Overview of Triggers

Like a stored procedure, a trigger is a named PL/SQL unit that is stored in the database and can be invoked repeatedly. Unlike a stored procedure, you can enable and disable a trigger, but you cannot explicitly invoke it. While a trigger is enabled, the database automatically invokes it—that is, the trigger fires—whenever its triggering event occurs. While a trigger is disabled, it does not fire.

You create a trigger with the CREATE TRIGGER statement. You specify the triggering event in terms of triggering statements and the item on which they act. The trigger is said to be created on or defined on the item, which is either a table, a view, a schema, or the database. You also specify the timing point, which determines whether the trigger fires before or after the triggering statement runs and whether it fires for each row that the triggering statement affects. By default, a trigger is created in the enabled state. For more information about the CREATE TRIGGER statement, see "CREATE TRIGGER Statement".

If the trigger is created on a table or view, then the triggering event is composed of DML statements, and the trigger is called a DML trigger. For more information, see "DML Triggers".

If the trigger is created on a schema or the database, then the triggering event is composed of either DDL or database operation statements, and the trigger is called a system trigger. For more information, see "System Triggers".

A conditional trigger has a WHEN clause that specifies a SQL condition that the database evaluates for each row that the triggering statement affects. For more information about the WHEN clause, see "WHEN (condition)".

When a trigger fires, tables that the trigger references might be undergoing changes made by SQL statements in other users' transactions. SQL statements running in triggers follow the same rules that standalone SQL statements do. Specifically:

- Queries in the trigger see the current read-consistent materialized view of referenced tables and any data changed in the same transaction.
- Updates in the trigger wait for existing data locks to be released before proceeding.

An INSTEAD OF trigger is either:

- A DML trigger created on either a noneditioning view or a nested table column of a noneditioning view
- A system trigger defined on a CREATE statement

The database fires the INSTEAD OF trigger instead of running the triggering statement.
**Reasons to Use Triggers**

Triggers let you customize your database management system. For example, you can use triggers to:

- Automatically generate virtual column values
- Log events
- Gather statistics on table access
- Modify table data when DML statements are issued against views
- Enforce referential integrity when child and parent tables are on different nodes of a distributed database
- Publish information about database events, user events, and SQL statements to subscribing applications
- Prevent DML operations on a table after regular business hours
- Prevent invalid transactions
- Enforce complex business or referential integrity rules that you cannot define with constraints (see "How Triggers and Constraints Differ")

**Caution:**

Triggers are not reliable security mechanisms, because they are programmatic and easy to disable. For high-assurance security, use Oracle Database Vault, described in *Oracle Database Vault Administrator's Guide*.

**How Triggers and Constraints Differ**

Both triggers and constraints can constrain data input, but they differ significantly.

A trigger always applies to new data only. For example, a trigger can prevent a DML statement from inserting a `NULL` value into a database column, but the column might contain `NULL` values that were inserted into the column before the trigger was defined or while the trigger was disabled.

A constraint can apply either to new data only (like a trigger) or to both new and existing data. Constraint behavior depends on constraint state, as explained in *Oracle Database SQL Language Reference*.

Constraints are easier to write and less error-prone than triggers that enforce the same rules. However, triggers can enforce some complex business rules that constraints cannot. Oracle strongly recommends that you use triggers to constrain data input only in these situations:

- To enforce referential integrity when child and parent tables are on different nodes of a distributed database
- To enforce complex business or referential integrity rules that you cannot define with constraints

**See Also:**

- *Oracle Database Advanced Application Developer's Guide* for information about using constraints to enforce business rules and prevent the entry of invalid information into tables
- "Triggers for Ensuring Referential Integrity" for information about using triggers and constraints to maintain referential integrity between parent and child tables
DML Triggers

A DML trigger is created on either a table or view, and its triggering event is composed of the DML statements DELETE, INSERT, and UPDATE. To create a trigger that fires in response to a MERGE statement, create triggers on the INSERT and UPDATE statements to which the MERGE operation decomposes.

A DML trigger is either simple or compound.

A simple DML trigger fires at exactly one of these timing points:

- Before the triggering statement runs
  (The trigger is called a BEFORE statement trigger or statement-level BEFORE trigger.)
- After the triggering statement runs
  (The trigger is called an AFTER statement trigger or statement-level AFTER trigger.)
- Before each row that the triggering statement affects
  (The trigger is called a BEFORE each row trigger or row-level BEFORE trigger.)
- After each row that the triggering statement affects
  (The trigger is called an AFTER each row trigger or row-level AFTER trigger.)

A compound DML trigger created on a table or editioning view can fire at one, some, or all of the preceding timing points. Compound DML triggers help program an approach where you want the actions that you implement for the various timing points to share common data. For more information, see "Compound DML Triggers".

A simple or compound DML trigger that fires at row level can access the data in the row that it is processing. For details, see "Correlation Names and Pseudorecords".

An INSTEAD OF DML trigger is a DML trigger created on either a noneditioning view or a nested table column of a noneditioning view. For more information, see "INSTEAD OF DML Triggers".

A crossedition trigger is a simple or compound DML trigger for use only in edition-based redefinition. For information about crossedition triggers, see Oracle Database Advanced Application Developer's Guide.

Except in an INSTEAD OF trigger, a triggering UPDATE statement can include a column list. With a column list, the trigger fires only when a specified column is updated. Without a column list, the trigger fires when any column of the associated table is updated. For more information about the column list, see "dml_event_clause".

Topics

- Conditional Predicates for Detecting Triggering DML Statement
- Correlation Names and Pseudorecords
Conditional Predicates for Detecting Triggering DML Statement

The triggering event of a DML trigger can be composed of multiple triggering statements. When one of them fires the trigger, the trigger can determine which one by using these conditional predicates:

<table>
<thead>
<tr>
<th>Conditional Predicate</th>
<th>TRUE if and only if:</th>
</tr>
</thead>
<tbody>
<tr>
<td>INSERTING</td>
<td>An INSERT statement fired the trigger.</td>
</tr>
<tr>
<td>UPDATING</td>
<td>An UPDATE statement fired the trigger.</td>
</tr>
<tr>
<td>UPDATING ('column')</td>
<td>An UPDATE statement that affected the specified column fired the trigger.</td>
</tr>
<tr>
<td>DELETING</td>
<td>A DELETE statement fired the trigger.</td>
</tr>
</tbody>
</table>

A conditional predicate can appear wherever a BOOLEAN expression can appear.

**Example 9-1** creates a DML trigger that uses conditional predicates to determine which of its four possible triggering statements fired it.

**Example 9-1 Trigger Uses Conditional Predicates to Detect Triggering Statement**

```sql
CREATE OR REPLACE TRIGGER t
BEFORE
  INSERT OR
    UPDATE OF salary, department_id OR
    DELETE
  ON employees
BEGIN
  CASE
```
WHEN INSERTING THEN

    DBMS_OUTPUT.PUT_LINE('Inserting');

WHEN UPDATING('salary') THEN

    DBMS_OUTPUT.PUT_LINE('Updating salary');

WHEN UPDATING('department_id') THEN

    DBMS_OUTPUT.PUT_LINE('Updating department ID');

WHEN DELETING THEN

    DBMS_OUTPUT.PUT_LINE('Deleting');

END CASE;

END;
/

Correlation Names and Pseudorecords

Note:
This topic applies only to triggers that fire at row level—that is, row-level simple DML triggers and compound DML triggers with row-level timing point sections.

A trigger that fires at row level can access the data in the row that it is processing by using correlation names. The default correlation names are OLD, NEW, and PARENT. To change the correlation names, use the REFERENCING clause of the CREATETRIGGER statement (see "referencing_clause ::=").

If the trigger is created on a nested table in a view (see "dml_event_clause ::="), then OLD and NEW refer to the current row of the nested table, and PARENT refers to the current row of the parent table. If the trigger is created on a table or view, then OLD and NEW refer to the current row of the table or view, and PARENT is undefined.

OLD, NEW, and PARENT are also called pseudorecords, because they have record structure, but are allowed in fewer contexts than records are. The structure of a pseudorecord is table_name%ROWTYPE, where table_name is the name of the table on which the trigger is created (for OLD and NEW) or the name of the parent table (for PARENT).

In the trigger_body of a simple trigger or the tps_body of a compound trigger, a correlation name is a placeholder for a bind variable. Reference the field of a pseudorecord with this syntax:

:pseudorecord_name.field_name

In the WHEN clause of a conditional trigger, a correlation name is not a placeholder for a bind variable. Therefore, omit the colon in the preceding syntax.
Table 9-1 shows the values of OLD and NEW fields for the row that the triggering statement is processing.

Table 9-1 OLD and NEW Pseudorecord Field Values

<table>
<thead>
<tr>
<th>Triggering Statement</th>
<th>OLD.field Value</th>
<th>NEW.field Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>INSERT</td>
<td>NULL</td>
<td>Post-insert value</td>
</tr>
<tr>
<td>UPDATE</td>
<td>Pre-update value</td>
<td>Post-update value</td>
</tr>
<tr>
<td>DELETE</td>
<td>Pre-delete value</td>
<td>NULL</td>
</tr>
</tbody>
</table>

The restrictions on pseudorecords are:

- A pseudorecord cannot appear in a record-level operation.
  
  For example, the trigger cannot include this statement:

  ```
  :NEW := NULL;
  ```

- A pseudorecord cannot be an actual subprogram parameter.
  
  (A pseudorecord field can be an actual subprogram parameter.)

- The trigger cannot change OLD field values.
  
  Trying to do so raises ORA-04085.

- If the triggering statement is DELETE, then the trigger cannot change NEW field values.
  
  Trying to do so raises ORA-04084.

- An AFTER trigger cannot change NEW field values, because the triggering statement runs before the trigger fires.
  
  Trying to do so raises ORA-04084.

A BEFORE trigger can change NEW field values before a triggering INSERT or UPDATE statement puts them in the table.

If a statement triggers both a BEFORE trigger and an AFTER trigger, and the BEFORE trigger changes a NEW field value, then the AFTER trigger "sees" that change.

Example 9-2 creates a log table and a trigger that inserts a row in the log table after any UPDATE statement affects the SALARY column of the EMPLOYEES table, and then updates EMPLOYEES.SALARY and shows the log table.
Example 9-2 Trigger Logs Changes to EMPLOYEES.SALARY

Create log table:

```sql
DROP TABLE Emp_log;
CREATE TABLE Emp_log (
    Emp_id     NUMBER,
    Log_date   DATE,
    New_salary NUMBER,
    Action     VARCHAR2(20));
```

Create trigger that inserts row in log table after EMPLOYEES.SALARY is updated:

```sql
CREATE OR REPLACE TRIGGER log_salary_increase
AFTER UPDATE OF salary ON employees
FOR EACH ROW
BEGIN
    INSERT INTO Emp_log (Emp_id, Log_date, New_salary, Action)
    VALUES (:NEW.employee_id, SYSDATE, :NEW.salary, 'New Salary');
END;
/
```

Update EMPLOYEES.SALARY:

```sql
UPDATE employees
SET salary = salary + 1000.0
WHERE Department_id = 20;
```

Result:

```
2 rows updated.
```
Show log table:

```
SELECT * FROM Emp_log;
```

Result:

<table>
<thead>
<tr>
<th>EMP_ID</th>
<th>LOG_DATE</th>
<th>NEW_SALARY</th>
<th>ACTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>201</td>
<td>28-APR-10</td>
<td>15049.13</td>
<td>New Salary</td>
</tr>
<tr>
<td>202</td>
<td>28-APR-10</td>
<td>6945.75</td>
<td>New Salary</td>
</tr>
</tbody>
</table>

2 rows selected.

Example 9-3 creates a conditional trigger that prints salary change information whenever a DELETE, INSERT, or UPDATE statement affects the EMPLOYEES table—unless that information is about the President. The database evaluates the WHEN condition for each affected row. If the WHEN condition is TRUE for an affected row, then the trigger fires for that row before the triggering statement runs. If the WHEN condition is not TRUE for an affected row, then trigger does not fire for that row, but the triggering statement still runs.

**Example 9-3 Conditional Trigger Prints Salary Change Information**

```
CREATE OR REPLACE TRIGGER print_salary_changes
    BEFORE DELETE OR INSERT OR UPDATE ON employees
    FOR EACH ROW
    WHEN (NEW.job_id <> 'AD_PRES') -- do not print information about President
DECLARE
    sal_diff NUMBER;
BEGIN
    sal_diff := :NEW.salary - :OLD.salary;
    DBMS_OUTPUT.PUT (:NEW.last_name || ' : ');
    DBMS_OUTPUT.PUT ('Old salary = ' || :OLD.salary || ', ');
```
DBMS_OUTPUT.PUT('New salary = ' || :NEW.salary || ', ');
DBMS_OUTPUT.PUT_LINE('Difference: ' || sal_diff);

END;
/

Query:

```
SELECT last_name, department_id, salary, job_id
FROM employees
WHERE department_id IN (10, 20, 90)
ORDER BY department_id, last_name;
```

Result:

```
LAST_NAME       DEPARTMENT_ID   SALARY  JOB_ID
-------------- --------- -------- ------
Whalen          10          2800    AD_ASST
Fay             20          6000    MK_REP
Hartstein       20          13000   MK_MAN
De Haan         90          17000   AD_VP
King            90          24000   AD_PRES
Kochhar         90          17000   AD_VP

6 rows selected.
```

Triggering statement:

```
UPDATE employees
SET salary = salary * 1.05
WHERE department_id IN (10, 20, 90);
```
Result:

Whalen: Old salary = 2800, New salary = 2940, Difference: 140
Hartstein: Old salary = 13000, New salary = 13650, Difference: 650
Fay: Old salary = 6000, New salary = 6300, Difference: 300
Kochhar: Old salary = 17000, New salary = 17850, Difference: 850
De Haan: Old salary = 17000, New salary = 17850, Difference: 850

6 rows updated.

Query:

SELECT salary FROM employees WHERE job_id = 'AD_PRES';

Result:

<table>
<thead>
<tr>
<th>SALARY</th>
</tr>
</thead>
<tbody>
<tr>
<td>-------</td>
</tr>
<tr>
<td>25200</td>
</tr>
</tbody>
</table>

1 row selected.

Example 9-4 creates a trigger that modifies CLOB columns.

Example 9-4 Trigger Modifies LOB Columns

```sql
DROP TABLE tab1;
CREATE TABLE tab1 (c1 CLOB);
INSERT INTO tab1 VALUES ('<h1>HTML Document Fragment</h1><p>Some text.');
CREATE OR REPLACE TRIGGER trg1
BEFORE UPDATE ON tab1
FOR EACH ROW
```
BEGIN

    DBMS_OUTPUT.PUT_LINE('Old value of CLOB column: '||:OLD.c1);

    DBMS_OUTPUT.PUT_LINE('Proposed new value of CLOB column: '||:NEW.c1);

    :NEW.c1 := :NEW.c1 || TO_CLOB('<hr><p>Standard footer paragraph.'));

    DBMS_OUTPUT.PUT_LINE('Final value of CLOB column: '||:NEW.c1);

END;
/

SET SERVEROUTPUT ON;

UPDATE tab1 SET c1 = '<h1>Different Document Fragment</h1><p>Different text.';

SELECT * FROM tab1;

Example 9-5 creates a table with the same name as a correlation name, new, and then creates a trigger on that table. To avoid conflict between the table name and the correlation name, the trigger references the correlation name as Newest.

Example 9-5 Trigger with REFERENCING Clause

CREATE TABLE new (  
    field1 NUMBER,  
    field2 VARCHAR2(20)
);

CREATE OR REPLACE TRIGGER Print_salary_changes  
BEFORE UPDATE ON new  
REFERENCING new AS Newest  
FOR EACH ROW
BEGIN
  :Newest.Field2 := TO_CHAR (:newest.field1);
END;
/

Show old and new values:

BEGIN
  FOR j IN (SELECT d, old_obj, new_obj FROM tbl_history) LOOP
    DBMS_OUTPUT.PUT_LINE (j.d || ' -- old: ' || j.old_obj.n || ' ' || j.old_obj.m || ' -- new: ' || j.new_obj.n || ' ' || j.new_obj.m);
  END LOOP;
END;
/

Result:

28-APR-10 -- old: 1 0 -- new: 2 0
28-APR-10 -- old: 2 0 -- new: 3 0
28-APR-10 -- old: 3 0 -- new: 4 0
28-APR-10 -- old: 4 0 -- new: 5 0
28-APR-10 -- old: 5 0 -- new: 6 0

All values of column n were increased by 1. The value of m remains 0.

INSTEAD OF DML Triggers

An INSTEAD OF DML trigger is a DML trigger created on a non-editioning view, or on a nested table column of a non-editioning view. The database fires the INSTEAD OF trigger instead of running the triggering DML statement. An INSTEAD OF trigger cannot be conditional.
An INSTEAD OF trigger is the only way to update a view that is not inherently updatable. Design the INSTEAD OF trigger to determine what operation was intended and do the appropriate DML operations on the underlying tables.

An INSTEAD OF trigger is always a row-level trigger. An INSTEAD OF trigger can read OLD and NEW values, but cannot change them.

Example 9-7 creates the view oe.order_info to display information about customers and their orders. The view is not inherently updatable (because the primary key of the orders table, order_id, is not unique in the result set of the join view). The example creates an INSTEAD OF trigger to process INSERT statements directed to the view. The trigger inserts rows into the base tables of the view, customers and orders.

Example 9-7 INSTEAD OF Trigger

```sql
CREATE OR REPLACE VIEW order_info AS
    SELECT c.customer_id, c.cust_last_name, c.cust_first_name,
           o.order_id, o.order_date, o.order_status
    FROM customers c, orders o
    WHERE c.customer_id = o.customer_id;

CREATE OR REPLACE TRIGGER order_info_insert
    INSTEAD OF INSERT ON order_info
    DECLARE
        duplicate_info EXCEPTION;
    PRAGMA EXCEPTION_INIT (duplicate_info, -00001);
    BEGIN
        INSERT INTO customers
                    (customer_id, cust_last_name, cust_first_name)
            VALUES (:new.customer_id,
                        :new.cust_last_name,
                        :new.cust_first_name);
        INSERT INTO orders (order_id, order_date, customer_id)
            VALUES (order_id, order_date, :customer_id);
```
(:new.order_id,
  :new.order_date,
  :new.customer_id);

EXCEPTION

  WHEN duplicate_info THEN

    RAISE_APPLICATION_ERROR ( 
      num=> -20107,
      msg=> 'Duplicate customer or order ID');

END order_info_insert;
/

Query to show that row to be inserted does not exist:

SELECT COUNT(*) FROM order_info WHERE customer_id = 999;

Result:

<table>
<thead>
<tr>
<th>COUNT(*)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
</tbody>
</table>

1 row selected.

Insert row into view:

INSERT INTO order_info VALUES
  (999, 'Smith', 'John', 2500, '13-MAR-2001', 0);

Result:

1 row created.

Query to show that row has been inserted in view:
SELECT COUNT(*) FROM order_info WHERE customer_id = 999;

Result:

COUNT(*)
-------
   1

1 row selected.

Query to show that row has been inserted in customers table:

SELECT COUNT(*) FROM customers WHERE customer_id = 999;

Result:

COUNT(*)
-------
   1

1 row selected.

Query to show that row has been inserted in orders table:

SELECT COUNT(*) FROM orders WHERE customer_id = 999;

Result:

COUNT(*)
-------
   1

1 row selected.