have created other sequences, these sequences also appear in the output.)

TIP

The USER_SEQUENCES data dictionary view contains other columns that provide additional information about sequences, such as the MINVALUE and MAXVALUE specifications, and the last value the sequence generated.

Using Sequences

To use sequences, you must first understand psuedocolumns. An Oracle10*g* **pseudocolumn** acts like a column in a database table, but is actually a command that returns a specific value. You can use sequence pseudocolumns within SQL commands to retrieve the current or next value of a sequence. Table 3-4 summarizes the sequence pseudocolumns.

Pseudocolumn Name	Output
CURRVAL	Most recent sequence value retrieved during the current user session
NEXTVAL	Next available sequence value

Table 3-4 Oracle sequence pseudocolumns

To retrieve the next value in a sequence, you reference the NEXTVAL pseudocolumn using the following general syntax:

```
sequence_name.NEXTVAL
```

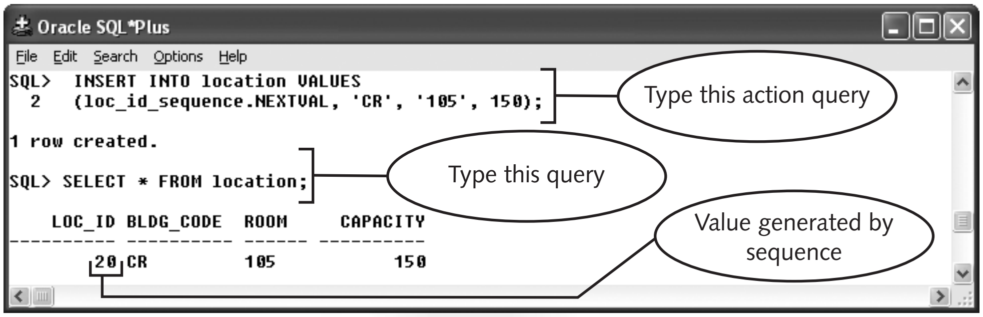
For example, you use the following INSERT action query to retrieve the next value in LOC_ID_SEQUENCE and insert the value into a row in the LOCATION table:

```
INSERT INTO location (LOC_ID)
VALUES(loc_id_sequence.NEXTVAL);
```

Next, you insert a row into the LOCATION table using LOC_ID_SEQUENCE and the NEXTVAL pseudocolumn.

To insert a new LOCATION row using the LOC_ID_SEQUENCE sequence:

1. Type the first action query shown in Figure 3-14.



```
SQL>  INSERT INTO location VALUES
  2   (loc_id_sequence.NEXTVAL, 'CR', '105', 150);

1 row created.

SQL> SELECT * FROM location;

    LOC_ID BLDG_CODE  ROOM     CAPACITY
---------- ---------- ------ ----------
        20 CR         105           150
```

Figure 3-14 Inserting a new row using a sequence

2. Execute the query. The confirmation message "1 row created." indicates that you inserted one row.
3. To view the value that the DBMS inserted in this row for LOC_ID, type the second query shown in Figure 3-14, and then press **Enter** to execute the query. This query retrieves the new rows from the LOCATION table, as shown in Figure 3-14. The new row should have a LOC_ID value of 20, which was the first value in the sequence, along with the other values you specified in the INSERT action query.

Sometimes you need to access the next value of a sequence, but you don't want to insert a new row. For example, suppose you want to display a faculty ID (F_ID) value on a form, but you need to have the faculty enter more information before you actually insert the new row into the database. To do this, you create a SELECT query that uses the sequence name with the NEXTVAL pseudocolumn using the following syntax:

```
SELECT sequence_name.NEXTVAL
FROM DUAL;
```

This query's FROM clause specifies the DUAL database table. **DUAL** is a simple table that belongs to the SYSTEM user schema. It has one row, which contains a single column that contains the character string 'X'. All database users can use DUAL in SELECT queries, but they cannot modify or delete values in DUAL. Whenever you execute a SELECT query using a pseudocolumn with any database table, the query returns a value for each table row. It is more efficient to retrieve psuedocolumns from DUAL, which has only one row, than from other tables that may contain many rows.

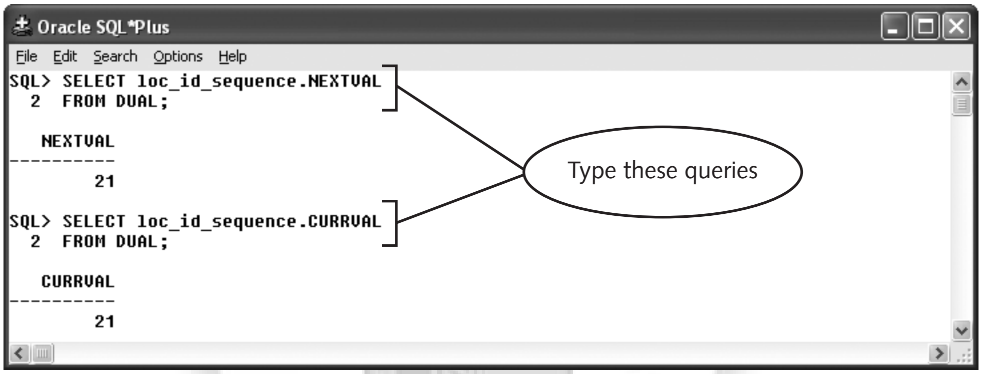
To view the current value of the sequence, you execute a SELECT query that uses the sequence name with the CURRVAL pseudocolumn, using the following syntax:

```
SELECT sequence_name.CURRVAL
FROM DUAL;
```

In the following set of steps, you use the SELECT queries to retrieve the next LOC_ID_SEQUENCE value, and then view the current value.

To use the SELECT queries to access sequence values:

1. Type and execute the first query shown in Figure 3-15. Your query output should show 21 as the NEXTVAL value in the LOC_ID_SEQUENCE. Now that it has been accessed, 21 is the current value of the sequence.
2. Type and execute the second query shown in Figure 3-15 to display the current sequence value. The query output confirms that 21 is now the current value of the LOC_ID_SEQUENCE.



```
SQL> SELECT loc_id_sequence.NEXTVAL
  2  FROM DUAL;

   NEXTVAL
----------
        21

SQL> SELECT loc_id_sequence.CURRVAL
  2  FROM DUAL;

   CURRVAL
----------
        21
```

Figure 3-15 Using the NEXTVAL and CURRVAL pseudocolumns in SELECT queries

Often multiple users simultaneously use a sequence to retrieve surrogate key values. The DBMS must guarantee that each user who accesses a sequence receives a unique sequence number. When a user connects to an Oracle10*g* database, the user creates a **user session**. A user session starts when a user starts a client application such as SQL*Plus, and logs onto the database. A user session terminates when the user exits the application. The Oracle10*g* DBMS uses user sessions to ensure that all sequence users receive unique sequence numbers. When the DBMS issues a sequence value to your user session, no other user's session can access that sequence value. This prevents two different users from assigning the same primary key values to rows.

You can use the CURRVAL pseudocolumn to access the last value that you retrieved from a sequence only during the same database user session in which you used the NEXTVAL pseudocolumn to retrieve the value. In the following steps you confirm the CURRVAL user session restriction by retrieving the next value in LOC_ID_SEQUENCE, exiting SQL*Plus, starting it again, and then trying to use the CURRVAL command to access the current sequence value.

To examine the CURRVAL database session restriction:

1. Type and execute the query **`SELECT loc_id_sequence.NEXTVAL FROM DUAL;`**. The value 22 should appear as the next sequence value.
2. Exit SQL*Plus.
3. Start SQL*Plus again, and log onto the database.
4. Type and execute the query **`SELECT loc_id_sequence.CURRVAL FROM DUAL;`**. You receive the error message "Error at line 1: ORA-08002: sequence LOC_ ID_SEQUENCE.CURRVAL is not yet defined in this session." This indicates that no CURRVAL exists for the sequence because you exited SQL*Plus and started a new SQL*Plus session, but have not yet retrieved a sequence value in this session. A CURRVAL can be selected from a sequence only after selecting a NEXTVAL.

Deleting Sequences

To delete a sequence from the database, you use the DROP SEQUENCE DDL command. For example, to drop LOC_ID_SEQUENCE, you use the command **`DROP SEQUENCE loc_id_sequence;`**. Do not drop LOC_ID_SEQUENCE right now, because you will need to use it for the rest of the exercises in this chapter. As with any DDL command, you do not need to commit the action explicitly when you drop a sequence.

Database Object Privileges

When you create database objects such as tables or sequences, other users cannot access or modify the objects in your user schema unless you give them explicit privileges to do so. Table 3-5 summarizes some of the commonly used Oracle10*g* object privileges for tables and sequences.

Object Type(s)	Privilege	Description
Table, Sequence	ALTER	Allows the user to change the object's structure using the ALTER command
Table, Sequence	DROP	Allows the user to drop (delete) the object
Table, Sequence	SELECT	Allows the user to view table records or select sequence values
Table	INSERT, UPDATE, DELETE	Allows the user to insert, update, or delete table records
Any object	ALL	Allows the user to perform all possible operations on the object

Table 3-5 Commonly used Oracle10*g* object privileges

The following sections describe how to grant and revoke object privileges.

Granting Object Privileges

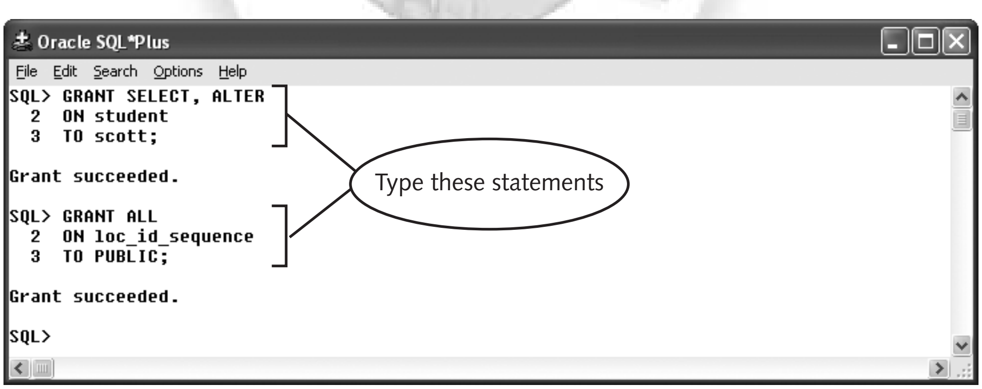
You grant object privileges to other users using the SQL GRANT command. The general syntax for the SQL GRANT command is:

```
GRANT privilege1, privilege2, ...
ON object_name
TO user1, user2, ...;
```

In this syntax, the GRANT clause lists each privilege to be granted, such as ALTER or DROP. *Object_name* represents the name of the object on which you are granting the privilege, such as the tablename or sequence name. The TO clause lists the name of the Oracle10*g* user account to which you are granting the privilege, such as SCOTT or DBUSER. Note that in a single command, you can grant privileges for only one object at a time, but you can grant privileges to many users at once. If you want to grant privileges to every database user, you can use the keyword PUBLIC in the TO clause. Next, you grant object privileges on some of your database objects to other database users.

To grant object privileges:

1. Type the first command shown in Figure 3-16 to grant SELECT and ALTER privileges on your STUDENT table to the SCOTT user account. (The Oracle10*g* DBMS automatically creates the SCOTT user account when the DBA installs the database.)



```
SQL> GRANT SELECT, ALTER
  2  ON student
  3  TO scott;

Grant succeeded.

SQL> GRANT ALL
  2  ON loc_id_sequence
  3  TO PUBLIC;

Grant succeeded.

SQL>
```

Figure 3-16 Granting object privileges

2. Execute the command. The message "Grant succeeded." confirms that the DBMS successfully granted the privileges.
3. Type the second command shown in Figure 3-16 to grant all privileges on your LOC_ID_SEQUENCE sequence to all database users, and then press **Enter** to execute the command. The message "Grant succeeded." again confirms that the DBMS successfully granted the privileges.

Revoking Table Privileges

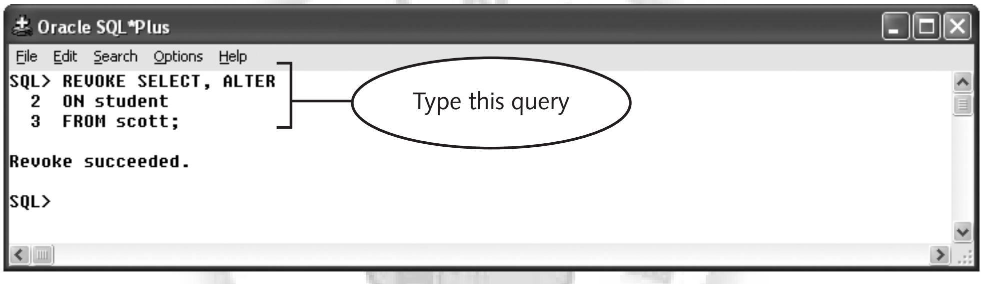
To cancel a user's object privileges, you use the REVOKE command. The general syntax for the REVOKE command is:

```
REVOKE privilege1, privilege2, ...
ON object_name
FROM user1, user2, ...;
```

Next, you revoke the privileges on the STUDENT table that you granted in the previous set of steps.

To revoke privileges:

1. Type the command shown in Figure 3-17 to revoke the SELECT and ALTER privileges that you granted to user SCOTT on the STUDENT table, and then execute the command.



```
SQL> REVOKE SELECT, ALTER
  2  ON student
  3  FROM scott;

Revoke succeeded.

SQL>
```

Figure 3-17 Revoking object privileges

2. Exit SQL*Plus.
3. Switch to Notepad, save the changes you made to Ch3AQueries.sql, and then exit Notepad.

NOTE When you grant a privilege to PUBLIC, the privilege can be revoked only by a user who has been granted the DBA system privileges.

Summary

- Database developers need to understand how to create SQL queries for inserting, updating, deleting, and viewing data because they use these commands when writing database applications.

- A script is a text file with a .sql extension that contains a sequence of SQL commands for creating or modifying database objects or manipulating data rows.
- You use an INSERT action query to insert new data rows into a table. Before you can add a new data row, you must ensure that all rows containing foreign key values referenced by the new row have already been added to the database.
- When you insert a new character column value, you must enclose the value in single quotation marks. Character column values are case sensitive. To add character column values that contain embedded single quotation marks, type the single quotation mark twice.
- To insert a new DATE column value, you insert the value as a character string, and then convert the value to the DATE format using the TO_DATE function. When you use the TO_DATE function, you specify the input format of the DATE character string using a format model, which is a text string that specifies how a data value is input or output.
- To insert a new INTERVAL column value, you insert the value as a character string, and then convert the value to an INTERVAL format using the TO_YMINTERVAL or TO_DSINTERVAL functions.
- One or more related action queries make up a transaction. A transaction is a logical unit of work. All parts of the transaction must be completed or the database might contain inconsistent data. You make the changes caused by a transaction's action queries permanent by issuing the COMMIT command. You use the ROLLBACK command to discard all of the transaction's changes.
- You use a SQL search condition to match one or more database rows. The DBMS must be able to evaluate a search condition as either TRUE or FALSE. You can combine search conditions using the AND, OR, and NOT logical operators.
- You update rows using the UPDATE action query. You can update multiple rows in a table using a single UPDATE command by specifying a search condition that matches multiple rows.
- You use the DELETE command to remove database rows. You can only delete rows from one table at a time. You cannot delete a row if one of its columns is referenced as a foreign key. To delete all of a table's rows quickly, you truncate the table. When you truncate a table, the DBMS does not save rollback information, so you cannot restore the rows.
- A sequence is a database object that automatically generates sequential numbers that are used for surrogate key values. You can access sequence values using the NEXTVAL and CURRVAL pseudocolumns within either an INSERT action query or a SELECT query.

REVIEW QUESTIONS

1. A(n) ________________ represents a logical unit of work.
2. The UPDATE command is used to add new rows to an existing table. True or False?
3. What is the rationale for using a date format model?
4. The ________________ command is used to update pending transactions to the database table(s).
5. What could prevent Oracle10*g* from executing a TRUNCATE command that does not have a syntax error?
6. The ________________ pseudocolumn is used to retrieve the next value in a sequence.
7. What is the difference between the AND and OR logical operators?
8. The DELETE command is used to remove objects from a database. True or False?
9. By default, an Oracle10*g* sequence is increased by a value of ________________ for each new integer generated.
10. How is a date or character string search condition different from a numeric search condition?

MULTIPLE CHOICE

1. Which of the following statements removes all rows from the ORDERS table?
   a. `DELETE * FROM orders;`
   b. `DROP * FROM orders;`
   c. `DELETE FROM orders;`
   d. `DROP FROM orders;`
2. Which of the following statements returns all orders placed on May 29, 2006?
   a. `SELECT * FROM orders WHERE o_date = 5/29/2006;`
   b. `SELECT * FROM orders WHERE o_date = '5/29/2006';`
   c. `SELECT * FROM orders WHERE o_date = '29-MAY-06';`
   d. none of the above
3. Which of the following statements correctly updates the received date for the shipment with the SHIP_ID of 2 to 11/19/2006?
   a.
   ```
   UPDATE shipment_line
   SET sl_date_received = TO_DATE('11/19/2006',
   'MM/DD/YYYY')WHERE ship_id = 2;
   ```
   b.
   ```
   UPDATE shipment_line
   SET sl_date_received = TO_DATE('11/19/2006',
   'MM/DD/YYYY');
   ```

c.
```
INSERT shipment_line
  SET sl_date_received = TO_DATE('11/19/2006',
  'MM/DD/YYYY');
```
d.
```
INSERT shipment_line
  SET sl_date_received = TO_DATE('11/19/2006',
  'MM/DD/YYYY')
  WHERE ship_id = 2;
```

4. Which of the following statements displays the next value in the ORDER_NO_SEQ sequence?
   a. `SELECT next.order_no_seq FROM DUAL;`
   b. `SELECT order_no_seq.nextvalue FROM DUAL;`
   c. `SELECT order_no_seq.currval FROM DUAL;`
   d. `SELECT order_no_seq.nextval FROM DUAL;`
5. Which of the following statements deletes all customers that reside in Florida?
   a. `DELETE FROM customer WHERE c_state = 'FL';`
   b. `DELETE FROM customer WHERE c_state = "FL";`
   c. `DELETE FROM customer WHERE c_state = FL;`
   d. `DELETE FROM customer WHERE c_state != FL;`
6. Which of the following commands is used to discard any uncommitted changes?
   a. ROLLBACK
   b. COMMIT
   c. DELETE
   d. UNDO
7. Which of the following SQL commands cannot be rolled back after it is executed?
   a. TRUNCATE
   b. DELETE
   c. INSERT
   d. UPDATE
8. Which of the following is a valid statement?
   a. You cannot truncate or drop a table that contains a constraint.
   b. You cannot drop a table that contains a foreign key constraint.
   c. You cannot drop a table that is being referenced by a foreign key constraint.
   d. all of the above

9. Which of the following is a valid statement?
   a. The CREATE SEQUENCE command is a DML, so the user must execute a COMMIT command before the sequence is permanently updated to the database.
   b. The CREATE SEQUENCE command is a DDL, so the user must execute a COMMIT command before the sequence is permanently updated to the database.
   c. The CREATE SEQUENCE command is a DML, so the sequence is permanently updated to the database when the command is executed.
   d. The CREATE SEQUENCE command is a DDL, so the sequence is permanently updated to the database when the command is executed.
10. Which of the following is a logical operator?
    a. !=
    b. OR
    c. IS NULL
    d. LIKE

Problem-solving Cases

Before starting any of the problem-solving cases, make certain you run the Ch3EmptyClearwater.sql script file stored in the Chapter03 folder of your Data Disk to create the necessary tables. Case 1 must be completed before attempting the remaining cases.

1. Create a script named Ch03Case1.sql to perform the actions indicated below. Store the script file in the Chapter03 folder of your Data Disk, and then execute the script in SQL*Plus.
   a. Add each customer shown in Figure 1-24 to the CUSTOMER table.
   b. Create a sequence named ORDERS_ID_SEQ to generate the O_ID values for the ORDERS table shown in Figure 1-24.
   c. Add each order shown in Figure 1-24 to the ORDERS table. Use the ORDERS_ID_SEQ sequence to enter the appropriate value for the O_ID column.
   d. Add each order source shown in Figure 1-24 to the ORDER_SOURCE table.
   e. Make certain the rows you have added are permanently updated to the database tables.

2. Execute the appropriate commands to complete the specified tasks. Save a copy of your commands in a script file named Ch03Case2.sql in the Chapter03 folder of your Data Disk.
   a. Remove the customer with the C_ID of 5 from the CUSTOMER table. If an error occurs while attempting to remove the customer's row, take the necessary corrective actions to complete the task.
   b. Update the order with the O_ID of 6 in the ORDERS table so the payment method is money order (MO) rather than credit card (CC).
   c. Add a new catalog to the ORDER_SOURCE database table. The new catalog is the Winter 2006 version and should be assigned the ID of 7.
   d. Discard the changes you made in the previous three steps.
3. Execute the appropriate commands to complete the specified tasks. Save a copy of your commands in a script file named Ch03Case3.sql in the Chapter03 folder of your Data Disk.
   a. Create a sequence named CUSTOMER_ID_SEQ that generates the ID for each customer added to the CUSTOMER table.
   b. Drop and then re-create the CUSTOMER_ID_SEQ sequence so each subsequent ID generated increases by a value of 10.
   c. Display the current contents of the CUSTOMER table to determine if the change to the sequence had any impact on the existing values in the C_ID column.
   d. Remove the CUSTOMER_ID_SEQ sequence from the database.
4. Execute the appropriate commands to complete the specified tasks. Save a copy of your commands in a script file named Ch03Case4.sql in the Chapter03 folder of your Data Disk.
   a. Disable the foreign key constraint in the Clearwater database that references the CUSTOMER table.
   b. Remove all customer rows from the CUSTOMER table.
   c. Issue the command `SELECT * FROM CUSTOMER;` to verify that there are no rows in the table. Use the ROLLBACK command to restore the rows. Then issue the SELECT command again to verify that the rows are now in the CUSTOMER table.
   d. Create a savepoint named SAVEPOINT1.
   e. Truncate the contents of the CUSTOMER, ORDERS, and ORDER_SOURCE tables. Use the command `SELECT * FROM ORDERS;` to verify removal of the table data. Attempt to roll back to the SAVEPOINT1 savepoint and then verify the contents of the ORDERS table. Were you able to roll back truncation of the table? Why or why not?

5. Execute the appropriate commands to complete the specified tasks. Save a copy of your commands in a script file named Ch03Case5.sql in the Chapter03 folder of your Data Disk.
   a. Grant the user SCOTT the privilege to view and alter your CUSTOMER table.
   b. Revoke from the user SCOTT the privilege to alter your CUSTOMER table.
   c. Delete the CUSTOMER, ORDERS, and ORDER_SOURCE tables from your database.
   d. Attempt to roll back deletion of the CUSTOMER table. Were you able to successfully complete this step? Why or why not?

◀ LESSON B ▶

After completing this chapter, you should be able to:

- ♦ Write SQL queries to retrieve data from a single database table
- ♦ Create SQL queries that perform calculations on retrieved data
- ♦ Use SQL group functions to summarize retrieved data

Now that you have learned how to insert, update, and delete data rows using SQL*Plus, the next step is to learn how to retrieve data. In this lesson, you learn how to write SQL queries to retrieve data from a single database table, sort the output, and perform calculations on data, for example, calculating a person's age from data stored in a date-of-birth column or calculating a total for an invoice. You also learn how to format retrieved data in SQL*Plus. For the rest of this chapter, you execute SELECT queries using fully populated Northwoods University database tables. You create these tables by running the Ch3Northwoods.sql script in the Chapter3 folder on your Data Disk. This script contains SQL commands to drop all of the tables in these databases, re-create the tables, and then insert all of the table rows. Follow the set of steps to run the script file.

To run the script files:

1. If necessary, start SQL*Plus and log onto the database.
2. Run the Ch3Northwoods.sql script by typing the following command at the SQL prompt:

   ```
   START c:\OraData\Chapter03\Ch3Northwoods.sql.
   ```

HELP

Recall that you might receive an error message if the script tries to drop a table that has not yet been created. (See Figure 3-1.) You can ignore this error message.

Retrieving Data from a Single Database Table

The basic syntax for a SQL query that retrieves data from a single database table is:

```
SELECT columnname1, columnname2, ...
FROM ownername.tablename
[WHERE search_condition];
```

The SELECT clause lists the columns whose values you want to retrieve. In the FROM clause, *ownername* specifies the table's user schema, and *tablename* specifies the name of the database table. If you are retrieving data from a table in your own user schema, you can omit *ownername* (and the period) in the FROM clause. If you are retrieving data from a table in another user's schema, you must include *ownername* in the FROM clause, and the table's owner must have granted you the SELECT privilege on the table. For example, you use the following FROM clause to retrieve data values from the LOCATION table in user SCOTT's user schema:

```
FROM scott.location
```

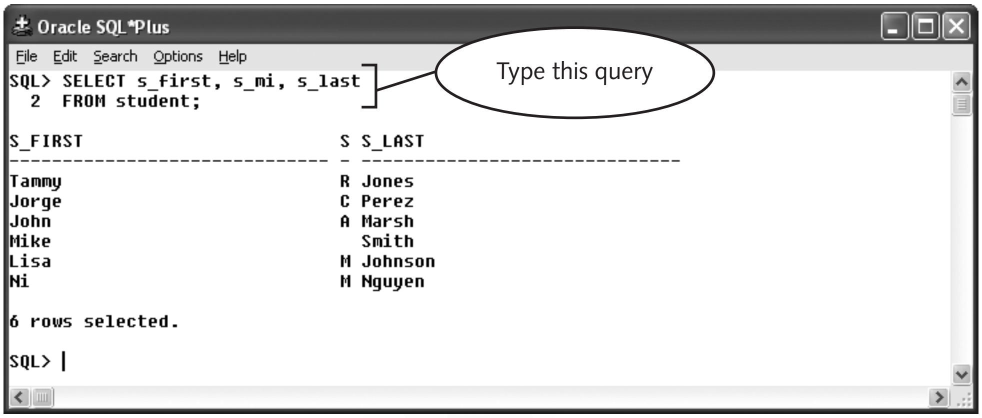
The WHERE clause optionally specifies a search condition that instructs the query to retrieve selected rows. To retrieve every row in a table, the data values do not need to satisfy a search condition, so you can omit the WHERE clause. Next, you retrieve the first name, middle initial, and last name from every row in the Northwoods University STUDENT table.

To retrieve selected column values from every row of the STUDENT table:

1. Start Notepad, create a new file named **Ch3BQueries.sql**, and then type the query in Figure 3-18 to retrieve the student first name, middle initial, and last name from every row in the STUDENT table. Copy and paste the query into SQL*Plus.

NOTE

In this lesson, you enter all of the tutorial commands in your Ch3BQueries.sql file, and then execute them in SQL*Plus.



```
SQL> SELECT s_first, s_mi, s_last
  2  FROM student;

S_FIRST                        S S_LAST
------------------------------ - ------------------------------
Tammy                          R Jones
Jorge                          C Perez
John                           A Marsh
Mike                             Smith
Lisa                           M Johnson
Ni                             M Nguyen

6 rows selected.

SQL> |
```

Figure 3-18 Retrieving selected column values

2. Press **Enter** to execute the query. Your output should look like the query output in Figure 3-18. Note that the column names appear as column headings, and the column values appear in the same order in which you listed them in the SELECT clause.

If you want to retrieve all of the columns in a table, you can use an asterisk (*) as a wildcard character in the SELECT clause instead of typing every column name. Now you write a query to retrieve every row and every column in the LOCATION table.

To retrieve every row and column from the LOCATION table:

1. Type and execute the following query to select all rows and columns from the LOCATION table:

```
SELECT *
FROM location;
```

NOTE

By default, the SQL*Plus environment shows query output one page at a time. You learn how to modify the display later in this lesson in the section titled "Modifying the SQL*Plus Display Environment."

Suppressing Duplicate Rows

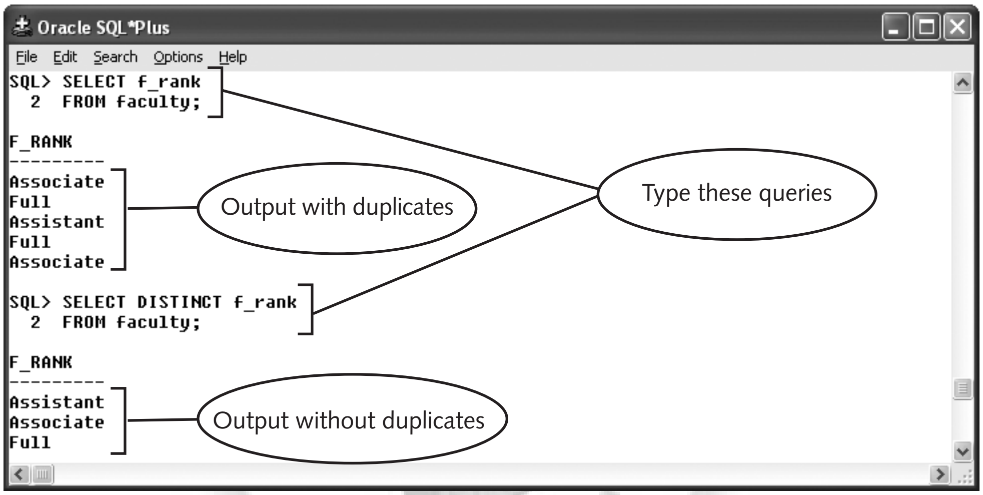
Sometimes a query retrieves duplicate data values. For example, suppose you want to view the different ranks of faculty members at Northwoods University. Some of the faculty members have the same rank, so the query `SELECT f_rank FROM faculty;` retrieves duplicate values. The SQL DISTINCT qualifier examines query output before it appears on your screen and suppresses duplicate values. The DISTINCT qualifier has the following general syntax in the SELECT command:

```
SELECT DISTINCT columnname;
```

For example, to suppress duplicate faculty ranks, you use the command `SELECT DISTINCT f_rank FROM faculty;`. Next, you execute the query to retrieve all of the faculty ranks and display the duplicate values. Then, you execute it again using the DISTINCT qualifier to suppress the duplicates.

To retrieve and suppress duplicate rows:

1. Type and execute the first query in Figure 3-19. The Associate value appears twice because two rows in the table contain the Associate value in the F_RANK column.



```
SQL> SELECT f_rank
  2  FROM faculty;

F_RANK
---------
Associate
Full
Assistant
Full
Associate

SQL> SELECT DISTINCT f_rank
  2  FROM faculty;

F_RANK
---------
Assistant
Associate
Full
```

Figure 3-19 Output with duplicate rows suppressed

2. To suppress duplicate values, type and execute the second query in Figure 3-19, which uses the DISTINCT qualifier. The output now appears with the duplicate rows suppressed.

Using Search Conditions in SELECT Queries

So far, the queries that you have executed have retrieved all of the rows in the table. To retrieve rows matching specific criteria, you can use search conditions. Recall that you used search conditions in UPDATE and DELETE action queries to match specific data rows, and that search conditions use comparison operators to match a data value or range of values. The following sections describe how to use search conditions in SELECT queries.

Exact and Inexact Search Conditions

An **exact search condition** uses the equal to comparison operator (=) to match a value exactly, whereas an **inexact search condition** uses the inequality comparison operators (>, <, >=, <=) to match a range of values. Next, you use an exact search condition to retrieve the rows in the FACULTY table for which the F_RANK value is Associate.

To use an exact search condition in a SELECT query:

1. Type the following query to retrieve specific rows in the FACULTY table:

   ```
   SELECT f_first, f_mi, f_last, f_rank
   FROM faculty
   WHERE f_rank = 'Associate';
   ```

2. Execute the query. The output lists the first name, middle initial, last name, and rank of all faculty members with the rank of Associate.

Next you create an inexact search condition that retrieves the number of every room in the Business ('BUS') building at Northwoods University that has a capacity greater than or equal to 40 seats. You use the greater than or equal to (>=) comparison operator in the inexact search condition.

To use an inexact search condition in a SELECT query:

1. Type the following query to retrieve specific rows in the LOCATION table:

   ```
   SELECT room
   FROM location
   WHERE bldg_code = 'BUS'
   AND capacity >= 40;
   ```

2. Execute the query. The output reports that rooms 105 and 211 match the search criteria.

Searching for NULL and NOT NULL Values

Sometimes you need to create a query to return rows in which the value of a particular column is NULL. For example, you might want to retrieve enrollment rows for courses in which the instructor has not yet assigned a grade, so the GRADE value is NULL. To search for NULL values, you use the following general syntax:

```
WHERE columnname IS NULL
```

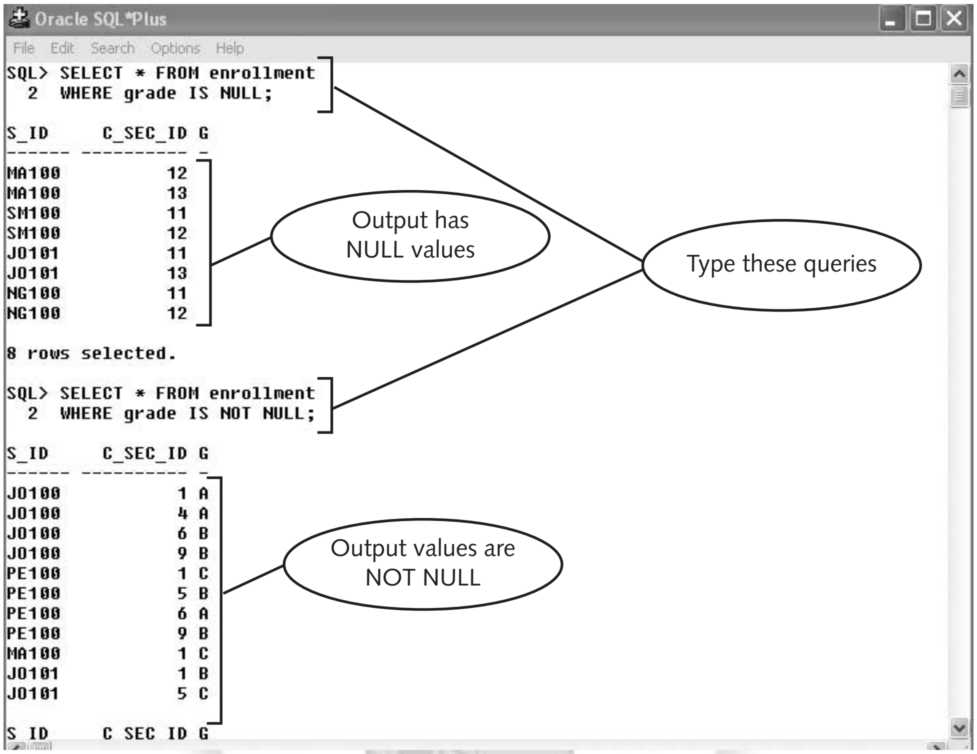
Similarly, to retrieve rows in which the value of a particular column is not NULL, you use the following syntax:

```
WHERE columnname IS NOT NULL
```

Next, you create queries to retrieve rows from the ENROLLMENT table in which specific column values are NULL or NOT NULL.

To create queries using the IS NULL and IS NOT NULL search conditions:

1. Type and execute the first query in Figure 3-20 to retrieve all rows in the ENROLLMENT table for which the GRADE column value is NULL. The returned rows show all enrollment rows for which the instructor has not yet assigned a grade value.



```
SQL> SELECT * FROM enrollment
  2  WHERE grade IS NULL;

S_ID     C_SEC_ID G
------ ---------- -
MA100          12
MA100          13
SM100          11
SM100          12
JO101          11
JO101          13
NG100          11
NG100          12

8 rows selected.

SQL> SELECT * FROM enrollment
  2  WHERE grade IS NOT NULL;

S_ID     C_SEC_ID G
------ ---------- -
JO100           1 A
JO100           4 A
JO100           6 B
JO100           9 B
PE100           1 C
PE100           5 B
PE100           6 A
PE100           9 B
MA100           1 C
JO101           1 B
JO101           5 C

S_ID     C_SEC_ID G
```

Figure 3-20 Queries using IS NULL and IS NOT NULL search conditions (partial output shown)

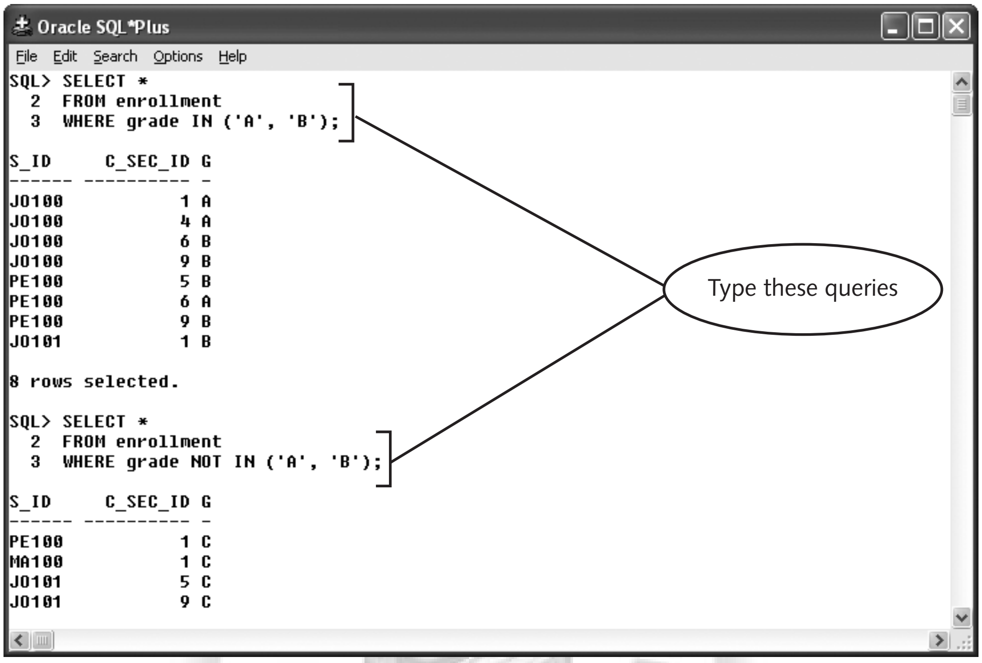
2. Type and execute the second query in Figure 3-20 to retrieve the enrollment rows for which the instructor has assigned a grade value.

Using the IN and NOT IN Comparison Operators

You can use the IN comparison operator to match data values that are members of a set of search values. For example, you can retrieve all enrollment rows in which the GRADE column value is a member of the set ('A', 'B'). Similarly, you can use the NOT IN comparison operator to match values that are not members of a set of search values. Now you retrieve rows using the IN and NOT IN comparison operators.

To retrieve rows using the IN and NOT IN comparison operators:

1. Type and execute the first query in Figure 3-21 to retrieve every enrollment row in which the GRADE value is either A or B.
2. Type and execute the second query in Figure 3-21 to retrieve every enrollment row in which the GRADE value is neither A nor B.



```
SQL> SELECT *
  2  FROM enrollment
  3  WHERE grade IN ('A', 'B');

S_ID     C_SEC_ID G
------ ---------- -
JO100           1 A
JO100           4 A
JO100           6 B
JO100           9 B
PE100           5 B
PE100           6 A
PE100           9 B
JO101           1 B

8 rows selected.

SQL> SELECT *
  2  FROM enrollment
  3  WHERE grade NOT IN ('A', 'B');

S_ID     C_SEC_ID G
------ ---------- -
PE100           1 C
MA100           1 C
JO101           5 C
JO101           9 C
```

Figure 3-21 Using the IN and NOT IN comparison operators

Using the LIKE Comparison Operators

Sometimes, you need to perform searches by matching part of a character string. For example, you might want to retrieve rows for students whose last name begins with the letter M, or find all courses with the text string MIS in the COURSE_NO column. To do this, you use the LIKE operator. The general syntax of a search condition that uses the LIKE operator is:

```
WHERE columnname LIKE 'character_string'
```

Character_string represents the text string to be matched and is enclosed in single quotation marks. *Character_string* must contain either the percent sign (`%`) or underscore (`_`) wildcard characters.

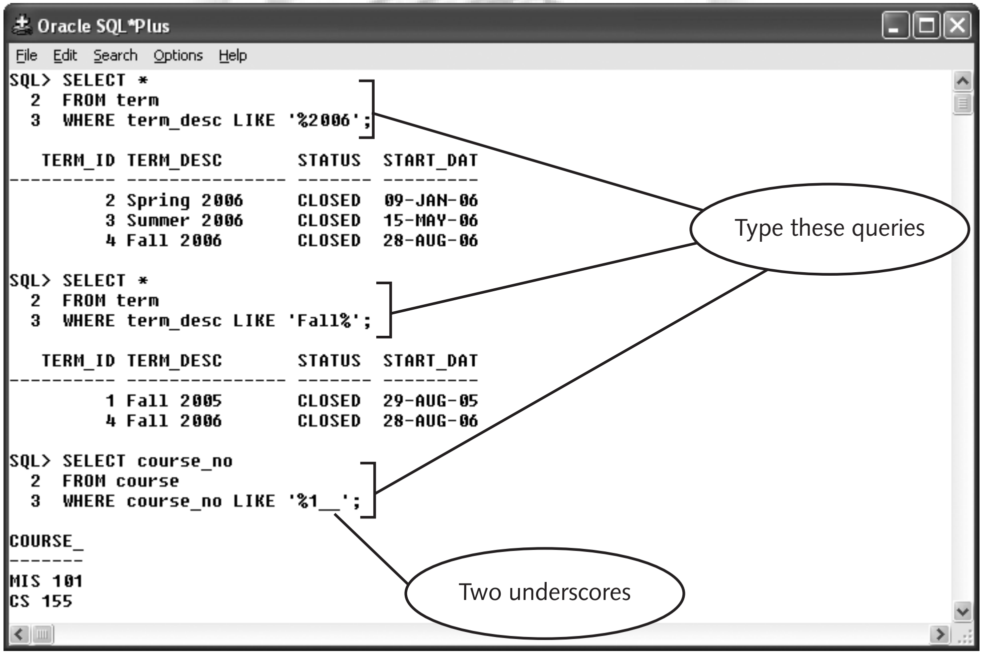
The percent sign (`%`) wildcard character represents multiple characters. If you place (`%`) on the left edge of the character string to be matched, the DBMS searches for an exact match on the far-right characters and allows an inexact match for the characters represented by (`%`). For example, the search condition `WHERE term_desc LIKE '%2006'` retrieves all term rows in which the last four characters in the TERM_DESC column are 2006. The DBMS ignores the characters on the left side of the character string up to the substring 2006. Similarly, the search condition `WHERE term_desc LIKE 'Fall%'` retrieves all term rows in which the first four characters are Fall, regardless of the value of the rest of the string. The search condition `WHERE course_name LIKE '%Systems%'` retrieves every row in the COURSE table in which the COURSENAME column contains the character string Systems anywhere in the string. In the search condition `LIKE '%Systems%'`, the (`%`) wildcard character does not require that characters be present before or after the character string Systems

The underscore (_) wildcard character represents a single character. For example, the search condition `WHERE s_class LIKE '_R'` retrieves all values for S_CLASS in which the first character can be any value, but the second character must be the letter R.

You can use the underscore (_) and percent sign (`%`) wildcard characters together in a single search condition. For example, the search condition `WHERE c_sec_day LIKE '_T%'` retrieves all course sections that meet on Tuesday, provided exactly one character precedes T in the C_SEC_DAY column. The search condition ignores all of the characters that follow T in the column value, so the query retrieves values such as MT, MTW, and MTWRF. Next, you create some queries that use the LIKE operator.

To create queries using the LIKE operator:

1. Type and execute the first query in Figure 3-22 to retrieve all rows from the TERM table in which the last four characters of the TERM_DESC column are 2006.



```
SQL> SELECT *
  2  FROM term
  3  WHERE term_desc LIKE '%2006';

   TERM_ID TERM_DESC       STATUS  START_DAT
---------- --------------- ------- ---------
         2 Spring 2006     CLOSED  09-JAN-06
         3 Summer 2006     CLOSED  15-MAY-06
         4 Fall 2006       CLOSED  28-AUG-06

SQL> SELECT *
  2  FROM term
  3  WHERE term_desc LIKE 'Fall%';

   TERM_ID TERM_DESC       STATUS  START_DAT
---------- --------------- ------- ---------
         1 Fall 2005       CLOSED  29-AUG-05
         4 Fall 2006       CLOSED  28-AUG-06

SQL> SELECT course_no
  2  FROM course
  3  WHERE course_no LIKE '%1__';

COURSE_
-------
MIS 101
CS 155
```

Figure 3-22 Using the LIKE comparison operator

2. Type and execute the second query in Figure 3-22 to retrieve all rows from the TERM table in which the first four letters are Fall. Note that the characters in single quotation marks are always case sensitive.
3. Type and execute the third query in Figure 3-22 to search for a COURSE_NO value from the COURSE table in which the third to last character is the number 1. This search expression uses the (`%`) wildcard operator to match any characters on the left end of the string, then specifies the number 1, followed by two underscores (`__`) to specify that exactly two characters follow the number 1.

Sorting Query Output

When you insert rows into an Oracle database, the DBMS does not store the rows in any particular order. When you retrieve rows using a SELECT query, the rows may appear in the same order in which you inserted them into the database, or they may appear in a different order, based on the database's storage configuration. You can sort query output by using the **ORDER BY clause** and specifying the **sort key**, which is the column the DBMS uses as a basis for ordering the data. The syntax for a SELECT query that uses the ORDER BY clause is as follows:

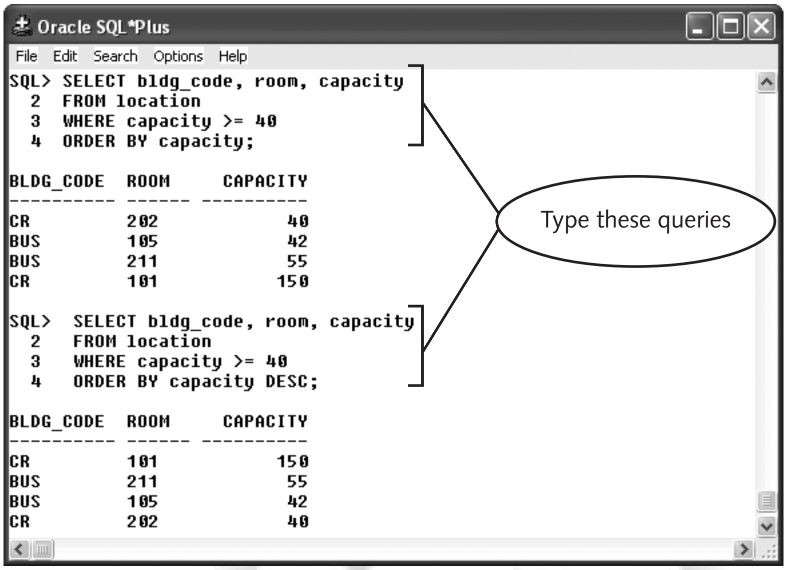
```
SELECT columnname1, columnname2, ...
FROM ownername.tablename
WHERE search_condition
ORDER BY sort_key_column;
```

In this syntax, *sort_key_column* can be the name of any column within the SELECT clause. If *sort_key_column* has the NUMBER data type, the DBMS by default sorts the rows in numerical ascending order. If *sort_key_column* has one of the character data types, the DBMS by default sorts the rows in alphabetical order. If *sort_key_column* has the DATE data type, the DBMS by default sorts the rows from older dates to more recent dates. To sort the rows in the reverse order from the default, use the DESC qualifier, which stands for *descending*. You place the DESC qualifier at the end of the ORDER BY clause, using the following syntax: `ORDER BY sort_key_column DESC`.

In the following steps, you retrieve rows from the LOCATION table and sort the rows based on the CAPACITY column.

To use the ORDER BY clause to sort data:

1. Type and execute the first query in Figure 3-23 to list the building code, room number, and capacity for every room with a capacity that is greater than or equal to 40, sorted in ascending order by capacity.



```
SQL> SELECT bldg_code, room, capacity
  2  FROM location
  3  WHERE capacity >= 40
  4  ORDER BY capacity;

BLDG_CODE  ROOM     CAPACITY
---------- ------ ----------
CR         202            40
BUS        105            42
BUS        211            55
CR         101           150

SQL>  SELECT bldg_code, room, capacity
  2   FROM location
  3   WHERE capacity >= 40
  4   ORDER BY capacity DESC;

BLDG_CODE  ROOM     CAPACITY
---------- ------ ----------
CR         101           150
BUS        211            55
BUS        105            42
CR         202            40
```

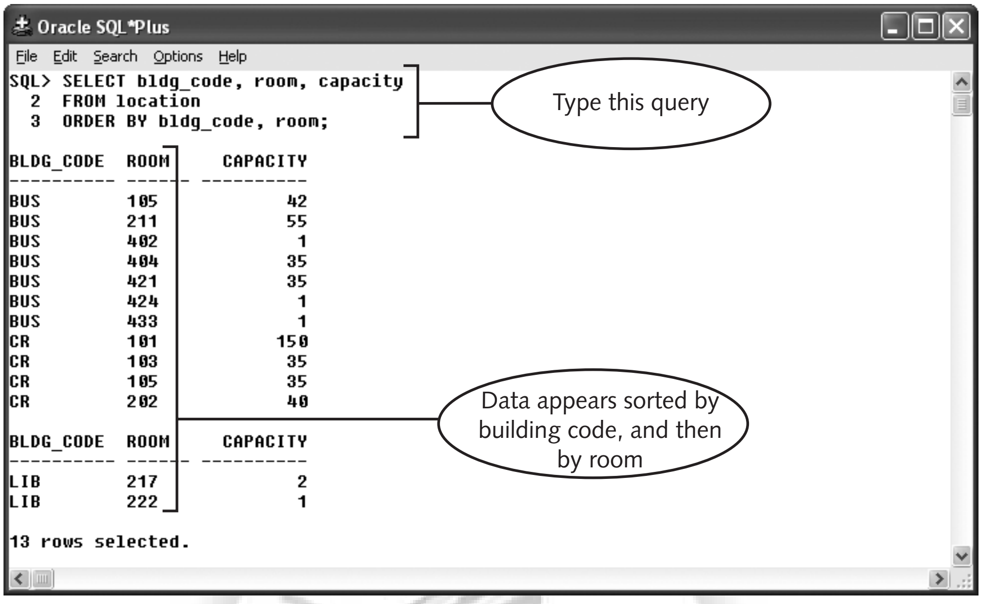
Figure 3-23 Using the ORDER BY clause to sort data

2. Type and execute the second query in Figure 3-23 to retrieve the same rows, but add the DESC qualifier at the end of the ORDER BY clause to sort the rows in descending order.

You can specify multiple sort keys to sort query output on the basis of multiple columns. You specify which column gets sorted first, second, and so forth, by the order of the sort keys in the ORDER BY clause. The next query lists all building codes, rooms, and capacities, sorted first by building code and then by room number.

To sort data on the basis of multiple columns:

1. Type the query in Figure 3-24 to sort the rows by building codes and then by room numbers.
2. Execute the query. The query output lists every row in the LOCATION table, first sorted alphabetically by building code, and then sorted within building codes by ascending room numbers.


Oracle SQL*Plus
File Edit Search Options Help
Type this query
Data appears sorted by building code, and then by room

```
SQL> SELECT bldg_code, room, capacity
  2  FROM location
  3  ORDER BY bldg_code, room;

BLDG_CODE  ROOM     CAPACITY
---------- ------ ----------
BUS        105            42
BUS        211            55
BUS        402             1
BUS        404            35
BUS        421            35
BUS        424             1
BUS        433             1
CR         101           150
CR         103            35
CR         105            35
CR         202            40

BLDG_CODE  ROOM     CAPACITY
---------- ------ ----------
LIB        217             2
LIB        222             1

13 rows selected.
```

Figure 3-24 Sorting data using multiple sort keys

USING CALCULATIONS IN SQL QUERIES

Database applications often display values based on calculations on retrieved data values. For example, an application may display a student's age based on his or her date of birth, or display the total cost of a customer order. You can perform many calculations directly within SQL queries. This is a very efficient way to perform calculations, because the DBMS returns to the client workstation only the calculated value, rather than all of the data values that contribute to the calculated value. For example, to calculate the total cost of a customer order yourself, you need to retrieve the item price and the order quantity for each order item. For an order with many different items, this could involve many rows. However, if the DBMS calculates the order total and sends only the calculated value to the client, it generates less network traffic, and system performance improves.

You can create SQL queries to perform basic arithmetic calculations and to use a variety of built-in functions. The following sections describe how to create queries using calculations and functions.

Performing Arithmetic Calculations

Table 3-6 lists the SQL arithmetic operations and their associated SQL operators in order of precedence.

Operation	Operator
Multiplication, Division	*, /
Addition, Subtraction	+, –

Table 3-6 Operators used in SQL query calculations

In mathematics and in programming languages, arithmetic expressions that combine multiple operations and contain more than one operator must be evaluated in a specific order. SQL evaluates division and multiplication operations first, and addition and subtraction operations last. SQL always evaluates arithmetic calculations enclosed within parentheses first.

You can perform arithmetic calculations on columns that have the NUMBER, DATE, or INTERVAL data types. The following sections describe how to perform these calculations for these data types.

Number Calculations

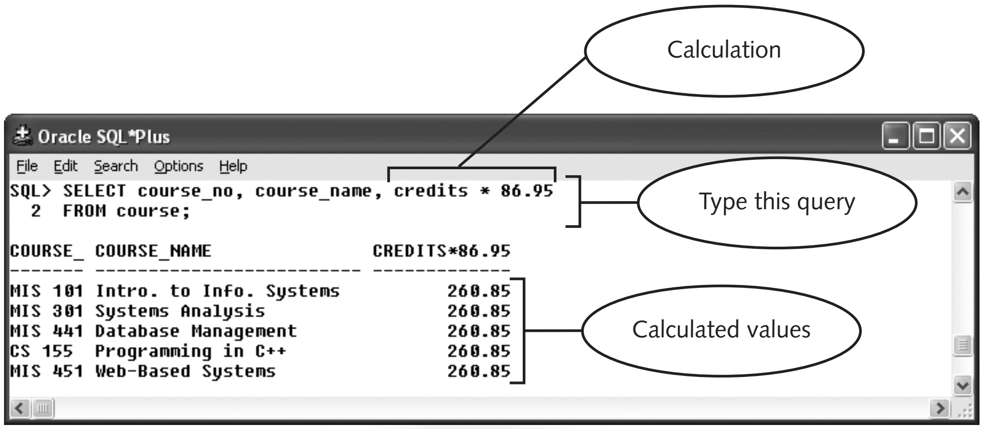
You can perform any basic arithmetic operations on columns that have the NUMBER data type by placing the operator in the SELECT clause along with the column names. For example, you use the following SELECT clause to retrieve the product of the INV_PRICE times the INV_QOH columns in the INVENTORY table:

```
SELECT inv_price * inv_qoh
```

In addition, you can include constants—actual numeric values—in the calculations performed by a query. For example, suppose Northwoods University charges $86.95 per credit hour for tuition. To determine how much tuition a student is charged for a class, you can simply multiply the number of credit hours earned for a course by the credit hour tuition rate. You use the multiplication operator (*) to derive the tuition amount for each retrieved row.

To calculate values in a SQL query:

1. Type the query in Figure 3-25.
2. Execute the query. Note that the query output shows the calculated value in a column whose heading is the calculation formula.



```
SQL> SELECT course_no, course_name, credits * 86.95
  2  FROM course;

COURSE_ COURSE_NAME              CREDITS*86.95
------- ------------------------ -------------
MIS 101 Intro. to Info. Systems         260.85
MIS 301 Systems Analysis                260.85
MIS 441 Database Management             260.85
CS 155  Programming in C++              260.85
MIS 451 Web-Based Systems               260.85
```

Figure 3-25 Query with calculated values

Date Calculations

Database applications often perform calculations involving dates. You may need to determine a date that is a specific number of days before or after a known date. Or you may need to determine the number of days between two known dates. Often, these calculations involve the current date. To retrieve the current system date from the database server, you use the **SYSDATE pseudocolumn**. (Recall from Lesson A in this chapter that a pseudocolumn acts like a column in a database table, but is actually a command that returns a specific value.) The following query retrieves the current system date:

```
SELECT SYSDATE
FROM DUAL;
```

TIP If you retrieve the SYSDATE value along with other database columns from another table, you can omit DUAL from the FROM clause.

To add a specific number of days to a known date, you add the number of days, expressed as an integer, to the known date. To subtract a specific number of days from a known date, you subtract the number of days, expressed as an integer, from the known date. For example, the following expression specifies a date that is 10 days after the order date (O_DATE) in the Clearwater Traders ORDERS table:

```
o_date + 10
```

For the first row in the ORDERS table, this expression returns the value 6/08/2006, which is 10 days after the O_DATE value of 5/29/2006.

TIP You cannot multiply or divide values of the DATE data type.

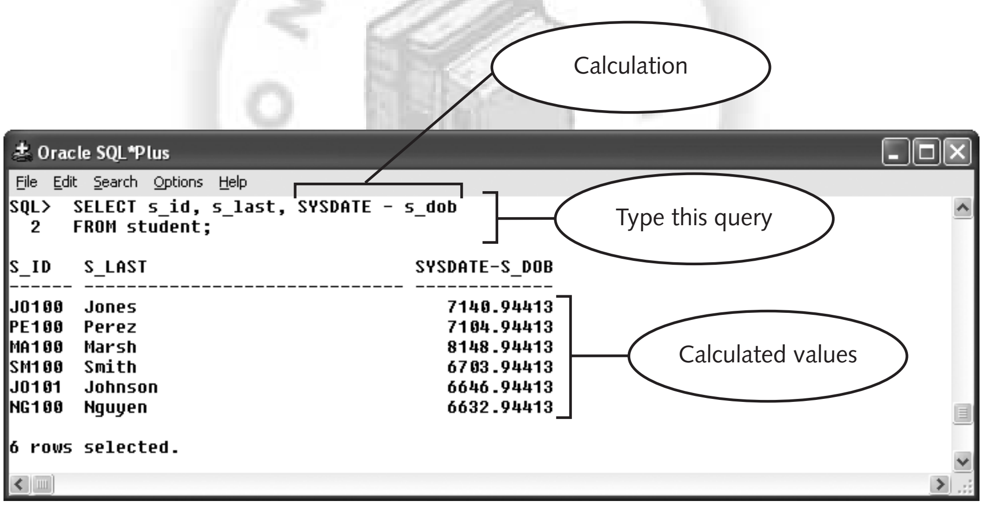
Sometimes you need to determine the number of days between two known dates. To return an integer value that represents the number of days between two dates, you subtract the first date from the second date. If the first date is after the second date, the result is a positive integer. If the first date is before the second date, the result is a negative integer. The following expression returns the number of days between the current date and the order date column in the ORDERS table:

```
SYSDATE - o_date
```

You usually do not store a person's age in a database because ages change from year to year. Rather, you store the person's date of birth and calculate his or her age based on the current system date. Next, you create a query that retrieves student information and calculates the student's age based on the student date of birth (S_DOB) data column and the current date.

To use the SYSDATE function to calculate student ages:

1. Type the query in Figure 3-26 to list the student ID, last name, and age for each student. The query calculates the student ages by subtracting the student dates of birth from the current date.



```
SQL>  SELECT s_id, s_last, SYSDATE - s_dob
  2   FROM student;

S_ID   S_LAST                          SYSDATE-S_DOB
------ ------------------------------ -------------
JO100  Jones                              7140.94413
PE100  Perez                              7104.94413
MA100  Marsh                              8148.94413
SM100  Smith                              6703.94413
JO101  Johnson                            6646.94413
NG100  Nguyen                             6632.94413

6 rows selected.
```

Figure 3-26 Calculating student ages based on dates of birth

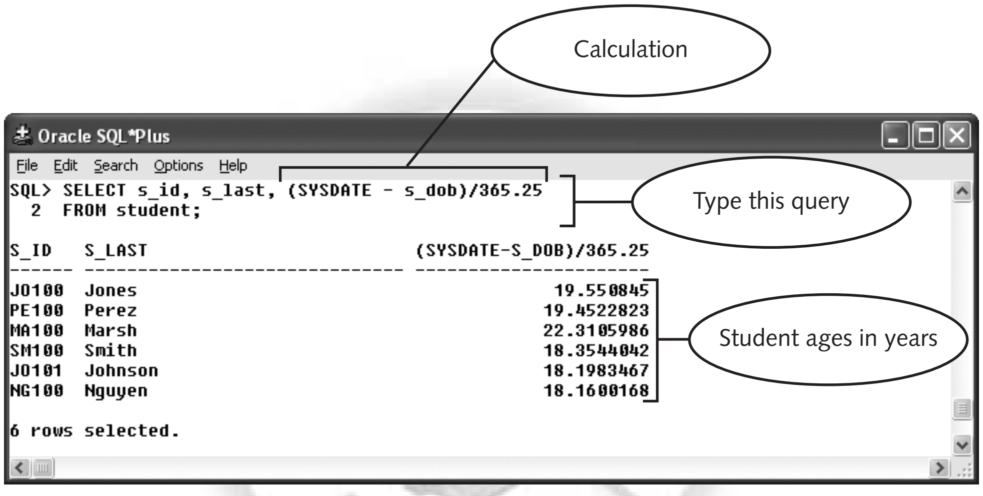
2. Execute the query. The query output lists the calculated ages in days rather than in years. (Your query output will be different because the output depends on the current system date.)

To express the calculated ages in years instead of days, you must divide these values by the number of days in a year, which is approximately 365.25 (including leap years). You can do this calculation in SQL by combining multiple arithmetic operations in a single query. Recall that in SQL commands, the DBMS evaluates multiplication and division operations first, and then evaluates addition and subtraction operations. To evaluate the subtraction operation *before* the division operation, you must place the subtraction operation in parentheses using the following expression:

```
(SYSDATE – S_DOB)/365.25
```

To convert the days into years by performing multiple arithmetic operations in a specific order:

1. Type the query in Figure 3-27, which uses parentheses to specify the order of the arithmetic operations.



```
SQL> SELECT s_id, s_last, (SYSDATE - s_dob)/365.25
  2  FROM student;

S_ID   S_LAST                         (SYSDATE-S_DOB)/365.25
------ ------------------------------ ----------------------
JO100  Jones                                       19.550845
PE100  Perez                                      19.4522823
MA100  Marsh                                      22.3105986
SM100  Smith                                      18.3544042
JO101  Johnson                                    18.1983467
NG100  Nguyen                                     18.1600168

6 rows selected.
```

Figure 3-27 Combining multiple arithmetic operations in a single query

2. Execute the query. The query output now lists the students' ages in years. (Your output will be different, based on your current system date.)

Interval Calculations

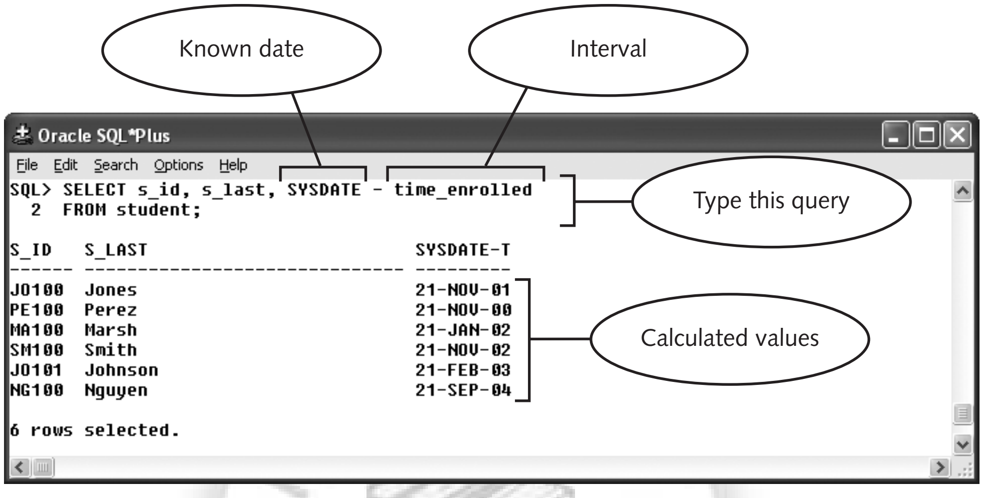
The Oracle10*g* DBMS can perform calculations using interval values that store elapsed time values. To specify a date that is before or after a known date, you can add to or subtract from the known date a column value that has the INTERVAL data type. Recall that in the STUDENT table, the TIME_ENROLLED column contains interval values that specify how long each student has been enrolled at Northwoods University. To determine the date on which a student enrolled in the university, you subtract the TIME_ENROLLED value from the current system date, as in the following expression:

```
SYSDATE – time_enrolled
```

Now you create a query that uses an interval calculation to determine the date each student enrolled in the university.

To use an interval calculation:

1. Type the query in Figure 3-28, which subtracts an INTERVAL value from a known date.



```
SQL> SELECT s_id, s_last, SYSDATE - time_enrolled
  2  FROM student;

S_ID   S_LAST                         SYSDATE-T
------ ------------------------------ ---------
JO100  Jones                          21-NOV-01
PE100  Perez                          21-NOV-00
MA100  Marsh                          21-JAN-02
SM100  Smith                          21-NOV-02
JO101  Johnson                        21-FEB-03
NG100  Nguyen                         21-SEP-04

6 rows selected.
```

Figure 3-28 Subtracting an interval from a known date

2. Execute the query. The query output lists the dates the students enrolled at the university, based on the INTERVAL value and the known date.

You can also add or subtract intervals from one another to calculate the sum or difference of two intervals. For example, suppose you need to update the TIME_ENROLLED column every month by adding one month to the interval value. You use the following query to add an interval of one month to the current TIME_ENROLLED value:

```
SELECT s_id, time_enrolled + TO_YMINTERVAL('0-1')
FROM student;
```

TIP Recall that you use the TO_YMINTERVAL function to convert a text string that represents a year and month interval to an INTERVAL data type.

Similarly, you use the following query to add an interval of 10 minutes to the C_SEC_DURATION column in the COURSE_SECTION table:

```
SELECT c_sec_id, c_sec_duration +
TO_DSINTERVAL('0 00:10:00')
FROM course_section;
```

TIP Recall that you use the TO_DSINTERVAL function to convert a text string that represents days, hours, minutes, and seconds to an INTERVAL data type.

Oracle10*g* SQL Functions

Oracle10*g* provides a number of built-in functions to perform calculations and manipulate retrieved data values. These functions are called **single-row functions**, because they return a single result for each row of data retrieved. The following subsections describe the Oracle10*g* SQL single-row functions for number, character, and date values.

Single-row Number Functions

Oracle10*g* SQL has several single-row number functions that you can use to manipulate retrieved data in ways that are more complex than simple arithmetic operations. For example, Oracle10*g* provides number functions for rounding data values or raising values to exponential powers. Table 3-7 summarizes some commonly used SQL single-row number functions.

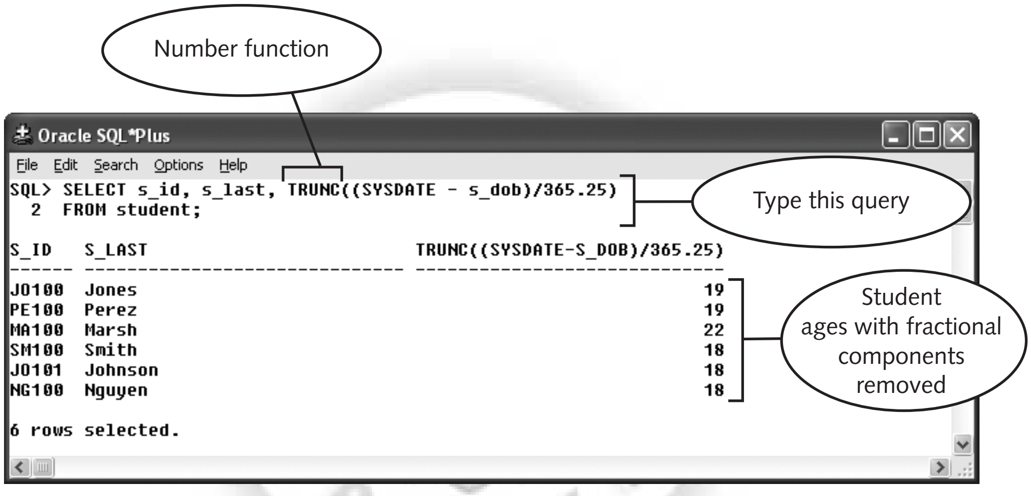
Function	Description	Example Query	Result
`ABS (`*`number`*`)`	Returns the absolute value of a number	`SELECT ABS(capacity) FROM location WHERE loc_id = 1;`	**`ABS(150) = 150`**
`CEIL (`*`number`*`)`	Returns the value of a number, rounded to the next highest integer	`SELECT CEIL (inv_price) FROM inventory WHERE inv_id = 1;`	**`CEIL(259.99) = 260`**
`FLOOR (`*`number`*`)`	Returns the value of a number, rounded down to the next integer	`SELECT FLOOR (inv_price) FROM inventory WHERE inv_id = 1;`	**`FLOOR(259.99) = 259`**
`MOD (`*`number`*`, `*`divisor`*`)`	Returns the remainder (modulus) for a number and its divisor	`SELECT MOD (inv_qoh, 10) FROM inventory WHERE inv_id = 1;`	**`MOD(16, 10) = 6`**
`POWER (`*`number`*`, `*`power`*`)`	Returns the value representing a number raised to the specified power	`SELECT POWER (inv_qoh, 2) FROM inventory WHERE inv_id = 2;`	**`POWER(12, 2) = 144`**
`ROUND (`*`number`*`, `*`precision`*`)`	Returns a number, rounded to a specified precision	`SELECT ROUND (inv_price, 0) FROM inventory WHERE inv_id = 1;`	**`ROUND(259.99, 0) = 260`**
`TRUNC (`*`number`*`, `*`precision`*`)`	Removes all digits from a number beyond the specified precision	`SELECT TRUNC (inv_price, 1) FROM inventory WHERE inv_id = 1;`	**`TRUNC (259.99, 1) = 259.9`**

Table 3-7 Oracle10*g* SQL single-row number functions

To use a SQL single-row number function, you list the function name in the SELECT clause, followed by the required parameter (or parameters) in parentheses. The next query demonstrates how to use a number function with a calculated database value. In Figure 3-27, you calculated each student's age based on his or her date of birth. The student ages currently appear in years, along with a fraction that represents the time since the student's last birthday. Usually, ages appear as whole numbers. You use the TRUNC function to truncate the fraction portion of the calculated student ages.

To use the TRUNC function to remove the fractional component:

1. Type the query in Figure 3-29.


Number function
Oracle SQL*Plus
File Edit Search Options Help
Type this query
Student ages with fractional components removed

```
SQL> SELECT s_id, s_last, TRUNC((SYSDATE - s_dob)/365.25)
  2  FROM student;

S_ID   S_LAST                         TRUNC((SYSDATE-S_DOB)/365.25)
------ ------------------------------ -----------------------------
JO100  Jones                                                     19
PE100  Perez                                                     19
MA100  Marsh                                                     22
SM100  Smith                                                     18
JO101  Johnson                                                   18
NG100  Nguyen                                                    18

6 rows selected.
```

Figure 3-29 Using a SQL number function

2. Execute the query. The query output shows the student ages in years without fractional values. (Your values will be different, based on your current system date.)

Single-row Character Functions

Oracle10*g* SQL also provides single-row character functions that you can use to format character output. Table 3-8 summarizes these character functions.

Function	Description	Example Query	String Used in Function	Function Result
`CONCAT` (*string1, string2*)	Concatenates (joins) two strings	`SELECT CONCAT (f_last, f_rank) FROM faculty WHERE f_id = 1;`	'Marx' and 'Associate'	'MarxAssociate'

Table 3-8 Oracle10*g* SQL single-row character functions

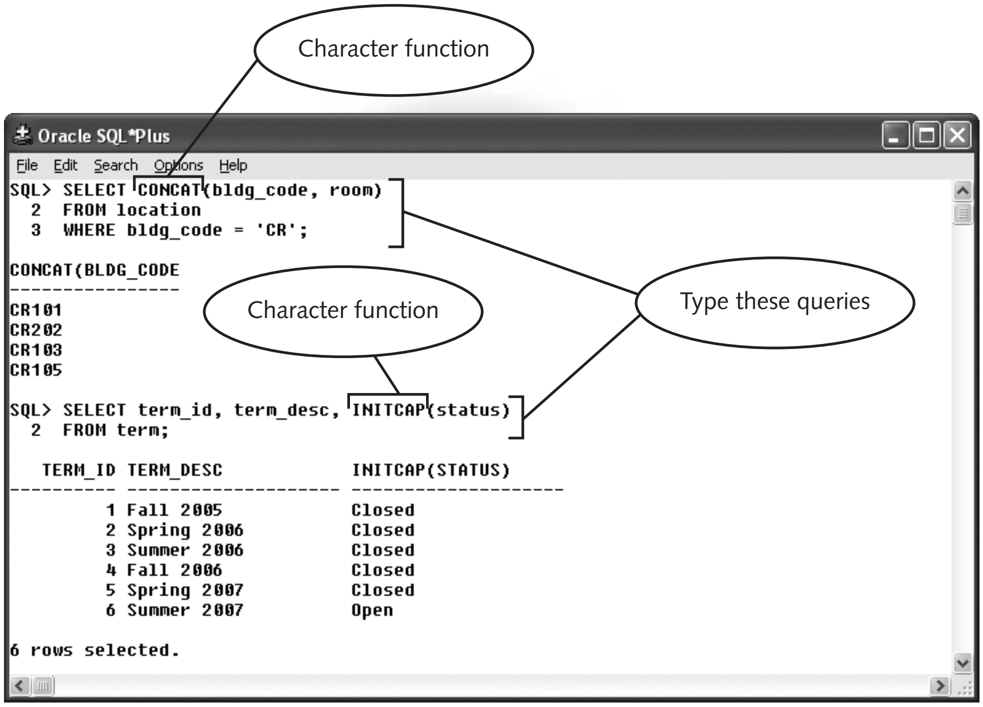
Function	Description	Example Query	String Used in Function	Function Result
`INITCAP` (*string*)	Returns the string with only the first letter in uppercase text	`SELECT INITCAP (bldg_code) FROM location WHERE loc_id = 1;`	'CR'	'Cr'
`LENGTH` (*string*)	Returns an integer representing the string length	`SELECT LENGTH (term_desc) FROM term WHERE term_id = 1;`	'Fall 2005'	9
`LPAD|RPAD` (*string, number_of_ characters_ to_add, padding_ character*);	Returns the value of the string with a specified number of padding characters added to the left/right	`SELECT LPAD (term_desc, 12, '*'), RPAD (term_desc, 12, '*') FROM term WHERE term_id = 1;`	'Fall 2005'	'***Fall 2005' 'Fall 2005***'
`LTRIM|RTRIM` (*string, search_string*)	Returns the string with all occurrences of the search string trimmed on the left/right side	`SELECT LTRIM (course_no, 'MIS') FROM course WHERE course_name LIKE '%Intro%';`	'MIS 101'	' 101'
`REPLACE` (*string, search_string, replacement_ string*)	Returns the string with every occurrence of the search string replaced with the replacement string	`SELECT REPLACE (term_desc, '200', '199') FROM term WHERE term_id = 1;`	'Fall 2005'	'Fall 2005'
`SUBSTR` (*string, start_position, length*)	Returns a string, starting at the start position, and of the specified length	`SELECT SUBSTR (term_desc, 1, 4) FROM term WHERE term_id = 1;`	'Fall 2005'	'Fall'
`UPPER|LOWER` (*string*)	Returns the string with all characters converted to uppercase or lowercase letters	`SELECT UPPER (term_desc) , LOWER (term_desc) FROM term WHERE term_id = 1;`	'Fall 2005'	'FALL 2005' 'fall 2005'

Table 3-8 Oracle10*g* SQL single-row character functions (continued)

Next, you create some queries that use the Oracle10*g* SQL character functions. First, you use the CONCAT function to display the Northwoods University location building codes and rooms as a single text string. Then, you use the INITCAP function to display the values in the STATUS column in the TERM table in mixed-case letters, with the first letter capitalized.

To create queries using the Oracle10*g* SQL character functions:

1. Type and execute the first query in Figure 3-30. The building codes and rooms appear as a single text string.



```
SQL> SELECT CONCAT(bldg_code, room)
  2  FROM location
  3  WHERE bldg_code = 'CR';

CONCAT(BLDG_CODE
----------------
CR101
CR202
CR103
CR105

SQL> SELECT term_id, term_desc, INITCAP(status)
  2  FROM term;

   TERM_ID TERM_DESC            INITCAP(STATUS)
---------- -------------------- --------------------
         1 Fall 2005            Closed
         2 Spring 2006          Closed
         3 Summer 2006          Closed
         4 Fall 2006            Closed
         5 Spring 2007          Closed
         6 Summer 2007          Open

6 rows selected.
```

Figure 3-30 Using SQL character functions

2. Type and execute the second query in Figure 3-30. The STATUS values appear in mixed-case letters, with the first letter capitalized. (Recall that the STATUS values are stored in the database in all capital letters.)

Single-row Date Functions

Earlier you learned how to use date arithmetic within SQL queries to display a date that occurs a specific number of days before or after a known date. Oracle10*g* SQL provides a variety of single-row date functions to support additional date operations. Table 3-9 summarizes these date functions.

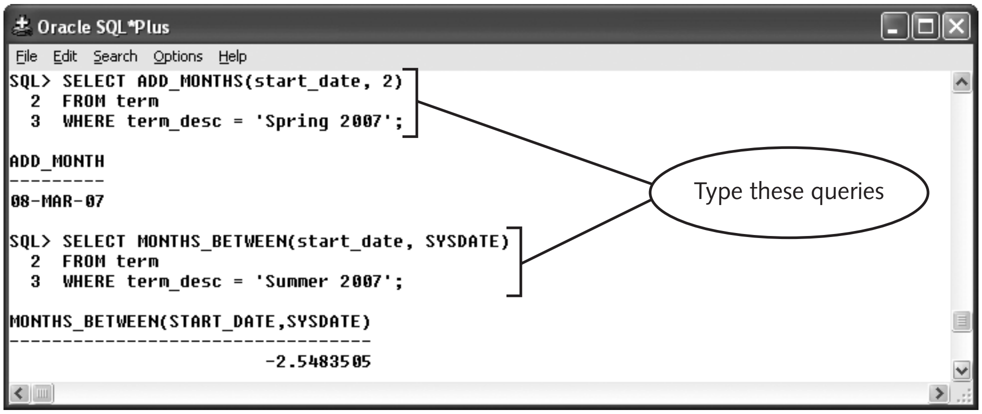
Function	Description	Example Query	Date(s) Used in Function	Function Result
`ADD_MONTHS (`*`date`*`, `*`months_ to_add`*`)`	Returns a date that is the specified number of months after the input date	`SELECT ADD_MONTHS (ship_date_ expected, 2) FROM shipment WHERE ship_id = 1;`	9/15/2006	11/15/2006
`LAST_DAY (`*`date`*`)`	Returns the date that is the last day of the month specified in the input date	`SELECT LAST_DAY (ship_date_ expected) FROM shipment WHERE ship_id = 1;`	9/15/2006	9/30/2006
`MONTHS_ BETWEEN (`*`date1`*`, `*`date2`*`)`	Returns the number of months, including decimal fractions, between two dates	`SELECT MONTHS_ BETWEEN (ship_ date_expected, TO_DATE('10-AUG- 2006', 'DD-MON- YYYY')) FROM shipment WHERE ship_id = 1;`	9/15/2006	1.1612903

Table 3-9 Oracle10*g* SQL single-row date functions

Now you execute some queries that use the Oracle10*g* SQL date functions. First, you use the ADD_MONTHS function to display a date that is two months after the date that the Spring 2007 term started. Then, you use the MONTHS_BETWEEN function to display the number of months between the current system date and the date on which the Summer 2007 term started.

To execute queries using the Oracle10*g* SQL date functions:

1. Type and execute the first query in Figure 3-31. The query output shows 08-MAR-07, which is two months after the retrieved date value.



```
SQL> SELECT ADD_MONTHS(start_date, 2)
  2  FROM term
  3  WHERE term_desc = 'Spring 2007';

ADD_MONTH
---------
08-MAR-07

SQL> SELECT MONTHS_BETWEEN(start_date, SYSDATE)
  2  FROM term
  3  WHERE term_desc = 'Summer 2007';

MONTHS_BETWEEN(START_DATE,SYSDATE)
----------------------------------
                        -2.5483505
```

Figure 3-31 Using SQL date functions

2. Type and execute the second query in Figure 3-31. The output shows the number of months between the current system date and the term's starting date. In this example, the term started approximately two and a half months before the computer's current system date. (Your value will be different, based on your current system date.)

ORACLE10*g* SQL GROUP FUNCTIONS

So far, you have displayed and manipulated individual database rows. However, database applications often need to display information about a group of rows. For example, a query might display the total number of students who enroll in a specific Northwoods University course section, or the total revenue generated from all orders that customers place using the Clearwater Traders Web site. To display data that summarizes multiple rows, you use one of the Oracle10*g* group functions. An Oracle10*g* SQL **group function** performs an operation on a group of queried rows and returns a single result, such as a column sum. Table 3-10 describes commonly used Oracle SQL group functions.

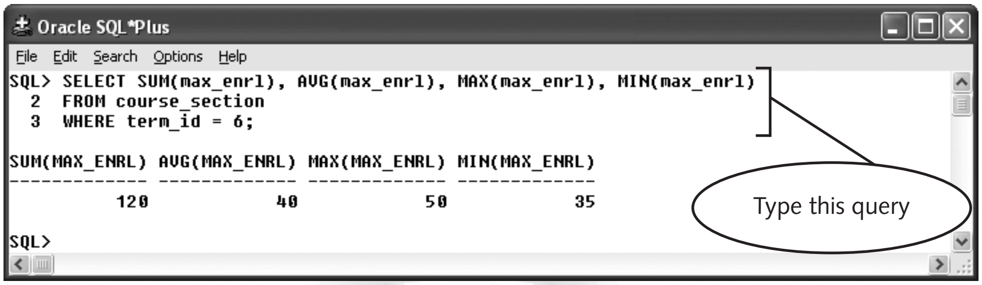
Function	Description	Example Query	Result
`AVG (`*`fieldname`*`)`	Returns the average value of a numeric field's returned values	`SELECT AVG(capacity) FROM location;`	33.30769
`COUNT(*)`	Returns an integer representing a count of the number of returned rows	`SELECT COUNT(*) FROM enrollment;`	20
`COUNT (`*`fieldname`*`)`	Returns an integer representing a count of the number of returned rows for which the value of *fieldname* is NOT NULL	`SELECT COUNT(grade) FROM enrollment;`	12
`MAX (`*`fieldname`*`)`	Returns the maximum value of a numeric field's returned values	`SELECT MAX(max_enrl) FROM course_section;`	140
`MIN (`*`fieldname`*`)`	Returns the minimum value of a numeric field's returned values	`SELECT MIN(max_enrl) FROM course_section;`	30
`SUM (`*`fieldname`*`)`	Sums a numeric field's returned values	`SELECT SUM(capacity) FROM location;`	432

Table 3-10 Oracle10*g* SQL group functions

To use a group function in a SQL query, you list the function name, followed by the column name on which to perform the calculation, in parentheses. In the following steps, you execute a query that uses group functions to sum the maximum enrollment for each course section in the Summer 2007 term and calculate the average, maximum, and minimum current enrollments.

To use group functions in a query:

1. Type the query in Figure 3-32 to sum the maximum enrollment for all course sections and calculate the average, maximum, and minimum current enrollment for each course section for the Summer 2007 term (TERM_ID = 6).



```
SQL> SELECT SUM(max_enrl), AVG(max_enrl), MAX(max_enrl), MIN(max_enrl)
  2  FROM course_section
  3  WHERE term_id = 6;

SUM(MAX_ENRL) AVG(MAX_ENRL) MAX(MAX_ENRL) MIN(MAX_ENRL)
------------- ------------- ------------- -------------
          120            40            50            35

SQL>
```

Figure 3-32 Using group functions in a SQL query

2. Execute the query. The output shows the summary data for the Summer 2007 term.

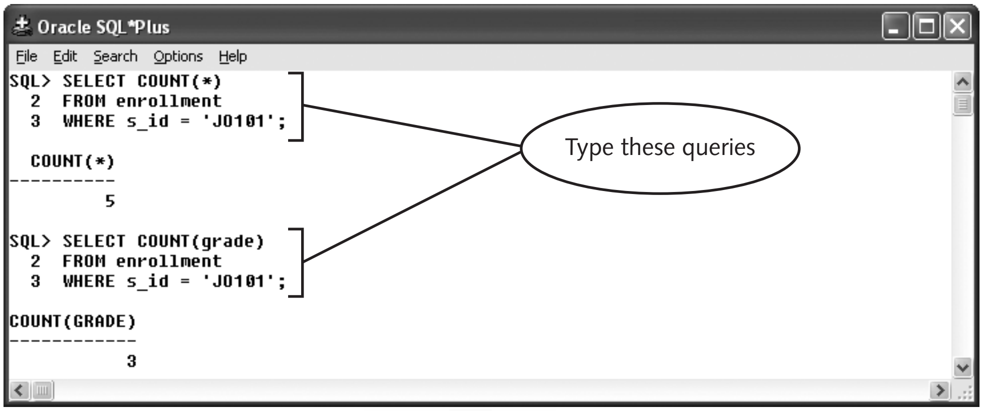
The following section describes how to use the COUNT group function, which works differently than the other group functions. You also learn how to use the GROUP BY clause to group similar data in group functions, and how to use the HAVING clause to add search conditions to group functions.

Using the COUNT Group Function

The COUNT group function returns an integer that represents the number of rows that a query returns. The COUNT(*) version of this function calculates the total number of rows in a table that satisfy a given search condition. The COUNT(*) function is the only group function in Table 3-10 that includes NULL values. The other functions ignore NULL values. The COUNT(*columnname*) version calculates the number of rows in a table that satisfy a given search condition and also contain a non-null value for the given column. Next, you use both versions of the COUNT function to count the total number of courses in which student Lisa Johnson (S_ID = JO101) has enrolled and the total number of courses in which Johnson has received a grade (GRADE is NOT NULL).

To use the COUNT group function:

1. Type and execute the first query in Figure 3-33, which uses the `COUNT(*)` group function to count the total number of courses in which student Lisa Johnson (S_ID = JO101) has enrolled.



```
SQL> SELECT COUNT(*)
  2  FROM enrollment
  3  WHERE s_id = 'JO101';

  COUNT(*)
----------
         5

SQL> SELECT COUNT(grade)
  2  FROM enrollment
  3  WHERE s_id = 'JO101';

COUNT(GRADE)
------------
           3
```

Figure 3-33 Using the COUNT group function

2. Type and execute the second query in Figure 3-33, which uses the `COUNT(columnname)` function to count the total number of courses for which student Lisa Johnson has received a grade.

Notice the difference between the output for the first and second queries. Lisa has enrolled in a total of five courses, and has been assigned a GRADE value in only three of those courses.

Using the GROUP BY Clause to Group Data

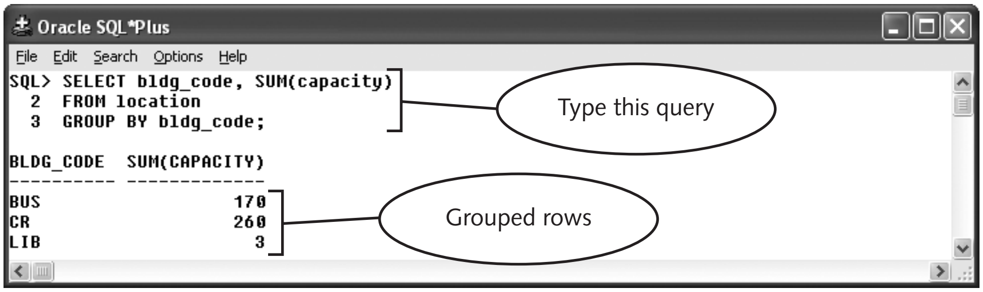
If a query retrieves multiple rows and the rows in one of the retrieved columns have duplicate values, you can group the output by the column with duplicate values and apply group functions to the grouped data. For example, you might want to retrieve the names of the different building codes at Northwoods University and calculate the sum of the capacity of each building. To do this, you add the GROUP BY clause after the query's FROM clause. The GROUP BY clause has the following syntax:

```
GROUP BY group_columnname;
```

Now you create a query that uses a group function with the GROUP BY clause to list the Northwoods University building codes and sum each building's capacity.

To use the GROUP BY clause to group rows:

1. Type the query in Figure 3-34 to list the building code and the total capacity of each building in the LOCATION table.
2. Press **Enter** to execute the query. The output lists the building codes and their associated capacities.



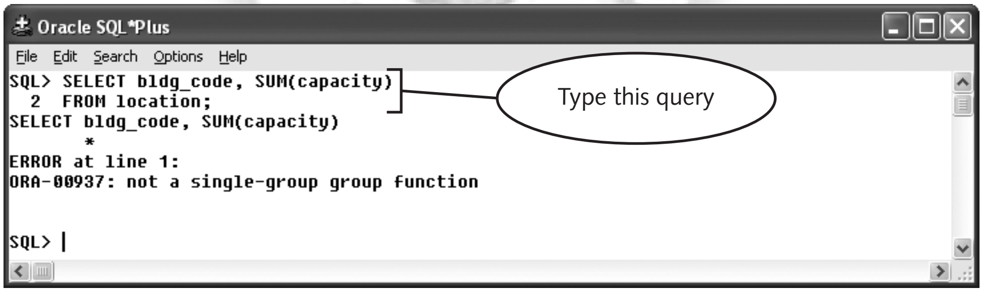
```
SQL> SELECT bldg_code, SUM(capacity)
  2  FROM location
  3  GROUP BY bldg_code;

BLDG_CODE  SUM(CAPACITY)
---------- -------------
BUS                  170
CR                   260
LIB                    3
```

Figure 3-34 SQL query that uses the GROUP BY clause to group rows

When you create a query containing a GROUP BY clause, all columns listed in the SELECT clause must be included in the GROUP BY clause. In other words, if ungrouped columns are included in the SELECT clause, Oracle10*g* will return an error message. This error occurs because SQL cannot display single-row results and group function results in the same query output. In the next query, you view the error that occurs when you attempt to mix single rows and grouped rows in the same query. You repeat the query to return building codes and the sums of their capacities, but omit the GROUP BY clause.

To repeat the query without the GROUP BY clause:

1. Type the query in Figure 3-35 to list the building code name and the total capacity of each building in the LOCATION table, omitting the GROUP BY clause.



```
SQL> SELECT bldg_code, SUM(capacity)
  2  FROM location;
SELECT bldg_code, SUM(capacity)
       *
ERROR at line 1:
ORA-00937: not a single-group group function


SQL> |
```

Figure 3-35 Error that occurs when you omit the GROUP BY clause

2. Execute the query. The error message "ORA-00937: not a single-group group function" appears.

This error message indicates that a SELECT clause cannot contain both a group function and an individual column expression unless the individual column expression is in a GROUP BY clause. To solve this problem, you must include BLDG_CODE in the GROUP BY clause, as shown in the query in Figure 3-34.

Using the HAVING Clause to Filter Grouped Data

You can use the HAVING clause to place a search condition on the results of queries that display group function calculations. The HAVING clause has the following syntax:

```
HAVING group_function comparison_operator value
```

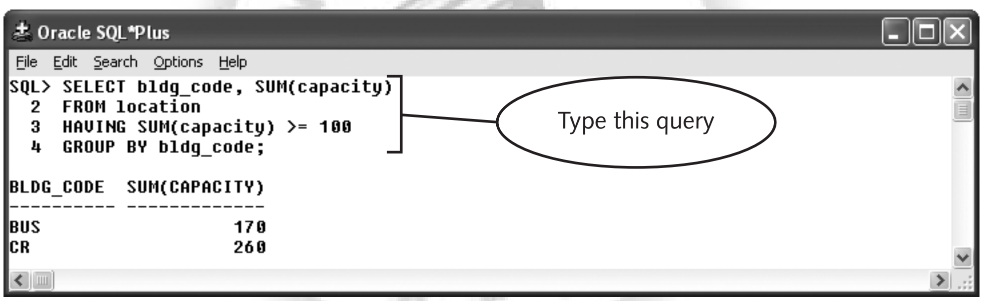
In this syntax, *group_function* is the group function expression, *comparison_operator* is one of the comparison operators in Table 3-3, and *value* is the value that the search condition matches. For example, suppose you want to retrieve the total capacity of each building at Northwoods University, but you are not interested in the data for buildings that have a capacity of less than 100. You use the following HAVING clause to filter the output:

```
HAVING SUM(capacity) >= 100
```

Next, you execute the query to retrieve this data.

To filter grouped data using the HAVING clause:

1. Type the query in Figure 3-36 to retrieve building codes and total capacities where the total capacity is greater than or equal to 100.



```
SQL> SELECT bldg_code, SUM(capacity)
  2  FROM location
  3  HAVING SUM(capacity) >= 100
  4  GROUP BY bldg_code;

BLDG_CODE  SUM(CAPACITY)
---------- -------------
BUS                  170
CR                   260
```

Figure 3-36 Using the HAVING clause with a group function

2. Execute the query. Compare the query output with the output in Figure 3-34, and note that the output omits the LIB building code because the sum of this building's capacity does not meet the search criteria in the HAVING clause.

FORMATTING OUTPUT IN SQL*PLUS

So far, you have accepted the default output formats in SQL*Plus—output column headings are the same as the database column names. You have also accepted the default screen widths and lengths. Now you learn how to modify the format of output data. You learn how to change the column headings, the SQL*Plus display width and length, and the format of retrieved data values.

Creating Alternate Column Headings

In SQL*Plus query output, column headings for retrieved columns are the names of the database table columns. When you create retrieve values that perform arithmetic calculations, or that use single-row functions or group functions, the output headings appear as the formula or function. For example, in Figure 3-36, the heading for the values that calculate the sum of the capacity for each building appears as SUM(CAPACITY). Similarly, when you calculated the student ages (see Figure 3-29), the heading appeared as the full formula for the calculation. To display a different heading in the query output, you can create alternate output heading text, or you can create an alias.

Alternate Output Heading Text

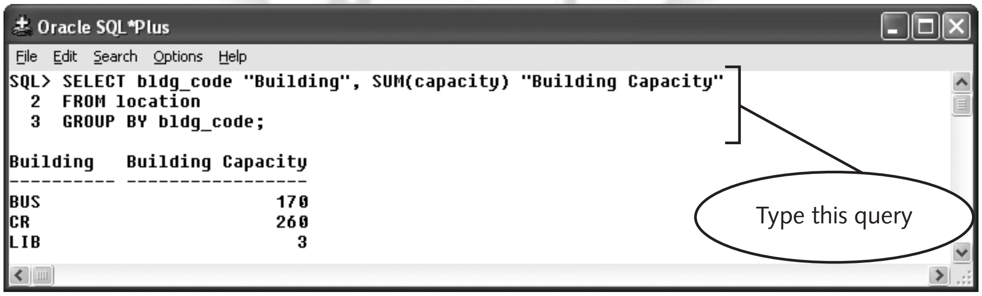
To specify alternate output heading text, you use the following syntax in the SELECT clause:

```
SELECT columnname1 "heading1_text",
columnname2 "heading2_text", ...
```

In this syntax, *heading1_text* specifies the alternate heading that appears for *columnname1*, *heading2_text* specifies the alternate heading that appears for *columnname2*, and so forth. You enclose the alternate heading text in double quotation marks, and it can contain characters, including blank spaces. In the following steps, you create a query that specifies alternate output headings.

To specify alternate output headings:

1. Type the query in Figure 3-37 to create the alternate output headings for the query columns.



```
SQL> SELECT bldg_code "Building", SUM(capacity) "Building Capacity"
  2  FROM location
  3  GROUP BY bldg_code;

Building   Building Capacity
---------- -----------------
BUS                      170
CR                       260
LIB                        3
```

Figure 3-37 Creating alternate output headings

2. Execute the query. The output appears with the alternate headings.

Changing the output heading changes only the SQL*Plus output display—you cannot use the alternate heading to reference the column in a GROUP BY or ORDER BY clause in a query. In the query in Figure 3-37, if you wanted to order the query output by the building capacity value, which is stored in the CAPACITY column, you need to specify the ORDER BY clause as `ORDER BY SUM(capacity);`. You could not order the query output by specifying the heading "Building Capacity" in the ORDER BY clause.

Aliases

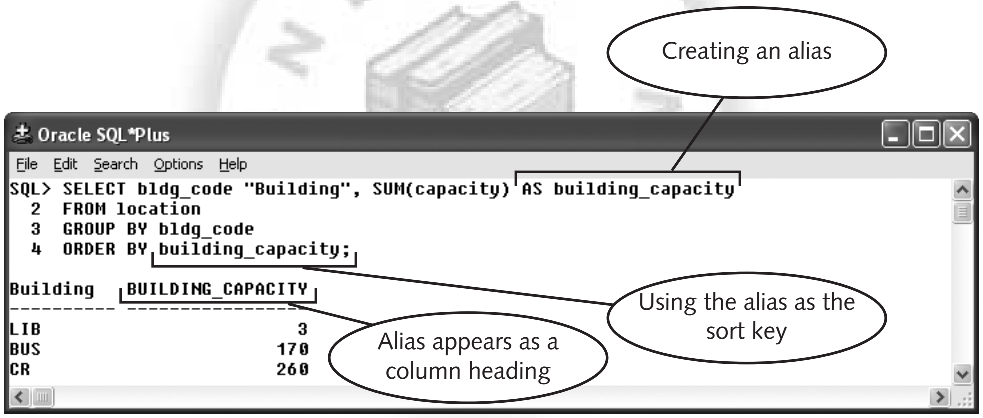
An **alias** is an alternate name for a query column. After you create an alias, you can reference the alias in other parts of the query, such as in the GROUP BY or ORDER BY clause. The general syntax for creating an alias is:

```
SELECT columnname1 AS alias_name1...
```

The *alias_name1* value must follow the Oracle naming standard and cannot contain blank spaces. You now repeat the query to sum the capacity of all of the buildings but modify it to create an alias for the summed capacity values. Then, you use the alias as the sort key to specify the sort order of the output.

To create an alias:

1. Type the query in Figure 3-38.


Creating an alias
Oracle SQL*Plus
File Edit Search Options Help
Using the alias as the sort key
Alias appears as a column heading

```
SQL> SELECT bldg_code "Building", SUM(capacity) AS building_capacity
  2  FROM location
  3  GROUP BY bldg_code
  4  ORDER BY building_capacity;

Building   BUILDING_CAPACITY
---------- -----------------
LIB                        3
BUS                      170
CR                       260
```

Figure 3-38 Creating an alias

2. Execute the query. The alias appears as the new output heading, and the output values appear sorted by capacity.

TIP

Note that the query uses the alias as the sort key. Also note that the output heading ("BUILDING_CAPACITY") appears in all uppercase letters. The DBMS automatically converts alias names to uppercase characters.

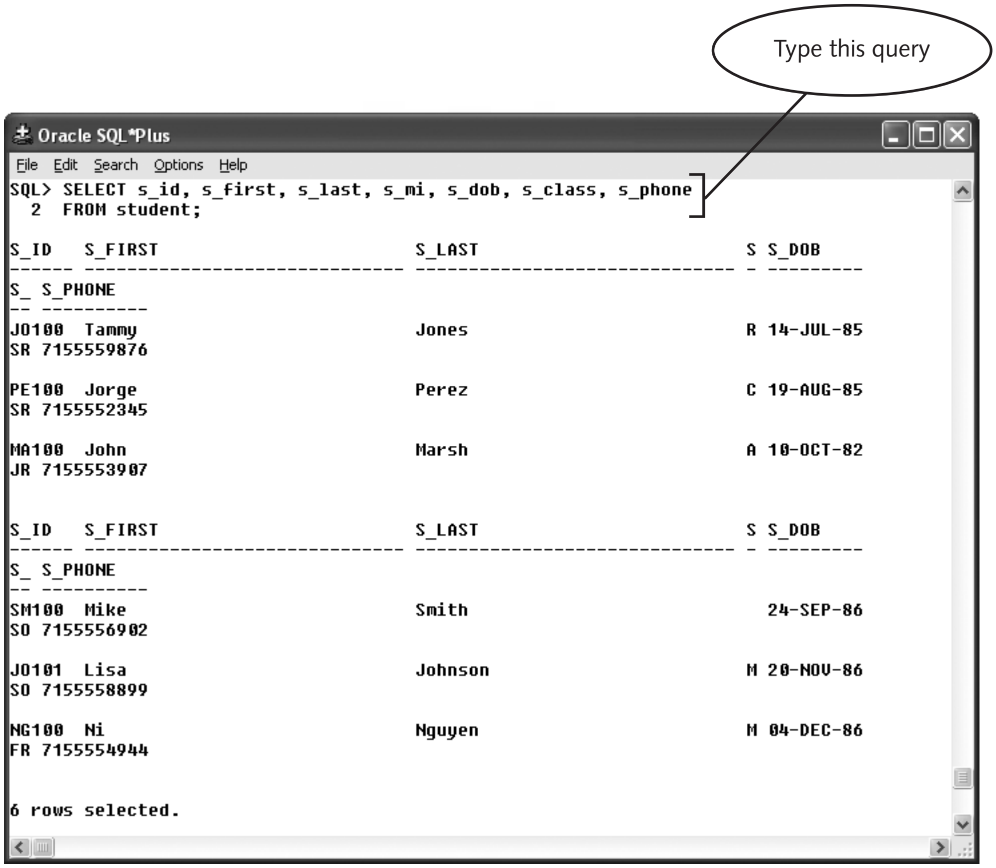
Modifying the SQL*Plus Display Environment

Recall that SQL*Plus displays query output one page at a time. A SQL*Plus page consists of a specific number of characters per line and a specific number of lines per page.

If the query output contains more characters than the current SQL*Plus environment's page width, then the output wraps to the next line. If the query output contains more lines than the current SQL*Plus page length, then the column headings repeat at the top of each page. Next, you type a query that spans multiple SQL*Plus lines and pages, and examine the output.

To type a query that spans multiple lines and pages:

1. Type the query in Figure 3-39.



```
SQL> SELECT s_id, s_first, s_last, s_mi, s_dob, s_class, s_phone
  2  FROM student;

S_ID   S_FIRST                        S_LAST                         S S_DOB
------ ------------------------------ ------------------------------ - ---------
S_ S_PHONE
-- ----------
JO100  Tammy                          Jones                          R 14-JUL-85
SR 7155559876

PE100  Jorge                          Perez                          C 19-AUG-85
SR 7155552345

MA100  John                           Marsh                          A 10-OCT-82
JR 7155553907


S_ID   S_FIRST                        S_LAST                         S S_DOB
------ ------------------------------ ------------------------------ - ---------
S_ S_PHONE
-- ----------
SM100  Mike                           Smith                            24-SEP-86
SO 7155556902

JO101  Lisa                           Johnson                        M 20-NOV-86
SO 7155558899

NG100  Ni                             Nguyen                         M 04-DEC-86
FR 7155554944


6 rows selected.
```

Figure 3-39 Query output that spans multiple lines and pages

2. Execute the query. The query output should look like Figure 3-39, spanning multiple lines and pages.

HELP

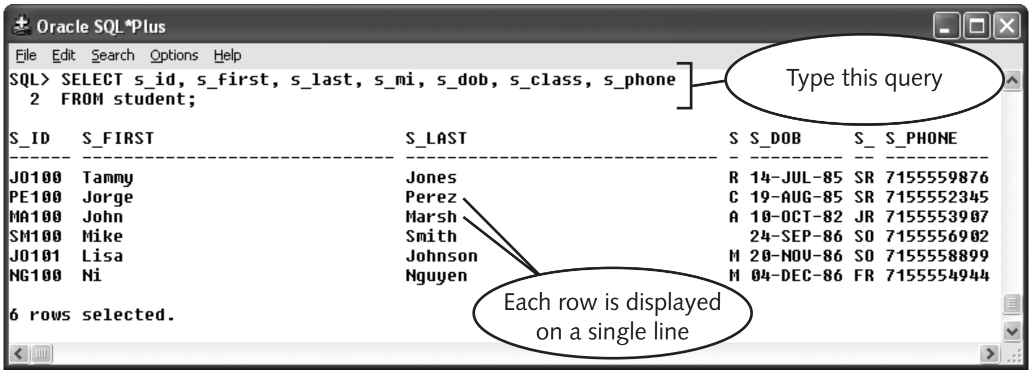
If your output does not look like Figure 3-39, it is because your SQL*Plus display environment settings have been changed from the default settings. You adjust your settings in the next set of steps.

The output spans multiple lines, and the data values for the S_CLASS and S_PHONE columns wrap to the next line. The output spans multiple pages, and the column headings repeat after the values for the first three rows. You can configure the SQL*Plus page and line size by clicking Options on the menu bar, clicking Environment, and then configuring the environment in the Environment dialog box. SQL*Plus saves these changes on your workstation, and the configuration values remain the same until you or someone else change them again.

The SQL*Plus environment **linesize** property specifies how many characters appear on a display line, and the **pagesize** property specifies how many lines appear on a SQL*Plus page. Now you modify your SQL*Plus display settings, and make the line and page sizes larger.

To modify the SQL*Plus display settings:

1. Click **Options** on the menu bar, and then click **Environment**. The Environment dialog box opens.
2. Select **linesize** in the Set Options list. Select the **Custom** option button, and type **120** in the Value column to display 120 characters per line.
3. Select **pagesize** in the Set Options list. Select the **Custom** option button, and then type **40** in the Value column to display 40 lines per page.
4. Click **OK** to save your settings.
5. Type the query in Figure 3-40 to test your new settings. The query output should appear as shown, with each row's value on the same line, and all of the values on the same display page.


Oracle SQL*Plus
File Edit Search Options Help
Type this query
Each row is displayed on a single line

```
SQL> SELECT s_id, s_first, s_last, s_mi, s_dob, s_class, s_phone
  2  FROM student;

S_ID   S_FIRST                        S_LAST                         S S_DOB     S_ S_PHONE
------ ------------------------------ ------------------------------ - --------- -- ----------
JO100  Tammy                          Jones                          R 14-JUL-85 SR 7155559876
PE100  Jorge                          Perez                          C 19-AUG-85 SR 7155552345
MA100  John                           Marsh                          A 10-OCT-82 JR 7155553907
SM100  Mike                           Smith                            24-SEP-86 SO 7155556902
JO101  Lisa                           Johnson                        M 20-NOV-86 SO 7155558899
NG100  Ni                             Nguyen                         M 04-DEC-86 FR 7155554944

6 rows selected.
```

Figure 3-40 Query output with modified display settings

Formatting Data Using Format Models

When you retrieve data from NUMBER or DATE data columns, the data values appear using the SQL*Plus default date and number formats. Sometimes, you might want to display number columns using a currency format model that shows a currency symbol, or you might want to display DATE columns using an alternate format model. To use an alternate format model, you can use the TO_CHAR function to convert the column to a character string, and then apply the desired format model to the value.

The TO_CHAR function has the following syntax:

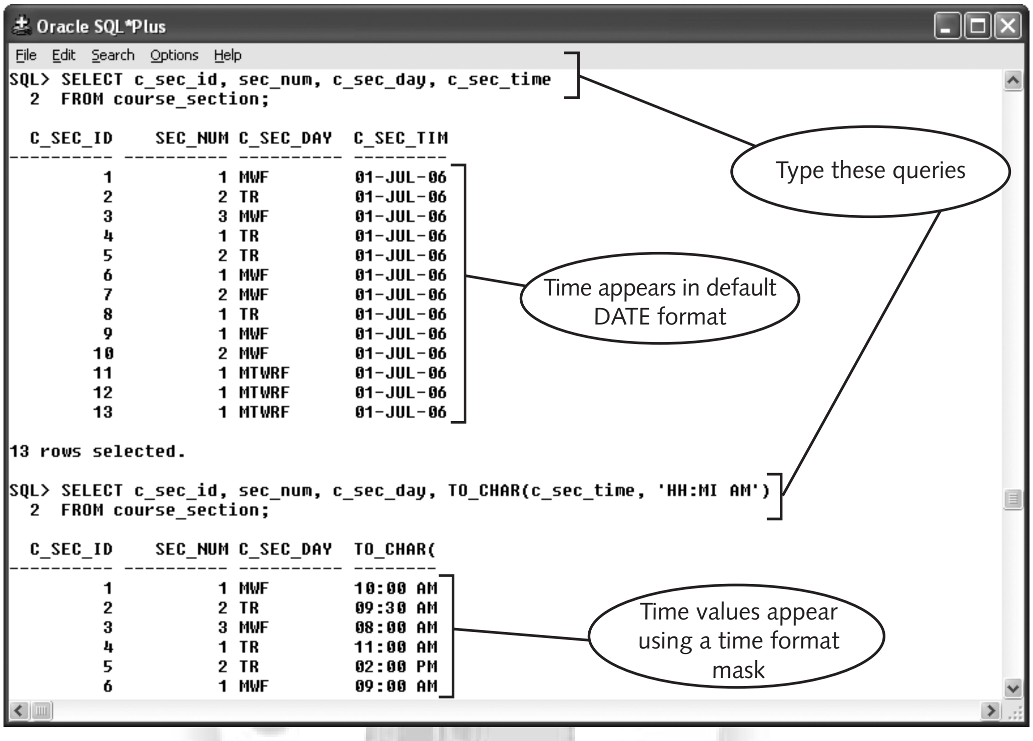
```
TO_CHAR(column_name, 'format_model')
```

In this syntax, *column_name* is the column value you wish to format, and *format_model* is the format model you wish to apply to the column value. Note that the format model appears in single quotation marks.

It is necessary to convert output columns to characters and apply a specific format model when you store time values in a column that has the DATE data type. Recall that DATE data columns store time as well as date information; however, the default DATE output format model is DD-MON-YY, which does not have a time component. Therefore, you need to format time output using an alternate format model. In the next set of steps, you retrieve the values from the C_SEC_TIME column in the COURSE_SECTION table to examine the output. When the Ch3Northwoods.sql script originally inserts the values for the C_SEC_TIME column into the COURSE_SECTION table, it specifies the time values in the default format. However, the script does not specify a date value, so the DBMS inserts the default date value, which is the first day of the current month. You realize that you first need to retrieve the values, then convert them to characters using the TO_CHAR function, and finally display the values using a time format model.

To examine and format the C_SEC_TIME column values:

1. Type and execute the first query in Figure 3-41 to retrieve the C_SEC_TIME column from the COURSE_SECTION table. Note that the values appear in the default DATE format and do not display the time values. (Your values will be different, because you ran your script during a different month.)
2. Type and execute the second query in Figure 3-41 to convert the course section times to characters, then display the values using a format model that has a time (rather than a date) component.


Oracle SQL*Plus
File Edit Search Options Help
Type these queries
Time appears in default DATE format
Time values appear using a time format mask

```
SQL> SELECT c_sec_id, sec_num, c_sec_day, c_sec_time
  2  FROM course_section;

  C_SEC_ID    SEC_NUM C_SEC_DAY  C_SEC_TIM
---------- ---------- ---------- ---------
         1          1 MWF        01-JUL-06
         2          2 TR         01-JUL-06
         3          3 MWF        01-JUL-06
         4          1 TR         01-JUL-06
         5          2 TR         01-JUL-06
         6          1 MWF        01-JUL-06
         7          2 MWF        01-JUL-06
         8          1 TR         01-JUL-06
         9          1 MWF        01-JUL-06
        10          2 MWF        01-JUL-06
        11          1 MTWRF      01-JUL-06
        12          1 MTWRF      01-JUL-06
        13          1 MTWRF      01-JUL-06

13 rows selected.

SQL> SELECT c_sec_id, sec_num, c_sec_day, TO_CHAR(c_sec_time, 'HH:MI AM')
  2  FROM course_section;

  C_SEC_ID    SEC_NUM C_SEC_DAY  TO_CHAR(
---------- ---------- ---------- --------
         1          1 MWF        10:00 AM
         2          2 TR         09:30 AM
         3          3 MWF        08:00 AM
         4          1 TR         11:00 AM
         5          2 TR         02:00 PM
         6          1 MWF        09:00 AM
```

Figure 3-41 Displaying time values using the TO_CHAR function (partial output shown)

You can also use the TO_CHAR function to apply format models to NUMBER columns. For example, the credit hours multiplied by the tuition rate for a course at Northwoods University represents a currency value. Next, you create a query that formats output values using a currency format model.

To create a query that formats number values as currency:

1. Type and execute the following command to format the INV_PRICE values using a currency format model:

   ```
   SELECT course_name, TO_CHAR(credits*86.95, '$99,999.99')
   FROM course
   WHERE course_no = 'MIS 101';
   ```

2. Close SQL*Plus. Switch to Notepad, save Ch3BQueries.sql, then close Notepad and all other open applications.

Summary

- In a single-table SELECT query, the SELECT clause lists the columns that you want to display in your query, the FROM clause lists the name of the table involved in the query, and the WHERE clause specifies a search condition.
- If you are querying a table that you did not create, you must preface the table name with the owner's username, and the owner must have given you the privilege to retrieve data from that table.
- To retrieve every row in a table, the data values do not need to satisfy a search condition, so you omit the WHERE clause.
- If you want to retrieve all of the columns in a table, you can use an asterisk (`*`) in place of the individual column names in the SELECT clause.
- You can use the DISTINCT qualifier to suppress duplicate rows in query output, and you can sort query outputs using the ORDER BY clause.
- To retrieve specific rows, you can use exact search conditions, which exactly match a value, or inexact search conditions, which match a range of values.
- To retrieve rows in which the value of a particular column value is NULL, you use the search condition `WHERE columnname IS NULL`. To retrieve rows in which the value of a particular column is NOT NULL, you use the search condition `WHERE columnname IS NOT NULL`.
- You use the IN comparison operator to match data values that are members of a set of search values, and you use the NOT IN comparison operator to match data values that are not members of a set of search values.
- You use the LIKE comparison operator to retrieve values that match partial text strings. To use the LIKE operator, you specify the search string using the percent (`%`) wildcard character to specify multiple characters and the underscore (`_`) wildcard character to specify a single character.
- You can create SQL queries that perform addition, subtraction, multiplication, and division operations on retrieved data values.
- The SYSDATE pseudocolumn returns the current system date and time. You can create queries that use date arithmetic to add or subtract a specific number of days from a known date or determine the number of days between two known dates. You can perform calculations using interval data values by adding or subtracting an interval to a known date. You can also calculate the sum or difference of two interval values.
- Oracle10*g* SQL provides single-row number, character, and date functions that allow you to manipulate retrieved values for each row of data that a query retrieves.
- Oracle10*g* SQL provides group functions to calculate the sum, average, or maximum or minimum value of a group of rows that a query retrieves. Oracle10*g* SQL

also provides a function to count the number of rows that a query retrieves. You can use the GROUP BY clause to group the output on a specific output value, and you can use the HAVING clause to add a search condition to a query that contains a group function.

- By default, SQL*Plus displays query output using database column names or formulas for queries that perform arithmetic calculations. You can create alternate output headings by specifying the alternate heading text for each output column. Or, you can create an alias, which is an alternate name for the column that can be used in the GROUP BY or ORDER BY clause.
- You can change the appearance of the SQL*Plus environment by changing the pagesize and linesize properties. The linesize property specifies the number of characters that appear on the SQL*Plus screen, and the pagesize property specifies the number of lines that appear on the screen.
- To format values stored in DATE and NUMBER columns in a SQL query, you can convert the column to a character column using the TO_CHAR function and format the column using a format model.

REVIEW QUESTIONS

1. The ________________ wildcard character is used to specify exactly one character.
2. The IN operator is used to identify that the search condition includes a pattern. True or False?
3. Which clause is used to restrict the number of rows returned by a query that is based on a stated condition?
4. The ________________ keyword is included in the query to suppress duplicate results.
5. The COUNT(*) function can be used to include NULL values in its results. True or False?
6. The ________________ clause is used to restrict the groups that are included in the output of a query.
7. Which logical operator requires that both conditions be met for a row to be included in the results?
8. Which function is used to format the display of a date value in the results of a query?
9. The ________________ keyword is used to denote a column alias.
10. Which SQL*Plus property determines how many lines are displayed for column headings that are repeated in the results?

Multiple choice

1. Which of the following clauses present query results in a sorted order?
   a. GROUP BY
   b. ORDERED BY
   c. SORT BY
   d. none of the above
2. The ________________ wildcard character is used to represent any number of characters.
   a. *
   b. %
   c. _
   d. ^
3. Based on its syntax, which of the following is a valid query?
   a. `SELECT acolumn FROM atable WHERE acolumn LIKE NULL;`
   b. `SELECT acolumn FROM atable WHERE acolumn = NULL;`
   c. `SELECT acolumn FROM atable WHERE acolumn IS NOT NULL;`
   d. `SELECT acolumn FROM atable WHERE acolumn <> NULL;`
4. Which function is used to determine the total credit hours currently being taken by a student?
   a. TOTAL
   b. SUM
   c. CALC
   d. ADD
5. Which function is used to determine how many professors currently hold the rank of assistant professor?
   a. TOTAL
   b. ADD
   c. SUM
   d. COUNT
6. Which keyword is used to retrieve the computer's current date and time?
   a. DATE
   b. SYSTEMTIME
   c. TIME
   d. none of the above

7. Which of the following is a single-row function?
   a. COUNT
   b. SUM
   c. INITIALCAP
   d. none of the above
8. When searching for nonnumeric data, the search condition must be enclosed in:
   a. double-quotation marks (")
   b. single-quotation marks (')
   c. parentheses ( )
   d. none of the above
9. If you use the character string Faculty as a column alias, the string must be enclosed in ________________ or it is converted to uppercase characters in the results.
   a. double-quotation marks (")
   b. single-quotation marks (')
   c. parentheses ( )
   d. none of the above
10. By default, Oracle10*g* sorts data in ________________.
    a. ascending order
    b. descending order
    c. natural order
    d. the order in which it is stored in the database table

Problem-solving Cases

For all cases, use Notepad or another text editor to write a script using the specified filename. Always use the search condition text exactly as the case specifies. Place the queries in the order listed, and save the script files in your Chapter03\Cases folder on your Data Disk. If you haven't done so already, run the Ch3Clearwater.sql script in the Chapter03 folder on your Data Disk to create and populate the case study database tables. All the following cases are based on the Clearwater Traders database.

1. Determine the inventory price of inventory item (INV_ID) #1.
2. Determine which customers were born in the 1970s.
3. Determine how many different categories of inventory are carried by Clearwater Traders.
4. Determine how many shipments have not yet been received.

5. Calculate the total quantity on hand for each inventory item in the INVENTORY table—ignore their different sizes and colors.
6. Determine the total number of orders received on May 29, 2006.
7. Determine how many orders placed on May 31, 2006, were paid by credit card.
8. Create a list of all Florida and Georgia customers.
9. List all inventory items that do not have an associated size.
10. Identify which shipments are expected by September 1, 2006.
11. Determine how many inventory items have an inventory price greater than $60.00 and are available in a size of L or XL.
12. In the INVENTORY table, ITEM_ID 5 is available in how many different colors?
13. In the INVENTORY table, ITEM_ID 5 is available in how many different sizes?
14. Determine the current age of each customer. Display the customer's first and last names and their age in years.
15. Display the SHIP_DATE_EXPECTED of each shipment in the SHIPMENT table using the format MONTH DD,YYYY.
16. Determine how many orders were received from each of the available order sources (OS_ID).
17. Identify which items in stock (that is INV_QOH > 0) are available in sizes Medium or Large and are available in the colors Royal, Bright Pink, or Spruce.
18. Create a listing that identifies the different items (ITEM_ID) in the INVENTORY table and the number of colors available for each item.
19. Determine how many items are not in categories 2 or 4.
20. List the unique ITEM_IDs along with their inventory prices from the INVENTORY table. Format the inventory prices so they are displayed in the format $999.99.

◀ LESSON C ▶

After completing this chapter, you should be able to:

- Create SQL queries that join multiple tables
- Create nested SQL queries
- Combine query results using set operators
- Create and use database views

So far, all of your SQL queries have retrieved data from a single database table. In this lesson, you learn how to create queries that retrieve data from multiple tables. You also learn how to create nested queries, in which the output from one query serves as a search condition in a second query. And, you learn how to create advanced SQL queries that select rows for updating; you learn how to create and query database views; finally, you learn how to create database indexes. For this lesson, you execute SQL commands and queries using the fully populated Northwoods University database tables. Your first task is to create or refresh these tables by running the Ch3Northwoods.sql script in the Chapter03 folder on your Data Disk. You also change the page and line sizes in your SQL*Plus environment.

To run the script files and change the page and line sizes:

1. If necessary, start SQL*Plus and log onto the database. Run the Ch3Northwoods.sql script by typing the following command at the SQL prompt:

   ```
   START c:\OraData\Chapter03\Ch3Northwoods.sql
   ```

2. Click **Options** on the menu bar, click **Environment**, and then select **linesize** in the Set Options list. Select the **Custom** option button, and type **120** in the Value column to display 120 characters per line.
3. Select **pagesize** in the Set Options list. Select the **Custom** option button, type **40** in the Value column to display 40 lines per page, and then click **OK** to save your settings.

NOTE

As an alternative to Steps 2 and 3, you can issue the commands SET LINESIZE 120 and SET PAGESIZE 40 on separate lines at the SQL*Plus prompt.

JOINING MULTIPLE TABLES

One of the strengths of SQL is its ability to **join**, or combine, data from multiple database tables using foreign key references. The general syntax of a SELECT query that joins two tables is:

```
SELECT column1, column2, ...
FROM table1, table2
WHERE table1.joincolumn = table2.joincolumn
AND search_condition(s);
```

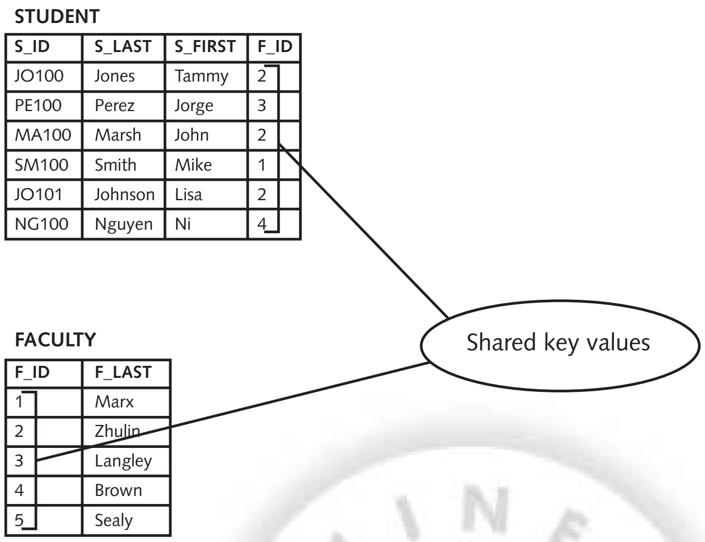
The SELECT clause lists the names of the columns to display in the query output. The FROM clause lists the names of all of the tables involved in the join operation. If you display a column that exists in more than one of the tables in the FROM clause, you must qualify the column name in the SELECT clause. To **qualify** a column name in the SELECT clause, you specify the name of the table that contains the column, followed by a period, and then specify the column name. Because the column exists in more than one table, you can qualify the column name using the name of any table listed in the FROM clause that contains the column. For example, suppose you write a query that lists the F_ID column in the SELECT clause, and the STUDENT and FACULTY tables in the FROM clause. Because F_ID exists in both the STUDENT and FACULTY tables, you must qualify the F_ID column in the SELECT clause. To do so, you can use the following syntax: `STUDENT.F_ID`. You could also qualify the F_ID column as `FACULTY.F_ID`.

The WHERE clause contains the **join condition**, which specifies the table names to be joined and column names on which to join the tables. The join condition contains the foreign key reference in one table and the primary key in the other table. You can add search conditions using the AND and OR operators.

SQL supports multiple types of join queries. In this book, you learn to create inner joins, outer joins, and self-joins.

Inner Joins

The simplest type of join occurs when you join two tables based on values in one table being equal to values in another table. This type of join is called an **inner join**, **equality join**, **equijoin**, or **natural join**. For example, suppose you want to retrieve student last and first names, along with each student's advisor's ID and last name. This query retrieves data from two database tables; the student last and first names are in the STUDENT table, and the advisor last names are in the FACULTY table. Each student's advisor ID appears in the F_ID column in the STUDENT table, and each F_ID value in the STUDENT table corresponds to an F_ID value in the FACULTY table, as shown in Figure 3-42.


STUDENT

S_ID	S_LAST	S_FIRST	F_ID
JO100	Jones	Tammy	2
PE100	Perez	Jorge	3
MA100	Marsh	John	2
SM100	Smith	Mike	1
JO101	Johnson	Lisa	2
NG100	Nguyen	Ni	4

FACULTY

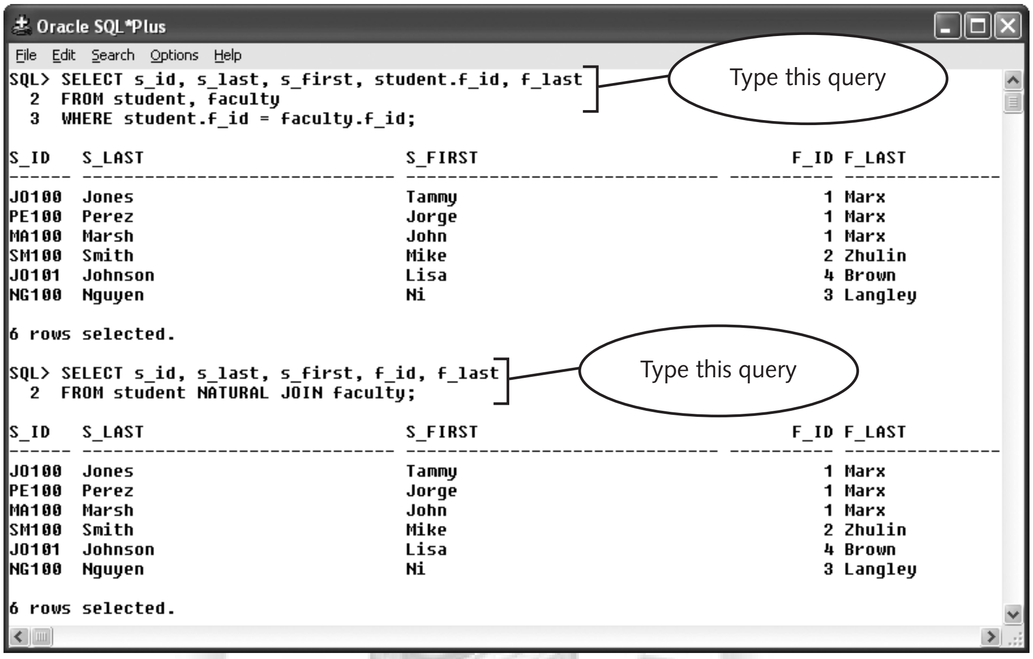
F_ID	F_LAST
1	Marx
2	Zhulin
3	Langley
4	Brown
5	Sealy

Figure 3-42 Joining two tables based on shared key values

Figure 3-42 shows a partial listing of the columns in the FACULTY and STUDENT tables, and shows how you join the tables on the F_ID column, which is the primary key in the FACULTY table and a foreign key in the STUDENT table. Note that because the F_ID column exists in both the STUDENT and FACULTY tables, you must qualify the F_ID column in the SELECT clause by prefacing the column name with the name of one of the tables. In the next set of steps, you create a SELECT query to retrieve student IDs and last and first names, along with each student's advisor's ID and last name, by joining the STUDENT and FACULTY tables.

To retrieve rows by joining two tables based on shared key values:

1. Start Notepad, create a new file named **Ch3CQueries.sql**, and then type the first query in Figure 3-43 to retrieve the student ID, student last and first names, advisor ID, and advisor last name. Note that you must qualify the F_ID column in the SELECT clause, because the F_ID column exists in both the STUDENT and FACULTY tables. You could qualify F_ID using either the STUDENT table or the FACULTY table.



```
SQL> SELECT s_id, s_last, s_first, student.f_id, f_last
  2  FROM student, faculty
  3  WHERE student.f_id = faculty.f_id;

S_ID   S_LAST                         S_FIRST                              F_ID F_LAST
------ ------------------------------ ------------------------------ ---------- ---------------
JO100  Jones                          Tammy                                   1 Marx
PE100  Perez                          Jorge                                   1 Marx
MA100  Marsh                          John                                    1 Marx
SM100  Smith                          Mike                                    2 Zhulin
JO101  Johnson                        Lisa                                    4 Brown
NG100  Nguyen                         Ni                                      3 Langley

6 rows selected.

SQL> SELECT s_id, s_last, s_first, f_id, f_last
  2  FROM student NATURAL JOIN faculty;

S_ID   S_LAST                         S_FIRST                              F_ID F_LAST
------ ------------------------------ ------------------------------ ---------- ---------------
JO100  Jones                          Tammy                                   1 Marx
PE100  Perez                          Jorge                                   1 Marx
MA100  Marsh                          John                                    1 Marx
SM100  Smith                          Mike                                    2 Zhulin
JO101  Johnson                        Lisa                                    4 Brown
NG100  Nguyen                         Ni                                      3 Langley

6 rows selected.
```

Figure 3-43 Queries joining two tables

2. Copy the command, paste the command into SQL*Plus, and then press **Enter** to execute the query. The output shows the values from both the STUDENT table and the FACULTY table.

NOTE

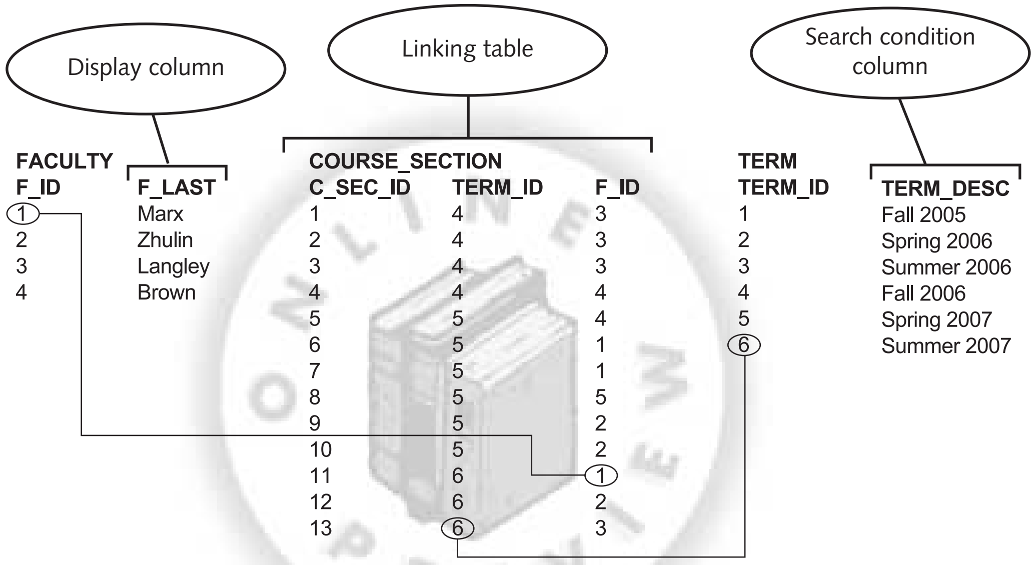
For the rest of this lesson, you type all queries into your Ch3CQueries.sql file, and then copy, paste, and execute the queries in SQL*Plus.

In cases where the tables have a single commonly named and defined column, you can use the NATURAL JOIN keywords to join the tables in the FROM clause of the SELECT statement, as shown by the second query in Figure 3-43. Notice the syntax of the FROM clause when the NATURAL JOIN keywords are included; there is no comma separating the table names, simply the keywords. In addition, notice that the qualifier is no longer included for the F_ID column in the SELECT clause of the statement. Why? For the tables to be joined correctly, the value of the F_ID column in each table must be equivalent so the same data is displayed regardless of which table is referenced. In fact, if you do include a qualifier, Oracle10*g* returns an error message.

In the preceding examples, you joined two tables. In SQL queries, you can join any number of tables in a SELECT command. When you join tables, the name of each table in the query must appear in the FROM clause. This includes tables whose columns are

display columns, which are columns that appear in the SELECT clause, and whose columns are **search columns**, which appear in search conditions. The primary key and foreign key columns on which you join the tables are called **join columns**.

When you join multiple tables, sometimes you must join the tables using an intermediary table whose columns are not display or search columns, but whose columns are join columns that serve to join the two tables. For example, suppose you want to create a query that lists the last names of all faculty members who teach during the Summer 2007 term. Figure 3-44 shows that this query involves three tables: FACULTY, COURSE_SECTION, and TERM.



FACULTY

F_ID	F_LAST
1	Marx
2	Zhulin
3	Langley
4	Brown

COURSE_SECTION

C_SEC_ID	TERM_ID	F_ID
1	4	3
2	4	3
3	4	3
4	4	4
5	5	4
6	5	1
7	5	1
8	5	5
9	5	2
10	5	2
11	6	1
12	6	2
13	6	3

TERM

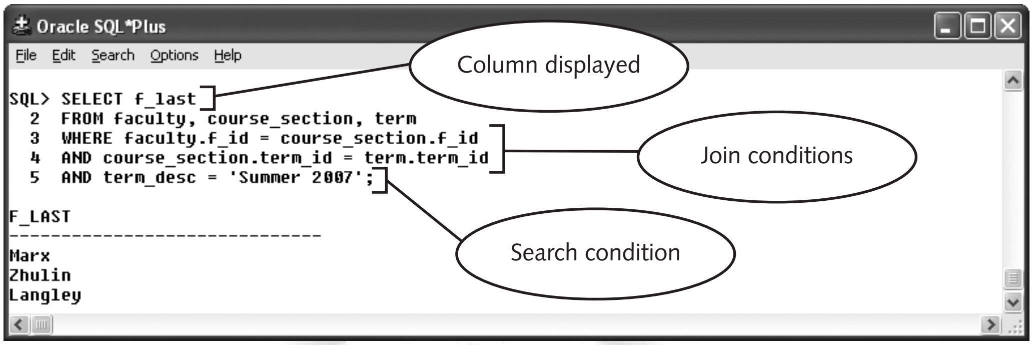
TERM_ID	TERM_DESC
1	Fall 2005
2	Spring 2006
3	Summer 2006
4	Fall 2006
5	Spring 2007
6	Summer 2007

Figure 3-44 Joining three tables

Figure 3-44 shows a partial listing of the columns in the FACULTY, COURSE_SECTION, and TERM tables. The FACULTY table contains the F_LAST column, which is the display column. The TERM table contains the TERM_DESC search column, which specifies the search condition as the Summer 2007 term. The query joins the FACULTY and TERM tables using the COURSE_SECTION table as in intermediary, or linking table. A **linking table** does not contribute any columns as display columns or search condition columns, but contains join columns that link the other tables through shared foreign key values. Even though the query does not display columns from the COURSE_SECTION table or include them in search conditions, you must include the COURSE_SECTION table in the query's FROM clause, and include join conditions to not only specify the links between the FACULTY and COURSE_SECTION tables, but also the links between the COURSE_SECTION and TERM tables. Now you execute this query in SQL*Plus.

To execute a query that joins three tables:

1. Type the query in Figure 3-45 to join the FACULTY, COURSE_SECTION, and TERM tables.



```
SQL> SELECT f_last
  2  FROM faculty, course_section, term
  3  WHERE faculty.f_id = course_section.f_id
  4  AND course_section.term_id = term.term_id
  5  AND term_desc = 'Summer 2007';

F_LAST
------------------------------
Marx
Zhulin
Langley
```

Figure 3-45 Query joining three tables

2. Execute the query. The query output shows the display column from the FACULTY table, based on the search column in the TERM table.

NOTE

The query in Figure 3-45 cannot be modified to join the three tables with the NATURAL JOIN keywords because the FACULTY and COURSE_SECTION have two commonly named and defined columns: F_ID and LOC_ID. If F_ID had been the only common column between the tables, the FROM clause could have read: FROM faculty NATURAL JOIN course_section NATURAL JOIN term, leaving the WHERE clause free to only restrict the rows retrieved to the course offered in the Summer term of 2007.

Sometimes queries that join multiple tables can become complex. For example, suppose that you want to create a query to display the COURSE_NO and GRADE values for each of student Tammy Jones' courses. This query requires you to join four tables: STUDENT (to search for S_FIRST and S_LAST), ENROLLMENT (to display GRADE), COURSE (to display COURSE_NO), and COURSE_SECTION (to join ENROLLMENT to COURSE using the C_SEC_ID join column).

You join the STUDENT and ENROLLMENT tables using the S_ID column as the join column, because S_ID is the primary key in the STUDENT table and a foreign key in the ENROLLMENT table. You need to include the COURSE_SECTION table to join the ENROLLMENT table to the COURSE table, using the COURSE_NO foreign key link between COURSE and COURSE_SECTION and the C_SEC_ID foreign key link between COURSE_SECTION and ENROLLMENT. At this point, you are probably hopelessly confused. For complex queries such as this one, it is helpful to draw a query design diagram such as the one in Figure 3-46.

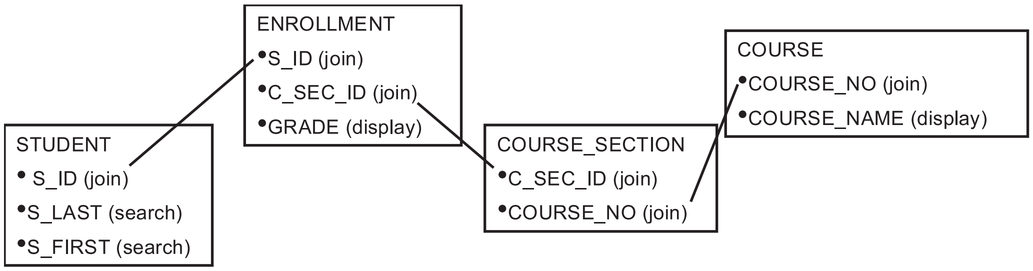


Figure 3-46 Query design diagram

A **query design diagram** shows the display and search columns in a SQL query that joins multiple tables, as well as the required join columns and their links. To create a query design diagram, you draw a rectangle to represent each table in the query. Within each table's rectangle, you list the names of each display column, search column, and join column. Then, you draw the links between the join columns as shown.

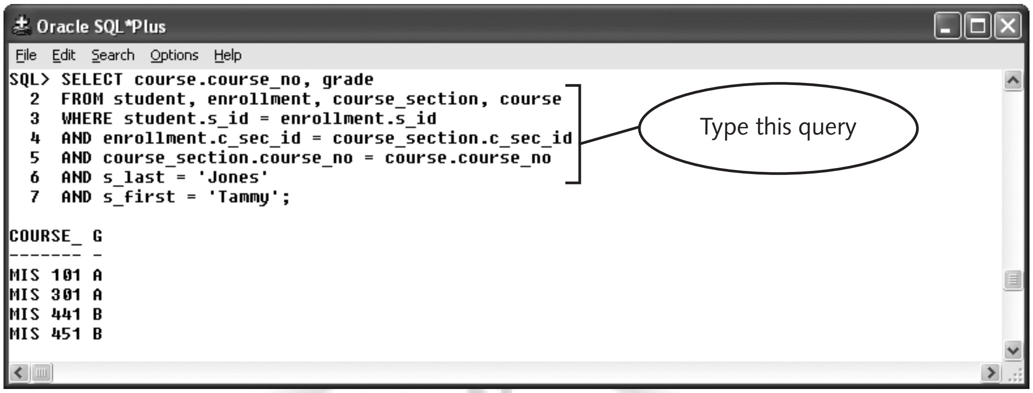
Table 3-11 describes the process for deriving the SQL query based on a query design diagram. Note that Figure 3-46 shows four tables and three links between the tables. Because there are three links, the query must have three join conditions. You must always have one fewer join condition than the total number of tables that the query joins. In this query, you are joining four tables, so you have three join conditions. Now you execute the query derived in Table 3-11 to join the four database tables.

Step	Process	Result
1	Create the SELECT clause by listing the display fields	`SELECT course_no, grade`
2	Create the FROM clause by listing the table names	`FROM student, enrollment, course_section, course`
3	Create a join condition for every link between the tables	`WHERE student.s_id = enrollment.s_id AND enrollment.c_sec_id = course_section.c_sec_id AND course_section.course_no = course.course_no`
4	Add additional search conditions for remaining search fields	`AND s_last = 'Jones' AND s_first = 'Tammy'`

Table 3-11 Deriving a SQL query from a query design diagram

To join four tables in a single query:

1. Type the query in Figure 3-47.



```
SQL> SELECT course.course_no, grade
  2  FROM student, enrollment, course_section, course
  3  WHERE student.s_id = enrollment.s_id
  4  AND enrollment.c_sec_id = course_section.c_sec_id
  5  AND course_section.course_no = course.course_no
  6  AND s_last = 'Jones'
  7  AND s_first = 'Tammy';

COURSE_ G
------- -
MIS 101 A
MIS 301 A
MIS 441 B
MIS 451 B
```

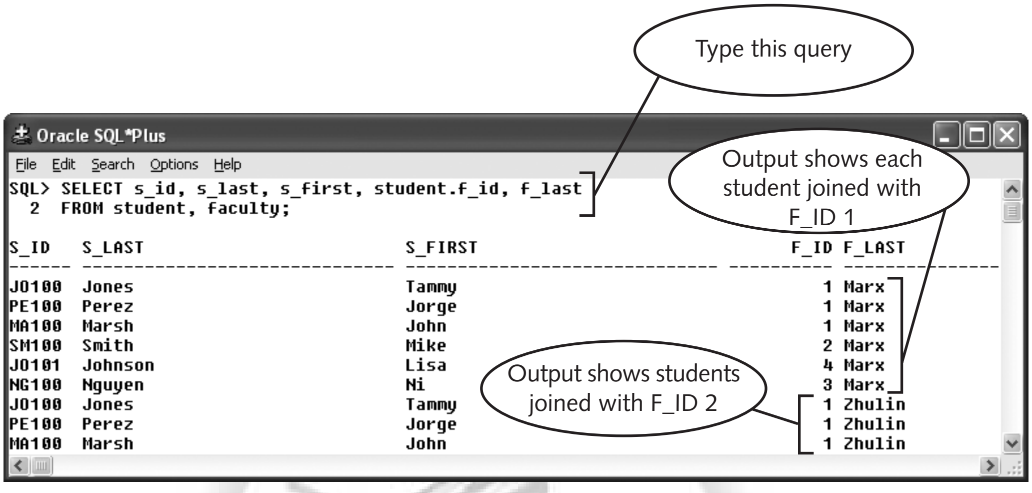
Figure 3-47 Query joining four tables

2. Execute the query. The output shows the values based on joining four tables in a single query.

If you accidentally omit a join condition in a multiple-table query, the output retrieves more rows than you expect. When you omit a join condition, the query creates a **Cartesian product**, whereby every row in one table is joined with every row in the other table. For example, suppose you repeat the query to show each student row, along with each student's advisor (see Figure 3-43), but you omit the join condition. Every row in the STUDENT table (six rows) is joined with every row in the FACULTY table (five rows). The result is 6 times 5 rows, or 30 rows. You create this query next.

To create a Cartesian product by omitting a join condition:

1. Type the query in Figure 3-48.


Type this query
Output shows each student joined with F_ID 1
Output shows students joined with F_ID 2
Oracle SQL*Plus
File Edit Search Options Help

```
SQL> SELECT s_id, s_last, s_first, student.f_id, f_last
  2  FROM student, faculty;

S_ID   S_LAST                         S_FIRST                         F_ID F_LAST
------ ------------------------------ ------------------------------ ---------- --------------
JO100  Jones                          Tammy                              1 Marx
PE100  Perez                          Jorge                              1 Marx
MA100  Marsh                          John                               1 Marx
SM100  Smith                          Mike                               2 Marx
JO101  Johnson                        Lisa                               4 Marx
NG100  Nguyen                         Ni                                 3 Marx
JO100  Jones                          Tammy                              1 Zhulin
PE100  Perez                          Jorge                              1 Zhulin
MA100  Marsh                          John                               1 Zhulin
```

Figure 3-48 Query that creates a Cartesian product (partial output shown)

2. Execute the query. The output appears as shown in Figure 3-48. (Some of the output appears off the screen.)

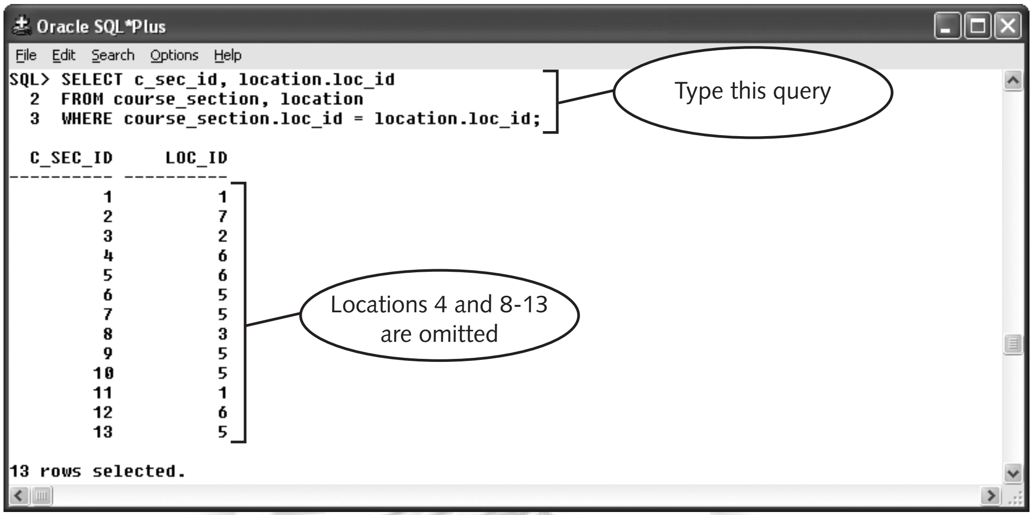
The query first joins each row in the STUDENT table with the first row in the FACULTY table (F_LAST 'Marx'). Next, each row in the STUDENT table is joined with the second row in the FACULTY table (F_LAST 'Zhulin'). This continues until each STUDENT row is joined with each FACULTY row, for a total of 30 rows returned. When a multiple-table query returns more rows than you expect, look for missing join conditions.

Outer Joins

An inner join returns rows only if values exist in all tables that are joined. If no values exist for a row in one of the joined tables, the inner join does not retrieve the row. For example, suppose you want to retrieve the different locations of the courses included in the COURSE_SECTION table. This query requires joining rows in the LOCATION and COURSE_SECTION tables. However, not every location in the LOCATION table has an associated COURSE_SECTION row, so the query retrieves rows only for locations that have associated COURSE_SECTION rows. Now you retrieve this data using an inner join, and see how the inner join query omits some of the rows.

To retrieve location and course section information using an inner join:

1. Type the query in Figure 3-49.



Oracle SQL*Plus
File Edit Search Options Help
Type this query
Locations 4 and 8-13 are omitted

```
SQL> SELECT c_sec_id, location.loc_id
  2  FROM course_section, location
  3  WHERE course_section.loc_id = location.loc_id;

  C_SEC_ID     LOC_ID
---------- ----------
         1          1
         2          7
         3          2
         4          6
         5          6
         6          5
         7          5
         8          3
         9          5
        10          5
        11          1
        12          6
        13          5

13 rows selected.
```

Figure 3-49 Inner join query that omits rows

2. Execute the query. The output displays 13 rows.

The output shows values for 13 course sections. However, notice that not all the locations from the LOCATION table are included in the output. The query retrieved information only for locations that have rows in the COURSE_SECTION table, and omitted LOC_ID values that did not have rows in the COURSE_SECTION table. To retrieve information for all locations, regardless of whether they have associated course sections or not, you must use an outer join.

An **outer join** returns all rows from one table, which is called the **inner table**. An outer join also retrieves matching rows from a second table, which is called the **outer table**. The query designer specifies which table is the inner table and which table is the outer table. In this case, because you want to retrieve all of the rows in the LOCATION table, you specify LOCATION as the inner table. Because you want to display the course section rows if they exist, you specify COURSE_SECTION as the outer table.

To create an outer join in Oracle10*g* SQL, you label the outer table in the join condition using the following syntax:

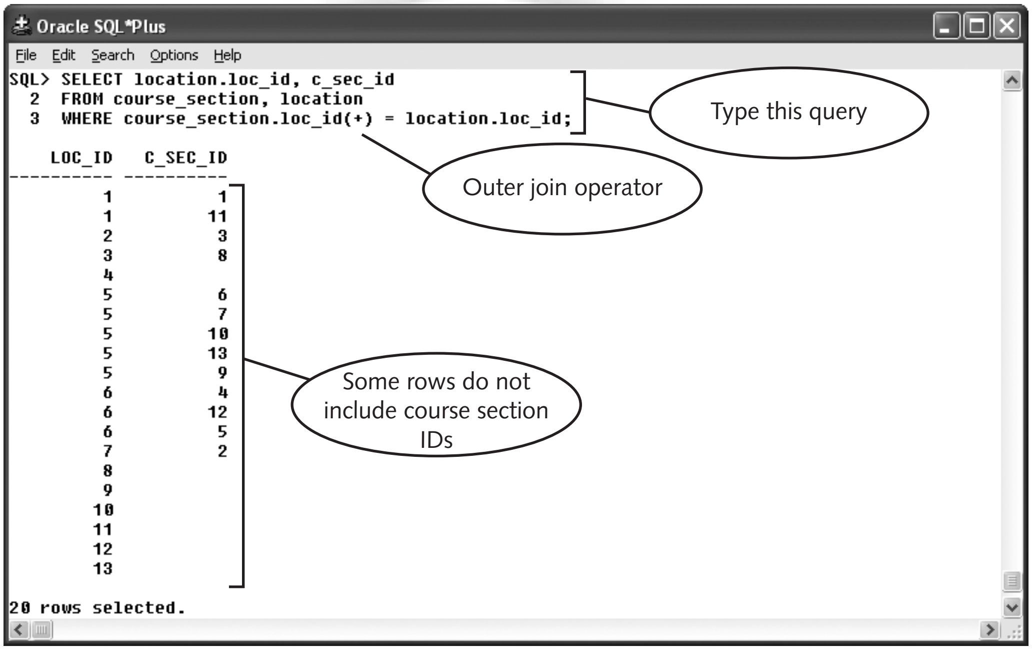
```
inner_table.join_column = outer_table.join_column(+)
```

The outer join operator (+) signals the DBMS to insert a NULL value for the columns in the outer table that do not have matching rows in the inner table. In the following set of steps, you modify the query in Figure 3-49 so that it retrieves all locations, even if those values that do not have associated rows in the COURSE_SECTION table. In this query, the LOCATION table is the inner table, and the COURSE_SECTION table is the outer table.

NOTE The outer join operator (+) can be placed on either side of the equal sign; the key is that it must be adjacent to the outer table.

To retrieve location and course_section information using an outer join:

1. Type the query in Figure 3-50, which includes the outer join operator `( + )` in the WHERE clause to specify that the COURSE_SECTION table is the outer table.


Oracle SQL*Plus
File Edit Search Options Help
Type this query
Outer join operator
Some rows do not include course section IDs

```
SQL> SELECT location.loc_id, c_sec_id
  2  FROM course_section, location
  3  WHERE course_section.loc_id(+) = location.loc_id;

    LOC_ID   C_SEC_ID
---------- ----------
         1          1
         1         11
         2          3
         3          8
         4
         5          6
         5          7
         5         10
         5         13
         5          9
         6          4
         6         12
         6          5
         7          2
         8
         9
        10
        11
        12
        13

20 rows selected.
```

Figure 3-50 Query using an outer join

NOTE Your output in Figure 3-50 may display the rows in a different order.

2. Execute the query. The output now shows values for every row in the LOCATION table, as well as the corresponding COURSE_SECTION rows, when they exist.

Note that the locations that do not have associated course sections scheduled in that location contain NULL values in the C_SEC_ID column. In Oracle10*g*, NULL (undefined) values appear as blank spaces rather than as the word NULL or as an alternate symbol.

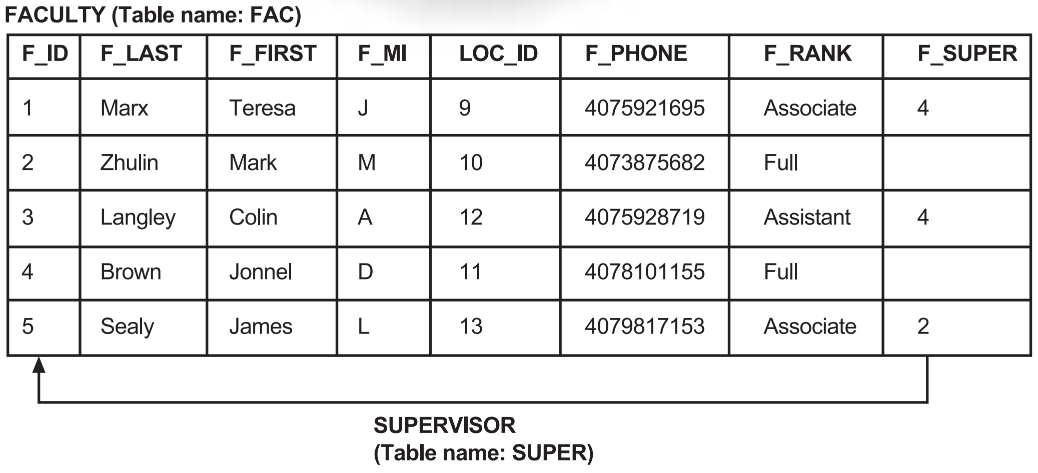
Self-joins

Sometimes a relational database table contains a foreign key that references a column in the same table. For example, at Northwoods University each assistant and associate professor is assigned to a full professor who serves as the junior professor's supervisor. The faculty ID of the supervisor for each professor is listed in the F_SUPER column of the FACULTY table. The ID stored in the F_SUPER column can be linked back to the F_ID column in the table to identify the name of the individual's supervisor. In essence, the F_SUPER column is a foreign key in the FACULTY table that references that same table's primary key. To create a query that lists the names of each junior faculty member and the names of their supervisor, you must join the FACULTY table to itself. When you create a query that joins a table to itself, you create a **self-join**.

To create a self-join, you must create a table alias and structure the query as if you are joining the table to a copy of itself. A **table alias** is an alternate name that you assign to the table in the query's FROM clause. The syntax to create a table alias in the FROM clause is:

```
FROM table1 alias1, ...
```

When you create a table alias, you must then use the table alias, rather than the table name, to qualify column names in the SELECT clause and in join conditions. Next, you create the query that lists the names of all junior faculty members and their supervisors. To make the process easier to understand, you create two table aliases—FAC for the faculty version of the table and SUPER for the supervisor version of the same table, as shown in Figure 3-51.



FACULTY (Table name: FAC)

F_ID	F_LAST	F_FIRST	F_MI	LOC_ID	F_PHONE	F_RANK	F_SUPER
1	Marx	Teresa	J	9	4075921695	Associate	4
2	Zhulin	Mark	M	10	4073875682	Full	
3	Langley	Colin	A	12	4075928719	Assistant	4
4	Brown	Jonnel	D	11	4078101155	Full	
5	Sealy	James	L	13	4079817153	Associate	2

SUPERVISOR
(Table name: SUPER)

Figure 3-51 Tables referenced in self-join query

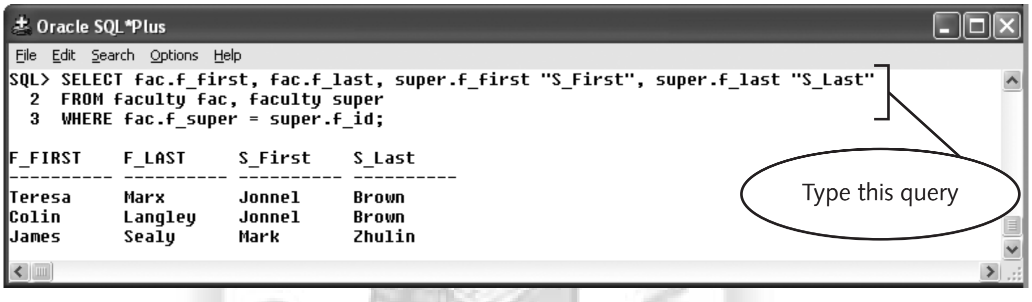
The query names the table that displays the original faculty member information as FAC and the table that displays supervisor information as SUPER. Because the F_SUPER column represents the faculty ID of the supervisor, the join condition is:

```
WHERE fac.f_super = super.f_id
```

Next, you type the query to list junior professors and their supervisor's names by creating a self-join on the FACULTY table.

To create a self-join on the FACULTY table:

1. Type the query in Figure 3-52. The SELECT clause specifies alternate column headings for the output columns to clarify the output.



```
SQL> SELECT fac.f_first, fac.f_last, super.f_first "S_First", super.f_last "S_Last"
  2  FROM faculty fac, faculty super
  3  WHERE fac.f_super = super.f_id;

F_FIRST    F_LAST     S_First    S_Last
---------- ---------- ---------- ----------
Teresa     Marx       Jonnel     Brown
Colin      Langley    Jonnel     Brown
James      Sealy      Mark       Zhulin
```

Figure 3-52 Self-join of the FACULTY table

2. Press **Enter** to execute the query. The output shows the names of all faculty members that have supervisors.

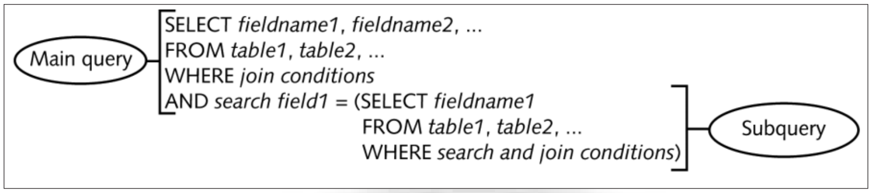
Creating Nested Queries

Sometimes you need to create a query that is based on the results of another query. For example, suppose you need to retrieve the first and last names of all students who have the same S_CLASS value (freshman, sophomore, and so forth) as student Jorge Perez. You first create a query that retrieves Jorge's S_CLASS value. Then, you create a second query to retrieve the names of the students with the same S_CLASS value. Rather than retrieving this data using two separate queries, you can retrieve the same data using a single nested query.

A **nested query** consists of a main query and one or more subqueries. The **main query** is the first query that appears in the SELECT command. A **subquery** retrieves values that the main query's search condition must match. In a nested query, the DBMS executes the subquery first, and the main query second. You create nested queries to retrieve intermediate results that are then used within the main query's search conditions. The following sections not only describe how to create nested queries containing subqueries that return a single value, but also how to create queries containing subqueries that might return multiple values. You also learn how to create nested queries with multiple subqueries, and subqueries that are also nested queries.

Creating Nested Queries with Subqueries that Return a Single Value

Figure 3-53 shows the general syntax for creating a nested query.



```
SELECT fieldname1, fieldname2, ...
FROM table1, table2, ...
WHERE join conditions
AND search field1 = (SELECT fieldname1
                     FROM table1, table2, ...
                     WHERE search and join conditions)
```

Figure 3-53 Syntax for nested query

In this syntax, the data values that the *subquery* retrieves specify the search condition *search_expression* value for the main query. The search column in the main query (*search_field1*) must reference the same column as the SELECT column (*fieldname1*) in the subquery. When the DBMS executes a nested query, it first evaluates the subquery. Then, the DBMS substitutes the subquery output into the main query, and executes the main query. If the subquery retrieves no rows or multiple rows, an error message appears.

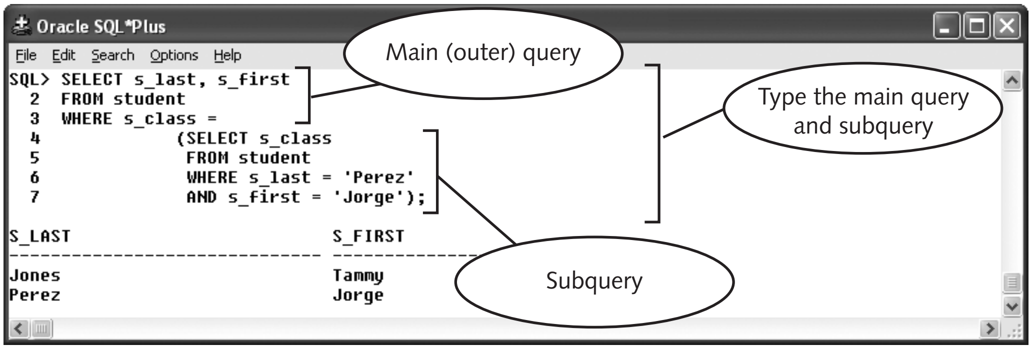
NOTE

You can also use subqueries in search conditions in UPDATE and DELETE commands.

In the next set of steps, you create a nested query that retrieves the first and last names of all students who have the same S_CLASS value as student Jorge Perez. The subquery retrieves a single value, which is Jorge's S_CLASS value. The main query retrieves the student first and last names.

To create a nested query:

1. Type the query in Figure 3-54 to retrieve the names of all students who have the same S_CLASS value as student Jorge Perez. To make the query easier to read and understand, indent the subquery as shown. Note that the subquery must retrieve the S_CLASS column, and the main query's search condition must also specify the S_CLASS column as the search column.
2. Execute the query. The output shows the students who have the S_CLASS value 'SR', which includes Tammy Jones and Jorge Perez.


Oracle SQL*Plus
File Edit Search Options Help
Main (outer) query
Type the main query and subquery
Subquery

```
SQL> SELECT s_last, s_first
  2  FROM student
  3  WHERE s_class =
  4             (SELECT s_class
  5              FROM student
  6              WHERE s_last = 'Perez'
  7              AND s_first = 'Jorge');

S_LAST                         S_FIRST
------------------------------ --------------
Jones                          Tammy
Perez                          Jorge
```

Figure 3-54 Creating a nested query that returns a single value

TIP To debug a nested query, run the subquery separately to ensure that it is retrieving the correct results before you combine it with the main query.

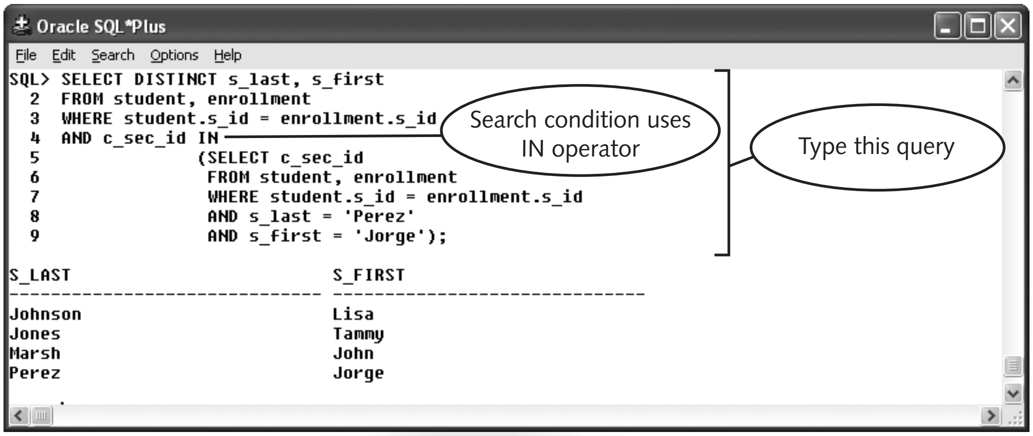
Creating Subqueries that Return Multiple Values

Because the subquery in Figure 3-54 returns a single value, which is Jorge Perez's S_CLASS value, the main query's search condition uses an equal to comparison operator (=). What if the subquery might return multiple results, each of which satisfies the search condition? For example, suppose you want to retrieve the names of all students who have ever been enrolled in the same course section as Jorge Perez. The subquery retrieves all C_SEC_ID values in the ENROLLMENT table for Jorge, which probably includes multiple values. The main query then retrieves the names of all students who have enrolled in any of these same course sections. To structure this query, you must use the IN comparison operator instead of the equal to comparison operator (=). (In Lesson B, you learned how to use the IN comparison operator to create a search condition to match a set of values.) You can also use the NOT IN operator to specify that the main query should retrieve all rows except those that satisfy the search condition.

Next, you create a nested query in which the subquery retrieves multiple values. The subquery retrieves the C_SEC_ID value of all courses in which Jorge Perez has enrolled, and the main query uses the IN operator to match the names of all students who have ever been enrolled in one of these courses. You use the DISTINCT qualifier in the main query's SELECT clause to suppress duplicate names.

To write a nested query with a subquery that retrieves multiple values:

1. Type the query in Figure 3-55 to retrieve the names of all students who have enrolled in the same course sections as Jorge Perez. Note that the main query's search condition uses the IN comparison operator, because the subquery returns multiple values.


Oracle SQL*Plus
File Edit Search Options Help
Search condition uses IN operator
Type this query

```
SQL> SELECT DISTINCT s_last, s_first
  2  FROM student, enrollment
  3  WHERE student.s_id = enrollment.s_id
  4  AND c_sec_id IN
  5               (SELECT c_sec_id
  6                FROM student, enrollment
  7                WHERE student.s_id = enrollment.s_id
  8                AND s_last = 'Perez'
  9                AND s_first = 'Jorge');

S_LAST                         S_FIRST
------------------------------ ------------------------------
Johnson                        Lisa
Jones                          Tammy
Marsh                          John
Perez                          Jorge
```

Figure 3-55 Creating a nested query whose subquery returns multiple values

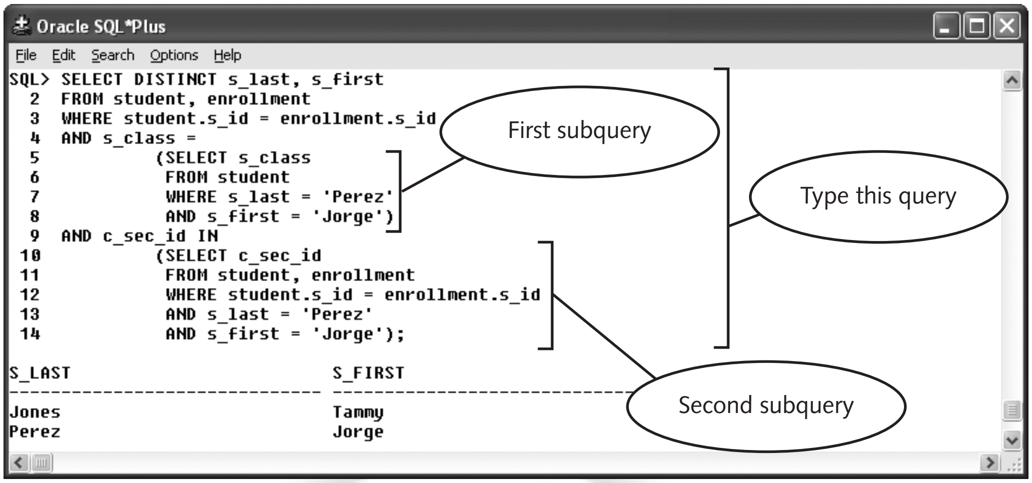
2. Execute the query. The output values appear as shown.

Using Multiple Subqueries Within a Nested Query

You can specify multiple subqueries within a single nested query by using the AND and OR operators to join the search conditions associated with the subqueries. For example, suppose you want to retrieve the names of all students who have the same S_CLASS value as Jorge Perez and have also been enrolled in a course section with him. You use a query that joins the two search conditions using the AND operator, and each search condition is specified using a separate subquery.

To create a nested query that uses multiple subqueries:

1. Type the query in Figure 3-56 to retrieve the names of all students who have the same S_CLASS value as Jorge Perez and who have also been enrolled in a course section with him.
2. Execute the query. The output shows that only one student other than Jorge himself, Tammy Jones, satisfies both these conditions.



```
SQL> SELECT DISTINCT s_last, s_first
  2  FROM student, enrollment
  3  WHERE student.s_id = enrollment.s_id
  4  AND s_class =
  5           (SELECT s_class
  6            FROM student
  7            WHERE s_last = 'Perez'
  8            AND s_first = 'Jorge')
  9  AND c_sec_id IN
 10           (SELECT c_sec_id
 11            FROM student, enrollment
 12            WHERE student.s_id = enrollment.s_id
 13            AND s_last = 'Perez'
 14            AND s_first = 'Jorge');

S_LAST                         S_FIRST
------------------------------ ------------------------------
Jones                          Tammy
Perez                          Jorge
```

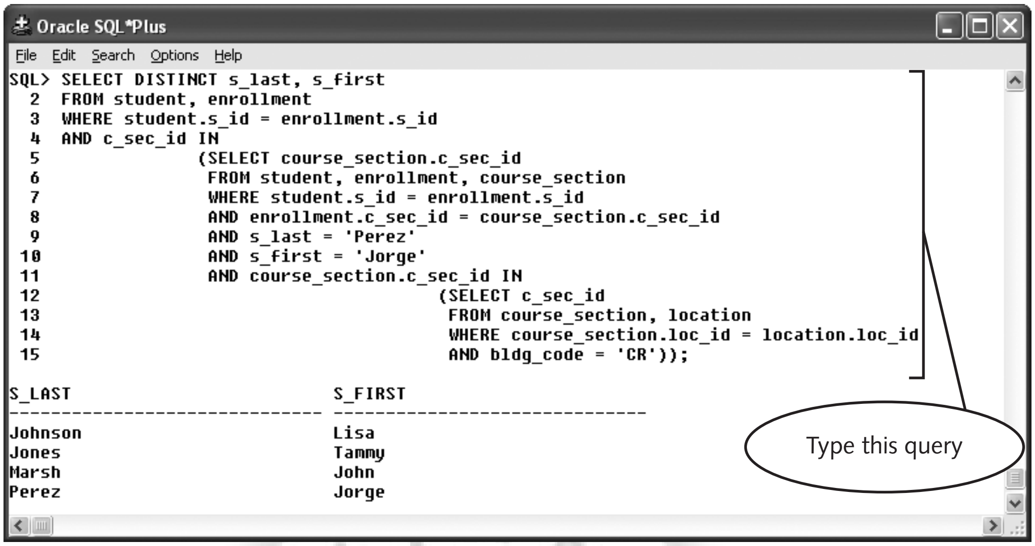
Figure 3-56 Nested query that has multiple subqueries

Creating Nested Subqueries

A **nested subquery** is a subquery that contains a second subquery that specifies its search expression. Now you use a nested subquery to create a query to retrieve the names of students who have taken courses with Jorge Perez in the CR building. The innermost subquery retrieves the course section ID values for all course sections located in the CR building. This subquery's main query retrieves the course sections from this subset taken by Jorge Perez. The outermost main query retrieves the names of all students who enrolled in the course sections that satisfy both of these conditions.

To create a query with a nested subquery:

1. Type the query in Figure 3-57.



```
SQL> SELECT DISTINCT s_last, s_first
  FROM student, enrollment
  WHERE student.s_id = enrollment.s_id
  AND c_sec_id IN
             (SELECT course_section.c_sec_id
              FROM student, enrollment, course_section
              WHERE student.s_id = enrollment.s_id
              AND enrollment.c_sec_id = course_section.c_sec_id
              AND s_last = 'Perez'
              AND s_first = 'Jorge'
              AND course_section.c_sec_id IN
                                       (SELECT c_sec_id
                                        FROM course_section, location
                                        WHERE course_section.loc_id = location.loc_id
                                        AND bldg_code = 'CR'));

S_LAST                         S_FIRST
------------------------------ ------------------------------
Johnson                        Lisa
Jones                          Tammy
Marsh                          John
Perez                          Jorge
```

Figure 3-57 Creating a query with a nested subquery

2. Execute the query. Note that the output excludes any student that may have taken the same course section as Jorge Perez, but the course was located in a different building.

You can create subqueries that nest to multiple levels. However, as a general rule, nested queries or queries with nested subqueries execute slower than queries that join multiple tables, so you should probably not use nested queries unless you cannot retrieve the desired result using a nonnested query.

Using Set Operators to Combine Query Results

Sometimes, you need to combine the results of two separate queries in ways that are not specified by foreign key relationships. For example, you might need to create a telephone directory that shows the names and telephone numbers of all students and faculty members at Northwoods University. Or, you might want to retrieve the first and last names of faculty members who have offices in the BUS building and have taught a class in BUS. These rows do not have any foreign key relationships, so the only way to retrieve them is to create multiple queries. Alternately, you can use common set operators to combine the results of separate queries into a single result. Table 3-12 describes the Oracle10*g* SQL set operators, and the following sections describe how to use them.

Set Operator	Description
UNION	Returns all rows from both queries, and suppresses duplicate rows
UNION ALL	Returns all rows from both queries, and displays duplicate rows
INTERSECT	Returns only rows returned by both queries
MINUS	Returns the rows returned by the first query minus the matching rows returned by the second query

Table 3-12 Oracle10*g* SQL set operators

UNION and UNION ALL

A query that uses the UNION set operator joins the output of two unrelated queries into a single output result. The general syntax of a query that uses the UNION operator is:

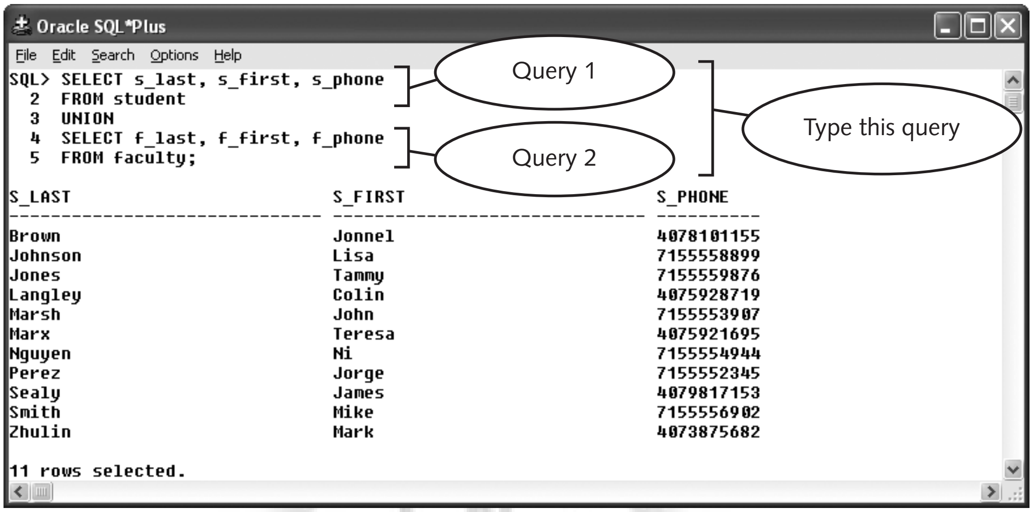
```
query1 UNION query2;
```

In this syntax, *query1* represents the first query, and *query2* represents the second query. Both queries must have the same number of display columns in their SELECT clauses, and each display column in the first query must have the same data type as the corresponding column in the second query. For example, if the display columns returned by *query1* are a NUMBER data column and then a VARCHAR2 data column, then the display columns returned by *query2* must also be a NUMBER data column followed by a VARCHAR2 data column.

To create a telephone directory of every student and faculty member at Northwoods University, you need to create a query to list the last name, first name, and telephone number of every student and every faculty member in the Northwoods University database. You can create two separate queries to retrieve these rows, but you cannot retrieve the results using a single query. (The foreign key student/faculty advising relationship that joins these tables is not relevant for this query, because you want to display all students and all faculty members, not just students and their faculty advisors.) Because a single query cannot return data from two unrelated queries, you must use a UNION set operator to join the query outputs. Next, you create a query that uses the UNION set operator.

To create a query that uses the UNION set operator:

1. Type the query in Figure 3-58.



```
SQL> SELECT s_last, s_first, s_phone
  2  FROM student
  3  UNION
  4  SELECT f_last, f_first, f_phone
  5  FROM faculty;

S_LAST                         S_FIRST                        S_PHONE
------------------------------ ------------------------------ ----------
Brown                          Jonnel                         4078101155
Johnson                        Lisa                           7155558899
Jones                          Tammy                          7155559876
Langley                        Colin                          4075928719
Marsh                          John                           7155553907
Marx                           Teresa                         4075921695
Nguyen                         Ni                             7155554944
Perez                          Jorge                          7155552345
Sealy                          James                          4079817153
Smith                          Mike                           7155556902
Zhulin                         Mark                           4073875682

11 rows selected.
```

Figure 3-58 Query using the set UNION operator

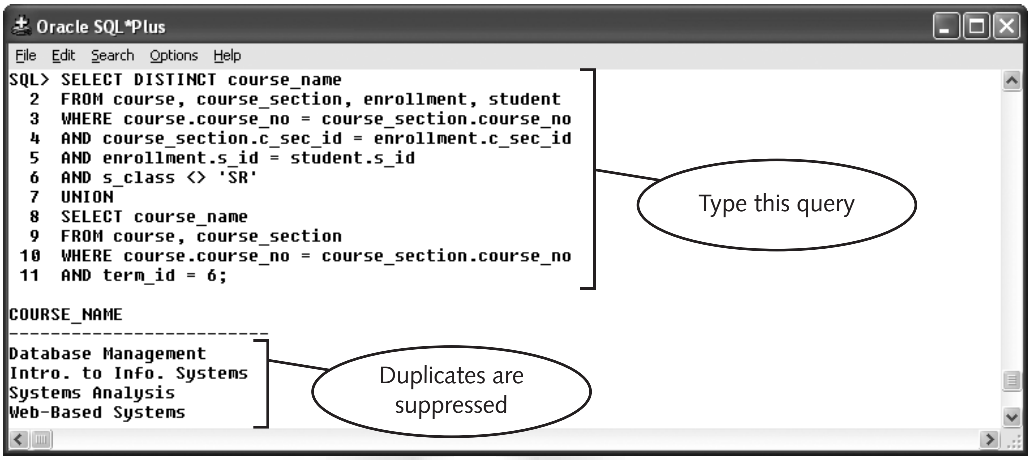
2. Execute the query. The output shows the student and faculty names and telephone numbers in a single list. The results are sorted by the first display column.

In the query output in Figure 3-58, the default output column titles are the column names in the first query's SELECT statement. The output results are sorted based on the first column of the first SELECT statement.

If one row in the first query exactly matches a row in the second query, the UNION output suppresses duplicates, and shows the duplicate row only once. To display duplicate rows, you must use the UNION ALL operator. For illustrative purposes, suppose you need to create a query that displays the name of every course that was taken by freshmen, sophomores, and juniors (i.e., not seniors) and a list of all courses offered in term 6. Because some of the students may have taken more than one course in a term, this query may retrieve duplicate names. If you use the UNION operator to combine the results of these two queries, the duplicate rows are suppressed. If you use the UNION ALL operator, then duplicate rows appear. In the next set of steps, you create a query with the UNION operator and then with the UNION ALL operator, and compare the results.

To create UNION and UNION ALL queries:

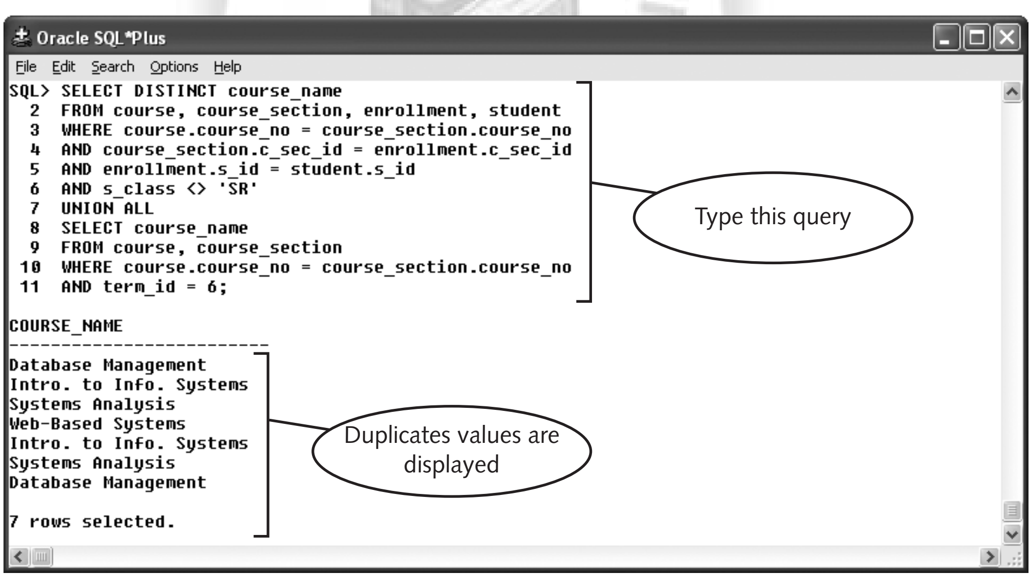
1. Type and execute the query in Figure 3-59, which uses the UNION operator. Note that duplicate rows do not appear.



```
SQL> SELECT DISTINCT course_name
  2  FROM course, course_section, enrollment, student
  3  WHERE course.course_no = course_section.course_no
  4  AND course_section.c_sec_id = enrollment.c_sec_id
  5  AND enrollment.s_id = student.s_id
  6  AND s_class <> 'SR'
  7  UNION
  8  SELECT course_name
  9  FROM course, course_section
 10  WHERE course.course_no = course_section.course_no
 11  AND term_id = 6;

COURSE_NAME
-------------------------
Database Management
Intro. to Info. Systems
Systems Analysis
Web-Based Systems
```

Figure 3-59 Duplicate course names suppressed by UNION set operator

2. Now type and execute the query in Figure 3-60, which uses the UNION ALL operator. This time the duplicate rows appear, and show that three courses satisfy the search requirements in both queries.


Oracle SQL*Plus
File Edit Search Options Help
Type this query
Duplicates values are displayed

```
SQL> SELECT DISTINCT course_name
  2  FROM course, course_section, enrollment, student
  3  WHERE course.course_no = course_section.course_no
  4  AND course_section.c_sec_id = enrollment.c_sec_id
  5  AND enrollment.s_id = student.s_id
  6  AND s_class <> 'SR'
  7  UNION ALL
  8  SELECT course_name
  9  FROM course, course_section
 10  WHERE course.course_no = course_section.course_no
 11  AND term_id = 6;

COURSE_NAME
-------------------------
Database Management
Intro. to Info. Systems
Systems Analysis
Web-Based Systems
Intro. to Info. Systems
Systems Analysis
Database Management

7 rows selected.
```

Figure 3-60 Using the UNION ALL set operator

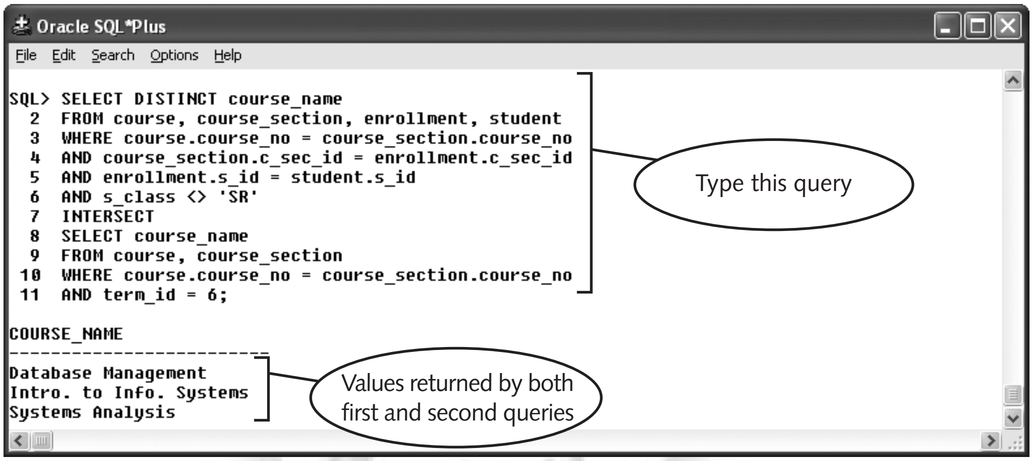
INTERSECT

Some queries require an output that finds the intersection, or matching rows, in two queries. Like a UNION, an INTERSECT query requires that both queries have the same number of display columns in the SELECT statement and that each column in the first query has the same data type as the corresponding column in the second query. An INTERSECT query automatically suppresses duplicate rows.

In the previous section, you created a UNION query that displayed the names of courses taken only by students who were not seniors, in addition to courses that were offered in term 6. Suppose you want to create a query that retrieves only the courses that satisfy both of these requirements. One approach is to re-create the query using the INTERSECT set operator. You do this next.

To create a query that uses the INTERSECT set operator:

1. Type the query in Figure 3-61.


Oracle SQL*Plus
File Edit Search Options Help
Type this query
Values returned by both first and second queries

```
SQL> SELECT DISTINCT course_name
  2  FROM course, course_section, enrollment, student
  3  WHERE course.course_no = course_section.course_no
  4  AND course_section.c_sec_id = enrollment.c_sec_id
  5  AND enrollment.s_id = student.s_id
  6  AND s_class <> 'SR'
  7  INTERSECT
  8  SELECT course_name
  9  FROM course, course_section
 10  WHERE course.course_no = course_section.course_no
 11  AND term_id = 6;

COURSE_NAME
-------------------------
Database Management
Intro. to Info. Systems
Systems Analysis
```

Figure 3-61 Query using the INTERSECT set operator

2. Execute the query. The query output shows that three courses satisfy both search conditions.

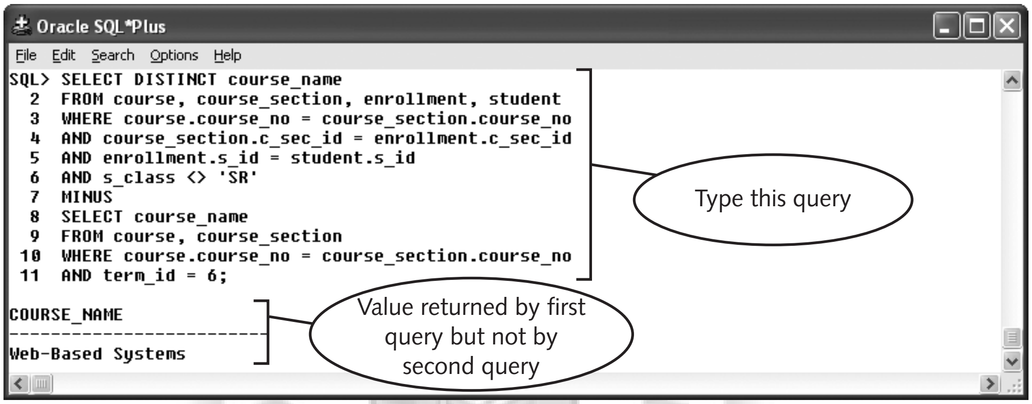
MINUS

The MINUS operator allows you to find the difference between two unrelated query result lists. As with the UNION and INTERSECT operators, the MINUS operator requires that both queries have the same number of display columns in the SELECT statement, and that each column in the first query has the same data type as the corresponding column in the second query. And as with the other set operators, the MINUS operator automatically suppresses duplicate rows.

Suppose you want to retrieve the courses that were taken by freshmen, sophomores, and juniors, but were not offered in term 6. Because the first portion of the previous query determined the courses that were not taken by seniors, and the latter portion identified the courses offered in term 6, you can simply modify the previous query by using the MINUS set operator. The results of the modified query display courses taken by freshmen, sophomores, and juniors with any courses not offered in term 6 removed from the results.

To create a query using the MINUS operator:

1. Type the query in Figure 3-62, which uses the MINUS operator to display the difference between two queries.


Oracle SQL*Plus
File Edit Search Options Help
Type this query
Value returned by first query but not by second query

```
SQL> SELECT DISTINCT course_name
  2  FROM course, course_section, enrollment, student
  3  WHERE course.course_no = course_section.course_no
  4  AND course_section.c_sec_id = enrollment.c_sec_id
  5  AND enrollment.s_id = student.s_id
  6  AND s_class <> 'SR'
  7  MINUS
  8  SELECT course_name
  9  FROM course, course_section
 10  WHERE course.course_no = course_section.course_no
 11  AND term_id = 6;

COURSE_NAME
-------------------------
Web-Based Systems
```

Figure 3-62 Query using the MINUS set operator

2. Execute the query. The query output shows only the course that meets the conditions specified by the first portion of the query, but did not meet the condition established by the second portion of the query.

Creating and Using Database Views

A database view presents data in a different format than the one in which the DBMS stores the data in the underlying database tables. A view is similar to storing the result of a query in the database. You create a view using a SQL query called the **source query**. The source query can specify a subset of a single table's columns or rows, or the query can join multiple tables. You can then retrieve data from the view the same way you retrieve data from a database table. You can use special views called **updatable views**, also called simple views, to insert, update, and delete data in the underlying database tables.

A view derives its data from database tables called **source tables**, or underlying base tables. When users insert, update, or delete data values in a view's source tables, the view reflects the updates as well. If a DBA alters the structure of a view's source tables, or if a DBA drops a view's source table, then the view becomes invalid and can no longer be used.

Once you create a view, you can use the view to retrieve rows just as with a table. This saves you from having to reenter complex query commands. For example, you can create a view to list the item description, size, color, price, and quantity ordered for all items associated with order ID (O_ID) 1 in the Clearwater Traders database. This view is based on a query that joins four tables: ORDERS, ORDER_LINE, INVENTORY, and ITEM. Once you create the view, you can easily query it, and you do not need to type the join or search conditions to display the rows.

DBAs use views to enforce database security by allowing certain users to view only selected table columns or rows. A specific user might be given privileges to view and edit data in a view based on a table, but she or he may not be given privileges to manipulate all of the data in the table. For example, suppose a data entry person is needed to insert, update, and view data in the FACULTY table. For security reasons, the DBA does not want this person to have access to the F_PIN column. The DBA creates a view based on the FACULTY table that contains all table columns except F_PIN. The following sections describe how to create and query views.

Creating Views

The syntax to create a view is:

```
CREATE VIEW view_name
AS source_query;
```

In this syntax, *view_name* must follow the Oracle naming standard. *View_name* cannot already exist in the user's database schema. If there is a possibility that you have already created a view using a specific name, you can use the following command to create or replace the existing view:

```
CREATE OR REPLACE VIEW view_name
AS source_query;
```

After you create a view, you can grant or revoke privileges to other users to perform operations such as SELECT, INSERT, or UPDATE. When you replace an existing view, you replace only the view column definitions. All of the existing object privileges that you granted on the view remain intact.

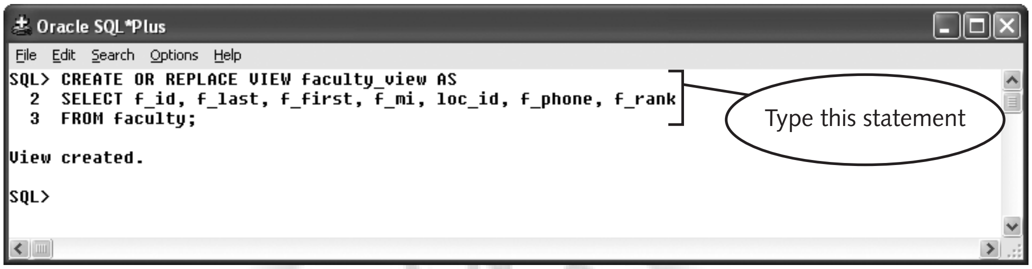
Source_query can retrieve columns from a single table or can join multiple tables. *Source_query*'s SELECT clause can contain arithmetic and single-row functions, as long as you create an alias for each column that performs a calculation or uses a function. *Source_query* can also contain a search condition.

Recall that you can create updatable views that users can use to insert, update, or delete values in the view's source tables. When you create an updatable view, the SQL query has specific restrictions: The SELECT clause can contain only column names, and it cannot specify arithmetic calculations or single-row functions. The query also cannot contain the ORDER BY, DISTINCT, or GROUP BY clauses, group functions, or set operators. And, the query's search condition cannot contain a nested query. Next, you create an updatable view named

FACULTY_VIEW, which is based on columns in the FACULTY table. This view contains all of the FACULTY columns except F_PIN and F_IMAGE.

To create an updatable view:

1. Type the command in Figure 3-63 to create the FACULTY_VIEW. Because the query does not contain any of the restricted items, this view is an updateable view.



```
SQL> CREATE OR REPLACE VIEW faculty_view AS
  2  SELECT f_id, f_last, f_first, f_mi, loc_id, f_phone, f_rank
  3  FROM faculty;

View created.

SQL>
```

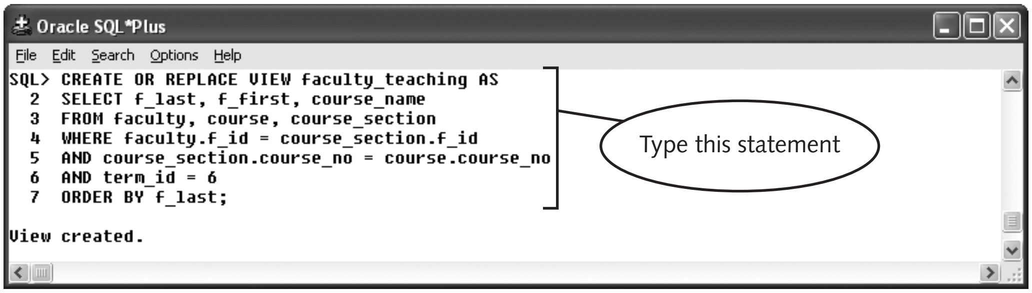
Figure 3-63 Creating an updateable view

2. Execute the command. The "View created." confirmation message confirms that the DBMS created the view.

Next you use a complex source query to create a view that contains a listing of the courses being taught by faculty members for term 6. This source query contains columns from multiple tables so it restricts the DML operations that can be performed. The view lists the faculty member's name and the name of the course he or she is teaching. In addition, the data is sorted in alphabetical order by the last names of the faculty members.

To create a complex view that cannot be updated:

1. Type the command in Figure 3-64 to create the view that cannot be updated.



```
SQL> CREATE OR REPLACE VIEW faculty_teaching AS
  2  SELECT f_last, f_first, course_name
  3  FROM faculty, course, course_section
  4  WHERE faculty.f_id = course_section.f_id
  5  AND course_section.course_no = course.course_no
  6  AND term_id = 6
  7  ORDER BY f_last;

View created.
```

Figure 3-64 Creating a view that cannot be updated

2. Execute the command. The message "View created." confirms that the DBMS successfully created the view.

Executing Action Queries Using Views

After you create an updatable view, you can use the view to execute action queries that insert, update, or delete data in the underlying source tables. In the following steps, you use FACULTY_VIEW to insert a row into the FACULTY source table.

To insert a row using the FACULTY_VIEW:

1. Type the following action query to insert a new row into FACULTY_VIEW:

   ```
   INSERT INTO faculty_view VALUES
   (6, 'May', 'Lisa', 'I', 11, '7155552508', 'INST');
   ```

2. Press **Enter** to execute the query. The message "1 row created." confirms that the DBMS successfully inserted the row. Although you inserted the row using the view, the DBMS actually inserted the row into the FACULTY table.
3. Type **SELECT * FROM faculty_view;** to determine whether the new faculty member is included in the table.
4. Type **DELETE FROM faculty_view WHERE f_last = 'MAY';** to delete the new faculty member.

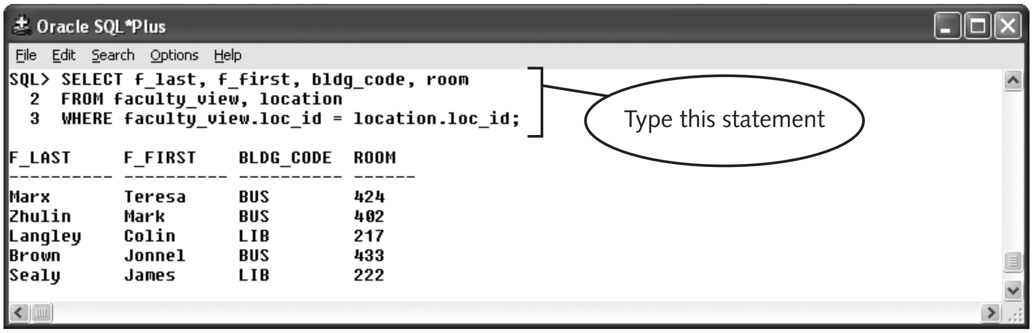
You can also execute update action queries and delete action queries using the view just as with a database table, provided that you do not violate any constraints that exist for the underlying database table, and provided that you have sufficient object privileges for updating or deleting rows in the view.

Retrieving Rows from Views

You can query a view using a SELECT statement, just as with a database table, and use the view in complex queries that involve join operations and subqueries. Next, you create queries using your database views. You create a query that joins FACULTY_VIEW with the LOCATION table to list the names of each faculty member, along with the building code and room number of the faculty member's office.

To create queries using database views:

1. Type the query in Figure 3-65 to retrieve the faculty information by joining FACULTY_VIEW with the LOCATION table.



```
SQL> SELECT f_last, f_first, bldg_code, room
  2  FROM faculty_view, location
  3  WHERE faculty_view.loc_id = location.loc_id;

F_LAST     F_FIRST    BLDG_CODE  ROOM
---------- ---------- ---------- ------
Marx       Teresa     BUS        424
Zhulin     Mark       BUS        402
Langley    Colin      LIB        217
Brown      Jonnel     BUS        433
Sealy      James      LIB        222
```

Figure 3-65 Query using a database view

2. Execute the query. Note that the query to retrieve data from the view is much simpler than the query to retrieve the data from the underlying database tables.

Removing Views

You use the DROP VIEW command to remove a view from your user schema. This command has the following syntax:

```
DROP VIEW view_name;
```

Recall that the DBMS bases a view on a source query. When you drop a view, you do not delete the data values that appear in the view—you drop only the view definition. Now you drop the FACULTY_VIEW view that you created earlier.

To drop FACULTY_VIEW:

1. Type and execute the following command at the SQL prompt:

   ```
   DROP VIEW faculty_view;
   ```

   The "View dropped." confirmation message indicates that the DBMS successfully dropped the view.
2. Close SQL*Plus. Switch to Notepad, save **3CQueries.sql**, and then close Notepad.

Summary

- To join multiple tables in a SELECT query, you must list in the FROM clause every table that contains a display column, search column, or join column. You include a join condition to specify every link between those tables in the WHERE clause.
- If two tables only have one commonly defined column, the NATURAL JOIN keyword can be used to join the tables in the FROM clause.
- If you display a column in a multiple-table query that exists in more than one of the tables, you must qualify the column name by writing the table name, followed by a period, and then the column name.
- If you accidentally omit a join condition in a multiple-table query, the result is a Cartesian product, which joins every row in one table with every row in the other table.
- An inner join retrieves all matching columns in two or more tables.
- An outer join retrieves all rows in the first, or inner, table, even if no corresponding values exist in the second, or outer, table.

- A self-join joins a table to itself. To create a self-join, you must create a table alias and structure the query as if you are joining the table to a copy of itself.
- A nested query consists of a main query and a subquery that retrieves an intermediate result that serves as the main query's search expression. You can nest subqueries to multiple levels.
- A query that uses the UNION set operator combines the results of two unrelated queries and suppresses duplicate rows. A query that uses the UNION ALL set operator combines the results of two unrelated queries and displays duplicate rows. Queries that use either one of the UNION set operators must have the same number of display columns in the SELECT clause, and each column in the first query must have the same data type as the corresponding column in the second query.
- A query that uses the INTERSECT set operator returns the intersection, or matching rows, in two unrelated queries.
- A query that uses the MINUS set operator retrieves the difference between two unrelated query result outputs.
- A database view is a logical table that you create using a query. You can use a view to enforce security measures within a table or to simplify the process of displaying data that a complex query retrieves.
- An updatable view is a view that allows users to insert, update, or delete values in the view's source tables. When you create an updatable view, the SQL query's SELECT clause can contain only column names, and cannot specify arithmetic calculations or single-row functions. The query also cannot contain the ORDER BY, DISTINCT, or GROUP BY clauses, group functions, set operators, or a nested query.

Review Questions

1. If you are creating a query that joins four tables, how many join conditions should you create?
2. The ________________ operator is used to return multiple values to an outer query.
3. In a query that includes the equal to (=) operator, to return values to an outer query, duplicate values are suppressed. True or False?
4. How are the underlying tables affected by removing a view?
5. When should you create a subquery?
6. The ________________ set operator does *not* suppress duplicate values.
7. The NATURAL JOIN keywords can be used in the FROM clause of a query to join tables that have a single commonly named and defined column. True or False?
8. A(n) ________________ view does not have an ORDER BY or GROUP BY clause and does not include any computed values.

9. A(n) ______________ join requires the use of a table alias to mimic the referencing of two different tables, when in reality, they are the same table.
10. An outer join operator is used to indicate that unmatched rows should be included in the query results. True or False?

Multiple Choice

1. Which of the following types of joins include a row from table A if it has an equivalent value in the common column in table B?
   a. inner join
   b. natural join
   c. equijoin
   d. all of the above
2. Table A contains eight rows and table B contains four rows; the output of their Cartesian product includes ______________ rows.
   a. 32
   b. 12
   c. 8
   d. 4
3. Which of the following is a set operator?
   a. MINUS
   b. INTERSEPT
   c. SUM
   d. all of the above
4. Which line in the following query causes an error to occur?

```
SELECT f_last, f_first
FROM faculty
WHERE f_id =
     (SELECT f_id, super_id
       FROM faculty
       WHERE rank = 'Assistant');
```

   a. WHERE f_id =
   b. (SELECT f_id, super_id
   c. WHERE rank = ‘Assistant’);
   d. none of the above

5. The following statement is an example of what type of query?

```
SELECT s_first, s_last
FROM student
WHERE s_id IN
     (SELECT s_id
      FROM enrollment
      WHERE grade = 'A');
```

   a. inner query

   b. outer query

   c. self-join query

   d. nested query

6. The following statement is an example of what type of query?

```
SELECT s_first, s_last, grade, course_sec_id
FROM student s, enrollment e
WHERE s_s_id = e_sid;
```

   a. inner query

   b. outer query

   c. self-join query

   d. nested query

7. Which of the following commands removes the view named FAC from the database?

   a. `DROP Fac`;

   b. `DROP VIEW` Fac;

   c. `DELETE VIEW Fac`;

   d. `DELETE Fac`;

8. Which of the following is a valid statement?

   a. A view is a logical table that prevents a user from accessing the underlying tables.

   b. An updateable view can be used to delete data from its underlying table.

   c. A view based on only one table can be used to update calculated columns.

   d. all of the above

9. If you create an inner join between table A that contains six rows and table B that contains four rows, what is the maximum number of rows that can be included in the results?

   a. 0

   b. 4

   c. 6

   d. 24

10. Which of the following operators is used to indicate the inner table in an outer join query?
    a. (*)
    b. (+)
    c. (–)
    d. (^)

Problem-solving Cases

For all cases, use Notepad or another text editor to write a script using the specified filename. Always use the search condition text exactly as the case specifies. Place the queries in the order listed, and save the script files in your Chapter03\Cases folder on your Data Disk. If you haven't done so already, run the Ch3Clearwater.sql script in the Chapter03 folder on your Data Disk to create and populate the case study database tables. All of the following cases are based on the Clearwater Traders database.

1. List the name of each item included on order #1.
2. Determine which customer placed order #1 and the total amount of the order.
3. List the customers who have ordered Boy's Surf Shorts and identify how many were purchased.
4. Identify which shipments have not yet been received by Clearwater Traders and the items on each of those shipments.
5. Create a view named ITEM that displays the name of each item in inventory, its retail price, and the quantity on hand.
6. Using the ITEM view, determine the total amount of investment Clearwater Traders currently has in its inventory. (*Hint*: You should only receive one value from this query.)
7. Create a query that displays each shipment currently listed in the SHIPMENT table with its expected shipment date. For the shipments that have already been received, include actual received dates in the results.
8. Determine which customers have purchased the same items as customer #3.
9. Determine the total amount received for all sales of outdoor gear.
10. Determine the total amount of sales generated by the company's Web site.