10

PL/SQL Triggers

A trigger is like a stored procedure that Oracle Database invokes automatically whenever a specified event occurs.

> **Note:**
>
> The database can detect only system-defined events. You cannot define your own events.

Topics

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.

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**.

A **crossedition trigger** is a DML trigger for use only in edition-based redefinition.

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**.

A **conditional trigger** is a DML or system trigger that has a `WHEN` clause that specifies a SQL condition that the database evaluates for each row that the triggering statement affects.

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.

> **Note:**
>
> A trigger is often called by the name of its triggering statement (for example, *`DELETE` trigger* or *`LOGON` trigger*), the name of the item on which it is defined (for example, *`DATABASE` trigger* or *`SCHEMA` trigger*), or its timing point (for example, *`BEFORE` statement trigger* or *`AFTER` each row trigger*).

> **See Also:**
>
> - "CREATE TRIGGER Statement" syntax diagram
> - "DML Triggers"
> - "System Triggers"
> - *Oracle Database Development Guide* for information about crossedition triggers
> - "CREATE TRIGGER Statement" for information about the `WHEN` clause

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 Development 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.*)

When a trigger is created on an `INSERT` statement with `FORALL`, the inserts are treated as a single operation. This means that all statement level triggers fire only once, not for each insert. When a trigger is created on an `UPDATE` or `DELETE` statement with `FORALL`, the trigger is executed for each DML statement. This results in better performance for insert operations.

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.

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.

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.

Topics

- Conditional Predicates for Detecting Triggering DML Statement
- INSTEAD OF DML Triggers

- Compound DML Triggers
- Triggers for Ensuring Referential Integrity
- FORALL Statement

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 10-1 Conditional Predicates

Conditional Predicate	TRUE if and only if:
`INSERTING`	An `INSERT` statement fired the trigger.
`UPDATING`	An `UPDATE` statement fired the trigger.
`UPDATING ('column')`	An `UPDATE` statement that affected the specified column fired the trigger.
`DELETING`	A `DELETE` statement fired the trigger.

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

Example 10-1 Trigger Uses Conditional Predicates to Detect Triggering Statement

This example creates a DML trigger that uses conditional predicates to determine which of its four possible triggering statements fired it.

```
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;
/
```

INSTEAD OF DML Triggers

An `INSTEAD OF DML` **trigger** is a DML trigger created on a noneditioning view, or on a nested table column of a noneditioning 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.

An INSTEAD OF trigger with the NESTED TABLE clause fires only if the triggering statement operates on the elements of the specified nested table column of the view. The trigger fires for each modified nested table element.

See Also:

- *Oracle Database SQL Language Reference* for information about inherently updatable views
- "Compound DML Trigger Structure" for information about compound DML triggers with the INSTEAD OF EACH ROW section

Example 10-2 INSTEAD OF Trigger

This example 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.

```
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 (
     :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:

```
  COUNT(*)
----------
         0

1 row selected.
```

Insert row into view:

```
INSERT INTO order_info VALUES
   (999, 'Smith', 'John', 2500, TO_DATE('13-MAR-2001', 'DD-MON-YYYY'), 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.
```

Example 10-3 INSTEAD OF Trigger on Nested Table Column of View

In this example, the view `dept_view` contains a nested table of employees, `emplist`, created by the `CAST` function (described in *Oracle Database SQL Language Reference*). To modify the `emplist` column, the example creates an `INSTEAD OF` trigger on the column.

```
-- Create type of nested table element:

CREATE OR REPLACE TYPE nte
AUTHID DEFINER IS
OBJECT (
  emp_id     NUMBER(6),
  lastname   VARCHAR2(25),
  job        VARCHAR2(10),
  sal        NUMBER(8,2)
);
/

-- Created type of nested table:

CREATE OR REPLACE TYPE emp_list_ IS
  TABLE OF nte;
/

-- Create view:

CREATE OR REPLACE VIEW dept_view AS
  SELECT d.department_id,
         d.department_name,
         CAST (MULTISET (SELECT e.employee_id, e.last_name, e.job_id, e.salary
                         FROM employees e
                         WHERE e.department_id = d.department_id
                        )
                        AS emp_list_
               ) emplist
  FROM departments d;

-- Create trigger:

CREATE OR REPLACE TRIGGER dept_emplist_tr
  INSTEAD OF INSERT ON NESTED TABLE emplist OF dept_view
  REFERENCING NEW AS Employee
              PARENT AS Department
  FOR EACH ROW
BEGIN
  -- Insert on nested table translates to insert on base table:
  INSERT INTO employees (
    employee_id,
    last_name,
    email,
    hire_date,
    job_id,
    salary,
    department_id
  )
  VALUES (
```

```
    :Employee.emp_id,                       -- employee_id
    :Employee.lastname,                     -- last_name
    :Employee.lastname || '@example.com',   -- email
    SYSDATE,                                -- hire_date
    :Employee.job,                          -- job_id
    :Employee.sal,                          -- salary
    :Department.department_id               -- department_id
  );
END;
/
```

Query view before inserting row into nested table:

```
SELECT emplist FROM dept_view WHERE department_id=10;
```

Result:

```
EMPLIST(EMP_ID, LASTNAME, JOB, SAL)
-------------------------------------------

EMP_LIST_(NTE(200, 'Whalen', 'AD_ASST', 4200))

1 row selected.
```

Query table before inserting row into nested table:

```
SELECT employee_id, last_name, job_id, salary
FROM employees
WHERE department_id = 10;
```

Result:

```
EMPLOYEE_ID LAST_NAME                 JOB_ID         SALARY
----------- ------------------------- ---------- ----------
        200 Whalen                    AD_ASST          4200

1 row selected.
```

Insert a row into nested table:

```
INSERT INTO TABLE (
  SELECT d.emplist
  FROM dept_view d
  WHERE department_id = 10
)
VALUES (1001, 'Glenn', 'AC_MGR', 10000);
```

Query view after inserting row into nested table:

```
SELECT emplist FROM dept_view WHERE department_id=10;
```

Result (formatted to fit page):

```
EMPLIST(EMP_ID, LASTNAME, JOB, SAL)
-------------------------------------------------------------------------------

EMP_LIST_(NTE(200, 'Whalen', 'AD_ASST', 4200),
          NTE(1001, 'Glenn', 'AC_MGR', 10000))

1 row selected.
```

Query table after inserting row into nested table:

```
SELECT employee_id, last_name, job_id, salary
FROM employees
WHERE department_id = 10;
```

Result:

```
EMPLOYEE_ID LAST_NAME                 JOB_ID         SALARY
----------- ------------------------- ---------- ----------
        200 Whalen                    AD_ASST          4200
       1001 Glenn                     AC_MGR          10000

2 rows selected.
```

Compound DML Triggers

A compound DML trigger created on a table or editioning view can fire at multiple timing points. Each timing point section has its own executable part and optional exception-handling part, but all of these parts can access a common PL/SQL state. The common state is established when the triggering statement starts and is destroyed when the triggering statement completes, even when the triggering statement causes an error.

A compound DML trigger created on a noneditioning view is not really compound, because it has only one timing point section.

A compound trigger can be conditional, but not autonomous.

Two common uses of compound triggers are:

- To accumulate rows destined for a second table so that you can periodically bulk-insert them
- To avoid the mutating-table error (ORA-04091)

Topics

Compound DML Trigger Structure

The optional declarative part of a compound trigger declares variables and subprograms that all of its timing-point sections can use. When the trigger fires, the declarative part runs before any timing-point sections run. The variables and subprograms exist for the duration of the triggering statement.

A compound DML trigger created on a noneditioning view is not really compound, because it has only one timing point section. The syntax for creating the simplest compound DML trigger on a noneditioning view is:

```
CREATE trigger FOR dml_event_clause ON view
COMPOUND TRIGGER
INSTEAD OF EACH ROW IS BEGIN
```

```
  statement;
END INSTEAD OF EACH ROW;
```

A compound DML trigger created on a table or editioning view has at least one timing-point section in Table 10-2. If the trigger has multiple timing-point sections, they can be in any order, but no timing-point section can be repeated. If a timing-point section is absent, then nothing happens at its timing point.

Table 10-2 Compound Trigger Timing-Point Sections

Timing Point	Section
Before the triggering statement runs	`BEFORE STATEMENT`
After the triggering statement runs	`AFTER STATEMENT`
Before each row that the triggering statement affects	`BEFORE EACH ROW`
After each row that the triggering statement affects	`AFTER EACH ROW`

> **See Also:**
>
> "CREATE TRIGGER Statement" for more information about the syntax of compound triggers

A compound DML trigger does not have an initialization section, but the `BEFORE STATEMENT` section, which runs before any other timing-point section, can do any necessary initialization.

If a compound DML trigger has neither a `BEFORE STATEMENT` section nor an `AFTER STATEMENT` section, and its triggering statement affects no rows, then the trigger never fires.

Compound DML Trigger Restrictions

In addition to the "Trigger Restrictions"), compound DML triggers have these restrictions:

- `OLD`, `NEW`, and `PARENT` cannot appear in the declarative part, the `BEFORE STATEMENT` section, or the `AFTER STATEMENT` section.
- Only the `BEFORE EACH ROW` section can change the value of `NEW`.
- A timing-point section cannot handle exceptions raised in another timing-point section.
- If a timing-point section includes a `GOTO` statement, the target of the `GOTO` statement must be in the same timing-point section.

Performance Benefit of Compound DML Triggers

A compound DML trigger has a performance benefit when the triggering statement affects many rows.

For example, suppose that this statement triggers a compound DML trigger that has all four timing-point sections in Table 10-2:

```
INSERT INTO Target
  SELECT c1, c2, c3
  FROM Source
  WHERE Source.c1 > 0
```

Although the BEFORE EACH ROW and AFTER EACH ROW sections of the trigger run for each row of Source whose column c1 is greater than zero, the BEFORE STATEMENT section runs only before the INSERT statement runs and the AFTER STATEMENT section runs only after the INSERT statement runs.

A compound DML trigger has a greater performance benefit when it uses bulk SQL, described in "Bulk SQL and Bulk Binding".

Using Compound DML Triggers with Bulk Insertion

A compound DML trigger is useful for accumulating rows destined for a second table so that you can periodically bulk-insert them. To get the performance benefit from the compound trigger, you must specify BULK COLLECT INTO in the FORALL statement (otherwise, the FORALL statement does a single-row DML operation multiple times). For more information about using the BULK COLLECT clause with the FORALL statement, see "Using FORALL Statement and BULK COLLECT Clause Together".

> **See Also:**
>
> "FORALL Statement"

Scenario: You want to log every change to hr.employees.salary in a new table, employee_salaries. A single UPDATE statement updates many rows of the table hr.employees; therefore, bulk-inserting rows into employee.salaries is more efficient than inserting them individually.

Solution: Define a compound trigger on updates of the table hr.employees, as in Example 10-4. You do not need a BEFORE STATEMENT section to initialize idx or salaries, because they are state variables, which are initialized each time the trigger fires (even when the triggering statement is interrupted and restarted).

> **Note:**
>
> To run Example 10-4, you must have the EXECUTE privilege on the package DBMS_LOCK.

Example 10-4 Compound Trigger Logs Changes to One Table in Another Table

```
CREATE TABLE employee_salaries (
  employee_id NUMBER NOT NULL,
  change_date DATE   NOT NULL,
  salary NUMBER(8,2) NOT NULL,
  CONSTRAINT pk_employee_salaries PRIMARY KEY (employee_id, change_date),
  CONSTRAINT fk_employee_salaries FOREIGN KEY (employee_id)
    REFERENCES employees (employee_id)
      ON DELETE CASCADE)
/
CREATE OR REPLACE TRIGGER maintain_employee_salaries
  FOR UPDATE OF salary ON employees
    COMPOUND TRIGGER
```

```
-- Declarative Part:
-- Choose small threshhold value to show how example works:
  threshhold CONSTANT SIMPLE_INTEGER := 7;

  TYPE salaries_t IS TABLE OF employee_salaries%ROWTYPE INDEX BY
SIMPLE_INTEGER;
  salaries  salaries_t;
  idx       SIMPLE_INTEGER := 0;

  PROCEDURE flush_array IS
    n CONSTANT SIMPLE_INTEGER := salaries.count();
  BEGIN
    FORALL j IN 1..n
      INSERT INTO employee_salaries VALUES salaries(j);
    salaries.delete();
    idx := 0;
    DBMS_OUTPUT.PUT_LINE('Flushed ' || n || ' rows');
  END flush_array;

  -- AFTER EACH ROW Section:

  AFTER EACH ROW IS
  BEGIN
    idx := idx + 1;
    salaries(idx).employee_id := :NEW.employee_id;
    salaries(idx).change_date := SYSTIMESTAMP;
    salaries(idx).salary := :NEW.salary;
    IF idx >= threshhold THEN
      flush_array();
    END IF;
  END AFTER EACH ROW;

  -- AFTER STATEMENT Section:

  AFTER STATEMENT IS
  BEGIN
    flush_array();
  END AFTER STATEMENT;
END maintain_employee_salaries;
/
```

Increase salary of every employee in department 50 by 10%:

```
UPDATE employees
  SET salary = salary * 1.1
  WHERE department_id = 50
/
```

Result:

```
Flushed 7 rows
Flushed 7 rows
Flushed 7 rows
Flushed 7 rows
```

```
Flushed 7 rows
Flushed 7 rows
Flushed 3 rows

45 rows updated.
```

Wait two seconds:

```
BEGIN
  DBMS_SESSION.SLEEP(2);
END;
/
```

Increase salary of every employee in department 50 by 5%:

```
UPDATE employees
  SET salary = salary * 1.05
  WHERE department_id = 50
/
```

Result:

```
Flushed 7 rows
Flushed 7 rows
Flushed 7 rows
Flushed 7 rows
Flushed 7 rows
Flushed 7 rows
Flushed 3 rows

45 rows updated.
```

See changes to employees table reflected in `employee_salaries` table:

```
SELECT employee_id, count(*) c
  FROM employee_salaries
  GROUP BY employee_id
/
```

Result:

```
EMPLOYEE_ID          C
----------- ----------
        120          2
        121          2
        122          2
        123          2
        124          2
        125          2
...
        199          2
```

```
45 rows selected.
```

Using Compound DML Triggers to Avoid Mutating-Table Error

A compound DML trigger is useful for avoiding the mutating-table error (ORA-04091) explained in "Mutating-Table Restriction".

Scenario: A business rule states that an employee's salary increase must not exceed 10% of the average salary for the employee's department. This rule must be enforced by a trigger.

Solution: Define a compound trigger on updates of the table `hr.employees`, as in Example 10-5. The state variables are initialized each time the trigger fires (even when the triggering statement is interrupted and restarted).

Example 10-5 Compound Trigger Avoids Mutating-Table Error

```
CREATE OR REPLACE TRIGGER Check_Employee_Salary_Raise
  FOR UPDATE OF Salary ON Employees
COMPOUND TRIGGER
  Ten_Percent                 CONSTANT NUMBER := 0.1;
  TYPE Salaries_t             IS TABLE OF Employees.Salary%TYPE;
  Avg_Salaries                Salaries_t;
  TYPE Department_IDs_t       IS TABLE OF Employees.Department_ID%TYPE;
  Department_IDs              Department_IDs_t;

  -- Declare collection type and variable:

  TYPE Department_Salaries_t  IS TABLE OF Employees.Salary%TYPE
                                INDEX BY VARCHAR2(80);
  Department_Avg_Salaries     Department_Salaries_t;

  BEFORE STATEMENT IS
  BEGIN
    SELECT               AVG(e.Salary), NVL(e.Department_ID, -1)
      BULK COLLECT INTO  Avg_Salaries, Department_IDs
      FROM               Employees e
      GROUP BY           e.Department_ID;
    FOR j IN 1..Department_IDs.COUNT() LOOP
      Department_Avg_Salaries(Department_IDs(j)) := Avg_Salaries(j);
    END LOOP;
  END BEFORE STATEMENT;

  AFTER EACH ROW IS
  BEGIN
    IF :NEW.Salary - :Old.Salary >
      Ten_Percent*Department_Avg_Salaries(:NEW.Department_ID)
    THEN
      Raise_Application_Error(-20000, 'Raise too big');
    END IF;
  END AFTER EACH ROW;
END Check_Employee_Salary_Raise;
```

Triggers for Ensuring Referential Integrity

You can use triggers and constraints to maintain referential integrity between parent and child tables, as Table 10-3 shows. (For more information about constraints, see *Oracle Database SQL Language Reference*.)

Table 10-3 Constraints and Triggers for Ensuring Referential Integrity

Table	Constraint to Declare on Table	Triggers to Create on Table
Parent	`PRIMARY KEY` or `UNIQUE`	One or more triggers that ensure that when `PRIMARY KEY` or `UNIQUE` values are updated or deleted, the desired action (`RESTRICT`, `CASCADE`, or `SET NULL`) occurs on corresponding `FOREIGN KEY` values. No action is required for inserts into the parent table, because no dependent foreign keys exist.
Child	`FOREIGN KEY`, if parent and child are in the same database. (The database does not support declarative referential constraints between tables on different nodes of a distributed database.) Disable this foreign key constraint to prevent the corresponding `PRIMARY KEY` or `UNIQUE` constraint from being dropped (except explicitly with the `CASCADE` option).	One trigger that ensures that values inserted or updated in the `FOREIGN KEY` correspond to `PRIMARY KEY` or `UNIQUE` values in the parent table.

Topics

- Foreign Key Trigger for Child Table
- UPDATE and DELETE RESTRICT Trigger for Parent Table
- UPDATE and DELETE SET NULL Trigger for Parent Table
- DELETE CASCADE Trigger for Parent Table
- UPDATE CASCADE Trigger for Parent Table
- Triggers for Complex Constraint Checking
- Triggers for Complex Security Authorizations
- Triggers for Transparent Event Logging
- Triggers for Deriving Column Values
- Triggers for Building Complex Updatable Views
- Triggers for Fine-Grained Access Control

Note:

The examples in the following topics use these tables, which share the column `Deptno`:

```
CREATE TABLE emp (
  Empno     NUMBER NOT NULL,
  Ename     VARCHAR2(10),
  Job       VARCHAR2(9),
  Mgr       NUMBER(4),
  Hiredate  DATE,
  Sal       NUMBER(7,2),
  Comm      NUMBER(7,2),
  Deptno    NUMBER(2) NOT NULL);

CREATE TABLE dept (
  Deptno    NUMBER(2) NOT NULL,
  Dname     VARCHAR2(14),
  Loc       VARCHAR2(13),
  Mgr_no    NUMBER,
  Dept_type NUMBER);
```

Several triggers include statements that lock rows (`SELECT FOR UPDATE`). This operation is necessary to maintain concurrency while the rows are being processed.

These examples are not meant to be used exactly as written. They are provided to assist you in designing your own triggers.

Foreign Key Trigger for Child Table

The trigger in Example 10-6 ensures that before an `INSERT` or `UPDATE` statement affects a foreign key value, the corresponding value exists in the parent key. The exception ORA-04091 (mutating-table error) allows the trigger `emp_dept_check` to be used with the `UPDATE_SET_DEFAULT` and `UPDATE_CASCADE` triggers. This exception is unnecessary if the trigger `emp_dept_check` is used alone.

Example 10-6 Foreign Key Trigger for Child Table

```
CREATE OR REPLACE TRIGGER emp_dept_check
  BEFORE INSERT OR UPDATE OF Deptno ON emp
  FOR EACH ROW WHEN (NEW.Deptno IS NOT NULL)

  -- Before row is inserted or DEPTNO is updated in emp table,
  -- fire this trigger to verify that new foreign key value (DEPTNO)
  -- is present in dept table.
DECLARE
  Dummy               INTEGER;  -- Use for cursor fetch
  Invalid_department  EXCEPTION;
  Valid_department    EXCEPTION;
  Mutating_table      EXCEPTION;
  PRAGMA EXCEPTION_INIT (Invalid_department, -4093);
  PRAGMA EXCEPTION_INIT (Valid_department, -4092);
  PRAGMA EXCEPTION_INIT (Mutating_table, -4091);

  -- Cursor used to verify parent key value exists.
```

```
  -- If present, lock parent key's row so it cannot be deleted
  -- by another transaction until this transaction is
  -- committed or rolled back.

  CURSOR Dummy_cursor (Dn NUMBER) IS
    SELECT Deptno FROM dept
    WHERE Deptno = Dn
    FOR UPDATE OF Deptno;
BEGIN
  OPEN Dummy_cursor (:NEW.Deptno);
  FETCH Dummy_cursor INTO Dummy;

  -- Verify parent key.
  -- If not found, raise user-specified error code and message.
  -- If found, close cursor before allowing triggering statement to complete:

  IF Dummy_cursor%NOTFOUND THEN
    RAISE Invalid_department;
  ELSE
    RAISE Valid_department;
  END IF;
  CLOSE Dummy_cursor;
EXCEPTION
  WHEN Invalid_department THEN
    CLOSE Dummy_cursor;
    Raise_application_error(-20000, 'Invalid Department'
      || ' Number' || TO_CHAR(:NEW.deptno));
  WHEN Valid_department THEN
    CLOSE Dummy_cursor;
  WHEN Mutating_table THEN
    NULL;
END;
/
```

UPDATE and DELETE RESTRICT Trigger for Parent Table

The trigger in Example 10-7 enforces the `UPDATE` and `DELETE RESTRICT` referential action on the primary key of the `dept` table.

> **Caution:**
>
> The trigger in Example 10-7 does not work with self-referential tables (tables with both the primary/unique key and the foreign key). Also, this trigger does not allow triggers to cycle (such as when A fires B, which fires A).

Example 10-7 UPDATE and DELETE RESTRICT Trigger for Parent Table

```
CREATE OR REPLACE TRIGGER dept_restrict
  BEFORE DELETE OR UPDATE OF Deptno ON dept
  FOR EACH ROW

  -- Before row is deleted from dept or primary key (DEPTNO) of dept is updated,
  -- check for dependent foreign key values in emp;
  -- if any are found, roll back.

DECLARE
  Dummy                 INTEGER;  -- Use for cursor fetch
  employees_present     EXCEPTION;
```

```
  employees_not_present  EXCEPTION;
  PRAGMA EXCEPTION_INIT (employees_present, -4094);
  PRAGMA EXCEPTION_INIT (employees_not_present, -4095);

  -- Cursor used to check for dependent foreign key values.
  CURSOR Dummy_cursor (Dn NUMBER) IS
    SELECT Deptno FROM emp WHERE Deptno = Dn;

BEGIN
  OPEN Dummy_cursor (:OLD.Deptno);
  FETCH Dummy_cursor INTO Dummy;

  -- If dependent foreign key is found, raise user-specified
  -- error code and message. If not found, close cursor
  -- before allowing triggering statement to complete.

  IF Dummy_cursor%FOUND THEN
    RAISE employees_present;     -- Dependent rows exist
  ELSE
    RAISE employees_not_present; -- No dependent rows exist
  END IF;
  CLOSE Dummy_cursor;

EXCEPTION
  WHEN employees_present THEN
    CLOSE Dummy_cursor;
    Raise_application_error(-20001, 'Employees Present in'
      || ' Department ' || TO_CHAR(:OLD.DEPTNO));
  WHEN employees_not_present THEN
    CLOSE Dummy_cursor;
END;
```

UPDATE and DELETE SET NULL Trigger for Parent Table

The trigger in Example 10-8 enforces the `UPDATE` and `DELETE SET NULL` referential action on the primary key of the `dept` table.

Example 10-8 UPDATE and DELETE SET NULL Trigger for Parent Table

```
CREATE OR REPLACE TRIGGER dept_set_null
  AFTER DELETE OR UPDATE OF Deptno ON dept
  FOR EACH ROW

  -- Before row is deleted from dept or primary key (DEPTNO) of dept is updated,
  -- set all corresponding dependent foreign key values in emp to NULL:

BEGIN
  IF UPDATING AND :OLD.Deptno != :NEW.Deptno OR DELETING THEN
    UPDATE emp SET emp.Deptno = NULL
    WHERE emp.Deptno = :OLD.Deptno;
  END IF;
END;
/
```

DELETE CASCADE Trigger for Parent Table

The trigger in Example 10-9 enforces the `DELETE CASCADE` referential action on the primary key of the `dept` table.

> **Note:**
>
> Typically, the code for `DELETE CASCADE` is combined with the code for `UPDATE SET NULL` or `UPDATE SET DEFAULT`, to account for both updates and deletes.

Example 10-9 DELETE CASCADE Trigger for Parent Table

```
CREATE OR REPLACE TRIGGER dept_del_cascade
  AFTER DELETE ON dept
  FOR EACH ROW

  -- Before row is deleted from dept,
  -- delete all rows from emp table whose DEPTNO is same as
  -- DEPTNO being deleted from dept table:

BEGIN
  DELETE FROM emp
  WHERE emp.Deptno = :OLD.Deptno;
END;
/
```

UPDATE CASCADE Trigger for Parent Table

The triggers in Example 10-10 ensure that if a department number is updated in the `dept` table, then this change is propagated to dependent foreign keys in the `emp` table.

> **Note:**
>
> Because the trigger `dept_cascade2` updates the `emp` table, the `emp_dept_check` trigger in Example 10-6, if enabled, also fires. The resulting mutating-table error is trapped by the `emp_dept_check` trigger. Carefully test any triggers that require error trapping to succeed to ensure that they always work properly in your environment.

Example 10-10 UPDATE CASCADE Trigger for Parent Table

```
-- Generate sequence number to be used as flag
-- for determining if update occurred on column:

CREATE SEQUENCE Update_sequence
  INCREMENT BY 1 MAXVALUE 5000 CYCLE;

CREATE OR REPLACE PACKAGE Integritypackage AUTHID DEFINER AS
  Updateseq NUMBER;
END Integritypackage;
/
CREATE OR REPLACE PACKAGE BODY Integritypackage AS
END Integritypackage;
/
-- Create flag col:

ALTER TABLE emp ADD Update_id NUMBER;

CREATE OR REPLACE TRIGGER dept_cascade1
  BEFORE UPDATE OF Deptno ON dept
```

```
DECLARE
  -- Before updating dept table (this is a statement trigger),
  -- generate sequence number
  -- & assign it to public variable UPDATESEQ of
  -- user-defined package named INTEGRITYPACKAGE:
BEGIN
  Integritypackage.Updateseq := Update_sequence.NEXTVAL;
END;
/
CREATE OR REPLACE TRIGGER dept_cascade2
  AFTER DELETE OR UPDATE OF Deptno ON dept
  FOR EACH ROW

  -- For each department number in dept that is updated,
  -- cascade update to dependent foreign keys in emp table.
  -- Cascade update only if child row was not updated by this trigger:
BEGIN
  IF UPDATING THEN
    UPDATE emp
    SET Deptno = :NEW.Deptno,
        Update_id = Integritypackage.Updateseq   --from 1st
    WHERE emp.Deptno = :OLD.Deptno
    AND Update_id IS NULL;

    /* Only NULL if not updated by 3rd trigger
       fired by same triggering statement */
  END IF;
  IF DELETING THEN
    -- After row is deleted from dept,
    -- delete all rows from emp table whose DEPTNO is same as
    -- DEPTNO being deleted from dept table:
    DELETE FROM emp
    WHERE emp.Deptno = :OLD.Deptno;
  END IF;
END;
/
CREATE OR REPLACE TRIGGER dept_cascade3
  AFTER UPDATE OF Deptno ON dept
BEGIN UPDATE emp
  SET Update_id = NULL
  WHERE Update_id = Integritypackage.Updateseq;
END;
/
```

Triggers for Complex Constraint Checking

Triggers can enforce integrity rules other than referential integrity. The trigger in Example 10-11 does a complex check before allowing the triggering statement to run.

> **Note:**
>
> Example 10-11 needs this data structure:
>
> ```
> CREATE TABLE Salgrade (
>   Grade               NUMBER,
>   Losal               NUMBER,
>   Hisal               NUMBER,
>   Job_classification  VARCHAR2(9));
> ```

Example 10-11 Trigger Checks Complex Constraints

```
CREATE OR REPLACE TRIGGER salary_check
  BEFORE INSERT OR UPDATE OF Sal, Job ON Emp
  FOR EACH ROW

DECLARE
  Minsal               NUMBER;
  Maxsal               NUMBER;
  Salary_out_of_range  EXCEPTION;
  PRAGMA EXCEPTION_INIT (Salary_out_of_range, -4096);

BEGIN
  /* Retrieve minimum & maximum salary for employee's new job classification
     from SALGRADE table into MINSAL and MAXSAL: */

  SELECT Losal, Hisal INTO Minsal, Maxsal
  FROM Salgrade
  WHERE Job_classification = :NEW.Job;

  /* If employee's new salary is less than or greater than
     job classification's limits, raise exception.
     Exception message is returned and pending INSERT or UPDATE statement
     that fired the trigger is rolled back: */

  IF (:NEW.Sal < Minsal OR :NEW.Sal > Maxsal) THEN
    RAISE Salary_out_of_range;
  END IF;
EXCEPTION
  WHEN Salary_out_of_range THEN
    Raise_application_error (
      -20300,
      'Salary '|| TO_CHAR(:NEW.Sal) ||' out of range for '
      || 'job classification ' ||:NEW.Job
      ||' for employee ' || :NEW.Ename
    );
  WHEN NO_DATA_FOUND THEN
    Raise_application_error(-20322, 'Invalid Job Classification');
END;
/
```

Triggers for Complex Security Authorizations

Triggers are commonly used to enforce complex security authorizations for table data. Use triggers only to enforce complex security authorizations that you cannot define using the

database security features provided with the database. For example, use a trigger to prohibit updates to the `employee` table during weekends and nonworking hours.

When using a trigger to enforce a complex security authorization, it is best to use a `BEFORE` statement trigger. Using a `BEFORE` statement trigger has these benefits:

- The security check is done before the triggering statement is allowed to run, so that no wasted work is done by an unauthorized statement.
- The security check is done only for the triggering statement, not for each row affected by the triggering statement.

The trigger in Example 10-12 enforces security by raising exceptions when anyone tries to update the table `employees` during weekends or nonworking hours.

> **See Also:**
>
> *Oracle Database Security Guide* for detailed information about database security features

Example 10-12 Trigger Enforces Security Authorizations

```
CREATE OR REPLACE TRIGGER Employee_permit_changes
  BEFORE INSERT OR DELETE OR UPDATE ON employees
DECLARE
  Dummy             INTEGER;
  Not_on_weekends   EXCEPTION;
  Nonworking_hours  EXCEPTION;
  PRAGMA EXCEPTION_INIT (Not_on_weekends, -4097);
  PRAGMA EXCEPTION_INIT (Nonworking_hours, -4099);
BEGIN
   -- Check for weekends:

   IF (TO_CHAR(Sysdate, 'DAY') = 'SAT' OR
     TO_CHAR(Sysdate, 'DAY') = 'SUN') THEN
       RAISE Not_on_weekends;
   END IF;

  -- Check for work hours (8am to 6pm):

  IF (TO_CHAR(Sysdate, 'HH24') < 8 OR
    TO_CHAR(Sysdate, 'HH24') > 18) THEN
      RAISE Nonworking_hours;
  END IF;

EXCEPTION
  WHEN Not_on_weekends THEN
    Raise_application_error(-20324,'Might not change '
      ||'employee table during the weekend');
  WHEN Nonworking_hours THEN
    Raise_application_error(-20326,'Might not change '
     ||'emp table during Nonworking hours');
END;
/
```

Triggers for Transparent Event Logging

Triggers are very useful when you want to transparently do a related change in the database following certain events.

The `REORDER` trigger example shows a trigger that reorders parts as necessary when certain conditions are met. (In other words, a triggering statement is entered, and the `PARTS_ON_HAND` value is less than the `REORDER_POINT` value.)

Triggers for Deriving Column Values

Triggers can derive column values automatically, based upon a value provided by an `INSERT` or `UPDATE` statement. This type of trigger is useful to force values in specific columns that depend on the values of other columns in the same row. `BEFORE` row triggers are necessary to complete this type of operation for these reasons:

- The dependent values must be derived before the `INSERT` or `UPDATE` occurs, so that the triggering statement can use the derived values.
- The trigger must fire for each row affected by the triggering `INSERT` or `UPDATE` statement.

The trigger in Example 10-13 derives new column values for a table whenever a row is inserted or updated.

> **Note:**
>
> Example 10-13 needs this change to this data structure:
>
> ```
> ALTER TABLE Emp ADD(
>    Uppername   VARCHAR2(20),
>    Soundexname VARCHAR2(20));
> ```

Example 10-13 Trigger Derives New Column Values

```
CREATE OR REPLACE TRIGGER Derived
BEFORE INSERT OR UPDATE OF Ename ON Emp

/* Before updating the ENAME field, derive the values for
   the UPPERNAME and SOUNDEXNAME fields. Restrict users
   from updating these fields directly: */
FOR EACH ROW
BEGIN
  :NEW.Uppername := UPPER(:NEW.Ename);
  :NEW.Soundexname := SOUNDEX(:NEW.Ename);
END;
/
```

Triggers for Building Complex Updatable Views

Views are an excellent mechanism to provide logical windows over table data. However, when the view query gets complex, the system implicitly cannot translate the DML on the view into those on the underlying tables. `INSTEAD OF` triggers help solve this problem. These triggers can be defined over views, and they fire instead of the actual DML.

Consider a library system where books are arranged by title. The library consists of a collection of book type objects:

```
CREATE OR REPLACE TYPE Book_t AS OBJECT (
  Booknum    NUMBER,
  Title      VARCHAR2(20),
  Author     VARCHAR2(20),
  Available  CHAR(1)
);
/
CREATE OR REPLACE TYPE Book_list_t AS TABLE OF Book_t;
/
```

The table `Book_table` is created and populated like this:

```
DROP TABLE Book_table;
CREATE TABLE Book_table (
  Booknum    NUMBER,
  Section    VARCHAR2(20),
  Title      VARCHAR2(20),
  Author     VARCHAR2(20),
  Available  CHAR(1)
);

INSERT INTO Book_table (
  Booknum, Section, Title, Author, Available
)
VALUES (
  121001, 'Classic', 'Iliad', 'Homer', 'Y'
);

INSERT INTO Book_table (
  Booknum, Section, Title, Author, Available
)
VALUES (
  121002, 'Novel', 'Gone with the Wind', 'Mitchell M', 'N'
);

SELECT * FROM Book_table ORDER BY Booknum;
```

Result:

```
   BOOKNUM SECTION              TITLE                AUTHOR               A
---------- -------------------- -------------------- -------------------- -
    121001 Classic              Iliad                Homer                Y
    121002 Novel                Gone with the Wind   Mitchell M           N

2 rows selected.
```

The table `Library_table` is created and populated like this:

```
DROP TABLE Library_table;
CREATE TABLE Library_table (Section VARCHAR2(20));

INSERT INTO Library_table (Section)
VALUES ('Novel');

INSERT INTO Library_table (Section)
VALUES ('Classic');

SELECT * FROM Library_table ORDER BY Section;
```

Result:

```
SECTION
--------------------
Classic
Novel

2 rows selected.
```

You can define a complex view over the tables `Book_table` and `Library_table` to create a logical view of the library with sections and a collection of books in each section:

```
CREATE OR REPLACE VIEW Library_view AS
  SELECT i.Section, CAST (
    MULTISET (
      SELECT b.Booknum, b.Title, b.Author, b.Available
      FROM Book_table b
      WHERE b.Section = i.Section
    ) AS Book_list_t
  ) BOOKLIST
  FROM Library_table i;
```

(For information about the `CAST` function, see *Oracle Database SQL Language Reference*.)

Make `Library_view` updatable by defining an `INSTEAD OF` trigger on it:

```
CREATE OR REPLACE TRIGGER Library_trigger
  INSTEAD OF
  INSERT ON Library_view
  FOR EACH ROW
DECLARE
  Bookvar  Book_t;
  i        INTEGER;
BEGIN
  INSERT INTO Library_table
  VALUES (:NEW.Section);

  FOR i IN 1..:NEW.Booklist.COUNT LOOP
    Bookvar := :NEW.Booklist(i);

    INSERT INTO Book_table (
      Booknum, Section, Title, Author, Available
    )
    VALUES (
      Bookvar.booknum, :NEW.Section, Bookvar.Title,
      Bookvar.Author, bookvar.Available
    );
  END LOOP;
END;
/
```

Insert a new row into `Library_view`:

```
INSERT INTO Library_view (Section, Booklist)
VALUES (
  'History',
  book_list_t (book_t (121330, 'Alexander', 'Mirth', 'Y'))
);
```

See the effect on `Library_view`:

```
SELECT * FROM Library_view ORDER BY Section;
```

Result:

```
SECTION
--------------------
BOOKLIST(BOOKNUM, TITLE, AUTHOR, AVAILABLE)
--------------------------------------------------------------------

Classic
BOOK_LIST_T(BOOK_T(121001, 'Iliad', 'Homer', 'Y'))

History
BOOK_LIST_T(BOOK_T(121330, 'Alexander', 'Mirth', 'Y'))

Novel
BOOK_LIST_T(BOOK_T(121002, 'Gone with the Wind', 'Mitchell M', 'N'))


3 rows selected.
```

See the effect on `Book_table`:

```
SELECT * FROM Book_table ORDER BY Booknum;
```

Result:

```
   BOOKNUM SECTION              TITLE                AUTHOR               A
---------- -------------------- -------------------- -------------------- -
    121001 Classic              Iliad                Homer                Y
    121002 Novel                Gone with the Wind   Mitchell M           N
    121330 History              Alexander            Mirth                Y

3 rows selected.
```

See the effect on `Library_table`:

```
SELECT * FROM Library_table ORDER BY Section;
```

Result:

```
SECTION
--------------------
Classic
History
Novel

3 rows selected.
```

Similarly, you can also define triggers on the nested table `booklist` to handle modification of the nested table element.

Triggers for Fine-Grained Access Control

You can use `LOGON` triggers to run the package associated with an application context. An application context captures session-related information about the user who is logging in to the database. From there, your application can control how much access this user has, based on their session information.

> **Note:**
>
> If you have very specific logon requirements, such as preventing users from logging in from outside the firewall or after work hours, consider using Oracle Database Vault instead of `LOGON` triggers. With Oracle Database Vault, you can create custom rules to strictly control user access.

> **See Also:**
>
> - *Oracle Database Security Guide* for information about creating a `LOGON` trigger to run a database session application context package
> - *Oracle Database Vault Administrator's Guide* for information about Oracle Database Vault

Correlation Names and Pseudorecords

> **Note:**
>
> This topic applies only to triggers that fire at row level. That is:
>
> - Row-level simple DML triggers
> - 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 `CREATE TRIGGER` statement (see "*referencing_clause* ::=").

If the trigger is created on a nested table, 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 10-4 shows the values of `OLD` and `NEW` fields for the row that the triggering statement is processing.

Table 10-4 OLD and NEW Pseudorecord Field Values

Triggering Statement	OLD.*field* Value	NEW.*field* Value
`INSERT`	`NULL`	Post-insert value
`UPDATE`	Pre-update value	Post-update value
`DELETE`	Pre-delete value	`NULL`

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 10-14 Trigger Logs Changes to EMPLOYEES.SALARY

This example 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.

Create log table:

```
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:

```
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:

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

Result:

```
2 rows updated.
```

Show log table:

```
SELECT * FROM Emp_log;
```

Result:

```
    EMP_ID LOG_DATE  NEW_SALARY ACTION
---------- --------- ---------- --------------------
       201 28-APR-10      13650 New Salary
       202 28-APR-10       6300 New Salary

2 rows selected.
```

Example 10-15 Conditional Trigger Prints Salary Change Information

This example 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.

```
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       4200 AD_ASST
Davis                                20       6000 MK_REP
Martinez                             20      13000 MK_MAN
Garcia                               90      17000 AD_VP
King                                 90      24000 AD_PRES
Yang                                 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 = 4200, New salary = 4410, Difference: 210
Martinez: Old salary = 13000, New salary = 13650, Difference: 650
Davis: Old salary = 6000, New salary = 6300, Difference: 300
Yang: Old salary = 17000, New salary = 17850, Difference: 850
Garcia: Old salary = 17000, New salary = 17850, Difference: 850

6 rows updated.
```

Query:

```
SELECT salary FROM employees WHERE job_id = 'AD_PRES';
```

Result:

```
    SALARY
----------
     25200

1 row selected.
```

Example 10-16 Trigger Modifies CLOB Columns

This example creates an `UPDATE` trigger that modifies `CLOB` columns.

For information about `TO_CLOB` and other conversion functions, see *Oracle Database SQL Language Reference*.

```
DROP TABLE tab1;
CREATE TABLE tab1 (c1 CLOB);
INSERT INTO tab1 VALUES ('<h1>HTML Document Fragment</h1><p>Some text.', 3);

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 10-17 Trigger with REFERENCING Clause

This example 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`.

```
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;
/
```

OBJECT_VALUE Pseudocolumn

A DML trigger on an object table can reference the SQL pseudocolumn `OBJECT_VALUE`, which returns system-generated names for the columns of the object table. The trigger can also invoke a PL/SQL subprogram that has a formal `IN` parameter whose data type is `OBJECT_VALUE`.

> **See Also:**
>
> - *Oracle Database SQL Language Reference* for more information about `OBJECT_VALUE`
> - *Oracle Database SQL Language Reference* for general information about pseudocolumns

Example 10-18 creates object table `tbl`, table `tbl_history` for logging updates to `tbl`, and trigger `Tbl_Trg`. The trigger runs for each row of `tbl` that is affected by a DML statement, causing the old and new values of the object `t` in `tbl` to be written in `tbl_history`. The old and new values are `:OLD.OBJECT_VALUE` and `:NEW.OBJECT_VALUE`.

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

Example 10-18 Trigger References OBJECT_VALUE Pseudocolumn

Create, populate, and show object table:

```
CREATE OR REPLACE TYPE t AUTHID DEFINER AS OBJECT (n NUMBER, m NUMBER)
/
CREATE TABLE tbl OF t
/
BEGIN
  FOR j IN 1..5 LOOP
    INSERT INTO tbl VALUES (t(j, 0));
  END LOOP;
END;
/
SELECT * FROM tbl ORDER BY n;
```

Result:

```
         N          M
---------- ----------
         1          0
         2          0
         3          0
         4          0
         5          0

5 rows selected.
```

Create history table and trigger:

```
CREATE TABLE tbl_history ( d DATE, old_obj t, new_obj t)
/
```

```
CREATE OR REPLACE TRIGGER Tbl_Trg
  AFTER UPDATE ON tbl
  FOR EACH ROW
BEGIN
  INSERT INTO tbl_history (d, old_obj, new_obj)
  VALUES (SYSDATE, :OLD.OBJECT_VALUE, :NEW.OBJECT_VALUE);
END Tbl_Trg;
/
```

Update object table:

```
UPDATE tbl SET tbl.n = tbl.n+1
/
```

Result:

```
5 rows updated.
```

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
```

System Triggers

A **system trigger** is created on either a schema or the database.

Its triggering event is composed of either DDL statements (listed in "*ddl_event*") or database operation statements (listed in "*database_event*").

A system 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 a *AFTER statement trigger* or *statement-level AFTER trigger.*)
- Instead of the triggering `CREATE` statement

  (The trigger is called an *INSTEAD OF CREATE trigger*.)

Topics

SCHEMA Triggers

A **`SCHEMA` trigger** is created on a schema and fires whenever the user who owns it is the current user and initiates the triggering event.

Suppose that both user1 and user2 own schema triggers, and user1 invokes a DR unit owned by user2. Inside the DR unit, user2 is the current user. Therefore, if the DR unit initiates the triggering event of a schema trigger that user2 owns, then that trigger fires. However, if the DR unit initiates the triggering event of a schema trigger that user1 owns, then that trigger does not fire.

Example 10-19 creates a `BEFORE` statement trigger on the sample schema `HR`. When a user connected as `HR` tries to drop a database object, the database fires the trigger before dropping the object.

Example 10-19 BEFORE Statement Trigger on Sample Schema HR

```
CREATE OR REPLACE TRIGGER drop_trigger
  BEFORE DROP ON hr.SCHEMA
  BEGIN
    RAISE_APPLICATION_ERROR (
      num => -20000,
      msg => 'Cannot drop object');
  END;
/
```

DATABASE Triggers

A **`DATABASE` trigger** is created on the database and fires whenever any database user initiates the triggering event.

Example 10-20 shows the basic syntax for a trigger to log errors. This trigger fires after an unsuccessful statement execution, such as unsuccessful logon.

> **Note:**
>
> An `AFTER SERVERERROR` trigger fires only if Oracle relational database management system (RDBMS) determines that it is safe to fire error triggers. For more information about `AFTER SERVERERROR` triggers, see CREATE TRIGGER Statement.

The trigger in Example 10-21 runs the procedure `check_user` after a user logs onto the database.

Example 10-20 AFTER Statement Trigger on Database

```
CREATE TRIGGER log_errors
  AFTER SERVERERROR ON DATABASE
  BEGIN
    IF (IS_SERVERERROR (1017)) THEN
      NULL;  -- (substitute code that processes logon error)
    ELSE
```

```
      NULL;  -- (substitute code that logs error code)
    END IF;
  END;
/
```

Example 10-21 Trigger Monitors Logons

```
CREATE OR REPLACE TRIGGER check_user
  AFTER LOGON ON DATABASE
  BEGIN
    check_user;
  EXCEPTION
    WHEN OTHERS THEN
      RAISE_APPLICATION_ERROR
        (-20000, 'Unexpected error: '|| DBMS_Utility.Format_Error_Stack);
 END;
/
```

INSTEAD OF CREATE Triggers

An `INSTEAD OF CREATE` trigger is a `SCHEMA` trigger whose triggering event is a `CREATE` statement. The database fires the trigger instead of executing its triggering statement.

Example 10-22 shows the basic syntax for an `INSTEAD OF CREATE` trigger on the current schema. This trigger fires when the owner of the current schema issues a `CREATE` statement in the current schema.

Example 10-22 INSTEAD OF CREATE Trigger on Schema

```
CREATE OR REPLACE TRIGGER t
  INSTEAD OF CREATE ON SCHEMA
  BEGIN
    EXECUTE IMMEDIATE 'CREATE TABLE T (n NUMBER, m NUMBER)';
  END;
/
```

Subprograms Invoked by Triggers

Triggers can invoke subprograms written in PL/SQL, C, and Java. The trigger in Example 10-4 invokes a PL/SQL subprogram. The trigger in Example 10-23 invokes a Java subprogram.

A subprogram invoked by a trigger cannot run transaction control statements, because the subprogram runs in the context of the trigger body.

If a trigger invokes an invoker rights (IR) subprogram, then the user who created the trigger, not the user who ran the triggering statement, is considered to be the current user. For information about IR subprograms, see "Invoker's Rights and Definer's Rights (AUTHID Property)".

If a trigger invokes a remote subprogram, and a time stamp or signature mismatch is found during execution of the trigger, then the remote subprogram does not run and the trigger is invalidated.

Example 10-23 Trigger Invokes Java Subprogram

```
CREATE OR REPLACE PROCEDURE Before_delete (Id IN NUMBER, Ename VARCHAR2)
IS LANGUAGE Java
name 'thjvTriggers.beforeDelete (oracle.jdbc.NUMBER, oracle.jdbc.CHAR)';

CREATE OR REPLACE TRIGGER Pre_del_trigger BEFORE DELETE ON Tab
```

```
FOR EACH ROW
CALL Before_delete (:OLD.Id, :OLD.Ename)
/
```

The corresponding Java file is `thjvTriggers.java`:

```
import java.sql.*
import java.io.*
import oracle.jdbc.*
import oracle.oracore.*
public class thjvTriggers
{
public static void
beforeDelete (NUMBER old_id, CHAR old_name)
Throws SQLException, CoreException
   {
   Connection conn = JDBCConnection.defaultConnection();
   Statement stmt = conn.CreateStatement();
   String sql = "insert into logtab values
   ("+ old_id.intValue() +", '"+ old_ename.toString() + ", BEFORE DELETE');
   stmt.executeUpdate (sql);
   stmt.close();
   return;
   }
}
```

Trigger Compilation, Invalidation, and Recompilation

The `CREATE TRIGGER` statement compiles the trigger and stores its code in the database. If a compilation error occurs, the trigger is still created, but its triggering statement fails, except in these cases:

- The trigger was created in the disabled state.
- The triggering event is `AFTER STARTUP ON DATABASE`.
- The triggering event is either `AFTER LOGON ON DATABASE` or `AFTER LOGON ON SCHEMA`, and someone logs on as `SYSTEM`.

To see trigger compilation errors, either use the `SHOW ERRORS` command in SQL*Plus or Enterprise Manager, or query the static data dictionary view `*_ERRORS` (described in *Oracle Database Reference*).

If a trigger does not compile successfully, then its exception handler cannot run. For an example, see "Remote Exception Handling".

If a trigger references another object, such as a subprogram or package, and that object is modified or dropped, then the trigger becomes invalid. The next time the triggering event occurs, the compiler tries to revalidate the trigger (for details, see *Oracle Database Development Guide*).

> **Note:**
>
> Because the `DBMS_AQ` package is used to enqueue a message, dependency between triggers and queues cannot be maintained.

To recompile a trigger manually, use the `ALTER TRIGGER` statement, described in "ALTER TRIGGER Statement".

Exception Handling in Triggers

In most cases, if a trigger runs a statement that raises an exception, and the exception is not handled by an exception handler, then the database rolls back the effects of both the trigger and its triggering statement.

In the following cases, the database rolls back only the effects of the trigger, not the effects of the triggering statement (and logs the error in trace files and the alert log):

- The triggering event is either `AFTER STARTUP ON DATABASE` or `BEFORE SHUTDOWN ON DATABASE`.
- The triggering event is `AFTER LOGON ON DATABASE` and the user has the `ADMINISTER DATABASE TRIGGER` privilege.
- The triggering event is `AFTER LOGON ON SCHEMA` and the user either owns the schema or has the `ALTER ANY TRIGGER` privilege.

In the case of a compound DML trigger, the database rolls back only the effects of the triggering statement, not the effects of the trigger. However, variables declared in the trigger are re-initialized, and any values computed before the triggering statement was rolled back are lost.

> **Note:**
>
> Triggers that enforce complex security authorizations or constraints typically raise user-defined exceptions, which are explained in "User-Defined Exceptions".

> **See Also:**
>
> PL/SQL Error Handling, for general information about exception handling

Remote Exception Handling

A trigger that accesses a remote database can do remote exception handling only if the remote database is available. If the remote database is unavailable when the local database must compile the trigger, then the local database cannot validate the statement that accesses the remote database, and the compilation fails. If the trigger cannot be compiled, then its exception handler cannot run.

The trigger in Example 10-24 has an `INSERT` statement that accesses a remote database. The trigger also has an exception handler. However, if the remote database is unavailable when the local database tries to compile the trigger, then the compilation fails and the exception handler cannot run.

Example 10-25 shows the workaround for the problem in Example 10-24: Put the remote `INSERT` statement and exception handler in a stored subprogram and have the trigger invoke the stored subprogram. The subprogram is stored in the local database in compiled form, with a validated statement for accessing the remote database. Therefore, when the remote `INSERT` statement fails because the remote database is unavailable, the exception handler in the subprogram can handle it.

Example 10-24 Trigger Cannot Handle Exception if Remote Database is Unavailable

```
CREATE OR REPLACE TRIGGER employees_tr
  AFTER INSERT ON employees
  FOR EACH ROW
BEGIN
  -- When remote database is unavailable, compilation fails here:
  INSERT INTO employees@remote (
    employee_id, first_name, last_name, email, hire_date, job_id
  )
  VALUES (
    99, 'Jane', 'Doe', 'jane.doe@example.com', SYSDATE, 'ST_MAN'
  );
EXCEPTION
  WHEN OTHERS THEN
    INSERT INTO emp_log (Emp_id, Log_date, New_salary, Action)
      VALUES (99, SYSDATE, NULL, 'Could not insert');
    RAISE;
END;
/
```

Example 10-25 Workaround for Example 10-24

```
CREATE OR REPLACE PROCEDURE insert_row_proc AUTHID CURRENT_USER AS
  no_remote_db EXCEPTION;  -- declare exception
  PRAGMA EXCEPTION_INIT (no_remote_db, -20000);
                           -- assign error code to exception
BEGIN
  INSERT INTO employees@remote (
    employee_id, first_name, last_name, email, hire_date, job_id
  )
  VALUES (
    99, 'Jane', 'Doe', 'jane.doe@example.com', SYSDATE, 'ST_MAN'
  );
EXCEPTION
  WHEN OTHERS THEN
    INSERT INTO emp_log (Emp_id, Log_date, New_salary, Action)
      VALUES (99, SYSDATE, NULL, 'Could not insert row.');

  RAISE_APPLICATION_ERROR (-20000, 'Remote database is unavailable.');
END;
/

CREATE OR REPLACE TRIGGER employees_tr
  AFTER INSERT ON employees
  FOR EACH ROW
BEGIN
  insert_row_proc;
END;
/
```

Trigger Design Guidelines

- Use triggers to ensure that whenever a specific event occurs, any necessary actions are done (regardless of which user or application issues the triggering statement).

  For example, use a trigger to ensure that whenever anyone updates a table, its log file is updated.

- Do not create triggers that duplicate database features.

For example, do not create a trigger to reject invalid data if you can do the same with constraints (see "How Triggers and Constraints Differ").

- Do not create triggers that depend on the order in which a SQL statement processes rows (which can vary).

  For example, do not assign a value to a global package variable in a row trigger if the current value of the variable depends on the row being processed by the row trigger. If a trigger updates global package variables, initialize those variables in a `BEFORE` statement trigger.

- Use `BEFORE` row triggers to modify the row before writing the row data to disk.
- Use `AFTER` row triggers to obtain the row ID and use it in operations.

  An `AFTER` row trigger fires when the triggering statement results in ORA-02292.

> **Note:**
>
> `AFTER` row triggers are slightly more efficient than `BEFORE` row triggers. With `BEFORE` row triggers, affected data blocks are read first for the trigger and then for the triggering statement. With `AFTER` row triggers, affected data blocks are read only for the trigger.

- If the triggering statement of a `BEFORE` row trigger is an `UPDATE` or `DELETE` statement that conflicts with an `UPDATE` statement that is running, then the database does a transparent `ROLLBACK` to `SAVEPOINT` and restarts the triggering statement. The database can do this many times before the triggering statement completes successfully. Each time the database restarts the triggering statement, the trigger fires. The `ROLLBACK` to `SAVEPOINT` does not undo changes to package variables that the trigger references. To ensure that there are no unwanted side effects with each restart, make sure that the `BEFORE` row trigger is idempotent, meaning the trigger should be written so that the result remains the same with each subsequent execution. Any additional work that should not be repeated can be handled in an `AFTER` row trigger. To detect this situation, you can also include a counter variable in the package.
- Do not create recursive triggers.

  For example, do not create an `AFTER UPDATE` trigger that issues an `UPDATE` statement on the table on which the trigger is defined. The trigger fires recursively until it runs out of memory.

- If you create a trigger that includes a statement that accesses a remote database, then put the exception handler for that statement in a stored subprogram and invoke the subprogram from the trigger.

  For more information, see "Remote Exception Handling".

- Use `DATABASE` triggers judiciously. They fire every time any database user initiates a triggering event.
- If a trigger runs the following statement, the statement returns the owner of the trigger, not the user who is updating the table:

  ```
  SELECT Username FROM USER_USERS;
  ```

- Only committed triggers fire.

  A trigger is committed, implicitly, after the `CREATE TRIGGER` statement that creates it succeeds. Therefore, the following statement cannot fire the trigger that it creates:

```
CREATE OR REPLACE TRIGGER my_trigger
  AFTER CREATE ON DATABASE
BEGIN
  NULL;
END;
/
```

- To allow the modular installation of applications that have triggers on the same tables, create multiple triggers of the same type, rather than a single trigger that runs a sequence of operations.

  Each trigger sees the changes made by the previously fired triggers. Each trigger can see `OLD` and `NEW` values.

Trigger Restrictions

In addition to the restrictions that apply to all PL/SQL units (see Table C-1), triggers have these restrictions:

- Trigger Size Restriction
- Trigger LONG and LONG RAW Data Type Restrictions
- Mutating-Table Restriction
- Only an autonomous trigger can run TCL or DDL statements.

  For information about autonomous triggers, see "Autonomous Triggers".
- A trigger cannot invoke a subprogram that runs transaction control statements, because the subprogram runs in the context of the trigger body.

  For more information about subprograms invoked by triggers, see "Subprograms Invoked by Triggers".
- A trigger cannot access a `SERIALLY_REUSABLE` package.

  For information about `SERIALLY_REUSABLE` packages, see "SERIALLY_REUSABLE Packages".

> **See Also:**
>
> "Compound DML Trigger Restrictions"

Trigger Size Restriction

The size of the trigger cannot exceed 32K.

If the logic for your trigger requires much more than 60 lines of PL/SQL source text, then put most of the source text in a stored subprogram and invoke the subprogram from the trigger. For information about subprograms invoked by triggers, see "Subprograms Invoked by Triggers".

Trigger LONG and LONG RAW Data Type Restrictions

> **Note:**
>
> Oracle supports the `LONG` and `LONG RAW` data types only for backward compatibility with existing applications.
>
> For information about how to migrate columns from `LONG` data types to `LOB` data types, see *Oracle Database SecureFiles and Large Objects Developer's Guide*.

In addition to the restrictions that apply to all PL/SQL units (see "LONG and LONG RAW Variables"), triggers have these restrictions:

- A trigger cannot declare a variable of the `LONG` or `LONG RAW` data type.
- A SQL statement in a trigger can reference a `LONG` or `LONG RAW` column only if the column data can be converted to the data type `CHAR` or `VARCHAR2`.
- A trigger cannot use the correlation name `NEW` or `PARENT` with a `LONG` or `LONG RAW` column.

Mutating-Table Restriction

> **Note:**
>
> This topic applies only to row-level simple DML triggers.

A **mutating table** is a table that is being modified by a DML statement (possibly by the effects of a `DELETE CASCADE` constraint). (A view being modified by an `INSTEAD OF` trigger is not considered to be mutating.)

The mutating-table restriction prevents the trigger from querying or modifying the table that the triggering statement is modifying. When a row-level trigger encounters a mutating table, ORA-04091 occurs, the effects of the trigger and triggering statement are rolled back, and control returns to the user or application that issued the triggering statement, as Example 10-26 shows.

> **Caution:**
>
> Oracle Database does not enforce the mutating-table restriction for a trigger that accesses remote nodes, because the database does not support declarative referential constraints between tables on different nodes of a distributed database.
>
> Similarly, the database does not enforce the mutating-table restriction for tables in the same database that are connected by loop-back database links. A loop-back database link makes a local table appear remote by defining an Oracle Net path back to the database that contains the link.

If you must use a trigger to update a mutating table, you can avoid the mutating-table error in either of these ways:

- Use a compound DML trigger (see "Using Compound DML Triggers to Avoid Mutating-Table Error").
- Use a temporary table.

  For example, instead of using one `AFTER` each row trigger that updates the mutating table, use two triggers—an `AFTER` each row trigger that updates the temporary table and an `AFTER` statement trigger that updates the mutating table with the values from the temporary table.

Mutating-Table Restriction Relaxed

As of Oracle Database 8*g* Release 1, a deletion from the parent table causes `BEFORE` and `AFTER` triggers to fire once. Therefore, you can create row-level and statement-level triggers that query and modify the parent and child tables. This allows most foreign key constraint actions to be implemented through their after-row triggers (unless the constraint is self-referential). Update cascade, update set null, update set default, delete set default, inserting a missing parent, and maintaining a count of children can all be implemented easily—see "Triggers for Ensuring Referential Integrity".

However, cascades require care for multiple-row foreign key updates. The trigger cannot miss rows that were changed but not committed by another transaction, because the foreign key constraint guarantees that no matching foreign key rows are locked before the after-row trigger is invoked.

In Example 10-27, the triggering statement updates `p` correctly but causes problems when the trigger updates `f`. First, the triggering statement changes (1) to (2) in `p`, and the trigger updates (1) to (2) in `f`, leaving two rows of value (2) in `f`. Next, the triggering statement updates (2) to (3) in `p`, and the trigger updates both rows of value (2) to (3) in `f`. Finally, the statement updates (3) to (4) in `p`, and the trigger updates all three rows in f from (3) to (4). The relationship between the data items in `p` and `f` is lost.

To avoid this problem, either forbid multiple-row updates to `p` that change the primary key and reuse existing primary key values, or track updates to foreign key values and modify the trigger to ensure that no row is updated twice.

Example 10-26 Trigger Causes Mutating-Table Error

```
-- Create log table

DROP TABLE log;
CREATE TABLE log (
  emp_id  NUMBER(6),
  l_name  VARCHAR2(25),
  f_name  VARCHAR2(20)
);

-- Create trigger that updates log and then reads employees

CREATE OR REPLACE TRIGGER log_deletions
  AFTER DELETE ON employees
  FOR EACH ROW
DECLARE
  n INTEGER;
BEGIN
  INSERT INTO log VALUES (
    :OLD.employee_id,
    :OLD.last_name,
```

```
    :OLD.first_name
  );

  SELECT COUNT(*) INTO n FROM employees;
  DBMS_OUTPUT.PUT_LINE('There are now ' || n || ' employees.');
END;
/

-- Issue triggering statement:

DELETE FROM employees WHERE employee_id = 197;
```

Result:

```
DELETE FROM employees WHERE employee_id = 197
            *
ERROR at line 1:
ORA-04091: table HR.EMPLOYEES is mutating, trigger/function might not see it
ORA-06512: at "HR.LOG_DELETIONS", line 10
ORA-04088: error during execution of trigger 'HR.LOG_DELETIONS'
```

Show that effect of trigger was rolled back:

```
SELECT count(*) FROM log;
```

Result:

```
  COUNT(*)
----------
         0

1 row selected.
```

Show that effect of triggering statement was rolled back:

```
SELECT employee_id, last_name FROM employees WHERE employee_id = 197;
```

Result:

```
EMPLOYEE_ID LAST_NAME
----------- -------------------------
        197 Feeney

1 row selected.
```

Example 10-27 Update Cascade

```
DROP TABLE p;
CREATE TABLE p (p1 NUMBER CONSTRAINT pk_p_p1 PRIMARY KEY);
INSERT INTO p VALUES (1);
INSERT INTO p VALUES (2);
INSERT INTO p VALUES (3);

DROP TABLE f;
CREATE TABLE f (f1 NUMBER CONSTRAINT fk_f_f1 REFERENCES p);
INSERT INTO f VALUES (1);
INSERT INTO f VALUES (2);
INSERT INTO f VALUES (3);

CREATE TRIGGER pt
  AFTER UPDATE ON p
  FOR EACH ROW
BEGIN
```

```
  UPDATE f SET f1 = :NEW.p1 WHERE f1 = :OLD.p1;
END;
/
```

Query:

```
SELECT * FROM p ORDER BY p1;
```

Result:

```
        P1
----------
         1
         2
         3
```

Query:

```
SELECT * FROM f ORDER BY f1;
```

Result:

```
        F1
----------
         1
         2
         3
```

Issue triggering statement:

```
UPDATE p SET p1 = p1+1;
```

Query:

```
SELECT * FROM p ORDER BY p1;
```

Result:

```
        P1
----------
         2
         3
         4
```

Query:

```
SELECT * FROM f ORDER BY f1;
```

Result:

```
        F1
----------
         4
         4
         4
```

Order in Which Triggers Fire

If two or more triggers *with different timing points* are defined for the same statement on the same table, then they fire in this order:

1. All `BEFORE STATEMENT` triggers
2. All `BEFORE EACH ROW` triggers
3. All `AFTER EACH ROW` triggers
4. All `AFTER STATEMENT` triggers

If it is practical, replace the set of individual triggers with different timing points with a single compound trigger that explicitly codes the actions in the order you intend. For information about compound triggers, see "Compound DML Triggers".

If you are creating two or more triggers *with the same timing point*, and the order in which they fire is important, then you can control their firing order using the `FOLLOWS` and `PRECEDES` clauses (see "FOLLOWS | PRECEDES").

If multiple compound triggers are created on a table, then:

- All `BEFORE STATEMENT` sections run at the `BEFORE STATEMENT` timing point, `BEFORE EACH ROW` sections run at the `BEFORE EACH ROW` timing point, and so forth.

  If trigger execution order was specified using the `FOLLOWS` clause, then the `FOLLOWS` clause determines the order of execution of compound trigger sections. If `FOLLOWS` is specified for some but not all triggers, then the order of execution of triggers is guaranteed only for those that are related using the `FOLLOWS` clause.

- All `AFTER STATEMENT` sections run at the `AFTER STATEMENT` timing point, `AFTER EACH ROW` sections run at the `AFTER EACH ROW` timing point, and so forth.

  If trigger execution order was specified using the `PRECEDES` clause, then the `PRECEDES` clause determines the order of execution of compound trigger sections. If `PRECEDES` is specified for some but not all triggers, then the order of execution of triggers is guaranteed only for those that are related using the `PRECEDES` clause.

> **Note:**
>
> `PRECEDES` applies only to reverse crossedition triggers, which are described in *Oracle Database Development Guide*.

The firing of compound triggers can be interleaved with the firing of simple triggers.

When one trigger causes another trigger to fire, the triggers are said to be **cascading**. The database allows up to 32 triggers to cascade simultaneously. To limit the number of trigger cascades, use the initialization parameter `OPEN_CURSORS` (described in *Oracle Database Reference*), because a cursor opens every time a trigger fires.

Trigger Enabling and Disabling

By default, the `CREATE TRIGGER` statement creates a trigger in the enabled state. To create a trigger in the disabled state, specify `DISABLE`. Creating a trigger in the disabled state lets you ensure that it compiles without errors before you enable it.

Some reasons to temporarily disable a trigger are:

- The trigger refers to an unavailable object.
- You must do a large data load, and you want it to proceed quickly without firing triggers.
- You are reloading data.

To enable or disable a single trigger, use this statement:

```
ALTER TRIGGER [schema.]trigger_name { ENABLE | DISABLE };
```

To enable or disable all triggers in all editions created on a specific table, use this statement:

```
ALTER TABLE table_name { ENABLE | DISABLE } ALL TRIGGERS;
```

In both of the preceding statements, *`schema`* is the name of the schema containing the trigger, and the default is your schema.

> **See Also:**
>
> - "ALTER TRIGGER Statement" for more information about the `ALTER TRIGGER` statement
> - *Oracle Database SQL Language Reference* for more information about the `ALTER TABLE` statement

Trigger Changing and Debugging

To change a trigger, you must either replace or re-create it. (The `ALTER TRIGGER` statement only enables, disables, compiles, or renames a trigger.)

To replace a trigger, use the `CREATE TRIGGER` statement with the `OR REPLACE` clause.

To re-create a trigger, first drop it with the `DROP TRIGGER` statement and then create it again with the `CREATE TRIGGER` statement.

To debug a trigger, you can use the facilities available for stored subprograms. For information about these facilities, see *Oracle Database Development Guide*.

> **See Also:**
>
> - "CREATE TRIGGER Statement" for more information about the `CREATE TRIGGER` statement
> - "DROP TRIGGER Statement" for more information about the `DROP TRIGGER` statement
> - "ALTER TRIGGER Statement" for more information about the `ALTER TRIGGER` statement

Triggers and Oracle Database Data Transfer Utilities

The Oracle database utilities that transfer data to your database, possibly firing triggers, are:

- **SQL*Loader (`sqlldr`)**

  SQL*Loader loads data from external files into tables of an Oracle database.

  During a SQL*Loader conventional load, `INSERT` triggers fire.

  Before a SQL*Loader direct load, triggers are disabled.

  > **See Also:**
  >
  > *Oracle Database Utilities* for more information about SQL*Loader

- **Data Pump Import (`impdp`)**

  Data Pump Import (`impdp`) reads an export dump file set created by Data Pump Export (`expdp`) and writes it to an Oracle database.

  If a table to be imported does not exist on the target database, or if you specify `TABLE_EXISTS_ACTION=REPLACE`, then `impdp` creates and loads the table before creating any triggers, so no triggers fire.

  If a table to be imported exists on the target database, and you specify either `TABLE_EXISTS_ACTION=APPEND` or `TABLE_EXISTS_ACTION=TRUNCATE`, then `impdp` loads rows into the existing table, and `INSERT` triggers created on the table fire.

  > **See Also:**
  >
  > *Oracle Database Utilities* for more information about Data Pump Import

- **Original Import (`imp`)**

  Original Import (the original Import utility, `imp`) reads object definitions and table data from dump files created by original Export (the original Export utility, `exp`) and writes them to the target database.

> **Note:**
>
> To import files that original Export created, you must use original Import. In all other cases, Oracle recommends that you use Data Pump Import instead of original Import.

If a table to be imported does not exist on the target database, then `imp` creates and loads the table before creating any triggers, so no triggers fire.

If a table to be imported exists on the target database, then the Import `IGNORE` parameter determines whether triggers fire during import operations. The `IGNORE` parameter specifies whether object creation errors are ignored or not, resulting in the following behavior:

- If `IGNORE=n` (default), then `imp` does not change the table and no triggers fire.
- If `IGNORE=y`, then `imp` loads rows into the existing table, and `INSERT` triggers created on the table fire.

> **See Also:**
>
> - *Oracle Database Utilities* for more information about the original Import utility
> - *Oracle Database Utilities* for more information about the original Export utility
> - *Oracle Database Utilities* for more information about `IGNORE`

Triggers for Publishing Events

To use a trigger to publish an event, create a trigger that:

- Has the event as its triggering event
- Invokes the appropriate subprograms in the `DBMS_AQ` package, which provides an interface to Oracle Advanced Queuing (AQ)

  For information about the `DBMS_AQ` package, see *Oracle Database PL/SQL Packages and Types Reference*.

  For information about AQ, see *Oracle Database Advanced Queuing User's Guide*.

By enabling and disabling such triggers, you can turn event notification on and off. For information about enabling and disabling triggers, see "Trigger Enabling and Disabling".

How Triggers Publish Events

When the database detects an event, it fires all enabled triggers that are defined on that event, except:

- Any trigger that is the target of the triggering event.

  For example, a trigger for all `DROP` events does not fire when it is dropped itself.
- Any trigger that was modified, but not committed, in the same transaction as the triggering event.

  For example, if a recursive DDL statement in a system trigger modifies another trigger, then events in the same transaction cannot fire the modified trigger.

When a trigger fires and invokes AQ, AQ publishes the event and passes to the trigger the publication context and specified attributes. The trigger can access the attributes by invoking event attribute functions.

The attributes that a trigger can specify to AQ (by passing them to AQ as `IN` parameters) and then access with event attribute functions depends on the triggering event, which is either a database event or a client event.

> **Note:**
>
> - A trigger always behaves like a definer rights (DR) unit. The trigger action of an event runs as the definer of the action (as the definer of the package or function in callouts, or as owner of the trigger in queues). Because the owner of the trigger must have `EXECUTE` privileges on the underlying queues, packages, or subprograms, this action is consistent. For information about DR units, see "Invoker's Rights and Definer's Rights (AUTHID Property)".
> - The database ignores the return status from callback functions for all events. For example, the database does nothing with the return status from a `SHUTDOWN` event.

Topics

Event Attribute Functions

By invoking system-defined event attribute functions in Table 10-5, a trigger can retrieve certain attributes of the triggering event. Not all triggers can invoke all event attribute functions —for details, see "Event Attribute Functions for Database Event Triggers" and "Event Attribute Functions for Client Event Triggers".

> **Note:**
>
> - In earlier releases, you had to access these functions through the `SYS` package. Now Oracle recommends accessing them with their public synonyms (the names starting with `ora_` in the first column of Table 10-5).
> - The function parameter `ora_name_list_t` is defined in package `DBMS_STANDARD` as:
>
> ```
> TYPE ora_name_list_t IS TABLE OF VARCHAR2(2*(ORA_MAX_NAME_LEN+2)+1);
> ```

Table 10-5 System-Defined Event Attributes

Attribute	Return Type and Value	Example

`ora_client_ip_address`	`VARCHAR2`: IP address of client in `LOGON` event when underlying protocol is TCP/IP	`DECLARE`
		`  v_addr VARCHAR2(11);`
		`BEGIN`
		`  IF (ora_sysevent = 'LOGON') THEN`
		`  v_addr := ora_client_ip_address;`
		`  END IF;`
		`END;`
		`/`

`ora_database_name`	`VARCHAR2(50)`: Database name	`DECLARE`
		`  v_db_name VARCHAR2(50);`
		`BEGIN`
		`  v_db_name := ora_database_name;`
		`END;`
		`/`

`ora_des_encrypted_password`	`VARCHAR2`: DES-encrypted password of user being created or altered	`IF (ora_dict_obj_type = 'USER') THEN`
		`  INSERT INTO event_table`
		`  VALUES (ora_des_encrypted_password);`
		`END IF;`

`ora_dict_obj_name`	`VARCHAR2(128)`: Name of dictionary object on which DDL operation occurred	`INSERT INTO event_table`
		`VALUES ('Changed object is ' ||`
		`  ora_dict_obj_name);`

`ora_dict_obj_name_list ( name_list OUT ora_name_list_t )`	`PLS_INTEGER`: Number of object names modified in event
	`OUT` parameter: List of object names modified in event
		`DECLARE`
		`  name_list ora_name_list_t;`
		`  number_modified PLS_INTEGER;`
		`BEGIN`
		`  IF (ora_sysevent='ASSOCIATE STATISTICS') THEN`
		`  number_modified :=`
		`  ora_dict_obj_name_list(name_list);`
		`  END IF;`
		`END;`

`ora_dict_obj_owner`	`VARCHAR2(128)`: Owner of dictionary object on which DDL operation occurred	`INSERT INTO event_table`
		`VALUES ('object owner is' ||`
		`  ora_dict_obj_owner);`

`ora_dict_obj_owner_list ( owner_list OUT ora_name_list_t )`	`PLS_INTEGER`: Number of owners of objects modified in event
	`OUT` parameter: List of owners of objects modified in event
		`DECLARE`
		`  owner_list ora_name_list_t;`
		`  number_modified PLS_INTEGER;`
		`BEGIN`
		`  IF (ora_sysevent='ASSOCIATE STATISTICS') THEN`
		`  number_modified :=`
		`  ora_dict_obj_name_list(owner_list);`
		`  END IF;`
		`END;`

Table 10-5 (Cont.) System-Defined Event Attributes

Attribute	Return Type and Value	Example

`ora_dict_obj_type`	`VARCHAR2(20)`: Type of dictionary object on which DDL operation occurred	`INSERT INTO event_table`
		`VALUES ('This object is a ' ||`
		`  **ora_dict_obj_type**);`

`ora_grantee (`
`user_list OUT ora_name_list_t`
`)`
	`PLS_INTEGER`: Number of grantees in grant event
	`OUT` parameter: List of grantees in grant event
		`DECLARE`
		`  user_list ora_name_list_t;`
		`  number_of_grantees PLS_INTEGER;`
		`BEGIN`
		`  IF (ora_sysevent = 'GRANT') THEN`
		`  number_of_grantees :=`
		`  **ora_grantee(user_list);**`
		`  END IF;`
		`END;`

`ora_instance_num`	`NUMBER`: Instance number	`IF (**ora_instance_num** = 1) THEN`
		`  INSERT INTO event_table VALUES ('1');`
		`END IF;`

`ora_is_alter_column (`
`column_name IN VARCHAR2`
`)`
	`BOOLEAN`: `TRUE` if specified column is altered, `FALSE` otherwise	`IF (ora_sysevent = 'ALTER' AND`
		`  ora_dict_obj_type = 'TABLE') THEN`
		`  alter_column := **ora_is_alter_column('C');**`
		`END IF;`

`ora_is_creating_nested_table`	`BOOLEAN`: `TRUE` if current event is creating nested table, `FALSE` otherwise	`IF (ora_sysevent = 'CREATE' AND`
		`  ora_dict_obj_type = 'TABLE' AND`
		`  **ora_is_creating_nested_table**) THEN`
		`  INSERT INTO event_table`
		`  VALUES ('A nested table is created');`
		`END IF;`

`ora_is_drop_column (`
`column_name IN VARCHAR2`
`)`
	`BOOLEAN`: `TRUE` if specified column is dropped, `FALSE` otherwise	`IF (ora_sysevent = 'ALTER' AND`
		`  ora_dict_obj_type = 'TABLE') THEN`
		`  drop_column := **ora_is_drop_column('C');**`
		`END IF;`

`ora_is_servererror (`
`error_number IN VARCHAR2`
`)`
	`BOOLEAN`: `TRUE` if given error is on error stack, `FALSE` otherwise	`IF **ora_is_servererror(error_number)** THEN`
		`  INSERT INTO event_table`
		`  VALUES ('Server error!!');`
		`END IF;`

`ora_login_user`	`VARCHAR2(128)`: Login user name	`SELECT **ora_login_user** FROM DUAL;`

`ora_partition_pos`	`PLS_INTEGER`: In `INSTEAD OF` trigger for `CREATE TABLE`, position in SQL text where you can insert `PARTITION` clause	`-- Retrieve ora_sql_txt into  sql_text variable`
		`v_n := **ora_partition_pos;**`
		`v_new_stmt := SUBSTR(sql_text,1,v_n - 1)`
		`  || ' ' || my_partition_clause`
		`  || ' ' || SUBSTR(sql_text, v_n));`

Table 10-5 (Cont.) System-Defined Event Attributes

Attribute	Return Type and Value	Example

`ora_privilege_list ( privilege_list OUT ora_name_list_t )`	`PLS_INTEGER`: Number of privileges in grant or revoke event
	`OUT` parameter: List of privileges granted or revoked in event
		`DECLARE`
		`  privilege_list ora_name_list_t;`
		`  number_of_privileges PLS_INTEGER;`
		`BEGIN`
		`  IF (ora_sysevent = 'GRANT' OR`
		`  ora_sysevent = 'REVOKE') THEN`
		`  number_of_privileges :=`
		`  ora_privilege_list(privilege_list);`
		`  END IF;`
		`END;`

`ora_revokee ( user_list OUT ora_name_list_t )`	`PLS_INTEGER`: Number of revokees in revoke event
	`OUT` parameter: List of revokees in event
		`DECLARE`
		`  user_list ora_name_list_t;`
		`  number_of_users PLS_INTEGER;`
		`BEGIN`
		`  IF (ora_sysevent = 'REVOKE') THEN`
		`  number_of_users := ora_revokee(user_list);`
		`  END IF;`
		`END;`

`ora_server_error ( position IN PLS_INTEGER )`	`NUMBER`: Error code at given position on error stack[1]	`INSERT INTO event_table`
		`VALUES ('top stack error ' ||`
		`  ora_server_error(1));`

`ora_server_error_depth`	`PLS_INTEGER`: Number of error messages on error stack	`n := ora_server_error_depth;` `-- Use n with functions such as ora_server_error`

`ora_server_error_msg ( position IN PLS_INTEGER )`	`VARCHAR2`: Error message at given position on error stack[1]	`INSERT INTO event_table`
		`VALUES ('top stack error message' ||`
		`  ora_server_error_msg(1));`

`ora_server_error_num_params ( position IN PLS_INTEGER )`	`PLS_INTEGER`: Number of strings substituted into error message (using format like `%s`) at given position on error stack[1]	`n := ora_server_error_num_params(1);`
`ora_server_error_param ( position IN PLS_INTEGER, param IN PLS_INTEGER )`	`VARCHAR2`: Matching substitution value (`%s`, `%d`, and so on) in error message at given position and parameter number[1]	`-- Second %s in "Expected %s, found %s":` `param := ora_server_error_param(1,2);`

Table 10-5 (Cont.) System-Defined Event Attributes

Attribute	Return Type and Value	Example

`ora_sql_txt ( sql_text OUT ora_name_list_t )`	`PLS_INTEGER`: Number of elements in PL/SQL table
	`OUT` parameter: SQL text of triggering statement (broken into multiple collection elements if statement is long)
		`CREATE TABLE event_table (col VARCHAR2(2030));`
		
		`DECLARE`
		`  sql_text ora_name_list_t;`
		`  n PLS_INTEGER;`
		`  v_stmt VARCHAR2(2000);`
		`BEGIN`
		`  n := ora_sql_txt(sql_text);`
		
		`  FOR i IN 1..n LOOP`
		`  v_stmt := v_stmt || sql_text(i);`
		`  END LOOP;`
		
		`  INSERT INTO event_table VALUES ('text of`
		`  triggering statement: ' || v_stmt);`
		`END;`

`ora_sysevent`	`VARCHAR2(20)`: Name of triggering event, as given in syntax	`INSERT INTO event_table` `VALUES (ora_sysevent);`

`ora_with_grant_option`	`BOOLEAN`: `TRUE` if privileges are granted with `GRANT` option, `FALSE` otherwise	`IF (ora_sysevent = 'GRANT' AND`
		`  ora_with_grant_option = TRUE) THEN`
		`  INSERT INTO event_table`
		`  VALUES ('with grant option');`
		`END IF;`

`ora_space_error_info ( error_number OUT NUMBER, error_type OUT VARCHAR2, object_owner OUT VARCHAR2, table_space_name OUT VARCHAR2, object_name OUT VARCHAR2, sub_object_name OUT VARCHAR2 )`	`BOOLEAN`: `TRUE` if error is related to out-of-space condition, `FALSE` otherwise
	`OUT` parameters: Information about object that caused error
		`IF (ora_space_error_info (`
		`  eno,typ,owner,ts,obj,subobj) = TRUE) THEN`
		`  DBMS_OUTPUT.PUT_LINE('The object '|| obj`
		`  || ' owned by ' || owner ||`
		`  ' has run out of space.');`
		`END IF;`

1 Position 1 is the top of the stack.

Event Attribute Functions for Database Event Triggers

Table 10-6 summarizes the database event triggers that can invoke event attribute functions. For more information about the triggering events in Table 10-6, see "*database_event*".

Table 10-6 Database Event Triggers

Triggering Event	When Trigger Fires	WHEN Conditions	Restrictions	Transaction	Attribute Functions

`AFTER STARTUP`	When database is opened.	None allowed	Trigger cannot do database operations.	Starts a separate transaction and commits it after firing the triggers.	`ora_sysevent`
					`ora_login_user`
					`ora_instance_num`
					`ora_database_name`

`BEFORE SHUTDOWN`	Just before server starts shutdown of an instance.
	This lets the cartridge shutdown completely. For nonstandard instance shutdown, this trigger might not fire.
		None allowed	Trigger cannot do database operations.	Starts separate transaction and commits it after firing triggers.	`ora_sysevent`
					`ora_login_user`
					`ora_instance_num`
					`ora_database_name`

`AFTER DB_ROLE_CHANGE`	When database is opened for first time after role change.	None allowed	None	Starts separate transaction and commits it after firing triggers.	`ora_sysevent`
					`ora_login_user`
					`ora_instance_num`
					`ora_database_name`

`AFTER SERVERERROR`	With condition, whenever specified error occurs. Without condition, whenever any error occurs.
	Trigger does not fire for errors listed in "*database_event*".
		`ERRNO = eno`	Depends on error.	Starts separate transaction and commits it after firing triggers.	`ora_sysevent`
					`ora_login_user`
					`ora_instance_num`
					`ora_database_name`
					`ora_server_error`
					`ora_is_servererror`
					`ora_space_error_info`

Event Attribute Functions for Client Event Triggers

Table 10-7 summarizes the client event triggers that can invoke event attribute functions. For more information about the triggering events in Table 10-7, see "*ddl_event*" and "*database_event*".

> **Note:**
>
> If a client event trigger becomes the target of a DDL operation (such as `CREATE OR REPLACE TRIGGER`), then it cannot fire later during the same transaction.

Table 10-7 Client Event Triggers

Triggering Event	When Trigger Fires	WHEN Conditions	Restrictions	Transaction	Attribute Functions
BEFORE ALTER AFTER ALTER	When catalog object is altered	Simple conditions on type and name of object, UID, and USER	Trigger cannot do DDL operations on object that caused event to be generated. DDL on other objects is limited to compiling an object, creating a trigger, and creating, altering, and dropping a table.	Fires triggers in current transaction.	ora_sysevent ora_login_user ora_instance_num ora_database_name ora_dict_obj_type ora_dict_obj_name ora_dict_obj_owner ora_des_encrypted_password (for ALTER USER events) ora_is_alter_column (for ALTER TABLE events) ora_is_drop_column (for ALTER TABLE events)
BEFORE DROP AFTER DROP	When catalog object is dropped	Simple conditions on type and name of object, UID, and USER	Trigger cannot do DDL operations on object that caused event to be generated. DDL on other objects is limited to compiling an object, creating a trigger, and creating, altering, and dropping a table.	Fires triggers in current transaction.	ora_sysevent ora_login_user ora_instance_num ora_database_name ora_dict_obj_type ora_dict_obj_name ora_dict_obj_owner
BEFORE ANALYZE AFTER ANALYZE	When ANALYZE statement is issued	Simple conditions on type and name of object, UID, and USER	Trigger cannot do DDL operations on object that caused event to be generated. DDL on other objects is limited to compiling an object, creating a trigger, and creating, altering, and dropping a table.	Fires triggers in current transaction.	ora_sysevent ora_login_user ora_instance_num ora_database_name ora_dict_obj_name ora_dict_obj_type ora_dict_obj_owner
BEFORE ASSOCIATE STATISTICS AFTER ASSOCIATE STATISTICS	When ASSOCIATE STATISTICS statement is issued	Simple conditions on type and name of object, UID, and USER	Trigger cannot do DDL operations on object that caused event to be generated. DDL on other objects is limited to compiling an object, creating a trigger, and creating, altering, and dropping a table.	Fires triggers in current transaction.	ora_sysevent ora_login_user ora_instance_num ora_database_name ora_dict_obj_name ora_dict_obj_type ora_dict_obj_owner ora_dict_obj_name_list ora_dict_obj_owner_list

Table 10-7 (Cont.) Client Event Triggers

Triggering Event	When Trigger Fires	WHEN Conditions	Restrictions	Transaction	Attribute Functions
BEFORE AUDIT AFTER AUDIT BEFORE NOAUDIT AFTER NOAUDIT	When AUDIT or NOAUDIT statement is issued	Simple conditions on type and name of object, UID, and USER	Trigger cannot do DDL operations on object that caused event to be generated. DDL on other objects is limited to compiling an object, creating a trigger, and creating, altering, and dropping a table.	Fires triggers in current transaction.	ora_sysevent ora_login_user ora_instance_num ora_database_name
BEFORE COMMENT AFTER COMMENT	When object is commented	Simple conditions on type and name of object, UID, and USER	Trigger cannot do DDL operations on object that caused event to be generated. DDL on other objects is limited to compiling an object, creating a trigger, and creating, altering, and dropping a table.	Fires triggers in current transaction.	ora_sysevent ora_login_user ora_instance_num ora_database_name ora_dict_obj_name ora_dict_obj_type ora_dict_obj_owner
BEFORE CREATE AFTER CREATE	When catalog object is created	Simple conditions on type and name of object, UID, and USER	Trigger cannot do DDL operations on object that caused event to be generated. DDL on other objects is limited to compiling an object, creating a trigger, and creating, altering, and dropping a table.	Fires triggers in current transaction.	ora_sysevent ora_login_user ora_instance_num ora_database_name ora_dict_obj_type ora_dict_obj_name ora_dict_obj_owner ora_is_creating_nested_table (for CREATE TABLE events)

Table 10-7 (Cont.) Client Event Triggers

Triggering Event	When Trigger Fires	WHEN Conditions	Restrictions	Transaction	Attribute Functions

`BEFORE DDL`
 	
`AFTER DDL`
	When most SQL DDL statements are issued. Not fired for `ALTER DATABASE`, `CREATE CONTROLFILE`, `CREATE DATABASE`, and DDL issued through the PL/SQL subprogram interface, such as creating an advanced queue.	Simple conditions on type and name of object, `UID`, and `USER`	Trigger cannot do DDL operations on object that caused event to be generated.
			DDL on other objects is limited to compiling an object, creating a trigger, and creating, altering, and dropping a table.
				Fires triggers in current transaction.	`ora_sysevent`
					`ora_login_user`
					`ora_instance_num`
					`ora_database_name`
					`ora_dict_obj_name`
					`ora_dict_obj_type`
					`ora_dict_obj_owner`

`BEFORE DISASSOCIATE STATISTICS`
 	
`AFTER DISASSOCIATE STATISTICS`
	When `DISASSOCIATE STATISTICS` statement is issued	Simple conditions on type and name of object, `UID`, and `USER`	Trigger cannot do DDL operations on object that caused event to be generated.
			DDL on other objects is limited to compiling an object, creating a trigger, and creating, altering, and dropping a table.
				Fires triggers in current transaction.	`ora_sysevent`
					`ora_login_user`
					`ora_instance_num`
					`ora_database_name`
					`ora_dict_obj_name`
					`ora_dict_obj_type`
					`ora_dict_obj_owner`
					`ora_dict_obj_name_list`
					`ora_dict_obj_owner_list`

`BEFORE GRANT`
 	
`AFTER GRANT`
	When `GRANT` statement is issued	Simple conditions on type and name of object, `UID`, and `USER`	Trigger cannot do DDL operations on object that caused event to be generated.
			DDL on other objects is limited to compiling an object, creating a trigger, and creating, altering, and dropping a table.
				Fires triggers in current transaction.	`ora_sysevent`
					`ora_login_user`
					`ora_instance_num`
					`ora_database_name`
					`ora_dict_obj_name`
					`ora_dict_obj_type`
					`ora_dict_obj_owner`
					`ora_grantee`
					`ora_with_grant_option`
					`ora_privilege_list`

`BEFORE LOGOFF`	At start of user logoff	Simple conditions on `UID` and `USER`	DDL on other objects is limited to compiling an object, creating a trigger, and creating, altering, and dropping a table.	Fires triggers in current transaction.	`ora_sysevent`
					`ora_login_user`
					`ora_instance_num`
					`ora_database_name`

Table 10-7 (Cont.) Client Event Triggers

Triggering Event	When Trigger Fires	WHEN Conditions	Restrictions	Transaction	Attribute Functions
AFTER LOGON	After successful user logon	Simple conditions on UID and USER	DDL on other objects is limited to compiling an object, creating a trigger, and creating, altering, and dropping a table.	Starts separate transaction and commits it after firing triggers.	ora_sysevent ora_login_user ora_instance_num ora_database_name ora_client_ip_address
BEFORE RENAME AFTER RENAME	When RENAME statement is issued	Simple conditions on type and name of object, UID, and USER	Trigger cannot do DDL operations on object that caused event to be generated. DDL on other objects is limited to compiling an object, creating a trigger, and creating, altering, and dropping a table.	Fires triggers in current transaction.	ora_sysevent ora_login_user ora_instance_num ora_database_name ora_dict_obj_name ora_dict_obj_owner ora_dict_obj_type
BEFORE REVOKE AFTER REVOKE	When REVOKE statement is issued	Simple conditions on type and name of object, UID, and USER	Trigger cannot do DDL operations on object that caused event to be generated. DDL on other objects is limited to compiling an object, creating a trigger, and creating, altering, and dropping a table.	Fires triggers in current transaction.	ora_sysevent ora_login_user ora_instance_num ora_database_name ora_dict_obj_name ora_dict_obj_type ora_dict_obj_owner ora_revokee ora_privilege_list
AFTER SUSPEND	After SQL statement is suspended because of out-of-space condition. (Trigger must correct condition so statement can be resumed.)	Simple conditions on type and name of object, UID, and USER	Trigger cannot do DDL operations on object that caused event to be generated. DDL on other objects is limited to compiling an object, creating a trigger, and creating, altering, and dropping a table.	Fires triggers in current transaction.	ora_sysevent ora_login_user ora_instance_num ora_database_name ora_server_error ora_is_servererror ora_space_error_info

Table 10-7 (Cont.) Client Event Triggers

Triggering Event	When Trigger Fires	WHEN Conditions	Restrictions	Transaction	Attribute Functions

`BEFORE TRUNCATE`
 	
`AFTER TRUNCATE`
	When object is truncated	Simple conditions on type and name of object, `UID`, and `USER`	Trigger cannot do DDL operations on object that caused event to be generated.
			DDL on other objects is limited to compiling an object, creating a trigger, and creating, altering, and dropping a table.
				Fires triggers in current transaction.	`ora_sysevent`
					`ora_login_user`
					`ora_instance_num`
					`ora_database_name`
					`ora_dict_obj_name`
					`ora_dict_obj_type`
					`ora_dict_obj_owner`

Views for Information About Triggers

The `*_TRIGGERS` static data dictionary views reveal information about triggers. For information about these views, see *Oracle Database Reference*.

Example 10-28 Viewing Information About Triggers

This example creates a trigger and queries the static data dictionary view `USER_TRIGGERS` twice —first to show its type, triggering event, and the name of the table on which it is created, and then to show its body.

```
CREATE OR REPLACE TRIGGER Emp_count
  AFTER DELETE ON employees
DECLARE
  n  INTEGER;
BEGIN
  SELECT COUNT(*) INTO n FROM employees;
  DBMS_OUTPUT.PUT_LINE('There are now ' || n || ' employees.');
END;
/
```

These SQL*Plus commands format the query results.

```
COLUMN Trigger_type FORMAT A15
COLUMN Triggering_event FORMAT A16
COLUMN Table_name FORMAT A11
COLUMN Trigger_body FORMAT A50
SET LONG 9999
```

Query:

```
SELECT Trigger_type, Triggering_event, Table_name
FROM USER_TRIGGERS
WHERE Trigger_name = 'EMP_COUNT';
```

Result:

```
TRIGGER_TYPE    TRIGGERING_EVENT TABLE_NAME
--------------- ---------------- -----------
AFTER STATEMENT DELETE           EMPLOYEES
```

Query:

```
SELECT Trigger_body
FROM USER_TRIGGERS
WHERE Trigger_name = 'EMP_COUNT';
```

Result:

```
TRIGGER_BODY
-------------------------------------------------
DECLARE
  n  INTEGER;
BEGIN
  SELECT COUNT(*) INTO n FROM employees;
  DBMS_OUTPUT.PUT_LINE('There are now ' || n || '
employees.');
END;
```