CHAPTER 11 Insert, Update, and Delete

CHAPTER OBJECTIVES

In this chapter, you will learn about:

- Creating Data and Transaction Control
- Updating and Deleting Data
- The SQL Developer Data Tab

In Chapters 1 through 10, you learned what data is and how to query and present data. In this chapter, you will learn how to modify the data in tables with the INSERT, UPDATE, DELETE, and MERGE statements, which are known as Data Manipulation Language (DML).

These statements enable you to create, change, or delete data from tables. In Lab 11.1 you will learn about creating data in tables with the different INSERT command options and how to make these changes permanent. Lab 11.2 illustrates how to delete data and shows various ways to change existing data in the tables. Furthermore, you will learn about Oracle’s locking and read-consistency features. Lab 11.3 shows how you can use SQL Developer’s Data tab to manipulate data using the graphical user interface (GUI) screen functionality, which is useful when you need to modify a small number of records.

LAB 11.1 Creating Data and Transaction Control

LAB OBJECTIVES

After this lab, you will be able to:

- Insert Data
- Rollback and Commit Transactions

When you create new data in a table, you need to not only understand the syntax options but also know about referential integrity, column defaults, and data types. You can insert into a table one row at a time or write an INSERT statement that creates data in multiple tables. A major key to understanding SQL is the concept of transaction control, which ensures data consistency.

Inserting an Individual Row

The INSERT statement creates new data in a table. It can insert into a table a single row or multiple rows (based on a subquery).

The following INSERT statement inserts one individual row into the ZIPCODE table.

```
INSERT INTO zipcode
VALUES
   ('11111', 'Westerly', 'MA',
   USER, TO_DATE('18-JAN-2010',
'DD-MON-YYYY'),
   USER, SYSDATE)
```

When the statement is executed, Oracle responds with the following message.

```
1 row created.
```

The INSERT INTO keywords always precede the name of the table into which you want to insert data. The VALUES keyword precedes a set of parentheses that enclose the values you want to insert. For each of the seven columns of the ZIPCODE table there are seven corresponding values with matching data types in the INSERT statement separated by commas. The values in the list are in the same order as the columns when you use the DESCRIBE command on the ZIPCODE table. It is good practice to include a column list, though, in case of future table changes. Following is the INSERT statement with the column list.

```
INSERT INTO zipcode
   (zip, city, state,
   created_by, created_date,
   modified_by, modified_date)
VALUES
   ('11111', 'Westerly', 'MA',
   USER, TO_DATE('18-JAN-2010',
'DD-MON-YYYY'),
   USER, SYSDATE)
```

The syntax of the single-row, single-table INSERT statement is as follows.

```
INSERT INTO tablename [(column [,
column]...)]
VALUES (expression|DEFAULT,
[expression|DEFAULT]...)
```

Remember that the syntax convention is to enclose optional parts in brackets, denoted as [ ]. Keywords are in uppercase. The three dots (...) means that the expression can be repeated. The vertical bar denotes options, and the braces, { }, enclose items of which only one is required.

You notice from the INSERT statement into the ZIPCODE table that a text literal such as ‘Westerly’ is enclosed in single quotation marks and to insert a date requires the TO_DATE function with the format mask unless the date has the appropriate NLS_DATE_FORMAT value.

The INSERT statement uses the SYSDATE function to insert the current date and time into the MODIFIED_DATE column. Similar to the SYSDATE function, the USER function is another function that does not take a parameter. It returns the schema name of the user logged in—in this case, the value STUDENT. This value is inserted in the CREATED_BY and MODIFIED_BY columns. You see the result of the USER function in the following example.

```
SELECT USER
  FROM dual
USER
---------------
STUDENT

1 row selected.
```

Not all columns of the ZIPCODE table require values; only columns defined as NOT NULL. When you are not inserting data into all columns of a table, you must explicitly name the columns to insert data into. The following statement inserts values into just five of the seven columns in the ZIPCODE table; no data is inserted into the CITY and STATE columns.

```
INSERT INTO zipcode
   (zip, created_by, created_date,
   modified_by, modified_date)
VALUES
   ('11111', USER, SYSDATE, USER,
SYSDATE)
```

Some columns may have default values defined as part of their column definition. Not listing the column in the INSERT statement automatically places the default value in the column, or you can also explicitly use the keyword DEFAULT.

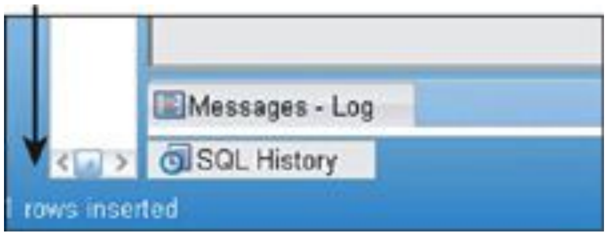
Alternatively, you can write the statement to omit the columns altogether and to insert NULL values in some of the columns instead.

```
INSERT INTO zipcode
VALUES
```

```
      ('11111', NULL, NULL, USER,
  SYSDATE, USER, SYSDATE)
```

When an INSERT command is successful, SQL Developer responds with a message on the left of the screen (see Figure 11.1). Later in the lab, you will see how to make this change permanent in the Oracle database by issuing a COMMIT command.

FIGURE 11.1 Message indicating a successfully issued INSERT statement



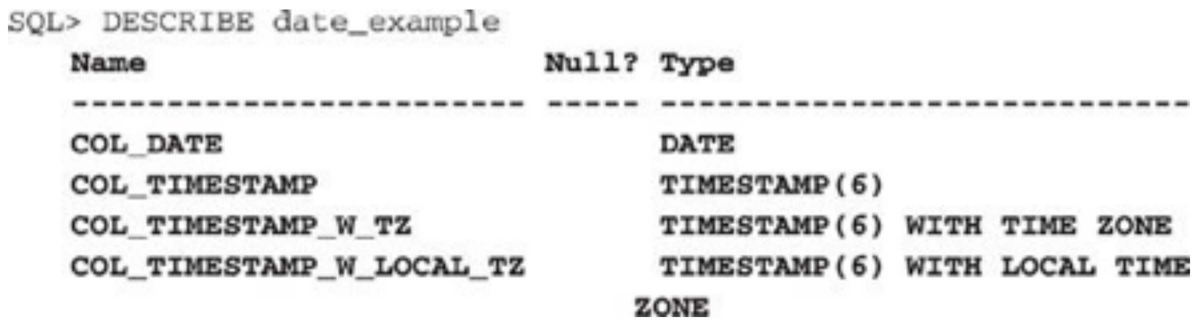
1 rows inserted

Data Types and Inserts

When inserting records, you always need to take the data type into consideration.

INSERTING DATES AND TIMES

To insert into a DATE data type column, you must specify the data in the required format for Oracle to understand and store the data. This is similar to using these literals in the WHERE clause of a SELECT statement. Following is the structure of the DATE_EXAMPLE table used in Chapter 5, “Date and Conversion Functions.”



```
SQL> DESCRIBE date_example
 Name                         Null? Type
 ---------------------------- ----- ------------------------------
 COL_DATE                           DATE
 COL_TIMESTAMP                      TIMESTAMP(6)
 COL_TIMESTAMP_W_TZ                 TIMESTAMP(6) WITH TIME ZONE
 COL_TIMESTAMP_W_LOCAL_TZ           TIMESTAMP(6) WITH LOCAL TIME
                                  ZONE
```

The next INSERT statement populates the table; it explicitly converts the literals, using the conversion functions, into the respective data types. The first column value is a DATE data type, and you use the TO_DATE function; the second value is a

TIMESTAMP data type, and the literal is converted to this data type with the TO_TIMESTAMP function. The third column value is of data type TIMESTAMP WITH TIME ZONE and uses the corresponding TO_TIMESTAMP_TZ function to convert to the correct data type. Finally, the fourth column is the date and time in the local time zone. There is no specific conversion function for this data type; it always displays the value in the local time.

```
INSERT INTO date_example
```

```
    (col_date,
    col_timestamp,
    col_timestamp_w_tz,
    col_timestamp_w_local_tz)
  VALUES
    (TO_DATE('24-FEB-2007 16:25:32',
      'DD-MON-YYYY HH24:MI:SS'),
    TO_TIMESTAMP('24-FEB-2007
  16:25:32.0000000',
      'DD-MON-YYYY HH24:MI:SS.FF'),
    TO_TIMESTAMP_TZ('24-FEB-2007
  16:25:32.0000000 -5:00',
      'DD-MON-YYYY HH24:MI:SS.FF
  TZH:TZM'),
    TO_TIMESTAMP('24-FEB-2007
  16:25:32.0000000',
      'DD-MON-YYYY HH24:MI:SS.FF'))
```

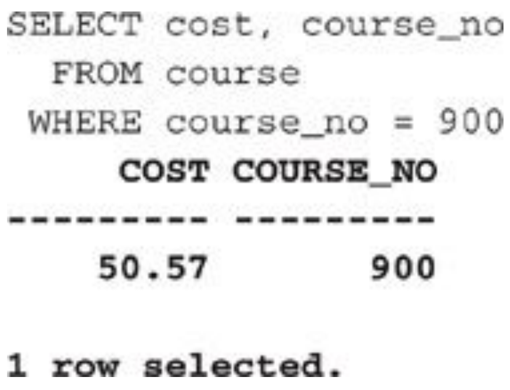
THE NUMBER DATA TYPE AND ROUNDING OF NUMBERS

The next statement attempts to insert a value that exceeds the scale of the COST column in the COURSE table. The COST column is defined as NUMBER(9,2), and the inserted value is 50.57499.

```
  INSERT INTO course
```

```
   (course_no, description, cost,
prerequisite,
   created_by, created_date,
modified_by, modified_date)
VALUES
   (900, 'Test Course', 50.57499,
NULL,
    'Your name', SYSDATE, 'Your
name', SYSDATE)
1 row created.
```

The INSERT statement proceeds successfully without any error, and the SELECT statement against the table reveals that Oracle rounds the number to 50.57.



```
SELECT cost, course_no
  FROM course
 WHERE course_no = 900
     COST COURSE_NO
--------- ---------
    50.57       900

1 row selected.
```

If the value exceeds the precision of the COST column, you get an error like the next message, where the precision is exceeded by one digit. The COST column 433 434 is defined as NUMBER(9,2) with a two-digits scale, thus allowing a maximum number of seven digits to the left of the decimal point.

```
INSERT INTO course
   (course_no, description, cost,
prerequisite,
   created_by, created_date,
modified_by, modified_date)
VALUES
   (901, 'Test Course',12345678,
NULL,
    'Your name', SYSDATE, 'Your
name', SYSDATE)
   (901, 'Test Course',12345678,
NULL,
*
ERROR at line 5:
ORA-01438: value larger than
specified precision allows for this
åcolumn
```

USING THE BINARY_FLOAT DATA TYPE AND INSERTING A FLOATING-POINT NUMBER

The following INSERT statement adds a floating-point number of the data type BINARY_FLOAT into the FLOAT_TEST table. A literal of the BINARY_FLOAT data type is followed by either an f or F.

```
INSERT INTO float_test
  (test_col)
VALUES
  (5f)
```

To indicate a BINARY_DOUBLE in a literal, follow it with d or D. You can also use the conversion functions TO_BINARY_FLOAT and TO_BINARY_DOUBLE to ensure the correct data type conversion.

If a value needs to be expressed as infinity or as not a number (NAN), you use the special literals BINARY_FLOAT_NAN, BINARY_DOUBLE_NAN, BINARY_FLOAT_INFINITY, and BINARY_DOUBLE_INFINITY.

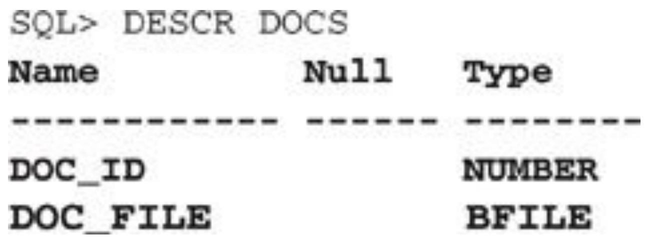
```
INSERT INTO float_test
  (test_col)
VALUES
  (BINARY_FLOAT_INFINITY)

SELECT *
  FROM float_test
TEST_COL
--------
5.0E+000
2.5E+000
     Nan
     Inf

4 rows selected.
```

INSERTING DATA INTO A BFILE DATA TYPE

The BFILE data type allows you to store the pointer to an external binary file, such as an image file. The actual file content is not stored within the database but in a specific file location or directory to which the pointer in the BFILE data type refers. You define this location of the files within Oracle by using the CREATE DIRECTORY command. (To issue this command successfully, you need the CREATE ANY DIRECTORY privileges. Chapter 15, "Security," discusses the granting of privileges.)

```
CREATE DIRECTORY my_docs AS 'c:
\ora_docs'
```

The DOCS table has a column called DOC_FILE with the BFILE data type.

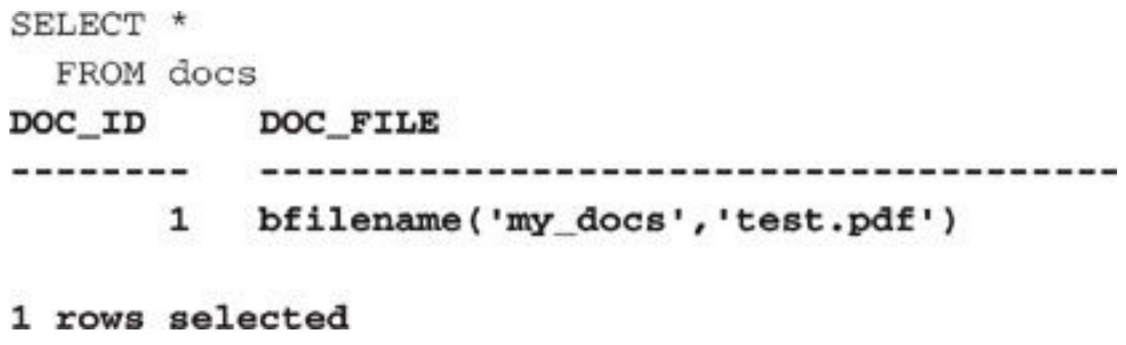


```
SQL> DESCR DOCS
Name         Null   Type
------------ ------ --------
DOC_ID              NUMBER
DOC_FILE            BFILE
```

To insert into the table, you must use the BFILENAME function with two parameters: the directory and the file name.

```
INSERT INTO docs VALUES
(1,
BFILENAME('my_docs','test.pdf'))
```

SQL*Plus shows the result as follows.



```
SELECT *
  FROM docs
DOC_ID     DOC_FILE
--------   -------------------------------------
       1   bfilename('my_docs','test.pdf')

1 rows selected
```

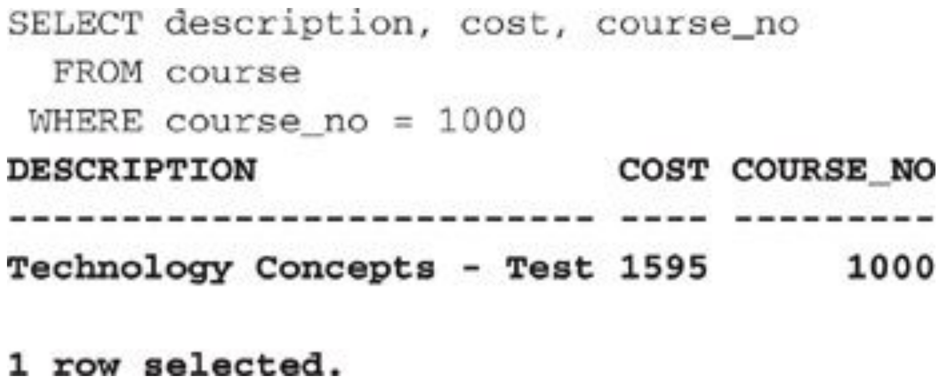
If you want to store multimedia content in the Oracle database, use the BLOB data type. You need to use a programming language and specialized functionality to accomplish this.

Inserts and Scalar Subqueries

A scalar subquery, which is defined as a subquery that returns a single row and column, is allowed within the VALUES clause of an INSERT statement. The following example shows two scalar subqueries: One inserts the description of COURSE_NO 10 and concatenates it with the word Test; the second scalar subquery inserts into the COST column the highest cost of any rows in the course table.

```
INSERT INTO course
  (course_no, description, cost,
  prerequisite, created_by,
created_date,
  modified_by, modified_date)
VALUES
  (1000, (SELECT description||' -
Test'
    FROM course
   WHERE course_no = 10),
  (SELECT MAX(cost)
  FROM course),
  20, 'MyName', SYSDATE,
  'MyName', SYSDATE)
```

Verify the result of the INSERT statement by querying the COURSE table for the COURSE_NO equal to 1000.



```
SELECT description, cost, course_no
  FROM course
 WHERE course_no = 1000
DESCRIPTION                COST COURSE_NO
-------------------------- ---- ---------
Technology Concepts - Test 1595      1000

1 row selected.
```

Inserting Multiple Rows

Another method for inserting data is to select data from another table via a subquery. The subquery may return one or multiple rows; thus, the INSERT statement inserts one or multiple rows at a time. Suppose there is a table called INTRO_COURSE in the STUDENT schema with columns similar to those in the COURSE table. The corresponding columns have compatible data types and column lengths; they do not have to have the same column names or column order. The following INSERT statement inserts data into the INTRO_COURSE table based on a query against the rows of the COURSE table. According to the subquery's WHERE clause, the only rows chosen are those where the course has no prerequisite.

```
   INSERT INTO intro_course
   (course_no, description_tx,
cost, prereq_no,
   created_by, created_date,
modified_by,
   modified_date)
SELECT course_no, description,
cost, prerequisite,
    created_by, created_date,
'Melanie',
    TO_DATE('01-JAN-2008', 'DD-MON-
YYYY')
   FROM course
  WHERE prerequisite IS NULL
```

The following is the syntax for a multiple-row INSERT based on a subquery.

```
INSERT INTO tablename [(column [,
column]...)]
subquery
```

Inserting into Multiple Tables

While most often you use a single-table, single-row INSERT command, you may occasionally need to insert rows into multiple tables simultaneously. This feature is useful when data is transferred from other system sources and the destination is a data warehouse system where the data is consolidated and denormalized for the purpose of providing end users simple query access to this data. You may also use a multitable INSERT command when you need to archive old data into separate tables.

Compared to executing multiple individual INSERT statements, using a multitable INSERT is not only faster but allows additional syntax options, providing further flexibility by enabling the conditional insert of data and perhaps eliminating the need to write specific programs. There are two different types of multitable inserts: INSERT ALL and INSERT FIRST. INSERT ALL can be divided into the unconditional INSERT and the conditional INSERT.

The next examples demonstrate multitable INSERT statements with the SECTION_HISTORY and the CAPACITY_HISTORY tables. You can add them to the STUDENT schema with the supplemental table scripts available from the companion Web site, at**www. oraclesqlbyexample.com**.

THE UNCONDITIONAL INSERT ALL STATEMENT

The INSERT statement chooses the sections that started more than one year ago and inserts these rows into both tables—SECTION_HISTORY and CAPACITY_HISTORY. There is no condition on the INSERT statement, other than the WHERE clause condition that determines the rows to be selected from the SECTION table.

```
INSERT ALL
  INTO section_history
   VALUES (section_id,
start_date_time, course_no,
section_no)
  INTO capacity_history
   VALUES (section_id, location,
capacity)
SELECT section_id, start_date_time,
course_no, section_no,
    location, capacity
  FROM section
  WHERE TRUNC(start_date_time) <
TRUNC(SYSDATE)-365
156 rows created.
```

THE CONDITIONAL INSERT ALL STATEMENT

The following statement chooses the same sections and inserts these rows into the tables, depending on whether the individual INSERT condition is satisfied. For example, for a SECTION_ID value of 130 and a capacity of 25, the statement enters the row in both tables. If only one of the conditions is true, it inserts the row only into the table with the true condition. If both conditions are false, the selected row is not inserted into either of the tables.

```
INSERTALL
 WHEN section_id BETWEEN 100 and
400 THEN
  INTO section_history
   VALUES (section_id,
start_date_time, course_no,
section_no)
 WHEN capacity >= 25 THEN
  INTO capacity_history
   VALUES (section_id, location,
capacity)
SELECT section_id, start_date_time,
course_no, section_no,
   location, capacity
  FROM section
 WHERE TRUNC(start_date_time) <
TRUNC(SYSDATE)-365
106 rows created.
```

The following is the syntax for the conditional INSERT ALL.

```
INSERT ALL
```

```
WHEN condition THEN
insert_clause [insert_clause...]
[WHEN condition THEN
insert_clause
[insert_clause...]...]
[ELSE
insert_clause [insert_clause...]]
(query)
```

The insert_clause syntax is defined as follows.

```
INTO tablename [(column [,
column]...)]
[VALUES (expression|DEFAULT,
[expression|DEFAULT]...)]
```

THE CONDITIONAL INSERT FIRST STATEMENT

The INSERT FIRST statement evaluates the WHEN clauses in order; if the first condition is true, the row is inserted, and subsequent conditions are no longer tested. For example, with a SECTION_ID value of 130 and a capacity of 25, the statement inserts the row in the SECTION_ HISTORY tables only because the first condition of the WHEN clause is satisfied. You can have an optional ELSE condition in case none of the conditions are true.

```
INSERTFIRST
 WHEN section_id BETWEEN 100 and
400 THEN
   INTO section_history
    VALUES (section_id,
start_date_time, course_no,
section_no)
 WHEN capacity >= 25 THEN
   INTO capacity_history
    VALUES (section_id, location,
capacity)
SELECT section_id, start_date_time,
course_no, section_no,
    location, capacity
   FROM section
 WHERE TRUNC(start_date_time) <
TRUNC(SYSDATE)-365
71 rows created.
```

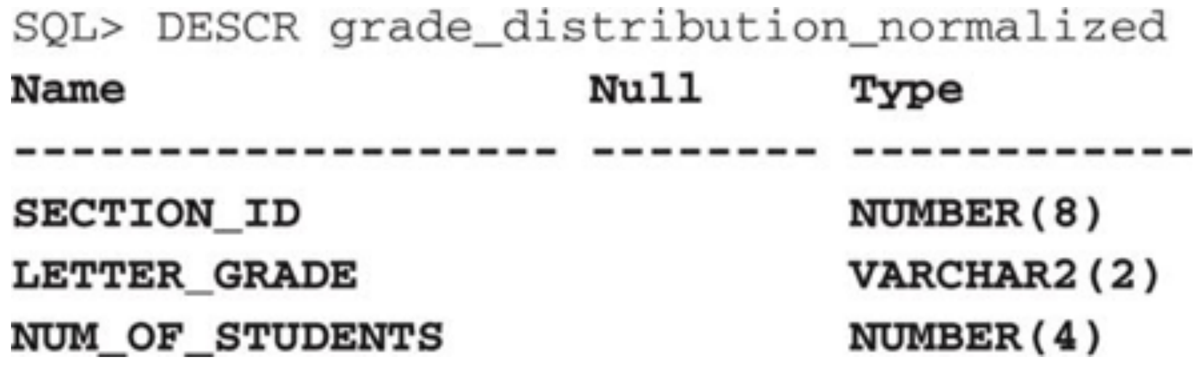
The syntax for the INSERT FIRST command is identical to that of the conditional INSERT ALL command except that you use the FIRST keyword instead of the ALL keyword.

PIVOTING INSERT ALL

The pivoting INSERT ALL statement is just like the unconditional INSERT ALL statement: It inserts the rows into multiple tables, and it also does not have a WHEN condition. The following is an example of pivoting a table (that is, flipping it on its side). The table used here, called GRADE_DISTRIBUTION, has a count of the different grades per each section. The first row, with SECTION_ID 400, shows 5 students with the letter grade A, 10 students with the letter grade B, 3 students with the letter grade C, and no D or F grades for any students in the section.

```
SELECT *
  FROM grade_distribution
SECTION_ID GRADE_A GRADE_B GRADE_C GRADE_D GRADE_F
---------- ------- ------- ------- ------- -------
       400       5      10       3       0       0
       401       1       3       5       1       0
       402       5      10       3       0       1

3 rows selected.
```

Suppose you want to move the data into a more normalized table format. In this case, you can use a pivoting INSERT ALL statement. The following example illustrates the insertion of the data into the table GRADE_DISTRIBUTION_NORMALIZED, which just lists the letter grade and the number of students. Here is the structure of the table.



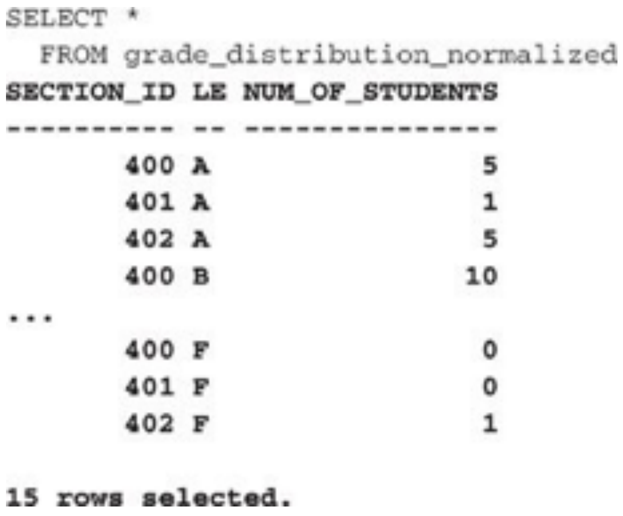
```
SQL> DESCR grade_distribution_normalized
Name                  Null     Type
--------------------- -------- ------------
SECTION_ID                     NUMBER(8)
LETTER_GRADE                   VARCHAR2(2)
NUM_OF_STUDENTS                NUMBER(4)
```

To insert the same data about SECTION_ID 400, five individual rows are needed. The following INSERT ALL statement transfers each individual selected row into the table, but in a normalized format whereby each grade is its own row.

```
INSERT ALL
  INTO
grade_distribution_normalized
   VALUES (section_id, 'A',
grade_a)
  INTO
grade_distribution_normalized
   VALUES (section_id, 'B',
grade_b)
  INTO
grade_distribution_normalized
```

```
   VALUES (section_id, 'C',
grade_c)
  INTO
grade_distribution_normalized
   VALUES (section_id, 'D',
grade_d)
  INTO
grade_distribution_normalized
   VALUES (section_id, 'F',
grade_f)
SELECT section_id, grade_a,
grade_b,
   grade_c, grade_d, grade_f
  FROM grade_distribution
15 rows created.
```

When selecting from the GRADE_DISTRIBUTION_NORMALIZED table, you see the rows in a normalized format.



```
SELECT *
  FROM grade_distribution_normalized
SECTION_ID LE NUM_OF_STUDENTS
---------- -- ---------------
       400 A                5
       401 A                1
       402 A                5
       400 B               10
...
       400 F                0
       401 F                0
       402 F                1

15 rows selected.
```

Transaction Control

Just as important as manipulating data is controlling when a change becomes permanent. DML statements are controlled within the context of a transaction. A transaction is a DML statement or group of DML statements that logically belong together, also referred to as a logical unit of work. The group of statements is defined by the commands COMMIT and ROLLBACK, in conjunction with the SAVEPOINT command.

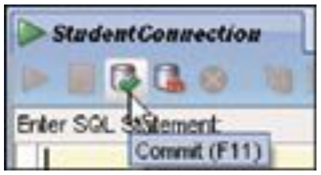
COMMIT

The COMMIT command makes a change to data permanent. Any previously uncommitted changes are now committed and cannot be undone. The effect of the COMMIT command is that it allows other sessions to see the data. The session issuing the DML command can always see the changes, but other sessions can see the changes only after you use COMMIT. Another effect of a commit is that locks for the changed rows are released, and other users can perform changes on the rows. You will learn more about locking in Lab 11.2.

Instead of typing the COMMIT command after your INSERT statement, you can click on the Commit icon in SQL Developer (see Figure 11.2).

Data Definition Language (DDL) statements, such as the CREATE TABLE command, or Data Control Language (DCL) statements, such as the GRANT command, implicitly issue a COMMIT to the database; there is no need to issue a COMMIT command. You'll learn about DDL commands in Chapter 12, "Create, Alter, and Drop Tables," and DCL commands in Chapter 15.

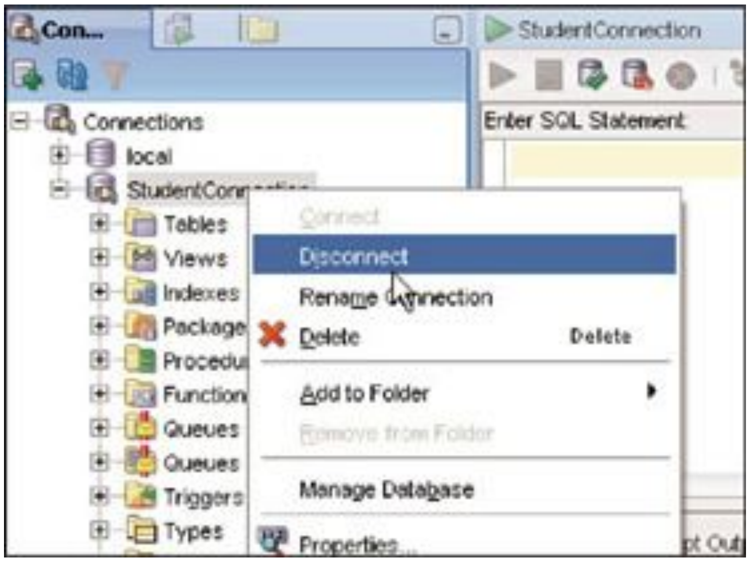
FIGURE 11.2 The COMMIT icon in SQL Developer



WHAT IS A SESSION?

A session is an individual connection to the Oracle database server. It starts as soon as the user is logged in and authenticated by the server with a valid login ID and password. The session ends when you explicitly disconnect, such as by using the Disconnect menu in SQL Developer (see Figure 11.3) or by exiting SQL Developer. In SQL*Plus, you log out by issuing a DISCONNECT command, using the EXIT command, or clicking the Close Window box.

FIGURE 11.3 Disconnecting from a session in SQL Developer

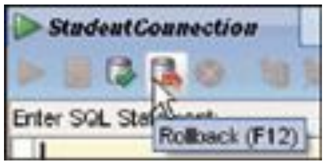


An individual database user may be connected to multiple concurrent sessions simultaneously. For example, if you invoke SQL Developer or SQL*Plus multiple times, each time, you establish an individual session.

ROLLBACK

The ROLLBACK command undoes any DML statements back to the last COMMIT command issued. Any pending changes are discarded, and any locks on the affected rows are released. In SQL Developer, you can click the Rollback icon (F12) instead of issuing the ROLLBACK command (see Figure 11.4).

FIGURE 11.4 The Rollback icon in SQL Developer



AN EXAMPLE OF A TRANSACTION

The following SQL statements all constitute a single transaction. The first INSERT statement starts the transaction, and the ROLLBACK command ends it.

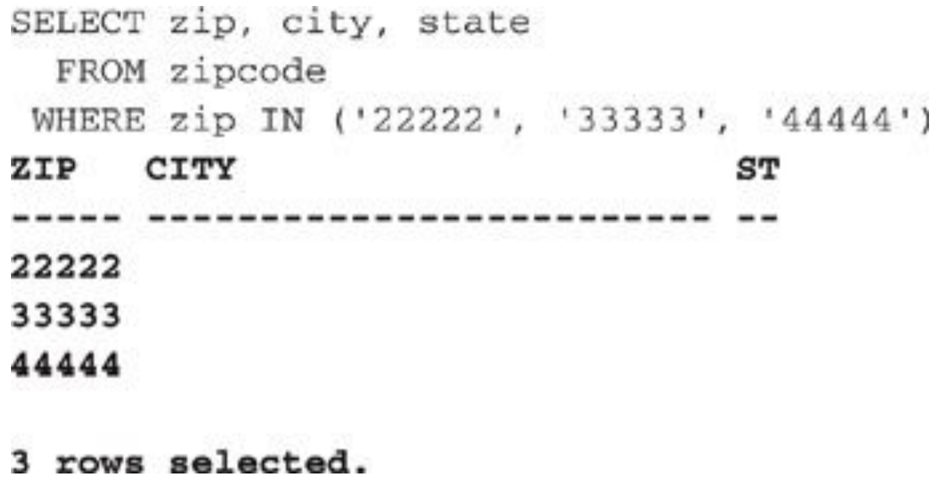
```
INSERT INTO zipcode
   (zip, city, state,
   created_by, created_date,
modified_by, modified_date)
VALUES
   ('22222', NULL, NULL,
   USER, SYSDATE, USER, SYSDATE)
1 row created.
```

```
INSERT INTO zipcode
   (zip, city, state,
   created_by, created_date,
modified_by, modified_date)
```

```
VALUES
   ('33333', NULL, NULL,
   USER, SYSDATE, USER, SYSDATE)
```
1 row created.

```
INSERT INTO zipcode
   (zip, city, state,
   created_by, created_date,
modified_by, modified_date)
VALUES
   ('44444', NULL, NULL,
   USER, SYSDATE, USER, SYSDATE)
```
1 row created.

The following is a query of the ZIPCODE table for the values inserted.



```
SELECT zip, city, state
  FROM zipcode
 WHERE zip IN ('22222', '33333', '44444')
ZIP   CITY                      ST
----- ------------------------- --
22222
33333
44444

3 rows selected.
```

442

443

Then, the following example issues the ROLLBACK command and performs the same query.

```
ROLLBACK
```

```
Rollback complete.
SELECT zip, city, state
  FROM zipcode
 WHERE zip IN ('22222', '33333',
'44444')
no rows selected
```

The values inserted are no longer in the ZIPCODE table; the ROLLBACK command prevents the values inserted by all three statements from being committed to the database. If a COMMIT command is issued between the first and second statements, the value ‘22222’ would be found in the ZIPCODE table, but not the values ‘33333’ and ‘44444’.

SAVEPOINT

The SAVEPOINT command allows you to save the results of DML transactions temporarily. The ROLLBACK command can then refer to a particular SAVEPOINT and roll back the transaction up to that point; any statements issued after the SAVEPOINT are rolled back.

The following example shows the same three DML statements used previously, but with SAVEPOINT commands issued in between.

```
INSERT INTO zipcode
   (zip, city, state,
   created_by, created_date,
modified_by, modified_date)
VALUES
   ('22222', NULL, NULL,
   USER, SYSDATE, USER, SYSDATE)
```
1 row created.

```
SAVEPOINT zip22222
```
Savepoint created.

```
INSERT INTO zipcode
   (zip, city, state,
   created_by, created_date,
modified_by, modified_date)
VALUES
   ('33333', NULL, NULL,
   USER, SYSDATE, USER, SYSDATE)
```
1 row created.

```
SAVEPOINT zip33333
```
Savepoint created.

```
INSERT INTO zipcode
   (zip, city, state,
   created_by, created_date,
modified_by, modified_date)
```

```
VALUES
   ('44444', NULL, NULL,
   USER, SYSDATE, USER, SYSDATE)
```
`1 row created.`

The next query checks the ZIPCODE table for the inserted values.

```
SELECT zip, city, state
  FROM zipcode
 WHERE zip IN ('22222', '33333', '44444')
ZIP   CITY                      ST
----- ------------------------- --
22222
33333
44444

3 rows selected.
```

Then, issue the command ROLLBACK TO SAVEPOINT zip33333 and perform the same query.

```
ROLLBACK TO SAVEPOINT zip33333
Rollback complete.

SELECT zip, city, state
  FROM zipcode
 WHERE zip IN ('22222', '33333', '44444')
ZIP   CITY                      ST
----- ------------------------- --
22222
33333

2 rows selected.
```

All statements issued after the zip33333 SAVEPOINT are rolled back. When you rollback to the previous SAVEPOINT, the same result occurs, and so on.

```
ROLLBACK TO SAVEPOINT zip22222
Rollback complete.

SELECT zip, city, state
  FROM zipcode
 WHERE zip IN ('22222', '33333', '44444')
ZIP   CITY                      ST
----- ------------------------- --
22222

1 row selected.
```

The three statements still constitute a single transaction; however, it is possible to mark parts of the transaction with savepoints in order to control how a statement is rolled back with the ROLLBACK TO SAVEPOINT command.

444

445

CONTROLLING TRANSACTIONS

It is important to control DML statements by using COMMIT, ROLLBACK, and SAVEPOINT. If the three previous statements logically belong together—in other words, one does not make sense without the others occurring—then another session should not see the results until all three are committed at once. Until the user performing the inserts issues a COMMIT command, no other database users or sessions are able to see the changes. A typical example of such a transaction is a transfer from a savings account to a checking account. You obviously want to avoid the scenario where transactions from one account are missing and the balances are out of sync. Unless both data manipulations are successful, the change does not become permanent and visible to other users.

Oracle places a lock on a row whenever the row is manipulated through a DML statement. This prevents other users from manipulating the row until it is either committed or rolled back. Users can continue to query the row and see the old values until the row is committed.

STATEMENT-LEVEL ROLLBACKS

If one individual statement fails in a series of DML statements, only this statement is rolled back, and Oracle issues an implicit SAVEPOINT. The other changes remain until a COMMIT or ROLLBACK occurs to end the transaction.

The following example shows two SQL statements. The first INSERT statement executes successfully, and the second fails.

```
INSERT INTO zipcode
  (zip, city, state,
  created_by, created_date,
modified_by, modified_date)
VALUES
  ('99999', NULL, NULL,
  USER, SYSDATE, USER, SYSDATE)
1 row created.
```

```
INSERT INTO zipcode
  (zip, city, state,
  created_by, created_date,
modified_by, modified_date)
VALUES
  (NULL, NULL, NULL,
  USER, SYSDATE, USER, SYSDATE)
INSERT INTO zipcode
*
ERROR at line 1:
ORA-01400: cannot insert NULL into
("STUDENT"."ZIPCODE"."ZIP")
```

The error message indicates the problem with the statement; it shows that a null value cannot be inserted into the ZIP column of the ZIPCODE table located in the STUDENT schema. Only the second statement is rolled back. The first statement remains intact and uncommitted, as you see when executing the next query. The entire transaction ends when a ROLLBACK or COMMIT occurs.

```
SELECT zip
  FROM zipcode
 WHERE zip = '99999'
ZIP
-----
99999

1 row selected.
```

LAB 11.1 EXERCISES

a) Write and execute an INSERT statement to insert a row into the GRADE_TYPE table for a grade type of 'Extra Credit', identified by the code 'EC'. Issue a COMMIT command afterward.

b) Explain what is wrong with the following INSERT statement. Hint: It is not the value course_no_seq.NEXTVAL, which inserts a value from a sequence, thus generating a unique number.

```
INSERT INTO course
   (course_no, description,
cost)
VALUES
   (course_no_seq.NEXTVAL,
'Intro to Linux', 1295)
```

c) Execute the following SQL statement. The SAMPLE clause chooses a random sample of 10 percent. Explain your observations and undo the change.

```
INSERT INTO instructor
   (instructor_id,
   salutation, first_name,
last_name,
   street_address, zip, phone,
   created_by, created_date,
modified_by, modified_date)
SELECT
instructor_id_seq.NEXTVAL,
    salutation, first_name,
last_name,
    street_address, zip, phone,
    USER, SYSDATE, USER, SYSDATE
   FROM student
SAMPLE (10)
```

d) Issue the following INSERT statements. Are the statements successful? If not, what do you observe?

```
INSERT INTO section
   (section_id, course_no,
section_no,
   start_date_time,
   location, instructor_id,
capacity, created_by,
   created_date, modified_by,
modified_date)
VALUES
   (500, 90, 1,
   TO_DATE('03-APR-2008 15:00',
'DD-MON-YYYY HH24:MI'),
   'L500', 103, 50, 'Your name
here',
   SYSDATE, 'Your name here',
SYSDATE)
```

```
INSERT INTO instructor
   (last_name, salutation,
instructor_id,
   created_by, created_date,
modified_by, modified_date)
VALUES
```

```
   ('Spencer', 'Mister', 200,
   'Your name', SYSDATE, 'Your
name', SYSDATE)
```

e) Insert the following row into the GRADE table and exit/log off SQL Developer or SQL*Plus without issuing a COMMIT statement. Log back in to the server and query the GRADE table for the inserted row. What do you observe?

```
INSERT INTO grade
   (student_id, section_id,
grade_type_code,
   grade_code_occurrence,
numeric_grade, created_by,
   created_date, modified_by,
modified_date)
VALUES
   (124, 83, 'MT',
   1, 90, 'MyName',
   SYSDATE, 'MyName', SYSDATE)
```

LAB 11.1 EXERCISE ANSWERS

a) Write and execute an INSERT statement to insert a row into the GRADE_TYPE table for a grade type of 'Extra Credit', identified by the code 'EC'. Issue a COMMIT command afterward.

ANSWER: All columns of the GRADE_TYPE table are identified as NOT NULL, so the INSERT statement needs to list all the columns and corresponding values.

```
INSERT INTO grade_type
   (grade_type_code,
description,
   created_by, created_date,
modified_by, modified_date)
VALUES
   ('EC', 'Extra Credit',
   USER, SYSDATE, USER, SYSDATE)
```
1 row created.

```
COMMIT
```
Commit complete.

It is not necessary to explicitly list the columns of the GRADE_TYPE table because values are supplied for all columns. However, it is good practice to name all the columns in the column list because if additional columns are added in the future or the order of columns in the table changes, the INSERT statement will fail. This is particularly important when the INSERT statement is used in a program for repeated use.

b) Explain what is wrong with the following INSERT statement. Hint: It is not the value course_no_seq.NEXTVAL, which inserts a value from a sequence, thus generating a unique number.

```
INSERT INTO course
   (course_no, description,
cost)
VALUES
   (course_no_seq.NEXTVAL,
'Intro to Linux', 1295)
```

ANSWER: The INSERT statement fails because it does not insert values into the NOT NULL columns CREATED_BY, CREATED_DATE, MODIFIED_BY, and MODIFIED_ DATE in the COURSE table.

```
INSERT INTO course
*
ERROR at line 1:
ORA-01400: cannot insert NULL
into
```

```
("STUDENT"."COURSE"."CREATED_BY"
)
```

The Oracle error message informs you that the column CREATED_BY requires a value. The correct command includes the NOT NULL columns and is successfully executed when issued as follows:

```
INSERT INTO course
   (course_no, description,
cost, created_date,
   modified_date, created_by,
modified_by)
VALUES
   (course_no_seq.NEXTVAL,
'Intro to Linux', 1295, SYSDATE,
   SYSDATE, 'AliceRischert',
'AliceRischert')
1 row created.
```

If you don't want to make this change permanent in the database, issue the ROLLBACK command.

```
ROLLBACK
Rollback complete.
```

The value supplied for the COURSE_NO column, course_no_seq.NEXTVAL, is not a text literal, number, or date. It is a value generated from a sequence called COURSE_NO_SEQ. A sequence is an Oracle database object that generates sequential numbers to ensure uniqueness whenever it is used, most commonly for generating primary keys. The keyword NEXTVAL indicates to Oracle to select the next value from the sequence. You'll learn more about sequences in Chapter 13, "Indexes, Sequences, and Views."

c) Execute the following SQL statement. The SAMPLE clause chooses a random sample of 10 percent. Explain your observations and undo the change.

```
INSERT INTO instructor
   (instructor_id,
   salutation, first_name,
last_name,
   street_address, zip, phone,
   created_by, created_date,
modified_by, modified_date)
SELECT
instructor_id_seq.NEXTVAL,
    salutation, first_name,
last_name,
    street_address, zip, phone,
    USER, SYSDATE, USER, SYSDATE
   FROM student
SAMPLE (10)
```

ANSWER: This is an example of a multirow INSERT statement. The INSERT statement contains a SELECT clause that retrieves values from all columns of the STUDENT table and inserts them into the INSTRUCTOR table. While there is no WHERE clause present in this SELECT statement, it contains the SAMPLE clause, which randomly chooses 10 percent of the students as rows for the insert operation. INSTRUCTOR_ID_SEQ is the name of the sequence that generates unique numbers. The pseudocolumn NEXTVAL retrieves the next value from the sequence. In this example, these generated unique numbers get inserted into the INSTRUCTOR_ID primary key column.

Be sure to undo the change afterward by using the ROLLBACK command.

```
ROLLBACK
Rollback complete.
```

d) Issue the following INSERT statements. Are the statements successful? If not, what do you observe?

```
INSERT INTO section
   (section_id, course_no,
section_no,
   start_date_time,
   location, instructor_id,
capacity, created_by,
   created_date, modified_by,
modified_date)
VALUES
   (500, 90, 1,
   TO_DATE('03-APR-2008 15:00',
'DD-MON-YYYY HH24:MI'),
   'L500', 103, 50, 'Your name
here',
   SYSDATE, 'Your name here',
SYSDATE)
```

```
INSERT INTO instructor
   (last_name, salutation,
instructor_id,
   created_by, created_date,
modified_by, modified_date)
VALUES
```

```
   ('Spencer', 'Mister', 200,
   'Your name', SYSDATE, 'Your
name', SYSDATE)
```

ANSWER: Both of the INSERT statements fail. You see the reason why after each individual statement is issued.

```
INSERT INTO section
   (section_id, course_no,
section_no,
   start_date_time,
   location, instructor_id,
capacity, created_by,
   created_date, modified_by,
modified_date)
VALUES
   (500, 90, 1,
   TO_DATE('03-APR-2008 15:00',
'DD-MON-YYYY HH24:MI'),
   'L500', 103, 50, 'Your name
here',
   SYSDATE, 'Your name here',
SYSDATE)
```

```
INSERT INTO section
*
ERROR at line 1:
```

```
ORA-02291: integrity constraint
(STUDENT.SECT_CRSE_FK)
violated - parent key not found
```

This statement fails because a parent row cannot be found. The foreign key constraint SECT_CRSE_FK is violated; this means a course number with the value 90 does not exist in the COURSE table, and thus the foreign key constraint prevents the creation of an orphan row. The constraint name is determined when you create a foreign key constraint, as discussed in Chapter 12. Ideally, you want to name the constraint so that it is apparent which columns and tables are involved. If you are unsure which column and table the constraint references, you can check the Constraints tab in SQL Developer or query the data dictionary view USER_CONSTRAINTS or 449

450 ALL_CONSTRAINTS, as discussed in Chapter 14,“The Data Dictionary, Scripting, and Reporting.” In this example, the constraint name is prefixed with the STUDENT schema name; this is unrelated to the STUDENT table name.

The next INSERT statement also fails because it attempts to insert a value that is larger than the defined five-character width of the SALUTATION column of the INSTRUCTOR table. The value 'Mister' is six characters long and therefore causes the following error message.

```
INSERT INTO instructor
   (last_name, salutation,
instructor_id,
   created_by, created_date,
modified_by, modified_date)
VALUES
   ('Spencer', 'Mister', 200,
   'Your name', SYSDATE, 'Your
name', SYSDATE)
INSERT INTO instructor
*
ERROR at line 1:
ORA-01401: inserted value too
large for column
```

USING SPECIAL CHARACTERS IN SQL STATEMENTS

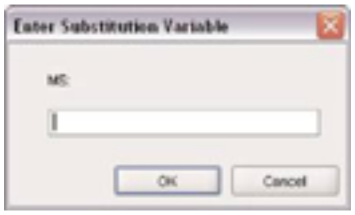
Some characters, such as the ampersand and the single quotation mark, have special meanings in SQL*Plus or within SQL statements.

THE AMPERSAND (&)

If you attempt to insert the following record, notice the message you receive. Oracle interprets any attempt to insert or update a column with a value containing an ampersand as a substitution variable and prompts you to enter a value. You will learn about this variable in Chapter 14. The & substitution parameter prompts for a value in Oracle's SQL*Plus and SQL Developer (see Figure 11.5).

```
INSERT INTO instructor
   (salutation, last_name,
instructor_id,
   created_by, created_date,
modified_by, modified_date)
VALUES
   ('Mr&Ms', 'Spencer', 300,
   'Your name', SYSDATE, 'Your
name', SYSDATE)
```

FIGURE 11.5 The Enter Substitution Variable dialog box



To temporarily turn off the substitution parameter functionality, you issue the SET DEFINE OFF command. Don’t forget to reset it back to its default value with the SET DEFINE ON command.

```
SQL>set define off
 SQL>INSERT INTO instructor
  (salutation, last_name,
instructor_id,
   created_by, created_date,
modified_by, modified_date)
VALUES
  ('Mr&Ms', 'Spencer', 300,
  'Your name', SYSDATE, 'Your
name', SYSDATE)
 /
 1 row created.
SQL> set define on
```

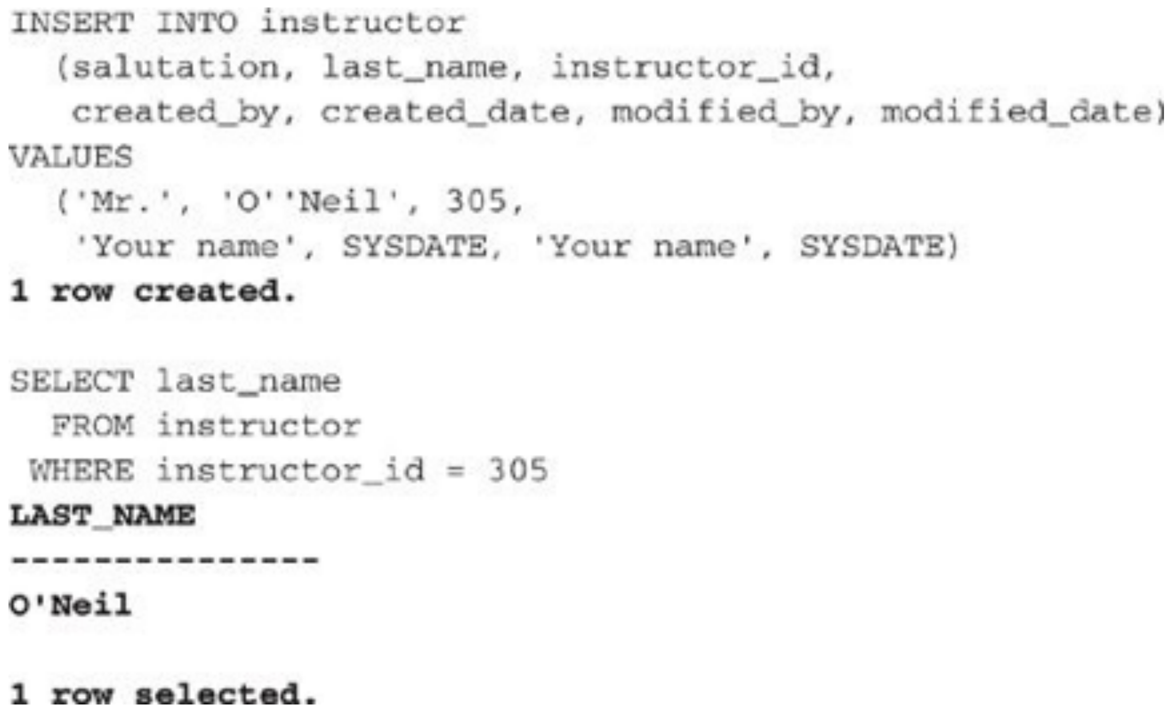
Alternatively, you can break the string into pieces and place the ampersand at the end of one string and then concatenate it with the remainder of the string.

```
INSERT INTO instructor
  (salutation, last_name,
instructor_id,
  created_by, created_date,
modified_by, modified_date)
```

```
VALUES
  ('Mr&'||'Ms', 'Spencer', 301,
  'Your name', SYSDATE, 'Your
name', SYSDATE)
```

THE SINGLE QUOTE (')

If you want to insert the instructor named O’Neil, you need to use a double set of single quotes to make Oracle understand that the single quote is to be taken as a literal quote.



```
INSERT INTO instructor
  (salutation, last_name, instructor_id,
   created_by, created_date, modified_by, modified_date)
VALUES
  ('Mr.', 'O''Neil', 305,
   'Your name', SYSDATE, 'Your name', SYSDATE)
1 row created.

SELECT last_name
  FROM instructor
 WHERE instructor_id = 305
LAST_NAME
---------------
O'Neil

1 row selected.
```

You can choose alternate quoting, as indicated with the letter q or Q. For example, the string 'O’’Neil' can be written as q'!O'Neil!', where the q indicates the alternate quoting mechanism. The letter q or Q follows a single quote and the chosen quote delimiter. In this example, the ! is the delimiter. The literal ends with the chosen delimiter and a single quote. As you see, O'Neil no longer requires two single quotes. You can choose as a delimiter any character except space, tab, and return. However, if the quote delimiter is [,{, <, or (, you must choose the corresponding closing delimiter.

```
INSERT INTO instructor
  (salutation, last_name,
instructor_id,
  created_by, created_date,
modified_by, modified_date)
VALUES
  ('Mr.', q'<O'Neil>', 305,
  'Your name', SYSDATE, 'Your
name', SYSDATE)
```

e) Insert the following row into the GRADE table and exit/log off SQL Developer or SQL*Plus without issuing a COMMIT statement. Log back in to the server and query the GRADE table for the inserted row. What do you observe?

```
INSERT INTO grade
  (student_id, section_id,
grade_type_code,
  grade_code_occurrence,
numeric_grade, created_by,
  created_date, modified_by,
modified_date)
VALUES
  (124, 83, 'MT',
  1, 90, 'MyName',
  SYSDATE, 'MyName', SYSDATE)
1 row created.
```

ANSWER: Depending on the program and how you exit each program, the data may or may not commit implicitly.

EXITING SQL DEVELOPER WITH UNCOMMITTED CHANGES

If you completely exit SQL Developer and then invoke the program again, you will not see the inserted row. SQL Developer assumes that unless you explicitly commit the data, the change should not occur.

If you disconnect from the session by using the Disconnect menu option available in the Connections pane (see Figure 11.6), you see the dialog box shown in Figure 11.7. It warns you that you have uncommitted changes and provides you with the choice to commit, roll back, or abort/cancel the disconnect operation.

FIGURE 11.6 Disconnecting from a SQL Developer session

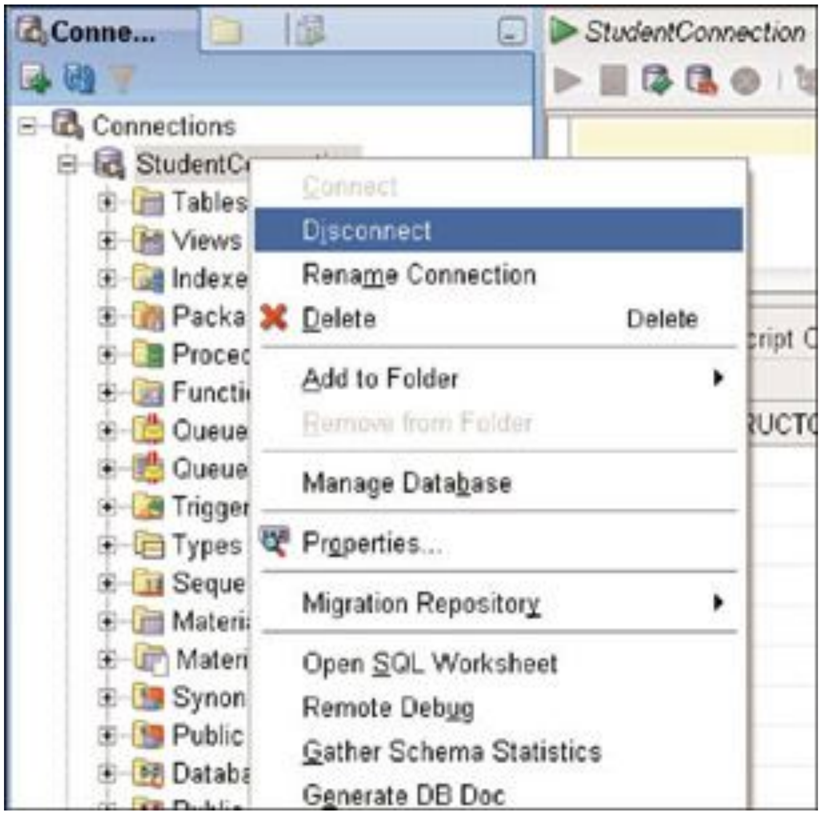
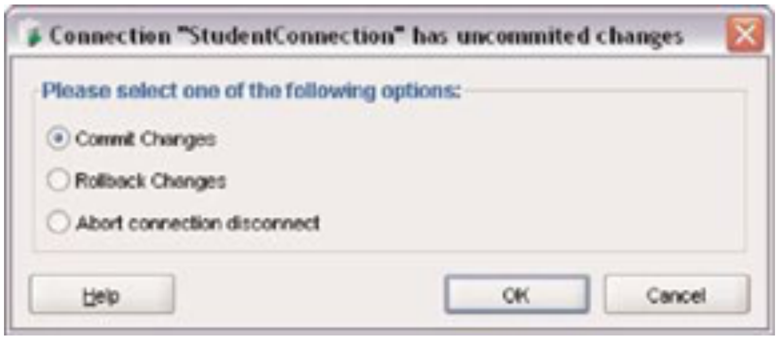


FIGURE 11.7 Warning upon disconnect of uncommitted changes

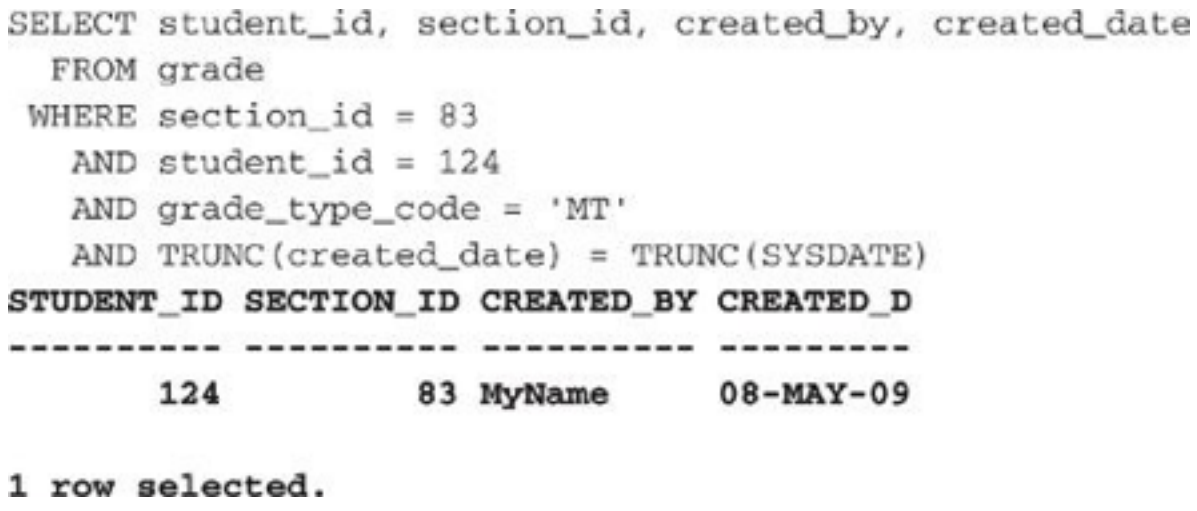

Connection "StudentConnection" has uncommited changes

Please select one of the following options:

- [✓ Commit Changes]
- Rollback Changes
- Abort connection disconnect

EXITING SQL*PLUS WITH UNCOMMITTED CHANGES

With SQL*Plus, after you log back in to the server and you query the GRADE table, the row exists, despite the missing COMMIT command. SQL*Plus implicitly issues the COMMIT when you correctly exit the program by typing the EXIT or DISCONNECT command.



```
SELECT student_id, section_id, created_by, created_date
  FROM grade
 WHERE section_id = 83
   AND student_id = 124
   AND grade_type_code = 'MT'
   AND TRUNC(created_date) = TRUNC(SYSDATE)
STUDENT_ID SECTION_ID CREATED_BY CREATED_D
---------- ---------- ---------- ---------
       124         83 MyName     08-MAY-09

1 row selected.
```

The implicit commit behavior is part of Oracle’s SQL*Plus program. You should explicitly commit or rollback your transactions. If you exit from either SQL Developer or SQL*Plus by clicking the CLOSE button in the window, the INSERT statement does not commit to the database. This is considered an abnormal exit, and modified rows are locked.

LOCKING OF ROWS THROUGH ABNORMAL TERMINATION

Rows may become locked when a session abnormally terminates, such as when a user reboots the machine without properly exiting or when the application program connected to the database raises an unhandled exception. If you do not exit your session properly, an uncommitted transaction may be pending, and the row is locked until Oracle eventually detects the dead session and rolls back the transaction. You can verify if in fact a lock is held on a particular row and table by querying the Oracle data dictionary view or using the SQL Developer Tools, Monitor Sessions menu, which is discussed in Lab 11.2. Sometimes, the database administrator (DBA) must intervene and manually release the lock if Oracle does not resolve the problem automatically.

Clean exits and frequent commits are some good habits that you should adopt; otherwise, locks will not be released, and other users cannot make modifications to the same rows you changed.

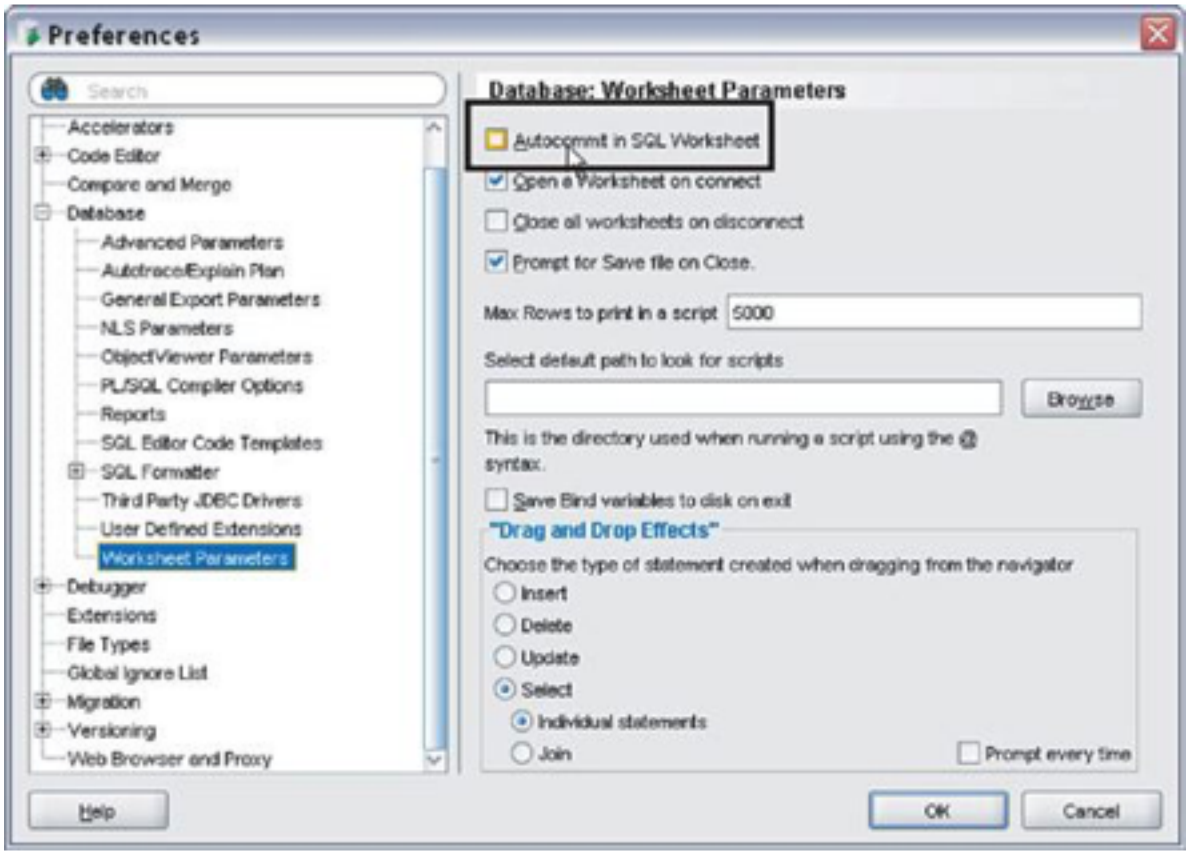
AUTOCOMMIT

In SQL Developer and SQL*Plus, you can choose to automatically commit every statement without allowing for rollbacks. In SQL Developer, you need to select Tools, Preferences, Database, Worksheet Parameters and then check the option Autocommit in SQL Worksheet (see Figure 11.8). The settings in the Preferences dialog in SQL Developer are valid for the current and all future sessions.

For SQL*Plus, you can use the command AUTOCOMMIT during a SQL*Plus session by typing SET AUTOCOMMIT ON or set AUTOCOMMIT IMMEDIATE. This command is valid only for the duration of the SQL*Plus session. (You can create a glogin.sql file that modifies session settings whenever you log in. You'll learn more about this in Appendix C, "SQL*Plus Command Reference.")

Clearly, the AUTOCOMMIT command and the corresponding menu option in SQL Developer are dangerous. A ROLLBACK command issued during that session has no effect because every statement has already automatically been committed.

FIGURE 11.8 Worksheet Parameters screen in SQL Developer



Database: Worksheet Parameters

☐ Autocommit in SQL Worksheet
☑ Open a Worksheet on connect
☐ Close all worksheets on disconnect
☑ Prompt for Save file on Close.

Max Rows to print in a script: 5000

Select default path to look for scripts:

This is the directory used when running a script using the @ syntax.

☐ Save Bind variables to disk on exit

"Drag and Drop Effects"

Choose the type of statement created when dragging from the navigator

○ Insert
○ Delete
○ Update
● Select
● Individual statements
○ Join
☐ Prompt every time

Lab 11.1 Quiz

In order to test your progress, you should be able to answer the following questions.

1) A DML command automatically issues a COMMIT.

_______**a)** True

_______**b)** False

2) A statement-level rollback ends a transaction.

_______**a)** True

_______**b)** False

3) An INSERT statement can insert only one row at a time into a table.

_______**a)** True

_______**b)** False

4) A COMMIT or ROLLBACK command ends a transaction.

_______**a)** True

_______**b)** False

5) Uncommitted changes can be seen by all users.

_______**a)** True

______**b)** False

6) A transaction is a logical unit of work.

______**a)** True

______**b)** False

ANSWERS APPEAR IN APPENDIX A.

LAB 11.2 Updating and Deleting Data

LAB OBJECTIVES

After this lab, you will be able to:

- Update, Delete, and Merge Data
- Understand the Effect of Data Manipulation on Other Users

The UPDATE, DELETE, and MERGE commands manipulate data. A database is usually shared by many users. At times, they contend for the same information simultaneously. It is helpful to understand how data manipulation affects your and other users' ability to change and query the data in a read-consistent way.

Updating Data

The UPDATE command manipulates existing data in a table. It always refers to a single table. For example, the following UPDATE statement updates the FINAL_GRADE column in the ENROLLMENT table to 90 for all students who enrolled in January 2007.

```
UPDATE enrollment
  SET final_grade = 90
WHERE enroll_date >=
TO_DATE('01/01/2007', 'MM/DD/YYYY')
  AND enroll_date <
TO_DATE('02/01/2007', 'MM/DD/YYYY')
11 rows updated.
```

The keyword UPDATE always precedes the name of the table to be updated, and the SET keyword precedes the column or columns to be changed. An UPDATE statement can update all rows in a table at once or just certain rows when restricted with a WHERE clause, as in the previous example. The general syntax for the UPDATE command is as follows.

```
UPDATE tablename
SET
{{(column[,column]...)=(subquery)|
   column={expression|(subquery)|
DEFAULT}
  }[,
{(column[,column]...)=(subquery)|
   column={expression|(subquery)|
DEFAULT}
  }]...}
[WHERE condition]
```

Updating Columns to Null Values

An UPDATE statement can update columns with a NULL value. The following UPDATE statement sets the FINAL_GRADE column to NULL for all rows in the ENROLLMENT table because there is no WHERE clause.

```
UPDATE enrollment
  SET final_grade = NULL
```

The IS NULL operator is used in a WHERE clause, not in the SET clause of an UPDATE statement.

Column Default Value

A column may have a default value defined; this value is entered when an INSERT statement did not specify an explicit value for a column. You can use the DEFAULT keyword in the UPDATE or INSERT command to explicitly set the default value defined for the column. The NUMERIC_GRADE column of the GRADE table has a default value of 0 defined. Examine the row before the change to the DEFAULT value.

```
UPDATE enrollment
   SET final_grade = 90
 WHERE enroll_date >= TO_DATE('01/01/2007', 'MM/DD/YYYY')
   AND enroll_date < TO_DATE('02/01/2007', 'MM/DD/YYYY')
11 rows updated.
```

To update the column to the default value of 0 for the first homework grade of student ID 211 in SECTION_ID 141, you issue the following UPDATE command.

```
UPDATE grade
  SET numeric_grade = DEFAULT
WHERE student_id = 211
  AND section_id = 141
  AND grade_type_code = 'HM'
  AND grade_code_occurrence = 1
1 row updated.
```

Examine the result of the change by requerying the record. The column default value of 0 is now shown.

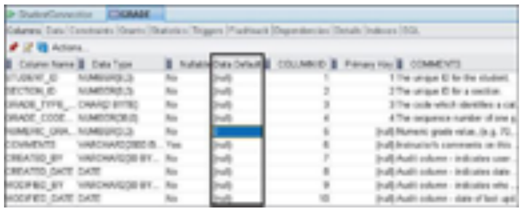
```
SELECT numeric_grade
  FROM grade
 WHERE student_id = 211
   AND section_id = 141
   AND grade_type_code = 'HM'
   AND grade_code_occurrence = 1
NUMERIC_GRADE
-------------
            0

1 row selected.
```

Now restore the value to the original value of 99 with the ROLLBACK command.

```
ROLLBACK
Rollback complete.
```

If you want to find out which columns have column default values, review the Data Default column on the Columns tab of the GRADE table (see Figure 11.9).

FIGURE 11.9 Data Default value on the GRADE table



StudentConnection | GRADE

Columns | Data | Constraints | Grants | Statistics | Triggers | Flashback | Dependencies | Details | Indexes | SQL

Actions...

Column Name	Data Type	Nullable	Data Default	COLUMN ID	Primary Key	COMMENTS
STUDENT_ID	NUMBER(8,0)	No	(null)	1	1	The unique ID for the student
SECTION_ID	NUMBER(8,0)	No	(null)	2	2	The unique ID for a section
GRADE_TYPE_...	CHAR(2 BYTE)	No	(null)	3	3	The code which identifies a ca
GRADE_CODE...	NUMBER(38,0)	No	(null)	4	4	The sequence number of one g
NUMERIC_GRA...	NUMBER(3,0)	No	0	5	(null)	Numeric grade value, (e.g. 70, ...
COMMENTS	VARCHAR2(2000 B...	Yes	(null)	6	(null)	Instructor's comments on this ...
CREATED_BY	VARCHAR2(30 BY...	No	(null)	7	(null)	Audit column - indicates user ...
CREATED_DATE	DATE	No	(null)	8	(null)	Audit column - indicates date ...
MODIFIED_BY	VARCHAR2(30 BY...	No	(null)	9	(null)	Audit column - indicates who ...
MODIFIED_DATE	DATE	No	(null)	10	(null)	Audit column - date of last upd

You can also query the data dictionary view USER_TAB_COLUMNS or ALL_TAB_COLUMNS, as discussed in detail in Chapter 14.

You will learn about the syntax for creating column defaults in Chapter 12.

Updates and the CASE Expression

CASE expressions can be used anywhere expressions are allowed. The following example shows the CASE expression in the SET clause of the UPDATE statement. The FINAL_GRADE column of the ENROLLMENT table is updated so that students enrolled in SECTION_ID 100 receive extra points for their FINAL_GRADE score.

```
UPDATE enrollment
 SET final_grade = CASE WHEN
final_grade <=80 THEN
   final_grade+5
  WHEN final_grade > 80 THEN
   final_grade+10
  END
WHERE section_id = 100
```

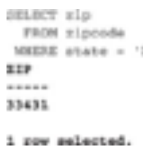
The CASE expression evaluates the current value of the FINAL_GRADE column. If the value is less than or equal to 80, the value of FINAL_GRADE is increased by 5 points; if the value is greater than 80, the increase is 10 points. No provision is made for null values; they remain unchanged because they do not satisfy any of the WHEN conditions. A null value is not greater than, less than, or equal to any value, and there is no ELSE clause in this statement.

Subqueries and the UPDATE Command

An update can occur based on data from other tables, using a subquery. The following example uses a subquery in the SET clause of the UPDATE command, and it updates the ZIP column of INSTRUCTOR_ID 108 to be equal to the ZIP value of the state of Florida.

```
UPDATE instructor
 SET zip = (SELECT zip
   FROM zipcode
  WHERE state = 'FL')
WHERE instructor_id = 108
```

In our ZIPCODE table, the state of Florida has a single value in the ZIPCODE table.



```
SELECT zip
  FROM zipcode
 WHERE state = 'FL'
ZIP
-----
33431

1 row selected.
```

The result of the update effectively changes the zip code to 33431 for INSTRUCTOR_ID 108.

```
INSTRUCTOR_ID ZIP
------------- -----
          108 33431

1 row selected.
```

SUBQUERIES THAT RETURN NULL VALUES

The following UPDATE query statement attempts to update the same instructor's zip code with a value for which you will not find any zip code in the ZIPCODE table.

```
UPDATE instructor
 SET zip = (SELECT zip
   FROM zipcode
  WHERE state = 'CA')
WHERE instructor_id = 108
1 row updated.
```

When you issue the query to see the effect of the update, the subquery returns a null value; it therefore updated the ZIP column to a null.

```
SELECT instructor_id, zip
  FROM instructor
 WHERE instructor_id = 108
INSTRUCTOR_ID ZIP
------------- ---
          108

1 row selected.
```

SUBQUERIES THAT RETURN MULTIPLE VALUES

The following subquery returns multiple zip codes for the state of Connecticut. The error message indicates that the subquery returns multiple rows, which is not allowed for an equal sign (=); therefore, the UPDATE statement fails.

```
UPDATE instructor
 SET zip = (SELECT zip
   FROM zipcode
  WHERE state = 'CT')
WHERE instructor_id = 108
 SET zip = (SELECT zip
*
ERROR at line 2:
ORA-01427: single-row subquery
returns more than one row
```

If you want just any one of the zip codes, no matter which one, you can use the MAX or MIN function. An aggregate function guarantees the return of a single row.

```
UPDATE instructor
 SET zip = (SELECT MAX(zip)
   FROM zipcode
```

```
  WHERE state = 'CT')
WHERE instructor_id = 108
```
1 row updated.

UPDATES AND CORRELATED SUBQUERIES

The following statement updates the FINAL_GRADE column to 90 and the MODIFIED_DATE column to March 13, 2009, for those sections taught by the instructor Hanks.

```
UPDATE enrollment e
 SET final_grade = 90,
  modified_date = TO_DATE('13-
MAR-2009', 'DD-MON-YYYY')
WHERE EXISTS
  (SELECT '*'
   FROM section s, instructor i
  WHERE e.section_id = s.section_id
   AND s.instructor_id =
i.instructor_id
   AND i.last_name = 'Hanks')
```

You can use any of the SELECT statements you have learned about to restrict the result set. In this example, a correlated subquery identifies the rows to be updated. A column from the outer table, in this case

ENROLLMENT, is referenced in the subquery through the column E.SECTION_ID. Every row of the ENROLLMENT table is updated where a corresponding SECTION_ID is returned by the subquery. As in the other correlated subqueries, every row in the outer table, here the ENROLLMENT table, is examined and evaluated against the inner query. The update occurs for those rows where the condition of the correlated subquery evaluates to true.

AVOIDING A COMMON SCENARIO WITH CORRELATED SUBQUERIES

The following correlated update, which involves tables TA and TB, changes one column with a value from another table. The values from TA need to be updated to reflect changes made in TB. The query shows a list of all the rows in the TA table.

```
SELECT *
  FROM ta
       ID COL1
--------- ----
        1 a
        2 b
        3 c
        4 d

4 rows selected.
```

This is a list of all the rows in table TB. The idea of the correlated update is to update the rows of TA based on table TB by joining on the common column called ID.

```
SELECT *
  FROM tb
       ID COL2
--------- ----
        1 w
        2 x
        5 y
        6 z

4 rows selected.
```

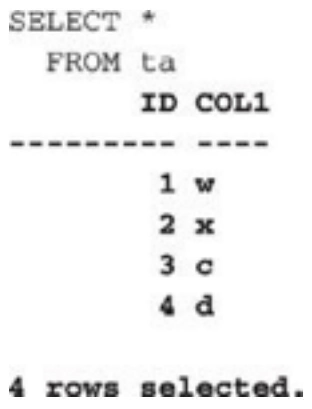
When you execute the UPDATE statement and subsequently query table TA, the rows with IDs 3 and 4 are updated with null values. The intention is to retain the original values.

```
UPDATE ta
   SET col1 = (SELECT col2
                 FROM tb
                WHERE ta.id = tb.id)
4 rows updated.

SELECT *
  FROM ta
       ID COL1
--------- ----
        1 w
        2 x
        3
        4

4 rows selected.
```

The correlated update query does not have a WHERE clause; therefore, all the rows of table TA are evaluated. The correlated subquery returns a null value for any row that was not found in table TB. You can avoid this behavior and retain the values in COL1 by including only the rows found in table TB with an appropriate WHERE clause in the UPDATE statement.

```
ROLLBACK
Rollback complete.

UPDATE ta
 SET col1 = (SELECT col2
   FROM tb
   WHERE ta.id = tb.id)
WHERE id IN (SELECT id
   FROM tb)
2 rows updated.
```

A query against the TB table verifies that the desired updates are done correctly.

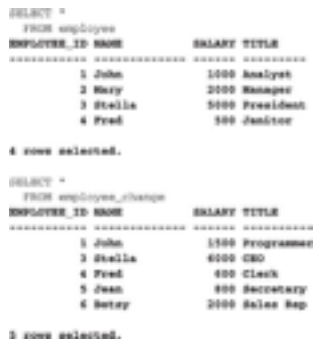


```
SELECT *
  FROM ta
       ID COL1
--------- ----
        1 w
        2 x
        3 c
        4 d

4 rows selected.
```

Be sure to check your results before committing, especially when you perform complicated updates to a table.

UPDATES AND SUBQUERIES THAT RETURN MULTIPLE COLUMNS

The following example uses the tables EMPLOYEE and EMPLOYEE_CHANGE. The EMPLOYEE table contains a list of employees, with their IDs, names, salaries, and titles. The purpose of the EMPLOYEE_CHANGE table is to hold all the changes that need to be made to the EMPLOYEE table. Perhaps the names, titles, and salary information comes from various other systems and are then recorded in the EMPLOYEE_CHANGE table that is to be used for updates to the master EMPLOYEE table.

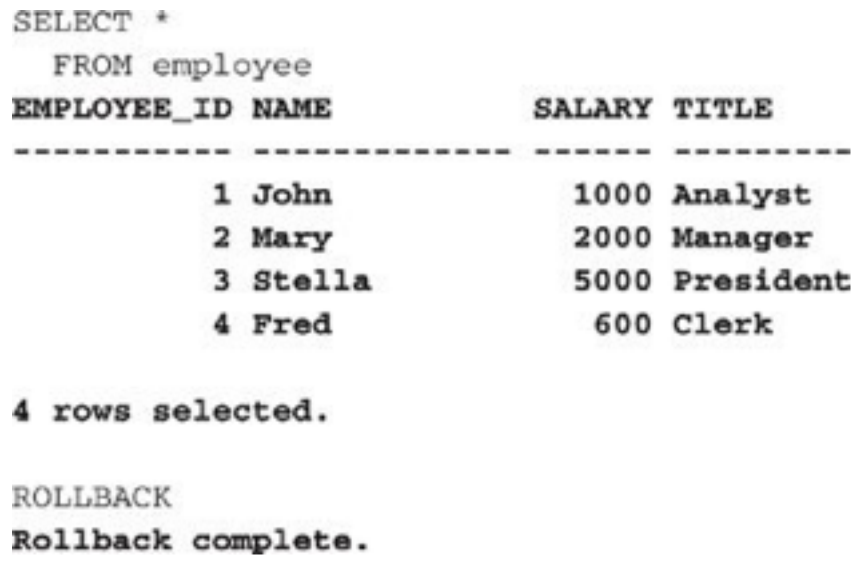


```
SELECT *
  FROM employee
EMPLOYEE_ID NAME          SALARY TITLE
----------- ------------- ------ ---------
          1 John            1000 Analyst
          2 Mary            2000 Manager
          3 Stella          5000 President
          4 Fred             500 Janitor

4 rows selected.

SELECT *
  FROM employee_change
EMPLOYEE_ID NAME          SALARY TITLE
----------- ------------- ------ ----------
          1 John            1500 Programmer
          3 Stella          6000 CEO
          4 Fred             400 Clerk
          5 Jean             800 Secretary
          6 Betsy           2000 Sales Rep

5 rows selected.
```

The next statement updates both the SALARY and TITLE columns of the EMPLOYEE table with the corresponding values from the EMPLOYEE_CHANGE table for the employee with ID 4, which is Fred the Janitor. In the subquery of this UPDATE statement, the equal sign indicates that the subquery must return a single row.

```
UPDATE employee
 SET (salary, title) = (SELECT
salary, title
   FROM employee_change
   WHERE employee_id = 4)
WHERE employee_id = 4
```
1 row updated.

You now see the change, and Fred now earns a different salary and has the title Clerk.



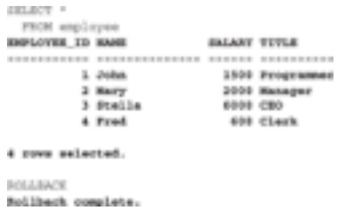
```
SELECT *
  FROM employee
EMPLOYEE_ID NAME         SALARY TITLE
----------- ------------ ------ ---------
          1 John           1000 Analyst
          2 Mary           2000 Manager
          3 Stella         5000 President
          4 Fred            600 Clerk

4 rows selected.

ROLLBACK
Rollback complete.
```

```
ROLLBACK
Rollback complete.
```

Undo the change with the ROLLBACK command. The next example shows how to update all the rows in the EMPLOYEE table instead of just one individual employee.

```
UPDATE employee e
 SET (salary, title) =
  (SELECT salary, title
   FROM employee_change c
  WHERE e.employee_id =
c.employee_id)
WHERE employee_id IN (SELECT
employee_id
   FROM employee_change)
3 rows updated.
```

Three rows are updated—for employees John, Stella, and Fred. The records for employees Jean and Betsy are not inserted into the EMPLOYEE table because the UPDATE statement just updates existing records and does not insert any new rows.



```
SELECT *
  FROM employee
EMPLOYEE_ID NAME                 SALARY TITLE
=========== ==================== ====== ==========
          1 John                   1500 Programmer
          2 Mary                   2000 Manager
          3 Stella                 6000 CEO
          4 Fred                    600 Clerk

4 rows selected.

ROLLBACK
Rollback complete.
```

MERGE: Combining INSERTs, UPDATEs, and DELETEs

You can perform combined INSERT, UPDATE, and DELETE operations with the MERGE command, using the following syntax.

```
MERGE INTO tablename
USING {query|tablename} ON
(condition)
[WHEN MATCHED THEN UPDATE
set_clause
[DELETE condition]]
[WHEN NOT MATCHED THEN INSERT
values_clause]
```

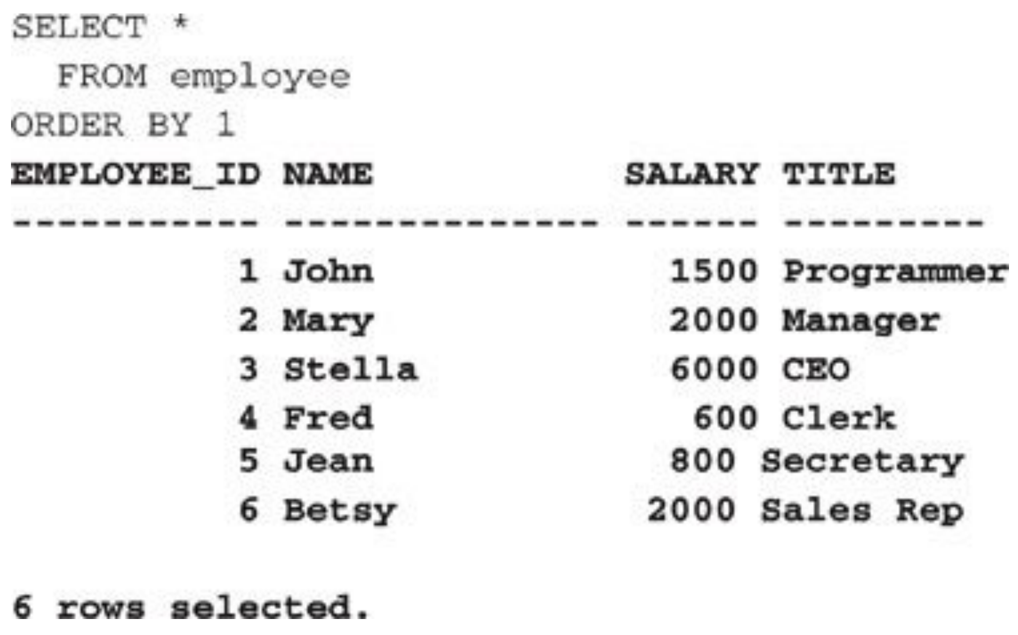
The table EMPLOYEE_CHANGE contains two additional rows, Jean and Betsy, that are not found in the EMPLOYEE table. The MERGE statement allows you to update the matching rows and lets you insert the rows found in the EMPLOYEE_CHANGE table but missing from the EMPLOYEE table.

```
MERGE INTO employee e
USING (SELECT employee_id, salary,
title, name
   FROM employee_change) c
```

```
  ON (e.employee_id =
c.employee_id)
WHEN MATCHED THEN
  UPDATE SET e.salary = c.salary,
   e.title = c.title
WHEN NOT MATCHED THEN
  INSERT (e.employee_id, e.salary,
e.title, e.name)
  VALUES (c.employee_id, c.salary,
c.title, c.name)
```

5 rows merged.

When you query the EMPLOYEE table, you see the changed values and the addition of the employees Jean and Betsy. Mary did not have a record in the EMPLOYEE_CHANGE table; therefore, no modification to her record is performed.



```
SELECT *
  FROM employee
ORDER BY 1
EMPLOYEE_ID NAME          SALARY TITLE
----------- ------------- ------ ---------
          1 John            1500 Programmer
          2 Mary            2000 Manager
          3 Stella          6000 CEO
          4 Fred             600 Clerk
          5 Jean            800 Secretary
          6 Betsy          2000 Sales Rep

6 rows selected.
```

The MERGE syntax contains an optional DELETE condition to the WHEN MATCHED THEN UPDATE clause. It allows you to remove rows from the table during this operation. The only rows deleted are the ones that satisfy both the DELETE and the ON conditions. The DELETE condition evaluates the rows based on the values after the update—not the original values. The next statement adds the DELETE condition, which effectively deletes Stella from the EMPLOYEE table because her SALARY column value now equals 6000.

```
ROLLBACK
Rollback complete.

MERGE INTO employee e
 USING (SELECT employee_id, salary, title, name
          FROM employee_change) c
    ON (e.employee_id = c.employee_id)
 WHEN MATCHED THEN
   UPDATE SET e.salary = c.salary,
              e.title = c.title
   DELETE WHERE salary = 6000
 WHEN NOT MATCHED THEN
   INSERT (e.employee_id, e.salary, e.title, e.name)
   VALUES (c.employee_id, c.salary, c.title, c.name)
5 rows merged.

SELECT *
  FROM employee
ORDER BY 1
EMPLOYEE_ID NAME           SALARY TITLE
----------- ------------- ------ ----------
          1 John            1500 Programmer
          2 Mary            2000 Manager
          4 Fred             600 Clerk
          5 Jean             800 Secretary
          6 Betsy           2000 Sales Rep

5 rows selected.
```

Deleting Data

You remove data from a table with the DELETE statement. It can delete all rows or just specific rows. The syntax is as follows.

```
DELETE FROM tablename
  [WHERE condition]
```

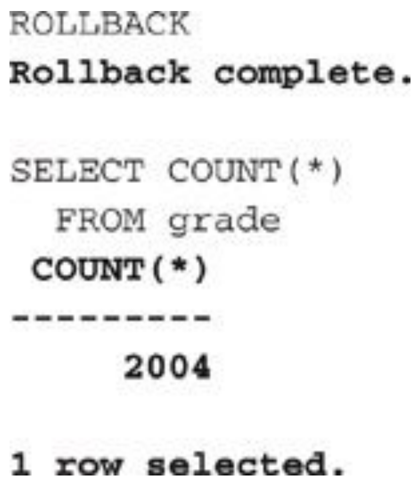
466

467

The following statement deletes all rows in the GRADE table.

```
DELETE FROM grade
2004 rows deleted.
```

When a ROLLBACK command is issued, the DELETE command is undone, and the rows are back in the GRADE table.



```
ROLLBACK
Rollback complete.

SELECT COUNT(*)
  FROM grade
 COUNT(*)
---------
     2004

1 row selected.
```

Referential Integrity and the DELETE Command

A DELETE operation on a row with dependent children rows has different effects, depending on how DELETEs on the foreign key are defined. There are three different ways you can specify a foreign key constraint with respect to deletes: RESTRICT, CASCADE, or SET NULL.

If you issue a DELETE on a parent table with associated child records, and the foreign key constraint is set to ON DELETE CASCADE, the children are automatically deleted. If the foreign key constraint is set to ON DELETE SET NULL, the child rows are updated to a null value, provided that the foreign key column of the child table allows nulls. The default option for a foreign key constraint with respect to DELETEs is RESTRICT. It disallows the deletion of a parent if child rows exist. In this case, you must delete the child rows first, before you delete the parent row.

In the STUDENT schema, all foreign key constraints are set to the default option, which restricts INSERT, UPDATE, and DELETE operations.

DELETES AND REFERENTIAL INTEGRITY IN ACTION

The default foreign key constraint does not allow you to delete any parent row, if any child records exist. In the following example, an attempt is made to delete zip code 10025. Because the ZIP column of the ZIPCODE table is referenced as a foreign key column in the STUDENT table and the table contains student rows with this zip code, you cannot delete the row. Oracle prevents you from creating orphan rows and responds with an error message.

```
DELETE FROM zipcode
  WHERE zip = '10025'
DELETE FROM zipcode
*
ERROR at line 1:
ORA-02292: integrity constraint
(STUDENT.INST_ZIP_FK)
violated - child record found
```

The constraint name error message consists of not only the constraint name but also the name of the schema, which in this case is the STUDENT schema. If you installed the tables into another user account, your schema name will be different. You will learn how to create constraints and specify constraint names in Chapter 12.

A DELETE statement may delete rows in other tables. If the foreign key constraint specifies the ON DELETE CASCADE option, a deletion of a parent row automatically deletes the associated child rows. Imagine that the referential integrity constraint between the STUDENT and ENROLLMENT tables is DELETE CASCADE. A DELETE statement would delete not only the individual STUDENT row but also any associated ENROLLMENT rows.

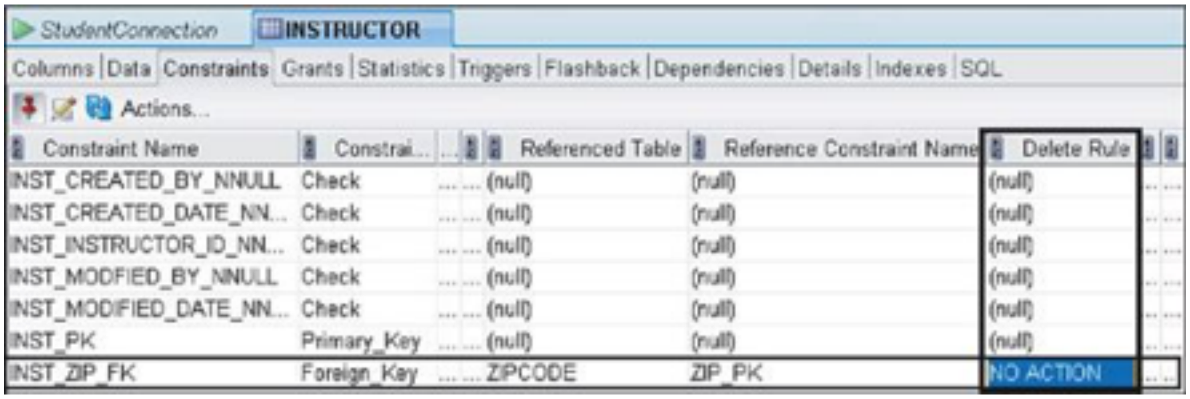
To take the scenario a step further, suppose that the student also has records in the GRADE table. The delete will be successful only if the constraint between the ENROLLMENT table and the GRADE table is also DELETE CASCADE. Then the corresponding rows in the GRADE tables are deleted as well. If the DELETE is RESTRICT, the ORA-02292 error will appear, informing you to delete all the child records first.

As you know, the ZIPCODE table is not only referenced by the STUDENT table but also by the INSTRUCTOR table. Suppose you have the ON DELETE SET NULL constraint as the foreign key. A deletion of zip code 10025 would cause an update of the ZIP column on the INSTRUCTOR table to a null value, provided that the STUDENT table does not contain this zip code.

To find out which foreign keys have DELETE CASCADE, the SET NULL constraint, or the default RESTRICT constraint, you can query the data dictionary views USER_CONSTRAINTS or ALL_CONSTRAINTS, as discussed in Chapter 14.

In SQL Developer, you can refer to the Delete Rule column displayed in the Constraints tab of the table. Figure 11.10 shows the value NO ACTION on the foreign key constraint between the INSTRUCTOR and ZIPCODE tables, which means that any deletions from the ZIPCODE parent table are restricted and do not cause any delete or update action on any related child rows contained in the INSTRUCTOR table.

FIGURE 11.10 The Delete Rule column value of the INSTRUCTOR table


StudentConnection
INSTRUCTOR
Columns | Data | Constraints | Grants | Statistics | Triggers | Flashback | Dependencies | Details | Indexes | SQL
Actions...

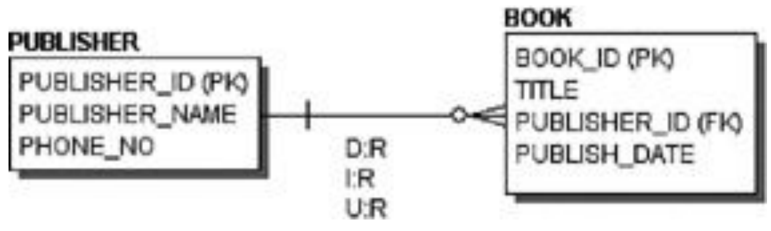
Constraint Name	Constrai...	...	Referenced Table	Reference Constraint Name	Delete Rule	...
INST_CREATED_BY_NNULL	Check	... ...	(null)	(null)	(null)	... ...
INST_CREATED_DATE_NN...	Check	... ...	(null)	(null)	(null)	... ...
INST_INSTRUCTOR_ID_NN...	Check	... ...	(null)	(null)	(null)	... ...
INST_MODIFIED_BY_NNULL	Check	... ...	(null)	(null)	(null)	... ...
INST_MODIFIED_DATE_NN...	Check	... ...	(null)	(null)	(null)	... ...
INST_PK	Primary_Key	... ...	(null)	(null)	(null)	... ...
INST_ZIP_FK	Foreign_Key	... ...	ZIPCODE	ZIP_PK	NO ACTION	...

THE SCHEMA DIAGRAM

Sometimes, schema diagrams depicting the physical relationships between tables show the referential integrity rules in place. Three types of data manipulation operations are possible in SQL: INSERT, UPDATE, and DELETE. On some schema diagrams, you may also find the letters I, U, and D, which are abbreviations for INSERT, UPDATE, and DELETE, respectively. These abbreviated letters indicate the valid rules that these data manipulation operations must follow.

Figure 11.11 shows a schema diagram of the PUBLISHER table and the BOOK table. The foreign key column PUBLISHER_ID is found in the BOOK table. A one-to-many mandatory relationship exists between the PUBLISHER and BOOK tables. I:R indicates that any INSERT operation filling in values in PUBLISHER_ID of the BOOK table is RESTRICTED to values found in the PUBLISHER table. By default, most database systems require this condition when a foreign key is defined on a column.

FIGURE 11.11 Relationship between PUBLISHER and BOOK tables



The U:R notation indicates that any UPDATE to the PUBLISHER_ID column of the BOOK table is RESTRICTED to values found in the PUBLISHER table. Attempting to UPDATE the column with an invalid value violates the U:R data integrity constraint and generates an error. Both the U:R and the I:R referential integrity rules are the default behaviors and are often not listed on schema diagrams.

The notation for the DELETE operation is listed as D:R, indicating that DELETE operations are restricted. Specifically, this means that you cannot delete a publisher row that is referenced in the BOOK table. If you were allowed to delete the row, you would not be able to tell the publisher of the book, and you would create an orphan row. The relationship between the two tables is mandatory, indicating that a null value for the PUBLISHER_ID is not acceptable.

If instead you see a D:C notation, it indicates DELETE CASCADE, meaning a deletion of a PUBLISHER row deletes any associated children rows in the BOOK table.

D:N indicates DELETE SET NULL. This means that upon the deletion of a PUBLISHER row, any corresponding child rows are automatically set to null in the PUBLISHER_ID column of the BOOK table, provided that nulls are allowed.

The TRUNCATE Command

The TRUNCATE command deletes all rows from a table, just like the DELETE command. However, the TRUNCATE command does not allow a WHERE clause and automatically issues a COMMIT. All rows are deleted, and you cannot roll back the change.

```
TRUNCATE TABLE class
Table truncated.
```

The TRUNCATE statement works more quickly than the DELETE statement to remove all rows from a table because the database does not have to store the undo information in case a ROLLBACK command is issued.

If you attempt to use TRUNCATE on a table that is referenced by another table as a foreign key, Oracle issues an error message, indicating that this action is not allowed because it doesn't let you create orphan rows. You must disable the foreign key constraint before you can succeed. Enabling and disabling constraints is discussed in Chapter 12.

```
TRUNCATE TABLE student
TRUNCATE TABLE student
*
ERROR at line 1:
ORA-02266: unique/primary keys in
table referenced by
```

`enabled foreign keys`

Triggers and DML Commands

Oracle enables you to attach triggers to tables that fire on DELETE, INSERT, and UPDATE commands. A table's triggers do not execute when the table is truncated. Triggers are written in the PL/SQL language and can perform sophisticated actions (for example, recording changes to another table for auditing purpose or updating summary values on derived columns).

Locking

In a real-world database system, many users access data concurrently. Occasionally, users collide because they want to manipulate the same piece of information. Locking ensures data consistency.

When you issue an INSERT, UPDATE, DELETE, or MERGE statement, Oracle automatically locks the modified rows. The lock prevents other sessions from making changes to these rows. The lock is released when the session initiating the change commits or rolls back. Other users or sessions can then modify the rows.

Queries do not place locks on rows. Data can always be queried, despite being locked; however, other sessions can see the committed data only. After the successful commit of a transaction, the new change is visible to all sessions, and the lock is released.

If a row is locked by a session, another session cannot acquire the lock and modify the row. The session attempting to acquire the locked row waits until the lock is released. The session might appear frozen while it waits. Users often think that perhaps their connection to the server dropped or that the DML operation is extremely slow. Users might terminate their session or reboot the machine, only to find out that if they retry the same action, the session continues to behave identically. Oracle waits until the lock is released by the other session before it proceeds with the new change.

When you anticipate multiple users contending for the same row(s) simultaneously, you should commit your data frequently.

THE LOST UPDATE PROBLEM

The WHERE clause of the next UPDATE statement not only lists the primary key column (the COURSE_NO column) but also includes the old COST column value.

```
UPDATE course
   SET cost = 800
```

```
  WHERE course_no = 25
    AND cost = 1195
```

Although this is may seem unnecessary, it can be helpful in case another user made changes to the values in the meantime. Then the UPDATE statement is not successful and returns 0 rows updated. This indicates that the row containing the old value is no longer found. Many end-user application programs append the values displayed on a user's screen to the WHERE clause of an UPDATE statement. If the UPDATE returns with the 0 rows updated message, the program can alert the user that changes have been made and request the user to requery the data. This prevents the user from unknowingly overwriting data that changed since he or she last retrieved the data.

You might wonder why Oracle doesn't automatically lock the data to prevent such a situation or place locks on queries. Oracle releases the lock after the user issues a COMMIT or ROLLBACK. A SELECT does not cause any locks; the other user may have queried the data, updated the data, and issued a COMMIT immediately afterward. Therefore, any subsequent updates do not interfere with another user's UPDATE statement because the lock is already released.

While Oracle automatically takes care of locking, you can explicitly acquire a lock by using the SELECT FOR UPDATE or the LOCK TABLE statement. This will override the default locking mechanism; however, this functionality is infrequently used in the real world. Oracle's implicit and automatic locking mechanism works very well for the vast majority of scenarios, and by adding the "old" values to the WHERE clause, you avoid overwriting any unwanted changes.

LOCKING OF ROWS BY DDL OPERATIONS

Locks are not just acquired on individual rows but also on the entire table when a DDL operation such as an ALTER TABLE or a CREATE INDEX command is issued. A DML operation cannot update the table while the DDL operation is in progress (for example, you cannot update rows while a table is being altered), and the same holds true for the reverse: A DDL command on a table cannot be executed if users are holding locks on the table (with some exceptions, such as the creation of online indexes, as discussed in Chapter 13).

Read Consistency of Data

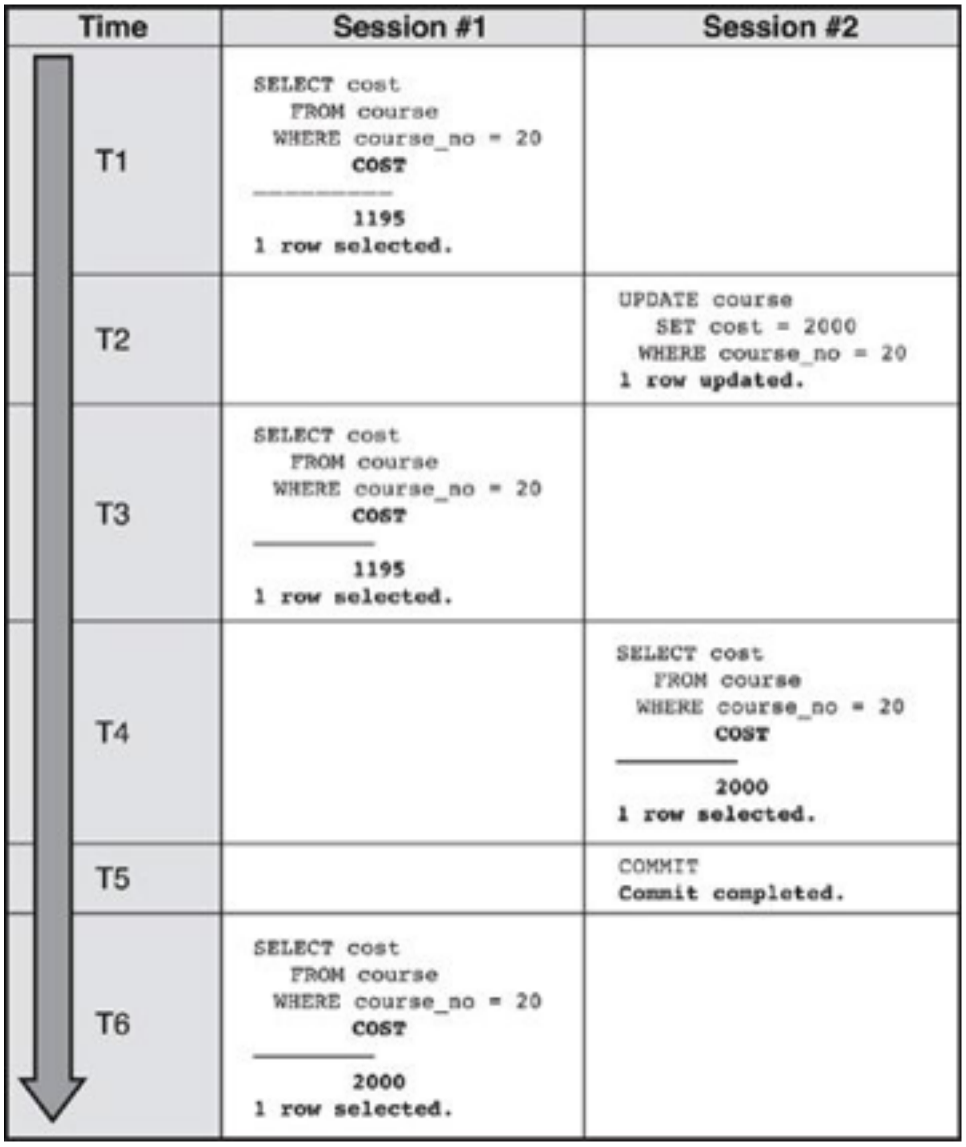
Whenever a user changes data with a DML operation, Oracle keeps track of the old values on a rollback segment. If the user rolls back the transaction with the ROLLBACK command, Oracle reads the old values from the rollback segment and returns the data to the previous state.

THE ROLLBACK SEGMENT AND UNDO TABLESPACE

The UNDO tablespace contains the rollback segments that keep track of changes not yet committed. It allows users to issue the ROLLBACK command to restore the data to its original state. Uncommitted data is not permanent and therefore not ready for other users to see yet. Before any data is changed on the actual table, the change is written to the rollback segments first.

Figure 11.12 illustrates the visibility and timing of any changes made to the COST column of the COURSE table for two individual sessions. For example, Session #2 updates the COST column value for COURSE_NO 20 to 2000 but does not commit the change. Session #1 still sees the old values, which are retrieved from the rollback segments. Session #1, or any other session for that matter, does not see the modified data until the user performing the change makes it permanent by issuing a COMMIT. Furthermore, until a COMMIT or a ROLLBACK is issued by Session #2, the row is locked and cannot be manipulated by another session.

FIGURE 11.12 The effect of the COMMIT command



Time	Session #1	Session #2
T1	SELECT cost FROM course WHERE course_no = 20 COST ---------- 1195 1 row selected.	

T2		UPDATE course
		SET cost = 2000
		WHERE course_no = 20
		1 row updated.

T3	SELECT cost FROM course WHERE course_no = 20 COST ---------- 1195 1 row selected.	
T4		SELECT cost FROM course WHERE course_no = 20 COST ---------- 2000 1 row selected.
T5		COMMIT Commit completed.
T6	SELECT cost FROM course WHERE course_no = 20 COST ---------- 2000 1 row selected.	

THE SYSTEM CHANGE NUMBERS (SCN) AND MULTI-VERSIONING

When long-running queries and DML operations occur simultaneously, Oracle automatically handles this with the use of the System Change Number (SCN), a unique number that tracks the order in which events occur. This feature enables queries to return a read-consistent result. For example, say that a query starts at 10:00 A.M. and ends at 10:05 A.M., and during this time it computes the sum of all salaries for all employees. At 10:03 A.M., the salary of one employee is updated, and a COMMIT is issued. What result does the query return? Because the query began before the UPDATE was issued, the result will return a read-consistent result, based on the point in time when the query started, which is 10:00 A.M. When a query reads the newly changed salary row, it recognizes that the SCN of the UPDATE is issued after the start of the query, and it looks for the old salary value on the rollback segment.

If you have very long-running queries, you might get an ORA-1555 ("snapshot too old") error message; this indicates that Oracle had to overwrite the rollback information you are attempting to access and therefore cannot return a read-consistent result. Rollback data can be overwritten by other transactions when the previous transaction is committed or rolled back. When this rollback data is no longer available, the long-running query is looking for undo information that no longer exists, so you get this error message. To eliminate this error, you can attempt to reissue the query, or, if there is a lot of activity on the system, you might need to increase the UNDO_RETENTION setting of the UNDO tablespace.

For more information on read consistency, database recovery, and the management of the rollback/UNDO tablespace, refer to the *Oracle Database Administrator's Guide*.

Flashback Queries

Oracle's flashback query feature allows you to look at values of a query at a specific time in the past, such as before specific DML statements occurred. This can be useful in case a user accidentally performs an unintended but committed DML change. Another possible application of the feature is to compare the current data against the previous day's data to see the changes. When using the flashback query, you can either specify an explicit time expression (such as an interval or a specific timestamp value) or indicate an individual SCN.

Data for flashback queries is kept only for a certain time period that is dependent on the undo management implemented by the DBA. You must familiarize yourself with the limitations of this feature. For example, issuing certain DDL commands, such as altering a table by dropping or modifying columns, invalidates the undo data for the individual table.

Not every user can perform flashbacks; the DBA has to provide a user with the either the FLASHBACK object privilege on the specific table or the FLASHBACK ANY TABLE system privilege. For the STUDENT user to be able to run these queries, you need to request these privileges; they are not part of the default privileges assigned to the account.

Following is an example that illustrates the use of the flashback query feature. The SELECT statement returns the current value of the table before any changes occur. You see the cost of course number 10 displayed as 1195. The subsequent UPDATE statement changes the cost to 9999 and makes the change permanent with the COMMIT command.

```
SELECT course_no, cost
  FROM course
 WHERE course_no = 10
COURSE_NO        COST
---------------- ----
              10 1195

1 row selected.

UPDATE course
   SET cost = 9999
 WHERE course_no = 10
1 row updated.

COMMIT
Commit complete.
```

STATEMENT-LEVEL FLASHBACK

The statement-level flashback ability allows the AS OF clause in the SELECT statement followed either by a TIMESTAMP value or a particular SCN. The next statement shows use of the TIMESTAMP clause to retrieve the value for the COST column for course number 10 as of February 3, 2009, at 4:30 P.M.

```
SELECT course_no, cost
  FROM course AS OF TIMESTAMP
  TO_TIMESTAMP('03-Feb-2009 04:30:00 PM',
               'DD-MON-YYYY HH:MI:SS AM')
 WHERE course_no = 10
COURSE_NO        COST
---------------- ----
              10 1195

1 row selected.
```

If flashback data is not available anymore, Oracle informs you with an ORA-08180: no snapshot found based on specified time error message. The syntax for the flashback query clause is as follows.

```
AS OF SCN|TIMESTAMP expr
```

If you want to run the flashback versions query using a specific SCN number, you can obtain the number with the next SQL statement. This may be useful for querying the number before the start of a batch job, and if anything goes wrong, you can query the changes easily. (The DBA has to grant you the EXECUTE privilege on the DBMS_FLASHBACK package to be able to run this query.)

```
SELECT DBMS_FLASHBACK.GET_SYSTEM_CHANGE_NUMBER
  FROM dual
GET_SYSTEM_CHANGE_NUMBER
------------------------
                 4423510

1 row selected.
```

The following statement retrieves the rows that were inserted or updated in the COURSE table within the last 30 minutes. If you want to find out what data was updated and deleted, reverse the two SELECT statements. If you prefer to see a detailed log of the changes, use the VERSIONS BETWEEN parameter, discussed next.

```
SELECT *
  FROM course
 MINUS
SELECT *
FROM course AS OF TIMESTAMP
SYSTIMESTAMP - INTERVAL '30' MINUTE
```

RETRIEVING FLASHBACK HISTORY WITH THE VERSIONS BETWEEN PARAMETER

The VERSIONS BETWEEN parameter allows you to retrieve the history of changes during a particular time period. For example, if the column value changes again—say the COST column is updated to 5555—you see each individual change.

```
UPDATE course
   SET cost = 5555
WHERE course_no = 10
1 row updated.
```

```
COMMIT
Commit complete.
```

The query determines the changes that occurred on the row within the last 10 minutes.

```
SELECT course_no, cost, VERSIONS_XID,
       VERSIONS_OPERATION
  FROM course
       VERSIONS BETWEEN TIMESTAMP
       SYSTIMESTAMP - INTERVAL '10' MINUTE
   AND SYSTIMESTAMP - INTERVAL '1' SECOND
 WHERE course_no = 10
 COURSE_NO  COST VERSIONS_XID     VERSIONS_OPERATION
---------- ----- ---------------- ------------------
        10  5555 0001001A000028B4 U
        10  9999 00030028000028B7 U
        10  1195

3 rows selected.
```

The columns VERSIONS_XID and VERSIONS_OPERATION in the SELECT list are pseudo-columns that store the transaction reference information regarding the change and the type of change. The VERSIONS_OPERATION pseudocolumn indicates the type of data manipulation that took place (for example, INSERT, UPDATE, DELETE). The VERSIONS_XID column allows you to trace back the various modification details in the data dictionary view FLASHBACK_TRANSACTION_QUERY. (You will see an example of this shortly. Also refer to Chapter 14 for more information about data dictionary views.) You can also see the timestamp of the changes. Table 11.1 lists other pseudocolumns that contain more details about DML changes.

TABLE 11.1 VERSIONS Query Pseudocolumns

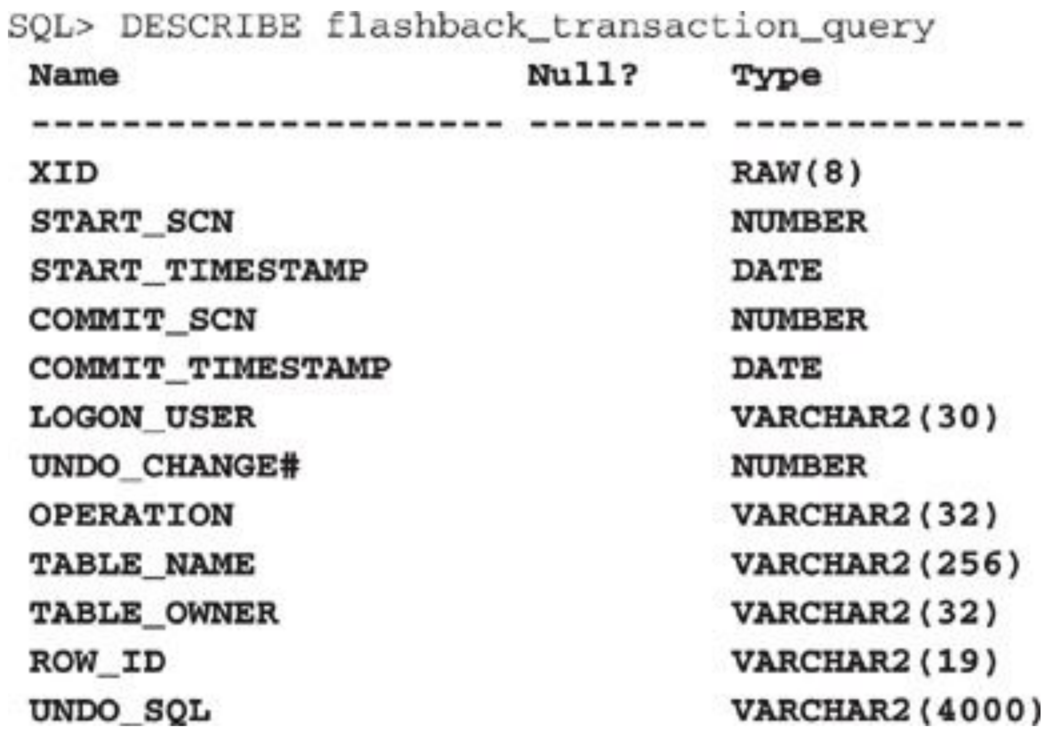
PSEUDOCOLUMN	DESCRIPTION
VERSIONS_STARTTIME	The timestamp of the first version of the row.
VERSIONS_ENDTIME	The timestamp of the last version of the row.
VERSIONS_OPERATION	The type of operation the row was subject to. Values are I (for INSERT), U (for UPDATE), and D (for DELETE).
VERSIONS_STARTSCN	The SCN for the first version of the row.
VERSIONS_ENDSCN	The SCN for the last version of the row.
VERSIONS_XID	The transaction ID generated by the change.

The syntax of this flashback query is as follows. Instead of the TIMESTAMP expression, you can choose a specific SCN.

```
VERSIONS BETWEEN {SCN|TIMESTAMP}
{expr|MINVALUE} AND {expr|MAXVALUE}
```

FLASHBACK_TRANSACTION_QUERY

The FLASHBACK_TRANSACTION_QUERY data dictionary contains not only details about the transaction, such as who performed the change and when, but also the SQL statement to undo the operation. The following SQL*Plus DESCRIBE command shows the available columns. The data dictionary references the XID column, which corresponds to the previously listed VERSIONS_XID pseudocolumn.



```
SQL> DESCRIBE flashback_transaction_query
 Name                      Null?    Type
 ------------------------- -------- -------------
 XID                                RAW(8)
 START_SCN                          NUMBER
 START_TIMESTAMP                    DATE
 COMMIT_SCN                         NUMBER
 COMMIT_TIMESTAMP                   DATE
 LOGON_USER                         VARCHAR2(30)
 UNDO_CHANGE#                       NUMBER
 OPERATION                          VARCHAR2(32)
 TABLE_NAME                         VARCHAR2(256)
 TABLE_OWNER                        VARCHAR2(32)
 ROW_ID                             VARCHAR2(19)
 UNDO_SQL                           VARCHAR2(4000)
```

The following statement lists one of the undo SQL statements for the table COURSE, which is owned by the STUDENT user. The UNDO statement lists the ROWID pseudocolumn, showing a unique way to identify a row within a table. You'll learn more about the ROWID pseudo-column in Chapter 13.

```
  SELECT undo_sql
    FROM flashback_transaction_query
   WHERE table_name = 'COURSE'
   AND table_owner = 'STUDENT'
UNDO_SQL
---------------------------------------------
update "STUDENT"."COURSE" set "COST" = '1195'
 where ROWID = 'AAAKgzAAAAAAAAAAAA';
...
```

Another noteworthy observation you might make is the double quotation marks around the schema, table, and column names. Oracle object names can be case-sensitive; using double quotes ensures the correct spelling of the object. Chapter 12 discusses the case-sensitivity of column and table names in more detail.

FLASHBACK TABLE AND FLASHBACK DATABASE COMMANDS

Oracle has implemented two more flashback-related commands. FLASHBACK TABLE and FLASHBACK

DATABASE allow you to revert an individual table and an entire database, respectively.

Keep in mind that while the flashback choices provide a fallback option for application or user error, flashback data is retained only up to a specific period in time.

The following statement illustrates a FLASHBACK TABLE statement that restores the COURSE table to its state 5 minutes ago.

```
FLASHBACK TABLE course
   TO TIMESTAMP(SYSTIMESTAMP -
INTERVAL '5' MINUTE)
Flashback complete.
```

To flashback a table or an entire database, you need special permissions, as discussed in Chapter 15. Furthermore, the FLASHBACK TABLE command does not work after you perform table structure changes (for example, adding, dropping, or modifying table columns). Another requirement for the FLASHBACK TABLE command is that the table must have the ROW MOVEMENT option enabled. (This is done when a table is initially created or with the ALTER TABLE table_name ENABLE ROW MOVEMENT syntax.)

The syntax of the FLASHBACK TABLE command is as follows. The BEFORE DROP option is discussed in Chapter 12 because it allows you to restore a dropped table.

```
FLASHBACK TABLE tablename
[,tablename...] TO
{{SCN|TIMESTAMP} expr [ENABLE|
DISABLE TRIGGERS]|
BEFORE DROP [RENAME TO
newtablename]}
```

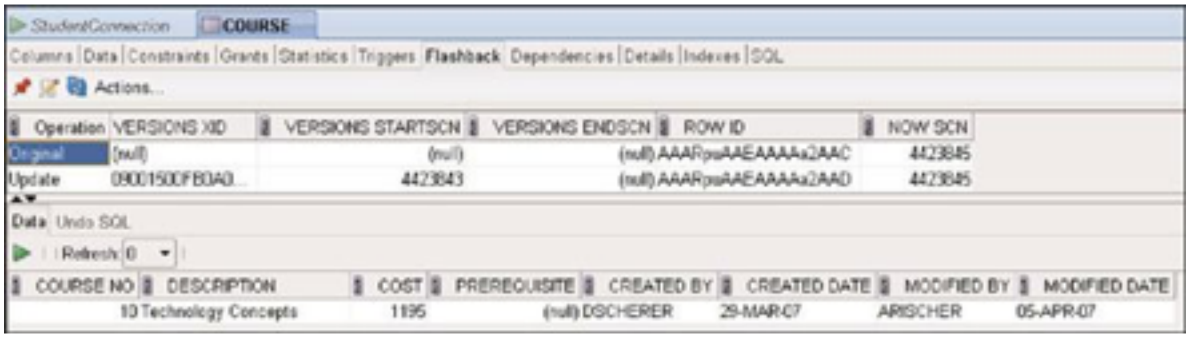
Although there are many options for correcting unintentional errors, you must nevertheless have proper measures in place to protect your data against any accidents or failures so you can recover at any time. The DBA is responsible for establishing, administering, and periodically testing the appropriate data safeguards and procedures.

SQL DEVELOPER AND FLASHBACK

SQL Developer includes a Flashback tab, which allows you to see the original and modified data for each row. Figure 11.13 shows the two rows; for each respective row in the table, you can see the values of the data in the Data tab below. The highlighted row shows the Original operation, and the row below shows the same course number with an updated COST column.

Next to the Data subtab is the Undo SQL tab. If you have the appropriate privileges, you can look at the Undo SQL to see the syntax required to reverse the change.

FIGURE 11.13 Flashback tab in SQL Developer


StudentConnection
COURSE
Columns | Data | Constraints | Grants | Statistics | Triggers | Flashback | Dependencies | Details | Indexes | SQL
Actions...
Data
Undo SQL
Refresh: 0

Operation	VERSIONS XID	VERSIONS STARTSCN	VERSIONS ENDSCN	ROW ID	NOW SCN
Original	(null)	(null)	(null)	AAARpuAAEAAAAa2AAC	4423845
Update	0900150DFB0A0...	4423843	(null)	AAARpuAAEAAAAa2AAD	4423845

COURSE NO	DESCRIPTION	COST	PREREQUISITE	CREATED BY	CREATED DATE	MODIFIED BY	MODIFIED DATE
10	Technology Concepts	1195	(null)	DSCHERER	29-MAR-07	ARISCHER	05-APR-07

Connection Sharing in SQL Developer

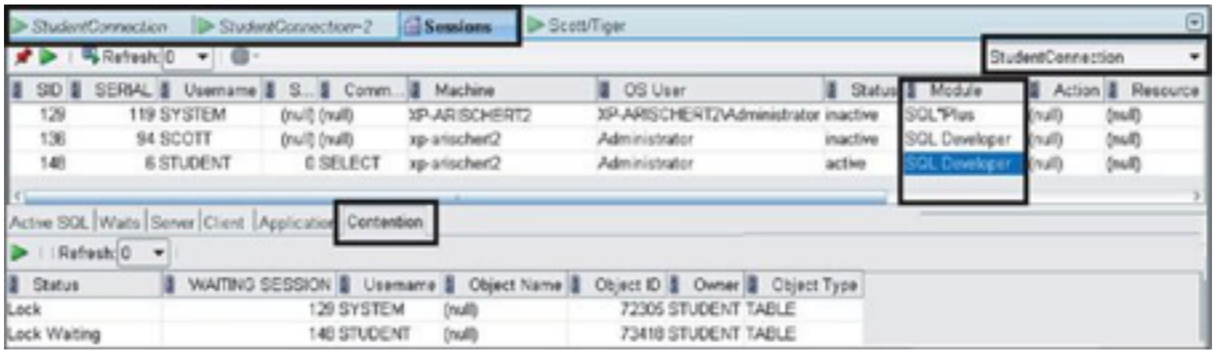
In SQL Developer, a *session* is an individual database connection. For example, if you open two SQL Worksheets and the Data tab, using the Connections navigator, this is considered one session, not three separate sessions. Any changes are visible within the same session; also, there are no locking conflicts if the same row is modified. If you want to create a separate, independent session in SQL Developer, you can press Ctrl+Shift+N.

Figure 11.14 displays the Sessions screen (which you open by selecting Tools, Monitor Sessions) in SQL Developer. The Sessions tab shows the listing of current sessions against the database instance. For each connected session, you see a row listing the username, machine name, operating system user account, and tool used to connect to the database instance.

The screenshot shows you that there are three users connected in the database instance: SYSTEM, SCOTT, and STUDENT. Both the SCOTT and STUDENT sessions are initiated by the same machine and by SQL Developer. The SYSTEM session was started using SQL*Plus (see the Module column).

There are several tabs at the top of the screen. The first three tabs all share the StudentConnection, which was the STUDENT user name. These tabs collectively represent a single session. For example, StudentConnection~2 is another open Worksheet that shares the same connection. Also, the Session tab is opened with the StudentConnection (see the current connection information on the right of the screen).

FIGURE 11.14 List of sessions



SID	SERIAL	Username	S...	Comm...	Machine	OS User	Status	Module	Action	Resource
129	119	SYSTEM	(null)	(null)	XP-ARISCHERT2	XP-ARISCHERT2\Administrator	inactive	SQL*Plus	(null)	(null)
136	94	SCOTT	(null)	(null)	xp-arischert2	Administrator	inactive	SQL Developer	(null)	(null)
148	6	STUDENT	0	SELECT	xp-arischert2	Administrator	active	SQL Developer	(null)	(null)

Status	WAITING SESSION	Username	Object Name	Object ID	Owner	Object Type
Lock	129	SYSTEM	(null)	72305	STUDENT	TABLE
Lock Waiting	148	STUDENT	(null)	73418	STUDENT	TABLE

If you click on any of the rows shown in the Sessions tab, subtabs change for the individual session you select. One of the very useful subtabs is the Contention subtab, which shows which users are holding locks. If you are concerned that you may be waiting for data that another user is locking, you can review this subtab.

In this case, the SYSTEM user performed an update to all the rows of the STUDENT table without committing the change, and the STUDENT user performed the same update from his user account. Now the STUDENT user is waiting for the SYSTEM user to perform a COMMIT or ROLLBACK operation to be able to proceed with the command.

Because the SYSTEM user is one of the most privileged users in the database, this account has the right to update tables in the STUDENT schema. To simulate the lock example, the following update was issued by SYSTEM.

```
UPDATE student.student
   SET zip = 10025
```

The STUDENT table is referenced by the owner of the table (student.student). Then the STUDENT user issues a subsequent update, attempting to change all the zip codes to 10026. Because the SYSTEM user issued the command against the rows first, this user holds the locks to all the modified rows, the STUDENT user has to wait for the needed locks to be released. Then the STUDENT user can perform the respective update command. The Contention sub tab shows that the STUDENT user is waiting for the locks to be released.

You can also simulate a lock contention even within the same STUDENT account. For example, within SQL Developer you can create an unshared Worksheet by pressing Ctrl+Shift+N. This creates a separate session.

Performance Considerations When Writing DML Statements

Much like the index in a book, Oracle allows the creation of indexes on tables, which can help speed up retrieval of rows. While the discussion of indexes takes place in Chapter 13, at this point, you should be aware of the effect indexes have on the performance of your data manipulation statement.

Indexes can slow down data manipulation because they may require maintenance of the values within the indexes. For example, if an UPDATE operation affects an indexed column, the value in the index needs to be updated. A DELETE statement requires the entries in the index to be marked for deletion. An INSERT command creates index entries for all supplied column values where an index exists.

However, indexes can be beneficial for UPDATE or DELETE statements if the statement updates a small portion of the table and if it contains a WHERE clause that refers to an indexed column. Rather than look through the entire table for the desired information, Oracle finds it quickly and performs the desired operation.

One of the reasons a database designer should consider using an index on the foreign key is if updates or deletions for the primary key on the parent table occur. If an index on the foreign key is present, referential integrity is maintained by temporarily locking the index of the child table rather than the entire child table. This greatly improves performance of these types of commands.

LAB 11.2 EXERCISES

a) Using an UPDATE statement, change the location to B111 for all sections where the location is currently L210.

b) Update the MODIFIED_BY column with the user login name and update the MODIFIED_DATE column with a date of March 31, 2009, using the TO_DATE function for all the rows updated in exercise a.

c) Update instructor Irene Willig's zip code to 90210. What do you observe?

d) What does the following query accomplish?

```
UPDATE enrollment e
```

```
   SET final_grade = (SELECT
AVG(numeric_grade)
   FROM grade g
   WHERE e.student_id =
g.student_id
   AND e.section_id =
g.section_id),
   modified_date = SYSDATE,
   modified_by = 'Your name here'
   WHERE student_id IN (SELECT
student_id
   FROM student
   WHERE last_name like 'S%')
```

e) Update the first name from Rick to Nick for the instructor with the ID 104.

f) Write and execute an UPDATE statement to update the phone numbers of instructors from 2125551212 to 212-555-1212 and the MODIFIED_BY and MODIFIED_DATE columns with the user logged in and today's date, respectively. Write a SELECT statement to prove the update worked correctly. Do not issue a COMMIT command.

g) If you use SQL Developer: Create another session, independent of the current

StudentConnection session, using Ctrl+Shift+N. If you use SQL*Plus: Invoke another SQL*Plus connection with student/learn.

Execute the same SELECT statement you wrote in exercise f to prove that your update worked correctly. Compare the result with the result in exercise f.

h) What is the result of the following statement?

```
MERGE INTO enrollment e
USING (SELECT AVG(numeric_grade)
final_grade,
section_id, student_id
FROM grade
GROUP BY section_id, student_id)
g
ON (g.section_id = e.section_id
AND g.student_id = e.student_id)
WHEN MATCHED THEN
UPDATE SET e.final_grade =
g.final_grade
WHEN NOT MATCHED THEN
INSERT (e.student_id,
e.section_id, e.enroll_date,
e.final_grade, e.created_by,
e.created_date,
e.modified_date, e.modified_by)
VALUES (g.section_id,
g.student_id, SYSDATE,
g.final_grade, 'MERGE', SYSDATE,
SYSDATE, 'MERGE')
```

i) Delete all rows from the GRADE_CONVERSION table. Then select all the data from the table, issue a ROLLBACK command, and explain the outcome.

j) If TRUNCATE is used in exercise i instead of DELETE, how would this change the outcome? Caution: Do not actually execute the TRUNCATE statement unless you are prepared to reload the data.

k) Delete the row inserted in exercise a in Lab 11.1 from the GRADE_TYPE table.

l) What is the effect of the following statement?

```
DELETE FROM enrollment
WHERE student_id NOT IN
(SELECT student_id
FROM student s, zipcode z
WHERE s.zip = z.zip
AND z.city = 'Brooklyn'
AND z.state = 'NY')
```

LAB 11.2 EXERCISE ANSWERS

a) Using an UPDATE statement, change the location to B111 for all sections where the location is currently L210.

ANSWER: The UPDATE statement updates the LOCATION column in 10 rows of the SECTION table.

```
UPDATE section
   SET location = 'B111'
  WHERE location = 'L210'
10 rows updated.
```

Without a WHERE clause, all rows in the SECTION table are updated, not just 10 rows. For example, if you want to make sure all students have their last names begin with a capital letter, issue the following UPDATE statement.

```
UPDATE student
   SET last_name =
INITCAP(last_name)
```

UPDATES TO MULTIPLE TABLES

Typically, the UPDATE statement affects a single table. However, if the table has a trigger associated with it, it may fire if the certain conditions specified in the trigger are true. The code in the trigger may cause insertions, updates, or deletions to other tables. Triggers can also add or modify values to rows you are changing. You can query the data dictionary view USER_TRIGGERS to see if any triggers are associated with your tables or review the Triggers tab for a table in SQL Developer.

b) Update the MODIFIED_BY column with the user login name and update the MODIFIED_DATE column with a date of March 31, 2009, using the TO_DATE function for all the rows updated in exercise a.

ANSWER: The MODIFIED_BY column is updated with the USER function to reflect an update by the user logged in, namely STUDENT, and the MODIFIED_DATE column is updated using the TO_DATE function. The update is based on the previously updated location.

```
UPDATE section
   SET modified_by = USER,
modified_date = TO_DATE('31-
MAR-2009', 'DD-MON-YYYY')
   WHERE location = 'B111'
```
10 rows updated.

Instead of writing them as individual UPDATE statements, exercises a and b can be combined in a single UPDATE statement, with the columns separated by commas.

```
UPDATE section
   SET location = 'B111',
   modified_by = USER,
   modified_date = TO_DATE('31-
MAR-2009', 'DD-MON-YYYY')
WHERE location = 'L210'
```

c) Update instructor Irene Willig's zip code to 90210. What do you observe?

ANSWER: The attempt to change the zip code to a value that does not exist in the ZIPCODE table results in a referential integrity constraint error.

```
UPDATE instructor
   SET zip = '90210'
WHERE last_name = 'Willig'
   AND first_name = 'Irene'
```
UPDATE instructor

```
*
ERROR at line 1:
ORA-02291: integrity constraint
(STUDENT.INST_ZIP_FK)
violated - parent key not found
```

Oracle does not allow any invalid values in a column if the foreign key constraint exists and is enabled.

A query checking for this zip code in the ZIPCODE table retrieves no rows.

```
SELECT zip
   FROM zipcode
WHERE zip = '90210'
no rows selected
```

UNIQUELY IDENTIFYING RECORDS

The WHERE clause in the previous UPDATE statement lists the first and last name of the instructor, and it happens to be unique and sufficient to identify the individual. Imagine a scenario in which you have instructors with the identical name, but who are in fact different individuals. When you perform manipulation of data, it is best to include the primary key value, such as the INSTRUCTOR_ID, to ensure that the correct row is changed.

d) What does the following query accomplish?

```
UPDATE enrollment e
   SET final_grade = (SELECT
AVG(numeric_grade)
   FROM grade g
   WHERE e.student_id =
g.student_id
   AND e.section_id =
g.section_id),
   modified_date = SYSDATE,
   modified_by = 'Your name here'
   WHERE student_id IN (SELECT
student_id
   FROM student
   WHERE last_name like 'S%')
```

ANSWER: This query updates the FINAL_GRADE, MODIFIED_DATE, and MODIFIED_BY columns of the ENROLLMENT table for students with last names starting with the letter S. The computed average grade is based on the individual grades received by the student for the respective section.

The example illustrates a correlated UPDATE statement. The outer query identifies the students with last names of beginning with S. For each individual outer row, the inner correlated subquery executes and computes the average of the individual grades from the GRADE table. The result is then updated in the FINAL_GRADE column of the ENROLLMENT table.

e) Update the first name from Rick to Nick for the instructor with the ID 104.

ANSWER: The primary key column INSTRUCTOR_ID identifies the instructor uniquely and is therefore used in the WHERE clause. In addition, it helps to include the old value of the FIRST_NAME column in the WHERE clause, in case any previous changes to the column have been made.

```
UPDATE instructor
   SET first_name = 'Nick'
WHERE instructor_id = 109
   AND first_name = 'Rick'
1 row updated.
```

f) Write and execute an UPDATE statement to update the phone numbers of instructors from

2125551212 to 212-555-1212 and the MODIFIED_BY and MODIFIED_DATE columns with the user logged in and today's date, respectively. Write a SELECT statement to prove the update worked correctly. Do not issue a COMMIT command.

ANSWER: A single UPDATE statement updates 3 columns in 10 rows simultaneously in the INSTRUCTOR table. The MODIFIED_BY column is updated with the USER function, and the MODIFIED_DATE column is updated with the SYSDATE function, entering today's date and time into the column.

```
UPDATE instructor
   SET phone = '212-555-1212',
       modified_by = USER,
       modified_date = SYSDATE
 WHERE phone = '2125551212'
10 rows updated.

SELECT instructor_id, phone, modified_by, modified_date
  FROM instructor
INSTRUCTOR_ID PHONE        MODIFIED_BY MODIFIED_
------------- ------------ ----------- ---------
          101 212-555-1212 STUDENT     09-MAY-09
          102 212-555-1212 STUDENT     09-MAY-09
...
          109 212-555-5555 STUDENT     09-MAY-09
          110 212-555-5555 STUDENT     09-MAY-09

10 rows selected.
```

g) If you use SQL Developer: Create another session, independent of the current StudentConnection session, using Ctrl+Shift+N. If you use SQL*Plus: Invoke another SQL*Plus connection with student/learn.

Execute the same SELECT statement you wrote in exercise f to prove that your update worked correctly. Compare the result with the result in exercise f.

ANSWER: You will observe when you execute the SQL statement from exercise f that this new session does not reflect the changes performed in the other session. Any other database user or session cannot see the updated values in the INSTRUCTOR table until a COMMIT command is issued in the original session.

```
SELECT instructor_id, phone, modified_by, modified_date
  FROM instructor
INSTRUCTOR_ID PHONE        MODIFIED_BY MODIFIED_
------------- ------------ ----------- ---------
          101 2125551212   ESILVEST    02-JAN-07
          102 2125551212   ESILVEST    02-JAN-07
...
          109 2125555555   ESILVEST    02-JAN-07
          110 2125555555   ARISCHER    11-MAR-07

10 rows selected.
```

When you are ready to move on to the next exercise, issue the ROLLBACK command in the first session to undo your changes and release the locks.

h) What is the result of the following statement?

```
MERGE INTO enrollment e
USING (SELECT AVG(numeric_grade)
final_grade,
section_id, student_id
FROM grade
GROUP BY section_id, student_id) g
   ON (g.section_id = e.section_id
   AND g.student_id =
e.student_id)
WHEN MATCHED THEN
   UPDATE SET e.final_grade =
g.final_grade
WHEN NOT MATCHED THEN
   INSERT (e.student_id,
e.section_id, e.enroll_date,
   e.final_grade, e.created_by,
e.created_date,
   e.modified_date, e.modified_by)
   VALUES (g.section_id,
g.student_id, SYSDATE,
   g.final_grade, 'MERGE',
SYSDATE,
   SYSDATE, 'MERGE')
```

ANSWER: The MERGE statement updates the column FINAL_GRADE to the average grade per student and section, based on the GRADE table. If the section and student are not found in the ENROLLMENT table, the MERGE command inserts the row.

Actually, the INSERT part of the MERGE statement will probably never be executed because a row in the GRADE table cannot exist unless an ENROLLMENT row exists. The foreign key relationship between the two tables enforces this. In this instance, the following correlated subquery UPDATE achieves the same result as the MERGE statement.

```
UPDATE enrollment e
   SET final_grade = (SELECT
AVG(numeric_grade)
   FROM grade g
   WHERE g.section_id =
e.section_id
   AND g.student_id =
e.student_id)
```

i) Delete all rows from the GRADE_CONVERSION table. Then select all the data from the table, issue a ROLLBACK command, and explain the outcome.

ANSWER: A DELETE statement deletes all rows in the GRADE_CONVERSION table. A subsequently issued SELECT statement shows no rows in the table. Issuing a ROLLBACK undoes the delete. You can verify this by issuing another SELECT statement against the table.

```
DELETE FROM grade_conversion
15 rows deleted.

SELECT *
   FROM grade_conversion
no rows selected

ROLLBACK
Rollback complete.
```

j) If TRUNCATE is used in exercise i instead of DELETE, how would this change the outcome? Caution: Do not actually execute the TRUNCATE statement unless you are prepared to reload the data.

ANSWER: When TRUNCATE is used, the data cannot be rolled back; the ROLLBACK statement has no effect. A subsequent SELECT statement reflects no rows in the GRADE_CONVERSION table.

```
TRUNCATE TABLE grade_conversion
Table truncated.

ROLLBACK
Rollback complete.

SELECT COUNT(*)
  FROM grade_conversion
  COUNT(*)
 ---------
         0

 1 row selected.
```

When the ROLLBACK command is issued, Oracle returns the “Rollback complete” message. This is misleading, because in this case a rollback did not occur; the data is permanently deleted. Be sure to use caution when using the TRUNCATE TABLE command.

k) Delete the row inserted in exercise a in Lab 11.1 from the GRADE_TYPE table.

ANSWER: A DELETE statement is written for the row where the grade type code is 'EC'.

```
DELETE FROM grade_type
   WHERE grade_type_code = 'EC'
1 row deleted.
```

l) What is the effect of the following statement?

```
DELETE FROM enrollment
   WHERE student_id NOT IN
   (SELECT student_id
   FROM student s, zipcode z
   WHERE s.zip = z.zip
   AND z.city = 'Brooklyn'
   AND z.state = 'NY')
```

ANSWER: The statement deletes enrollment rows for all students except those who live in Brooklyn, NY.

The DELETE statement narrows down the records in the WHERE clause by using a NOT IN subquery to find students who do not live in Brooklyn, NY. Alternatively, the DELETE statement can be rewritten as a correlated subquery, using the NOT EXISTS operator, which under certain circumstances can execute faster.

```
DELETE FROM enrollment e
   WHERE NOT EXISTS
   (SELECT 'x'
   FROM student s, zipcode z
   WHERE s.zip = z.zip
   AND s.student_id = e.student_id
   AND z.city = 'Brooklyn'
   AND z.state = 'NY')
```

Because the STUDENT_ID in the STUDENT table is defined as NOT NULL, the NOT IN and NOT EXISTS statements are equivalent. For more information on the differences between NOT IN and NOT EXISTS, see Chapter 8,“Subqueries,” and Chapter 18,“SQL Optimization.”

The DELETE command is not successful because GRADE records exist for these students.

Lab 11.2 Quiz

In order to test your progress, you should be able to answer the following questions.

1) It is possible to restore rows that have been deleted using the DELETE command.

______**a)** True

______**b)** False

2) There is no syntax error in the following UPDATE statement.

```
UPDATE grade_type
SET description = 'Exams'
WHERE grade_type_code IN ('FI',
'MT')
```

______**a)** True

______**b)** False

3) The SELECT command always places locks on retrieved rows.

______**a)** True

______**b)** False

4) Oracle achieves read consistency by reading uncommitted data.

______**a)** True

______**b)** False

5) Oracle releases the lock of a row after the session issues a COMMIT or ROLLBACK command.

______**a)** True

______**b)** False

ANSWERS APPEAR IN APPENDIX A.

LAB 11.3 The SQL Developer Data Tab

LAB OBJECTIVES

After this lab, you will be able to:

- Manipulate Data Using SQL Developer
- Export Data to Different File Formats
- Import Data from Different Data Sources

SQL Developer's GUI allows you to insert and manipulate existing data directly with the Data tab. This is a quick, simple, and convenient way to make changes. While the Data tab is a very powerful feature in SQL Developer, it does not eliminate the need to know DML SQL command syntax altogether.

Display the Data tab by selecting the table in the Connection navigator. You see a number of icons on the top, some of which you have already learned about. Table 11.2 provides a brief description of the icons.

TABLE 11.2 Data Tab Icons

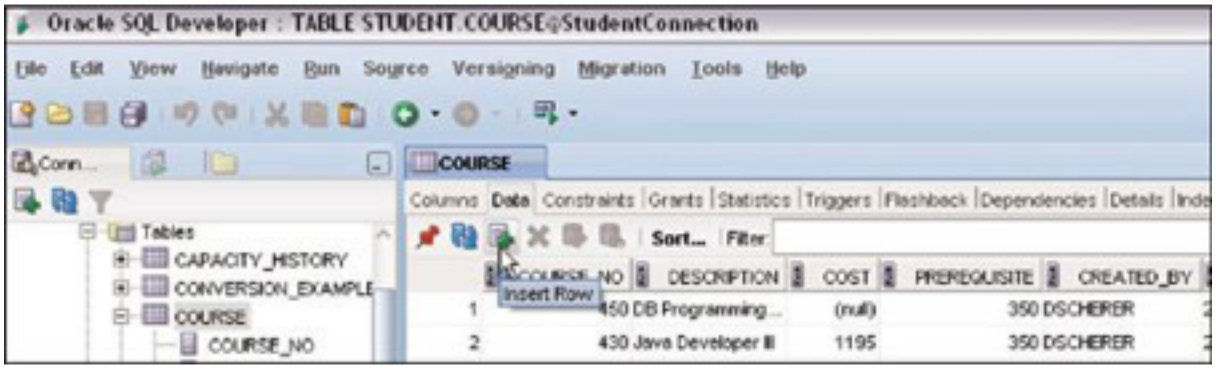
ICON/MENU	DESCRIPTION
Freeze View (pin)	Keeps the tab displayed even if you click on another object in the Connection navigation.
Refresh (circulating arrows)	Updates the data grid with the latest changes.
Insert row (plus sign)	Inserts a new row for new data entry.
Delete row (minus sign)	Deletes the selected row.
Commit (checkmark)	Issues the COMMIT command.
Rollback (undo arrow)	Issues the ROLLBACK command.
Filter	Narrows down the data displayed; similar to a WHERE clause.
Actions	Actions you can perform within the Data tab (for example, export). Some of these are discussed in Chapter 12.

You have two additional menu choices: the Actions menu and the context-sensitive menu available within the data grid, and we will discuss the relevant choices as you work through this lab.

Inserting Data Using the SQL Developer Interface

Instead of writing and issuing a SQL INSERT command, you can add data with SQL Developer's user interface. It allows you to enter data quickly rather than write lengthy SQL statements. You access the Data tab by double-clicking the COURSE table on the Connections navigator. Click the Insert Row icon (see Figure 11.15) to show a new empty row with a plus sign in front of the row number. To enter the data, simply type the data into the appropriate columns.

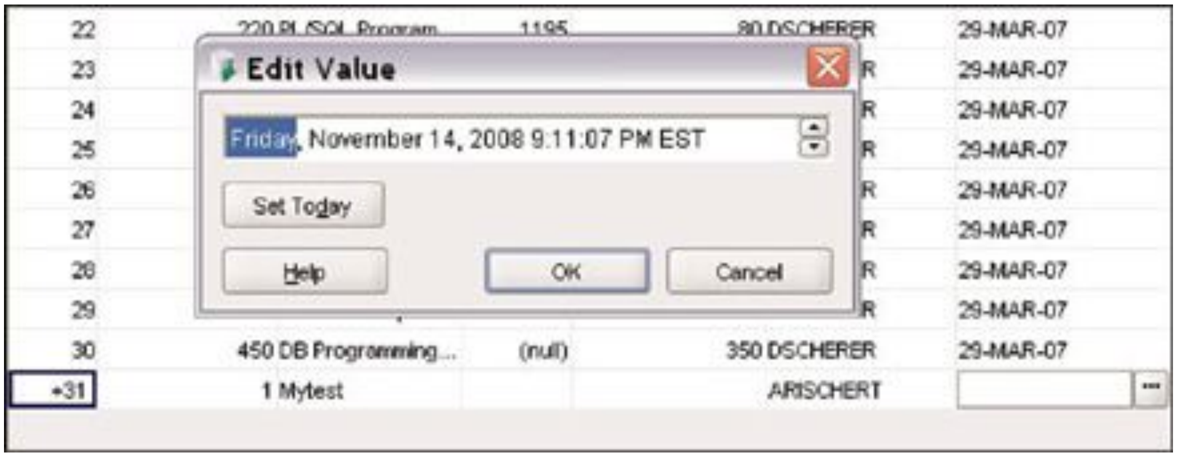
FIGURE 11.15 The Data tab in SQL Developer


Oracle SQL Developer : TABLE STUDENT.COURSE@StudentConnection
File Edit View Navigate Run Source Versioning Migration Tools Help
Conn...
Tables
CAPACITY_HISTORY
CONVERSION_EXAMPLE
COURSE
COURSE_NO
COURSE
Columns Data Constraints Grants Statistics Triggers Flashback Dependencies Details
Sort... Filter
Insert Row

	COURSE_NO	DESCRIPTION	COST	PREREQUISITE	CREATED_BY
1	450	DB Programming ...	(null)	350	DSCHERER
2	430	Java Developer II	1195	350	DSCHERER

If you double-click the cell, you see an ellipsis (…) button, with a box next to it that looks like a drop-down box. When you double-click it, a dialog box appears. If the column is of DATE data type, the dialog box shows today's date and time (see Figure 11.16). You can simply accept it or change the date. SQL Developer enforces a valid date entry.

FIGURE 11.16 Inserting data using the SQL Developer Data tab



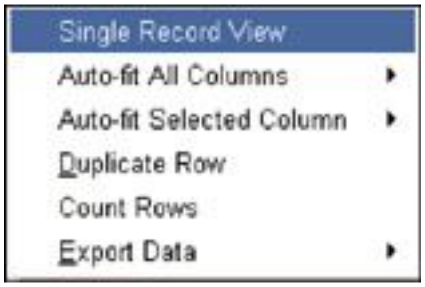
22	220 PL/SQL Program	1195	80 DSCHERER	29-MAR-07
23			R	29-MAR-07
24			R	29-MAR-07
25			R	29-MAR-07
26			R	29-MAR-07
27			R	29-MAR-07
28			R	29-MAR-07
29			R	29-MAR-07
30	450 DB Programming...	(null)	350 DSCHERER	29-MAR-07
+31	1 Mytest		ARISCHERT	

Edit Value

Friday, November 14, 2008 9:11:07 PM EST

A right-click within the data grid reveals a number of context-sensitive menu choices (see Figure 11.17 and Table 11.3). Some of these choices have been described in Chapter 2, "SQL: The Basics," and you will have already found them useful for displaying the data in the manner you like.

FIGURE 11.17 Context menu on the Data grid



You can copy and paste data from existing rows to create the new row; be sure to modify the primary key, otherwise your insert will not be successful. Or if you right-click the Data tab grid, you can choose to duplicate an existing row with the Duplicate Row menu option (see Figure 11.7).

TABLE 11.3 Data Grid Context Menu Choices

MENU CHOICE	DESCRIPTION
Single Record View	Edit and display one record at a time.
Auto-fit All Columns	Adjust the column width according to the submenu choices, which are either by header, by data, or by best fit.
Auto-fit Selected Columns	Adjust selected columns according to the same submenu choices.
Duplicate Row	Copy the row; useful if you need to create another row with similar values.
Count Rows	Count the number of rows in the data grid.
Export	Export data into a variety of formats.

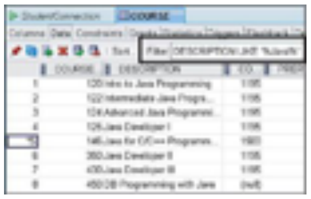
Updating Records

Using the Data tab, you can perform updates directly in the data grid. This works well when you need to make simple data modifications affecting one or a few rows. The Filter box permits you to narrow down the records you want to update. The filter functionality performs much like a WHERE clause, without the WHERE keyword.

Figure 11.18 shows the filter used to narrow down the result set to courses with the word Java in the DESCRIPTION column of the COURSE table. Notice that a change to the COST column was performed for course number 146. As a result, the row number on the left of the grid is flagged with a change asterisks (*) after the change.

Using the Data tab is a great way to perform an update quickly, but it does not eliminate the need to learn about the SQL UPDATE command altogether. If you want to modify large numbers of records with complex conditions, using the SQL UPDATE command is the best way to perform changes.

FIGURE 11.18 Updated record in SQL Developer grid, as indicated by the asterisk



StudentConnection | COURSE

Columns | Data | Constraints | Grants | Statistics | Triggers | Flashback

Sort | Filter: DESCRIPTION LIKE '%Java%'

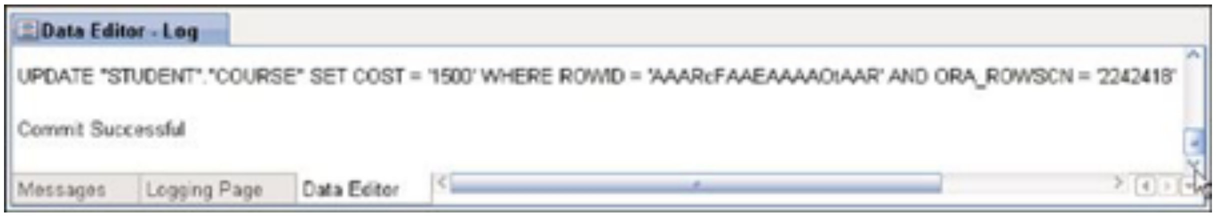
	COURSE...	DESCRIPTION	CO...	PREREQ
1	120	Intro to Java Programming	1195	
2	122	Intermediate Java Progra...	1195	
3	124	Advanced Java Programmi...	1195	
4	125	Java Developer I	1195	
[illegible]	146	Java for C/C++ Programm...	[illegible]	
6	350	Java Developer II	1195	
7	430	Java Developer III	1195	
8	450	DB Programming with Java	(null)	

On the bottom of the SQL Developer screen is the Data Editor - Log tab. This tab records all changes that occur in the Data tab and translates them into SQL commands. When you review the statement in Figure 11.19, you find that the COURSE table is prefixed with the STUDENT schema name; this ensures that the correct table owned by the correct user account is updated. While the connection you are using may be called the StudentConnection, the login name and owner of the COURSE table is the STUDENT user.

The WHERE clause uses the ROWID pseudocolumn to uniquely identify which record is updated. (ROWID is discussed in greater detail in Chapter 13.) Another new column is the ORA_ROWSCN pseudocolumn, and its values represent the SCN for the given row. This pseudocolumn is not shown on the Data tab grid, but it is included in the update to ensure that the row has not changed since it was retrieved.

You cannot use the statements to collect and perform changes to another database instance because the ROWID and ORA_ROWSCN numbers will be different.

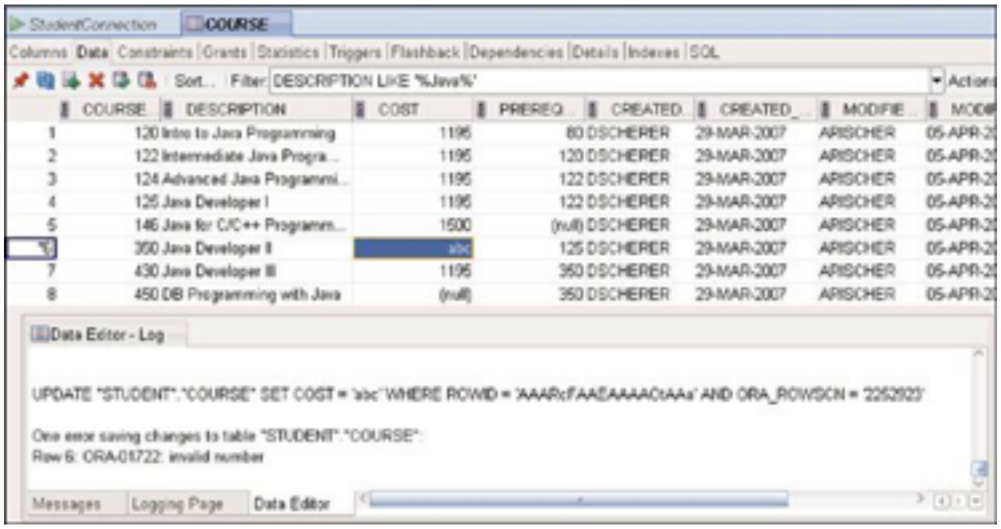
FIGURE 11.19 The Data Editor - Log results



```
UPDATE "STUDENT"."COURSE" SET COST = '1500' WHERE ROWID = 'AAARcFAAEAAAAOtAAR' AND ORA_ROWSCN = '2242418'

Commit Successful
```

After you modify any data in the grid and click the Commit icon, the log lets you know whether the transaction was successful. If an error occurs, you see an 492
Oracle error message. Figure 11.20 shows an attempt to 493
enter an invalid number into the COST column of the COURSE table. Oracle rolls back the change, as indicated with the ORA-01722 error message.

FIGURE 11.20 Error in the Data Editor - Log



Filter: DESCRIPTION LIKE '%Java%'

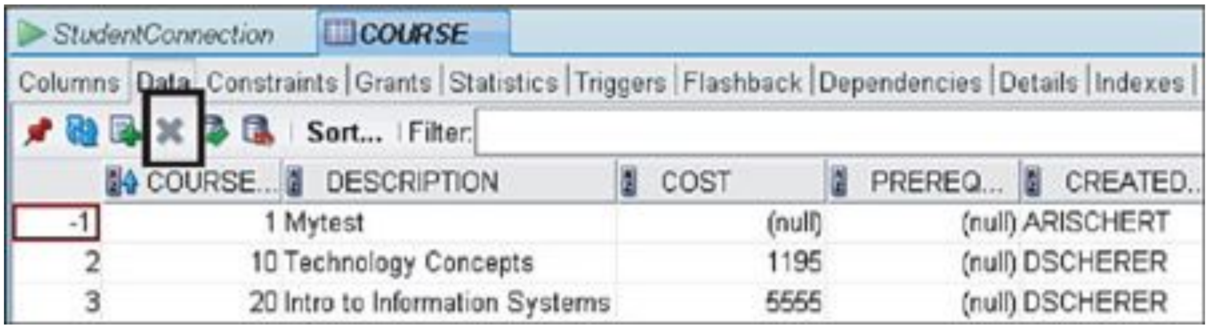
	COURSE...	DESCRIPTION	COST	PREREQ...	CREATED...	CREATED_...	MODIFIE...	MOD
1	120	Intro to Java Programming	1195	80	DSCHERER	29-MAR-2007	ARISCHER	05-APR-2
2	122	Intermediate Java Progra...	1195	120	DSCHERER	29-MAR-2007	ARISCHER	05-APR-2
3	124	Advanced Java Programmi...	1195	122	DSCHERER	29-MAR-2007	ARISCHER	05-APR-2
4	125	Java Developer I	1195	122	DSCHERER	29-MAR-2007	ARISCHER	05-APR-2
5	146	Java for C/C++ Programm...	1500	(null)	DSCHERER	29-MAR-2007	ARISCHER	05-APR-2
	350	Java Developer II	abc	125	DSCHERER	29-MAR-2007	ARISCHER	05-APR-2
7	430	Java Developer III	1195	350	DSCHERER	29-MAR-2007	ARISCHER	05-APR-2
8	450	DB Programming with Java	(null)	350	DSCHERER	29-MAR-2007	ARISCHER	05-APR-2

```
UPDATE "STUDENT"."COURSE" SET COST = 'abc' WHERE ROWID = 'AAARcFAAEAAAACtAAa' AND ORA_ROWSCN = '2252923'

One error saving changes to table "STUDENT"."COURSE":
Row 6: ORA-01722: invalid number
```

Deleting Records

To remove records from a table, click the Delete icon on the Data tab; the record will be marked for deletion, as indicated by the negative number (see Figure 11.21). The change is made permanent when you click the Commit icon or rolled back when you click the Rollback icon.

FIGURE 11.21 Deleted row in SQL Developer's Data tab



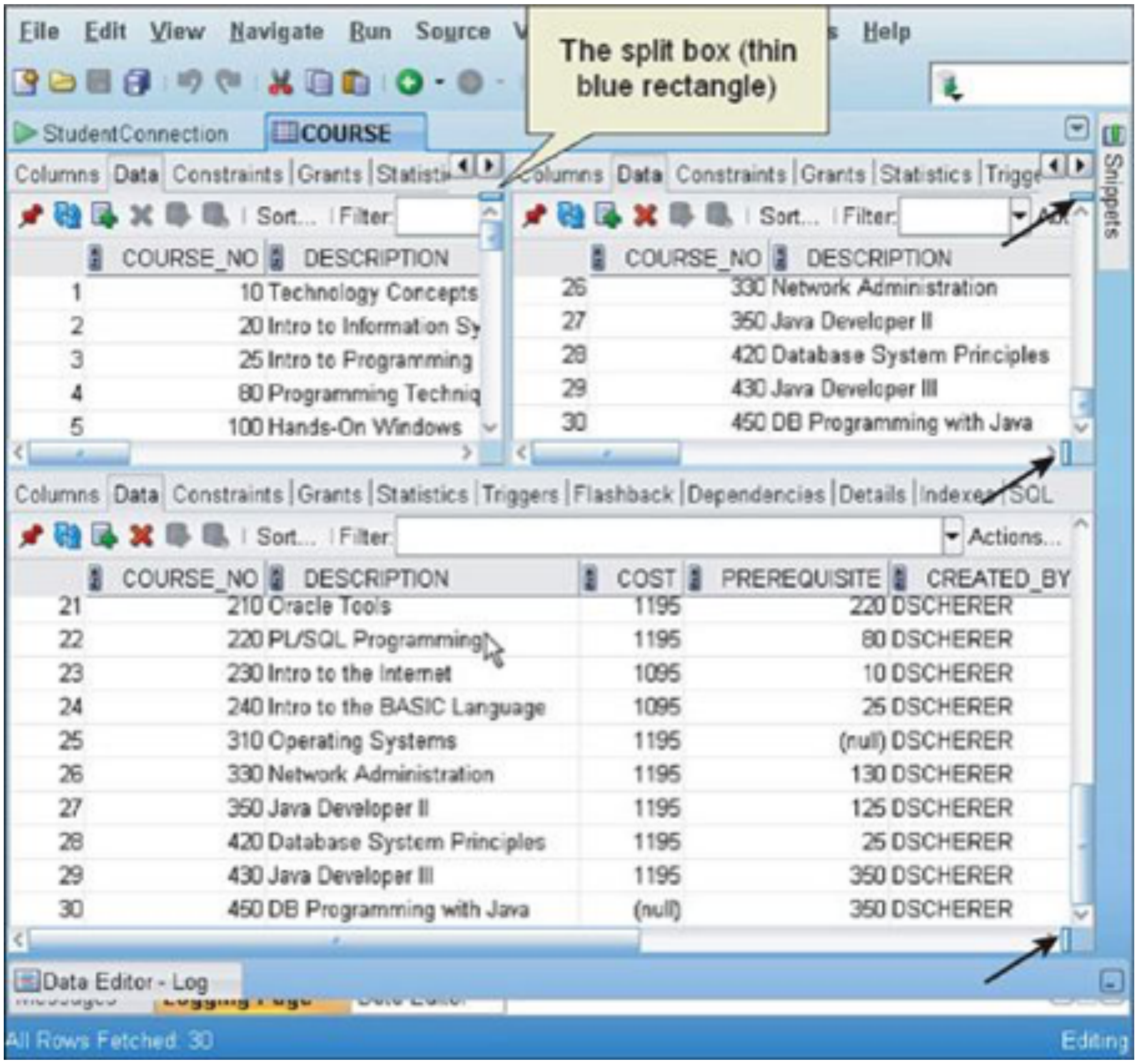
	COURSE...	DESCRIPTION	COST	PREREQ...	CREATED..
-1	1	Mytest	(null)	(null)	ARISCHERT
2	10	Technology Concepts	1195	(null)	DSCHERER
3	20	Intro to Information Systems	5555	(null)	DSCHERER

Splitting the Display Vertically and Horizontally

In the Data tab, you can split the display vertically and horizontally to scroll independently. You can perform filtering and sorts within each grid. The split box, a thin blue rectangle on the top and bottom of the screen, located on the right of the scrollbar, allows you to perform this action. Figure 11.22 shows the data of the

COURSE table split into three different display screens. To remove the split, drag the screen back into the direction of the respective split box.

FIGURE 11.22 The Data tab, split horizontally and vertically


File Edit View Navigate Run Source Help
The split box (thin blue rectangle)
StudentConnection
COURSE
Columns Data Constraints Grants Statistics Triggers Flashback Dependencies Details Indexes SQL
Sort... Filter: Actions...
Snippets
Data Editor - Log
Editing

	COURSE_NO	DESCRIPTION
1	10	Technology Concepts
2	20	Intro to Information Sy
3	25	Intro to Programming
4	80	Programming Techniq
5	100	Hands-On Windows

	COURSE_NO	DESCRIPTION
26	330	Network Administration
27	350	Java Developer II
28	420	Database System Principles
29	430	Java Developer III
30	450	DB Programming with Java

	COURSE_NO	DESCRIPTION	COST	PREREQUISITE	CREATED_BY
21	210	Oracle Tools	1195	220	DSCHERER
22	220	PL/SQL Programming	1195	80	DSCHERER
23	230	Intro to the Internet	1095	10	DSCHERER
24	240	Intro to the BASIC Language	1095	25	DSCHERER
25	310	Operating Systems	1195	(null)	DSCHERER
26	330	Network Administration	1195	130	DSCHERER
27	350	Java Developer II	1195	125	DSCHERER
28	420	Database System Principles	1195	25	DSCHERER
29	430	Java Developer III	1195	350	DSCHERER
30	450	DB Programming with Java	(null)	350	DSCHERER

All Rows Fetched: 30

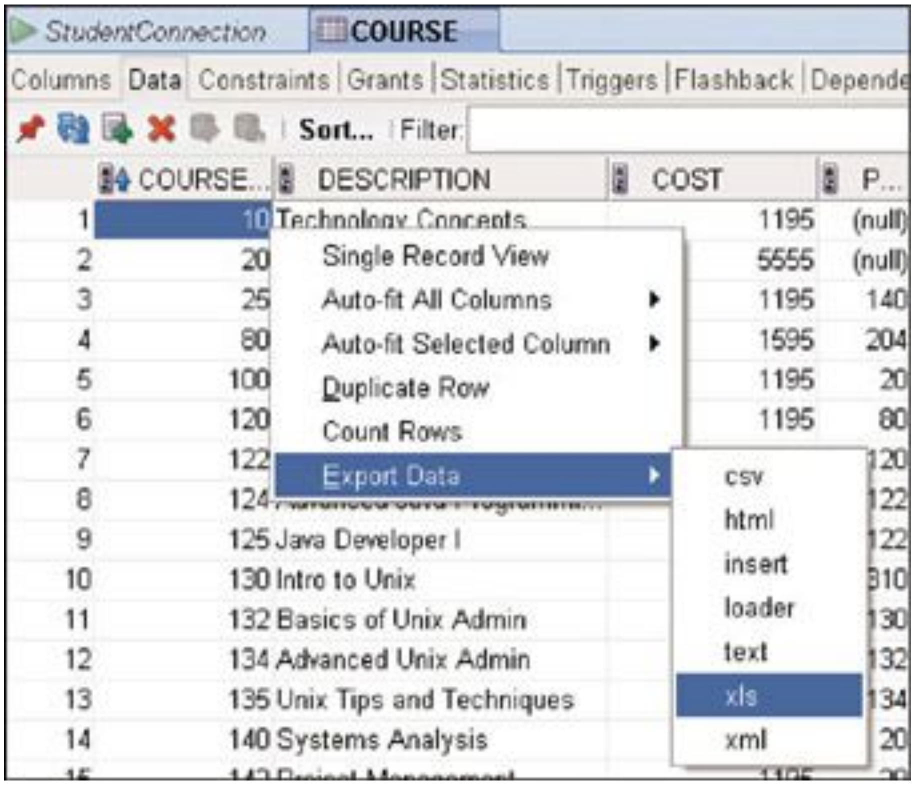
Exporting Data

On the Data tab's grid or within a SQL query's Results window, when you right-click, you get a context menu, and one of the options is Export. Alternatively, you can get to the Export Data menu choice by clicking the Actions menu item. As you can see, there are many different ways to perform the same task in SQL Developer.

The Export Data option allows you to export data in a variety of formats, including Excel and HTML (see Figure 11.23). Another great option is the generation of INSERT SQL statements; you can create SQL scripts that you can then use to re-create the data. The CSV option creates a comma-separated value (CSV) file, a common data format that is useful for transfer between different databases. The loader choice creates a SQL*Loader file along with the data; you can use this file to transfer data into another Oracle database.

As part of each export, you can restrict the columns and the rows you want to export. You can narrow the export criteria by using the Columns and Where tabs that are shown after you choose the Export menu option.

FIGURE 11.23 Export Data menu option



	COURSE...	DESCRIPTION	COST	P...
1	10	Technology Concepts	1195	(null)
2	20		5555	(null)
3	25		1195	140
4	80		1595	204
5	100		1195	20
6	120		1195	80
7	122			120
8	124			122
9	125	Java Developer I		122
10	130	Intro to Unix		310
11	132	Basics of Unix Admin		130
12	134	Advanced Unix Admin		132
13	135	Unix Tips and Techniques		134
14	140	Systems Analysis		20

In the Tools menu, you can also find the Database Export menu option. It allows you to selectively generate the DDL and DML not just for one table at a time but for multiple users, tables, or objects types.

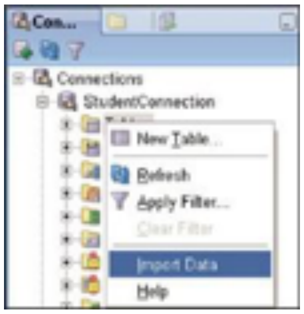
If you need to move larger volumes of data, particularly between two Oracle database instances, you might want to consider using the COPY command (select Tools, Database Copy). Unlike the previously mentioned options, which create files containing the data, this command immediately copies the appropriate information to the Oracle target database.

Outside the SQL Developer tool, Oracle provides a database utility called Data Pump that allows you to export individual tables, individual users, or the entire database, in binary format. DBAs typically use this utility for data transfer between Oracle databases or as a backup choice.

Importing Data

SQL Developer allows import of data. You can choose either Excel or CSV files and import into an existing table or create a new table. If you want to import to a nonexistent table, you right-click the Tables node below the respective connection (see Figure 11.24). To import into an existing table, right-click on the specific table to see the Import Data menu option.

FIGURE 11.24 Import Data menu option

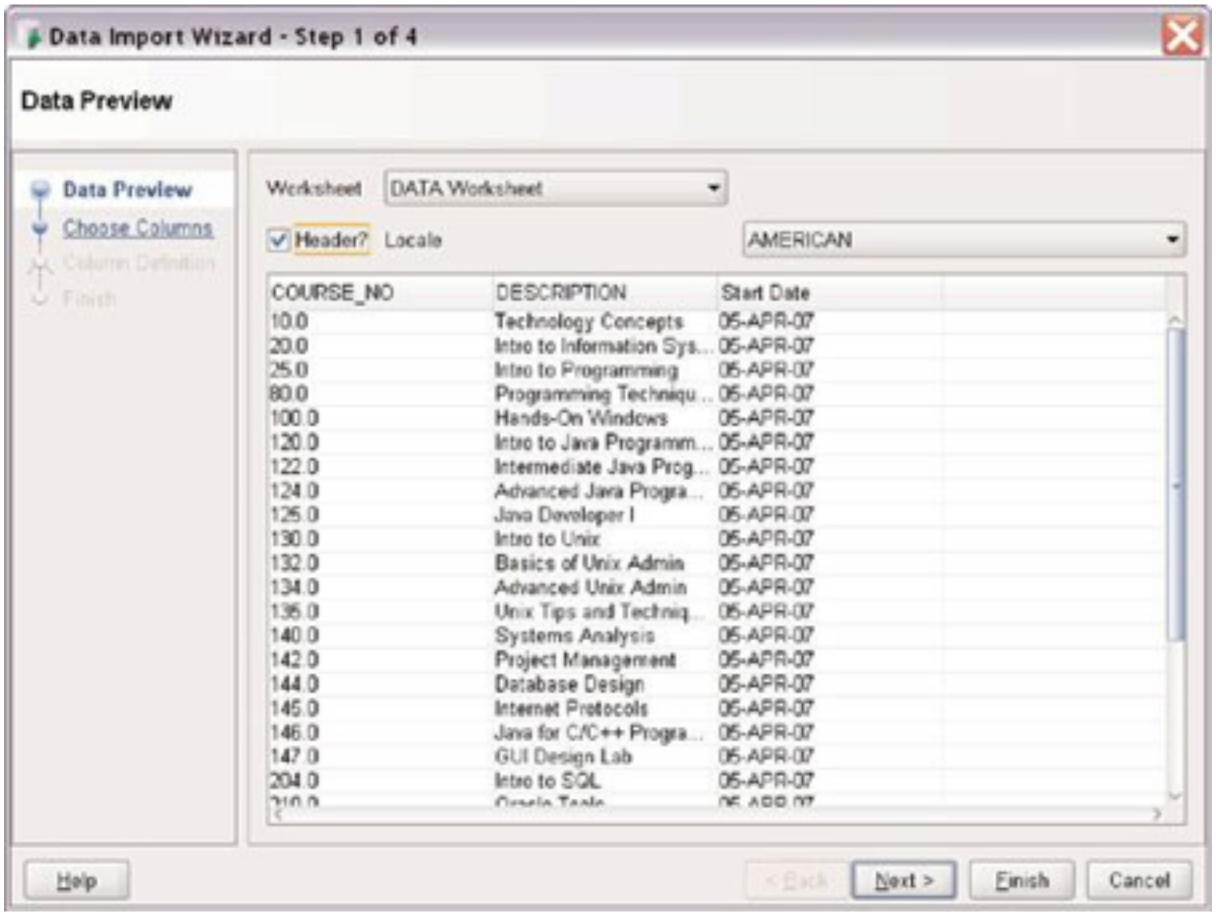


After you specify a file location and name, the Data Import Wizard appears. This wizard guides you through a series of steps to ensure that the data is correctly transferred into an Oracle table. In the example shown in Figure 11.25, an Excel spreadsheet was chosen to be imported. The first step in the wizard is to provide a preview of the data. You choose the Excel Worksheet tab name and identify whether the first row is the header row or data.

The second step is to choose the columns you want to import. The third step is to map the chosen source columns to the appropriate columns in the Oracle target table (see Figure 11.26). If you are choosing to import into a new table, enter a target table name and modify the suggested target column names and data types.

The last step is to verify the import parameters. If an error occurs, you must go back to the previous steps and correct the issue reported. When all is ready, you can either import the data into the table or click the box labeled Send to Worksheet, which opens a SQL Developer Worksheet window that contains the appropriate SQL INSERT statements.

FIGURE 11.25 Import Wizard

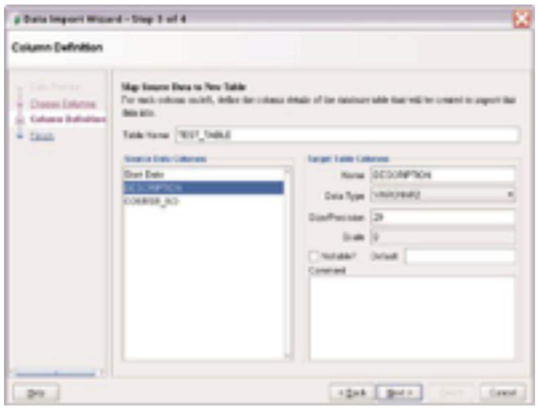


Worksheet: DATA Worksheet

[✓ Header?] Locale: AMERICAN

COURSE_NO	DESCRIPTION	Start Date
10.0	Technology Concepts	05-APR-07
20.0	Intro to Information Sys...	05-APR-07
25.0	Intro to Programming	05-APR-07
80.0	Programming Techniqu...	05-APR-07
100.0	Hands-On Windows	05-APR-07
120.0	Intro to Java Programm...	05-APR-07
122.0	Intermediate Java Prog...	05-APR-07
124.0	Advanced Java Progra...	05-APR-07
125.0	Java Developer I	05-APR-07
130.0	Intro to Unix	05-APR-07
132.0	Basics of Unix Admin	05-APR-07
134.0	Advanced Unix Admin	05-APR-07
135.0	Unix Tips and Techniq...	05-APR-07
140.0	Systems Analysis	05-APR-07
142.0	Project Management	05-APR-07
144.0	Database Design	05-APR-07
145.0	Internet Protocols	05-APR-07
146.0	Java for C/C++ Progra...	05-APR-07
147.0	GUI Design Lab	05-APR-07
204.0	Intro to SQL	05-APR-07

FIGURE 11.26 Map the source to the target columns



Map Source Data to New Table

[illegible]

Table Name: TEST_TABLE

Source Data Columns: Start Date, DESCRIPTION, COURSE_NO

Target Table Columns: Name: DESCRIPTION; Data Type: VARCHAR2; Size/Precision: 29; Scale: 0; Nullable? Default: ; Comment:

LAB 11.3 EXERCISES

a) In the ZIPCODE table, change the city name for zip code 07024 from Ft. Lee to Fort Lee, using the Data tab. Update the MODIFIED_BY column with your name and the MODIFIED_DATE column with the current date and time. Commit the change. Describe the steps you performed.

b) Export the columns COURSE_NO and DESCRIPTION from the COURSE table to an Excel file format. Choose only those records where the prerequisite is not null. What do you observe about the SQL tab contained in the Excel file?

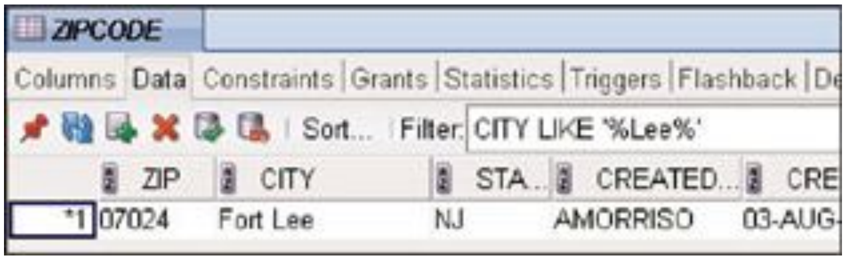
c) Export the data of the MEETING table in the INSERT format. Then execute the resulting INSERT statements.

LAB 11.3 EXERCISE ANSWERS

a) In the ZIPCODE table, change the city name for zip code 07024 from Ft. Lee to Fort Lee, using the Data tab. Update the MODIFIED_BY column with your name and the MODIFIED_DATE column with the current date and time. Commit the change. Describe the steps you performed.

ANSWER: Using the Data tab, you can narrow down the result set with the filter. Figure 11.27 shows the chosen criteria. After you change the city name, you see the asterisk (*).

FIGURE 11.27 The changes performed in the Data tab



Filter: CITY LIKE '%Lee%'

	ZIP	CITY	STA...	CREATED...	CRE
*1	07024	Fort Lee	NJ	AMORRISO	03-AUG-

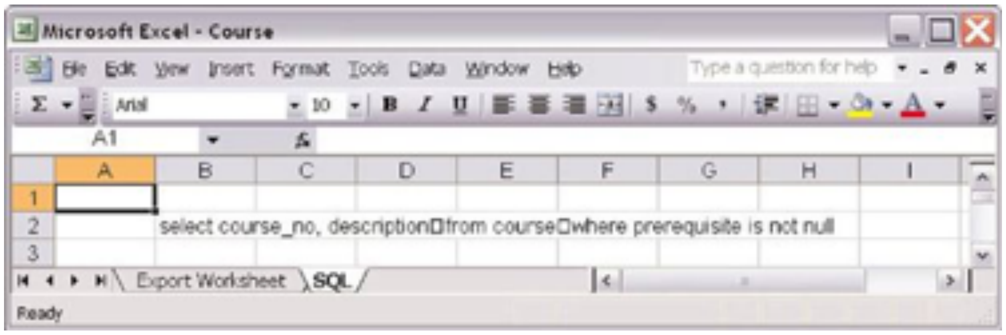
You perform the change to the MODIFIED_DATE column by double-clicking the column's cell and clicking the Set Today button. Enter your name for MODIFIED_BY. Click the Commit icon to make the change permanent.

You can refresh the display when you make changes to the Data tab by clicking the Refresh icon.

b) Export the columns COURSE_NO and DESCRIPTION from the COURSE table to an Excel file format. Choose only those records where the prerequisite is not null. What do you observe about the SQL tab contained in the Excel file?

ANSWER: The export can be accomplished via the Export Data menu. To narrow down the result set, use the Columns and Where tabs shown at the top of the Export Data window (see Figure 11.28). Alternatively, you can execute a SQL statement and then choose the Export menu option from the Results window.

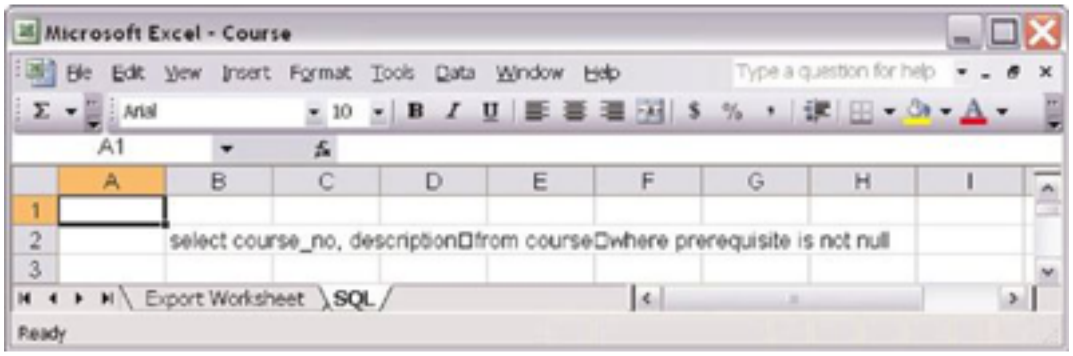
FIGURE 11.28 Export Data options



```
select course_no, description from course where prerequisite is not null
```

The data is shown the Export Worksheet tab in the Excel spreadsheet. The Excel file also contains a SQL tab (see Figure 11.29), which shows the SQL used to select the data in the exported file.

FIGURE 11.29 SQL tab within an Excel spreadsheet, showing the SQL that was used to export the data



```
select course_no, description□from course□where prerequisite is not null
```

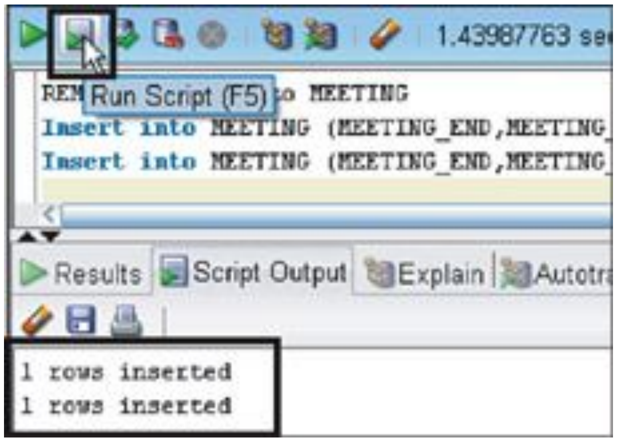
c) Export the data of the MEETING table in the INSERT format. Then execute the resulting INSERT statements.

ANSWER: When exporting data, you can either save the data to a file or to the Clipboard.

Because the table you're working with here contains only two rows, you can save it to the Clipboard and then paste the records into a SQL Worksheet window. Or, if you save it to a file, you can open the script file within SQL Developer; it will be placed in its own SQL Worksheet window.

To execute the two SQL INSERT statements, click the Run Script icon or press F5; the result of the script is displayed in the Script Output window (see Figure 11.30). You can roll back the changes afterward.

FIGURE 11.30 Result of insert script execution



```
REM         o MEETING
Insert into MEETING (MEETING_END,MEETING_
Insert into MEETING (MEETING_END,MEETING_
```

```
1 rows inserted
1 rows inserted
```

Lab 11.3 Quiz

In order to test your progress, you should be able to answer the following questions.

1) The Data tab allows you to perform all the major DML commands, using SQL Developer's user interface.

______a) True

______b) False

2) To extract data containing multiple Oracle user accounts, you can use SQL Developer's Tools, Database Export menu.

______a) True

______b) False

3) The Auto-fit All Columns on Header menu option in the data grid adjusts the width of the column headers.

______a) True

______b) False

4) The filter box in the Data tab requires the entry of the WHERE keyword.

______ **a)** True

______**b)** False

5) SQL Developer's Export Data option allows you to selectively export certain rows and columns.

______**a)** True

______**b)** False

ANSWERS APPEAR IN APPENDIX A.

501

502

WORKSHOP

The projects in this section are meant to prompt you to utilize all the skills you have acquired throughout this chapter. The answers to these projects can be found at the companion Web site to this book, located at **www.oraclesqlbyexample.com**.

1) Write and execute two INSERT commands to create rows in the ZIPCODE table for the following two cities: Newton, MA 02199 and Cleveland, OH 43011. After your INSERT statements are successful, make the changes permanent.

2) Make yourself a student by writing and executing an INSERT statement to insert a row into the

STUDENT table with data about you. Use one of the zip codes you inserted in exercise 1. Insert values into the columns STUDENT_ID (use the value 900), FIRST_NAME, LAST_NAME, ZIP, REGISTRATION_DATE (use a date that is five days after today), CREATED_BY, CREATED_DATE, MODIFIED_BY, and MODIFIED_DATE. Issue a COMMIT command when you're done.

3) Write an UPDATE statement to update the data about you in the STUDENT table. Update the columns SALUTATION, STREET_ADDRESS, PHONE, and EMPLOYER. Be sure to also update the MODIFIED_DATE column and make the changes permanent.

4) Delete the row you created in the STUDENT table and the two rows you created in the ZIPCODE table. Be sure to issue a COMMIT command afterward. You can perform this action by using a SQL command or SQL Developer's Data tab.

5) Delete the zip code 10954 from the ZIPCODE table by using SQL Developer. Commit your change after you delete the row. Describe the results of your actions.

If you performed the exercises in this chapter, you will have changed data in most of the tables of the STUDENT schema. If you go back to the previous chapters and reexecute those queries, you might find that the results are different than they were before. Therefore, if you want to reload the tables and data, you can run the rebuildStudent.sql script. Refer to the readme file you downloaded from the companion Web site for more information on how to perform this step.