Les09-DML-insert etc

9-1 – p17 in pdf



OBJECTIVES of this topic

9-2

# 3 things you can do with data

# ADD

# CHANGE

# DELETE

After completing this lesson, you should be able to do the following:

• Describe each data manipulation language (DML) statement

• Insert rows into a table - Adding data

• Update rows in a table - Changing content

• Delete rows from a table - deleting data

• Control transactions – COMMIT SAVEPOINT ROLLBACK

Topics Covered

9-3

Adding new rows in a table

– INSERT statement

• Changing data in a table

– UPDATE statement

• Removing rows from a table:

– DELETE statement

– TRUNCATE statement

• Database transactions control using COMMIT, ROLLBACK, and SAVEPOINT

• Read consistency

• FOR UPDATE clause in a SELECT statement

DML – Data Manipulation Language

# What is DML?

The SQL that manipulates data.

Data can be added, changed or deleted

A DML statement is executed when you:

– Add new rows to a table

– Modify existing rows in a table

– Remove existing rows from a table

• A *transaction* consists of a collection of DML statements that form a logical unit of work – such as inserting students registering

EXAMPLE:

Student comes in to pay for tuition

1) Decrease the amount owing to Seneca

2) Increase the total amount collected for the day

3) Store copy of activity (transaction) in a journal

4) Add it to the end of day check list

5) Print a receipt

NOTE:

Steps 1, 2, 3 and 4 must be completed before a transaction is complete.

If any one of them fails, then all actions must be undone.

ADDING A NEW ROW

9-5 graphic

9-6 SQL

**INSERT Statement Syntax**

• Add new rows to a table by using the INSERT statement:

• With this syntax, only one row is inserted at a time.

**INSERT INTO *table* [(*column* [*, column...*])]**

**VALUES *(value* [*, value...*]);**

Example: Inserting New Row

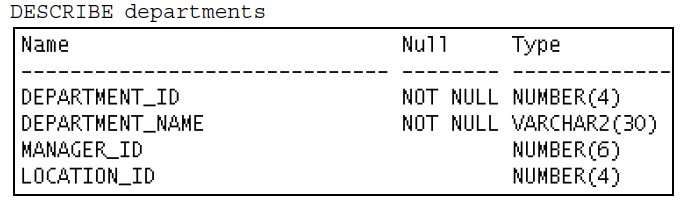
9-7

Insert a new row containing values for each column.

2ways

• List values in the default order of the columns in the table

.



• Optionally, list the columns in the INSERT clause.

**INSERT INTO departments**

**(department\_id, department\_name, manager\_id, location\_id)**

**VALUES (70, 'Public Relations', 100, 1700);**

Note: 4 columns in the INSERT and 4 values

• Enclose character and date values within single quotation marks.

Insert – with NULL values

Implicit method: Omit the column from the column list.

**INSERT INTO departments (department\_id, department\_name) 🡸**

**VALUES (30, 'Purchasing');**

🡺The specific columns were **stated** in the INSERT

That means it knows where it goes. Could do it in different order

Explicit method: Specify the NULL keyword in the VALUES clause.

**INSERT INTO departments**

**VALUES (100, 'Finance', NULL, NULL);**

There are 4 columns in the table

None of the columns were mentioned

so there needs to be 4 columns in the VALUES clause

with data that matches the columns info

INSERT MULTIPLE LINES

-- multiple inserts

insert all

into testing values (2, 'two', 20)

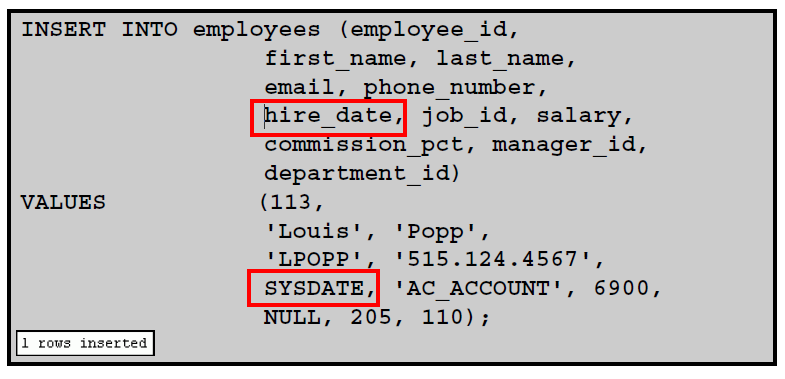
into testing values (3, 'three', 30)

**select 1 from dual; 🡸 because Oracle needs a SELECT for multiple inserts.**

Special Values on INSERT

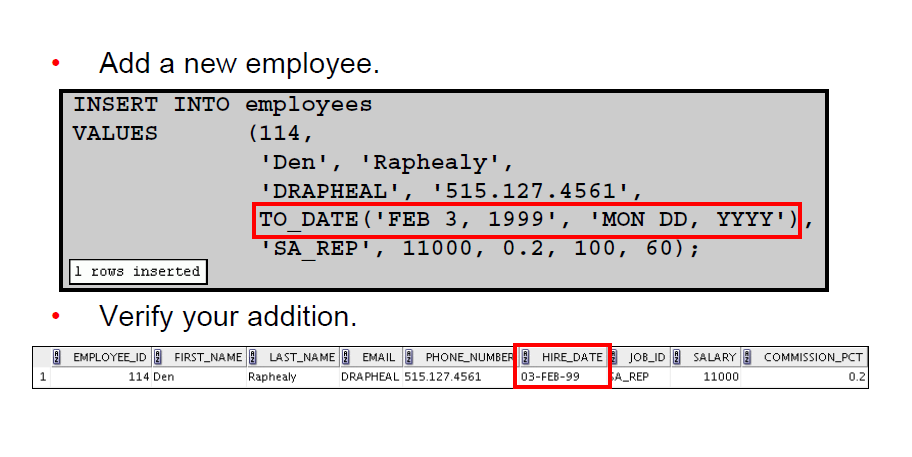
9-9

Put SYSDATE into 🡺 HIRE\_DATE



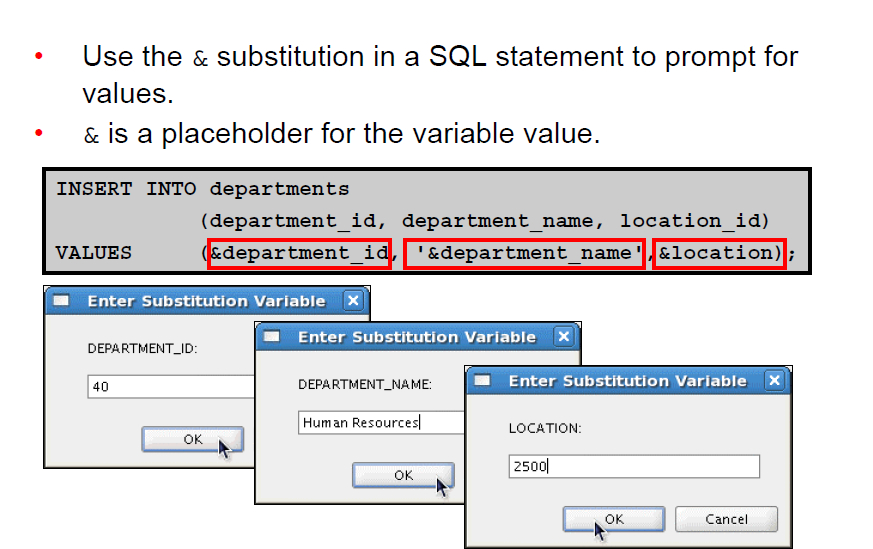
There are other functions like USER available to do the same thing as SYSDATE example

INSERT – with specific DATE and TIME



Might be preferable to use full century rather than let the system determine it with the RR for year

SCRIPT



Can do script in a file and run it

NOTE:

Better of course, is to have proper user screens for input. This is just a poor substitution as we have not learned how to develop front-end user interfaces yet.

INSERT – by COPYING FROM ANOTHER TABLE

9-12

NOTE: Table must exist already

Use a subquery

**INSERT INTO sales\_reps(id, name, salary, commission\_pct)**

**SELECT employee\_id, last\_name, salary, commission\_pct**

**FROM employees**

**WHERE job\_id LIKE '%REP%';**

NOTE:

🡺 No VAUES clause

🡺 Number of columns must match

🡺 Data type must match

EXAMPLE: of complete table

INSERT INTO EMPCOPY

SELECT \*

FROM employees;

Insert into testing

Select employee\_id, last\_name, salary/1000

From employees

NOTE: table EMPCOPY must exist

UPDATE – modify/change data

# Modify existing values

# Update 1 or more rows at a time

# Generic syntax

**UPDATE *table***

**SET *column* = *value* [, *column* = *value, ...*]**

**[WHERE *condition*];**

# Values for a specific row to change

You did this in lab06

**UPDATE employees**

**SET department\_id = 50**

**WHERE employee\_id = 113;**

# Change ALL rows – with no WHERE clause

VERY DANGEROUS

**UPDATE employees**

**SET department\_id = 110;**

# Update 2 items – separate by a comma

**UPDATE employees**

**SET job\_id = 'IT\_PROG', 🡸 one change**

**commission\_pct = NULL 🡸 two changes**

**WHERE employee\_id = 114;**

Updating Two Columns with a Subquery

# Update employee 113 with same data from another employee

# A) JOB ID SAME AS EMPLOYEE 205

# B) SALARY SAME AS 205

Method 1:

**UPDATE employees**

**SET job\_id = (SELECT job\_id**

**FROM employees**

**WHERE employee\_id = 205),**

**salary = (SELECT salary**

**FROM employees**

**WHERE employee\_id = 205)**

**WHERE employee\_id = 113;**

Method 2:

**UPDATE employees**

**SET (job\_id, salary) = (SELECT job\_id, salary**

**FROM employees**

**WHERE employee\_id = 205)**

**WHERE employee\_id = 113;**

Updating - rows based on another table

9-18

# SUBQUERY

**UPDATE employees**

**SET department\_id = (SELECT department\_id**

**FROM employees**

**WHERE employee\_id = 100)**

**WHERE job\_id = (SELECT job\_id**

**FROM employees**

**WHERE employee\_id = 200);**

**What does this do?**

**🡺Changes all employees that have a job\_id the same as employee 200**

**🡺 to have the same department ID as employee 100**

So Far 1 Added data

2 Changed data

Next 3 Remove data

using

# DELETE

# TRUNCATE

DELETE statement

# Remove a specific row or rows

**DELETE FROM departments**

**WHERE department\_name = 'Finance';**

**DELETE FROM employees**

**WHERE employee\_id = 114;**

**DELETE FROM departments**

**WHERE department\_id IN (30, 40);**

**Remove ALL rows**

**DELETE FROM employees; 🡸 no where condition**

# Remove rows based on another table

**DELETE FROM employees**

**WHERE department\_id =**

**(SELECT department\_id**

**FROM departments**

**WHERE department\_name LIKE '%Public%');**

How Processed:

Look in the department table for a department name that contains the string Public.

Use that rows department number and return it to the outer level statement

Delete rows in the employees table that has the same department number

TRUNCATE statement

# Removes ALL rows from a table, but leaves the table structure

**TRUNCATE employees;**

**WHY USED?**

**More efficient than DELETE**

**Delete checks all delete triggers**

**Truncate is a DDL statement and does not create a copy to allow for ROLLBACK**

**If the table is the parent you need to drop constraint of the FK to be able to do this**

Review so far

INSERT - naming the columns

- using NULL as a placeholder if not naming the columns

UPDATE – SET

DELETE

TRUNCATE

COMMIT, ROOLBACK, SAVEPOINT

# Database transaction control

**Transaction example:**

**To pay tuition**

**1) Accept payment and reduce amount owing by student**

**2) Increase amount collected that day by Seneca, to keep Seneca's book in order**

**A transaction is a series of statements forming an event**

**If any part of the series fails, need to undo all statements**

**Database Transactions: Start and End**

• Begin when the first DML SQL statement is executed.

- a statement that makes changes to the database

- INSERT, UPDATE, DELETE

• End with one of the following events:

– A COMMIT or ROLLBACK statement is issued.

– A DDL or DCL statement that executes with an automatic commit

or

– The user exits SQL Developer or SQL\*Plus.

– The system crashes.

**COMMIT and ROLLBACK Statements**

With COMMIT and ROLLBACK statements, you can:

• Ensure data consistency

• Preview data changes before making changes permanent

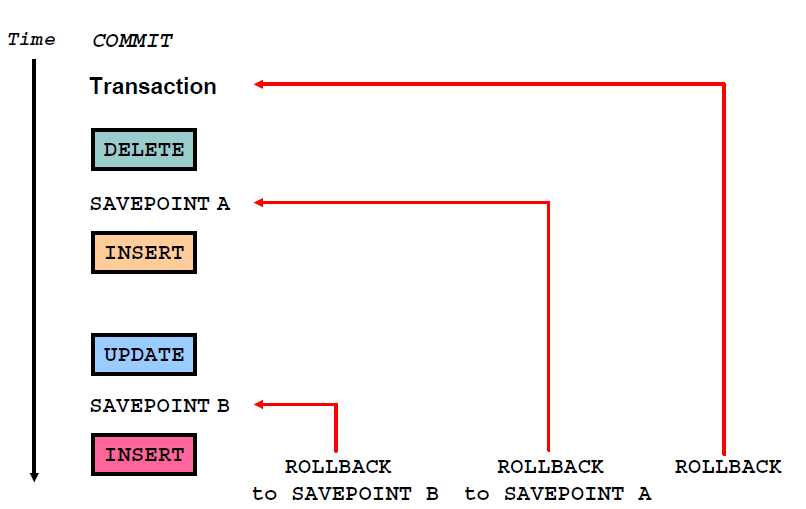
• Group logically-related operations

i.e. Gives you control over making changes to the data permanent

EXAMPLE SCENARIO:

Taking a customer order involves updating the inventory. If customer cancels the order part way through, there needs to be a way to cancel all the updates that were made and put the inventory back to its proper state.

How it works



Issue a delete statement

ETC …

Start here

|  |  |
| --- | --- |
| COMMIT | Ends last transaction making the pending data changes before the commit to be permanent |
| SAVEPOINT name | SAVEPOINT *name* marks a savepoint within the current transaction |
| ROLLBACK | ROLLBACK ends the current transaction by discarding all pending data changes back to the last commit or the start of activity |
| ROLLBACK TO  *SAVEPOINT name* | ROLLBACK TO SAVEPOINT rolls back the current transaction to the specified savepoint, thereby discarding any changes and/or savepoints that were created after the savepoint to which you are rolling back. If you omit the TO  SAVEPOINT clause, the ROLLBACK statement rolls back the entire transaction. Because savepoints are logical, there is no way to list the savepoints that you have created |

Example:

**UPDATE...**

**SAVEPOINT update\_done; 🡸 receive a message SAVEPOINT update\_done succeeded**

**INSERT...**

**ROLLBACK TO update\_done; 🡸 receive a message ROLLBACK succeeded**

(Note if create a savepoint with same name the previous one is deleted)

SYSTEM FAILURES

When a transaction is interrupted by a system failure, the entire transaction is automatically rolled back. This prevents the error from causing unwanted changes to the data and returns the tables to the state at the time of the last commit. In this way, the Oracle server protects the integrity of the tables

State of Data

# 1 Before a COMMIT or ROLLBACK

# - While making data changes (DML)

- The previous state of the data can be recovered

- The current state can be reviewed with the SELECT

- make changes then use a select to see the results

- Other users cannot see the effect of the DML changes

- Each user sees the state the data was in at the last commit

- Affected rows are locked

- Other users cannot change the affected rows

State of Data

# 2 After a COMMIT

- Data changes are saved in the database.

- The previous state of the data is overwritten.

- All users can view the results.

- Locks on the affected rows are released;

Those rows are available for other users to manipulate.

- All savepoints are erased.

Example: COMMITING data

9-35

# Make some changes –

**DELETE FROM employees**

**WHERE employee\_id = 99999;**

**INSERT INTO departments**

**VALUES (290, 'Corporate Tax', NULL, 1700);**

# Commit the changes –

**COMMIT;**

**EXAMPLE 2:**

1 Remove departments 290 and 300 in the DEPARTMENTS table

2 Update a row in the EMPLOYEES table.

3 Save the data change.

1

**DELETE FROM departments**

**WHERE department\_id IN (290, 300);**

**2**

**UPDATE employees**

**SET department\_id = 80**

**WHERE employee\_id = 206;**

**3**

**COMMIT;**

State of data after a ROLLBACK

Discard all pending changes by using the ROLLBACK statement:

• Data changes are undone.

• Previous state of the data is restored.

• Locks on the affected rows are released.

**DELETE FROM employees;**

**ROLLBACK;**

**State of the Data After ROLLBACK: Example**

**DELETE FROM test;**

**25,000 rows deleted. 🡸**

**ROLLBACK;**

**Rollback complete. 🡸**

**DELETE FROM test WHERE id = 100;**

**1 row deleted. 🡸**

**SELECT \* FROM test WHERE id = 100;**

**No rows selected. 🡸**

**COMMIT;**

**Commit complete. 🡸**

**Statement-Level Rollback**

If a single DML statement fails during execution, only that statement is rolled back.

• The Oracle server implements an implicit savepoint.

• All other changes are retained and can be committed or rolled back

**Read Consistency**

• Read consistency guarantees a consistent view of the data at all times.

• Changes made by one user do not conflict with the changes made by another user.

• Read consistency ensures that, on the same data:

– Readers (SELECT) do not wait for writers

– Writers (INSERT, UPDATE, and DELETE) do not wait for readers

– Writers wait for writers

**FOR UPDATE Clause in a SELECT Statement**

• Locks the rows in the EMPLOYEES table where job\_id is SA\_REP.

**SELECT employee\_id, salary, commission\_pct, job\_id**

**FROM employees**

**WHERE job\_id = 'SA\_REP'**

**FOR UPDATE**

**ORDER BY employee\_id;**

• Lock is released only when you issue a ROLLBACK or a COMMIT.

• If the SELECT statement attempts to lock a row that is locked by another user, the database waits until the row is available, and then returns the results of the SELECT statement.

You can issue a NO WAIT as well so that it does not wait, but returns control to the user who can do other work. Without NO WAIT you could be waiting until the other users issues a ROLLBACK or COMMIT

We are talking about an active business operation and not about IT people programming

EXAMPLES – FOR UPDATE

Using on multiple tables

**SELECT e.employee\_id, e.salary, e.commission\_pct**

**FROM employees e JOIN departments d**

**USING (department\_id)**

**WHERE job\_id = 'ST\_CLERK’ AND location\_id = 1500**

**FOR UPDATE**

**ORDER BY e.employee\_id;**

- Rows in both tables are locked

-

You can further qualify the UPDATE with a column name to restrict locking to only the table with that column in it

Example:

The following statement locks only those rows in the EMPLOYEES table with ST\_CLERK located in

LOCATION\_ID 1500. No rows are locked in the DEPARTMENTS table:

**SELECT e.employee\_id, e.salary, e.commission\_pct**

**FROM employees e JOIN departments d**

**USING (department\_id)**

**WHERE job\_id = 'ST\_CLERK' AND location\_id = 1500**

**FOR UPDATE OF e.salary**

**ORDER BY e.employee\_id;**

WAIT EXAMPLE: Waits 5 seconds

**SELECT employee\_id, salary, commission\_pct, job\_id**

**FROM employees**

**WHERE job\_id = 'SA\_REP'**

**FOR UPDATE WAIT 5**

**ORDER BY employee\_id;**