

---

## Using Triggers

Triggers are procedures that are stored in the database and implicitly run, or **fired**, when something happens.

Traditionally, triggers supported the execution of a PL/SQL block when an `INSERT`, `UPDATE`, or `DELETE` occurred on a table or view. Starting with Oracle8i, triggers support system and other data events on `DATABASE` and `SCHEMA`. Oracle also supports the execution of a PL/SQL or Java procedure.

This chapter discusses DML triggers, `INSTEAD OF` triggers, and system triggers (triggers on `DATABASE` and `SCHEMA`). Topics include:

- [Designing Triggers](#)
- [Creating Triggers](#)
- [Compiling Triggers](#)
- [Modifying Triggers](#)
- [Enabling and Disabling Triggers](#)
- [Viewing Information About Triggers](#)
- [Examples of Trigger Applications](#)
- [Responding to System Events through Triggers](#)

## Designing Triggers

Use the following guidelines when designing your triggers:

- Use triggers to guarantee that when a specific operation is performed, related actions are performed.
- Do not define triggers that duplicate features already built into Oracle. For example, do not define triggers to reject bad data if you can do the same checking through declarative integrity constraints.
- Limit the size of triggers. If the logic for your trigger requires much more than 60 lines of PL/SQL code, it is better to include most of the code in a stored procedure and call the procedure from the trigger.
- Use triggers only for centralized, global operations that should be fired for the triggering statement, regardless of which user or database application issues the statement.
- **Do not create recursive triggers.** For example, creating an AFTER UPDATE statement trigger on the Emp\_tab table that itself issues an UPDATE statement on Emp\_tab, causes the trigger to fire recursively until it has run out of memory.
- Use triggers on DATABASE judiciously. They are executed for *every user every* time the event occurs on which the trigger is created.

## Creating Triggers

Triggers are created using the CREATE TRIGGER statement. This statement can be used with any interactive tool, such as SQL\*Plus or Enterprise Manager. When using an interactive tool, a single slash (/) on the last line is necessary to activate the CREATE TRIGGER statement.

The following statement creates a trigger for the Emp\_tab table.

```
CREATE OR REPLACE TRIGGER Print_salary_changes
  BEFORE DELETE OR INSERT OR UPDATE ON Emp_tab
  FOR EACH ROW
  WHEN (new.Empno > 0)
  DECLARE
    sal_diff number;
  BEGIN
    sal_diff := :new.sal - :old.sal;
    dbms_output.put('Old salary: ' || :old.sal);
    dbms_output.put(' New salary: ' || :new.sal);
```

```

        dbms_output.put_line(' Difference ' || sal_diff);
    END;
/

```

The trigger is fired when DML operations (INSERT, UPDATE, and DELETE statements) are performed on the table. You can choose what combination of operations should fire the trigger.

Because the trigger uses the **BEFORE** keyword, it can access the new values before they go into the table, and can change the values if there is an easily-corrected error by assigning to `:NEW.column_name`. You might use the **AFTER** keyword if you want the trigger to query or change the same table, because triggers can only do that after the initial changes are applied and the table is back in a consistent state.

Because the trigger uses the **FOR EACH ROW** clause, it might be executed multiple times, such as when updating or deleting multiple rows. You might omit this clause if you just want to record the fact that the operation occurred, but not examine the data for each row.

Once the trigger is created, entering the following SQL statement:

```
UPDATE Emp_tab SET sal = sal + 500.00 WHERE deptno = 10;
```

fires the trigger once for each row that is updated, in each case printing the new salary, old salary, and the difference.

The **CREATE** (or **CREATE OR REPLACE**) statement fails if any errors exist in the PL/SQL block.

---



---

**Note:** The size of the trigger cannot be more than 32K.

---



---

The following sections use this example to illustrate the way that parts of a trigger are specified.

**See Also:** For more realistic examples of **CREATE TRIGGER** statements, see "[Examples of Trigger Applications](#)" on page 15-32.

## Types of Triggers

A trigger is either a stored PL/SQL block or a PL/SQL, C, or Java procedure associated with a table, view, schema, or the database itself. Oracle automatically

executes a trigger when a specified event takes place, which may be in the form of a system event or a DML statement being issued against the table.

Triggers can be:

- DML triggers on tables.
- `INSTEAD OF` triggers on views.
- System triggers on `DATABASE` or `SCHEMA`: With `DATABASE`, triggers fire for each event for all users; with `SCHEMA`, triggers fire for each event for that specific user.

**See Also:** *Oracle9i SQL Reference* explains the syntax for creating triggers.

### Overview of System Events

You can create triggers to be fired on any of the following:

- DML statements (`DELETE`, `INSERT`, `UPDATE`)
- DDL statements (`CREATE`, `ALTER`, `DROP`)
- Database operations (`SERVERERROR`, `LOGON`, `LOGOFF`, `STARTUP`, `SHUTDOWN`)

### Getting the Attributes of System Events

You can get certain event-specific attributes when the trigger is fired.

**See Also:** For a complete list of the functions you can call to get the event attributes, see [Chapter 16, "Working With System Events"](#).

Creating a trigger on `DATABASE` implies that the triggering event is outside the scope of a user (for example, database `STARTUP` and `SHUTDOWN`), and it applies to all users (for example, a trigger created on `LOGON` event by the DBA).

Creating a trigger on `SCHEMA` implies that the trigger is created in the current user's schema and is fired only for that user.

For each trigger, publication can be specified on DML and system events.

**See Also:** ["Responding to System Events through Triggers"](#) on page 15-52.

## Naming Triggers

Trigger names must be unique with respect to other triggers in the same schema. Trigger names do not need to be unique with respect to other schema objects, such as tables, views, and procedures. For example, a table and a trigger can have the same name (however, to avoid confusion, this is not recommended).

## When Is the Trigger Fired?

A trigger is fired based on a **triggering statement**, which specifies:

- The SQL statement or the system event, database event, or DDL event that fires the trigger body. The options include `DELETE`, `INSERT`, and `UPDATE`. One, two, or all three of these options can be included in the triggering statement specification.
- The table, view, `DATABASE`, or `SCHEMA` associated with the trigger.

---

---

**Note:** Exactly one table or view can be specified in the triggering statement. If the `INSTEAD OF` option is used, then the triggering statement may only specify a view; conversely, if a view is specified in the triggering statement, then only the `INSTEAD OF` option may be used.

---

---

For example, the `PRINT_SALARY_CHANGES` trigger fires after any `DELETE`, `INSERT`, or `UPDATE` on the `Emp_tab` table. Any of the following statements trigger the `PRINT_SALARY_CHANGES` trigger given in the previous example:

```
DELETE FROM Emp_tab;
INSERT INTO Emp_tab VALUES ( ... );
INSERT INTO Emp_tab SELECT ... FROM ... ;
UPDATE Emp_tab SET ... ;
```

## Do Import and SQL\*Loader Fire Triggers?

`INSERT` triggers fire during `SQL*Loader` conventional loads. (For direct loads, triggers are disabled before the load.)

The `IGNORE` parameter of the `IMP` command determines whether triggers fire during import operations:

- If `IGNORE=N` (default) and the table already exists, then import does not change the table and no existing triggers fire.
- If the table does not exist, then import creates and loads it before any triggers are defined, so again no triggers fire.
- If `IGNORE=Y`, then import loads rows into existing tables. Any existing triggers fire, and indexes are updated to account for the imported data.

### How Column Lists Affect UPDATE Triggers

An `UPDATE` statement might include a list of columns. If a triggering statement includes a column list, the trigger is fired only when one of the specified columns is updated. If a triggering statement omits a column list, the trigger is fired when any column of the associated table is updated. A column list cannot be specified for `INSERT` or `DELETE` triggering statements.

The previous example of the `PRINT_SALARY_CHANGES` trigger could include a column list in the triggering statement. For example:

```
... BEFORE DELETE OR INSERT OR UPDATE OF ename ON Emp_tab ...
```

Notes:

- You cannot specify a column list for `UPDATE` with `INSTEAD OF` triggers.
- If the column specified in the `UPDATE OF` clause is an object column, then the trigger is also fired if any of the attributes of the object are modified.
- You cannot specify `UPDATE OF` clauses on collection columns.

### Controlling When a Trigger Is Fired (BEFORE and AFTER Options)

The `BEFORE` or `AFTER` option in the `CREATE TRIGGER` statement specifies exactly when to fire the trigger body in relation to the triggering statement that is being run. In a `CREATE TRIGGER` statement, the `BEFORE` or `AFTER` option is specified just before the triggering statement. For example, the `PRINT_SALARY_CHANGES` trigger in the previous example is a `BEFORE` trigger.

---

---

**Note:** AFTER row triggers are slightly more efficient than BEFORE row triggers. With BEFORE row triggers, affected data blocks must be read (logical read, not physical read) once for the trigger and then again for the triggering statement.

Alternatively, with AFTER row triggers, the data blocks must be read only once for both the triggering statement and the trigger.

---

---

## Modifying Complex Views (INSTEAD OF Triggers)

The INSTEAD OF option can also be used in triggers. INSTEAD OF triggers provide a transparent way of modifying views that cannot be modified directly through UPDATE, INSERT, and DELETE statements. These triggers are called INSTEAD OF triggers because, unlike other types of triggers, Oracle fires the trigger *instead* of executing the triggering statement. The trigger must determine what operation was intended and perform UPDATE, INSERT, or DELETE operations directly on the underlying tables.

With an INSTEAD OF trigger, you can write normal UPDATE, INSERT, and DELETE statements against the view, and the INSTEAD OF trigger works invisibly in the background to make the right actions take place.

INSTEAD OF triggers can only be activated for each row.

**See Also:** ["Firing Triggers One or Many Times \(FOR EACH ROW Option\)"](#) on page 15-11

Notes:

- The INSTEAD OF option can *only* be used for triggers created over views.
- The BEFORE and AFTER options *cannot* be used for triggers created over views.
- The CHECK option for views is not enforced when inserts or updates to the view are done using INSTEAD OF triggers. The INSTEAD OF trigger body must enforce the check.

### Views that Require INSTEAD OF Triggers

A view cannot be modified by UPDATE, INSERT, or DELETE statements if the view query contains any of the following constructs:

- Set operators
- Group functions

- GROUP BY, CONNECT BY, or START WITH clauses
- The DISTINCT operator
- Joins (a subset of join views are updatable)

If a view contains pseudocolumns or expressions, then you can only update the view with an UPDATE statement that does not refer to any of the pseudocolumns or expressions.

### INSTEAD OF Trigger Example

---

---

**Note:** You may need to set up the following data structures for this example to work:

```
CREATE TABLE Project_tab (  
    Prj_level NUMBER,  
    Projno    NUMBER,  
    Resp_dept NUMBER);  
CREATE TABLE Emp_tab (  
    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_tab (  
    Deptno   NUMBER(2) NOT NULL,  
    Dname    VARCHAR2(14),  
    Loc      VARCHAR2(13),  
    Mgr_no   NUMBER,  
    Dept_type NUMBER);
```

---

---

The following example shows an INSTEAD OF trigger for inserting rows into the MANAGER\_INFO view.

```
CREATE OR REPLACE VIEW manager_info AS  
    SELECT e.ename, e.empno, d.dept_type, d.deptno, p.prj_level,  
           p.projno  
    FROM   Emp_tab e, Dept_tab d, Project_tab p  
    WHERE  e.empno = d.mgr_no  
    AND    d.deptno = p.resp_dept;
```



```

CREATE OR REPLACE TRIGGER manager_info_insert
INSTEAD OF INSERT ON manager_info
REFERENCING NEW AS n                -- new manager information

FOR EACH ROW
DECLARE
    rowcnt number;
BEGIN
    SELECT COUNT(*) INTO rowcnt FROM Emp_tab WHERE empno = :n.empno;
    IF rowcnt = 0 THEN
        INSERT INTO Emp_tab (empno,ename) VALUES (:n.empno, :n.ename);
    ELSE
        UPDATE Emp_tab SET Emp_tab.ename = :n.ename
        WHERE Emp_tab.empno = :n.empno;
    END IF;
    SELECT COUNT(*) INTO rowcnt FROM Dept_tab WHERE deptno = :n.deptno;
    IF rowcnt = 0 THEN
        INSERT INTO Dept_tab (deptno, dept_type)
        VALUES(:n.deptno, :n.dept_type);
    ELSE
        UPDATE Dept_tab SET Dept_tab.dept_type = :n.dept_type
        WHERE Dept_tab.deptno = :n.deptno;
    END IF;
    SELECT COUNT(*) INTO rowcnt FROM Project_tab
    WHERE Project_tab.projno = :n.projno;
    IF rowcnt = 0 THEN
        INSERT INTO Project_tab (projno, prj_level)
        VALUES(:n.projno, :n.prj_level);
    ELSE
        UPDATE Project_tab SET Project_tab.prj_level = :n.prj_level
        WHERE Project_tab.projno = :n.projno;
    END IF;
END;

```

The actions shown for rows being inserted into the `MANAGER_INFO` view first test to see if appropriate rows already exist in the base tables from which `MANAGER_INFO` is derived. The actions then insert new rows or update existing rows, as appropriate. Similar triggers can specify appropriate actions for `UPDATE` and `DELETE`.

### Object Views and INSTEAD OF Triggers

INSTEAD OF triggers provide the means to modify object view instances on the client-side through OCI calls.

**See Also:** *Oracle Call Interface Programmer's Guide*

To modify an object materialized by an object view in the client-side object cache and flush it back to the persistent store, you must specify INSTEAD OF triggers, unless the object view is modifiable. If the object is read only, then it is not necessary to define triggers to pin it.

### Triggers on Nested Table View Columns

INSTEAD OF triggers can also be created over nested table view columns. These triggers provide a way of updating elements of the nested table. They fire for each nested table element being modified. The row correlation variables inside the trigger correspond to the nested table element. This type of trigger also provides an additional correlation name for accessing the parent row that contains the nested table being modified.

---

---

**Note:** These triggers:

- Can only be defined over nested table columns in views.
  - Fire only when the nested table elements are modified using the THE() or TABLE() clauses. They do not fire when a DML statement is performed on the view.
- 
- 

For example, consider a department view that contains a nested table of employees.

```
CREATE OR REPLACE VIEW Dept_view AS
SELECT d.Deptno, d.Dept_type, d.Dept_name,
       CAST (MULTISET ( SELECT e.Empno, e.Empname, e.Salary
                       FROM Emp_tab e
                       WHERE e.Deptno = d.Deptno) AS Emp_list_ Emplist
FROM Dept_tab d;
```

The CAST (MULTISET..) operator creates a multi-set of employees for each department. Now, if you want to modify the emp\_list column, which is the nested table of employees, then you can define an INSTEAD OF trigger over the column to handle the operation.

The following example shows how an insert trigger might be written:

```

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
-- The insert on the nested table is translated to an insert on the base table:
  INSERT INTO Emp_tab VALUES (
    :Employee.Empno, :Employee.Empname, :Employee.Salary, :Department.Deptno);
END;

```

Any INSERT into the nested table fires the trigger, and the Emp\_tab table is filled with the correct values. For example:

```

INSERT INTO TABLE (SELECT d.Emplist FROM Dept_view d WHERE Deptno = 10)
  VALUES (1001, 'John Glenn', 10000)

```

The :department.deptno correlation variable in this example would have a value of 10.

## Firing Triggers One or Many Times (FOR EACH ROW Option)

The FOR EACH ROW option determines whether the trigger is a *row* trigger or a *statement* trigger. If you specify FOR EACH ROW, then the trigger fires once for each row of the table that is affected by the triggering statement. The absence of the FOR EACH ROW option indicates that the trigger fires only once for each applicable statement, but not separately for each row affected by the statement.

For example, you define the following trigger:

---



---

**Note:** You may need to set up the following data structures for certain examples to work:

```

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

```

---



---

```

CREATE OR REPLACE TRIGGER Log_salary_increase
  AFTER UPDATE ON Emp_tab
  FOR EACH ROW

```

```
WHEN (new.Sal > 1000)
BEGIN
    INSERT INTO Emp_log (Emp_id, Log_date, New_salary, Action)
        VALUES (:new.Empno, SYSDATE, :new.SAL, 'NEW SAL');
END;
```

Then, you enter the following SQL statement:

```
UPDATE Emp_tab SET Sal = Sal + 1000.0
    WHERE Deptno = 20;
```

If there are five employees in department 20, then the trigger fires five times when this statement is entered, because five rows are affected.

The following trigger fires only once for each UPDATE of the Emp\_tab table:

```
CREATE OR REPLACE TRIGGER Log_emp_update
AFTER UPDATE ON Emp_tab
BEGIN
    INSERT INTO Emp_log (Log_date, Action)
        VALUES (SYSDATE, 'Emp_tab COMMISSIONS CHANGED');
END;
```

**See Also:** For the order of trigger firing, see *Oracle9i Database Concepts*.

The statement level triggers are useful for performing validation checks for the entire statement.

### Firing Triggers Based on Conditions (WHEN Clause)

Optionally, a trigger restriction can be included in the definition of a row trigger by specifying a Boolean SQL expression in a WHEN clause.

---

---

**Note:** A WHEN clause cannot be included in the definition of a statement trigger.

---

---

If included, then the expression in the WHEN clause is evaluated for each row that the trigger affects.

If the expression evaluates to TRUE for a row, then the trigger body is fired on behalf of that row. However, if the expression evaluates to FALSE or NOT TRUE for a row (unknown, as with nulls), then the trigger body is not fired for that row. The evaluation of the WHEN clause does not have an effect on the execution of the

triggering SQL statement (in other words, the triggering statement is not rolled back if the expression in a `WHEN` clause evaluates to `FALSE`).

For example, in the `PRINT_SALARY_CHANGES` trigger, the trigger body is not run if the new value of `Empno` is zero, `NULL`, or negative. In more realistic examples, you might test if one column value is less than another.

The expression in a `WHEN` clause of a row trigger can include correlation names, which are explained later. The expression in a `WHEN` clause must be a SQL expression, and it cannot include a subquery. You cannot use a PL/SQL expression (including user-defined functions) in the `WHEN` clause.

---

---

**Note:** You cannot specify the `WHEN` clause for `INSTEAD OF` triggers.

---

---

## Coding the Trigger Body

The trigger body is a `CALL` procedure or a PL/SQL block that can include SQL and PL/SQL statements. The `CALL` procedure can be either a PL/SQL or a Java procedure that is encapsulated in a PL/SQL wrapper. These statements are run if the triggering statement is entered and if the trigger restriction (if included) evaluates to `TRUE`.

The trigger body for row triggers has some special constructs that can be included in the code of the PL/SQL block: correlation names and the `REFERENCEING` option, and the conditional predicates `INSERTING`, `DELETING`, and `UPDATING`.

---

---

**Note:** The `INSERTING`, `DELETING`, and `UPDATING` conditional predicates cannot be used for the `CALL` procedures; they can only be used in a PL/SQL block.

---

---

### Example: Monitoring Logons with a Trigger

---

---

**Note:** You may need to set up data structures similar to the following for certain examples to work:

```
CONNECT system/manager
GRANT ADMINISTER DATABASE TRIGGER TO scott;
CONNECT scott/tiger
CREATE TABLE audit_table (
  seq number,
  user_at VARCHAR2(10),
  time_now DATE,
  term VARCHAR2(10),
  job VARCHAR2(10),
  proc VARCHAR2(10),
  enum NUMBER);
```

---

---

```

CREATE OR REPLACE PROCEDURE foo (c VARCHAR2) AS
BEGIN
    INSERT INTO Audit_table (user_at) VALUES(c);
END;

CREATE OR REPLACE TRIGGER logontrig AFTER LOGON ON DATABASE
-- Just call an existing procedure. The ORA_LOGIN_USER is a function
-- that returns information about the event that fired the trigger.
CALL foo (ora_login_user)
/

```

### Example: Calling a Java Procedure from a Trigger

Although triggers are declared using PL/SQL, they can call procedures in other languages, such as Java:

```

CREATE OR REPLACE PROCEDURE Before_delete (Id IN NUMBER, Ename VARCHAR2)
IS language Java
name 'thjvTriggers.beforeDelete (oracle.sql.NUMBER, oracle.sql.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.sql.*
import oracle.oraocore.*
public class thjvTriggers
{
    public state 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;
    }
}

```

## Accessing Column Values in Row Triggers

Within a trigger body of a row trigger, the PL/SQL code and SQL statements have access to the old and new column values of the current row affected by the triggering statement. Two correlation names exist for every column of the table being modified: one for the old column value, and one for the new column value. Depending on the type of triggering statement, certain correlation names might not have any meaning.

- A trigger fired by an `INSERT` statement has meaningful access to new column values only. Because the row is being created by the `INSERT`, the old values are null.
- A trigger fired by an `UPDATE` statement has access to both old and new column values for both `BEFORE` and `AFTER` row triggers.
- A trigger fired by a `DELETE` statement has meaningful access to `:old` column values only. Because the row no longer exists after the row is deleted, the `:new` values are `NULL`. However, you cannot modify `:new` values: `ORA-4084` is raised if you try to modify `:new` values.

The new column values are referenced using the `new` qualifier before the column name, while the old column values are referenced using the `old` qualifier before the column name. For example, if the triggering statement is associated with the `Emp_tab` table (with the columns `SAL`, `COMM`, and so on), then you can include statements in the trigger body. For example:

```
IF :new.Sal > 10000 ...  
IF :new.Sal < :old.Sal ...
```

Old and new values are available in both `BEFORE` and `AFTER` row triggers. A new column value can be assigned in a `BEFORE` row trigger, but not in an `AFTER` row trigger (because the triggering statement takes effect before an `AFTER` row trigger is fired). If a `BEFORE` row trigger changes the value of `new.column`, then an `AFTER` row trigger fired by the same statement sees the change assigned by the `BEFORE` row trigger.

Correlation names can also be used in the Boolean expression of a `WHEN` clause. A colon must precede the `old` and `new` qualifiers when they are used in a trigger's body, but a colon is not allowed when using the qualifiers in the `WHEN` clause or the `REFERENCING` option.



**Example: Modifying LOB Columns with a Trigger**

Formerly, you could examine LOB columns within a trigger body, but not modify them. Now, you can treat them the same as other columns, using regular SQL and PL/SQL functions with CLOB columns, and calls to the DBMS\_LOB package with BLOB columns:

```
drop table tabl;

create table tabl (c1 clob);
insert into tabl values ('<h1>HTML Document Fragment</h1><p>Some text.');
```

```
create or replace trigger trgl
  before update on tabl
  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);

  -- Previously, we couldn't change the new value for a LOB.
  -- Now, we can replace it, or construct a new value using SUBSTR, INSTR, etc.
  -- operations for a CLOB, or DBMS_LOB calls for a BLOB.
  :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 tabl set c1 = '<h1>Different Document Fragment</h1><p>Different text.';

select * from tabl;
```

**INSTEAD OF Triggers on Nested Table View Columns**

In the case of INSTEAD OF triggers on nested table view columns, the new and old qualifiers correspond to the new and old nested table elements. The parent row corresponding to this nested table element can be accessed using the parent qualifier. The parent correlation name is meaningful and valid only inside a nested table trigger.

### Avoiding Name Conflicts with Triggers (REFERENCING Option)

The `REFERENCING` option can be specified in a trigger body of a row trigger to avoid name conflicts among the correlation names and tables that might be named `old` or `new`. Because this is rare, this option is infrequently used.

For example, assume you have a table named `new` with columns `field1` (number) and `field2` (character). The following `CREATE TRIGGER` example shows a trigger associated with the `new` table that can use correlation names and avoid naming conflicts between the correlation names and the table name:

---

---

**Note:** You may need to set up the following data structures for certain examples to work:

```
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;
```

Notice that the `new` qualifier is renamed to `newest` using the `REFERENCING` option, and it is then used in the trigger body.

### Detecting the DML Operation That Fired a Trigger (INSERTING, UPDATING, and DELETING Predicates)

If more than one type of DML operation can fire a trigger (for example, `ON INSERT OR DELETE OR UPDATE OF Emp_tab`), the trigger body can use the conditional predicates `INSERTING`, `DELETING`, and `UPDATING` to check which type of statement fire the trigger.

Within the code of the trigger body, you can execute blocks of code depending on the kind of DML operation fired the trigger:

```
IF INSERTING THEN ... END IF;  
IF UPDATING THEN ... END IF;
```

The first condition evaluates to `TRUE` only if the statement that fired the trigger is an `INSERT` statement; the second condition evaluates to `TRUE` only if the statement that fired the trigger is an `UPDATE` statement.

In an `UPDATE` trigger, a column name can be specified with an `UPDATING` conditional predicate to determine if the named column is being updated. For example, assume a trigger is defined as the following:

```
CREATE OR REPLACE TRIGGER ...  
... UPDATE OF Sal, Comm ON Emp_tab ...  
BEGIN  
  
... IF UPDATING ('SAL') THEN ... END IF;  
  
END;
```

The code in the `THEN` clause runs only if the triggering `UPDATE` statement updates the `SAL` column. This way, the trigger can minimize its overhead when the column of interest is not being changed.

### **Error Conditions and Exceptions in the Trigger Body**

If a predefined or user-defined error condition or exception is raised during the execution of a trigger body, then all effects of the trigger body, as well as the triggering statement, are rolled back (unless the error is trapped by an exception handler). Therefore, a trigger body can prevent the execution of the triggering statement by raising an exception. User-defined exceptions are commonly used in triggers that enforce complex security authorizations or integrity constraints.

The only exception to this is when the event under consideration is database `STARTUP`, `SHUTDOWN`, or `LOGIN` when the user logging in is `SYSTEM`. In these scenarios, only the trigger action is rolled back.

## **Triggers and Handling Remote Exceptions**

A trigger that accesses a remote site cannot do remote exception handling if the network link is unavailable. For example:

```
CREATE OR REPLACE TRIGGER Example
AFTER INSERT ON Emp_tab
FOR EACH ROW
BEGIN
    INSERT INTO Emp_tab@Remote      -- <- compilation fails here
    VALUES ('x');                 --   when dblink is inaccessible
EXCEPTION
    WHEN OTHERS THEN
        INSERT INTO Emp_log
        VALUES ('x');
END;
```

A trigger is compiled when it is created. Thus, if a remote site is unavailable when the trigger must compile, then Oracle cannot validate the statement accessing the remote database, and the compilation fails. The previous example exception statement cannot run, because the trigger does not complete compilation.

Because stored procedures are stored in a compiled form, the work-around for the previous example is as follows:

```
CREATE OR REPLACE TRIGGER Example
AFTER INSERT ON Emp_tab
FOR EACH ROW
BEGIN
    Insert_row_proc;
END;

CREATE OR REPLACE PROCEDURE Insert_row_proc AS
BEGIN
    INSERT INTO Emp_tab@Remote
    VALUES ('x');
EXCEPTION
    WHEN OTHERS THEN
        INSERT INTO Emp_log
        VALUES ('x');
END;
```

The trigger in this example compiles successfully and calls the stored procedure, which already has a validated statement for accessing the remote database; thus, when the remote `INSERT` statement fails because the link is down, the exception is caught.

## Restrictions on Creating Triggers

Coding triggers requires some restrictions that are not required for standard PL/SQL blocks. The following sections discuss these restrictions.

### Maximum Trigger Size

The size of a trigger cannot be more than 32K.

### SQL Statements Allowed in Trigger Bodies

The body of a trigger can contain DML SQL statements. It can also contain `SELECT` statements, but they must be `SELECT... INTO...` statements or the `SELECT` statement in the definition of a cursor.

DDL statements are not allowed in the body of a trigger. Also, no transaction control statements are allowed in a trigger. `ROLLBACK`, `COMMIT`, and `SAVEPOINT` cannot be used. For system triggers, `{CREATE/ALTER/DROP} TABLE` statements and `ALTER...COMPILE` are allowed.

---

---

**Note:** A procedure called by a trigger cannot run the previous transaction control statements, because the procedure runs within the context of the trigger body.

---

---

Statements inside a trigger can reference remote schema objects. However, pay special attention when calling remote procedures from within a local trigger. If a timestamp or signature mismatch is found during execution of the trigger, then the remote procedure is not run, and the trigger is invalidated.

### Trigger Restrictions on LONG and LONG RAW Datatypes

`LONG` and `LONG RAW` datatypes in triggers are subject to the following restrictions:

- A SQL statement within a trigger can insert data into a column of `LONG` or `LONG RAW` datatype.
- If data from a `LONG` or `LONG RAW` column can be converted to a constrained datatype (such as `CHAR` and `VARCHAR2`), then a `LONG` or `LONG RAW` column can be referenced in a SQL statement within a trigger. The maximum length for these datatypes is 32000 bytes.
- Variables cannot be declared using the `LONG` or `LONG RAW` datatypes.
- `:NEW` and `:PARENT` cannot be used with `LONG` or `LONG RAW` columns.

### **BEFORE Triggers Fired Multiple Times**

If an UPDATE or DELETE statement detects a conflict with a concurrent UPDATE, then Oracle performs a transparent ROLLBACK to SAVEPOINT and restarts the update. This can occur many times before the statement completes successfully. Each time the statement is restarted, the BEFORE statement trigger is fired again. The rollback to savepoint does not undo changes to any package variables referenced in the trigger. The package should include a counter variable to detect this situation.

### **Row Evaluation Order for Triggers**

A relational database does not guarantee the order of rows processed by a SQL statement. Therefore, do not create triggers that depend on the order in which rows are processed. For example, do not assign a value to a global package variable in a row trigger if the current value of the global variable is dependent on the row being processed by the row trigger. Also, if global package variables are updated within a trigger, then it is best to initialize those variables in a BEFORE statement trigger.

When a statement in a trigger body causes another trigger to be fired, the triggers are said to be *cascading*. Oracle allows up to 32 triggers to cascade at any one time. However, you can effectively limit the number of trigger cascades using the initialization parameter OPEN\_CURSORS, because a cursor must be opened for every execution of a trigger.

### **Trigger Evaluation Order**

Although any trigger can run a sequence of operations either in-line or by calling procedures, using multiple triggers of the same type enhances database administration by permitting the modular installation of applications that have triggers on the same tables.

Oracle executes all triggers of the same type before executing triggers of a different type. If you have multiple triggers of the same type on a single table, then Oracle chooses an arbitrary order to execute these triggers.

**See Also:** *Oracle9i Database Concepts* has more information on the firing order of triggers.

Each subsequent trigger sees the changes made by the previously fired triggers. Each trigger can see the old and new values. The old values are the original values, and the new values are the current values, as set by the most recently fired UPDATE or INSERT trigger.

To ensure that multiple triggered actions occur in a specific order, you must consolidate these actions into a single trigger (for example, by having the trigger call a series of procedures).

You cannot open a database that contains multiple triggers of the same type if you are using any version of Oracle before release 7.1. You also cannot open such a database if your `COMPATIBLE` initialization parameter is set to a version earlier than 7.1.0. For system triggers, compatibility must be 8.1.0.

### Trigger Restrictions on Mutating Tables

A **mutating** table is a table that is currently being modified by an `UPDATE`, `DELETE`, or `INSERT` statement, or a table that might be updated by the effects of a `DELETE CASCADE` constraint.

The session that issued the triggering statement cannot query or modify a mutating table. This restriction prevents a trigger from seeing an inconsistent set of data.

This restriction applies to all triggers that use the `FOR EACH ROW` clause, and statement triggers that are fired as the result of a `DELETE CASCADE`. Views being modified in `INSTEAD OF` triggers are not considered mutating.

When a trigger encounters a mutating table, a runtime error occurs, the effects of the trigger body and triggering statement are rolled back, and control is returned to the user or application.

Consider the following trigger:

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

If the following SQL statement is entered:

```
DELETE FROM Emp_tab WHERE Empno = 7499;
```

An error is returned because the table is mutating when the row is deleted:

```
ORA-04091: table SCOTT.Emp_tab is mutating, trigger/function may not see it
```

If you delete the line "FOR EACH ROW" from the trigger, it becomes a statement trigger which is not subject to this restriction, and the trigger.

If you need to update a mutating table, you could bypass these restrictions by using a temporary table, a PL/SQL table, or a package variable. For example, in place of a single AFTER row trigger that updates the original table, resulting in a mutating table error, you might use two triggers—an AFTER row trigger that updates a temporary table, and an AFTER statement trigger that updates the original table with the values from the temporary table.

Declarative integrity constraints are checked at various times with respect to row triggers.

**See Also:** *Oracle9i Database Concepts* has information about the interaction of triggers and integrity constraints.

Because declarative referential integrity constraints are currently not supported between tables on different nodes of a distributed database, the mutating table restrictions do not apply to triggers that access remote nodes. These restrictions are also not enforced among 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.

Do not use loop-back database links to circumvent the trigger restrictions. Such applications might behave unpredictably.

### **Restrictions on Mutating Tables Relaxed**

Before Oracle8i, there was a "constraining error" that prevented a row trigger from modifying a table when the parent statement implicitly read that table to enforce a foreign key constraint. Starting with Oracle8i, there is no constraining error. Also, checking of the foreign key is deferred until at least the end of the parent statement.

The mutating error still prevents the trigger from reading or modifying the table that the parent statement is modifying. However, starting in Oracle release 8.1, a delete against the parent table causes before/after statement triggers to be fired once. That way, you can create triggers (just not row triggers) to read and modify the parent and child tables.

This allows most foreign key constraint actions to be implemented through their obvious after-row trigger, providing the constraint is not 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. For example, this is an implementation of update cascade:

```
create table p (p1 number constraint ppk primary key);
create table f (f1 number constraint ffk references p);
create trigger pt after update on p for each row begin
    update f set f1 = :new.p1 where f1 = :old.p1;
end;
/
```

This implementation requires care for multirow updates. For example, if a table p has three rows with the values (1), (2), (3), and table f also has three rows with the values (1), (2), (3), then the following statement updates p correctly but causes problems when the trigger updates f:

```
update p set p1 = p1+1;
```

The statement first updates (1) to (2) in p, and the trigger updates (1) to (2) in f, leaving two rows of value (2) in f. Then the 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 of the data in p and f is lost.

To avoid this problem, you must forbid multirow updates to p that change the primary key and reuse existing primary key values. It could also be solved by tracking which foreign key values have already been updated, then modifying the trigger so that no row is updated twice.

That is the only problem with this technique for foreign key updates. The trigger cannot miss rows that have been 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 called.

### System Trigger Restrictions

Depending on the event, different event attribute functions are available. For example, certain DDL operations may not be allowed on DDL events. Check "[Event Attribute Functions](#)" on page 16-2 before using an event attribute function, because its effects might be undefined rather than producing an error condition.

Only committed triggers are fired. For example, if you create a trigger that should be fired after all CREATE events, then the trigger itself does not fire after the creation, because the correct information about this trigger was not committed at the time when the trigger on CREATE events was fired. On the other hand, if you

DROP a trigger that should be fired *before* all DROP events, then the trigger fires before the DROP.

For example, if you execute the following SQL statement:

```
CREATE OR REPLACE TRIGGER Foo AFTER CREATE ON DATABASE
BEGIN null;
END;
```

Then, trigger `foo` is not fired after the creation of `foo`. Oracle does not fire a trigger that is not committed.

### Foreign Function Callouts

All restrictions on foreign function callouts also apply.

## Who Is the Trigger User?

The following statement, inside a trigger, returns the owner of the trigger, not the name of user who is updating the table:

```
SELECT Username FROM USER_USERS;
```

## Privileges Needed to Work with Triggers

To create a trigger in your schema, you must have the `CREATE TRIGGER` system privilege, and either:

- Own the table specified in the triggering statement, or
- Have the `ALTER` privilege for the table in the triggering statement, or
- Have the `ALTER ANY TABLE` system privilege

To create a trigger in another user's schema, or to reference a table in another schema from a trigger in your schema, you must have the `CREATE ANY TRIGGER` system privilege. With this privilege, the trigger can be created in any schema and can be associated with any user's table. In addition, the user creating the trigger must also have `EXECUTE` privilege on the referenced procedures, functions, or packages.

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

The object privileges to the schema objects referenced in the trigger body must be granted to the trigger's owner explicitly (not through a role). The statements in the trigger body operate under the privilege domain of the trigger's owner, not the privilege domain of the user issuing the triggering statement. This is similar to the privilege model for stored procedures.

## Compiling Triggers

Triggers are similar to PL/SQL anonymous blocks with the addition of the `:new` and `:old` capabilities, but their compilation is different. A PL/SQL anonymous block is compiled each time it is loaded into memory. Compilation involves three stages:

1. Syntax checking: PL/SQL syntax is checked, and a parse tree is generated.
2. Semantic checking: Type checking and further processing on the parse tree.
3. Code generation: The pcode is generated.

Triggers, in contrast, are fully compiled when the `CREATE TRIGGER` statement is entered, and the pcode is stored in the data dictionary. Hence, firing the trigger no longer requires the opening of a shared cursor to run the trigger action. Instead, the trigger is executed directly.

If errors occur during the compilation of a trigger, then the trigger is still created. If a DML statement fires this trigger, then the DML statement fails. (Runtime trigger errors always cause the DML statement to fail.) You can use the `SHOW ERRORS` statement in SQL\*Plus or Enterprise Manager to see any compilation errors when you create a trigger, or you can `SELECT` the errors from the `USER_ERRORS` view.

## Dependencies for Triggers

Compiled triggers have dependencies. They become invalid if a depended-on object, such as a stored procedure or function called from the trigger body, is modified. Triggers that are invalidated for dependency reasons are recompiled when next invoked.

You can examine the `ALL_DEPENDENCIES` view to see the dependencies for a trigger. For example, the following statement shows the dependencies for the triggers in the `SCOTT` schema:

```
SELECT NAME, REFERENCED_OWNER, REFERENCED_NAME, REFERENCED_TYPE
FROM ALL_DEPENDENCIES
WHERE OWNER = 'SCOTT' and TYPE = 'TRIGGER';
```

Triggers may depend on other functions or packages. If the function or package specified in the trigger is dropped, then the trigger is marked invalid. An attempt is made to validate the trigger on occurrence of the event. If the trigger cannot be validated successfully, then it is marked `VALID WITH ERRORS`, and the event fails.

---

---

**Note:**

- There is an exception for `STARTUP` events: `STARTUP` events succeed even if the trigger fails. There are also exceptions for `SHUTDOWN` events and for `LOGON` events if you login as `SYSTEM`.
  - Because the `DBMS_AQ` package is used to enqueue a message, dependency between triggers and queues cannot be maintained.
- 
- 

## Recompiling Triggers

Use the `ALTER TRIGGER` statement to recompile a trigger manually. For example, the following statement recompiles the `PRINT_SALARY_CHANGES` trigger:

```
ALTER TRIGGER Print_salary_changes COMPILE;
```

To recompile a trigger, you must own the trigger or have the `ALTER ANY TRIGGER` system privilege.

## Migration Issues for Triggers

Non-compiled triggers cannot be fired under compiled trigger releases (such as Oracle 7.3 and Oracle8). If you are upgrading from a non-compiled trigger release to a compiled trigger release, then all existing triggers must be compiled. The upgrade script `cat73xx.sql` invalidates all triggers, so that they are automatically recompiled when first run. (The `xx` stands for a variable minor release number.)

Downgrading from Oracle 7.3 or later to a release prior to 7.3 requires that you run the `cat73xxd.sql` downgrade script. This handles portability issues between stored and non-stored trigger releases.

## Modifying Triggers

Like a stored procedure, a trigger cannot be explicitly altered: It must be replaced with a new definition. (The `ALTER TRIGGER` statement is used only to recompile, enable, or disable a trigger.)

When replacing a trigger, you must include the `OR REPLACE` option in the `CREATE TRIGGER` statement. The `OR REPLACE` option is provided to allow a new version of an existing trigger to replace the older version, without affecting any grants made for the original version of the trigger.

Alternatively, the trigger can be dropped using the `DROP TRIGGER` statement, and you can rerun the `CREATE TRIGGER` statement.

To drop a trigger, the trigger must be in your schema, or you must have the `DROP ANY TRIGGER` system privilege.

## Debugging Triggers

You can debug a trigger using the same facilities available for stored procedures.

**See Also:** ["Debugging Stored Procedures"](#) on page 9-41

## Enabling and Disabling Triggers

A trigger can be in one of two distinct modes:

**Enabled.** An enabled trigger executes its trigger body if a triggering statement is entered and the trigger restriction (if any) evaluates to `TRUE`.

**Disabled.** A disabled trigger does not execute its trigger body, even if a triggering statement is entered and the trigger restriction (if any) evaluates to `TRUE`.

## Enabling Triggers

By default, a trigger is automatically enabled when it is created; however, it can later be disabled. After you have completed the task that required the trigger to be disabled, re-enable the trigger, so that it fires when appropriate.

Enable a disabled trigger using the `ALTER TRIGGER` statement with the `ENABLE` option. To enable the disabled trigger named `REORDER` of the `INVENTORY` table, enter the following statement:

```
ALTER TRIGGER Reorder ENABLE;
```

All triggers defined for a specific table can be enabled with one statement using the `ALTER TABLE` statement with the `ENABLE` clause with the `ALL TRIGGERS` option. For example, to enable all triggers defined for the `INVENTORY` table, enter the following statement:

```
ALTER TABLE Inventory
  ENABLE ALL TRIGGERS;
```

### Disabling Triggers

You might temporarily disable a trigger if:

- An object it references is not available.
- You need to perform a large data load, and you want it to proceed quickly without firing triggers.
- You are reloading data.

By default, triggers are enabled when first created. Disable a trigger using the `ALTER TRIGGER` statement with the `DISABLE` option.

For example, to disable the trigger named `REORDER` of the `INVENTORY` table, enter the following statement:

```
ALTER TRIGGER Reorder DISABLE;
```

All triggers associated with a table can be disabled with one statement using the `ALTER TABLE` statement with the `DISABLE` clause and the `ALL TRIGGERS` option. For example, to disable all triggers defined for the `INVENTORY` table, enter the following statement:

```
ALTER TABLE Inventory
  DISABLE ALL TRIGGERS;
```

### Viewing Information About Triggers

The following data dictionary views reveal information about triggers:

- `USER_TRIGGERS`
- `ALL_TRIGGERS`
- `DBA_TRIGGERS`

The new column, `BASE_OBJECT_TYPE`, specifies whether the trigger is based on `DATABASE`, `SCHEMA`, table, or view. The old column, `TABLE_NAME`, is null if the base object is not table or view.

The column `ACTION_TYPE` specifies whether the trigger is a call type trigger or a PL/SQL trigger.

The column `TRIGGER_TYPE` includes two additional values: `BEFORE EVENT` and `AFTER EVENT`, applicable only to system events.

The column `TRIGGERING_EVENT` includes all system and DML events.

**See Also:** The *Oracle9i Database Reference* provides a complete description of these data dictionary views.

For example, assume the following statement was used to create the `REORDER` trigger:

---



---

**Caution:** You may need to set up data structures for certain examples to work:

---



---

```
CREATE OR REPLACE TRIGGER Reorder
AFTER UPDATE OF Parts_on_hand ON Inventory
FOR EACH ROW
WHEN(new.Parts_on_hand < new.Reorder_point)
DECLARE
    x NUMBER;
BEGIN
    SELECT COUNT(*) INTO x
    FROM Pending_orders
    WHERE Part_no = :new.Part_no;
    IF x = 0 THEN
        INSERT INTO Pending_orders
        VALUES (:new.Part_no, :new.Reorder_quantity,
                sysdate);
    END IF;
END;
```

The following two queries return information about the `REORDER` trigger:

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

```
TYPE          TRIGGERING_STATEMENT  TABLE_NAME
-----
AFTER EACH ROW  UPDATE              INVENTORY

SELECT Trigger_body
       FROM USER_TRIGGERS
       WHERE Trigger_name = 'REORDER';

TRIGGER_BODY
-----
DECLARE
  x NUMBER;
BEGIN
  SELECT COUNT(*) INTO x
     FROM Pending_orders
     WHERE Part_no = :new.Part_no;
  IF x = 0
     THEN INSERT INTO Pending_orders
           VALUES (:new.Part_no, :new.Reorder_quantity,
                 sysdate);
  END IF;
END;
```

## Examples of Trigger Applications

You can use triggers in a number of ways to customize information management in an Oracle database. For example, triggers are commonly used to:

- Provide sophisticated auditing
- Prevent invalid transactions
- Enforce referential integrity (either those actions not supported by declarative integrity constraints or across nodes in a distributed database)
- Enforce complex business rules
- Enforce complex security authorizations
- Provide transparent event logging
- Automatically generate derived column values
- Enable building complex views that are updatable
- Track system events



This section provides an example of each of these trigger applications. These examples are not meant to be used exactly as written: They are provided to assist you in designing your own triggers.

### Auditing with Triggers: Example

Triggers are commonly used to supplement the built-in auditing features of Oracle. Although triggers can be written to record information similar to that recorded by the `AUDIT` statement, triggers should be used only when more detailed audit information is required. For example, use triggers to provide value-based auditing for each row.

Sometimes, the Oracle `AUDIT` statement is considered a *security* audit facility, while triggers can provide *financial* audit facility.

When deciding whether to create a trigger to audit database activity, consider what Oracle's auditing features provide, compared to auditing defined by triggers.

Audit Feature	Comparison of Built-In Auditing and Trigger-Based Auditing
DML and DDL Auditing	Standard auditing options permit auditing of DML and DDL statements regarding all types of schema objects and structures. Comparatively, <i>triggers</i> permit auditing of DML statements entered against tables, and DDL auditing at <code>SCHEMA</code> or <code>DATABASE</code> level.
Centralized Audit Trail	All database audit information is recorded centrally and automatically using the auditing features of Oracle.
Declarative Method	Auditing features enabled using the standard Oracle features are easier to declare and maintain, and less prone to errors, when compared to auditing functions defined by triggers.
Auditing Options can be Audited	Any changes to existing auditing options can also be audited to guard against malicious database activity.
Session and Execution time Auditing	Using the database auditing features, records can be generated once every time an audited statement is entered ( <code>BY ACCESS</code> ) or once for every session that enters an audited statement ( <code>BY SESSION</code> ). Triggers cannot audit by session; an audit record is generated each time a trigger-audited table is referenced.

Audit Feature	Comparison of Built-In Auditing and Trigger-Based Auditing
Auditing of Unsuccessful Data Access	Database auditing can be set to audit when unsuccessful data access occurs. However, unless autonomous transactions are used, any audit information generated by a trigger is rolled back if the triggering statement is rolled back. For more information on autonomous transactions, see <i>Oracle9i Database Concepts</i> .
Sessions can be Audited	Connections and disconnections, as well as session activity (physical I/Os, logical I/Os, deadlocks, and so on), can be recorded using standard database auditing.

When using triggers to provide sophisticated auditing, **AFTER** triggers are normally used. By using **AFTER** triggers, auditing information is recorded after the triggering statement is subjected to any applicable integrity constraints, preventing cases where the audit processing is carried out unnecessarily for statements that generate exceptions to integrity constraints.

Choosing between **AFTER** row and **AFTER** statement triggers depends on the information being audited. For example, row triggers provide value-based auditing for each table row. Triggers can also require the user to supply a "reason code" for issuing the audited SQL statement, which can be useful in both row and statement-level auditing situations.

The following example demonstrates a trigger that audits modifications to the `Emp_tab` table for each row. It requires that a "reason code" be stored in a global package variable before the update. This shows how triggers can be used to provide value-based auditing and how to use public package variables.

**Note:** You may need to set up the following data structures for the examples to work:

```
CREATE OR REPLACE PACKAGE Auditpackage AS
  Reason VARCHAR2(10);
PROCEDURE Set_reason(Reason VARCHAR2);
END;
CREATE TABLE Emp99 (
  Empno          NOT NULL   NUMBER(4)
  Ename          VARCHAR2(10)
  Job            VARCHAR2(9)
  Mgr            NUMBER(4)
  Hiredate      DATE
  Sal            NUMBER(7,2)
  Comm           NUMBER(7,2)
  Deptno        NUMBER(2)
  Bonus         NUMBER
  Ssn            NUMBER
  Job_classification NUMBER);

CREATE TABLE Audit_employee (
  Oldssn        NUMBER
  Oldname       VARCHAR2(10)
  Oldjob        VARCHAR2(2)
  Oldsal        NUMBER
  Newssn        NUMBER
  Newname       VARCHAR2(10)
  Newjob        VARCHAR2(2)
  Newsal        NUMBER
  Reason        VARCHAR2(10)
  User1         VARCHAR2(10)
  Systemdate    DATE);
```

---

## Examples of Trigger Applications

---

```
CREATE OR REPLACE TRIGGER Audit_employee
AFTER INSERT OR DELETE OR UPDATE ON Emp99
FOR EACH ROW
BEGIN
/* AUDITPACKAGE is a package with a public package
variable REASON. REASON could be set by the
application by a command such as EXECUTE
AUDITPACKAGE.SET_REASON(reason_string). Note that a
package variable has state for the duration of a
session and that each session has a separate copy of
all package variables. */

IF Auditpackage.Reason IS NULL THEN
    Raise_application_error(-20201, 'Must specify reason'
        || ' with AUDITPACKAGE.SET_REASON(Reason_string)');
END IF;

/* If the above conditional evaluates to TRUE, the
user-specified error number and message is raised,
the trigger stops execution, and the effects of the
triggering statement are rolled back. Otherwise, a
new row is inserted into the predefined auditing
table named AUDIT_EMPLOYEE containing the existing
and new values of the Emp_tab table and the reason code
defined by the REASON variable of AUDITPACKAGE. Note
that the "old" values are NULL if triggering
statement is an INSERT and the "new" values are NULL
if the triggering statement is a DELETE. */

INSERT INTO Audit_employee VALUES
(:old.Ssn, :old.Ename, :old.Job_classification, :old.Sal,
:new.Ssn, :new.Ename, :new.Job_classification, :new.Sal,
auditpackage.Reason, User, Sysdate );
END;
```

Optionally, you can also set the reason code back to NULL if you wanted to force the reason code to be set for every update. The following simple AFTER statement trigger sets the reason code back to NULL after the triggering statement is run:

```
CREATE OR REPLACE TRIGGER Audit_employee_reset
AFTER INSERT OR DELETE OR UPDATE ON Emp_tab
BEGIN
    auditpackage.set_reason(NULL);
END;
```

Notice that the previous two triggers are both fired by the same type of SQL statement. However, the AFTER row trigger is fired once for each row of the table affected by the triggering statement, while the AFTER statement trigger is fired only once after the triggering statement execution is completed.

This next trigger also uses triggers to do auditing. It tracks changes made to the Emp\_tab table and stores this information in AUDIT\_TABLE and AUDIT\_TABLE\_VALUES.

---

---

**Note:** You may need to set up the following data structures for the example to work:

```
CREATE TABLE Audit_table (
  Seq      NUMBER,
  User_at  VARCHAR2(10),
  Time_now DATE,
  Term     VARCHAR2(10),
  Job      VARCHAR2(10),
  Proc     VARCHAR2(10),
  enum     NUMBER);
CREATE SEQUENCE Audit_seq;
CREATE TABLE Audit_table_values (
  Seq      NUMBER,
  Dept     NUMBER,
  Dept1    NUMBER,
  Dept2    NUMBER);
```

---

---

```
CREATE OR REPLACE TRIGGER Audit_emp
AFTER INSERT OR UPDATE OR DELETE ON Emp_tab
FOR EACH ROW
DECLARE
  Time_now DATE;
  Terminal CHAR(10);
BEGIN
  -- get current time, and the terminal of the user:
  Time_now := SYSDATE;
  Terminal := USERENV('TERMINAL');
  -- record new employee primary key
  IF INSERTING THEN
    INSERT INTO Audit_table
      VALUES (Audit_seq.NEXTVAL, User, Time_now,
              Terminal, 'Emp_tab', 'INSERT', :new.Empno);
  -- record primary key of the deleted row:
  ELSIF DELETING THEN
```

```
INSERT INTO Audit_table
VALUES (Audit_seq.NEXTVAL, User, Time_now,
Terminal, 'Emp_tab', 'DELETE', :old.Empno);
-- for updates, record the primary key
-- of the row being updated:
ELSE
INSERT INTO Audit_table
VALUES (audit_seq.NEXTVAL, User, Time_now,
Terminal, 'Emp_tab', 'UPDATE', :old.Empno);
-- and for SAL and DEPTNO, record old and new values:
IF UPDATING ('SAL') THEN
INSERT INTO Audit_table_values
VALUES (Audit_seq.CURRVAL, 'SAL',
:old.Sal, :new.Sal);

ELSIF UPDATING ('DEPTNO') THEN
INSERT INTO Audit_table_values
VALUES (Audit_seq.CURRVAL, 'DEPTNO',
:old.Deptno, :new.DEPTNO);
END IF;
END IF;
END;
```

### Integrity Constraints and Triggers: Examples

Triggers and declarative integrity constraints can both be used to constrain data input. However, triggers and integrity constraints have significant differences.

Declarative integrity constraints are statements about the database that are always true. A constraint applies to existing data in the table and any statement that manipulates the table.

**See Also:** [Chapter 4, "Maintaining Data Integrity Through Constraints"](#)

Triggers constrain what a transaction can do. A trigger does not apply to data loaded before the definition of the trigger; therefore, it is not known if all data in a table conforms to the rules established by an associated trigger.

Although triggers can be written to enforce many of the same rules supported by Oracle's declarative integrity constraint features, triggers should only be used to enforce complex business rules that cannot be defined using standard integrity constraints. The declarative integrity constraint features provided with Oracle offer the following advantages when compared to constraints defined by triggers:

**Centralized integrity checks.** All points of data access must adhere to the global set of rules defined by the integrity constraints corresponding to each schema object.

**Declarative method.** Constraints defined using the standard integrity constraint features are much easier to write and are less prone to errors, when compared with comparable constraints defined by triggers.

While most aspects of data integrity can be defined and enforced using declarative integrity constraints, triggers can be used to enforce complex business constraints not definable using declarative integrity constraints. For example, triggers can be used to enforce:

- `UPDATE` and `DELETE SET NULL`, and `UPDATE` and `DELETE SET DEFAULT` referential actions.
- Referential integrity when the parent and child tables are on different nodes of a distributed database.
- Complex check constraints not definable using the expressions allowed in a `CHECK` constraint.

### Referential Integrity Using Triggers

Many cases of referential integrity can be enforced using triggers. However, only use triggers when you want to enforce the `UPDATE` and `DELETE SET NULL` (when referenced data is updated or deleted, all associated dependent data is set to `NULL`), and `UPDATE` and `DELETE SET DEFAULT` (when referenced data is updated or deleted, all associated dependent data is set to a default value) referential actions, or when you want to enforce referential integrity between parent and child tables on different nodes of a distributed database.

When using triggers to maintain referential integrity, declare the `PRIMARY` (or `UNIQUE`) `KEY` constraint in the parent table. If referential integrity is being maintained between a parent and child table in the same database, then you can also declare the foreign key in the child table, but disable it; this prevents the corresponding `PRIMARY KEY` constraint from being dropped (unless the `PRIMARY KEY` constraint is explicitly dropped with the `CASCADE` option).

To maintain referential integrity using triggers:

- A trigger must be defined for the child table that guarantees values inserted or updated in the foreign key correspond to values in the parent key.
- One or more triggers must be defined for the parent table. These triggers guarantee the desired referential action (`RESTRICT`, `CASCADE`, or `SET NULL`) for values in the foreign key when values are updated or deleted in the parent key.

No action is required for inserts into the parent table (no dependent foreign keys exist).

The following sections provide examples of the triggers necessary to enforce referential integrity. The Emp\_tab and Dept\_tab table relationship is used in these examples.

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

**Foreign Key Trigger for Child Table** The following trigger guarantees that before an INSERT or UPDATE statement affects a foreign key value, the corresponding value exists in the parent key. The mutating table exception included in the following example allows this trigger to be used with the UPDATE\_SET\_DEFAULT and UPDATE\_CASCADE triggers. This exception can be removed if this trigger is used alone.

```
CREATE OR REPLACE TRIGGER Emp_dept_check
BEFORE INSERT OR UPDATE OF Deptno ON Emp_tab
FOR EACH ROW WHEN (new.Deptno IS NOT NULL)

-- Before a row is inserted, or DEPTNO is updated in the Emp_tab
-- table, fire this trigger to verify that the new foreign
-- key value (DEPTNO) is present in the Dept_tab table.
DECLARE
    Dummy                INTEGER; -- used for cursor fetch below
    Invalid_department   EXCEPTION;
    Valid_department    EXCEPTION;
    Mutating_table       EXCEPTION;
    PRAGMA EXCEPTION_INIT (Mutating_table, -4091);

-- Cursor used to verify parent key value exists.  If
-- present, lock parent key's row so it can't be
-- deleted by another transaction until this
-- transaction is committed or rolled back.
CURSOR Dummy_cursor (Dn NUMBER) IS
    SELECT Deptno FROM Dept_tab
        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 number 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 following trigger is defined on the DEPT\_TAB table to enforce the UPDATE and DELETE RESTRICT referential action on the primary key of the DEPT\_TAB table:

```

CREATE OR REPLACE TRIGGER Dept_restrict
BEFORE DELETE OR UPDATE OF Deptno ON Dept_tab
FOR EACH ROW

-- Before a row is deleted from Dept_tab or the primary key
-- (DEPTNO) of Dept_tab is updated, check for dependent
-- foreign key values in Emp_tab; rollback if any are found.
DECLARE
    Dummy                INTEGER;        -- used for cursor fetch below
    Employees_present    EXCEPTION;
    employees_not_present EXCEPTION;

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

BEGIN
    OPEN Dummy_cursor (:old.Deptno);
    FETCH Dummy_cursor INTO Dummy;
    -- If dependent foreign key is found, raise user-specified
    -- error number and message. If not found, close cursor
    -- before allowing triggering statement to complete.

```

## Examples of Trigger Applications

---

```
IF Dummy_cursor%FOUND THEN
    RAISE Employees_present;    -- dependent rows exist
ELSE
    RAISE Employees_not_present; -- no dependent rows
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;
```

---

**Caution:** This trigger 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, A fires B fires A).

---

**UPDATE and DELETE SET NULL Triggers for Parent Table: Example** The following trigger is defined on the DEPT\_TAB table to enforce the UPDATE and DELETE SET NULL referential action on the primary key of the DEPT\_TAB table:

```
CREATE OR REPLACE TRIGGER Dept_set_null
AFTER DELETE OR UPDATE OF Deptno ON Dept_tab
FOR EACH ROW

-- Before a row is deleted from Dept_tab or the primary key
-- (DEPTNO) of Dept_tab is updated, set all corresponding
-- dependent foreign key values in Emp_tab to NULL:
BEGIN
    IF UPDATING AND :OLD.Deptno != :NEW.Deptno OR DELETING THEN
        UPDATE Emp_tab SET Emp_tab.Deptno = NULL
            WHERE Emp_tab.Deptno = :old.Deptno;
    END IF;
END;
```

**DELETE Cascade Trigger for Parent Table: Example** The following trigger on the DEPT\_TAB table enforces the DELETE CASCADE referential action on the primary key of the DEPT\_TAB table:

```
CREATE OR REPLACE TRIGGER Dept_del_cascade
```

```

AFTER DELETE ON Dept_tab
FOR EACH ROW

-- Before a row is deleted from Dept_tab, delete all
-- rows from the Emp_tab table whose DEPTNO is the same as
-- the DEPTNO being deleted from the Dept_tab table:
BEGIN
  DELETE FROM Emp_tab
  WHERE Emp_tab.Deptno = :old.Deptno;
END;

```

---

**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.

---

**UPDATE Cascade Trigger for Parent Table: Example** The following trigger ensures that if a department number is updated in the Dept\_tab table, then this change is propagated to dependent foreign keys in the Emp\_tab table:

```

-- Generate a sequence number to be used as a flag for
-- determining if an update has occurred on a column:
CREATE SEQUENCE Update_sequence
  INCREMENT BY 1 MAXVALUE 5000
  CYCLE;

CREATE OR REPLACE PACKAGE Integritypackage AS
  Updateseq NUMBER;
END Integritypackage;

CREATE OR REPLACE PACKAGE BODY Integritypackage AS
END Integritypackage;
-- create flag col:
ALTER TABLE Emp_tab ADD Update_id NUMBER;

CREATE OR REPLACE TRIGGER Dept_cascade1 BEFORE UPDATE OF Deptno ON Dept_tab
DECLARE
  Dummy NUMBER;

-- Before updating the Dept_tab table (this is a statement
-- trigger), generate a new sequence number and assign
-- it to the public variable UPDATESEQ of a user-defined
-- package named INTEGRITYPACKAGE:
BEGIN
  SELECT Update_sequence.NEXTVAL

```

## Examples of Trigger Applications

---

```
        INTO Dummy
        FROM dual;
    Integritypackage.Updateseq := Dummy;
END;

CREATE OR REPLACE TRIGGER Dept_cascade2 AFTER DELETE OR UPDATE
    OF Deptno ON Dept_tab FOR EACH ROW

-- For each department number in Dept_tab that is updated,
-- cascade the update to dependent foreign keys in the
-- Emp_tab table. Only cascade the update if the child row
-- has not already been updated by this trigger:
BEGIN
    IF UPDATING THEN
        UPDATE Emp_tab
            SET Deptno = :new.Deptno,
                Update_id = Integritypackage.Updateseq --from 1st
        WHERE Emp_tab.Deptno = :old.Deptno
            AND Update_id IS NULL;
        /* only NULL if not updated by the 3rd trigger
           fired by this same triggering statement */
    END IF;
    IF DELETING THEN

        -- Before a row is deleted from Dept_tab, delete all
        -- rows from the Emp_tab table whose DEPTNO is the same as
        -- the DEPTNO being deleted from the Dept_tab table:
        DELETE FROM Emp_tab
            WHERE Emp_tab.Deptno = :old.Deptno;
    END IF;
END;

CREATE OR REPLACE TRIGGER Dept_cascade3 AFTER UPDATE OF Deptno ON Dept_tab
BEGIN UPDATE Emp_tab
    SET Update_id = NULL
    WHERE Update_id = Integritypackage.Updateseq;
END;
```

---

---

**Note:** Because this trigger updates the Emp\_tab table, the Emp\_dept\_check trigger, if enabled, is also fired. The resulting mutating table error is trapped by the Emp\_dept\_check trigger. You should carefully test any triggers that require error trapping to succeed to ensure that they always work properly in your environment.

---

---

### Trigger for Complex Check Constraints: Example

Triggers can enforce integrity rules other than referential integrity. For example, this trigger performs a complex check before allowing the triggering statement to run.

---

**Note:** You may need to set up the following data structures for the example to work:

---

```
CREATE TABLE Salgrade (
  Grade          NUMBER,
  Losal          NUMBER,
  Hisal          NUMBER,
  Job_classification  NUMBER)
```

---

```
CREATE OR REPLACE TRIGGER Salary_check
BEFORE INSERT OR UPDATE OF Sal, Job ON Emp99
FOR EACH ROW
DECLARE
  Minsal          NUMBER;
  Maxsal          NUMBER;
  Salary_out_of_range  EXCEPTION;
BEGIN

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

  SELECT Minsal, Maxsal INTO Minsal, Maxsal FROM Salgrade
  WHERE Job_classification = :new.Job;

  /* If the employee's new salary is less than or greater
  than the job classification's limits, the exception is
  raised. The exception message is returned and the
  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 '
        || :new.Job_classification);
END;
```

### Complex Security Authorizations and Triggers: Example

Triggers are commonly used to enforce complex security authorizations for table data. Only use triggers to enforce complex security authorizations that cannot be defined using the database security features provided with Oracle. For example, a trigger can prohibit updates to salary data of the `Emp_tab` table during weekends, holidays, and non-working 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 performed only once for the triggering statement, not for each row affected by the triggering statement.

This example shows a trigger used to enforce security.

---

---

**Note:** You may need to set up the following data structures for the example to work:

```
CREATE TABLE Company_holidays (Day DATE);
```

---

---

```
CREATE OR REPLACE TRIGGER Emp_permit_changes
BEFORE INSERT OR DELETE OR UPDATE ON Emp99
DECLARE
    Dummy          INTEGER;
    Not_on_weekends EXCEPTION;
    Not_on_holidays EXCEPTION;
    Non_working_hours EXCEPTION;
BEGIN
    /* check for weekends: */
    IF (TO_CHAR(Sysdate, 'DY') = 'SAT' OR
        TO_CHAR(Sysdate, 'DY') = 'SUN') THEN
        RAISE Not_on_weekends;
    END IF;
```

```

/* check for company holidays:*/
SELECT COUNT(*) INTO Dummy FROM Company_holidays
WHERE TRUNC(Day) = TRUNC(Sysdate);
/* TRUNC gets rid of time parts of dates: */
IF dummy > 0 THEN
RAISE Not_on_holidays;
END IF;
/* Check for work hours (8am to 6pm): */
IF (TO_CHAR(Sysdate, 'HH24') < 8 OR
TO_CHAR(Sysdate, 'HH24') > 18) THEN
RAISE Non_working_hours;
END IF;
EXCEPTION
WHEN Not_on_weekends THEN
Raise_application_error(-20324,'May not change '
||'employee table during the weekend');
WHEN Not_on_holidays THEN
Raise_application_error(-20325,'May not change '
||'employee table during a holiday');
WHEN Non_working_hours THEN
Raise_application_error(-20326,'May not change '
||'Emp_tab table during non-working hours');
END;

```

### Transparent Event Logging and Triggers

Triggers are very useful when you want to transparently perform 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.)

### Derived Column Values and Triggers: Example

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 the following 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 following example illustrates how a trigger can be used to derive new column values for a table whenever a row is inserted or updated.

---

---

**Note:** You may need to set up the following data structures for the example to work:

```
ALTER TABLE Emp99 ADD(  
    Uppername   VARCHAR2(20),  
    Soundexname VARCHAR2(20));
```

---

---

```
CREATE OR REPLACE TRIGGER Derived  
BEFORE INSERT OR UPDATE OF Ename ON Emp99  
  
/* Before updating the ENAME field, derive the values for  
   the UPPERNAME and SOUNDEXNAME fields. Users should be  
   restricted from updating these fields directly: */  
FOR EACH ROW  
BEGIN  
    :new.Uppername := UPPER(:new.Ename);  
    :new.Soundexname := SOUNDEX(:new.Ename);  
END;
```

### Building Complex Updatable Views Using Triggers: Example

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 under their respective titles. The library consists of a collection of book type objects. The following example explains the schema.



```

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;

```

Assume that the following tables exist in the relational schema:

Table Book\_table (Booknum, Section, Title, Author, Available)

Booknum	Section	Title	Author	Available
121001	Classic	Iliad	Homer	Y
121002	Novel	Gone With the Wind	Mitchell M	N

Library consists of library\_table(section).

### Section

Geography

Classic

Now you can define a complex view over these tables 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;

```

Make this view updatable by defining an INSTEAD OF trigger over the view.

```

CREATE OR REPLACE TRIGGER Library_trigger INSTEAD OF INSERT ON Library_view FOR
EACH ROW
  Bookvar BOOK_T;
  i      INTEGER;

```

```
BEGIN
  INSERT INTO Library_table VALUES (:NEW.Section);
  FOR i IN 1..:NEW.Booklist.COUNT LOOP
    Bookvar := Booklist(i);
    INSERT INTO book_table
      VALUES ( Bookvar.booknum, :NEW.Section, Bookvar.Title, Bookvar.Author,
bookvar.Available);
  END LOOP;
END;
/
```

Now, the `library_view` is an updatable view, and any `INSERTs` on the view are handled by the trigger that gets fired automatically. For example:

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

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

### Tracking System Events Using Triggers

**Fine-Grained Access Control Using Triggers: Example** System triggers can be used to set application context. Application context is a relatively new feature that enhances your ability to implement fine-grained access control. Application context is a secure session cache, and it can be used to store session-specific attributes.

In the example that follows, procedure `set_ctx` sets the application context based on the user profile. The trigger `setexpensectx` ensures that the context is set for every user.

```
CONNECT secdemo/secdemo
```

```
CREATE OR REPLACE CONTEXT Expenses_reporting USING Secdemo.Exprep_ctx;
```

```
REM =====
REM Creation of the package which implements the context:
REM =====
```

```
CREATE OR REPLACE PACKAGE Exprep_ctx AS
  PROCEDURE Set_ctx;
END;
```

```
SHOW ERRORS
```

```
CREATE OR REPLACE PACKAGE BODY Exprep_ctx IS
  PROCEDURE Set_ctx IS
    Empnum    NUMBER;
    Countrec  NUMBER;
    Cc        NUMBER;
    Role      VARCHAR2(20);
  BEGIN

    -- SET emp_number:
    SELECT Employee_id INTO Empnum FROM Employee
       WHERE Last_name = SYS_CONTEXT('userenv', 'session_user');

    DBMS_SESSION.SET_CONTEXT('expenses_reporting', 'emp_number', Empnum);

    -- SET ROLE:
    SELECT COUNT (*) INTO Countrec FROM Cost_center WHERE Manager_id=Empnum;
    IF (countrec > 0) THEN
      DBMS_SESSION.SET_CONTEXT('expenses_reporting', 'exp_role', 'MANAGER');
    ELSE
      DBMS_SESSION.SET_CONTEXT('expenses_reporting', 'exp_role', 'EMPLOYEE');
    END IF;

    -- SET cc_number:
    SELECT Cost_center_id INTO Cc FROM Employee
       WHERE Last_name = SYS_CONTEXT('userenv', 'session_user');
    DBMS_SESSION.SET_CONTEXT('expenses_reporting', 'cc_number', Cc);
  END;
END;
```

### CALL Syntax

```
CREATE OR REPLACE TRIGGER Secdemo.Setexpseetx
AFTER LOGON ON DATABASE
CALL Secdemo.Exprep_ctx.Set_ctx
```

## Responding to System Events through Triggers

Oracle's system event publication lets applications subscribe to database events, just like they subscribe to messages from other applications.

**See Also:** [Chapter 16, "Working With System Events"](#)

Oracle's system events publication framework includes the following features:

- Infrastructure for publish/subscribe, by making the database an active publisher of events.
- Integration of data cartridges in the server. The system events publication can be used to notify cartridges of state changes in the server.
- Integration of fine-grained access control in the server.

By creating a trigger, you can specify a procedure that runs when an event occurs. DML events are supported on tables, and system events are supported on `DATABASE` and `SCHEMA`. You can turn notification on and off by enabling and disabling the trigger using the `ALTER TRIGGER` statement.

This feature is integrated with the Advanced Queueing engine. Publish/subscribe applications use the `DBMS_AQ.ENQUEUE()` procedure, and other applications such as cartridges use callouts.

**See Also:** *Oracle9i SQL Reference*

For details on how to subscribe to published events, see *Oracle9i Application Developer's Guide - Advanced Queuing*

### How Events Are Published Through Triggers

When events are detected by the server, the trigger mechanism executes the action specified in the trigger. As part of this action, you can use the `DBMS_AQ` package to publish the event to a queue, so that subscribers get notifications.

---

---

**Note:** Only a built-in set of events can be detected this way. You cannot define your own event conditions.

---

---

When an event occurs, all triggers that are enabled on that event are fired, with some exceptions:

- If the trigger is actually the target of the triggering event, it is not fired. For example, a trigger for all `DROP` events is not fired when it is dropped itself.
- If a trigger is not fired if it has been modified but not committed within the same transaction as the firing event. For example, recursive DDL within a system trigger might modify a trigger, which prevents the modified trigger from being fired by events within the same transaction.

More than one trigger can be created on an object. When an event fires more than one trigger, the order is not defined and you should not rely on the triggers being fired in a particular order.

### Publication Context

When an event is published, certain runtime context and attributes, as specified in the parameter list, are passed to the callout procedure. A set of functions called event attribute functions are provided.

**See Also:** For event-specific attributes, see "[Event Attribute Functions](#)" on page 16-2.

For each system event supported, event-specific attributes are identified and predefined for the event. You can choose the parameter list to be any of these attributes, along with other simple expressions. For callouts, these are passed as `IN` arguments.

### Error Handling

Return status from publication callout functions for all events are ignored. For example, with `SHUTDOWN` events, the server cannot do anything with the return status.

**See Also:** For details on return status, see "[List of Database Events](#)" on page 16-8.

### Execution Model

Traditionally, triggers execute as the definer of the trigger. The trigger action of an event is executed 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 procedure, this behavior is consistent.