**Compound Triggers**

A compound trigger can fire at more than one timing point.

**Why Use Compound Triggers?**

The compound trigger makes it easier to program an approach where you want the actions you implement for the various timing points to share common data. To achieve the same effect with simple triggers, you had to model the common state with an ancillary package. This approach was both cumbersome to program and subject to memory leak when the triggering statement caused an error and the after-statement trigger did not fire.

A compound trigger has an optional declarative part and a section for each of its timing points (see [Example 9-2](http://docs.oracle.com/cd/B28359_01/appdev.111/b28370/triggers.htm#CIHDJFBC)). All of these sections 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.

***Example 9-2 Compound Trigger***

SQL> CREATE OR REPLACE TRIGGER compound\_trigger

2 FOR UPDATE OF salary ON employees

3 COMPOUND TRIGGER

4

5 -- Declarative part (optional)

6 -- Variables declared here have firing-statement duration.

7 threshold CONSTANT SIMPLE\_INTEGER := 200;

8

9 BEFORE STATEMENT IS

10 BEGIN

11 NULL;

12 END BEFORE STATEMENT;

13

14 BEFORE EACH ROW IS

15 BEGIN

16 NULL;

17 END BEFORE EACH ROW;

18

19 AFTER EACH ROW IS

20 BEGIN

21 NULL;

22 END AFTER EACH ROW;

23

24 AFTER STATEMENT IS

25 BEGIN

26 NULL;

27 END AFTER STATEMENT;

28 END compound\_trigger;

29 /

Trigger created.

SQL>

Two common reasons to use compound triggers are:

* To accumulate rows destined for a second table so that you can periodically bulk-insert them (as in[Compound Trigger Example](http://docs.oracle.com/cd/B28359_01/appdev.111/b28370/triggers.htm#CIHFHIBH))
* To avoid the mutating-table error (ORA-04091) (as in [Using Compound Triggers to Avoid Mutating-Table Error](http://docs.oracle.com/cd/B28359_01/appdev.111/b28370/triggers.htm#CHDFEBFJ))

**Compound Trigger Sections**

A compound trigger has a declarative part and at least one timing-point section. It cannot have multiple sections for the same timing point.

The optional **declarative part** (the first part) declares variables and subprograms that timing-point sections can use. When the trigger fires, the declarative part executes before any timing-point sections execute. Variables and subprograms declared in this section have firing-statement duration.

A compound trigger defined on a view has an INSTEAD OF EACH ROW timing-point section, and no other timing-point section.

A compound trigger defined on a table has one or more of the timing-point sections described in [Table 9-1](http://docs.oracle.com/cd/B28359_01/appdev.111/b28370/triggers.htm#g4897004). Timing-point sections must appear in the order shown in [Table 9-1](http://docs.oracle.com/cd/B28359_01/appdev.111/b28370/triggers.htm#g4897004). If a timing-point section is absent, nothing happens at its timing point.

A timing-point section cannot be enclosed in a PL/SQL block.

[Table 9-1](http://docs.oracle.com/cd/B28359_01/appdev.111/b28370/triggers.htm#g4897004) summarizes the timing point sections of a compound trigger that can be defined on a table.

***Table 9-1 Timing-Point Sections of a Compound Trigger Defined***

| **Timing Point** | **Section** |
| --- | --- |
| Before the triggering statement executes | BEFORE STATEMENT |
| After the triggering statement executes | AFTER STATEMENT |
| Before each row that the triggering statement affects | BEFORE EACH ROW |
| After each row that the triggering statement affects | AFTER EACH ROW |

Any section can include the functions Inserting, Updating, Deleting, and Applying.

**Triggering Statements of Compound Triggers**

The triggering statement of a compound trigger must be a DML statement.

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

It is when the triggering statement affects many rows that a compound trigger has a performance benefit. This is why it is important to use the BULK COLLECT clause with the FORALL statement. For example, without the BULKCOLLECT clause, a FORALL statement that contains an INSERT statement simply performs a single-row insertion operation many times, and you get no benefit from using a compound trigger. For more information about using the BULK COLLECT clause with the FORALL statement, see [Using FORALL and BULK COLLECT Together](http://docs.oracle.com/cd/B28359_01/appdev.111/b28370/tuning.htm#BCGICBDF).

If the triggering statement of a compound trigger is an INSERT statement that includes a subquery, the compound trigger retains some of its performance benefit. For example, suppose that a compound trigger is triggered by the following statement:

INSERT INTO Target

SELECT c1, c2, c3

FROM Source

WHERE Source.c1 > 0

For each row of Source whose column c1 is greater than zero, the BEFORE EACH ROW and AFTER EACH ROW sections of the compound trigger execute. However, the BEFORE STATEMENT and AFTER STATEMENT sections each execute only once (before and after the INSERT statement executes, respectively).

**Compound Trigger Restrictions**

* The body of a compound trigger must be a compound trigger block.
* A compound trigger must be a DML trigger.
* A compound trigger must be defined on either a table or a view.
* The declarative part cannot include PRAGMA AUTONOMOUS\_TRANSACTION.
* A compound trigger body cannot have an initialization block; therefore, it cannot have an exception section.

This is not a problem, because the BEFORE STATEMENT section always executes exactly once before any other timing-point section executes.

* An exception that occurs in one section must be handled in that section. It cannot transfer control to another section.
* If a section includes a GOTO statement, the target of the GOTO statement must be in the same section.
* :OLD, :NEW, and :PARENT cannot appear in the declarative part, the BEFORE STATEMENT section, or the AFTERSTATEMENT section.
* Only the BEFORE EACH ROW section can change the value of :NEW.
* If, after the compound trigger fires, the triggering statement rolls back due to a DML exception:
  + Local variables declared in the compound trigger sections are re-initialized, and any values computed thus far are lost.
  + Side effects from firing the compound trigger are not rolled back.
* The firing order of compound triggers is not guaranteed. Their firing can be interleaved with the firing of simple triggers.
* If compound triggers are ordered using the FOLLOWS option, and if the target of FOLLOWS does not contain the corresponding section as source code, the ordering is ignored.

**Compound Trigger Example**

**Scenario:** You want to record every change to hr.employees.salary in a new table, employee\_salaries. A singleUPDATE statement will update many rows of the table hr.employees; therefore, bulk-inserting rows intoemployee.salaries is more efficient than inserting them individually.

**Solution:** Define a compound trigger on updates of the table hr.employees, as in [Example 9-3](http://docs.oracle.com/cd/B28359_01/appdev.111/b28370/triggers.htm#CIHGJFAB). You do not need aBEFORE 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).

***Example 9-3 Compound Trigger Records 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 := SYSDATE();

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

/

/\* Wait two seconds: \*/

BEGIN

DBMS\_LOCK.SLEEP(2);

END;

/

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

UPDATE employees

SET salary = salary \* 1.05

WHERE department\_id = 50

/

**Using Compound Triggers to Avoid Mutating-Table Error**

You can use compound triggers to avoid the mutating-table error (ORA-04091) described in [Trigger Restrictions on Mutating Tables](http://docs.oracle.com/cd/B28359_01/appdev.111/b28370/triggers.htm#g1699708).

**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 9-4](http://docs.oracle.com/cd/B28359_01/appdev.111/b28370/triggers.htm#CHDFGDAH). The state variables are initialized each time the trigger fires (even when the triggering statement is interrupted and restarted).

***Example 9-4 Compound Trigger that 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;

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;