Managing Tables Using DML Statements

ORACLE

Objectives

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

- Describe each data manipulation language (DML) statement
- Control transactions

In this lesson, you learn how to use the data manipulation language (DML) statements to insert rows into a table, update existing rows in a table, and delete existing rows from a table. You also learn how to control transactions with the `COMMIT`, `SAVEPOINT`, and `ROLLBACK` statements.

Lesson Agenda

- 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

Data Manipulation Language

- 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.

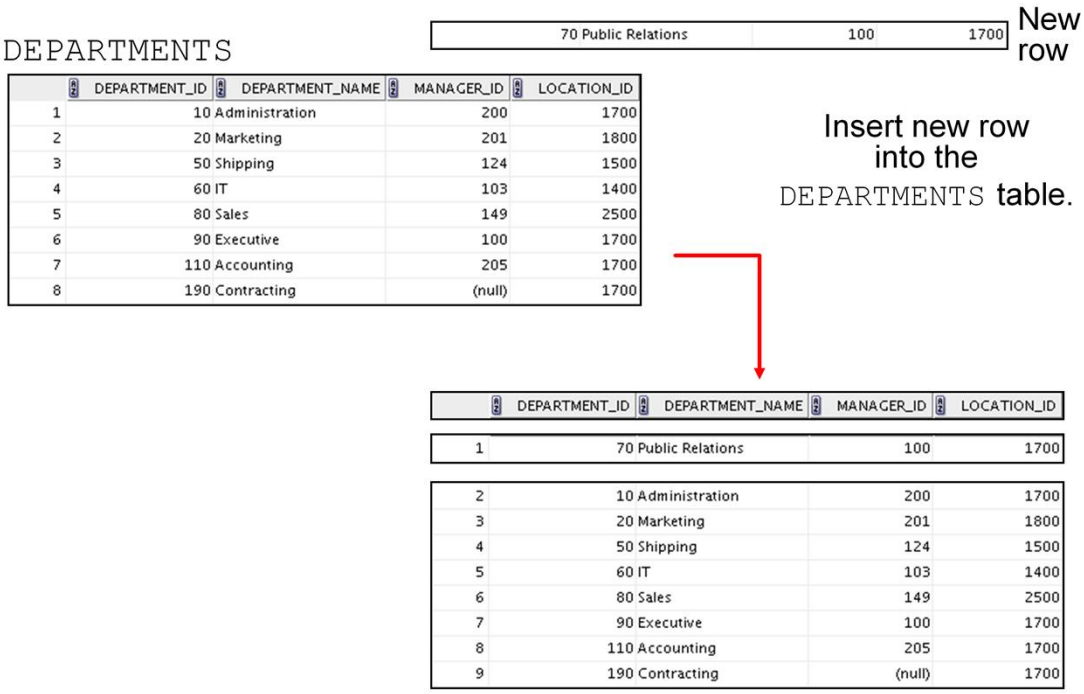
Data manipulation language (DML) is a core part of SQL. When you want to add, update, or delete data in the database, you execute a DML statement. A collection of DML statements that form a logical unit of work is called a *transaction*.

Consider a banking database. When a bank customer transfers money from a savings account to a checking account, the transaction might consist of three separate operations: decreasing the savings account, increasing the checking account, and recording the transaction in the transaction journal. The Oracle server must guarantee that all the three SQL statements are performed to maintain the accounts in proper balance. When something prevents one of the statements in the transaction from executing, the other statements of the transaction must be undone.

Note

- Most of the DML statements in this lesson assume that no constraints on the table are violated. Constraints are discussed later in this course.
- In SQL Developer, click the Run Script icon or press [F5] to run the DML statements. The feedback messages will be shown on the Script Output tabbed page.

Adding a New Row to a Table



The graphic in the slide illustrates the addition of a new department to the DEPARTMENTS table.

INSERT Statement Syntax

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

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

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

You can add new rows to a table by issuing the INSERT statement.

In the syntax:

table	Is the name of the table
column	Is the name of the column in the table to populate
value	Is the corresponding value for the column

Note: This statement with the VALUES clause adds only one row at a time to a table.

Inserting New Rows

- Insert a new row containing values for each column.
- 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);
1 rows inserted
```

- Enclose character and date values within single quotation marks.

Because you can insert a new row that contains values for each column, the column list is not required in the `INSERT` clause. However, if you do not use the column list, the values must be listed according to the default order of the columns in the table, and a value must be provided for each column.

```
DESCRIBE departments
```

```
DESCRIBE  departments
Name            Null     Type
--------------- -------- ------------
DEPARTMENT_ID   NOT NULL NUMBER(4)
DEPARTMENT_NAME NOT NULL VARCHAR2(30)
MANAGER_ID               NUMBER(6)
LOCATION_ID              NUMBER(4)
```

For clarity, use the column list in the `INSERT` clause.
Enclose character and date values within single quotation marks; however, it is not recommended that you enclose numeric values within single quotation marks.

Inserting Rows with Null Values

- Implicit method: Omit the column from the column list.

```
INSERT INTO   departments (department_id,
                           department_name)
VALUES        (30, 'Purchasing');
1 rows inserted
```

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

```
INSERT INTO   departments
VALUES        (100, 'Finance', NULL, NULL);
1 rows inserted
```

ORACLE

Method	Description
Implicit	Omit the column from the column list.
Explicit	Specify the NULL keyword in the VALUES list; specify the empty string ('') in the VALUES list for character strings and dates.

Be sure that you can use null values in the targeted column by verifying the Null status with the DESCRIBE command.

The Oracle server automatically enforces all data types, data ranges, and data integrity constraints. Any column that is not listed explicitly obtains a null value in the new row unless we have default values for the missing columns that are used.

Common errors that can occur during user input are checked in the following order:

- Mandatory value missing for a NOT NULL column
- Duplicate value violating any unique or primary key constraint
- Any value violating a CHECK constraint
- Referential integrity maintained for foreign key constraint
- Data type mismatches or values too wide to fit in column

Note: Use of the column list is recommended because it makes the `INSERT` statement more readable and reliable, or less prone to mistakes.

Inserting Special Values

The SYSDATE function records the current date and time.

```
INSERT INTO employees (employee_id,
                       first_name, last_name,
                       email, phone_number,
                       hire_date, job_id, salary,
                       commission_pct, manager_id,
                       department_id)
VALUES                 (113,
                       'Louis', 'Popp',
                       'LPOPP', '515.124.4567',
                       SYSDATE, 'AC_ACCOUNT', 6900,
                       NULL, 205, 110);
1 rows inserted
```


You can use functions to enter special values in your table.

The slide example records information for employee Popp in the EMPLOYEES table. It supplies the current date and time in the HIRE_DATE column. It uses the SYSDATE function that returns the current date and time of the database server. You may also use the CURRENT_DATE function to get the current date in the session time zone. You can also use the USER function when inserting rows in a table. The USER function records the current username.

Confirming Additions to the Table

```
SELECT employee_id, last_name, job_id, hire_date, commission_pct
FROM   employees
WHERE  employee_id = 113;
```

	EMPLOYEE_ID	LAST_NAME	JOB_ID	HIRE_DATE	COMMISSION_PCT
1	113	Popp	AC_ACCOUNT	24-AUG-12	(null)

Note: The hire date may vary from the screenshot and it will fetch data as per the data insert date.

Inserting Specific Date and Time Values

- Add a new employee.

```
INSERT INTO employees
VALUES      (114,
             'Den', 'Raphealy',
             'DRAPHEAL', '515.127.4561',
             TO_DATE('FEB 3, 2003', 'MON DD, YYYY'),
             'SA_REP', 11000, 0.2, 100, 60);
1 rows inserted
```

- Verify your addition.

	EMPLOYEE_ID	FIRST_NAME	LAST_NAME	EMAIL	PHONE_NUMBER	HIRE_DATE	JOB_ID	SALARY	COMMISSION_PCT	MANAGER_ID
1	114	Den	Raphealy	DRAPHEAL	515.127.4561	03-FEB-03	SA_REP	11000	0.2	100

The DD-MON-RR format is generally used to insert a date value. With the RR format, the system provides the correct century automatically.

You may also supply the date value in the DD-MON-YYYY format. This is recommended because it clearly specifies the century and does not depend on the internal RR format logic of specifying the correct century.

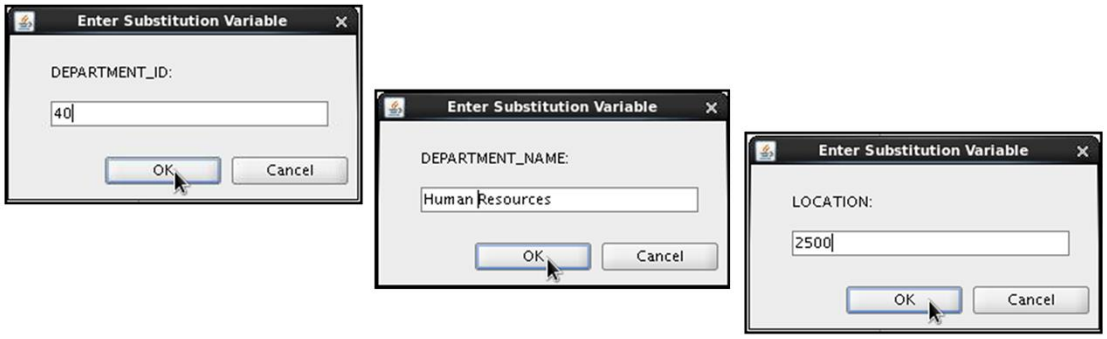
If a date must be entered in a format other than the default format (for example, with another century or a specific time), you must use the TO_DATE function.

The example in the slide records information for employee Raphealy in the EMPLOYEES table. It sets the HIRE_DATE column to be February 3, 2003.

Creating a Script

- Use the & substitution in a SQL statement to prompt for values.
- & is a placeholder for the variable value.

```
INSERT INTO departments
            (department_id, department_name, location_id)
VALUES      (&department_id, '&department_name',&location);
```



You can save commands with substitution variables to a file and execute the commands in the file. The example in the slide records information for a department in the DEPARTMENTS table.

Run the script file and you are prompted for input for each of the ampersand (&) substitution variables. After entering a value for the substitution variable, click the OK button. The values that you input are then substituted into the statement. This enables you to run the same script file over and over, but supply a different set of values each time you run it.

Copying Rows from Another Table

- Write your INSERT statement with 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%';

5 rows inserted.
```

- Do not use the VALUES clause.
- Match the number of columns in the INSERT clause to those in the subquery.
- Inserts all the rows returned by the subquery in the table, sales_reps.

ORACLE

You can use the INSERT statement to add rows to a table where the values are derived from existing tables. In the example in the slide, for the INSERT INTO statement to work, you must have already created the sales_reps table using the CREATE TABLE statement. CREATE TABLE is discussed in the lesson titled " Introduction to "Introduction to Data Definition Language."

In place of the VALUES clause, you use a subquery.

Syntax

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

In the syntax:

table	Is the name of the table
column	Is the name of the column in the table to populate
subquery	Is the subquery that returns rows to the table

The number of columns and their data types in the column list of the INSERT clause must match the number of values and their data types in the subquery. Zero or more rows are added depending on the number of rows returned by the subquery. To create a copy of the rows of a table, use SELECT * in the subquery:

```
INSERT INTO copy_emp
  SELECT *
  FROM   employees;
```

Changing Data in a Table

EMPLOYEES

EMPLOYEE_ID	FIRST_NAME	LAST_NAME	SALARY	MANAGER_ID	COMMISSION_PCT	DEPARTMENT_ID
100	Steven	King	24000	(null)	(null)	90
101	Neena	Kochhar	17000	100	(null)	90
102	Lex	De Haan	17000	100	(null)	90
103	Alexander	Hunold	9000	102	(null)	60
104	Bruce	Ernst	6000	103	(null)	60
107	Diana	Lorentz	4200	103	(null)	60
124	Kevin	Mourgos	5800	100	(null)	50

Update rows in the EMPLOYEES table:

EMPLOYEE_ID	FIRST_NAME	LAST_NAME	SALARY	MANAGER_ID	COMMISSION_PCT	DEPARTMENT_ID
100	Steven	King	24000	(null)	(null)	90
101	Neena	Kochhar	17000	100	(null)	90
102	Lex	De Haan	17000	100	(null)	90
103	Alexander	Hunold	9000	102	(null)	80
104	Bruce	Ernst	6000	103	(null)	80
107	Diana	Lorentz	4200	103	(null)	80
124	Kevin	Mourgos	5800	100	(null)	50

The slide illustrates changing the department number for employees in department 60 to department 80.

UPDATE Statement Syntax

- Modify existing values in a table with the UPDATE statement:

```
UPDATE      table
SET         column = value [, column = value, ...]
[WHERE      condition];
```

- Update more than one row at a time (if required).

You can modify the existing values in a table by using the UPDATE statement.

In the syntax:

table	Is the name of the table
column	Is the name of the column in the table to populate
value	Is the corresponding value or subquery for the column
condition	Identifies the rows to be updated and is composed of column names, expressions, constants, subqueries, and comparison operators

Confirm the update operation by querying the table to display the updated rows.

For more information, see the section on "UPDATE" in *Oracle Database SQL Language Reference* for 19c database.

Note: In general, use the primary key column in the WHERE clause to identify a single row for update. Using other columns can unexpectedly cause several rows to be updated. For example, identifying a single row in the EMPLOYEES table by name is dangerous, because more than one employee may have the same name.

Updating Rows in a Table

- Values for a specific row or rows are modified if you specify the `WHERE` clause:

```
UPDATE employees
SET    department_id = 50
WHERE  employee_id = 113;
1 rows updated
```

- Values for all the rows in the table are modified if you omit the `WHERE` clause:

```
UPDATE   copy_emp
SET      department_id = 110;
22 rows updated
```

- Specify `SET` *`column_name`*`= NULL` to update a column value to `NULL`.

ORACLE

The `UPDATE` statement modifies the values of a specific row or rows if the `WHERE` clause is specified. The example in the slide shows the transfer of employee 113(Popp) to department 50.

If you omit the `WHERE` clause, values for all the rows in the table are modified. Examine the updated rows in the `COPY_EMP` table.

```
SELECT last_name, department_id
FROM   copy_emp;
```

	LAST_NAME	DEPARTMENT_ID
1	King	110
2	Kochhar	110
3	De Haan	110

. . .

For example, an employee who was an `SA_REP` has now changed his job to an `IT_PROG`. Therefore, his `JOB_ID` needs to be updated and the commission field needs to be set to `NULL`.

```
UPDATE employees
SET job_id = 'IT_PROG', commission_pct = NULL
WHERE employee_id = 114;
```

Note: The `COPY_EMP` table has the same data as the `EMPLOYEES` table.

Updating Two Columns with a Subquery

Update employee 113's job and salary to match those of employee 205.

```
UPDATE    employees
SET       (job_id,salary)  = (SELECT  job_id,salary
                      FROM    employees
                      WHERE   employee id = 205)
          WHERE    employee_id    =  103;

1 rows updated
```


You can update multiple columns in the `SET` clause of an `UPDATE` statement by writing multiple subqueries. The syntax is as follows:

```
UPDATE table
SET     column  =
                    (SELECT  column
                     FROM table
                     WHERE condition)
       [ ,
        column  =
                    (SELECT  column
                     FROM table
                     WHERE condition)]
[WHERE  condition ]    ;
```

Updating Rows Based on Another Table

Use the subqueries in the `UPDATE` statements to update row values in a table based on values from another table:

```
UPDATE  copy_emp
SET     department_id  =  (SELECT department_id
                          FROM employees
                          WHERE employee_id = 100)
WHERE   job_id         =  (SELECT job_id
                          FROM employees
                          WHERE employee_id = 200);
1 rows updated
```


You can use the subqueries in the `UPDATE` statements to update values in a table. The example in the slide updates the `COPY_EMP` table based on the values from the `EMPLOYEES` table. It changes the department number of all employees with employee 200's job `ID` to employee 100's current department number.

Removing a Row from a Table

DEPARTMENTS

	DEPARTMENT_ID	DEPARTMENT_NAME	MANAGER_ID	LOCATION_ID
1	10	Administration	200	1700
2	20	Marketing	201	1800
3	50	Shipping	124	1500
4	60	IT	103	1400
5	80	Sales	149	2500
6	90	Executive	100	1700
7	110	Accounting	205	1700
8	190	Contracting	(null)	1700

Delete a row from the DEPARTMENTS table:

	DEPARTMENT_ID	DEPARTMENT_NAME	MANAGER_ID	LOCATION_ID
1	10	Administration	200	1700
2	20	Marketing	201	1800
3	50	Shipping	124	1500
4	60	IT	103	1400
5	80	Sales	149	2500
6	90	Executive	100	1700
7	110	Accounting	205	1700

The Contracting department has been removed from the DEPARTMENTS table (assuming no constraints on the DEPARTMENTS table are violated), as shown by the graphic in the slide.

DELETE Statement

You can remove existing rows from a table by using the DELETE statement:

```
DELETE [FROM]   table
[WHERE          condition];
```


You can remove existing rows from a table by using the DELETE statement.

In the syntax:

table	Is the name of the table
condition	Identifies the rows to be deleted, and is composed of column names, expressions, constants, subqueries, and comparison operators

Note: If no rows are deleted, the message "0 rows deleted" is returned (on the Script Output tab in SQL Developer).

For more information, see the section on "DELETE" in *Oracle Database SQL Language Reference* for 19c database.

Deleting Rows from a Table

- Specific rows are deleted if you specify the `WHERE` clause:

```
DELETE FROM departments
WHERE  department_name = 'Finance';
1 rows deleted
```

- All rows in the table are deleted if you omit the `WHERE` clause:

```
DELETE FROM  copy_emp;
22 rows deleted
```

ORACLE

You can delete specific rows by specifying the `WHERE` clause in the `DELETE` statement. The first example in the slide deletes the Accounting department from the `DEPARTMENTS` table. You can confirm the delete operation by displaying the deleted rows using the `SELECT` statement.

```
SELECT  *
FROM    departments
WHERE   department_name = 'Finance';
```

```
no rows selected
```

However, if you omit the `WHERE` clause, all rows in the table are deleted. The second example in the slide deletes all rows from the `COPY_EMP` table, because no `WHERE` clause was specified.

Example

Remove rows identified in the `WHERE` clause.

```
DELETE FROM  employees WHERE employee_id = 114;
```

```
1 rows deleted
```

```
DELETE FROM  departments WHERE department_id IN (30, 40);
```

```
2 rows deleted
```

Deleting Rows Based on Another Table

Use the subqueries in the `DELETE` statements to remove rows from a table based on values from another table:

```
DELETE FROM employees
WHERE  department_id IN
                (SELECT department_id
                 FROM   departments
                 WHERE  department_name
                        LIKE '%Public%');
1 rows deleted
```


You can use the subqueries to delete rows from a table based on values from another table. The example in the slide deletes all the employees in a department, where the department name contains the string `Public`.

The subquery searches the `DEPARTMENTS` table to find the department number based on the department name containing the string `Public`. The subquery then feeds the department number to the main query, which deletes rows of data from the `EMPLOYEES` table based on this department number.

TRUNCATE Statement

- Removes all rows from a table, leaving the table empty and the table structure intact
- Is a data definition language (DDL) statement rather than a DML statement; cannot easily be undone
- Syntax:

```
TRUNCATE TABLE table_name;
```

- Example:

```
TRUNCATE TABLE copy_emp;
```

ORACLE

A more efficient method of emptying a table is by using the TRUNCATE statement.
You can use the TRUNCATE statement to quickly remove all rows from a table or cluster. Removing rows with the TRUNCATE statement is faster than removing them with the DELETE statement for the following reasons:

- The TRUNCATE statement is a data definition language (DDL) statement and generates no rollback information. Rollback information is covered later in this lesson.
- Truncating a table does not fire the delete triggers of the table.

If the table is the parent of a referential integrity constraint, you cannot truncate the table. You need to disable the constraint before issuing the TRUNCATE statement. Disabling constraints is covered in the lesson titled “Introduction to DDL Statements.”

Database Transactions

- The Oracle server ensures data consistency based on transactions. Transactions give you more flexibility and control when changing data, and they ensure data consistency in the event of user process failure or system failure.
- Transactions consist of DML statements that constitute one consistent change to the data.
- For example, a transfer of funds between two accounts should include the debit in one account and the credit to another account of the same amount. Both actions should either fail or succeed together; the credit should not be committed without the debit.

ORACLE

Transaction Types

Database Transactions: Start and End

- Begin when the first DML SQL statement is executed.
- End with one of the following events:
  - A `COMMIT` or `ROLLBACK` statement is issued.
  - A DDL or DCL statement executes (automatic commit).
  - The user exits SQL Developer or SQL*Plus.
  - The system crashes.

Type	Description
Data manipulation language (DML)	Consists of any number of DML statements that the Oracle server treats as a single entity or a logical unit of work
Data definition language (DDL)	Consists of only one DDL statement
Data control language (DCL)	Consists of only one DCL statement

ORACLE

When does a database transaction start and end?

A transaction begins when the first DML statement is encountered and ends when one of the following occurs:

- A `COMMIT` or `ROLLBACK` statement is issued.
- A DDL statement, such as `CREATE`, is issued.
- A DCL statement is issued.
- The user exits SQL Developer or SQL*Plus.
- A machine fails or the system crashes.

After one transaction ends, the next executable SQL statement automatically starts the next transaction.

A DDL statement or a DCL statement is automatically committed and, therefore, implicitly ends a transaction.

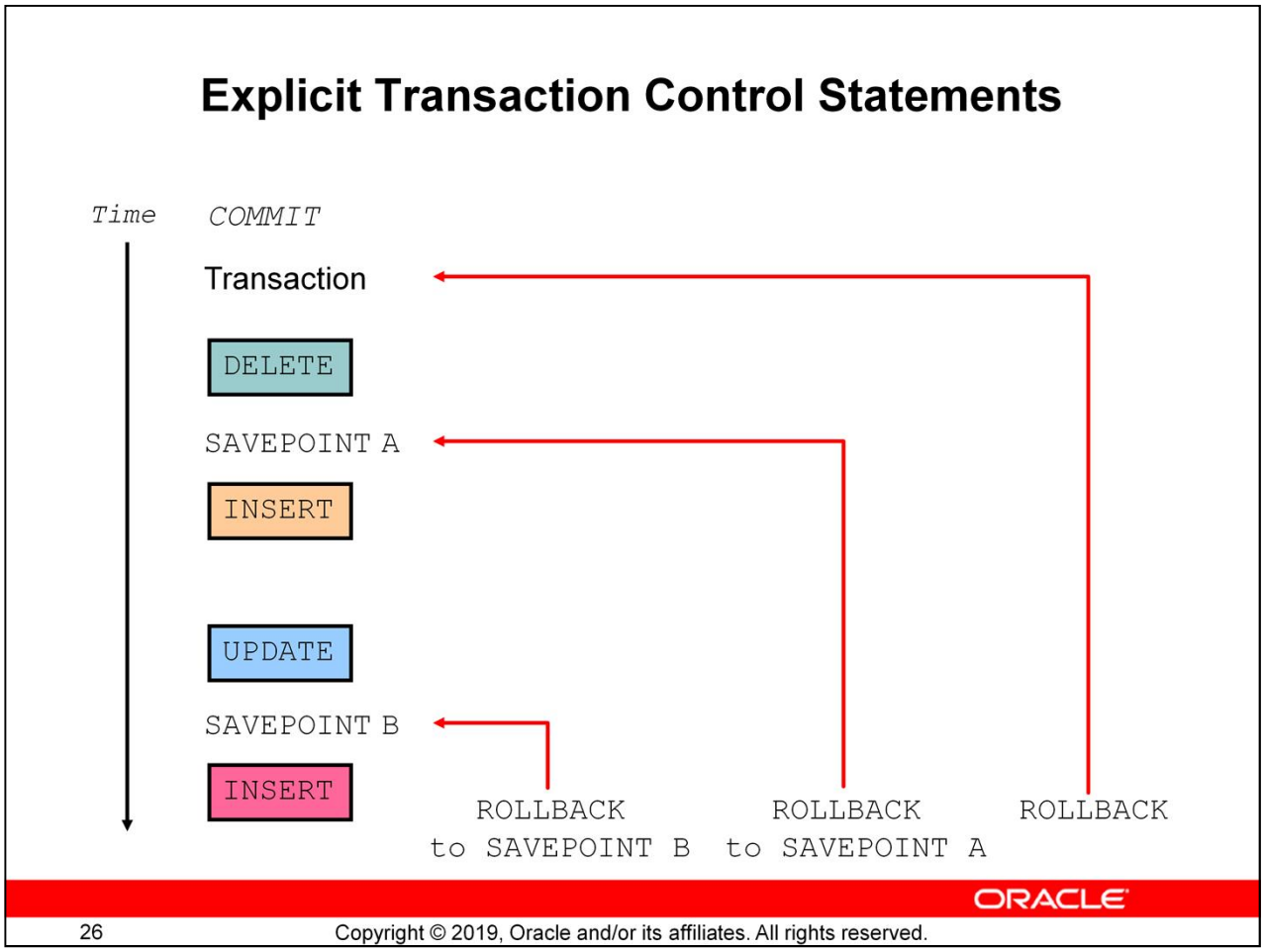
Advantages of COMMIT and ROLLBACK Statements

With COMMIT and ROLLBACK statements, you can:

- Ensure data consistency
- Preview data changes before making changes permanent
- Group logically related operations

- With the COMMIT and ROLLBACK statements, you have control over making changes to the data permanent.

- You can control the logic of transactions by using the COMMIT, SAVEPOINT, and ROLLBACK statements.



You can control the logic of transactions by using the `COMMIT`, `SAVEPOINT`, and `ROLLBACK` statements.

Statement	Description
`COMMIT`	`COMMIT` ends the current transaction by making all pending data changes permanent.
`SAVEPOINT` *name*	`SAVEPOINT` *name* marks a savepoint within the current transaction.
`ROLLBACK`	`ROLLBACK` ends the current transaction by discarding all pending data changes.
`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.

Note: You cannot `COMMIT` to a `SAVEPOINT`. `SAVEPOINT` is not ANSI-standard SQL.

Rolling Back Changes to a Marker

- Create a marker in the current transaction by using the `SAVEPOINT` statement.
- Roll back to that marker by using the `ROLLBACK TO SAVEPOINT` statement.

```
UPDATE...
SAVEPOINT update_done;
SAVEPOINT update_done succeeded.
INSERT...
ROLLBACK TO update_done;
ROLLBACK TO succeeded.
```

- You can create a marker in the current transaction by using the `SAVEPOINT` statement, which divides the transaction into smaller sections. You can then discard pending changes up to that marker by using the `ROLLBACK TO SAVEPOINT` statement.

Implicit Transaction Processing

- An automatic commit occurs in the following circumstances:
  - A DDL statement issued
  - A DCL statement issued
  - Normal exit from SQL Developer or SQL*Plus, without explicitly issuing `COMMIT` or `ROLLBACK` statements
- An automatic rollback occurs when there is an abnormal termination of SQL Developer or SQL*Plus or a system failure.

ORACLE

Status	Circumstances
Automatic commit	DDL statement or DCL statement issued
	SQL Developer or SQL*Plus exited normally, without explicitly issuing `COMMIT` or `ROLLBACK` commands
Automatic rollback	Abnormal termination of SQL Developer or SQL*Plus or system failure

Note: In SQL*Plus, the `AUTOCOMMIT` command can be toggled `ON` or `OFF`. If set to `ON`, each individual `DML` statement is committed as soon as it is executed. You cannot roll back the changes. If set to `OFF`, the `COMMIT` statement can still be issued explicitly. Also, the `COMMIT` statement is issued when a DDL statement is issued or when you exit SQL*Plus. The `SET AUTOCOMMIT ON/OFF` command is skipped in SQL Developer. DML is committed on a normal exit from SQL Developer only if you have the Autocommit preference enabled. To enable Autocommit, perform the following:

- In the Tools menu, select Preferences. In the Preferences dialog box, expand Database and select Worksheet Parameters.
- In the right pane, select the “Autocommit in SQL Worksheet” option. Click OK

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.

In SQL Developer, a normal exit from the session is accomplished by selecting Exit from the File menu. In SQL*Plus, a normal exit is accomplished by entering the `EXIT` command at the

prompt. Closing the window is interpreted as an abnormal exit.

State of the Data Before COMMIT or ROLLBACK

- Every data change made during the transaction is temporary until the transaction is committed.
  - The previous state of the data can be recovered.
  - The current session can review the results of the DML operations by using the SELECT statement.
  - Other sessions *cannot* view the results of the DML statements issued by the current session.
  - The affected rows are *locked*; other session cannot change the data in the affected rows.

ORACLE

Every data change made during the transaction is temporary until the transaction is committed.

The state of the data before COMMIT or ROLLBACK statements are issued can be described as follows:

- Data manipulation operations primarily affect the database buffer; therefore, the previous state of the data can be recovered.
- The current session can review the results of the data manipulation operations by querying the tables.
- Other sessions cannot view the results of the data manipulation operations made by the current session. The Oracle server institutes read consistency to ensure that each session sees data as it existed at the last commit.
- The affected rows are locked; other session cannot change the data in the affected rows.

State of the Data After COMMIT

- Make all pending changes permanent by using the COMMIT statement. Here is what happens after a COMMIT statement:
  - Data changes are saved in the database.
  - The previous state of the data is overwritten.
  - All sessions can view the results.
  - Locks on the affected rows are released; those rows are available for other sessions to manipulate.
  - All savepoints are erased.

Make all pending changes permanent by using the COMMIT statement. Here is what happens after a COMMIT statement:

- Data changes are written to the database.
- The previous state of the data is no longer available with normal SQL queries.
- All sessions can view the results of the transaction.
- The locks on the affected rows are released; the rows are now available for other sessions to perform new data changes.
- All savepoints are erased.

Committing Data

- Make the changes:

```
DELETE FROM EMPLOYEES
WHERE employee_id=113;
1 rows deleted
INSERT INTO departments
VALUES (290, 'Corporate Tax', NULL, 1700);
1 rows inserted
```

- Commit the changes:

```
COMMIT;
commited.
```

ORACLE

In the example in the slide, a row is deleted from the EMPLOYEES table and a new row is inserted into the DEPARTMENTS table. The changes are saved by issuing the COMMIT statement.

Example

Remove departments 290 and 300 in the DEPARTMENTS table and update a row in the EMPLOYEES table. Save the data change.

```
DELETE FROM departments
WHERE  department_id IN (290, 300);

UPDATE  employees
  SET   department_id = 80
  WHERE employee_id = 206;

COMMIT;
```

State of the Data After ROLLBACK

Discard all pending changes by using the ROLLBACK statement, which results in the following:

- Data changes are undone.
- Previous state of the data is restored.
- Locks on the affected rows are released.

```
DELETE FROM copy_emp;
ROLLBACK ;
```


Discard all pending changes by using the ROLLBACK statement, which results in the following:

- Data changes are undone.
- The previous state of the data is restored.
- Locks on the affected rows are released.

State of the Data After ROLLBACK: Example

- While attempting to remove a record from the TEST table, you may accidentally empty the table.
- However, you can correct the mistake, reissue a proper statement, and make the data change permanent.

```
DELETE FROM test;
4 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.
```

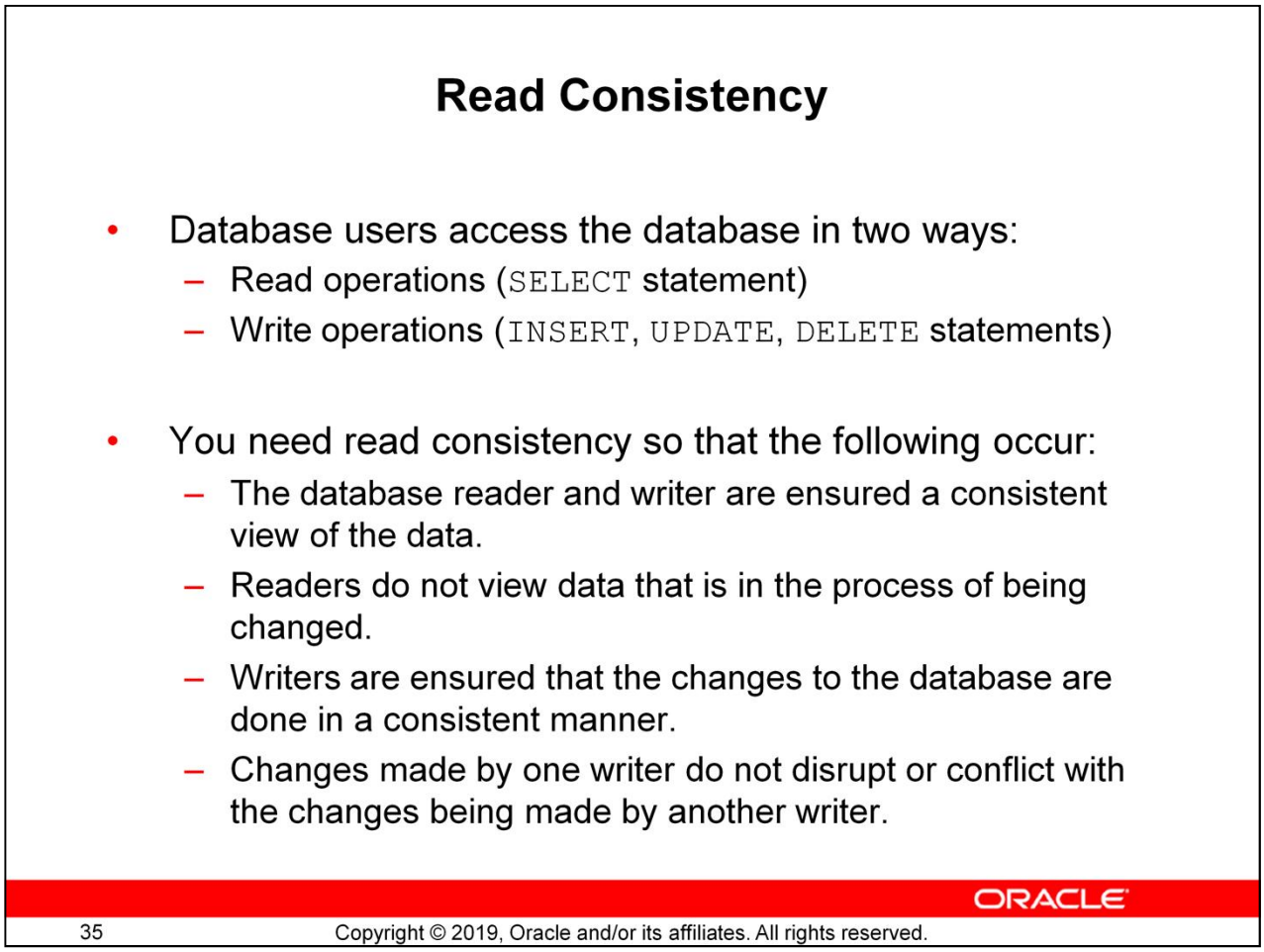
Statement-Level Rollback

- 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.
- The user should terminate transactions explicitly by executing a `COMMIT` or `ROLLBACK` statement.

A part of a transaction can be discarded through an implicit rollback if a statement execution error is detected. If a single DML statement fails during execution of a transaction, its effect is undone by a statement-level rollback, but the changes made by the previous DML statements in the transaction are not discarded. They can be committed or rolled back explicitly by the user.

The Oracle server issues an implicit commit before and after any DDL statement. So, even if your DDL statement does not execute successfully, you cannot roll back the previous statement because the server issued a commit.

Terminate your transactions explicitly by executing a `COMMIT` or `ROLLBACK` statement.



Database users access the database in two ways:

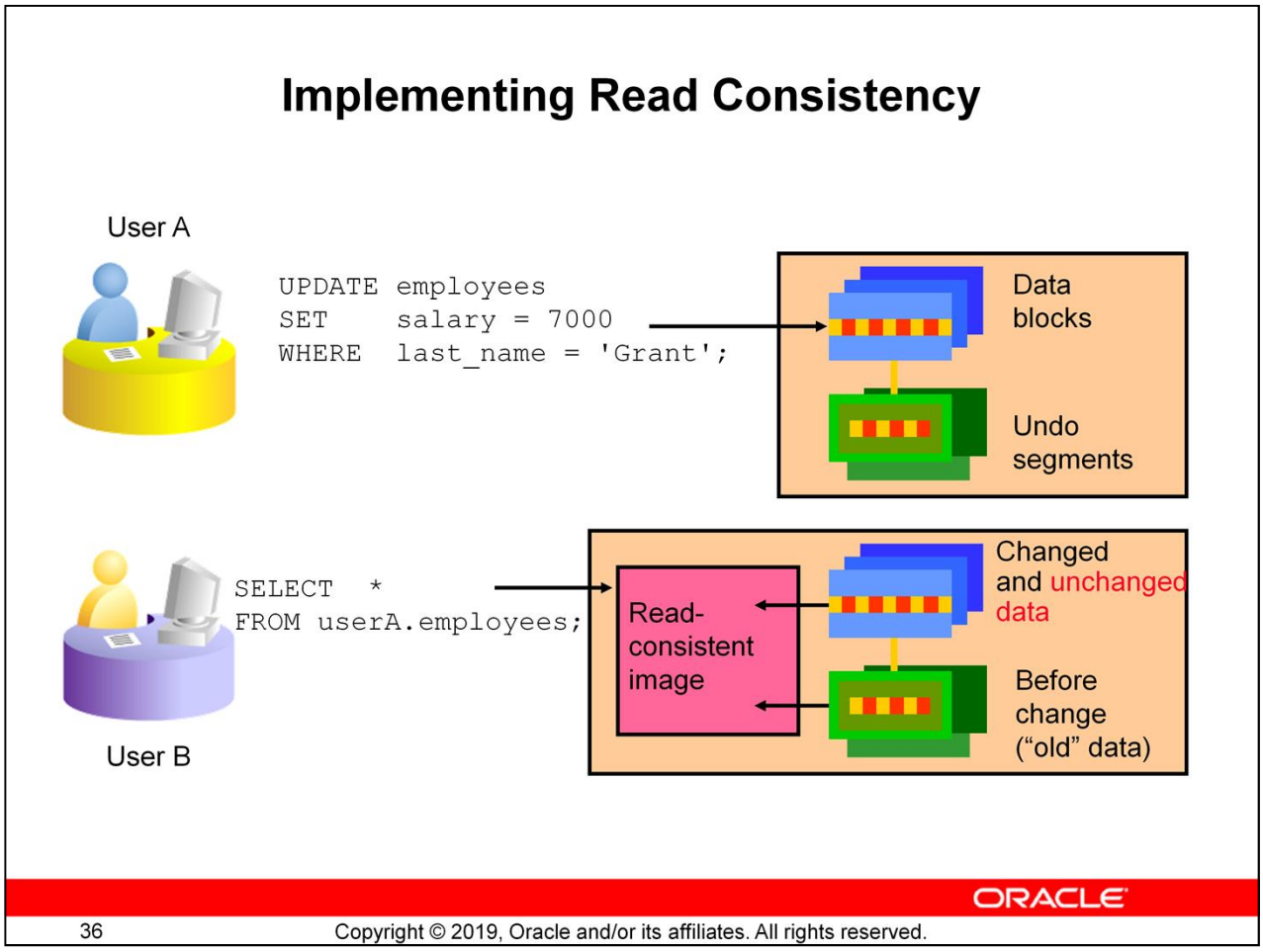
- Read operations (`SELECT` statement)
- Write operations (`INSERT`, `UPDATE`, `DELETE` statements)

You need read consistency so that the following occur:

- The database reader and writer are ensured a consistent view of the data.
- Readers do not view data that is in the process of being changed.
- Writers are ensured that the changes to the database are done in a consistent manner.
- Changes made by one writer do not disrupt or conflict with the changes being made by another writer.

The purpose of read consistency is to ensure that each user sees data as it existed at the last commit, before a DML operation started.

Note: The same user can log in to different sessions. Each session maintains read consistency in the manner described above, even if they are the same users.



Read consistency is an automatic implementation. It keeps a partial copy of the database in the undo segments. The read-consistent image is constructed from the committed data in the table and the old data that is being changed and is not yet committed from the undo segment.

When an insert, update, or delete operation is made on the database, the Oracle server takes a copy of the data before it is changed and writes it to an *undo segment*.

All readers, except the one who issued the change, see the database as it existed before the changes started; they view the undo segment's "snapshot" of the data.

Before the changes are committed to the database, only the user who is modifying the data sees the database with the alterations. Everyone else sees the snapshot in the undo segment. This guarantees that readers of the data read consistent data that is not currently undergoing change.

When a DML statement is committed, the change made to the database becomes visible to anyone issuing a `SELECT` statement *after* the commit is done. The space occupied by the *old* data in the undo segment file is freed for reuse.

If the transaction is rolled back, the changes are undone:

- The original, older version of the data in the undo segment is written back to the table.

- All users see the database as it existed before the transaction began.

Quiz

The following statements produce the same results:

```
DELETE FROM copy_emp;
```

```
TRUNCATE TABLE copy_emp;
```

a. True
b. False

Answer: b

Summary

In this lesson, you should have learned how to use the following statements:

Function	Description
INSERT	Adds a new row to the table
UPDATE	Modifies existing rows in the table
DELETE	Removes existing rows from the table
TRUNCATE	Removes all rows from a table
COMMIT	Makes all pending changes permanent
SAVEPOINT	Is used to roll back to the savepoint marker
ROLLBACK	Discards all pending data changes

In this lesson, you should have learned how to manipulate data in the Oracle database by using the `INSERT`, `UPDATE`, `DELETE`, and `TRUNCATE` statements, as well as how to control data changes by using the `COMMIT`, `SAVEPOINT`, and `ROLLBACK` statements. You also learned how to use the `FOR UPDATE` clause of the `SELECT` statement to lock rows for your changes only.

Remember that the Oracle server guarantees a consistent view of data at all times.

Flashback Version Query

- Flashback version query allows the versions of a specific row to be tracked during a specified time period using the VERSIONS BETWEEN clause.

```
CREATE TABLE arif (sno NUMBER(10),sname  VARCHAR2(50));
INSERT INTO arif VALUES(1, 'ahmed');
COMMIT;
SELECT * FROM arif;
SELECT TO_CHAR(SYSDATE, 'dd-mon-yy hh24:mi:ss') FROM
dual;
27-oct-21 08:21:15
```

```
update arif set sname ='arif' where sno=1;
commit;
select to_char(sysdate, 'dd-mon-yy hh24:mi:ss') from dual;
27-oct-21 08:22:29

update arif set sname ='salim' where sno=1;
commit;
select to_char(sysdate, 'dd-mon-yy hh24:mi:ss') from dual;
27-oct-21 08:24:43

Select * from arif;
1       salim
```

Make a note of all the three timings……

Available Pseudocolumns

- VERSIONS_STARTSCN or VERSIONS_STARTTIME
  - Starting SCN and TIMESTAMP when row took on this value. The value of NULL is returned if the row was created before the lower bound SCN or TIMESTAMP.
- VERSIONS_ENDSCN or VERSIONS_ENDTIME
  - Ending SCN and TIMESTAMP when row last contained this value. The value of NULL is returned if the value of the row is still current at the upper bound SCN or TIMESTAMP.
- VERSIONS_XID
  - ID of the transaction that created the row in it's current state.
- VERSIONS_OPERATION
  - Operation performed by the transaction ((I)nsert, (U)pdate or (D)elete)

```
SELECT versions_startscn, versions_starttime,
versions_endscn, versions_endtime,
      versions_xid, versions_operation, sno, sname from arif
      versions between TIMESTAMP TO_TIMESTAMP('27-oct-
21 08:21:15', 'dd-mon-yy hh24:mi:ss')
                  AND TO_TIMESTAMP('27-oct-21 08:24:43', 'dd-
mon-yy hh24:mi:ss')
```