Database 2 Day Developer's Guide

6 Using Triggers

This chapter contains:

About Triggers

A **trigger** is a PL/SQL unit that is stored in the database and (if it is in the enabled state) automatically executes ("fires") in response to a specified event.

A trigger has this structure:

```
TRIGGER trigger_name triggering_event [ trigger_restriction ] BEGIN triggered_action; END;
```

The *`trigger_name`* must be unique for triggers in the schema. A trigger can have the same name as another kind of object in the schema (for example, a table); however, Oracle recommends using a naming convention that avoids confusion.

If the trigger is in the **enabled** state, the *`triggering_event`* causes the database to execute the *`triggered_action`* if the *`trigger_restriction`* is either `TRUE` or omitted. The *`triggering_event`* is associated with either a table, a view, a schema, or the database, and it is one of these:

- DML statement (described in "About Data Manipulation Language (DML) Statements" (tdddg_dml.htm#BCGBGBIE))
- DDL statement (described in "About Data Definition Language (DDL) Statements" (tdddg_objects.htm#CIHGAJDJ))
- Database operation (`SERVERERROR`, `LOGON`, `LOGOFF`, `STARTUP`, or `SHUTDOWN`)

If the trigger is in the **disabled** state, the *`triggering_event`* does not cause the database to execute the *`triggered_action`*, even if the *`trigger_restriction`* is `TRUE` or omitted.

By default, a trigger is created in the enabled state. You can disable an enabled trigger, and enable a disabled trigger.

Unlike a subprogram, a trigger cannot be invoked directly. A trigger is invoked only by its triggering event, which can be caused by any user or application. You might be unaware that a trigger is executing unless it causes an error that is not handled properly.

A **simple trigger** can fire at exactly one of these **timing points**:

- Before the triggering event executes (statement-level `BEFORE` trigger)
- After the triggering event executes (statement-level `AFTER` trigger)
- Before each row that the event affects (row-level `BEFORE` trigger)

- After each row that the event affects (row-level `AFTER` trigger)

A **compound trigger** can fire at multiple timing points. For information about compound triggers, see *Oracle Database PL/SQL Language Reference* (../LNPLS/triggers.htm#LNPLS2005) .

An **`INSTEAD OF` trigger** is defined on a view, and its triggering event is a DML statement. Instead of executing the DML statement, Oracle Database executes the `INSTEAD OF` trigger. For more information, see "Creating an INSTEAD OF Trigger" (#BABECIAE) .

A **system trigger** is defined on a schema or the database. A trigger defined on a schema fires for each event associated with the owner of the schema (the current user). A trigger defined on a database fires for each event associated with all users.

One use of triggers is to enforce business rules that apply to all client applications. For example, suppose that data added to the `EMPLOYEES` table must have a certain format, and that many client applications can add data to this table. A trigger on the table can ensure the proper format of all data added to it. Because the trigger executes whenever any client adds data to the table, no client can circumvent the rules, and the code that enforces the rules can be stored and maintained only in the trigger, rather than in every client application. For other uses of triggers, see *Oracle Database PL/SQL Language Reference* (../LNPLS/triggers.htm#LNPLS723) .

See Also:

Oracle Database PL/SQL Language Reference (../LNPLS/overview.htm#LNPLS020) for complete information about triggers

Creating Triggers

To create triggers, use either the SQL Developer tool Create Trigger or the DDL statement `CREATE TRIGGER`. This section shows how to use both of these ways to create triggers.

By default, a trigger is created in the enabled state. To create a trigger in disabled state, use the `CREATE TRIGGER` statement with the `DISABLE` clause.

Note:

To create triggers, you must have appropriate privileges; however, for this discussion, you do not need this additional information.

This section contains:

- About OLD and NEW Pseudorecords (#BCFIGJAE)
- Tutorial: Creating a Trigger that Logs Table Changes (#BABGEAIA)
- Tutorial: Creating a Trigger that Generates a Primary Key for a Row Before It Is Inserted (#BABCGBBD)
- Creating an INSTEAD OF Trigger (#BABECIAE)
- Tutorial: Creating Triggers that Log LOGON and LOGOFF Events (#BABFBEJH)

Note:

To do the tutorials in this document, you must be connected to Oracle Database as the user `HR` from SQL Developer.

See Also:

- *Oracle SQL Developer User's Guide* (../RPTUG/dialogs.htm#RPTUG30000) for information about SQL Developer dialog boxes for creating objects
- *Oracle Database PL/SQL Language Reference* (../LNPLS/create_trigger.htm#LNPLS01374) for more information about the `CREATE TRIGGER` statement
- "Editing Installation Scripts that Create Triggers" (tdddg_deploying.htm#BABJCJIJ)

About OLD and NEW Pseudorecords

When a row-level trigger fires, the PL/SQL runtime system creates and populates the two pseudorecords `OLD` and `NEW`. They are called pseudorecords because they have some, but not all, of the properties of records.

For the row that the trigger is processing:

- For an `INSERT` trigger, `OLD` contains no values, and `NEW` contains the new values.
- For an `UPDATE` trigger, `OLD` contains the old values, and `NEW` contains the new values.
- For a `DELETE` trigger, `OLD` contains the old values, and `NEW` contains no values.

To reference a pseudorecord, put a colon before its name—:`OLD` or :`NEW`—as in Example 6-1 (#BABBFAGD) .

> **See Also:**
> *Oracle Database PL/SQL Language Reference* (../LNPLS/triggers.htm#LNPLS99955) for more information about `OLD` and `NEW` pseudorecords

Tutorial: Creating a Trigger that Logs Table Changes

This tutorial shows how to use the `CREATE TRIGGER` statement to create a trigger, `EVAL_CHANGE_TRIGGER`, which adds a row to the table `EVALUATIONS_LOG` whenever an `INSERT`, `UPDATE`, or `DELETE` statement changes the `EVALUATIONS` table.

The trigger adds the row *after* the triggering statement executes, and uses the **conditional predicates** `INSERTING`, `UPDATING`, and `DELETING` to determine which of the three possible DML statements fired the trigger.

`EVAL_CHANGE_TRIGGER` is a **statement-level trigger** and an **AFTER trigger**.

To create EVALUATIONS_LOG and EVAL_CHANGE_TRIGGER:

1. Create the `EVALUATIONS_LOG` table:

   ```
   CREATE TABLE EVALUATIONS_LOG ( log_date DATE , action VARCHAR2(50));
   ```

2. Create `EVAL_CHANGE_TRIGGER`:

   ```
   CREATE OR REPLACE TRIGGER EVAL_CHANGE_TRIGGER AFTER INSERT OR UPDATE OR DELETE ON EVALUATIONS DECLARE
   log_action EVALUATIONS_LOG.action%TYPE; BEGIN IF INSERTING THEN log_action := 'Insert'; ELSIF UPDATING THEN
   log_action := 'Update'; ELSIF DELETING THEN log_action := 'Delete'; ELSE DBMS_OUTPUT.PUT_LINE('This code is
   not reachable.'); END IF; INSERT INTO EVALUATIONS_LOG (log_date, action) VALUES (SYSDATE, log_action); END;
   ```

> **See Also:**
> *Oracle Database PL/SQL Language Reference* (../LNPLS/triggers.htm#LNPLS750) for more information about conditional predicates

Tutorial: Creating a Trigger that Generates a Primary Key for a Row Before It Is Inserted

The sequence `EVALUATIONS_SEQUENCE` (created in "Tutorial: Creating a Sequence" (tdddg_objects.htm#BABJIDEG)) generates primary keys for the `EVALUATIONS` table (created in "Creating Tables" (tdddg_objects.htm#BABFEIDE)). However, these primary keys are not inserted into the table automatically.

This tutorial shows how to use the SQL Developer Create Trigger tool to create a trigger named `NEW_EVALUATION_TRIGGER`, which fires *before* a row is inserted into the `EVALUATIONS` table, and generates the unique number for the primary key of that row, using `EVALUATIONS_SEQUENCE`. The trigger fires once *for each row* affected by the triggering `INSERT` statement.

`NEW_EVALUATION_TRIGGER` is a **row-level trigger** and a **BEFORE trigger**.

To create the NEW_EVALUATION trigger:

1. In the Connections frame, expand **hr_conn**.
2. In the list of schema object types, right-click **Triggers**.

3. In the list of choices, click **New Trigger**.

4. In the Create Trigger window:

    a. In the Name field, type `NEW_EVALUATION_TRIGGER` over the default value `TRIGGER1`.

    b. For Base Object, select **`EVALUATIONS`** from the menu.

    c. Move **`INSERT`** from Available Events to Selected Events.

    (Select **`INSERT`** and click **>**.)

    d. Deselect the option **Statement Level**.

    e. Click **OK**.

    The `NEW_EVALUATION_TRIGGER` pane opens, showing the `CREATE TRIGGER` statement that created the trigger:

```
CREATE OR REPLACE TRIGGER NEW_EVALUATION_TRIGGER BEFORE INSERT ON EVALUATIONS FOR EACH ROW BEGIN NULL;
END;
```

    The title of the `NEW_EVALUATION_TRIGGER` pane is in italic font, indicating that the trigger is not yet saved in the database.

5. In the `CREATE TRIGGER` statement, replace `NULL` with this:

```
:NEW.evaluation_id := evaluations_sequence.NEXTVAL
```

6. From the File menu, select **Save**.

   Oracle Database compiles the procedure and saves it. The title of the `NEW_EVALUATION_TRIGGER` pane is no longer in italic font.

Creating an INSTEAD OF Trigger

A view presents the output of a query as a table. If you want to change a view as you would change a table, then you must create `INSTEAD OF` triggers. Instead of changing the view, they change the underlying tables.

For example, consider the view `EMP_LOCATIONS`, whose `NAME` column is created from the `LAST_NAME` and `FIRST_NAME` columns of the `EMPLOYEES` table:

```
CREATE VIEW EMP_LOCATIONS AS SELECT e.EMPLOYEE_ID, e.LAST_NAME || ', ' || e.FIRST_NAME NAME, d.DEPARTMENT_NAME
DEPARTMENT, l.CITY CITY, c.COUNTRY_NAME COUNTRY FROM EMPLOYEES e, DEPARTMENTS d, LOCATIONS l, COUNTRIES c WHERE
e.DEPARTMENT_ID = d.DEPARTMENT_ID AND d.LOCATION_ID = l.LOCATION_ID AND l.COUNTRY_ID = c.COUNTRY_ID ORDER BY
LAST_NAME;
```

To update the view `EMP_LOCATIONS.NAME` (created in "Creating Views with the CREATE VIEW Statement" (tdddg_objects.htm#BABFHHIA)), you must update `EMPLOYEES.LAST_NAME` and `EMPLOYEES.FIRST_NAME`. This is what the `INSTEAD OF` trigger in Example 6-1 (#BABBFAGD) does.

`NEW` and `OLD` are **pseudorecords** that the PL/SQL runtime engine creates and populates whenever a row-level trigger fires. `OLD` and `NEW` store the original and new values, respectively, of the record being processed by the trigger. They are called pseudorecords because they do not have all properties of PL/SQL records.

Example 6-1 Creating an INSTEAD OF Trigger

```
CREATE OR REPLACE TRIGGER update_name_view_trigger INSTEAD OF UPDATE ON emp_locations BEGIN UPDATE employees SET
first_name = substr( :NEW.name, instr( :new.name, ',' )+2), last_name = substr( :NEW.name, 1, instr( :new.name,
',')-1) WHERE employee_id = :OLD.employee_id; END;
```

See Also:

- *Oracle Database PL/SQL Language Reference* (../LNPLS/triggers.htm#LNPLS20041) for more information about `INSTEAD OF` triggers
- *Oracle Database PL/SQL Language Reference* (../LNPLS/triggers.htm#LNPLS99955) for more information about `OLD` and `NEW`

Tutorial: Creating Triggers that Log LOGON and LOGOFF Events

This tutorial shows how to use the `CREATE TRIGGER` statement to create two triggers, `HR_LOGON_TRIGGER` and `HR_LOGOFF_TRIGGER`. *After* someone logs on as user `HR`, `HR_LOGON_TRIGGER` adds a row to the table `HR_USERS_LOG`. *Before* someone logs off as user `HR`, `HR_LOGOFF_TRIGGER` adds a row to the table `HR_USERS_LOG`.

`HR_LOGON_TRIGGER` and `HR_LOGOFF_TRIGGER` are **system triggers**. `HR_LOGON_TRIGGER` is an **AFTER trigger** and `HR_LOGOFF_TRIGGER` is a **BEFORE trigger**.

To create HR_USERS_LOG, HR_LOGON_TRIGGER, and HR_LOGOFF_TRIGGER:

1. Create the `HR_USERS_LOG` table:

   ```
   CREATE TABLE hr_users_log ( user_name VARCHAR2(30), activity VARCHAR2(20), event_date DATE );
   ```

2. Create `hr_logon_trigger`:

   ```
   CREATE OR REPLACE TRIGGER hr_logon_trigger AFTER LOGON ON HR.SCHEMA BEGIN INSERT INTO hr_users_log
   (user_name, activity, event_date) VALUES (USER, 'LOGON', SYSDATE); END;
   ```

3. Create `hr_logoff_trigger`:

   ```
   CREATE OR REPLACE TRIGGER hr_logoff_trigger BEFORE LOGOFF ON HR.SCHEMA BEGIN INSERT INTO hr_users_log
   (user_name, activity, event_date) VALUES (USER, 'LOGOFF', SYSDATE); END;
   ```

See Also:

Oracle Database PL/SQL Language Reference (../LNPLS/triggers.htm#LNPLS99887) for more information about system triggers

Changing Triggers

To change a trigger, use either the SQL Developer tool Edit or the DDL statement `CREATE TRIGGER` with the `OR REPLACE` clause.

To change a trigger using the Edit tool:

1. In the Connections frame, expand **hr_conn**.

2. In the list of schema object types, expand **Triggers**.

3. In the list of triggers, click the trigger to change.

4. In the frame to the right of the Connections frame, the Code pane appears, showing the code that created the trigger.

   The Code pane is in write mode. (Clicking the pencil icon switches the mode from write mode to read only, or the reverse.)

5. In the Code pane, change the code.

   The title of the pane is in italic font, indicating that the change is not yet saved in the database.

6. From the File menu, select **Save**.

   Oracle Database compiles the trigger and saves it. The title of the pane is no longer in italic font.

See Also:

- "About Data Definition Language (DDL) Statements" (tdddg_objects.htm#CIHGAJDJ) for general information that applies to the `CREATE OR REPLACE TRIGGER` statement

- *Oracle Database PL/SQL Language Reference* (../LNPLS/create_trigger.htm#LNPLS01374) for more information about the `CREATE OR REPLACE TRIGGER` statement

Disabling and Enabling Triggers

You might need to temporarily disable triggers if they reference objects that are unavailable, or if you must upload a large amount of data without the delay that triggers cause (as in a recovery operation). After the referenced objects become available, or you have finished uploading the data, you can re-enable the triggers.

This section contains:

- Disabling or Enabling a Single Trigger (#BABIJDCB)
- Disabling or Enabling All Triggers on a Single Table (#BABEHHFH)

See Also:

- *Oracle Database PL/SQL Language Reference* (../LNPLS/alter_trigger.htm#LNPLS99996) for more information about the `ALTER TRIGGER` statement
- *Oracle Database SQL Language Reference* (../SQLRF/statements_3001.htm#SQLRF01001) for more information about the `ALTER TABLE` statement

Disabling or Enabling a Single Trigger

To disable or enable a single trigger, use either the Disable Trigger or Enable Trigger tool or the `ALTER TRIGGER` statement with the `DISABLE` or `ENABLE` clause.

For example, these statements disable and enable the `eval_change_trigger`:

```
ALTER TRIGGER eval_change_trigger DISABLE; ALTER TRIGGER eval_change_trigger ENABLE;
```

To use the Disable Trigger or Enable Trigger tool:

1. In the Connections frame, expand **hr_conn**.
2. In the list of schema object types, expand **Triggers**.
3. In the list of triggers, right-click the desired trigger.
4. In the list of choices, select **Disable** or **Enable**.
5. In the Disable or Enable window, click **Apply**.
6. In the Confirmation window, click **OK**.

Disabling or Enabling All Triggers on a Single Table

To disable or enable all triggers on a specific table, use either the Disable All Triggers or Enable All Triggers tool or the `ALTER TABLE` statement with the `DISABLE ALL TRIGGERS` or `ENABLE ALL TRIGGERS` clause.

For example, these statements disable and enable all triggers on the `evaluations` table:

```
ALTER TABLE evaluations DISABLE ALL TRIGGERS; ALTER TABLE evaluations ENABLE ALL TRIGGERS;
```

To use the Disable All Triggers or Enable All Triggers tool:

1. In the Connections frame, expand **hr_conn**.
2. In the list of schema object types, expand **Tables**.
3. In the list of tables, right-click the desired table.
4. In the list of choices, select **Triggers**.
5. In the list of choices, select **Disable All** or **Enable All**.
6. In the Disable All or Enable All window, click **Apply**.
7. In the Confirmation window, click **OK**.

About Trigger Compilation and Dependencies

Running a `CREATE TRIGGER` statement compiles the trigger being created. If this compilation causes an error, then the `CREATE TRIGGER` statement fails. To see the compilation errors, use this statement:

```
SELECT * FROM USER_ERRORS WHERE TYPE = 'TRIGGER';
```

Compiled triggers depend on the schema objects on which they are defined. For example, `NEW_EVALUATION_TRIGGER` depends on the `EVALUATIONS` table:

```
CREATE OR REPLACE TRIGGER NEW_EVALUATION_TRIGGER BEFORE INSERT ON EVALUATIONS FOR EACH ROW BEGIN
:NEW.evaluation_id := evaluations_seq.NEXTVAL; END;
```

To see the schema objects on which triggers depend, use this statement:

```
SELECT * FROM ALL_DEPENDENCIES WHERE TYPE = 'TRIGGER';
```

If an object on which a trigger depends is dropped, or changed such that there is a mismatch between the trigger and the object, then the trigger is invalidated. The next time the trigger is invoked, it is recompiled. To recompile a trigger immediately, use the `ALTER TRIGGER` statement with the `COMPILE` clause. For example:

```
ALTER TRIGGER NEW_EVALUATION_TRIGGER COMPILE;
```

See Also:

Oracle Database PL/SQL Language Reference (../LNPLS/triggers.htm#LNPLS2007) for more information about trigger compilation and dependencies

Dropping Triggers

You must drop a trigger before dropping the objects on which it depends.

To drop a trigger, use either the SQL Developer Connections frame and Drop tool, or the DDL statement `DROP TRIGGER`.

This statement drops the trigger `EVAL_CHANGE_TRIGGER`:

```
DROP TRIGGER EVAL_CHANGE_TRIGGER;
```

To drop a trigger using the Drop tool:

1. In the Connections frame, expand **hr_conn**.
2. In the list of schema object types, expand **Triggers**.
3. In the list of triggers, right-click the name of the trigger to drop.
4. In the list of choices, click **Drop Trigger**.
5. In the Drop window, click **Apply**.
6. In the Confirmation window, click **OK**.

See Also:

- "About Data Definition Language (DDL) Statements" (tdddg_objects.htm#CIHGAJDJ) for general information that applies to the `DROP TRIGGER` statement
- *Oracle Database PL/SQL Language Reference* (../LNPLS/drop_trigger.htm#LNPLS99990) for information about the `DROP TRIGGER` statement

(https://docs.oracle.com/database/121/TDDDG/tdddg_subprograms.htm) (https://docs.oracle.com/database/121/TDDDG/tdddg_globalization.htm)

(#)