17

CHAPTER

Using DML Statements

In This Chapter

Modifying data in tables and views

Inserting new data

Updating existing data

Deleting data

Substituting a sub-query for a table name

Chapter 16, "Selecting Data with SQL," covers constructing queries, and if you've read that chapter, you understand how to choose particular rows out of a table. You call upon this ability when you modify data in the data-base as well. If you want to change the data in one or more rows of a table, you must know how to specify which rows you are after. This chapter describes how to modify data using three SQL commands: `INSERT`, `UPDATE`, and `DELETE`. Each command has several variations, which will be illustrated with examples.

Modifying Data in Tables and Views

The real power of SQL derives from its operation on many rows of data at one time. A single line of SQL code can change every row in a table. You can use the same techniques you use to gather many rows together for a report to modify many rows at the same time. The three basic commands for data manipulation are `UPDATE`, `INSERT`, and `DELETE`.

You can insert, update, or delete rows from a table or a view. You must have the corresponding privilege (`INSERT` to insert rows, `UPDATE` to modify rows, and `DELETE` to delete rows) on the table or view. Modifying the rows in a view actually modifies the rows in the underlying table. For more information about modifying data in a view, you can read the sidebar.

Modifying Data in a View

Some restrictions apply when using `INSERT`, `UPDATE`, and `DELETE` commands on a view rather than on a table. If the view was created with the `WITH CHECK OPTION`, you can modify data through view only if the resulting data satisfies the view's defining query. You can't insert data into a view created using this option.

You cannot modify data using a view if the view's defining query contains any of the following constructs:

- Certain types of joins
- Set operator (`UNION`, `UNION ALL`, `MINUS`, `INTERSECT`)
- The `GROUP BY` clause
- Group function (`MAX`, `MIN`, `AVG`, and so forth)
- The `DISTINCT` operator

Cross-Reference

For more information about updateable views, including details on the type of joins that are updateable, see Chapter 15, "Working with Views, Synonyms, and Sequences."

Inserting New Data

The `INSERT` command has two forms. The first allows you to supply values for one specific row to be inserted, while the second uses a subquery to return values for new rows. This section describes each variation on using the `INSERT` command to modify data.

Inserting one row

The following example demonstrates the usual way to insert a row into a table:

```
INSERT INTO tablename
   (columnname1, columnname2,...)
   VALUES (value1, value2,...);
```

Replace *tablename* with the name of the table into which you are inserting the row. Replace *columnname1, columnname2,...* with the names of the columns for which you are supplying data. Replace *value1, value2,...* with the actual data that you are inserting. You must supply one value for each column in the column list. Text strings should be enclosed within single quotes. Dates should be enclosed in quotes, and the `TO_DATE` function should be used to explicitly convert them from text to a date value.

Note You can omit the entire list of columns (*columnname1, columnname2*,...) as long as your list of values matches the table's columns exactly.

Suppose that you add a new type of salad to your lunch counter. You need to add a new row to the SALAD_TYPE table for this new salad type. Here's an insert statement you can use to add this new variety:

```
INSERT INTO salad_type
   (salad_type, description_text,
    organic_flag, price_per_pound)
   VALUES ('CEASAR', 'Ceasar Salad', 'YES', 9.50);
```

This insert supplies values for all columns, except LAST_CHANGE_DATE. When you insert a row like this without supplying values for all the columns in the table, Oracle will need to do something with those remaining columns. Usually, it will set those columns to null. The exception is when a default value has been defined. If a default value was defined, and no value was supplied for the column in the INSERT statement, Oracle stores the default value in the column for the new row. See Chapter 13, "Managing Tables," for more information on defining default values for columns in a table.

You can explicitly place null values into a column by using the NULL keyword. For example, the following INSERT statement explicitly sets the ORGANIC_FLAG column to null:

```
INSERT INTO SALAD_TYPE
   (salad_type, description_text,
    organic_flag, price_per_pound, last_change_date)
values ('GREENS','Garden Greens, Dandelions, etc',
        NULL, 2.75, TRUNC(SYSDATE));
```

When you explicitly specify a null value this way, the column will be set to null regardless of any default value that may be defined for that column.

Inserting the results of a query

If you're trying to copy data from one table to another, or if you can derive the data that you want to insert from other tables in the database, you can use a subquery as the source of values for an INSERT statement. This allows you to insert many rows at one time. The basic format for this kind of insert is the following:

```
INSERT INTO tablename (column1, column2, ... ) subquery;
```

You may omit the list of columns if the subquery returns a value for every column in the table. You'll probably want to do this often when you're duplicating the contents of a table.

As an example of this technique, suppose that you acquire a mailing list and that you import that list into your database as a table named MILLIES_MAILING_LIST. Having imported the data, you now want to insert all the rows into your CLIENT table. You can do this by writing an INSERT statement with a subquery on the imported table. For example:

```
SQL> INSERT INTO client
  2  SELECT customer_id+125, name_of_client,
  3         street_address || ' ' || city_state_zip, NULL
  4  FROM millies_mailing_list;

4 rows created.
```

Consider these interesting points about this example:

- ✦ No column list was given in the INSERT statement, meaning that all columns in the CLIENT table must be populated from corresponding columns in the SELECT query.
- ✦ The MILLIES_MAILING_LIST option has a two-column address that is concatenated and stored as one column in the CLIENT table.
- ✦ A value of 125 is added to each new customer ID to avoid possible conflicts with existing data.
- ✦ Since MILLIES_MAILING_LIST did not have a country column, the subquery returned a NULL for that value.

Another use of subqueries in INSERT statements is to generate missing parent records in preparation for creating a foreign key constraint. Say that you intend to create a foreign key from SALAD_TRAY to SALAD_TYPE and that you know you have a number of salad trays defined with invalid types. If you want to create the constraint and yet keep those trays, you can issue an INSERT statement like this:

```
INSERT INTO salad_type tp
   (salad_type, description_text, organic_flag)
   SELECT ty.salad_type, MAX(ty.sample_description), MAX('NO')
   FROM salad_tray ty
   WHERE ty.salad_type NOT IN (
         SELECT tp2.salad_type
         FROM salad_type tp2
         )
   GROUP BY ty.salad_type;
```

This is certainly a mind-bending statement. It actually contains two subqueries, one nested inside the other. The innermost subquery retrieves a list of salad types that have already been defined. These don't need to be redefined, so the parent query excludes those types from the rows that it selects. The GROUP BY clause exists to ensure that only one row is returned for each distinct salad type in the SALAD_TRAY table that hasn't yet been defined in the SALAD_TYPE table. You might think that you could use DISTINCT here, but you can't because salad tray descriptions are all different. The MAX function is applied to the SAMPLE_DESCRIPTION column to arbitrarily choose one salad tray description as the description for the new type being created. All of this data is returned to the INSERT statement, which inserts the new rows into the SALAD_TYPE table.

You can also use the subquery method of inserting to quickly generate large amounts of test data. You can do this by inserting a copy of a table's current data back into that table, repeating the process until you have the volume of data that you are after. Consider this example:

```
SQL> INSERT INTO client
  2     SELECT * FROM client;

9 rows created.

SQL> /

18 rows created.

SQL> /

36 rows created.
```

Each time the forward slash is used to execute the statement, the number of rows in the table is doubled.

Updating Existing Data

The UPDATE statement enables you to modify existing table data. There are two forms of the UPDATE statement. One form allows you to update a table and to explicitly provide new values for columns in rows being modified. The second form allows you to base the new values on a subquery. Both of these forms are described in this section.

Performing a standard update

The most common form of the UPDATE statement simply sets one or more columns to a new value. The basic format of the statement looks like this:

```
UPDATE tablename
SET columnname = expression,
    columnname2 = expression2,
    ...
WHERE conditions;
```

Replace *tablename* with your table's name and replace *columnname*, *columnname2*, and so forth, with the names of the columns you want to modify. The *expression* and *expression2* values represent the new values for those columns. These expressions may be a literal value, another column, or the result of any valid SQL expression. The datatype of the expression, of course, must be compatible with the column being changed. Replace the WHERE clause with any valid WHERE clause to specify which rows you want to modify. You can update all the rows in a table by omitting the WHERE clause entirely.

The following example demonstrates where you add *red* to the list of colors for a specific fish in the FISH table. The new value of the column is actually computed

by an expression that references both the old value and a text constant.

```
UPDATE fish
SET colors = colors || ',red'
WHERE name_of_fish = 'Wesley';
```

You can set a column's value to null by using the NULL keyword as the new value. The following example sets DEATH_DATE to NULL for all rows in the FISH table:

```
UPDATE fish
SET death_date = NULL;
```

This statement updates every row in the table because no WHERE clause is supplied.

Using UPDATE statements based on subqueries

The second variation on the UPDATE statement has a great amount of flexibility because it uses a subquery — a query within the UPDATE statement. The basic format is shown in the following example:

```
UPDATE tablename
   SET (columnname, columnname2, ...) =
      (SELECT columnname3, columnname4, ...
       FROM tablename2
       WHERE conditions)
WHERE conditions;
```

Replace *tablename* with the table to be updated. Replace *columnname*, *columnname2*, and so forth, with a list of columns to be updated. The subquery is everything inside the parentheses after the equal sign. You can place any query inside the subquery as long as the columns in the subquery (*columnname3*, *columname4*, . . .) match the columns in the UPDATE statement's column list.

A subquery used in the SET clause of an UPDATE statement may be correlated or noncorrelated. Correlated subqueries offer more flexibility and are often the reason that you want a subquery in the first place.

Note Whenever you use a subquery in the SET clause of an update statement, you must be sure that the subquery returns exactly one row. If the subquery returns no rows at all, then all the columns being updated will be set to null. If the subquery returns more than one row, Oracle will return an error.

Using a Noncorrelated Subquery Example

When you use a noncorrelated subquery in an UPDATE statement, the query will return the same value for each row being updated. This typically is useful only if just one row is being updated. The following example shows such an update. The MONHLY_SALES table is being updated to reflect the sales for the month of January 1996:

```
UPDATE MONTHLY_SALES
SET (SALES_AMOUNT) =
        (SELECT SUM(SALES_AMOUNT)
         FROM DAILY_SALES
         WHERE SALES_DATE BETWEEN
               TO_DATE('01-JAN-1996','DD-MON-YYYY')
               AND TO_DATE('31-JAN-1996','DD-MON-YYYY'))
WHERE SALES_MONTH = TO_DATE('JAN-1996','MON-YYYY');
```

Remember, for an update of this type to succeed, the row being updated must already exist. Don’t confuse this with an INSERT...SELECT FROM statement.

Using Correlated Subqueries to Update Data

A subquery is said to be correlated when it references values from the table being updated. When this occurs, Oracle must execute the subquery once for each row that you are changing. This represents the most versatile use of subqueries in an UPDATE statement because it not only allows you to update a large number of rows, but it also allows you to use values from those rows to derive the new values for the columns that you are changing.

The following statement shows an example of a correlated subquery being used with an UPDATE statement. It goes through and changes the descriptions for all the salad tray records. The new description for each salad tray is taken from the record in the SALAD_TYPE table that corresponds to the type of salad tray being modified. Here’s the UPDATE statement that does all this work for you:

```
UPDATE salad_tray tray
SET sample_description = (
      SELECT description_text
      FROM salad_type type
      WHERE type.salad_type = tray.salad_type)
WHERE tray.salad_type IS NOT NULL;
```

Notice the use of table aliases in this command. The SALAD_TRAY table is assigned the alias TRAY, and the SALAD_TYPE table is assigned the alias TYPE. These aliases are used in the WHERE clause of the subquery to qualify the column names being used. They make it clear which table each column belongs to. The WHERE clause in the main query restricts the update so that it is only applied to SALAD_TRAY rows that actually have an associated salad type code.

Using correlated UPDATE statements like this allows you to update a lot of data with one statement. The keys to doing this successfully lie in creating a set of rules for deriving the new values from existing data, and in writing WHERE clauses that can be used to correlate those new values with the proper rows in the table being modified.

Tip

If you're going to issue a correlated subquery like this to do a mass update of your data, be sure to think through thoroughly the implications of what you are doing. You might want to run some SELECTs first to be certain that you are selecting the right records to be updated. You might also run some SELECTs to verify that your subqueries are working properly. When doing a mass update like this, it's easy to overlook oddities in your data that can cause problems.

Deleting Data

You use the DELETE command to delete rows from a table. The basic form of the command is relatively simple and looks like this:

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

Replace *tablename* with an actual table name. Replace *condition* with any valid WHERE clause conditions. These may be simple comparison expressions, or they may be nested subqueries.

The following DELETE statement uses a simple WHERE clause with no subqueries and deletes all fruit-salad trays from the SALAD_TRAY table:

```
DELETE
FROM salad_tray
WHERE salad_type = 'FRUIT';
```

Tip

Test your DELETE statements by creating SELECT statements using the same WHERE clause. Then, execute those SELECT statements. With this approach, you can see which rows are selected for deletion before you actually delete them.

You can use subqueries to good effect in DELETE statements. The following example uses a noncorrelated subquery to find and delete all salad types for which no trays exist:

```
DELETE
FROM salad_type
WHERE salad_type NOT IN (
   SELECT salad_type
   FROM salad_tray);
```

Note

You can test these DELETE statements against the sample database, but you might want to roll back the delete afterwards, so as not to delete all your sample data.

Correlated subqueries are useful as well. The following example shows a DELETE similar to the previous one, but it uses a correlated subquery. It will delete any salad tray records containing invalid salad-type values:

```
DELETE
FROM salad_tray tray
WHERE NOT EXISTS (
   SELECT *
   FROM salad_type type
   WHERE tray.salad_type = type.salad_type);
```

If you intend to add a foreign key constraint from the SALAD_TRAY table to the SALAD_TYPE table, you need to be sure that all the salad trays have valid types. Otherwise, Oracle won't allow you to add the constraint. A DELETE statement such as the one shown in the previous example allows you to delete any records that have invalid type codes, allowing you to create the constraint.

Tip

If you need to use DELETE for all the rows in a table and the table is very large, consider using the TRUNCATE command instead. Truncating a table goes much faster because Oracle doesn't allow a rollback to be done. The tradeoff is that your risk of losing data is increased, because if you truncate the wrong table by mistake, you can't roll back the transaction.

Substituting a Subquery for a Table Name

The actual table or view to be modified is usually listed in an INSERT, UPDATE, or DELETE command. However, you can use a subquery in place of the table or view name. When you do this, the subquery is treated as if it were a view. In fact, it represents an inline view. Therefore, the same rules apply here as when modifying a view. (See the sidebar "Modifying Data in a View" at the beginning of the chapter.)

Here's an example showing an INSERT into an inline view:

```
INSERT INTO
   (SELECT BREAD_NO, BREAD_NAME FROM BREAD)
   VALUES (10,'Poppy Seed');
```

Inline views are really used much more often in SELECT statements than in any of the INSERT, UPDATE, or DELETE statements.

Summary

In this chapter, you learned:

- The ability to write queries helps you write commands to modify data in tables and views.
- You are allowed to use only specific kinds of views when modifying data. When modifying data using a view, the underlying table's data actually gets modified.
- An UPDATE command modifies data in existing rows in a table or view. The UPDATE command has two basic variations: one using literals or expressions to set new values for the columns, and one using a subquery to perform that function.
- The INSERT command adds a new row (or set of new rows) to a table. You may insert one row at a time, or you may use a subquery to insert several rows at once.
- The DELETE command removes rows from a table. Use a WHERE clause to tell Oracle which rows you want to delete. If you leave off the WHERE clause, you'll delete all rows in a table.

✦ ✦ ✦