

Creating Compound, DDL, and Event Database Triggers

ORACLE™

Copyright © 2019, Oracle and/or its affiliates. All rights reserved.

Objectives

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

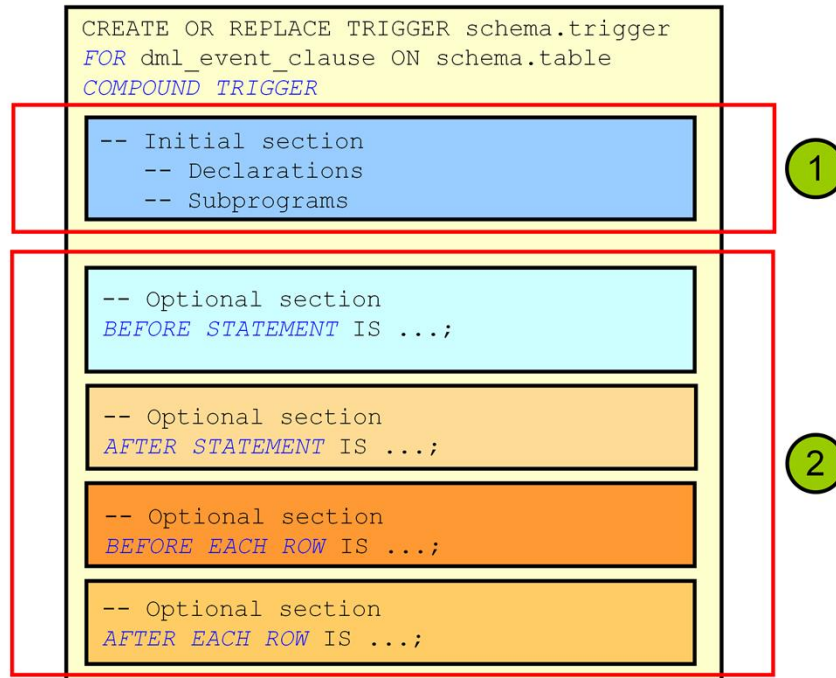
- Describe compound triggers
- Describe mutating tables
- Create triggers on DDL statements
- Create triggers on system events
- Display information about triggers

In this lesson, you learn how to create and use database triggers.

What Is a Compound Trigger?

- A compound trigger is a single trigger on a table that allows you to specify actions for each of the four triggering timing points:
 - Before the firing statement
 - Before each row that the firing statement affects
 - After each row that the firing statement affects
 - After the firing statement

Compound Trigger Structure for Tables



A compound trigger has two main sections:

- An initial section where variables and subprograms are declared. The code in this section executes before any of the code in the optional section.
- An optional section that defines the code for each possible trigger point. Depending on whether you are defining a compound trigger for a table or for a view, these triggering points are different and are listed in the image shown above and on the following page. The code for the triggering points must follow the order shown above.

Note: For additional information about Compound Triggers, refer to *Oracle Database PL/SQL Language Reference*.

Timing-Point Sections of a Table Compound Trigger

A compound trigger defined on a table has one or more of the following timing-point sections. Timing-point sections must appear in the order shown in the table.

Timing Point	Compound Trigger Section
Before the triggering statement executes	<code>BEFORE statement</code>
After the triggering statement executes	<code>AFTER statement</code>
Before each row that the triggering statement affects	<code>BEFORE EACH ROW</code>
After each row that the triggering statement affects	<code>AFTER EACH ROW</code>

Note

Timing-point sections must appear in the order shown in the slide. If a timing-point section is absent, nothing happens at its timing point.

The Benefits of Using a Compound Trigger

You can use compound triggers to:

- Program an approach where you want the actions you implement for the various timing points to share common data.
- Accumulate rows destined for a second table so that you can periodically bulk-insert them
- Avoid the mutating-table error (`ORA-04091`) by allowing rows destined for a second table to accumulate and then bulk-inserting them

Compound Trigger Restrictions

- A compound trigger must be a DML trigger and defined on either a table or a view.
- The body of a compound trigger must be compound trigger block, written in PL/SQL.
- A compound trigger body cannot have an initialization block; therefore, it cannot have an exception section.
- The firing order of compound triggers is not guaranteed unless you use the `FOLLOWS` clause.

The following are some of the restrictions when working with compound triggers:

- The body of a compound trigger must compound trigger block, written in PL/SQL.
- A compound trigger must be a DML trigger.
- A compound trigger must be defined on either a table or a view.
- A compound trigger body cannot have an initialization block; therefore, it cannot have an exception section. This is not a problem, because the `BEFORE STATEMENT` section always executes exactly once before any other timing-point section executes.
- An exception that occurs in one section must be handled in that section. It cannot transfer control to another section.
- `:OLD`, `:NEW`, and `:PARENT` cannot appear in the declaration section, the `BEFORE STATEMENT` section, or the `AFTER STATEMENT` section.
- The firing order of compound triggers is not guaranteed unless you use the `FOLLOWS` clause.

Mutating Tables

- A mutating table is a table that is being modified by an `UPDATE`, `DELETE`, or `INSERT` statement, or
- A table that might be updated by the effects of a `DELETE CASCADE` constraint
- A mutating table error (`ORA-4091`) occurs when a row-level trigger attempts to change or examine a table that is already undergoing change via a DML statement.

Rules Governing Triggers

Reading and writing data using triggers is subject to certain rules. The restrictions apply only to row triggers, unless a statement trigger is fired as a result of `ON DELETE CASCADE`.

Mutating Table

A mutating table is a table that is currently being modified by an `UPDATE`, `DELETE`, or `INSERT` statement, or a table that might need to be updated by the effects of a declarative `DELETE CASCADE` referential integrity action. For `STATEMENT` triggers, a table is not considered a mutating table.

A mutating table error (`ORA-4091`) occurs when a row-level trigger attempts to change or examine a table that is already undergoing change via a DML statement.

The triggered table itself is a mutating table, as well as any table referencing it with the `FOREIGN KEY` constraint. This restriction prevents a row trigger from seeing an inconsistent set of data.

Mutating Table: Example

```
CREATE OR REPLACE TRIGGER check_salary
  BEFORE INSERT OR UPDATE OF salary, job_id
  ON employees
  FOR EACH ROW
  WHEN (NEW.job_id <> 'AD_PRES')
DECLARE
  v_minsalary employees.salary%TYPE;
  v_maxsalary employees.salary%TYPE;
BEGIN
  SELECT MIN(salary), MAX(salary)
    INTO   v_minsalary, v_maxsalary
  FROM     employees
  WHERE    job_id = :NEW.job_id;
  IF :NEW.salary < v_minsalary OR :NEW.salary > v_maxsalary THEN
    RAISE_APPLICATION_ERROR(-20505, 'Out of range');
  END IF;
END;
/
```

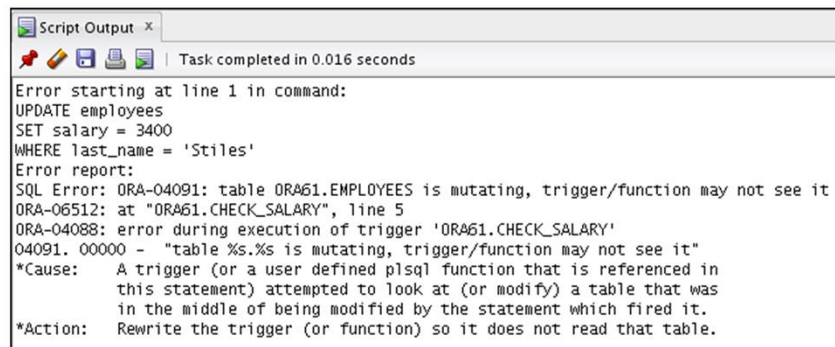
ORACLE

The `CHECK_SALARY` trigger in the slide example attempts to guarantee that whenever a new employee is added to the `EMPLOYEES` table or whenever an existing employee's salary or job ID is changed, the employee's salary falls within the established salary range for the employee's job.

When an employee record is updated, the `CHECK_SALARY` trigger is fired for each row that is updated. The trigger code queries the same table that is being updated. Therefore, it is said that the `EMPLOYEES` table is a mutating table.

Mutating Table: Example

```
UPDATE employees
SET salary = 3400
WHERE last_name = 'Stiles';
```



```
Script Output x
Task completed in 0.016 seconds

Error starting at line 1 in command:
UPDATE employees
SET salary = 3400
WHERE last_name = 'Stiles'
Error report:
SQL Error: ORA-04091: table ORA61.EMPLOYEES is mutating, trigger/function may not see it
ORA-06512: at "ORA61.CHECK_SALARY", line 5
ORA-04088: error during execution of trigger 'ORA61.CHECK_SALARY'
04091. 00000 - "table %.s is mutating, trigger/function may not see it"
*Cause:      A trigger (or a user defined plsql function that is referenced in
              this statement) attempted to look at (or modify) a table that was
              in the middle of being modified by the statement which fired it.
*Action:     Rewrite the trigger (or function) so it does not read that table.
```

ORACLE

In the example in the slide, the trigger code tries to read or select data from a mutating table.

If you restrict the salary within a range between the minimum existing value and the maximum existing value, then you get a run-time error. The `EMPLOYEES` table is mutating, or in a state of change; therefore, the trigger cannot read from it.

Remember that functions can also cause a mutating table error when they are invoked in a DML statement.

Possible Solutions

Possible solutions to this mutating table problem include the following:

- Use a compound trigger as described earlier in this lesson.
- Store the summary data (the minimum salaries and the maximum salaries) in another summary table, which is kept up-to-date with other DML triggers.
- Store the summary data in a PL/SQL package, and access the data from the package. This can be done in a `BEFORE` statement trigger.

Depending on the nature of the problem, a solution can become more convoluted and difficult to solve. In this case, consider implementing the rules in the application or middle tier and avoid using database triggers to perform overly complex business rules. An insert statement

in the code example in the slide will not generate a mutating table example.

Using a Compound Trigger to Resolve the Mutating Table Error

```
CREATE OR REPLACE TRIGGER check_salary
  FOR INSERT OR UPDATE OF salary, job_id
  ON employees
  WHEN (NEW.job_id <> 'AD_PRES')
  COMPOUND TRIGGER

  TYPE salaries_t          IS TABLE OF employees.salary%TYPE;
  min_salaries             salaries_t;
  max_salaries             salaries_t;

  TYPE department_ids_t    IS TABLE OF employees.department_id%TYPE;
  department_ids           department_ids_t;

  TYPE department_salaries_t IS TABLE OF employees.salary%TYPE
                           INDEX BY VARCHAR2(80);
  department_min_salaries  department_salaries_t;
  department_max_salaries  department_salaries_t;

  -- example continues on next slide
```

ORACLE

The `CHECK_SALARY` compound trigger resolves the mutating table error in the earlier example. This is achieved by storing the values in PL/SQL collections, and then performing a bulk insert/update in the “before statement” section of the compound trigger. This code works from Oracle Database 11g onwards. In the example in the slide, PL/SQL collections are used. The element types used are based on the `SALARY` and `DEPARTMENT_ID` columns from the `EMPLOYEES` table. To create collections, you define a collection type, and then declare variables of that type. Collections are instantiated when you enter a block or subprogram, and cease to exist when you exit. `min_salaries` is used to hold the minimum salary for each department and `max_salaries` is used to hold the maximum salary for each department. `department_ids` is used to hold the department IDs. If the employee who earns the minimum or maximum salary does not have an assigned department, you use the `NVL` function to store `-1` for the department id instead of `NULL`. Next, you collect the minimum salary, maximum salary, and the department ID using a bulk insert into the `min_salaries`, `max_salaries`, and `department_ids` respectively grouped by department ID. The select statement returns 13 rows. The values of the `department_ids` are used as an index for the `department_min_salaries` and `department_max_salaries` tables. Therefore, the index for those two tables (`VARCHAR2`) represents the actual `department_ids`.

Using a Compound Trigger to Resolve the Mutating Table Error

```
. . .
BEFORE STATEMENT IS
BEGIN
    SELECT MIN(salary), MAX(salary), NVL(department_id, -1)
    BULK COLLECT INTO min_salaries, max_salaries, department_ids
    FROM employees
    GROUP BY department_id;
    FOR j IN 1..department_ids.COUNT() LOOP
        department_min_salaries(department_ids(j)) := min_salaries(j);
        department_max_salaries(department_ids(j)) := max_salaries(j);
    END LOOP;
END BEFORE STATEMENT;

AFTER EACH ROW IS
BEGIN
    IF :NEW.salary < department_min_salaries(:NEW.department_id)
    OR :NEW.salary > department_max_salaries(:NEW.department_id) THEN
        RAISE_APPLICATION_ERROR(-20505, 'New Salary is out of acceptable
        range');
    END IF;
END AFTER EACH ROW;
END check_salary;
```

ORACLE

After each row is added, if the new salary is less than the minimum salary for that department or greater than the department's maximum salary, then an error message is displayed.

To test the newly created compound trigger, issue the following statement:

```
UPDATE employees
SET salary = 3400
WHERE last_name = 'Stiles';
```

1 rows updated

To ensure that the salary for employee Stiles was updated, issue the following query using the F9 key in SQL Developer:

```
SELECT employee_id, first_name, last_name, job_id, department_id,
       salary
FROM employees
WHERE last_name = 'Stiles';
```

	EMPLOYEE_ID	FIRST_NAME	LAST_NAME	JOB_ID	DEPARTMENT_ID	SALARY
1	138	Stephen	Stiles	ST_CLERK	50	3400

Creating Triggers on DDL Statements

```
CREATE [OR REPLACE] TRIGGER trigger_name
BEFORE | AFTER -- Timing
[ddl_event1 [OR ddl_event2 OR ...]]
ON {DATABASE | SCHEMA}
trigger_body
```

Sample DDL Events	Fires When
CREATE	Any database object is created using the CREATE command.
ALTER	Any database object is altered using the ALTER command.
DROP	Any database object is dropped using the DROP command.

ORACLE

You can specify one or more types of DDL statements that can cause the trigger to fire. You can create triggers for these events on `DATABASE` or `SCHEMA` unless otherwise noted. You can also specify `BEFORE` and `AFTER` for the timing of the trigger. The Oracle database fires the trigger in the existing user transaction.

You cannot specify as a triggering event any DDL operation performed through a PL/SQL procedure.

The trigger body in the syntax in the slide represents a complete PL/SQL block.

DDL triggers fire only if the object being created is a cluster, function, index, package, procedure, role, sequence, synonym, table, tablespace, trigger, type, view, or user.

Creating Triggers on DDL Statements

- In case of schema/user auditing using DDL trigger, create a table in the same schema which you are auditing.
- In case of Database auditing using DDL trigger, create a table in sys or system schema (sys or system both schemas can be used to perform database auditing).

```
CREATE TABLE schema_audit ( ddl_date DATE,  
ddl_user VARCHAR2(15), object_created VARCHAR2(15),  
object_name VARCHAR2(15),  
ddl_operation VARCHAR2(15));
```

Login to pdb1 as HR user

```
CREATE OR REPLACE TRIGGER hr_audit_tr
AFTER DDL ON SCHEMA
BEGIN
INSERT INTO schema_audit VALUES
( sysdate,
sys_context('USERENV','CURRENT_USER'),
ora_dict_obj_type,
ora_dict_obj_name,
ora_sysevent);
END;
/
```


Login to pdb1 as sys user

```
CREATE OR REPLACE TRIGGER hr_audit_tr
AFTER DDL ON SCHEMA
BEGIN
INSERT INTO schema_audit VALUES
( sysdate,
sys_context('USERENV','CURRENT_USER'),
ora_dict_obj_type,
ora_dict_obj_name,
ora_sysevent);
END;
/
```

Creating Database-Event Triggers

- Triggering user event:
 - CREATE, ALTER, or DROP
 - Logging on or off
- Triggering database or system event:
 - Shutting down or starting up the database
 - A specific error (or any error) being raised

ORACLE

Creating Database Triggers

Before coding the trigger body, decide on the components of the trigger.

Triggers on system events can be defined at the database or schema level. For example, a database shutdown trigger is defined at the database level. Triggers on data definition language (DDL) statements, or a user logging on or off, can also be defined at either the database level or schema level. Triggers on data manipulation language (DML) statements are defined on a specific table or a view.

A trigger defined at the database level fires for all users whereas a trigger defined at the schema or table level fires only when the triggering event involves that schema or table.

Triggering events that can cause a trigger to fire:

- A data definition statement on an object in the database or schema
- A specific user (or any user) logging on or off
- A database shutdown or startup
- Any error that occurs

Creating Triggers on System Events

```
CREATE [OR REPLACE] TRIGGER trigger_name
BEFORE | AFTER -- timing
[database_event1 [OR database_event2 OR ...]]
ON {DATABASE | SCHEMA}
trigger_body
```

Database Event	Triggers Fires When
AFTER SERVERERROR	An Oracle error is raised
AFTER LOGON	A user logs on to the database
BEFORE LOGOFF	A user logs off the database
AFTER STARTUP	The database is opened
BEFORE SHUTDOWN	The database is shut down normally

You can create triggers for the events listed in the table in the slide on DATABASE or SCHEMA, except SHUTDOWN and STARTUP, which apply only to DATABASE.

LOGON and LOGOFF Triggers: Example

```
-- Create the log_trig_table shown in the notes page
-- first

CREATE OR REPLACE TRIGGER logon_trig
AFTER LOGON ON SCHEMA
BEGIN
    INSERT INTO log_trig_table(user_id,log_date,action)
    VALUES (USER, SYSDATE, 'Logging on');
END;
/
```

```
CREATE OR REPLACE TRIGGER logoff_trig
BEFORE LOGOFF ON SCHEMA
BEGIN
    INSERT INTO log_trig_table(user_id,log_date,action)
    VALUES (USER, SYSDATE, 'Logging off');
END;
/
```

ORACLE

You can create these triggers to monitor how often you log on and off, or you may want to write a report that monitors the length of time for which you are logged on. When you specify ON SCHEMA, the trigger fires for the specific user. If you specify ON DATABASE, the trigger fires for all users.

The definition of the log_trig_table used in the slide examples is as follows:

```
CREATE TABLE log_trig_table(
    user_id VARCHAR2(30),
    log_date DATE,
    action VARCHAR2(40))
/
```

Benefits of Database-Event Triggers

- Improved data security:
 - Provides enhanced and complex security checks
 - Provides enhanced and complex auditing
- Improved data integrity:
 - Enforces dynamic data integrity constraints
 - Enforces complex referential integrity constraints
 - Ensures that related operations are performed together implicitly

ORACLE

You can use database triggers:

- As alternatives to features provided by the Oracle server
- If your requirements are more complex or more simple than those provided by the Oracle server
- If your requirements are not provided by the Oracle server at all

System Privileges Required to Manage Triggers

The following system privileges are required to manage triggers:

- The privileges that enable you to create, alter, and drop triggers in any schema:
 - `GRANT CREATE TRIGGER TO ora61`
 - `GRANT ALTER ANY TRIGGER TO ora61`
 - `GRANT DROP ANY TRIGGER TO ora61`
- The privilege that enables you to create a trigger on the database:
 - `GRANT ADMINISTER DATABASE TRIGGER TO ora61`
- The `EXECUTE` privilege (if your trigger refers to any objects that are not in your schema)

To create a trigger in your schema, you need the `CREATE TRIGGER` system privilege, and you must own the table specified in the triggering statement, have the `ALTER` privilege for the table in the triggering statement, or have the `ALTER ANY TABLE` system privilege. You can alter or drop your triggers without any further privileges being required.

If the `ANY` keyword is used, you can create, alter, or drop your own triggers and those in another schema and can be associated with any user's table.

You do not need any privileges to invoke a trigger in your schema. A trigger is invoked by DML statements that you issue. But if your trigger refers to any objects that are not in your schema, the user creating the trigger must have the `EXECUTE` privilege on the referenced procedures, functions, or packages, and not through roles.

To create a trigger on `DATABASE`, you must have the `ADMINISTER DATABASE TRIGGER` privilege. If this privilege is later revoked, you can drop the trigger but you cannot alter it.

Note: Similar to stored procedures, statements in the trigger body use the privileges of the trigger owner, not the privileges of the user executing the operation that fires the trigger.

Quiz

Which of the following statements are true?

- a. A trigger is defined with a `CREATE TRIGGER` statement
- b. A trigger's source code is contained in the `USER_TRIGGERS` data dictionary.
- c. A trigger is explicitly invoked.
- d. A trigger is implicitly invoked by DML.
- e. The `COMMIT`, `SAVEPOINT`, and `ROLLBACK` are not allowed when working with a trigger.

Answer: a, b, d, e

Summary

In this lesson, you should have learned how to:

- Describe compound triggers
- Describe mutating tables
- Create triggers on DDL statements
- Create triggers on system events
- Display information about triggers