

CERTIFICATION OBJECTIVE 10.02

Insert Rows into a Table

The simplest form of the INSERT statement inserts one row into one table, using values provided in line as part of the command. The syntax is as follows:

```
INSERT INTO table [(column [,column...])] VALUES (value [,value...]);
```

For example:

```
insert into hr.regions values (10,'Great Britain');
insert into hr.regions (region_name, region_id) values
('Australasia',11);
insert into hr.regions (region_id) values (12);
insert into hr.regions values (13,null);
```

The first of the preceding commands provides values for both the columns of the REGIONS table. If the table had a third column, the statement would fail because it relies upon *positional notation*. The statement does not say which value should be inserted into which column; it relies on the position of the values: their ordering in the command. When the database receives a statement using positional notation, it will match the order of the values to the order in which the columns of the table are defined. The statement would also fail if the column order were wrong; the database would attempt the insertion but would fail because of data type mismatches.

The second command nominates the columns to be populated and the values with which to populate them. Note that the order in which columns are mentioned now becomes irrelevant—as long as the order of the columns is the same as the order of the values.

The third example lists one column, and therefore only one value. All other columns will be left null. This statement would fail if the REGION_NAME column were not nullable. The fourth example will produce the same result, but because there is no column list, a value of some sort must be provided for each column—at the least, a NULL.



It is often considered good practice not to rely on positional notation and instead always to list the columns. This is more work but makes the code self-documenting (always a good idea!) and also makes the code more resilient against table structure changes. For instance, if a column is added to a table, all the INSERT statements that rely on positional notation will fail until they are rewritten to include a NULL for the new column. INSERT code that names the columns will continue to run.

Very often, an INSERT statement will include functions to do type casting or other editing work. Consider this statement:

```
insert into employees (employee_id, last_name, hire_date)
values (1000, 'WATSON', '03-Nov-07');
```

in contrast with this:

```
insert into employees (employee_id, last_name, hire_date)
values (1000, upper('Watson'), to_date('03-Nov-07', 'dd-mon-yy'));
```

The rows inserted with each statement would be identical. But the first will insert exactly the literals provided. It may well be that the application relies on employee surnames being in uppercase—without this, perhaps sort orders will be wrong and searches on surname will give unpredictable results. Also, the insertion of the date value relies on automatic type casting of a string to a date, which is always bad for performance and can result in incorrect values being entered. The second statement forces the surname into uppercase whether it was entered that way or not, and specifies exactly the format mask of the date string before explicitly converting it into a date. There is no question that the second statement is a better piece of code than the first.

The following is another example of using functions:

```
insert into employees (employee_id, last_name, hire_date)
values (1000 + 1, user, sysdate - 7);
```

In the preceding statement, the EMPLOYEE_ID column is populated with the result of some arithmetic, the LAST_NAME column is populated with the result of the function USER (which returns the database logon name of the user), and the HIRE_DATE column is populated with the result of a function and arithmetic: the date seven days before the current system date.

Figure 10-2 shows the execution of the previous three insertions, followed by a query showing the results.

Using functions to preprocess values before inserting rows can be particularly important when running scripts with substitution variables, as they will allow the code to correct many of the unwanted variations in data input that can occur when users enter values interactively.

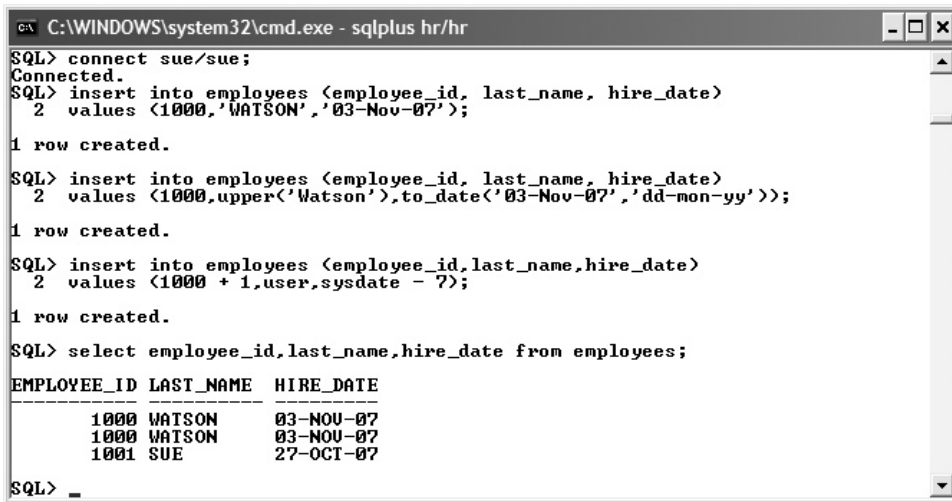
To insert many rows with one INSERT command, the values for the rows must come from a query. The syntax is as follows:

```
INSERT INTO table [ (column [, column...] ) ] subquery;
```

Note that this syntax does not use the VALUES keyword. If the column list is omitted, then the subquery must provide values for every column in the table.

FIGURE 10-2

Using functions
with the INSERT
command



```

C:\WINDOWS\system32\cmd.exe - sqlplus hr/hr

SQL> connect sue/sue;
Connected.
SQL> insert into employees (employee_id, last_name, hire_date)
  2 values (1000,'WATSON','03-Nov-07');

1 row created.

SQL> insert into employees (employee_id, last_name, hire_date)
  2 values (1000,upper('Watson'),to_date('03-Nov-07','dd-mon-yy'));

1 row created.

SQL> insert into employees (employee_id,last_name,hire_date)
  2 values (1000 + 1,user,sysdate - ?);

1 row created.

SQL> select employee_id,last_name,hire_date from employees;

EMPLOYEE_ID LAST_NAME HIRE_DATE
-----
1000 WATSON 03-NOV-07
1000 WATSON 03-NOV-07
1001 SUE 27-OCT-07

SQL> _

```

To copy every row from one table to another, if the tables have the same column structure, a command such as this is all that is needed:

```
insert into regions_copy select * from regions;
```

This presupposes that the table REGIONS_COPY does exist (with or without any rows). The SELECT subquery reads every row from the source table, which is REGIONS, and the INSERT inserts them into the target table, which is REGIONS_COPY.

There are no restrictions on the nature of the subquery. Any query returns (eventually) a two-dimensional array of rows; if the target table (which is also a two-dimensional array) has columns to receive them, the insertion will work. A common requirement is to present data to end users in a form that will make it easy for them to extract information and impossible for them to misinterpret it. This will usually mean denormalizing relational tables, making aggregations, renaming columns, and adjusting data that can distort results if not correctly processed.

Consider a simple case within the HR schema: a need to report on the salary bill for each department. The query will need to perform a full outer join to ensure that any employees without a department are not missed, and that all departments are listed whether or not they have employees. It should also ensure that any null values will not distort any arithmetic by substituting zeros or strings for nulls. This query is perfectly straightforward for any SQL programmer, but when end users attempt to run this sort of query they are all too likely to produce inaccurate results by omitting

the checks. A daily maintenance job in a data warehouse that would assemble the data in a suitable form could be a script such as this:

```
truncate table department_salaries;
insert into department_salaries (department,staff,salaries)
select
    coalesce(department_name,'Unassigned'),
    count(employee_id),
    sum(coalesce(salary,0))
from employees e full outer join departments d
on e.department_id = d.department_id
group by department_name
order by department_name;
```

exam

Watch

Any *SELECT* statement, specified as a subquery, can be used as the source of rows passed to an *INSERT*. This enables insertion of many rows. Alternatively, using the *VALUES* clause will insert one row. The values can be literals or prompted for as substitution variables.

The TRUNCATE command will empty the table, which is then repopulated from the subquery. The end users can be let loose on this table, and it should be impossible for them to misinterpret the contents—a simple natural join with no COALESCE functions, which might be all an end user would do, might be very misleading. By doing all the complex work in the INSERT statement, users can then run much simpler queries against the denormalized and aggregated data in the summary table. Their queries will be fast, too: all the hard work has been done already.

To conclude the description of the INSERT command, it should be mentioned that it is possible to insert rows into several tables with one statement. This is not part of the SQL OCP examination, but for completeness here is an example:

```
insert all
when l=1 then
    into emp_no_name (department_id,job_id,salary,commission_pct,hire_date)
    values (department_id,job_id,salary,commission_pct,hire_date)
when department_id <> 80 then
    into emp_non_sales (employee_id,department_id,salary,hire_date)
    values (employee_id,department_id,salary,hire_date)
when department_id = 80 then
    into emp_sales (employee_id,salary,commission_pct,hire_date)
    values (employee_id,salary,commission_pct,hire_date)
select employee_id,department_id,job_id,salary,commission_pct,hire_date
from employees where hire_date > sysdate - 30;
```

To read this statement, start at the bottom. The subquery retrieves all employees recruited in the last 30 days. Then go to the top. The ALL keyword means that every row selected will be considered for insertion into all the tables following, not just into the first table for which the condition applies. The first condition is `1=1`, which is always true, so every source row will create a row in EMP_NO_NAME. This is a copy of the EMPLOYEES table with the personal identifiers removed, a common requirement in a data warehouse. The second condition is `DEPARTMENT_ID <> 80`, which will generate a row in EMP_NON_SALES for every employee who is not in the sales department; there is no need for this table to have the COMMISSION_PCT column. The third condition generates a row in EMP_SALES for all the salesmen; there is no need for the DEPARTMENT_ID column, because they will all be in department 80.

This is a simple example of a multitable insert, but it should be apparent that with one statement, and therefore only one pass through the source data, it is possible to populate many target tables. This can take an enormous amount of strain off the database.

EXERCISE 10-1

Use the INSERT Command

In this exercise, use various techniques to insert rows into a table.

1. Connect to the HR schema, with either SQL Developer or SQL*Plus.
2. Query the REGIONS table, to check what values are already in use for the REGION_ID column:

```
select * from regions;
```

This exercise assumes that values above 100 are not in use. If they are, adjust the values suggested below to avoid primary key conflicts.

3. Insert a row into the REGIONS table, providing the values in line:

```
insert into regions values (101, 'Great Britain');
```

4. Insert a row into the REGIONS table, providing the values as substitution variables:

```
insert into regions values (&Region_number, '&Region_name');
```

When prompted, give the values 102 for the number, Australasia for the name. Note the use of quotes around the string.

5. Insert a row into the REGIONS table, calculating the REGION_ID to be one higher than the current high value. This will need a scalar subquery:

```
insert into regions values ((select max(region_id)+1
from regions), 'Oceania');
```

6. Confirm the insertion of the rows:

```
select * from regions;
```

7. Commit the insertions:

```
commit;
```

The following illustration shows the results of the exercise, using SQL*Plus:

```

C:\WINDOWS\system32\cmd.exe - sqlplus hr/hr
SQL> insert into regions values (101,'Great Britain');
1 row created.
SQL> insert into regions values (&Region_number,&'Region_name');
Enter value for region_number: 102
Enter value for region_name: Australasia
old 1: insert into regions values (&Region_number,&'Region_name')
new 1: insert into regions values (102,'Australasia')
1 row created.
SQL> insert into regions values
  2 ((select max(region_id)+1 from regions),'Oceania');
1 row created.
SQL> select * from regions;
  REGION_ID REGION_NAME
-----
      101 Great Britain
      102 Australasia
      103 Oceania
         1 Europe
         2 Americas
         3 Asia
         4 Middle East and Africa
7 rows selected.
SQL> commit;
Commit complete.
SQL> _

```